

The tiko Programming Language

Code you can read.
Speed you can measure.
Binaries you can ship.

Text licensed CC BY-NC-SA 4.0. Example code licensed MIT.

Contents

Preface 1
Who this book is for 1
How this book is organized 1
Conventions 1
The version this book describes 1
The reference: the tiko wiki 1
The example programs 1
Getting Started 2
Installing tiko 2
The one thing you must install: clang 3
Hello, tiko 4
Choosing a target 5
Reading a diagnostic 6
Turn the checks on now 7
An editor 8
Looking things up 8
Summary 8
Exercises 9
Editors: tkEdit and the Language Server 10
What the editor knows, and what only the compiler knows 10
tkEdit 11
The language server 15
A clean editor is not a clean build 18
What it costs 19
Summary 19
Exercises 20
A Tour of tiko 21
Reaching the standard library 21
Splitting the text into words 22
Counting 23
Ranking, with a lambda 24
Reporting 25
The whole program 25
What you have seen 26
Summary 27
Exercises 27

Types, Variables and Constants 28
Writing the program 28
Writing values 28
Numbers that state their width 29
Naming values 31
One type becoming another 32
Asking about a type 34
Summary 34
Exercises 34
Operators and Expressions 35
Arithmetic 35
Precedence, and its four surprises 36
Two families of logical operator 37
Working with bits 39
Joining, choosing, and shorthand 40
Summary 41
Exercises 41
Control Flow, Scope and defer 42
Choosing 42
Repeating 43
Blocks that exist to bound a lifetime 45
defer 45
Labels and goto 48
Summary 48
Exercises 49
Procedures 50
Subs, functions, and three ways to answer 50
declare, for a name the compiler has not read yet 52
How arguments get in 53
Arguments you can leave out 55
One name, several signatures 56
Recursion, and locals that remember 58
Returning a place instead of a value 59
Any number of arguments 60
Procedures as values 65
What it costs 66
Summary 66
Exercises 67
Arrays 68

Bounds, not length 68
Arrays that change size 69
An array indexed by what it counts 71
More than one subscript 72
Filling an array with braces 73
Passing an array to a procedure 75
Returning an array 76
Walking it 79
When the index is wrong 81
Text stored in an array 83
The descriptor 83
When to reach for something else 84
What it costs 85
Summary 85
Exercises 86
Strings and Text 87
Five ways to hold text 87
Text that spans lines 88
The operations that come with the language 91
Numbers and text 93
What `len` is actually counting 94
Bytes that are not UTF-8 96
The standard library 97
Part of a string, without a copy 99
Laying out a line 105
Reading a table 109
A string as bytes 111
Bytes as text: base64, hex and URLs 112
Working with letters 114
A value with no printable form 115
What it costs 115
Summary 116
Exercises 117
User-Defined Types and Layout 119
Grouping values 119
A type is a value 121
Passing and returning 122
Where the bytes go 123
Several views of the same bytes 126

Fields narrower than a byte 127
Naming a type once 128
Types that refer to each other 129
Arrays of records 131
What it costs 132
Summary 133
Exercises 133
Pointers, References and Memory 135
An address, and what lives at it 135
Arithmetic counts elements 137
Changing what a pointer points at 138
Asking for memory 140
Typed allocation 143
Structures that link 145
References 146
The memory behind a string 147
Reading and writing raw memory 148
When the pointer is null 149
What it costs 150
Summary 150
Exercises 151
Objects 152
A type with its own procedures 152
Constructors 154
Copying an object that owns something 156
Properties 158
Keeping fields private 162
Members that belong to the type 163
Objects on the heap 165
A builder that needs none of the three 167
What it costs 169
Summary 169
Exercises 170
Operator Overloading 171
A value type that reads like a number 171
Member or global 173
Conversions in both directions 174
Reaching into an object 177
A type that counts 179

Taking over allocation 181
What not to overload 183
What it costs 183
Summary 183
Exercises 184
Inheritance and Polymorphism 185
Part 1: The classic model 185
Part 2: Interfaces 203
What it costs 216
Summary 217
Exercises 218
Modules and Packages 219
What `#include` cannot do 219
A module is a directory 220
Who can see what 221
A module with several files 222
Importing 224
Packages 226
Settings that cross the boundary, and settings that do not 229
Initialisation 230
Macros and generics across modules 232
Building 235
The standard library as a package 237
Converting a header library into a module 237
What it costs 239
Summary 240
Exercises 240
Namespaces, Headers and the Preprocessor 242
A program in several files 242
Linkage 245
Startup and shutdown 249
Namespaces 250
Macros 253
Conditional compilation 257
Compile-time checks 258
Source information and pragmas 259
What it costs 260
Summary 261
Exercises 262

Generics 263
A type with a parameter 263
A procedure with a parameter 264
Members, and bodies written out of line 264
Inheritance and virtual methods 266
A type parameter promises nothing 267
Making your own type fit a generic's demands 268
Filling a container from a list 270
Overloading, and when a second generic is unreachable 271
Instantiation has a depth limit 272
What it costs 272
Summary 272
Exercises 273
Errors: Optional, Result and Runtime Checks 274
A value that means "nothing" 274
Optional 275
Result 276
Statements that return an error code 281
Runtime errors 282
Runtime checking 285
Assertions 286
What it costs 287
Summary 287
Exercises 288
Testing, Debugging and Tooling 289
Unit tests with fbcunit 289
Tracing 294
Debugging with -g 296
Warnings and the diagnostics workflow 298
Looking inside a build 300
What it costs 302
Summary 302
Exercises 303
Tuples, Iterators and For Each 304
Returning more than one value 304
Tuple types 305
The shapes of for each 308
The iterator protocol 312
Writing an iterable generic collection 315

Laziness 317
What it costs 319
Summary 320
Exercises 320
Lambdas and Closures 321
Lambdas in expressions 321
A lambda that captures nothing is a procedure pointer 322
Capturing variables 323
Closures carry state 326
Comparers and predicates 327
Lambdas inside generics 328
What it costs 330
Summary 330
Exercises 331
Resource Management 332
The rule 332
defer, or a type with a destructor 333
Writing a handle type 333
Shared ownership: SharedPtr 335
Cycles, and WeakPtr 338
SpCast and the virtual destructor 340
Threads 342
What it costs 342
Summary 343
Exercises 343
Concurrency 344
The thread-safe runtime 344
OS threads 345
Goroutines 347
Channels 350
Waiting on several channels: select channel 353
Coordinating goroutines 354
Shared state 357
Threads or goroutines? 359
What it costs 360
Summary 360
Exercises 360
Containers and Sorting 362
Arrays that grow: Array(of T) 362

Hashed containers 366
Ordered containers 371
LinkedList(of T) 372
Deque(of T) 373
PriorityQueue(of T) 374
BitSet 376
Filling a container from a list 377
Sorting 378
Choosing a container 382
What it costs 383
Summary 383
Exercises 384
Files, Processes and the Environment 385
Files and streams 385
Paths, metadata and directories 395
Configuration files and CSV 406
Coming from the file statements 413
The console as a device 413
Processes and the environment 414
Talking over the network 419
What it costs 425
Summary 426
Exercises 427
Dates and Times 429
The types and the tick 429
Validity 430
Offsets and instants 432
Dates as text 433
Calendar arithmetic 437
A worked example: ages and birthdays 440
Time zones 441
Clocks 442
The legacy bridge 444
A worked example: merging logs 446
What it costs 447
Summary 447
Exercises 448
Numbers 449
The built-in functions 449

Exact integer arithmetic 450
Rounding and selection 453
The special values 454
The floating-point environment 455
Random numbers 457
Identifiers 458
Statistics 459
Numbers as text 462
Money 463
What it costs 466
Summary 467
Exercises 467
Logging, Terminal UIs and SQLite 469
Logging 469
Terminal user interfaces 473
SQLite 478
The finance library 484
What it costs 484
Summary 485
Exercises 485
Interfacing with C 487
Declaring a C function 487
Matching C's types 488
Callbacks 493
Variadic functions 494
Calling conventions and names 495
Linking 497
Reading the C header: #cinclude 500
Loading a library at run time 502
What it costs 504
Summary 504
Exercises 505
Using C libraries 506
Before you start 507
What a library package contains 508
Translating a header 509
Static or dynamic 513
Vendored or installed 516
SQLite 518

raylib 525
GLFW 529
SDL3 532
A binding header 538
When something goes wrong 542
What it costs 544
Summary 544
Exercises 545
Building and Shipping 546
Targets 546
Separate compilation 547
Static libraries 548
Shared libraries 550
Windows programs 551
Build profiles 553
Projects 555
Shipping 558
What it costs 558
Summary 559
Exercises 559
Classic BASIC and Compatibility 561
One dialect 561
Inline data: data, read and restore 561
Computed jumps 563
Classic dates and times 563
common and chain 565
What tiko refuses 567
Bringing old code across 569
What it costs 569
Summary 570
Exercises 570
Capstone: a CLI Search Tool 571
The specification 571
The layout 571
The module 572
The program 577
Running it 580
Testing 581
What it costs 584

Summary 585
Exercises 585
Capstone: a Concurrent Pipeline 586
The specification 586
The shape 586
The records 587
Stage 1: walking the tree 588
Stage 2: hashing, N at a time 589
Back-pressure 590
Stopping early 591
Stage 3: collecting 593
Running it 598
What it costs 599
Summary 599
Exercises 599
Capstone: a TUI Ledger 601
The specification 601
The layout 601
The model 602
Storage 604
Importing a bank's CSV 610
The report 612
The screens 614
The program 624
Running it 627
Shipping 628
What it costs 628
Summary 628
Exercises 629
Capstone: a Notepad for Windows 630
The specification 631
The layout 632
The document 632
The window 639
The program 654
Testing a window with nobody at it 655
Running it 660
Shipping 660
What it costs 660

Summary 661
Exercises 661
Capstone: a Cross-Platform Notepad 663
The specification 664
The layout 664
The document 665
What a window is made of 665
A dialog that does not stop your code 668
The text, and the callback that runs afterwards 670
Two ways of counting 671
Layout, menus, keys 675
Files, and Find 678
A Quit that is not a close 681
Creating it all 682
The program 683
Testing on three systems 685
Running it 691
Shipping 692
What changed from Chapter 36 692
What it costs 693
Summary 694
Exercises 694
Appendix A: Keywords and Operator Precedence 696
Reserved words (247) 696
Runtime names (104) 698
Contextual words 699
Fixed-width type names 700
The predeclared constant 700
Operator precedence 701
Appendix B: Diagnostics 702
Source and file handling (TKO0001–TKO0099) 702
Driver and command line (TKO0100–TKO0199) 702
Lexer (TKO0200–TKO0299) 704
Preprocessor (TKO0300–TKO0399) 704
Parser (TKO0400–TKO0599) 706
Semantic analysis (TKO0600–TKO0899) 708
Notes (TKO0900–TKO0989) 710
Compiler limits and internal errors (TKO0990–TKO0999) 712
Interfaces (TKO1000–TKO1049) 712

String slices (TKO1050–TKO1099) 714
Appendix C: Command-line reference 715
Subcommands 716
General 716
Input 717
Output 717
Target 719
Diagnostics 719
C header import (`#cinclude` and `-cimport`) 719
Developer and compiler introspection 720
Appendix D: Coming from FreeBASIC 722
Dialects and the features they carry 722
threadcall 723
assert 723
The intrinsic defines 723
Command-line options 724
Diagnostics 725
Other differences 726
What fbc 1.20 does not have 727
Appendix F: Compiler Internals 728
The pipeline 728
The runtime: tiko-rt 732
The graphics library: libtkogfx 732
Self-hosting 732
How it is tested 733
Appendix G: Solutions 734

Preface

Who this book is for

For programmers who already know a language and want tiko. It does not assume you know BASIC, and it does not assume you have met FreeBASIC. Where tiko differs from FreeBASIC in a way that would surprise someone who knows it, a sidebar says so.

How this book is organized

Conventions

The version this book describes

Every program in this book was compiled and run, on Windows and on Linux, with one pinned version of the compiler. The pin, and the exact toolchains, are recorded in the book's repository.

The reference: the tiko wiki

This book teaches the language; it does not list every keyword, option and library function. That is the job of the **tiko wiki**, github.com/PaulSquires/tiko-lang/wiki, the language's reference documentation. Keep it open beside the book. The pages you will use most:

- **Feature Index** — every language feature and library, one line each, with a link to its reference page.
- **Function Finder** — every function, grouped by what you want to do, with one line on what each one does.
- **Keyword Index** — every keyword, statement and operator, A to Z, each with its full definition.
- **Compatibility** — for code brought from FreeBASIC: what differs, and what is the same.

The wiki also carries tutorials (concurrency, packages and modules, translating C headers) and a page for each part of the toolchain. Where a chapter shows one form of a statement or a few members of a library, the wiki has the rest.

The example programs

Getting Started

tiko is a compiled, statically typed language for applications and systems programming. It produces native executables with no garbage collector, no virtual machine and no runtime for your users to install: a finished program is one file you can copy to a machine and run.

Under the hood the compiler translates your program to LLVM IR and calls clang to assemble and link it. That matters for exactly one reason, which this chapter gets out of the way first: clang has to be on the machine, and tiko is particular about where it looks.

This chapter installs the compiler, builds three small programs, reads a diagnostic, and turns on the runtime checks you should leave on while you work through the book.

Installing tiko

A tiko installation is a directory. Unpack it wherever you like; nothing is written to the registry, no environment variable has to be set, and nothing else on the machine is touched. Moving the directory later is fine — the compiler finds its own headers and libraries relative to the executable.

> **ON WINDOWS**
> The compiler is `tiko.exe` at the root. Beside it, `inc\` holds the headers and `lib\windows-x86_64` the libraries.

> **ON LINUX**
> The compiler is `bin/tiko`, with headers under `inc/` and libraries in `lib/linux-x86_64`.

> **ON MACOS**
> The compiler is `bin/tiko`, with headers under `inc/` and libraries in `lib/macos-arm64`. It runs on Apple Silicon Macs with macOS 13 or newer.
>
> A browser marks what it downloads, and macOS then refuses the unsigned compiler: it hangs instead of running. Clear the mark once, right after unpacking:
>
> ```
> xattr -dr com.apple.quarantine tiko-macos
> ```

Check that it runs:

```
tiko --version
```

It prints the compiler's version and the target it builds for by default. Keep that version handy: this book was written against one specific release, and the exact one is named in the preface.

The one thing you must install: clang

tiko contains no assembler and no linker. It emits LLVM IR, then runs `clang` to turn that into an object file and `lld` to link it. **You need LLVM 18 or newer installed**, and it is not bundled.

On Windows, install the LLVM package; on Linux, your distribution's `clang` package; on macOS, Homebrew's (`brew install llvm`), together with Xcode's Command Line Tools (`xcode-select --install`), which hold the SDK and the linker every macOS link uses. Then check:

```
clang --version
```

If clang is installed in the usual place, there is nothing to configure. tiko looks for it in four places, and takes the first that exists:

	Where	Why it comes first
1	The path you pass on the command line	Someone who names a compiler has a reason, and it is never second-guessed
2	The `TIKO_CLANG` environment variable	Settles it for a whole shell or a CI job
3	The conventional install: `C:\Program Files\LLVM\bin\clang.exe`, `/usr/bin/clang`, or Homebrew's `/opt/homebrew/opt/llvm/bin/clang` on macOS	See the pitfall below
4	`PATH`	Catches a portable LLVM, a scoop install, or anything under `/usr/local`

So a clang on `PATH` is found — but only when there is no clang at the conventional location.

PITFALL

The conventional install outranks `PATH` deliberately, and it is worth knowing which clang you are actually getting. On Windows the `clang` on `PATH` is as often a stale copy belonging to some other toolchain as it is the one you meant, and a backend that quietly used a different clang from the one its data layout was measured against produces the kind of failure that takes days to find.

If you have several, name the one you want rather than reordering `PATH`:

```
export TIKO_CLANG=/home/you/.local/bin/clang
```

`llvm-ar`, `llvm-rc` and libclang are then taken from that same directory, never searched for separately — so one setting settles all of them. When no clang is found at all you get diagnostic `TKO0111`, "no usable toolchain", and the process exits with status 3.

Hello, tiko

Source files are `.tiko`, and the headers you write yourself are `.ti`. Here is the whole program:

examples/ch01/hello.tiko

```
'' hello.tiko -- the smallest complete tiko program.
print "hello, tiko"
```

Output

examples/ch01/hello.out

```
hello, tiko
```

Compile and run it:

```
tiko hello.tiko -o hello
./hello
```

`-o` names the output. Without it, the executable takes the source file's name. Two other options are worth knowing on day one: `-c` compiles to an object file without linking, and `-v` echoes every clang and linker command tiko runs, which is the quickest way to see what a build actually did.

When a build finishes, the exit status tells you what happened, which matters as soon as you put tiko in a script:

Status	Meaning
0	success
1	errors in your program
2	a bad command line
3	no usable toolchain — usually clang
4	an internal compiler error
5	clang or the linker failed

Status 4 is a bug in the compiler. It is worth reporting, with the source that caused it.

Reading the command line

Programs get their arguments through `command`, which returns one argument by position, counting from 1:

examples/ch01/greet.tiko

```
'' greet.tko -- a program that reads its command line.
dim as ustring who = command( 1 )
if len( who ) = 0 then who = "world"
print "hello, " & who
```

```
./greet tiko
hello, tiko
```

Two things in those four lines are worth naming now, because every later chapter assumes them.

`ustring` is the string type. It holds UTF-16 text, its length is O(1), and it behaves the same on every platform. Chapter 9 covers text properly, including the older `string` type that holds bytes and the two cases where you still want it.

Types say their width. `int32`, `int64`, `float64` and their siblings mean what they say on every target — no guessing whether `int` is 32 or 64 bits today. Chapter 4 lists all ten, plus `integer`, the pointer-sized type that deliberately has no fixed-width name.

Choosing a target

tiko builds for two targets, and the default is the machine you are on.

Target	What it is	Default on
`windows-x86_64`	Microsoft's ABI, linked against `ucrtbase.dll`	Windows
`linux-x86_64`	The System V ABI, linked against the system's C library	Linux

Name one explicitly with `-target`:

```
tiko hello.tiko -target windows-x86_64 -o hello
```

Targets are not interchangeable at link time: an object file or static library built for one will not link into a program for the other. Chapter 31 covers what each target links and how to ship the result.

Reading a diagnostic

Every message tiko prints carries a stable identifier: `TKO` followed by four digits. The number never changes meaning and is never reused, so it is worth searching for, and worth quoting in a bug report. Appendix B lists them all.

This program has a type error:

`examples/ch01/oops.tiko`

```
'' oops.tko -- deliberately broken, to show a diagnostic.
dim as int32 count
count = "seven"
```

Compiling it says so:

```
oops.tiko(3,1): error TKO0702: a string and a number do not convert into
each other
     |
   3 | count = "seven"
     | ^
```

The parts are the file, the line and column, the severity, the identifier, the message, and then the offending line with a caret under the position. Five options shape this output:

Option	Effect
`--diagnostics-format gnu`	`file:line:col: error: message [TKO0702]`, the shape most editors and CI tools expect
`--color always` / `never`	Force colour on or off, rather than deciding by whether output is a terminal
`-Wno-TKO0830`	Silence one warning, by identifier
`-Werror`	Treat every warning as an error
`-fmax-errors 1`	Stop after the first error, instead of listing everything

Turn the checks on now

By default, a tiko program does no bounds checking. Reading past the end of an array is undefined: usually it quietly returns nonsense, occasionally it corrupts something, and it always wastes an afternoon.

`-exx` turns on array-bounds and null-pointer checking, and the cost is irrelevant while you are learning. **Use it for everything in this book.**

A program can carry its own build options, which is how the listings in this book stay self-contained — the first source file may hold a `#cmdline` line, and the compiler restarts itself with those options applied:

examples/ch01/bounds.tiko

```
'' bounds.tko -- what a checked build catches.
#cmdline "-exx"

dim as int32 a( 1 to 3 )
dim as int32 i = 5

a( i ) = 42          '' index 5 is outside 1..3
print "not reached"
```

Running it stops at the bad index rather than carrying on with a corrupted array:

examples/ch01/bounds.out

```

Aborting due to runtime error 6 (out of bounds array access) at line 7
of bounds.tiko::(),
'A' accessed with invalid index = 5, must be between 1 and 3
```

The process exits with status 6. Without `-exx` the same program writes 42 somewhere it should not and prints `not reached`, which is the outcome the check exists to prevent.

The check covers every array, including the ones that grow while the program runs, which Chapter 8 introduces. The null-pointer check is partial: it catches `*p`, but not yet `p->field` or `p[i]`, which Chapter 11 covers.

There are three levels, and they stack: `-e` enables error checking, `-ex` adds to it, and `-exx` adds array bounds and null-pointer checks on top. Chapter 18 comes back to them alongside the error-handling model.

An editor

None of this book requires a particular editor; every example is a file and a command line. But two tools come with tiko, and both are worth the five minutes:

- **tkEdit** is the editor in the `tiko-lang` folder. It builds and runs your program with one key, lists the compiler's errors and takes you to each one, and shows a procedure's parameters as you type its call.
- **tiko-lsp** is tiko's language server. It gives the editor you already use — VS Code, Zed, Neovim, Sublime Text and others — errors as you type, completion, go to definition, rename and formatting.

Chapter 2 sets up both.

Looking things up

The reference for everything this book does not spell out — every keyword's full syntax, every compiler option, every function in the standard library — is the **tiko wiki**, at github.com/PaulSquires/tiko-lang/wiki. Two pages answer most questions: the Feature Index, one line per language feature and library, and the Function Finder, every function grouped by the job it does. The wiki's Quick Start covers building a toolchain from source, which this chapter does not.

Summary

- A tiko installation is a directory you unpack anywhere; the compiler locates its own headers and libraries.
- **LLVM 18+ must be installed.** tiko finds clang in four places, in order: an explicit path, `TIKO_CLANG`, the conventional install location, then `PATH`.
- Source files are `.tiko`; headers you write are `.ti`.
- `tiko hello.tiko -o hello` compiles and links in one step; `-c` stops at the object file and `-v` shows every command.

- Two targets: `windows-x86_64` (the Windows default, and no Visual Studio needed) and `linux-x86_64` (the Linux default). Objects from different targets do not link together.
- Diagnostics carry stable `TKOnnnn` identifiers; `--diagnostics-format gnu` suits tools, and `-Werror` suits discipline.
- Build with `-exx` while you learn, and let a listing carry its own options with `#cmdline`.
- `ustring` is the string type, and numeric types say their width: `int32`, `int64`, `float64`.
- The reference for every keyword, option and function is the tiko wiki; start from its Feature Index or Function Finder.

Exercises

1. Write a program that reports which compiler built it and which version that compiler is. `__TIKO_COMPILER__` is defined by tiko and by nothing else, and `__TIKO_VERSION__` holds the version as a string. Test the first with `#ifdef`. {solution="ex01-01″}
2. Make a program greet you by name and then exit with status 3, and check the status from your shell (`echo $?`, or `echo %ERRORLEVEL%` in `cmd`). {solution="ex01-02″}
3. Call a procedure that was never declared, and note the identifier of the diagnostic. Then look it up in Appendix B. {solution="ex01-03″}
4. Compile `bounds.tiko` again with the `#cmdline` line deleted, and run it. What does it print, and what is its exit status? That is the difference `-exx` makes.

Editors: tkEdit and the Language Server

The compiler tells you about a mistake when you build. That is a round trip: save, switch to a terminal, run `tiko`, read a line number, go back and find the line. For a missing `end if` the trip costs more than the fix.

An editor that understands tiko shortens it. It underlines the missing `end if` while you are still on the line, shows a procedure's parameters while you type its call, takes you to a declaration with one key, and builds and runs the program with another.

tiko comes with two tools for this. **tkEdit** is an editor of its own, in the same folder as the compiler. **tiko-lsp**, the tiko *language server*, gives the editor you already use — VS Code, Zed, Neovim, Sublime Text and others — the same understanding. This chapter sets up both. Neither is required by anything else in this book: every example is a file and a command line.

What the editor knows, and what only the compiler knows

Both tools read your code as you type, and both do it **without compiling it**. They find the declarations — which names a file declares and where, what type a variable was declared with, which members that type has, what an `#include` or an `#import` brings in — in well under the time between two keystrokes: the language server reads a 900-line file in about a millisecond. That is enough for completion, codetips and going to a declaration. The language server also checks the code's *shape* — a `function` with no `end function`, a string with no closing quote — and underlines a mistake in it as you type.

Neither knows what only compiling tells you: whether a name used in an expression was ever declared, whether a string is being assigned to a number, whether a call has the right number of arguments. Those errors come from the compiler, when you build. Both tools then show the compiler's errors in the source and in a list, beside their own, so the difference is easy to forget. The Pitfall at the end of the chapter shows what that costs.

This program is the one the chapter edits:

examples/ch02/rect.tiko

```
'' The program Chapter 2 edits: a type, a procedure that
'' uses it, and two built-ins.

type Rect
    w as int32
```

```
    h as int32
end type

function Area( byref r as Rect ) as int32
    return r.w * r.h
end function

dim as Rect box
box.w = 3
box.h = 4

dim as ustring label = "box"
print ucase( left( label, 1 ) ) & ": " & Area( box )
```

examples/ch02/rect.out

```
B: 12
```

tkEdit

tkEdit is a small, keyboard-driven editor for tiko, on Windows, Linux and macOS. It comes with the toolchain, so there is nothing else to install:

System	tkEdit
Windows	`tiko-lang\tkedit.exe`
Linux	`tiko-lang/tkedit`
macOS	`tiko-lang/tkEdit.app`

Start it with the files to edit, or with none to restore the last session:

```
tkedit rect.tiko
```

The editor fills the window, with a tab for each open file above it and no toolbar. The menu bar appears when you press `Ctrl+Shift+M` and goes again when you pick a command. On a Mac, every `Ctrl` chord in this section is `Cmd`.

tkEdit finds the compiler beside itself, in the same `tiko-lang` folder, or on your `PATH`; if it cannot, it asks for it once, when it starts.

Building and running

Key	What it does
`F5`	Build, then run
`Ctrl+F5`	Build only
`Shift+F5`	Run the last build
`Ctrl+Alt+F5`	Rebuild everything
`Ctrl+Shift+F5`	Set the arguments the program is run with
`F7`	Choose a build configuration: *Debug* (`-g -exx`, the default) or *Release* (`-O2`)
`Alt+PgDn` / `Alt+PgUp`	Go to the next or previous error

What `F5` builds depends on what you have open. With a project — a folder with a `tiko.toml`, opened by opening that file — it builds the project's target, as `tiko build` would. Without one, it builds the file you are editing. Modified source files are saved first.

When the build fails, the compiler's errors open in the output pane at the bottom of the window, each with its `TKOnnnn` identifier, and are underlined in the source; `Alt+PgDn` takes you from one to the next. A console program runs in a terminal of its own, which waits for a key when the program ends, so its output stays on the screen.

Debugging

tkEdit can stop your program on a line and show you its variables. It does this through **lldb-dap**, the LLVM debugger that editors talk to, reading the DWARF debug information that `-g` writes — the same information Chapter 19 reads with lldb from a terminal.

Key	What it does
`F9`	Set or remove a breakpoint on the cursor's line
`F6`	Build for debugging and start; in a session, continue to the next stop
`F10`	Step over: the next line, running any call on this one to its end
`F11`	Step into: the next line, inside a procedure this one calls
`Shift+F11`	Step out: run until the current procedure returns
`Ctrl+F10`	Run to the cursor's line
`Ctrl+F6`	Break: stop the running program where it is
`Shift+F6`	Stop debugging; the program ends
`Ctrl+Shift+F9`	Remove every breakpoint

The same commands are on the **Debug** menu, after **Build**.

A breakpoint is a red circle in the margin. `F9` sets it; a click in the margin sets a bookmark instead, unless the setting `margin_click = "breakpoint"` is in the `[editor]` table of `settings.toml`. Breakpoints are kept with the session, so they are still there tomorrow.

`F6` builds with your configuration's options, without its `-O` and with `-g -exx` added, because an optimised program has lost the variables you want to look at. Then it runs the program in its terminal, as `F5` does, and stops on the first breakpoint it reaches — or, with no breakpoint set, on the program's first statement. A yellow arrow in the margin marks the line about to run, and the output pane opens on its **Debug** tab:

- The buttons along the top are the commands in the table: Continue, Break, Stop, Step, Step Over, Step Out, Run to Cursor. The status line at the bottom of the window says why the program stopped and where — `Stopped - breakpoint rect.tiko:10  in Area`.
- **Locals**, **Globals** and **Watch** share the left side, with their tabs along its bottom. Each variable has a row: its name, its value and its type. A record opens to its fields and an array to its elements, named the way the program names them — `rs(1)`, `rs(2)`, `rs(3)` for an array declared `rs( 1 to 3 )`. A large array opens in groups of a hundred elements first. A value that changed since the last stop is

shown in red, and what you opened stays open as you step. A click on a column heading sorts by it.
- **Call Stack** and **Breakpoints** share the right side. A click on a frame of the call stack shows that procedure's locals and goes to its line. The Breakpoints list has every breakpoint in every open file, with the text of its line; a double click goes to it. Of the two buttons at the end of its tab row, the first removes the selected breakpoint and the second removes them all.

On the Watch tab, the `+` button adds a row; type a name to watch, written as in the program — `r.w`, `rs(2).sensor`, in any case — and press `Enter`. Its value is read again at every stop. A double click on the row edits it, and emptying it removes it. A watch is a variable, a field or an element, not a calculation: `r.w * 2` shows `<not in scope>`, as does a name the stopped procedure cannot see.

While the program is stopped, rest the mouse on a name in the source and its value appears beside it.

> **BEFORE THE FIRST SESSION**
> tkEdit looks for lldb-dap on your `PATH`, then where LLVM installs it. If it cannot find one, it says so in the status line; set its path as `adapter` in the `[debug]` table of `settings.toml`.
>
> - **Windows:** lldb-dap is in LLVM's `bin` folder, beside clang. It needs **Python 3.10** (64-bit) on the `PATH`, because it loads `python310.dll`; tkEdit checks for it before it starts.
> - **Linux:** install your distribution's `lldb` package, which provides `lldb-dap` or a numbered `lldb-dap-NN`; tkEdit takes the newest.
> - **macOS:** lldb-dap comes with Apple's Command Line Tools. macOS lets one program control another only with **Developer Mode** on, or when you allow it in the dialog it shows. Turn it on once with
>
>   ```
>   sudo DevToolsSecurity -enable
>   ```
>
>   Without it, the status line says so, and no session starts.

Finding your way

Key	What it does
`F12`	Go to the declaration of the name under the cursor
`Alt+Left` / `Alt+Right`	Go back, or forward again
`Ctrl+P`	Search every file and declaration of the project by name
`Ctrl+Shift+P`	List the procedures of this file, or of every open file
`Ctrl+Shift+F`	Search the text of every project file
`Ctrl+E`	List the open files; `Ctrl+Tab` switches to the last one
`Ctrl+\`	Split the window; `Ctrl+Tilde` moves between the panes

Codetips and completion

Type `Area(` in `rect.tiko` and tkEdit shows the procedure's parameter list, read from its declaration. Type `left(` and it shows `Left(string, n)`: the built-in procedures have codetips too, square brackets marking the arguments you may leave out. Type `box.` and the members of `Rect` are offered; type three letters of a name and the names that start with them are offered, and after `as`, the types. `Ctrl+Space` completes a word at any time. The setting `codetips` turns all of this off.

Settings

tkEdit is configured with three text files, and saving one applies it:

Key	File
`Ctrl+Comma`	`settings.toml` — fonts, tabs, the build, the language of the menus
`Ctrl+K`	`keybindings.toml` — every key in this section can be moved
`Ctrl+Shift+T`	the theme, light or dark

The language server

tiko-lsp speaks the **Language Server Protocol**, which most editors understand. The editor starts it in the background and asks it questions — what is under the cursor, what could be typed here, where is this declared — and it answers in terms of tiko. One server, so every editor gets the same answers.

Feature	What it does
Errors as you type	Underlines a mistake in the code's shape, such as an `if` with no `end if`
Completion	Variables, procedures and types in scope; fields after `.`; a module's exports after `strings.`
Hover	The declaration of the name under the mouse
Signature help	The parameter list while you type a call — built-ins included — with the current one marked
Go to definition	Across files, into `#include`d headers and `#import`ed modules
Find references, rename	Every use in the project; a rename it cannot do safely is refused, with the reason
Formatting	Re-indents a file; it never touches a file that has errors
Outline, workspace symbols	The procedures and types of a file, or of the whole project, by name

VS Code

1. In VS Code, open the Extensions view (`Ctrl+Shift+X`), search for **tiko** and install **Tiko**. VSCodium, Cursor and the other editors built on VS Code install it the same way, from their own Extensions view.
2. Open the folder your program is in, with *File > Open Folder*. The server reads the whole folder, so opening a single file gives you less.
3. Open `rect.tiko`. Hover `Area`; press `F12` on it; type `box.` on a new line and see `w` and `h` offered.

The extension carries its own copy of the server, for Windows, Linux and macOS on Apple silicon.

Building and running. Press `Ctrl+F5`, or click the ▸ button at the top right of the editor, to run the program: the extension runs `tiko run` in VS Code's terminal, so the program can read input. **Tiko: Build** builds without running; `Ctrl+Shift+B` offers the same build. As in tkEdit, a `tiko.toml` at or above the file is built as a project, and otherwise the file itself is. The compiler's errors go to the **Problems** view (`Ctrl+Shift+M`); click one to go to it.

Finding the toolchain. For the standard library and the system headers — `#import "std/strings"`, `#include "windows.bi"` — the server needs the toolchain. It finds it by itself if `tiko` is on your `PATH`, or if the `tiko-lang` folder is in your home folder

or your Downloads folder. Otherwise VS Code says so, once, with a **Locate Toolchain...** button: choose the `tiko-lang` folder and the extension saves it as the setting `tiko.toolchainPath`. The command **Tiko: Show Language Server Output** names the toolchain in use.

Files. `.tiko`, `.ti` and `.tko` files are tiko. The `.bas`, `.bi` and `.inc` extensions of older tiko code are also used by other languages, so the extension leaves them alone until you claim them, in the folder's `.vscode/settings.json`:

```
"files.associations": { "*.bas": "tiko", "*.bi": "tiko", "*.inc":
"tiko" }
```

Other editors

Zed, Neovim, Sublime Text, Helix and any other editor that speaks the protocol run the same server, and every tiko release carries it beside the compiler:

System	The server
Windows	`tiko-lang\tiko-lsp.exe`
Linux, macOS	`tiko-lang/bin/tiko-lsp`

So if `tiko` is on your `PATH`, `tiko-lsp` is too. The tiko packages for Zed, Neovim and Sublime Text find it there, or in the `tiko-lang` folder you name in their settings; the server then takes the headers and packages of that same toolchain. Their instructions are in the `editors` folder of the tiko-lsp repository. Any other editor needs three things: the command `tiko-lsp --stdio`, the extensions `.tiko`, `.ti` and `.tko` mapped to a language called `tiko`, and — if `tiko` is not on your `PATH` — the toolchain folder as the option `toolchainPath`. For Helix, in `languages.toml`:

```
[language-server.tiko-lsp]
command = "tiko-lsp"
args = ["--stdio"]
config = { toolchainPath = "/home/you/tiko-lang" }

[[language]]
name = "tiko"
scope = "source.tiko"
file-types = ["tiko", "ti", "tko"]
roots = ["tiko.toml", ".git"]
comment-token = "'"
language-servers = ["tiko-lsp"]
```

Updating

The server in your other editors is the one in the toolchain, so **installing a newer tiko updates it**; restart the server, or the editor, afterwards. VS Code updates its extension, and the server inside it, by itself.

When something is wrong

- **No colour, no completion.** The editor has not recognised the file as tiko. In VS Code, the language mode at the bottom right should say *Tiko*. If VS Code shows *Restricted Mode*, the folder is not trusted, and the extension does not start a program in an untrusted folder.
- **`std/` and `windows.bi` names do not complete.** The server has not found the toolchain. Name the `tiko-lang` folder (in VS Code, with **Tiko: Locate Toolchain…**) and restart the server.
- **An underline on code that compiles.** That is a defect in the server. Report it at the tiko-lsp repository with the smallest program that shows it.
- **You want to see what the server is doing.** Start it with `--log-level debug --log-file` and a path; it writes nothing else to disk. `tiko-lsp --version` names the version you have.

A clean editor is not a clean build

PITFALL

This file is complete. Every block is closed, every statement ends where it should, and neither tkEdit nor the language server underlines anything in it:

`examples/ch02/typo.tiko`

```
'' Every block is closed and every statement is complete,
'' so the editor underlines nothing. The compiler still
'' refuses it: `totl` was never declared.

dim as int32 total = 10
print totl
```

It does not compile:

```
typo.tiko(6,7): error TKO0743: 'totl' is not declared
     |
   6 | print totl
     |       ^
```

The editor reads the code's shape; whether `totl` names anything is a question for the compiler. A file with no underline has not been checked the way a build

checks it. Build before you believe it — `F5` in tkEdit, `Ctrl+F5` in VS Code — and read the errors that come back.

What it costs

- **The first look at a large header takes seconds.** `windows.bi` and what it includes declare about 190,000 names. The server reads them once, in the background, when a project uses them; after that, completion costs tens of milliseconds a keystroke instead of a few.
- **The reader does not run the preprocessor.** Both branches of an `#ifdef` are read, so completion offers `MessageBoxA` and `MessageBoxW` alike, and a name made by `#define` is not offered at all.
- **Rename works by name.** It follows what each file can see, and refuses where a local variable shares the name it would change, but two unrelated declarations with one name in files that see each other are one name to it.
- **Codetips for the built-ins are a fixed list,** shipped with the tools; a procedure you declare is described by its own declaration instead, and wins over a built-in of the same name.

Summary

- tkEdit and tiko-lsp read your code as you type, without compiling it. They catch mistakes in its shape; the compiler catches the rest, when you build.
- tkEdit comes with tiko (`tiko-lang/tkedit`). `F5` builds and runs — a `tiko.toml` project, or else the current file — and `Alt+PgDn` walks the compiler's errors.
- `F12` goes to a declaration, `Ctrl+P` finds any name, codetips show parameters as you type, and three TOML files hold every setting.
- tiko-lsp gives any editor that speaks the Language Server Protocol completion, hover, signature help, go to definition, references, rename, formatting and errors as you type.
- In VS Code, install the **Tiko** extension: it carries the server, runs your program with `Ctrl+F5`, and puts the compiler's errors in **Problems**.
- Every tiko release carries `tiko-lsp` beside the compiler, for every other editor; a newer tiko updates it.
- `F9` sets a breakpoint and `F6` runs to it; `F10`, `F11` and `Shift+F11` step. The Debug tab shows locals, globals and watches beside the call stack and the breakpoints. It needs lldb-dap, and on macOS Developer Mode.
- **No underline is not a successful build.** Build, and read what the compiler says.

Exercises

1. Open `rect.tiko` in tkEdit and run it with `F5`. Delete the line `end function`, build again, and use `Alt+PgDn` to reach the error. Which line does it point at, and why that one?
2. In tkEdit, put the cursor on `Area` in the last line, go to its declaration with `F12`, and come back with `Alt+Left`. Then, in an editor with the language server, rename `Rect` to `Box2` (in VS Code, `F2`), and check that the declaration and the `dim` changed together.
3. In an editor with the language server, start typing `print mid(` and watch the parameter list. Type an argument and a comma at a time: which parameter is marked after each comma, and what happens after the third?
4. Copy `typo.tiko` into your editor. Find out, without building, which tool would tell you about `totl`, then build and compare the identifier you get with the one in this chapter.
5. In tkEdit, set a breakpoint on line 10 of `rect.tiko` (`return r.w * r.h`) and press `F6`. Open `r` on the Locals tab. Then select the second row of the Call Stack: what does Locals show now, and where is the editor's cursor? Step out with `Shift+F11` and watch the arrow.

A Tour of tiko

The last chapter got a compiler working. This one writes something with it — a program that reads a file, counts the words in it, and reports the three most common. It is small enough to hold in your head and big enough to need most of what the language is made of: the standard library, strings, a hash map, iteration, sorting, a lambda, and errors that are values rather than surprises.

Nothing here is explained completely. The point is to see the shape of a tiko program before taking it apart, and every idea below is picked up properly in a later chapter. Read it the way you would read someone else's code: get the sense of it, and let the details wait.

The file we are counting is three lines:

examples/ch03/words.txt

```
the quick brown fox
jumps over the lazy dog
the dog sleeps
```

Reaching the standard library

tiko's library comes in modules, and a program says which ones it wants:

examples/ch03/step1.tiko

```
'' step1.tiko -- read the file, and say so if it is not there.
#import "std/fs"

var loaded = fs.ReadAllText( "words.txt" )
if loaded.IsOk( ) = false then
    print "cannot read words.txt"
    end 1
end if

var text = loaded.Value( )
print "characters: " & len( text )
```

examples/ch03/step1.out

```
characters: 59
```

`#import "std/fs"` makes the filesystem module available under the name `fs`, and every name it provides is reached through that prefix — `fs.ReadAllText`. That prefix is not decoration; it is how a reader of your code knows where a function came from.

The interesting part is what `ReadAllText` gives back. Not a string: a **`Result`**, which is either the text or the reason there is no text. There is no exception to catch and no error code to remember to check, because the value itself will not hand over its contents until you have asked whether it has any:

- `loaded.IsOk( )` — did it work?
- `loaded.Value( )` — the text, once you know it worked.

A program that forgets to check does not silently read nonsense; it fails to compile in the shape you would have wanted. Chapter 18 is about designing with `Result`.

> **NOTE**
> `var` declares a variable and takes its type from the value, the way `dim as ustring text = …` would if you wrote the type out. Use `var` when the type is obvious from the line, and the full form when it is not.

Splitting the text into words

examples/ch03/step2.tiko

```
'' step2.tiko -- split the text into words.
#import "std/fs"
#import "std/strings"

var loaded = fs.ReadAllText( "words.txt" )
if loaded.IsOk( ) = false then
    print "cannot read words.txt"
    end 1
end if

var flat = strings.Replace( loaded.Value( ), !"\n", " " )
var words = strings.Split( flat, " " )
print "words: " & words.Count

for each w in words
    if len( w ) > 0 then print "  [" & w & "]"
next
```

examples/ch03/step2.out

```
words: 13
  [the]
  [quick]
  [brown]
  [fox]
  [jumps]
```

```
  [over]
  [the]
  [lazy]
  [dog]
  [the]
  [dog]
  [sleeps]
```

Two library calls do the work. `strings.Replace` flattens the newlines into spaces, and `strings.Split` cuts the result on spaces and hands back an array. `words.Count` is how many it found.

`for each w in words` walks the array without an index, without a bound, and without the off-by-one that an index loop invites. The loop variable's type comes from the array, so `w` is a `ustring` — tiko's string type, which holds text rather than bytes and is the same on every platform.

The `if len( w ) > 0` is there because splitting on a single space leaves empty pieces wherever two spaces met. Real text is untidy; the program says so rather than pretending.

Counting

examples/ch03/step3.tiko

```
'' step3.tiko -- count each word.
#import "std/containers"
#import "std/fs"
#import "std/strings"

var loaded = fs.ReadAllText( "words.txt" )
if loaded.IsOk( ) = false then
    print "cannot read words.txt"
    end 1
end if

var flat = strings.Replace( loaded.Value( ), !"\n", " " )
var words = strings.Split( flat, " " )

dim counts as containers.Map( of ustring, int32 )
for each w in words
    if len( w ) > 0 then counts[ LCase( w ) ] += 1
next

print "distinct words: " & counts.Count
```

```
print "the: " & counts[ "the" ]
print "dog: " & counts[ "dog" ]
```

examples/ch03/step3.out

```
distinct words: 9
the: 3
dog: 2
```

`containers.Map( of ustring, int32 )` is a hash map from text to a number. The `( of … )` is tiko's generics syntax: this is not a map of "anything", it is a map whose keys are `ustring` and whose values are `int32`, checked as thoroughly as if you had written the type by hand. Chapter 17 is about writing your own.

`counts[ LCase( w ) ] += 1` does the whole count. Indexing a map with a key that is not there inserts it with a zero value, so the `+= 1` works the first time a word appears as well as the fifth. `LCase` folds the case, so *The* and *the* are one word.

Ranking, with a lambda

A map has no order, so the last step copies the counts into an array and sorts it:

examples/ch03/wordcount.tiko

```
dim table as containers.Array( of Tally_ )
for each entry in counts
    dim as Tally_ t = ( entry.Key, entry.Value )
    table.Push( t )
next

containers.SortBy( table, _
    function( byref a as Tally_, byref b as Tally_ ) as boolean
        return a.count > b.count
    end function )
```

`for each entry in counts` walks the map, and each `entry` carries a `Key` and a `Value`.

`containers.SortBy` takes the array and a **lambda** — a function written where it is used, with no name. This one answers a single question: does `a` come before `b`? Sorting by descending count is `a.count > b.count`, and changing the ranking means changing that one line. Chapter 20 is about lambdas, including the ones that capture variables from around them.

The `_` at the end of the `SortBy` line continues the statement onto the next line. tiko has no statement terminator, so a line break ends a statement unless you say otherwise.

Reporting

examples/ch03/wordcount.tiko

```
print "the three most common words"
for i as int32 = 0 to 2
    print using "  \    \ #"; table[ i ].word; table[ i ].count
next
```

`print using` formats output against a picture: `\  \` is a text field of that width, and `#` is a digit. It is the one piece of this program that will look unfamiliar to most readers, and it is worth the surprise — lining up columns with string concatenation is miserable.

The whole program

examples/ch03/wordcount.tiko

```
#import "std/containers"
#import "std/fs"
#import "std/strings"

type Tally_
    as ustring word
    as int32 count
end type

var loaded = fs.ReadAllText( "words.txt" )
if loaded.IsOk( ) = false then
    print "cannot read words.txt"
    end 1
end if

var flat = strings.Replace( loaded.Value( ), !"\n", " " )
var words = strings.Split( flat, " " )

dim counts as containers.Map( of ustring, int32 )
for each w in words
    if len( w ) > 0 then counts[ LCase( w ) ] += 1
next

dim table as containers.Array( of Tally_ )
for each entry in counts
    dim as Tally_ t = ( entry.Key, entry.Value )
    table.Push( t )
next
```

```
containers.SortBy( table, _
    function( byref a as Tally_, byref b as Tally_ ) as boolean
        return a.count > b.count
    end function )

print "the three most common words"
for i as int32 = 0 to 2
    print using "  \    \ #"; table[ i ].word; table[ i ].count
next
```

examples/ch03/wordcount.out

```
the three most common words
  the    3
  dog    2
  quick  1
```

Forty lines, no memory management, no build file, and one command to build it:

```
tiko wordcount.tiko -o wordcount
./wordcount
```

The result is a native executable that depends on nothing you have to ship beside it.

PITFALL

`read` is not available as a variable name — it belongs to the `data`/`read` statement pair that tiko keeps for compatibility. The compiler says `TKO0600: 'read' is already declared in this scope`, which is accurate but startling when you were not thinking about `data` at all.

The same applies to other short, tempting names: `str`, `left`, `mid`, `window`, `log`. When a name is refused for no reason you can see, that is usually why — pick another.

What you have seen

Each of these has a chapter of its own:

In this program	Chapter
`#import` and the `std` modules	14
`Result`, and errors as values	17
`ustring` and the string library	8
`Map`, `Array`, `SortBy`	22
`for each`	18
generics, the `( of … )` syntax	16
lambdas	19
`print using` and console output	23

Summary

- A program reaches the library with `#import "std/…"`, and uses the module's name as a prefix.
- Fallible calls return a `Result`: ask `IsOk( )`, then take `Value( )`.
- `var` infers a variable's type; `dim as T` states it.
- `for each` walks an array without an index; the loop variable's type comes from the elements.
- `Map( of K, V )` is a checked hash map, and indexing a missing key inserts it.
- A lambda is a function written where it is used — `SortBy` takes one to decide the order.
- One command compiles and links a native executable.

Exercises

1. Report the longest word in the file as well as the most common ones. {solution="ex03-01″}
2. List the words that appear exactly once, in alphabetical order. `OrderedMap` keeps its keys sorted, where `Map` does not. {solution="ex03-02″}
3. Make the program take the file name from the command line, falling back to `words.txt` when none is given. Chapter 1 showed `command( 1 )`.
4. The ranking prints three words. Make it print the top three *counts*, so that words tied for third all appear.

Types, Variables and Constants

Every value in a tiko program has a type, fixed when the program is compiled and never changing afterwards. That is what lets the compiler catch a whole class of mistake before you run anything, and it is why a tiko program runs at the speed it does: the machine code knows exactly what it is working with.

This chapter is the vocabulary the rest of the book assumes — how to write values, how to name them, and what happens when one type becomes another.

Writing the program

A few rules apply everywhere, and they are quickly stated.

Case does not matter to the compiler. `Count`, `count` and `COUNT` are one name. It matters very much to readers, so the book follows one convention throughout: keywords lowercase, and types and procedures in PascalCase.

Comments start with `''` and run to the end of the line. A block comment is `/'` … `'/`, and those nest.

One statement per line, with no terminator to forget. A line can be continued with a trailing `_`, and two short statements can share a line with `:` between them — the book uses that only for setup lines that read better together.

Writing values

examples/ch04/literals.tiko

```
'' literals.tiko -- how values are written.
print "hex   &hFF      = " & &hFF
print "octal &o755     = " & &o755
print "binary &b1011   = " & &b1011
print
print "a suffix types the literal: " & sizeof( 1LL ) & " bytes for 1LL"
print
'' A plain string is literal; !"..." reads escapes; $"..." takes none.
print !"escaped:\tone tab, then a newline\n"
print $"raw: \t stays two characters"
print
print "true and false are values: " & true & " " & false
```

examples/ch04/literals.out

```
hex   &hFF      = 255
octal &o755     = 493
binary &b1011   = 11
```

```
a suffix types the literal: 8 bytes for 1LL

escaped:  one tab, then a newline

raw: \t stays two characters

true and false are values: true false
```

Numbers can be written in hexadecimal, octal or binary with `&h`, `&o` and `&b`. A suffix pins a literal's type where it matters — `1LL` is 64-bit.

Strings come in three spellings, and the difference is what a backslash means:

Written	`\t` is
`"plain"`	two characters, a backslash and a `t`
`!"escaped"`	a tab
`$"raw"`	two characters, explicitly

`true` and `false` are ordinary values of type `boolean`.

Numbers that state their width

tiko's numeric types say how big they are:

examples/ch04/widths.tiko

```
'' widths.tiko -- each type states its size in bytes.
print "int8    " & sizeof( int8 )   & "   uint8   " & sizeof( uint8 )
print "int16   " & sizeof( int16 )  & "   uint16  " & sizeof( uint16 )
print "int32   " & sizeof( int32 )  & "   uint32  " & sizeof( uint32 )
print "int64   " & sizeof( int64 )  & "   uint64  " & sizeof( uint64 )
print "float32 " & sizeof( float32 ) & "   float64 " & sizeof( float64 )
print
print "integer " & sizeof( integer ) & "   -- pointer-sized"
print "long    " & sizeof( long ) & "   -- 32 bits, despite the name"
```

examples/ch04/widths.out

```
int8    1   uint8   1
int16   2   uint16  2
int32   4   uint32  4
int64   8   uint64  8
float32 4   float64 8
```

```
integer 8   -- pointer-sized
long    4   -- 32 bits, despite the name
```

`int8`, `int16`, `int32`, `int64` and the `uint` forms cover the integers; `float32` and `float64` cover the rest. **Use these.** A reader never has to ask how wide `int32` is, and the answer is the same on every target.

Two types deliberately have no fixed-width name:

- **`integer`** is pointer-sized — 8 bytes on the 64-bit targets. Reach for it when a value has to match the machine's word, such as an index into something that could be huge.
- **`uinteger`** is its unsigned twin.

You will also meet the older names in existing code: `byte`, `short`, `long`, `longint`, `single`, `double`, and their unsigned forms. They are what the fixed-width names are made of — `int32` *is* `long` — so the two mix freely.

> **PITFALL**
> **`long` is 32 bits.** It is the one name most likely to mislead a C or Java programmer, who will read it as 64. The fixed-width spellings exist partly to make that impossible to get wrong: write `int64` when you mean 64 bits.

The fixed-width names are typedefs the compiler predeclares rather than reserved words, which has a visible consequence:

`examples/ch04/shadow.tiko`

```
'' shadow.tiko -- the fixed-width names are typedefs, not keywords,
'' not keywords, so a program is free to take one for itself.
dim as int32 n = 1

scope
    dim as int32 int32 = 7     '' a variable named after its own type
    print "shadowed: " & int32
end scope

print "outside:  " & n
```

`examples/ch04/shadow.out`

```
shadowed: 7
outside:  1
```

A program can declare a variable called `int32` and it wins inside its scope. That is not an oversight: existing headers already declare names like `INT32`, and making them keywords would have broken every one of them.

Naming values

examples/ch04/declare.tiko

```
'' declare.tiko -- naming values.
dim as int32 counted = 3            '' the type, stated
var inferred = 3                    '' inferred from the value
var text = "hello"                  '' a ustring
dim as float64 ratio = 1.5

print counted & " " & inferred & " " & text & " " & ratio

'' Several names of one type, and one left deliberately uninitialised.
dim as int32 x, y, z
dim as int32 scratch = any          '' left uninitialised
x = 1 : y = 2 : z = 3
print x + y + z

'' A constant cannot be assigned to, and its type comes from the value.
const MaxRetries = 3
const Greeting as ustring = "hi"
print MaxRetries & " " & Greeting

'' An enumeration names related constants.
enum Colour_
    Red
    Green = 10
    Blue
end enum
print Red & " " & Green & " " & Blue
```

examples/ch04/declare.out

```
3 3 hello 1.5
 6
3 hi
0 10 11
```

`dim as T name` declares a variable of a stated type. `var name = value` declares one and takes the type from the value. The book's rule: **`var` when the type is obvious from the line, `dim as` when it is not.**

Other forms in that listing:

- `dim as int32 x, y, z` declares several at once.
- `= any` leaves a variable uninitialised, holding whatever was in that memory. It is a deliberate, visible choice, not the default.
- `const` names a value that cannot be assigned to. Its type comes from the value unless you state one.
- `enum` names a set of related constants; values follow on from the last one given, so `Blue` is

  11.

Every variable must be declared. There are no implicit variables, which means a typo is caught rather than silently becoming a second variable:

examples/ch04/implicit.tiko

```
'' implicit.tiko -- every variable is declared. There is no implicit
'' creation, so a typo is a compile error, not a second variable.
dim as int32 total = 0

totl = total + 1      '' a typo, not a new variable
print totl
```

```
implicit.tiko(5,1): error TKO0743: 'totl' is not declared
     |
   5 | totl = total + 1      '' a typo, not a new variable
     | ^
```

> **NOTE**
> A type-suffix sigil does not give a variable a type: `dim count%` is error `TKO0320`. Next to an explicit type, as in `dim as int32 count%`, the suffix is ignored, with warning `TKO0206`: the type comes from `as int32`. Write `dim as int32 count`. Chapter 32 covers code that still relies on suffixes.

One type becoming another

Assigning a value of one numeric type to another converts it, and conversions between integer types are the obvious ones: the value is kept when it fits and wraps when it does not.

Converting a float to an integer is where tiko surprises people who come from C:

examples/ch04/rounding.tiko

```
'' rounding.tiko -- converting a float to an integer ROUNDS.
dim as float64 half = 2.5
dim as float64 up   = 2.6
```

```
dim as float64 neg  = -2.5

print "cint( 2.5 )  = " & cint( half )    '' rounds to even
print "cint( 3.5 )  = " & cint( 3.5 )
print "cint( 2.6 )  = " & cint( up )
print "cint( -2.5 ) = " & cint( neg )
print
print "int( 2.6 )   = " & int( up )       '' towards minus infinity
print "int( -2.5 )  = " & int( neg )
print "fix( 2.6 )   = " & fix( up )       '' towards zero, like C
print "fix( -2.5 )  = " & fix( neg )
print
dim as int32 assigned = up                 '' an assignment rounds too
print "dim as int32 = 2.6 gives " & assigned
```

examples/ch04/rounding.out

```
cint( 2.5 )  = 2
cint( 3.5 )  = 4
cint( 2.6 )  = 3
cint( -2.5 ) = -2

int( 2.6 )   = 2
int( -2.5 )  = -3
fix( 2.6 )   = 2
fix( -2.5 )  = -2

dim as int32 = 2.6 gives 3
```

PITFALL

It rounds. C truncates. `cint( 2.6 )` is 3, not 2 — and a plain assignment into an integer variable rounds in exactly the same way.

And a half goes to the *even* neighbour: `cint( 2.5 )` is 2 while `cint( 3.5 )` is 4. That is C's `lrint( )`, not its `round( )`, which takes halves away from zero and gives 3 and 4 — the answer most people expect.

When you want C's behaviour, say so: **`fix( )` truncates towards zero**. `int( )` is a third answer again — it rounds towards minus infinity, so it and `fix( )` agree on positive numbers and differ on negative ones.

The `c*` family converts explicitly: `cint`, `clng`, `clngint`, `csng`, `cdbl`, `cbool`, `cbyte`, `cshort`, and the unsigned forms. `cast( T, value )` converts to any type you name, and `cunsg` / `csign` reinterpret a value's sign without changing its bits.

Asking about a type

- **`sizeof( T )`** is the size in bytes, as the listing above used.
- **`len( x )`** is the length of a string, or the size of anything else.
- **`typeof( x )`** is the type of an expression, usable where a type is expected.

`fblimits.bi` names the minimum and maximum of each type, if you would rather not write `2147483647` and hope.

Summary

- Types are fixed at compile time; every variable is declared, so a typo is an error.
- Prefer the width-stating names: `int8`…`int64`, `uint8`…`uint64`, `float32`, `float64`.
- `integer` and `uinteger` are pointer-sized and have no fixed-width spelling — that is the reason to use them.
- **`long` is 32 bits.** The older names still appear in existing code, and mix freely with the new ones.
- The fixed-width names are typedefs, so a program may shadow one.
- `var` infers a type from the value; `dim as T` states it; `const` names a value; `enum` names a set.
- **Float to integer rounds, half to even** — `fix( )` truncates like C, `int( )` floors.
- Numbers may be written in hex, octal or binary; `!"…"` reads escapes and `$"…"` takes none.

Exercises

1. Print the largest value each signed integer type can hold, then add one to the `int32` maximum and explain what you see. `std/numeric`'s `AddChecked` gives a second opinion. {solution="ex04-01″}
2. Build a table comparing `cint`, `int` and `fix` over 2.4, 2.5, 2.6 and their negatives. Which pair agrees on positive numbers but not negative ones? {solution="ex04-02″}
3. Declare a variable with `= any`, print it before assigning anything, and run the program twice. Is the value the same both times? Should you ever rely on it?
4. `const Pi = 3.14159` — what type does `Pi` get? Use `sizeof( typeof( Pi ) )` to find out, then state the type explicitly and compare.

Operators and Expressions

Operators are the part of a language people assume they already know, which is exactly why they are worth a chapter. Most of tiko's behave as you expect. A handful do not, and each of those is a bug waiting in code that was written by assumption rather than by reading.

Everything in this chapter was measured against the compiler. Where tiko's grouping differs from the language you came from, the listing prints both readings so you can see which one won.

Arithmetic

examples/ch05/arithmetic.tiko

```
'' arithmetic.tiko -- the operators, and the three that surprise people.
print "7 / 2   = " & ( 7 / 2 )      '' always a float
print "7 \ 2   = " & ( 7 \ 2 )      '' integer division
print "7 mod 2 = " & ( 7 mod 2 )
print
'' mod takes the sign of the LEFT operand: a remainder, not a modulus.
print "-7 mod 2 = " & ( -7 mod 2 )
print " 7 mod -2 = " & ( 7 mod -2 )
print
'' '^' is floating point even when both operands are whole numbers.
print "2 ^ 10 = " & ( 2 ^ 10 ) & _
      "   and it is " & sizeof( typeof( 2 ^ 10 ) ) & " bytes wide"
```

examples/ch05/arithmetic.out

```
7 / 2   = 3.5
7 \ 2   = 3
7 mod 2 = 1

-7 mod 2 = -1
 7 mod -2 = 1

2 ^ 10 = 1024   and it is 8 bytes wide
```

Three of those need naming.

`/` is always floating point. `7 / 2` is 3.5 even though both operands are whole numbers. Integer division is a separate operator, `\`, and it truncates: `7 \ 2` is 3.

`mod` is a remainder, not a modulus. It takes the sign of its left operand, so `-7 mod 2` is −1 rather than 1. When you want the mathematical modulus — always

non-negative for a positive divisor — `std/numeric`'s `FloorMod` is the one to reach for. This matters the moment you use `mod` to wrap an index that might be negative.

`^` produces a float. `2 ^ 10` is 1024, but it is a `float64` that happens to hold a whole number, and it was computed in floating point. For integer powers where exactness matters, `std/numeric` has `IPow`.

Precedence, and its four surprises

tiko's precedence table was established by compiling 484 operator pairs and reading what the compiler did with each. Four of the answers differ from what a C, Python or Java programmer expects, and this listing prints each one beside both possible groupings:

examples/ch05/precedence.tiko

```
'' precedence.tiko -- four places the table is not what people expect.
dim as int32 a = 3, b = 0

'' 1. '^' groups to the LEFT. Most languages group it to the right.
print "2 ^ 3 ^ 2   = " & ( 2 ^ 3 ^ 2 )
print "  (2^3)^2   = " & ( ( 2 ^ 3 ) ^ 2 )
print "  2^(3^2)   = " & ( 2 ^ ( 3 ^ 2 ) )
print

'' 2. 'shl' and 'shr' bind TIGHTER than '+'.
print "1 shl 2 + 3 = " & ( 1 shl 2 + 3 )
print "  (1 shl 2) + 3 = " & ( ( 1 shl 2 ) + 3 )
print "  1 shl (2 + 3) = " & ( 1 shl ( 2 + 3 ) )
print

print "not a = b     = " & ( not a = b )
print "  not (a = b) = " & ( not ( a = b ) )
print "not a = b       = " & ( not a = b )
print "  not (a = b)   = " & ( not ( a = b ) )
print "  (not a) = b   = " & ( ( not a ) = b )
print
'' 4. 'andalso' and 'orelse' share one level: they group left to right.

print "true orelse false andalso false = " & _
      ( true orelse false andalso false )
print "  (true orelse false) andalso false = " & _
      ( ( true orelse false ) andalso false )
```

examples/ch05/precedence.out

```
2 ^ 3 ^ 2   = 64
  (2^3)^2   = 64
  2^(3^2)   = 512

1 shl 2 + 3 = 7
  (1 shl 2) + 3 = 7
  1 shl (2 + 3) = 32

not a = b     = -1
  not (a = b) = -1
not a = b       = -1
  not (a = b)   = -1
  (not a) = b   = 0

true orelse false andalso false = false
  (true orelse false) andalso false = false
```

Expression	tiko groups it	Many languages
`2 ^ 3 ^ 2`	`(2 ^ 3) ^ 2` = 64	right, giving 512
`1 shl 2 + 3`	`(1 shl 2) + 3` = 7	`+` first, giving 32
`not a = b`	`not (a = b)`	`(not a) = b`
`x orelse y andalso z`	left to right	`andalso` first

PITFALL

`not` is looser than the comparisons. `not a = b` asks "is it not the case that a equals b", which is usually what was meant — but it is the opposite of C, where `!a == b` compares `!a` against `b`.

`andalso` and `orelse` share one precedence level, so they group strictly left to right. In C, Python and nearly everything else, `and` binds tighter than `or`, so `x or y and z` means `x or (y and z)`. Here it means `(x orelse y) andalso z`, which is a different answer.

Both are cases where code that looks right and compiles clean can be wrong. Parenthesise any mixed `andalso`/`orelse` expression, every time.

Two families of logical operator

`examples/ch05/logic.tiko`

```
'' logic.tiko -- two families of operator that look alike and are not.
declare function Loud( byval v as boolean ) as boolean
```

```
function Loud( byval v as boolean ) as boolean
    print "   (Loud was called)"
    return v
end function

print "andalso stops once the answer is known:"
if false andalso Loud( true ) then print "unreachable"

print "and always evaluates both:"
if false and Loud( true ) then print "unreachable"

print
'' On integers the bitwise family works bit by bit.
print "12 and 10 = " & ( 12 and 10 )
print "12 or  10 = " & ( 12 or 10 )
print "12 xor 10 = " & ( 12 xor 10 )
print "not 12    = " & ( not 12 )
```

examples/ch05/logic.out

```
andalso stops once the answer is known:
and always evaluates both:
   (Loud was called)

12 and 10 = 8
12 or  10 = 14
12 xor 10 = 6
not 12    = -13
```

`andalso` and `orelse` **short-circuit**: once the answer is known the right-hand side is never evaluated. `and`, `or`, `xor` and `not` do not — they are bitwise operators that also serve for logic, and they always evaluate both sides.

The difference matters whenever the right-hand side has an effect, or would fail:

- `if p <> 0 andalso p->count > 0` is safe.
- `if p <> 0 and p->count > 0` dereferences a null pointer.

Use `andalso` and `orelse` for conditions, and reserve `and`/`or`/`xor`/`not` for bits.

> **NOTE**
> tiko refuses to mix the two families silently. Combining a boolean and a non-boolean in an `andalso` chain draws a warning, because the result is almost never what was meant — hoist the value into a `boolean` first.

Working with bits

examples/ch05/bits.tiko

```
'' bits.tiko -- shifting, masking, and the single-bit helpers.
dim as uint32 flags = 0

flags = bitset( flags, 0 )        '' turn bit 0 on
flags = bitset( flags, 3 )
print "flags        = " & flags & "  (binary " & bin( flags ) & ")"
print "bit 3 set?   = " & cbool( bit( flags, 3 ) <> 0 )
flags = bitreset( flags, 3 )
print "after reset  = " & bin( flags )
print

print "1 shl 8      = " & ( 1 shl 8 )
print "256 shr 4    = " & ( 256 shr 4 )
print

'' Picking halves out of a wider value.
dim as uint32 packed = &hDEADBEEF
print "packed       = " & hex( packed )
print "low word     = " & hex( loword( packed ) )
print "high word    = " & hex( hiword( packed ) )
print "low byte     = " & hex( lobyte( packed ) )
```

examples/ch05/bits.out

```
flags        = 9  (binary 1001)
bit 3 set?   = true
after reset  = 1

1 shl 8      = 256
256 shr 4    = 16

packed       = DEADBEEF
low word     = BEEF
high word    = DEAD
low byte     = EF
```

`shl` and `shr` shift; `bitset`, `bitreset` and `bit` turn one bit on, off, or read it; `lobyte`, `hibyte`, `loword` and `hiword` pull a piece out of a wider value. `bin( )` and `hex( )` print a value in a base where the bits are visible, which is most of the battle when debugging a mask.

Remember that shifts bind tighter than arithmetic. `1 shl n + 1` shifts by `n` and then adds — not what the spacing suggests.

Joining, choosing, and shorthand

examples/ch05/expressions.tiko

```
'' expressions.tiko -- joining text, choosing a value, and shorthand.
dim as ustring who = "world"

'' '&' joins text. '+' also adds numbers, so '&' always means join.
print "hello, " & who
print "1 & 2 gives the text " & ( 1 & 2 )
print "1 + 2 gives the number " & ( 1 + 2 )
print

'' iif picks between two values inside an expression.
dim as int32 stock = 0
print "in stock: " & iif( stock > 0, "yes", "no" )
print

'' Compound assignment, and swapping two variables.
dim as int32 total = 10
total += 5 : total *= 2
print "total = " & total

dim as ustring left_ = "first", right_ = "second"
swap left_, right_
print left_ & " " & right_
```

examples/ch05/expressions.out

```
hello, world
1 & 2 gives the text 12
1 + 2 gives the number 3

in stock: no

total = 30
second first
```

`&` joins text; `+` adds numbers. `+` will also join two strings, which is exactly why the book always uses `&`: `1 & 2` is the text `12`, while `1 + 2` is the number 3, and `&` never has to be guessed at.

`iif( condition, a, b )` chooses between two values inside an expression, where an `if` statement would need several lines. Both arms are evaluated, so do not use it where one of them has an effect.

Compound assignment — `+=`, `-=`, `*=`, `/=`, `\=`, `&=` — and **`swap a, b`**, which exchanges two variables of the same type without a temporary.

Summary

- `/` is floating point, `\` is integer division, and `^` produces a `float64`.
- `mod` is a remainder and takes its left operand's sign; `FloorMod` in `std/numeric` is the mathematical one.
- **`^` groups to the left**, and **`shl`/`shr` bind tighter than `+`**.
- **`not` is looser than the comparisons**, and **`andalso`/`orelse` share one level**, grouping left to right — parenthesise mixed conditions.
- `andalso`/`orelse` short-circuit; `and`/`or`/`xor`/`not` are bitwise and always evaluate both.
- `&` joins text and never means anything else; `iif` chooses within an expression; `swap` exchanges two variables.

Exercises

1. Build a permissions value from three named bit positions, then read one back. Do it twice — once with `bitset`/`bit`, once with `1 shl n` masks — and check the two agree. {solution="ex05-01″}
2. Write expressions where parentheses change the answer, and one where they only make the existing grouping obvious. Which of the four surprises above would have bitten you? {solution="ex05-02″}
3. `7 mod -2` and `-7 mod 2` give different signs. Predict both before running them, then work out what `FloorMod` would give instead.
4. Rewrite `if p <> 0 and p->count > 0` so it is safe, and explain in one sentence what the original does when `p` is null.

Control Flow, Scope and defer

Programs choose and repeat. tiko's constructs for both will look familiar, and the reason they get a chapter is the third thing here: `defer`, which arranges for something to happen on the way out of a block — and which has one rule that catches everybody once.

Choosing

examples/ch06/branching.tiko

```
'' branching.tiko -- if, and select case for more than two ways.
dim as int32 score = 72

if score >= 90 then
    print "excellent"
elseif score >= 60 then
    print "a pass"
else
    print "not yet"
end if

'' A whole if on one line needs no 'end if'.
if score > 0 then print "there is a score"

'' select case compares one value against ranges, lists and open tests.
select case score
case 90 to 100
    print "band: top"
case 60 to 89
    print "band: middle"
case 0, 1, 2
    print "band: barely started"
case is < 0
    print "band: impossible"
case else
    print "band: low"
end select
```

examples/ch06/branching.out

```
a pass
there is a score
band: middle
```

`if` / `elseif` / `else` / `end if` needs no parentheses around the condition. A complete `if` can sit on one line, and then it takes no `end if` — useful for a guard, and confusing if you try to extend it later, so keep those short.

`select case` compares one value against many possibilities, and the cases are more expressive than a C `switch`:

Case	Matches
`case 5`	one value
`case 0, 1, 2`	any in a list
`case 60 to 89`	a range, ends included
`case is < 0`	an open comparison
`case else`	everything remaining

Cases are tested in order and the first match wins; there is no fall-through and nothing to `break` out of.

> **NOTE**
> `select case as const` compiles to a jump table, which is faster for a dense set of integer cases — and it accepts only constant values, so `case is < 0` inside one is `TKO0419`.

> **NOTE**
> **`select case true` runs the first case whose test is true.** `case ( n mod 3 ) = 0` works there: the comparison's −1 or 0 is converted to the `boolean` subject, as every case value is. When each branch asks a different question, though, `if` / `elseif` is usually the clearer shape.

Repeating

`examples/ch06/loops.tiko`

```
'' loops.tiko -- counting, repeating, and leaving early.
for i as int32 = 1 to 3
    print "for " & i
next

for i as int32 = 10 to 0 step -5
    print "down " & i
next

dim as int32 n = 0
```

```
do while n < 2
    print "while " & n
    n += 1
loop

n = 0
do
    print "until " & n
    n += 1
loop until n = 2

'' exit leaves the loop; continue skips to the next pass.
for i as int32 = 1 to 6
    if i = 2 then continue for
    if i = 4 then exit for
    print "picked " & i
next
```

examples/ch06/loops.out

```
for 1
for 2
for 3
down 10
down 5
down 0
while 0
while 1
until 0
until 1
picked 1
picked 3
```

`for … next` counts, and `step` sets the stride — negative to count down. Declaring the counter in the loop, `for i as int32 = 1 to 3`, keeps it out of the surrounding scope, which is what you want almost always.

`do … loop` carries its condition at either end, with either sense:

- `do while c … loop` — test first, may run zero times
- `do until c … loop` — the same, inverted
- `do … loop while c` and `do … loop until c` — test last, so the body always runs once

`exit for` and `exit do` leave a loop; `continue for` and `continue do` skip to the next pass. Naming the loop kind makes nested loops readable.

An older form, `while … wend`, also exists and behaves as `do while … loop`. Existing code uses it; new code has no reason to.

Blocks that exist to bound a lifetime

examples/ch06/scopeblock.tiko

```
'' scopeblock.tiko -- a block that exists only to bound a lifetime.
dim as int32 outer = 1

scope
    dim as int32 inner = 2
    defer print "  leaving the scope"
    print "  inside: outer " & outer & ", inner " & inner
end scope

'' 'inner' is gone here, and the defer has already run.
print "outside: outer " & outer
```

examples/ch06/scopeblock.out

```
  inside: outer 1, inner 2
  leaving the scope
outside: outer 1
```

`scope … end scope` introduces a block with no condition and no loop — just a boundary. Variables declared inside it end there. It is how you keep a temporary from leaking into the rest of a long procedure, and, as the next section shows, how you control exactly when a `defer` fires.

defer

`defer` takes a statement and runs it when the enclosing block is left — however it is left:

examples/ch06/deferorder.tiko

```
'' deferorder.tiko -- defer runs on the way out, in reverse order.
sub Work( byval failEarly as boolean )
    print "  open A"
    defer print "  close A"

    print "  open B"
    defer print "  close B"

    if failEarly then
        print "  leaving early"
```

```
        exit sub
    end if

    print "  doing the work"
end sub

print "normal path:"
Work( false )
print
print "early exit:"
Work( true )
```

examples/ch06/deferorder.out

```
normal path:
  open A
  open B
  doing the work
  close B
  close A

early exit:
  open A
  open B
  leaving early
  close B
  close A
```

Two things to read off that output. Deferred statements run in **reverse order**, so resources are released in the opposite order to the one they were acquired in — B before A. And they run on **every** path out, including the early `exit sub`, which is the entire point: the cleanup is written next to the thing it cleans up, and cannot be skipped by a return you add later.

That makes `defer` the answer to the oldest bug in resource handling — the path that forgot to close something:

```
open "data.txt" for input as #1
defer close #1
```

Now the rule that catches everyone.

PITFALL

`defer` belongs to the nearest enclosing block, not to the procedure. An `if`, a loop body and a `scope` are all blocks.

examples/ch06/defertrap.tiko

```
'' defertrap.tiko -- defer belongs to the nearest enclosing BLOCK.
sub Guarded( )
    print "  start"
    if true then
        print "    acquired"
        defer print "    released"    '' runs at end if, not end sub
        print "    still inside the if"
    end if
    print "  after the if -- already released"
end sub

sub EachIteration( )
    for i as int32 = 1 to 3
        defer print "    end of pass " & i
        print "  pass " & i
    next
end sub

print "a defer inside an if:"
Guarded( )
print
print "a defer inside a loop body:"
EachIteration( )
```

examples/ch06/defertrap.out

```
a defer inside an if:
  start
    acquired
    still inside the if
    released
  after the if -- already released

a defer inside a loop body:
  pass 1
    end of pass 1
  pass 2
    end of pass 2
  pass 3
    end of pass 3
```

Read the first half: the release happens at `end if`, while the procedure is still running. A guard written as *acquire inside an `if`, then defer* releases the resource immediately and leaves the rest of the procedure using something already closed — and nothing in the code looks wrong.

The fix is to leave early instead of nesting: test the failure case, `exit sub`, and put the acquire-and-defer pair in the procedure's own body. Exercise 2 shows both side by side.

In a loop body the same rule is exactly what you want — the deferred statement runs at the end of **each** pass, as the second half of the output shows.

`defer` needs a block to be on the way out of, so it cannot appear at module level:

examples/ch06/modulelevel.tiko

```
'' modulelevel.tiko -- defer needs a scope to leave, and a module
'' body is not one. Does not build.
defer print "this never had a scope to leave"

print "hello"
```

```
modulelevel.tiko(3,1): error TKO0814: 'defer' is not allowed at module
level; put it
in a procedure or a block
```

Labels and goto

tiko has `goto` and line labels. They exist for two reasons: compatibility with older code, and `on error goto`, which Chapter 18 covers.

Nothing in this book uses `goto` for control flow. Every loop construct above, plus `exit` and `continue`, covers what it was for — and a `goto` that jumps across a `defer` is refused (`TKO0827`), because the compiler cannot say what should have run on the way.

Summary

- `if` / `elseif` / `else`; a one-line `if` takes no `end if`.
- `select case` matches values, lists, ranges and open comparisons, first match wins, no fall-through. `as const` builds a jump table and takes only constants.
- **A comparison yields an integer, not a `boolean`** — wrap with `cbool` before comparing one against `true`.
- `for … next` with `step`; `do … loop` with the test at either end; `exit` and `continue`.
- `scope … end scope` bounds a lifetime deliberately.

- **`defer` runs on the way out of its block, in reverse order, on every path** — including the end of each loop pass, and including an `if` you wrapped it in.

Exercises

1. Write FizzBuzz for 1 to 15. Which reads better here, `if`/`elseif` or `select case`, and why? {solution="ex06-01″}
2. Write a procedure that acquires something inside an `if` and defers its release, and a second that guards first and defers in the procedure body. Print from both and compare the order. {solution="ex06-02″}
3. Put a `defer` in a loop body that prints the loop counter. Before running it, predict how many times it fires and with which values.
4. Rewrite a `do … loop until` as a `do while … loop` that behaves identically, including when the condition is already true at the start. Which needed more care?

Procedures

A procedure is the unit you give a name to, and naming a piece of work is most of what programming is. tiko's are ordinary — until you look at how arguments get in and results get out, where three things behave differently from what the syntax suggests.

Start with the reason parameters exist at all. A procedure cannot see the variables around it:

examples/ch07/sealed.tiko

```
'' sealed.tiko -- a procedure cannot see the module's variables.
dim as int32 counter = 5

sub ShowIt( )
    print counter        '' error: procedures start with an empty slate
end sub

ShowIt( )
```

```
sealed.tiko(5,11): error TKO0743: 'counter' is not declared
     |
   5 |     print counter        '' error: procedures start with an empty
slate
     |           ^
```

A module-level variable is visible to the module's own statements and to nothing else. Adding `shared` to the declaration would open it up to every procedure in the file, which is the reason this book almost never does: a procedure whose inputs all arrive as parameters can be read on its own, and tested on its own.

Subs, functions, and three ways to answer

examples/ch07/basics.tiko

```
'' basics.tiko -- a sub does something; a function answers something.

sub Banner( byref title as ustring )
    print "== " & title & " =="
end sub

'' Three ways to hand back a result. They compile to the same thing.
function Doubled( byval n as int32 ) as int32
    return n * 2
end function
```

```
function Tripled( byval n as int32 ) as int32
    function = n * 3
end function

function Quadrupled( byval n as int32 ) as int32
    Quadrupled = n * 4
end function

Banner( "results" )
print Doubled( 5 ), Tripled( 5 ), Quadrupled( 5 )

'' A function called for its answer and ignored is still called.
function Noisy( ) as int32
    print "Noisy ran"
    return 0
end function

Banner( "called for effect" )
Noisy( )
```

examples/ch07/basics.out

```
== results ==
 10            15            20
== called for effect ==
Noisy ran
```

A `sub` does something. A `function` answers something, and has three spellings for handing that answer back:

- **`return value`** — sets the result and leaves immediately.
- **`function = value`** — sets the result and carries on. Useful when the answer is refined further down.
- **`Name = value`**, the function's own name — the oldest form, and the one most likely to be misread as a recursive call.

They compile to the same thing. This book uses `return`, and `function =` only where the procedure really does keep working after the answer is settled.

A function may also be called and its answer discarded, as `Noisy( )` is above — nothing forces you to use it.

The answer is usually a single value, but it can be a whole array: write `( )` after the result type, as in `function Squares( ... ) as int32( )`. Chapter 8 covers what such a function may return and who owns the result.

> **NOTE**
> A function that reaches `end function` without ever setting its result returns a zero-initialised value, silently. There is no warning. If a branch can fall off the end, put an explicit `return` there and say what the answer should be.

declare, for a name the compiler has not read yet

The compiler reads a file once, top to bottom, and a name has to exist before it is used. Two procedures that call each other cannot both be written first:

examples/ch07/forward.tiko

```
'' forward.tiko -- declare names a procedure before it exists.
''
'' IsEven and IsOdd call each other, so whichever is written first
'' refers to a name the compiler has not read yet.

declare function IsOdd( byval n as int32 ) as boolean

function IsEven( byval n as int32 ) as boolean
    if n = 0 then return true
    return IsOdd( n - 1 )
end function

function IsOdd( byval n as int32 ) as boolean
    if n = 0 then return false
    return IsEven( n - 1 )
end function

for i as int32 = 0 to 4
    print i; " even="; IsEven( i ); " odd="; IsOdd( i )
next
```

examples/ch07/forward.out

```
 0 even=true odd=false
 1 even=false odd=true
 2 even=true odd=false
 3 even=false odd=true
 4 even=true odd=false
```

`declare` states the signature and nothing else. Once it has been seen, the name can be called anywhere below, and the body can arrive whenever it likes. Without the `declare`, the call is `TKO0743: '…' is not declared` — the same error a typo gets, because to the compiler it is the same mistake.

> **NOTE**
> When you name a procedure you are competing with the names the compiler predeclares. `function Double( )` does not compile — `double` is one of the classic type names — and it fails with `TKO0600: 'double' is already declared in this scope` rather than with anything about types. Appendix A lists the names that collide.

How arguments get in

This is the part to read twice. tiko lets you write a parameter with no `byval` or `byref`, and **what that means depends on the parameter's type**:

examples/ch07/passing.tiko

```
'' passing.tiko -- what a parameter with no byval or byref means.
''
'' Every procedure here writes to its parameter. Whether the caller
'' sees that write is the whole question.

sub PlainInt( n as int32 )
    n = 99
end sub

sub PlainStr( s as ustring )
    s = "changed"
end sub

'' The same two, written the other way round.
sub ByrefInt( byref n as int32 )
    n = 99
end sub

sub ByvalStr( byval s as ustring )
    s = "changed"
end sub

dim as int32 a = 1
PlainInt( a )
print "int32   with no keyword: "; a

dim as ustring t = "kept"
PlainStr( t )
print "ustring with no keyword: " & t
```

```
dim as int32 b = 1
ByrefInt( b )
print "int32   byref:           "; b

dim as ustring u = "kept"
ByvalStr( u )
print "ustring byval:           " & u
```

examples/ch07/passing.out

```
int32   with no keyword:  1
ustring with no keyword: changed
int32   byref:            99
ustring byval:           kept
```

With no keyword, an `int32` was **copied** — the caller still holds 1. A `ustring` was **shared** — the caller's text changed under it.

Both procedures are written identically.

The rule the compiler applies:

Parameter type	Default
Numbers, `boolean`, pointers, enums	`byval` — a copy
`string`, `ustring`	`byref` — the caller's own variable
Types you define	`byref` — the caller's own variable

PITFALL

An omitted keyword means two different things. The same spelling copies a number and aliases a string, so a procedure that tidies up its own parameter is harmless with an `int32` and quietly corrupts the caller's data with a `ustring`.

It is the direction of the surprise that makes it expensive. Nobody writes `sub Fix( n as int32 )` expecting to change the caller's number. Plenty of people write `sub Fix( s as ustring )` and use `s` as scratch space — and the damage then appears somewhere else entirely, in a variable they were not thinking about.

Write `byval` or `byref` on every parameter. Every listing in this book does. It costs six characters and it removes the question.

`byref` is not a wart to be avoided — it is how you pass something large without copying it, and how a procedure reports more than one answer. Exercise 1 builds a `MinMax` that returns two values that way, which is the tool to reach for until Chapter 20 introduces tuples.

const, for a reference you promise not to write through

`byref` gives speed and gives up safety: the callee can now write to the caller's variable. Adding `const` takes the safety back while keeping the speed.

examples/ch07/constparam.tiko

```
'' constparam.tiko -- const says the procedure will not write to it.
sub Bump( byval n as const int32 )
    n += 1              '' error: that is what const forbids
end sub

Bump( 1 )
```

```
constparam.tiko(3,5): error TKO0617: cannot modify 'n': it is a constant
     |
   3 |     n += 1              '' error: that is what const forbids
     |     ^
```

`byref x as const T` is the right shape for any parameter you only intend to read, and it is documentation the compiler checks — for every type, text included: `byref s as const ustring` may not be assigned either.

Arguments you can leave out

examples/ch07/optional.tiko

```
'' optional.tiko -- defaults, and leaving an argument out of the middle.

sub Label( byref who as ustring, byval times as int32 = 1, _
           byref mark as ustring = "!" )
    for i as int32 = 1 to times
        print who & mark
    next
end sub

Label( "once" )
Label( "twice", 2 )
Label( "marked", , "?" )      '' times keeps its default
```

examples/ch07/optional.out

```
once!
twice!
twice!
marked?
```

A parameter with `= value` may be omitted at the call. Optional parameters must come after the required ones, and an argument in the *middle* can be skipped by leaving its slot empty — `Label( "marked", , "?" )` takes the default for `times` and supplies `mark`.

The empty slot is easy to misread. Two of them in a row, `Label( "x", , )`, is legal and tells the reader nothing; when a call needs more than one hole, the parameters are in the wrong order.

One name, several signatures

`examples/ch07/overloads.tiko`

```
'' overloads.tiko -- one name, several signatures.

declare function Area overload ( byval side as float64 ) as float64
declare function Area overload ( byval w as float64, _
                                 byval h as float64 ) as float64
declare function Area overload ( byref tag as ustring ) as float64

function Area overload ( byval side as float64 ) as float64
    print "  (square)"
    return side * side
end function

function Area overload ( byval w as float64, _
                         byval h as float64 ) as float64
    print "  (rectangle)"
    return w * h
end function

function Area overload ( byref tag as ustring ) as float64
    print "  (by name: " & tag & ")"
    return 0.0
end function

'' Each call prints which overload ran, then its answer.
var square = Area( 3.0 )
print "Area( 3.0 )      ="; square

var rect = Area( 2.0, 5.0 )
print "Area( 2.0, 5.0 ) ="; rect

var widened = Area( 3 )
print "Area( 3 )        ="; widened
```

```
var named = Area( "lot" )
print "Area( ""lot"" )    ="; named
```

examples/ch07/overloads.out

```
  (square)
Area( 3.0 )      = 9
  (rectangle)
Area( 2.0, 5.0 ) = 10
  (square)
Area( 3 )        = 9
  (by name: lot)
Area( "lot" )    = 0
```

`overload` says that a name is deliberately shared. It is required: declaring a second `Area` without it is `TKO0600: 'area' is already declared in this scope`, the same error as any duplicate name — which is the right default, because most repeated names are accidents.

Note what happened to `Area( 3 )`. There is no `int32` overload, so the argument was converted to `float64` and the one-parameter version ran. The compiler scores the conversions each candidate would need and takes the cheapest total. When two candidates tie, it refuses rather than guessing:

examples/ch07/ambiguous.tiko

```
'' ambiguous.tiko -- neither overload is a better match than the other.
declare function Pick overload ( byval a as int32, _
                                 byval b as float64 ) as int32
declare function Pick overload ( byval a as float64, _
                                 byval b as int32 ) as int32

function Pick overload ( byval a as int32, _
                         byval b as float64 ) as int32
    return 1
end function

function Pick overload ( byval a as float64, _
                         byval b as int32 ) as int32
    return 2
end function

print Pick( 1, 1 )       '' error: int32 converts to float64 either way
```

```
ambiguous.tiko(17,11): error TKO0660: the call to 'pick' matches more
than one overload
     |
  17 | print Pick( 1, 1 )      '' error: int32 converts to float64 either
way
     |           ^
```

Scoring counts conversions; it does not rank them by whether they can lose information. Exercise 2 shows an `int64` argument picking an `int32` overload — narrowing — in preference to widening to `float64`. If a width matters, give it an overload of its own.

Recursion, and locals that remember

examples/ch07/gcd.tiko

```
'' gcd.tiko -- recursion, and a static local that outlives the call.

function Gcd( byval a as int64, byval b as int64 ) as int64
    static as int32 calls = 0
    calls += 1

    if b = 0 then
        print "  reached the base case after"; calls; " calls"
        calls = 0
        return a
    end if

    return Gcd( b, a mod b )
end function

print "Gcd( 1071, 462 ) ="; Gcd( 1071, 462 )
print "Gcd( 270, 192 )  ="; Gcd( 270, 192 )
```

examples/ch07/gcd.out

```
Gcd( 1071, 462 ) =  reached the base case after 4 calls
 21
Gcd( 270, 192 )  =  reached the base case after 5 calls
 6
```

A procedure may call itself; each call gets its own parameters and its own locals. `static` is the exception — a `static` local is created once, keeps its value between calls, and so is shared by every level of a recursion. That is what makes it usable as a

call counter, and it is also why `Gcd` has to reset it before returning: the count belongs to one top-level call, and nothing else will clear it.

`static` initialisers run once, on the first call. A `static` is the smallest thing that gives a procedure memory, and the smallest thing that stops it being safe to call from two threads at once — Chapter 23 returns to that.

Returning a place instead of a value

A function can hand back a reference to storage, which the caller may then assign *to*. Write `byref` after the parameter list:

examples/ch07/byrefresult.tiko

```
'' byrefresult.tiko -- a function that returns a place, not a value.

function SlotAt( byval i as int32 ) byref as int32
    static as int32 first  = 0
    static as int32 second = 0
    if i = 0 then return first
    return second
end function

print "start:            "; SlotAt( 0 ); SlotAt( 1 )

'' The obvious spelling. Watch what it does.
SlotAt( 0 ) = 42
print "after SlotAt(0)=42:"; SlotAt( 0 ); SlotAt( 1 )

'' Parenthesise the call, and the assignment lands.
( SlotAt( 0 ) ) = 42
print "after (SlotAt(0))=42:"; SlotAt( 0 ); SlotAt( 1 )

'' Or bind the reference to a name first, which reads better.
dim byref as int32 slot = SlotAt( 1 )
slot = 7
print "after slot = 7:   "; SlotAt( 0 ); SlotAt( 1 )
```

examples/ch07/byrefresult.out

```
start:             0 0
after SlotAt(0)=42: 0 0
after (SlotAt(0))=42: 42 0
after slot = 7:    42 7
```

Read the second line of that output before the third.

PITFALL

`SlotAt( 0 ) = 42` does nothing, and says nothing.

Inside an argument list `=` is the equality operator, so the compiler reads `SlotAt( 0 = 42 )`: a call with one `boolean` argument, whose result is thrown away. It is a complete, legal statement. The assignment you wrote is gone, and the program carries on with the old value.

Two spellings work, both in the listing above:

```
( SlotAt( 0 ) ) = 42                    '' parenthesise the call
```

```
dim byref as int32 slot = SlotAt( 1 )       '' or bind it to a name
slot = 7
```

The second is worth preferring. It names the thing being modified, and it cannot be broken later by someone removing a pair of parentheses that looked redundant.

The compiler says so, at least: it warns `TKO0931: '=' is read as a comparison inside the argument of 'slotat', not as an assignment to its byref result`. Read that warning as an error. (With two or more arguments there is no ambiguity — `( a, b ) = v` is not an expression — and `f( a, b ) = v` assigns through the result as written.)

Any number of arguments

A `Sum` that adds two numbers is easy, and so is one that adds three. A `Sum` that adds however many it is given is the one you actually want, and overloads cannot get there: there would have to be one per count. An array parameter can, but then every caller builds an array before every call.

A **typed variadic** is the array parameter with the caller's convenience:

`examples/ch07/typedvariadic.tiko`

```
'' typedvariadic.tiko -- any number of arguments, every one typed.
''
'' The '...' after the type makes nums an array parameter that the
'' caller fills with a plain list. Each argument is converted to
'' int32 exactly as it would be for an ordinary int32 parameter.

function Sum( nums() as const int32 ... ) as int64
    dim as int64 total = 0
    for i as int32 = lbound( nums ) to ubound( nums )
        total += nums( i )
    next
```

```
    return total
end function

'' Fixed parameters come first; the variadic is always the last.
sub Report( byref title as const ustring, _
            nums() as const int32 ... )
    print title; ":"; ubound( nums ) - lbound( nums ) + 1; _
          " values, sum"; Sum( nums( ) )
end sub

print "Sum( 10, 20, 30 ) ="; Sum( 10, 20, 30 )
print "Sum( 7 )          ="; Sum( 7 )
print "Sum( )            ="; Sum( )

Report( "three", 1, 2, 3 )
Report( "none" )

'' A whole array of the element type IS the list -- nothing is
'' copied, and no count has to be passed alongside it.
dim as int32 week( 0 to 6 ) = { 3, 5, 2, 8, 6, 4, 7 }
Report( "week", week( ) )
```

examples/ch07/typedvariadic.out

```
Sum( 10, 20, 30 ) = 60
Sum( 7 )          = 7
Sum( )            = 0
three: 3 values, sum 6
none: 0 values, sum 0
week: 7 values, sum 35
```

`nums() as const int32 ...` is an array parameter with `...` after its type. The caller writes a plain list. The compiler converts each argument to `int32`, exactly as it would for an ordinary `int32` parameter, and packs them into a temporary array that lives until the end of the statement. Inside, `nums` is an ordinary array: `lbound` is 0, and `ubound` is −1 when nothing was passed, so the loop above needs no special case for `Sum( )`.

Fixed parameters come first and the variadic is always last — anywhere else is `TKO0973`. It takes no default value (`TKO0974`), because no arguments already means an empty array.

A whole array of the element type is the list itself. `Report( "week", week( ) )` passes `week` as it stands: nothing is copied and no count travels alongside it, because

the array carries its own bounds. That is the form to use when the values are already in an array, or when the list is built while the program runs.

Every argument is checked, which is the whole difference from the C mechanism further down:

examples/ch07/variadictyped.tiko

```
'' variadictyped.tiko -- every packed argument is checked.

function Sum( nums() as const int32 ... ) as int64
    dim as int64 total = 0
    for i as int32 = lbound( nums ) to ubound( nums )
        total += nums( i )
    next
    return total
end function

print Sum( 1, "two", 3 )    '' error: "two" is no int32
```

```
variadictyped.tiko(11,15): error TKO0698: this argument does not convert
to the parameter's type
     |
  11 | print Sum( 1, "two", 3 )    '' error: "two" is no int32
     |               ^
```

It is the same error a non-converting argument to an ordinary parameter gets, at the argument that caused it.

An overload that takes its arguments without packing them always wins over one that needs to:

examples/ch07/variadicpick.tiko

```
'' variadicpick.tiko -- an exact match beats a packed one.

sub Show overload ( byval n as int32 )
    print "one int32:"; n
end sub

sub Show overload ( nums() as const int32 ... )
    print "a list of"; ubound( nums ) - lbound( nums ) + 1
end sub

Show( 5 )            '' the plain overload: no packing needed
Show( 5, 6 )         '' only the list can take two
Show( )              '' ... or none
```

examples/ch07/variadicpick.out

```
one int32: 5
a list of 2
a list of 0
```

So adding a variadic overload never changes which procedure an existing call runs. Among candidates that all pack, each packed argument is scored like any other argument.

PITFALL

A packed list is a copy; a spread array is the caller's own. Without `const`, a variadic procedure may write to its array, and where the writes land depends on how it was called:

examples/ch07/variadicalias.tiko

```
'' variadicalias.tiko -- a packed list is a copy; a spread array
'' is the caller's own.

sub Bump( nums() as int32 ... )      '' no const: it may write
    for i as int32 = lbound( nums ) to ubound( nums )
        nums( i ) += 100
    next
end sub

dim as int32 a = 1, b = 2
Bump( a, b )                         '' packed into a temporary
print "after Bump( a, b ):    "; a; b

dim as int32 both( 0 to 1 ) = { 1, 2 }
Bump( both( ) )                      '' the array itself
print "after Bump( both( ) ): "; both( 0 ); both( 1 )
```

examples/ch07/variadicalias.out

```
after Bump( a, b ):     1 2
after Bump( both( ) ):  101 102
```

`Bump( a, b )` changed a temporary, which disappeared at the end of the statement; `a` and `b` are untouched. `Bump( both( ) )` changed `both`. The procedure is the same and the call looks almost the same, and the two do different things without a word from the compiler.

Declare the variadic `const` unless writing to the caller's array is the point. Then the question cannot arise.

The element type can be one of your own, and the conversion then goes through that type's constructors — Chapter 12. That is how `strings.Format` in Chapter 9 takes values of every type in one list: its element type has a constructor for each.

C's ...

examples/ch07/varargs.tiko

```
'' varargs.tiko -- a procedure taking any number of arguments.
''
'' This is C's mechanism, so it needs C's calling convention and
'' none of the type checking you get everywhere else.

#include once "crt/stdarg.bi"

function SumOf cdecl ( byval count as int32, ... ) as int64
    dim as cva_list args
    dim as int64 total = 0

    cva_start( args, count )
    for i as int32 = 1 to count
        total += cva_arg( args, int32 )
    next
    cva_end( args )

    return total
end function

print "SumOf( 3, 10, 20, 30 ) ="; SumOf( 3, 10, 20, 30 )
print "SumOf( 1, 7 )          ="; SumOf( 1, 7 )
print "SumOf( 0 )             ="; SumOf( 0 )
```

examples/ch07/varargs.out

```
SumOf( 3, 10, 20, 30 ) = 60
SumOf( 1, 7 )          = 7
SumOf( 0 )             = 0
```

A bare `...` as the last parameter, with no name and no type, is C's mechanism, with C's consequences: the procedure must be `cdecl`, there is no type checking, and the count is recorded nowhere — `SumOf` is *told* how many to read because there is no other way for it to find out. Reading one argument too many reads whatever happened to be next.

Declaring the procedure without `cdecl` is refused: `TKO0781: '...' needs a 'cdecl' procedure, and this one is 'stdcall'.`

C's variadics exist so that C libraries can be called; `printf` is the reason the machinery is there. In tiko code, use a typed variadic. Chapter 29 covers `cva_list` in full, alongside the rest of C interoperation.

Procedures as values

examples/ch07/callbacks.tiko

```
'' callbacks.tiko -- a procedure held in a variable.

function Twice( byval x as float64 ) as float64
    return x * 2.0
end function

function Square( byval x as float64 ) as float64
    return x * x
end function

'' The parameter type names a shape, not a procedure.
sub Tabulate( byref title as ustring, _
              byval f as function( byval as float64 ) as float64 )
    print title
    for i as int32 = 1 to 3
        print "  f("; i; " ) ="; f( i )
    next
end sub

Tabulate( "doubling", @Twice )
Tabulate( "squaring", procptr( Square ) )

'' A variable of that type can be reassigned like any other.
dim as function( byval as float64 ) as float64 op = @Twice
print "op( 4 ) ="; op( 4.0 )
op = @Square
print "op( 4 ) ="; op( 4.0 )
```

examples/ch07/callbacks.out

```
doubling
  f( 1 ) = 2
  f( 2 ) = 4
  f( 3 ) = 6
squaring
  f( 1 ) = 1
  f( 2 ) = 4
  f( 3 ) = 9
```

```
op( 4 ) = 8
op( 4 ) = 16
```

A procedure has a type, spelled the way it is declared minus the names: `function( byval as float64 ) as float64`. A variable or parameter of that type holds any procedure matching the shape, and is called with ordinary call syntax.

Two spellings get the address: `@Twice` and `procptr( Twice )`. They are the same thing, and `procptr` is the one to use for an overloaded name, where the signature picks which one you meant.

A procedure pointer is the plainest form of the idea that the caller decides *what* to do and the procedure decides *when*. Chapter 21 extends it to lambdas, which add the ability to capture variables from around them — the one thing a procedure pointer cannot do, because a pointer is a single address with no room for anything else.

What it costs

A call is cheap and not free: arguments are placed, a return address is pushed, the callee may save registers. None of that matters at ordinary granularity, and all of it matters in a loop running millions of times.

The costs worth knowing are the ones you choose:

- **`byval` on a large value copies it**, every call. That is what `byref ... as const` is for.
- **`byref` on a small value** adds an indirection to save copying eight bytes. It is a loss.
- **A procedure pointer cannot be inlined**, because the target is not known until it runs. A direct call can be.
- **A packed variadic call builds a temporary array**, every call. Passing an array you already have costs nothing extra.
- **`static` locals make a procedure stateful**, which is a correctness cost before it is ever a performance one.

Summary

- A procedure sees only what it is given; module-level variables need `shared`, which is rarely worth it.
- `return`, `function =` and assigning the procedure's own name are three spellings of one thing. A function that sets no result returns zero, silently.
- `declare` introduces a name before its body, which mutual recursion requires.
- **With no keyword, numbers are passed `byval`, and strings and your own types `byref`.** Write the keyword on every parameter.

- `byref x as const T` is a reference you cannot write through, whatever T is.
- Optional parameters follow the required ones, and a middle argument can be left empty.
- `overload` is required to share a name; resolution scores conversions and refuses ties with `TKO0660`. It does not avoid narrowing.
- `static` locals persist between calls and are shared by every level of a recursion.
- **Assigning to a one-argument `byref` function result needs `( f( x ) ) = v` or a `dim byref` name** — the bare form parses as an equality test, and `TKO0931` warns that it did.
- **A typed variadic, `nums() as const T ...`**, takes any number of arguments, each converted to `T` and checked; the procedure sees an ordinary array. A whole array of `T` is the list itself, with no copy.
- An overload that needs no packing beats one that does. **A packed list is a copy and a spread array is the caller's own** — declare the variadic `const`.
- C's bare `...` needs `cdecl` and does no type checking; it is for calling C.
- Procedure pointers hold any procedure of a matching shape; `@P` and `procptr( P )` are the same.
- The costs you choose: a `byval` copy of something large, a `byref` indirection to save eight bytes, a procedure pointer that cannot be inlined, and a `static` that makes a procedure stateful.

Exercises

1. Write `MinMax`, which takes three `int32` values and reports the smallest and largest through `byref` parameters. Which of its five parameters are `byval` and which `byref`, and what happens if you omit the keywords entirely? {solution="ex07-01″}
2. Overload `Describe` for `int32`, `float64` and `ustring`, then call it with an `int16` and an `int64`. Predict which overload each picks before you run it. {solution="ex07-02″}
3. Write a function with a `byref` result, then assign to it with the bare `f( i ) = v` form. Before running the program, say what you expect; afterwards, explain what the compiler actually compiled.
4. Give a recursive procedure a `static` call counter and *don't* reset it. Call the procedure twice and explain the second number.
5. Take `Tabulate` from `callbacks.tiko` and add a third operation without touching `Tabulate`. Then do the same with an overload instead of a procedure pointer. Which change would you rather review?
6. Write `Average( xs() as const float64 ... )`. Decide what `Average( )` should return, and make it do that. Then pass it a whole `int32` array: why is that refused, when `Average( 1, 2, 3 )` with `int32` arguments is not?

Arrays

An array is many values of one type under one name, reached by number. It is the oldest data structure there is and still the one to reach for first: the elements sit next to each other in memory, so walking an array is the fastest thing a processor does.

tiko's arrays carry their bounds with them — not a length, but a first and last index, which the compiler and the runtime both know. That one decision explains most of what follows.

Bounds, not length

examples/ch08/bounds.tiko

```
'' bounds.tiko -- an array's bounds are part of its declaration.

dim as int32 a( 10 )           '' how many elements is that?
dim as int32 b( 1 to 5 )
dim as int32 c( -2 to 2 )

print "a( 10 )      "; lbound( a ); " to"; ubound( a ); _
      "  elements:"; ubound( a ) - lbound( a ) + 1
print "b( 1 to 5 )  "; lbound( b ); " to"; ubound( b ); _
      "  elements:"; ubound( b ) - lbound( b ) + 1
print "c( -2 to 2 ) "; lbound( c ); " to"; ubound( c ); _
      "  elements:"; ubound( c ) - lbound( c ) + 1

'' Every element starts at zero. Nothing is left uninitialised.
print "a( 7 ) before anything is written:"; a( 7 )

'' An initialiser gives the values in order.
dim as int32 d( 1 to 3 ) = { 10, 20, 30 }
print "d ="; d( 1 ); d( 2 ); d( 3 )
```

examples/ch08/bounds.out

```
a( 10 )       0 to 10  elements: 11
b( 1 to 5 )   1 to 5  elements: 5
c( -2 to 2 ) -2 to 2  elements: 5
a( 7 ) before anything is written: 0
d = 10 20 30
```

`lbound( a )` and `ubound( a )` are the first and last valid index. Every loop over an array in this book is written between them, and never between `0` and some remembered count.

PITFALL

`dim as int32 a( 10 )` declares eleven elements, not ten. A single number is the *upper bound*, and the lower bound defaults to 0 — so the valid indices are 0 through 10.

Anyone arriving from C, Java, Python or Rust will read that line as "ten elements" and write a loop from 0 to 9, quietly ignoring the last one. The reverse mistake is worse: read it as "indices 0 to 9" in a language where the eleventh exists, and a bug hides in the element nobody looks at.

Write both bounds. `dim as int32 a( 0 to 9 )` says exactly what it means, in the same number of keystrokes, and it cannot be misread.

Elements start at zero — or empty, for a string — so there is no such thing as an uninitialised array element. An initialiser in braces fills them in order.

A lower bound other than 0 is worth using when the index means something. `band( 0 to 9 )` in the histogram below is indexed by tens, and `temperature( 1900 to 2000 )` indexed by year needs no arithmetic at the point of use, which is where arithmetic goes wrong.

Arrays that change size

examples/ch08/dynamic.tiko

```
'' dynamic.tiko -- an array whose size is decided while running.

dim as int32 a( )              '' declared, but with no room yet

print "empty:  lbound="; lbound( a ); " ubound="; ubound( a )
print "  ubound < lbound is how an empty array reads"

redim a( 1 to 3 )
a( 1 ) = 10 : a( 2 ) = 20 : a( 3 ) = 30
print "filled:"; a( 1 ); a( 2 ); a( 3 )

'' redim on its own does NOT keep what was there.
redim a( 1 to 5 )
print "after redim( 1 to 5 ):";
for i as int32 = lbound( a ) to ubound( a ) : print a( i ); : next
print

redim a( 1 to 3 )
a( 1 ) = 10 : a( 2 ) = 20 : a( 3 ) = 30
redim preserve a( 1 to 5 )
```

```
print "after redim preserve:";
for i as int32 = lbound( a ) to ubound( a ) : print a( i ); : next
print

'' Shrinking with preserve keeps the elements that still fit.
redim preserve a( 1 to 2 )
print "after shrinking:";
for i as int32 = lbound( a ) to ubound( a ) : print a( i ); : next
print

erase a
print "after erase: lbound="; lbound( a ); " ubound="; ubound( a )
```

examples/ch08/dynamic.out

```
empty:  lbound= 0 ubound=-1
  ubound < lbound is how an empty array reads
filled: 10 20 30
after redim( 1 to 5 ): 0 0 0 0 0
after redim preserve: 10 20 30 0 0
after shrinking: 10 20
after erase: lbound= 0 ubound=-1
```

`dim as int32 a( )` declares an array with no room yet. Its `ubound` is **below** its `lbound`, which is how an empty array reads — and why `for i = lbound( a ) to ubound( a )` over an empty array runs zero times instead of needing a guard.

`redim` gives it a size. `redim preserve` gives it a size **and keeps what was there**, growing with zeros and shrinking by truncation.

> **PITFALL**
> **`redim` without `preserve` discards every element.** It is not a resize; it is a fresh array at the same name.
>
> That is easy to accept when you read it here and easy to miss in code, because nothing about `redim a( 1 to 5 )` looks destructive, and the array is still there afterwards — just full of zeros. The listing above shows `10 20 30` becoming `0 0 0 0 0`.
>
> Reach for `redim preserve` by default, and use plain `redim` only when you mean to start over.

`erase` does two different things, and which one depends on the array:

examples/ch08/erasefixed.tiko

```
'' erasefixed.tiko -- erase means two different things.

dim as int32 fixed_( 1 to 3 ) = { 7, 8, 9 }
dim as int32 grown( )
redim grown( 1 to 3 )
grown( 1 ) = 7 : grown( 2 ) = 8 : grown( 3 ) = 9

erase fixed_
erase grown

print "fixed after erase: ubound="; ubound( fixed_ ); _
      "  values:"; fixed_( 1 ); fixed_( 2 ); fixed_( 3 )
print "dynamic after erase: ubound="; ubound( grown ); _
      "  (the storage is gone)"
```

examples/ch08/erasefixed.out

```
fixed after erase: ubound= 3  values: 0 0 0
dynamic after erase: ubound=-1  (the storage is gone)
```

On a **dynamic** array, `erase` frees the storage and leaves it empty. On a **fixed** array there is nothing to free, so it resets the elements and the bounds stay. One keyword, two meanings, neither surprising once you know which array you are holding.

An array indexed by what it counts

examples/ch08/histogram.tiko

```
'' histogram.tiko -- an array indexed by the thing you are counting.

'' Sixteen measurements, each a score from 0 to 100.
dim as int32 scores( 1 to 16 ) = { 93, 71, 55, 88, 64, 79, 100, 42, _
                                   67, 85, 90, 73, 58, 96, 81, 77 }

'' One counter per band of ten. The bounds say what the index MEANS:
'' band 4 counts 40-49. A perfect score is clamped into the top band.
dim as int32 band( 0 to 9 )

for i as int32 = lbound( scores ) to ubound( scores )
    dim as int32 b = scores( i ) \ 10
    if b > ubound( band ) then b = ubound( band )
    band( b ) += 1
next

for b as int32 = ubound( band ) to lbound( band ) step -1
```

```
    dim as ustring bar = ""
    for n as int32 = 1 to band( b ) : bar &= "#" : next
    print b * 10; "-"; b * 10 + 9; " "; bar
next
```

examples/ch08/histogram.out

```
 90- 99 ####
 80- 89 ###
 70- 79 ####
 60- 69 ##
 50- 59 ##
 40- 49 #
 30- 39
 20- 29
 10- 19
 0- 9
```

This is the shape worth recognising: the index *is* the data. `band( score \ 10 )` needs no search and no comparison chain — the value computes its own slot, and counting is one addition.

The bounds document the meaning. `band( 0 to 9 )` says there are ten bands of ten; the clamp says what happens to a perfect score. Both are visible at the declaration rather than implied by a loop somewhere below.

More than one subscript

examples/ch08/matrix.tiko

```
'' matrix.tiko -- more than one subscript, and how it sits in memory.

dim as int32 m( 0 to 1, 0 to 2 )

dim as int32 n = 0
for r as int32 = lbound( m, 1 ) to ubound( m, 1 )
    for c as int32 = lbound( m, 2 ) to ubound( m, 2 )
        m( r, c ) = n
        n += 1
    next
next

print "dimension 1:"; lbound( m, 1 ); " to"; ubound( m, 1 )
print "dimension 2:"; lbound( m, 2 ); " to"; ubound( m, 2 )

print "as a grid:"
```

```
for r as int32 = lbound( m, 1 ) to ubound( m, 1 )
    print "  ";
    for c as int32 = lbound( m, 2 ) to ubound( m, 2 )
        print m( r, c );
    next
    print
next

'' The same storage, read as one flat run of int32.
dim as int32 ptr p = @m( 0, 0 )
print "in memory:";
for i as int32 = 0 to 5
    print p[ i ];
next
print
```

examples/ch08/matrix.out

```
dimension 1: 0 to 1
dimension 2: 0 to 2
as a grid:
   0 1 2
   3 4 5
in memory: 0 1 2 3 4 5
```

Dimensions are separated by commas, and `lbound`/`ubound` take the dimension number as a second argument: `ubound( m, 2 )` is the last column index. An array may have up to eight dimensions.

The last line of that output is the one to keep. Reading the same storage as a flat run of `int32` gives `0 1 2 3 4 5` — the order the elements were written, row by row. **The last subscript varies fastest.** A row is contiguous; a column is strided.

That is not a detail of the implementation you are free to ignore, because it decides how fast the loop runs. Iterating with the last subscript innermost walks memory forwards and every cache line that is fetched gets used. Swapping the loops reads one element per cache line, and on a large array that is the difference between fast and slow, with identical source.

Filling an array with braces

examples/ch08/braces.tiko

```
'' braces.tiko -- a brace list fills an array: one value per element,
'' and one level of braces per dimension.
```

```
dim as int32 primes( 0 to 4 ) = { 2, 3, 5, 7, 11 }
dim as int32 grid( 0 to 1, 0 to 2 ) = { { 1, 2, 3 }, { 4, 5, 6 } }

'' '...' as the upper bound: the list decides how many there are.
dim as int32 squares( 1 to ... ) = { 1, 4, 9, 16 }

print "primes:";
for i as int32 = lbound( primes ) to ubound( primes )
    print primes( i );
next
print

print "grid, row 1:";
for c as int32 = lbound( grid, 2 ) to ubound( grid, 2 )
    print grid( 1, c );
next
print

print "squares:"; ubound( squares ); " of them, the last";
print squares( ubound( squares ) )
```

examples/ch08/braces.out

```
primes: 2 3 5 7 11
grid, row 1: 4 5 6
squares: 4 of them, the last 16
```

A brace list gives the elements in index order, from `lbound` up. An array with more than one dimension takes one level of braces per dimension, in the order the last section described: the first inner list is row 0, so `{ { 1, 2, 3 }, { 4, 5, 6 } }` puts `4 5 6` in row 1.

`...` as the upper bound lets the list decide the size. `squares( 1 to ... )` has four elements because four values were given, so adding a fifth value cannot leave the bounds behind.

Braces are for arrays, and only at the declaration. A single value takes `=` and the value. A brace list on anything that is not an array is refused:

examples/ch08/bracescalar.tiko

```
'' bracescalar.tiko -- a brace list initializes an ARRAY. A single
'' value takes '=' and the value.

dim as int32 total = { 0 }          '' refused
print total
```

```
bracescalar.tiko(4,22): error TKO0845: 'total' is not an array, so it
cannot take a '{ ... }' initializer
     |
   4 | dim as int32 total = { 0 }          '' refused
     |                      ^
```

A brace list is not a value either, so it cannot be assigned later:

examples/ch08/bracelater.tiko

```
'' bracelater.tiko -- braces belong to the declaration. Filling an
'' array again is a loop, or one element at a time.

dim as int32 scores( 0 to 2 ) = { 70, 85, 90 }
scores = { 0, 0, 0 }                '' refused
print scores( 0 )
```

```
bracelater.tiko(5,10): error TKO0846: a '{ ... }' list can only initialize
an array, and nothing here is one
     |
   5 | scores = { 0, 0, 0 }                '' refused
     |          ^
```

To refill an array, loop over it, or assign the elements one at a time. A container from the standard library is the one other thing a brace list can fill; Chapter 17 shows how. fbc refuses both lines too, with its errors 63 and 73.

Passing an array to a procedure

examples/ch08/params.tiko

```
'' params.tiko -- an array argument carries its bounds with it.

sub FillWith( a( ) as int32, byval v as int32 )
    for i as int32 = lbound( a ) to ubound( a )
        a( i ) = v
    next
end sub

function SumOf( a( ) as int32 ) as int64
    dim as int64 total = 0
    for i as int32 = lbound( a ) to ubound( a )
        total += a( i )
    next
    return total
```

```
end function

'' A parameter's rank is written with commas, and nothing else.
sub Describe2D( m( , ) as int32 )
    print "  rows"; lbound( m, 1 ); " to"; ubound( m, 1 ); _
          ", columns"; lbound( m, 2 ); " to"; ubound( m, 2 )
end sub

dim as int32 a( 5 to 8 )
FillWith( a( ), 3 )
print "bounds inside were"; lbound( a ); " to"; ubound( a )
print "sum ="; SumOf( a( ) )

dim as int32 grid( 1 to 2, 0 to 3 )
print "Describe2D saw:"
Describe2D( grid( ) )
```

examples/ch08/params.out

```
bounds inside were 5 to 8
sum = 12
Describe2D saw:
  rows 1 to 2, columns 0 to 3
```

A parameter is written `a( ) as int32` and the argument is written `a( )` — the empty parentheses are required at both ends. An array is always passed by reference; there is no copy, and writes inside the procedure are writes to the caller's array.

The bounds travel with it. `FillWith` was given an array indexed 5 to 8 and saw exactly that, which is why a procedure that works from `lbound` and `ubound` works on any array of the right type without being told anything else.

A multi-dimensional parameter states its **rank** with commas and nothing more: `m( , ) as int32` takes any two-dimensional `int32` array. Rank is fixed at compile time; bounds are not.

Returning an array

A function that builds a table has to give it to its caller somehow, and the old answers are both awkward. The caller can create the array first and let the function fill it in, which means the caller must know the size before the function has decided it. Or the array can be wrapped in a container just to cross one `return`. tiko lets a function return the array itself: write `( )` after the result type.

examples/ch08/returning.tiko

```
'' returning.tiko -- a function can hand back a dynamic array.

'' A brace list: the array is sized to the list and filled.
function FirstPrimes( ) as int32( )
    return { 2, 3, 5, 7, 11 }
end function

'' A local array, built in a loop and moved out. Nothing is copied,
'' and the bounds the function chose travel with it.
function Squares( byval n as int32 ) as int32( )
    dim r( ) as int32
    redim r( 1 to n )
    for i as int32 = 1 to n
        r( i ) = i * i
    next
    return r
end function

'' Any element type works.
function Greetings( ) as ustring( )
    return { "hello", "bonjour", "hola" }
end function

sub ShowSum( a( ) as int32 )
    dim as int32 total = 0
    for i as int32 = lbound( a ) to ubound( a )
        total += a( i )
    next
    print "sum of " & lbound( a ) & " to " & ubound( a ) & " = " & total
end sub

'' var takes the result as it is: type, bounds and all.
var p = FirstPrimes( )
print "primes " & lbound( p ) & " to " & ubound( p ) & _
      ", p(4) = " & p( 4 )

var sq = Squares( 5 )
print "squares " & lbound( sq ) & " to " & ubound( sq ) & _
      ", sq(5) = " & sq( 5 )

'' A result can go straight to a parameter that takes an array.
ShowSum( Squares( 4 ) )

'' for each walks a result, and the array is freed afterwards.
```

```
for each g as ustring in Greetings( )
    print g
next

'' Assigning moves the new array in and frees the old one.
dim as int32 m( )
m = FirstPrimes( )
m = Squares( 3 )
print "m is now " & lbound( m ) & " to " & ubound( m )
```

examples/ch08/returning.out

```
primes 0 to 4, p(4) = 11
squares 1 to 5, sq(5) = 25
sum of 1 to 4 = 30
hello
bonjour
hola
m is now 1 to 3
```

The result is an ordinary dynamic array. It is what `dim a( )` makes, it is indexed the same way, and a parameter written `a( )` accepts it. It is not a container. Only an empty `( )` means an array: `as int32( 3 )` is refused (`TK00836`), where it used to be quietly read as a plain `int32`.

`return` takes three things:

- **A brace list.** The array is sized to the list, indexed from 0, and filled. A list inside a list is refused (`TK00839`); write it flat. Any element type works, and each element is built once.
- **A local array**, as `Squares` does. It is **moved**, not copied: the function's own variable is left empty and the caller receives the array it built, bounds included. `sq` runs from 1 to 5 because `Squares` said so.
- **A call that returns an array**, handed straight through.

`function = value` takes the same three and carries on afterwards, as it does for any result. A function that never sets one returns an empty array: `ubound` is −1 and a loop over it runs zero times.

The caller owns what comes back. `var p = FirstPrimes( )` takes the array itself. `m = Squares( 3 )` frees what `m` held and moves the new array in. `ShowSum( Squares( 4 ) )` and `for each` over a call use the array and free it when the statement ends, and a result nobody uses is freed there too. Two calls never share a buffer, so there is nothing to release by hand.

Only an array that belongs to the function can be moved out. A parameter, a `shared` or `static` array and a field all belong to somebody else: handing one back would leave its owner pointing at freed memory, and the other way to do it is a copy, which tiko does not make for you yet.

examples/ch08/returnparam.tiko

```
'' returnparam.tiko -- an array that is not the function's own cannot
'' be handed back: its owner would be left holding freed memory.

function Same( a( ) as int32 ) as int32( )
    return a
end function
```

```
returnparam.tiko(5,5): error TKO0840: 'a' is not a local array of this
procedure, so returning it would need a copy, which is not supported yet;
return a local array or a brace list
     |
   5 |     return a
     |     ^
```

To hand back a copy, build a local array element by element and return that.

The same care applies at the call. The name of a function that returns an array needs its parentheses, so `var b = FirstPrimes` is `TKO0841` rather than a call. An array is not a number, so `dim as int32 x = FirstPrimes( )` is `TKO0842`. And `dim a( ) as int32 = FirstPrimes( )` is `TKO0612`, as it is for any initialised dynamic array: `var` takes the result, `dim` does not.

> **STATUS IN THIS VERSION**
> Two limits remain. A function type cannot return an array (`as function( ) as int32( )` is `TKO0837`), and array results of properties and operators have not been tried.

Walking it

examples/ch08/walking.tiko

```
'' walking.tiko -- three ways across an array, and what each can do.

dim as int32 a( 1 to 4 ) = { 10, 20, 30, 40 }

print "by index:";
for i as int32 = lbound( a ) to ubound( a )
    print a( i );
```

```
next
print

print "for each:";
for each v as int32 in a
    print v;
next
print

'' for each hands you a COPY. Writing to it changes nothing.
for each v as int32 in a
    v = 0
next
print "after writing to the copy:";
for each v as int32 in a : print v; : next
print

'' for each byref hands you the element itself.
for each byref v as int32 in a
    v *= 2
next
print "after for each byref:";
for each v as int32 in a : print v; : next
print
```

examples/ch08/walking.out

```
by index: 10 20 30 40
for each: 10 20 30 40
after writing to the copy: 10 20 30 40
after for each byref: 20 40 60 80
```

Three ways across, and the difference between the last two matters:

- **By index**, `for i = lbound( a ) to ubound( a )`. You have the position as well as the value, which you need whenever the answer depends on where you are.
- **`for each v as T in a`** hands you a **copy** of each element. Writing to `v` changes nothing — the listing proves it.
- **`for each byref v as T in a`** hands you the element itself, so writes land in the array.

Use `for each` when you only need the values, and `byref` when you mean to modify them. The `byref` form also avoids copying each element, which matters when the elements are large.

`for each` will not walk a grid:

examples/ch08/foreach2d.tiko

```
'' foreach2d.tiko -- there is no single order to walk a grid in.
dim as int32 m( 0 to 1, 0 to 2 )

for each v as int32 in m        '' refused: which way, rows or columns?
    print v;
next
```

```
foreach2d.tiko(4,1): error TKO0608: 'for each' over a multi-dimensional
array has no defined order
     |
   4 | for each v as int32 in m        '' refused: which way, rows or
columns?
     | ^
```

The compiler could have picked row-major and said nothing. Refusing is the better answer: an iteration order over a grid is a decision the code should state, not one to inherit silently from the memory layout.

When the index is wrong

An index outside the bounds reads or writes memory that belongs to something else. Chapter 1 introduced `-exx`, which adds a check to every array access and stops the program with runtime error 6 rather than letting it continue with nonsense.

The check covers a dynamic array as well as a fixed one. Its bounds are not in the source, so the check reads them from the array's descriptor at run time, on every access, and a `redim` that shrinks the array moves the limit with it:

examples/ch08/exxdynamic.tiko

```
'' exxdynamic.tiko -- -exx checks a dynamic array too: its bounds are
'' read from the descriptor, at run time, on every access.
#cmdline "-exx"

dim as int32 a( )
redim a( 1 to 3 )
a( 3 ) = 30                     '' fine: 3 is inside 1 to 3

redim preserve a( 1 to 2 )      '' the limit moves with the array
a( 3 ) = 42                     '' stops here, with runtime error 6
print "not reached"
```

examples/ch08/exxdynamic.out

```

```

```
Aborting due to runtime error 6 (out of bounds array access) at line 10
of exxdynamic.tiko::(),
'A' accessed with invalid index = 3, must be between 1 and 2
```

The program stops with exit code 6 at the first bad write, before it can corrupt anything.

In a build without `-exx`, the check you write yourself is the one you can rely on:

`examples/ch08/guard.tiko`

```
'' guard.tiko -- checking an index yourself, which is always available.

function ValueAt( a( ) as int32, byval i as int32, _
                  byval fallback as int32 ) as int32
    if i < lbound( a ) orelse i > ubound( a ) then return fallback
    return a( i )
end function

dim as int32 a( )
redim a( 1 to 3 )
a( 1 ) = 10 : a( 2 ) = 20 : a( 3 ) = 30

print "ValueAt( 2 )  ="; ValueAt( a( ), 2, -1 )
print "ValueAt( 9 )  ="; ValueAt( a( ), 9, -1 )
print "ValueAt( 0 )  ="; ValueAt( a( ), 0, -1 )

redim preserve a( 1 to 2 )
print "after shrinking, ValueAt( 3 ) ="; ValueAt( a( ), 3, -1 )
```

`examples/ch08/guard.out`

```
ValueAt( 2 )  = 20
ValueAt( 9 )  =-1
ValueAt( 0 )  =-1
after shrinking, ValueAt( 3 ) =-1
```

Comparing against `lbound` and `ubound` costs two comparisons and works on every array, of every kind, in every build. Where an index comes from outside the program — a file, an argument, a calculation you are not certain of — that guard is not optional.

> **NOTE**
> **One array is not assigned to another.** `a = b` between arrays is refused — `TKO0930: a whole array cannot be assigned or swapped` — and so is `swap a,`

`b`, because the two may not even be the same size. To copy an array, loop over its elements.

Text stored in an array

examples/ch08/textarray.tiko

```
'' textarray.tiko -- an array of text that grows as rows arrive.

dim as ustring names( )

'' Each append makes room for one more and writes into the new slot.
dim as ustring incoming( 1 to 3 ) = { "Ada", "Grace", "Barbara" }

for i as int32 = lbound( incoming ) to ubound( incoming )
    redim preserve names( 0 to ubound( names ) + 1 )
    names( ubound( names ) ) = incoming( i )
next

'' Take the count into a variable of its own before printing it.
dim as int32 count = ubound( names ) - lbound( names ) + 1
print "count ="; count

for i as int32 = lbound( names ) to ubound( names )
    print i; ": " & names( i )
next
```

examples/ch08/textarray.out

```
count = 3
 0: Ada
 1: Grace
 2: Barbara
```

Growing an array one element at a time — `redim preserve` to `ubound + 1`, then write the new slot — is the pattern for building a list whose length is not known in advance. It is correct, and it is also the slow way to do it: each `redim preserve` may copy the whole array, so appending n items can cost n^2 work.

For a handful of rows that does not matter. For a file of them it does, and the answer is either to grow in chunks — double the size when it fills — or to use the container that already does exactly that, which is the next section.

The descriptor

examples/ch08/descriptor.tiko

```
'' descriptor.tiko -- what the compiler keeps alongside the elements.
#include once "fbc-int/array.bi"

dim as int32 a( 1 to 4 )
dim as int32 grid( 0 to 1, 0 to 2 )
dim as int32 empty_( )

print "a( 1 to 4 )       len ="; FB.ArrayLen( a( ) ); _
      " bytes ="; FB.ArraySize( a( ) )
print "grid( 0..1, 0..2 ) len ="; FB.ArrayLen( grid( ) ); _
      " bytes ="; FB.ArraySize( grid( ) )
print "an empty array     len ="; FB.ArrayLen( empty_( ) )

dim as FBC.FBARRAY ptr d = FBC.ArrayDescriptorPtr( a( ) )
print "descriptor: dimensions ="; d->dimensions; _
      " element_len ="; d->element_len
print "            dimTb( 0 ) ="; d->dimTb( 0 ).lbound; " to"; _
      d->dimTb( 0 ).ubound
```

examples/ch08/descriptor.out

```
a( 1 to 4 )       len =4 bytes =16
grid( 0..1, 0..2 ) len =6 bytes =24
an empty array     len =0
descriptor: dimensions =1 element_len =4
            dimTb( 0 ) = 1 to 4
```

An array variable is a **descriptor**: a pointer to the elements plus the bounds, the element size and the number of dimensions. That is what travels when an array is passed, and what `lbound` reads.

`fbc-int/array.bi` exposes it. `FB.ArrayLen` is the total element count across all dimensions and `FB.ArraySize` the total bytes; `FBC.ArrayDescriptorPtr` hands over the record itself. You will not often need these, and when you do — writing a procedure that takes an array of any type, checking what a `redim` actually allocated — nothing else will do.

When to reach for something else

An array is the right answer when the index is meaningful and the size is known, or grows rarely. It is the wrong answer when you find yourself writing the bookkeeping around it: a separate count of how many slots are really in use, a doubling strategy, a search to find where something is.

That bookkeeping is what `containers.Array( of T )` already contains, along with `Map`, `Set` and the rest of the containers in Chapter 24. The rule is the ordinary one: use the raw array while it is simpler, and stop as soon as it is not.

What it costs

- **Indexing is an address calculation**, not a search. Cost does not depend on which element.
- **`redim preserve` may copy everything.** Growing one at a time is quadratic; grow in chunks.
- **Row-major order is a performance fact**, not trivia — put the last subscript in the inner loop.
- **`-exx` costs a comparison per access.** It is worth it while you are writing the program; measure before you decide it is not worth it afterwards.
- **`for each byref` avoids copying each element**, which is free to write and matters for large ones.
- **Returning an array moves it.** The cost is one descriptor, a fixed few hundred bytes, whatever the length of the array; the elements are never copied.

Summary

- An array carries **bounds**, not a length: `lbound` and `ubound` are the first and last index.
- **`a( 10 )` is eleven elements**, indexed 0 to 10. Write both bounds.
- Elements are zero-initialised; an initialiser in braces fills them in order, one level of braces per dimension, and `1 to ...` lets the list set the size.
- **Braces initialize arrays and nothing else**: not a single value (`TKO0845`), and not an assignment after the declaration (`TKO0846`).
- `dim a( )` is empty, with `ubound` below `lbound`, so loops over it run zero times.
- **`redim` discards the contents; `redim preserve` keeps them.** `erase` frees a dynamic array and zeroes a fixed one.
- Dimensions are comma-separated, `ubound( m, 2 )` names one, and **the last subscript varies fastest**.
- Arrays pass by reference as `a( )`, carrying their bounds; a rank-2 parameter is written `m( , )`.
- A function returns an array with `as int32( )`. `return` takes a brace list, a **local** array (moved, never copied) or another call, and the caller owns the result: `var` takes it, `m = f( )` replaces what `m` held. A parameter or `shared` array cannot be returned (`TKO0840`).
- `for each` gives copies, `for each byref` gives the elements, and neither will walk a multi-dimensional array (`TKO0608`).

- **`-exx` checks every subscript, fixed or dynamic**, against the bounds the array has at that moment. Without it, compare against `lbound` and `ubound` yourself when the index is not obviously safe.
- **An array is not assigned or swapped whole** (`TKO0930`); copy it element by element.
- The array variable is a descriptor; `fbc-int/array.bi` exposes `FB.ArrayLen`, `FB.ArraySize` and the record itself.
- **Stop using a raw array when you start writing bookkeeping around it** — a live count, a doubling strategy, a search. `containers.Array( of T )` and the containers in Chapter 24 already contain it, and growing one element at a time with `redim preserve` is quadratic.

Exercises

1. Write `Reverse`, which reverses an array in place and works whatever its bounds — test it on arrays indexed from 1, from 0 and from −2, and on an empty one. Use no literal 0 and no literal length. {solution="ex08-01″}
2. Transpose a two-dimensional array into a second array whose dimensions are the original's, swapped. Where do the new bounds come from? {solution="ex08-02″}
3. Predict how many elements `dim as int32 a( 5 )` has before running anything, then print `lbound`, `ubound` and the count. Rewrite the declaration so the answer is obvious.
4. Take the `redim` line in `dynamic.tiko`, add `preserve`, and say in one sentence what each version is for.
5. Build a 200 × 200 array and sum it twice: once with the last subscript innermost, once with the loops swapped. Both give the same total. Which do you expect to be faster, and why?
6. Write `Evens( byval n as int32 )`, returning the first `n` even numbers as an array indexed from 1, and `Doubled( a( ) as int32 )`, returning a new array of the same bounds with every element doubled. Take `var e = Evens( 4 )` and print `Doubled( e( ) )` with `for each`. Why can `Doubled` not simply `return a`?

Strings and Text

Text is where a program meets people, and it is where most of the assumptions a programmer carries turn out to be wrong. A string is not a sequence of characters. A character is not a byte. The length of a word depends on what you are counting.

This chapter is long because that is the honest size of the subject, and because getting it wrong is silent: text that looks right on your machine arrives as nonsense on someone else's.

The short version, which the rest of the chapter earns: **use `ustring`, and keep bytes and text apart in your head.**

Five ways to hold text

examples/ch09/types.tiko

```
'' types.tiko -- five ways to hold text, and what len counts in each.

dim as string       a = "hello"     '' bytes, length-counted, grows
dim as ustring      b = "hello"     '' UTF-16 code units, grows
dim as zstring * 16 c = "hello"    '' bytes, NUL-terminated, fixed room
dim as wstring * 16 d = "hello"     '' wide characters, fixed room
dim as string * 8   e = "hello"     '' bytes, fixed length, PADDED

print "string       len ="; len( a )
print "ustring      len ="; len( b )
print "zstring * 16 len ="; len( c )
print "wstring * 16 len ="; len( d )
print "string * 8   len ="; len( e ); "   <- not 5"
print "string * 8   = ["; e; "]"

'' A string variable is a descriptor, not the characters themselves.
print "sizeof( string )  ="; sizeof( string )
print "sizeof( ustring ) ="; sizeof( ustring )
```

examples/ch09/types.out

```
string       len = 5
ustring      len = 5
zstring * 16 len = 5
wstring * 16 len = 5
string * 8   len = 8   <- not 5
string * 8   = [hello   ]
sizeof( string )  = 24
sizeof( ustring ) = 24
```

Type	Holds	Sized
`ustring`	UTF-16 code units	grows
`string`	bytes	grows
`zstring * n`	bytes, NUL-terminated	fixed room
`wstring * n`	wide characters	fixed room
`string * n`	bytes, **padded** to `n`	fixed length

`ustring` is the one to reach for. It is the type this book uses everywhere, it holds text rather than bytes, and `len` on it is a single read rather than a walk.

`string` is the byte type. It is right for file contents, network payloads and anything you do not intend to interpret — and wrong for anything a person will read, unless you also know its encoding.

`zstring` and `wstring` exist for talking to C, which Chapter 29 covers. They are the shapes a C library expects: a pointer to bytes ending in a NUL, or the platform's wide character.

> **PITFALL**
> **A `string * n` is always exactly `n` characters.** Assigning `"hello"` to a `string * 8` gives you `"hello   "` — padded with spaces — and `len` reports 8, not 5.
>
> That is the point of the type, and it is a trap when one arrives from a record layout or a fixed-width file and gets compared against text that was not padded. `"hello   " = "hello"` is false. Wrap it in `rtrim( )` at the boundary, once, rather than at every comparison.

> **ON WINDOWS AND LINUX**
> A `wstring` character is **2 bytes on Windows and 4 bytes on Linux**, because it follows the platform's `wchar_t`. `len` counts characters on both, so code that uses `len` is portable and code that assumes a byte width is not.
>
> `ustring` is UTF-16 on every target, which is one reason to prefer it: its storage does not change underneath you when the target does.

Text that spans lines

A line ends a statement in tiko, so an ordinary string literal ends where its line does. A query, a usage message or a block of JSON is several lines of text, and writing it as one literal per line, glued with `&` and `chr( 10 )`, buries the text under the punctuation. A **block string** holds it the way it reads:

`examples/ch09/blockstrings.tiko`

```
dim as ustring query = """
    select id, name
      from users
     where name = "Ada"
    """
print query
print len( query )

dim as ustring poem = """
  roses
    are
  red

  """
print poem;
print "---"

dim as ustring shifted = """
    indented
"""
print "["; shifted; "]"

dim as ustring row = !"""
    name\tvalue
    one long line, \
    continued
    """
print row
print instr( row, chr( 9 ) )
```

examples/ch09/blockstrings.out

```
select id, name
  from users
 where name = "Ada"
 48
roses
  are
red
---
[    indented]
name  value
one long line, continued
 5
```

""" at the end of a line opens one, and the first line that starts with """ closes it. Only spaces may follow the opening `"""`. Anything after the closing one is ordinary code, so a block string can sit inside an expression or an argument list like any other literal.

The closing line's indentation is the margin. It is removed from the front of every line, so the text indents with the code around it, while indentation that belongs to the text — the two extra spaces before `from` — is kept. A blank line is always allowed. Any other line that starts left of the margin is an error:

examples/ch09/blockmargin.tiko

```
'' A line that starts left of the closing line is an error.
dim as ustring note = """
    first
  second
    """
print note
```

```
blockmargin.tiko(4,1): error TKO0208: block string line does not start
with the closing line's indentation
     |
   4 |   second
     | ^~~~~~~~
```

The line breaks at the two ends are not part of the text. `query` is 48 characters and does not end in a line feed. To end with one, leave an empty line before the closing line, as `poem` does — which is why `print poem;` still put `---` on a line of its own.

Inside the block, nothing needs escaping. A `"` is an ordinary character, so `"Ada"` is written as it reads. Every line break is a single line feed, `chr( 10 )`, whatever line endings the file was saved with, so the same source gives the same string on every machine.

`!"""` reads escapes, exactly as `!"…"` does: `\t` became the tab `instr` found at position 5. A backslash at the very end of a line joins it to the next with no line break, for a long line you want to wrap in the source.

> **PITFALL**
> **The closing line sets the margin, not the first line.** `shifted` closes at the left edge, so nothing is removed: the four spaces that were only there to indent the code are now part of the text. It compiles, and the output above shows them,

inside the brackets. When a block string comes out shifted right, look at where its closing """ sits.

The operations that come with the language

examples/ch09/intrinsics.tiko

```
'' intrinsics.tiko -- the operations that come with the language.

dim as ustring s = "The quick brown fox"

print "len            ="; len( s )
print "left( s, 3 )   = ["; left( s, 3 ); "]"
print "right( s, 3 )  = ["; right( s, 3 ); "]"
print "mid( s, 5, 5 ) = ["; mid( s, 5, 5 ); "]"
print "mid( s, 11 )   = ["; mid( s, 11 ); "]"

print "instr( s, ""brown"" )  ="; instr( s, "brown" )
print "instr( 5, s, ""o"" )   ="; instr( 5, s, "o" )
print "instrrev( s, ""o"" )   ="; instrrev( s, "o" )
print "instr( s, ""zebra"" )  ="; instr( s, "zebra" ); "   <- not found"

print "ucase = ["; ucase( s ); "]"
print "lcase = ["; lcase( s ); "]"

dim as ustring padded = "   spaces   "
print "trim  = ["; trim( padded ); "]"
print "ltrim = ["; ltrim( padded ); "]"
print "rtrim = ["; rtrim( padded ); "]"
print "trim any = ["; trim( "xxcorexx", any "x" ); "]"

print "space( 3 )        = ["; space( 3 ); "]"
print "string( 5, ""-"" )   = ["; string( 5, "-" ); "]"
print "chr( 65 )         = ["; chr( 65 ); "]"
print "asc( ""A"" )        ="; asc( "A" )
```

examples/ch09/intrinsics.out

```
len            = 19
left( s, 3 )   = [The]
right( s, 3 )  = [fox]
mid( s, 5, 5 ) = [quick]
mid( s, 11 )   = [brown fox]
instr( s, "brown" )  = 11
instr( 5, s, "o" )   = 13
```

```
instrrev( s, "o" )   = 18
instr( s, "zebra" )  = 0   <- not found
ucase = [THE QUICK BROWN FOX]
lcase = [the quick brown fox]
trim  = [spaces]
ltrim = [spaces   ]
rtrim = [   spaces]
trim any = [core]
space( 3 )        = [   ]
string( 5, "-" )   = [-----]
chr( 65 )         = [A]
asc( "A" )        =65
```

Two conventions to absorb, because everything else follows from them.

Positions start at 1. `mid( s, 5, 5 )` means five characters starting at the fifth. This is not an accident to be worked around — it is consistent across every intrinsic here, and mixing it with zero-based thinking is the single commonest source of off-by-one bugs in string code.

Not found is 0. `instr` returns 0 when the needle is absent, which is never a valid position, so `if instr( s, x ) > 0` is the test. There is no separate "not found" value to remember.

`mid` also works on the left of an assignment, where it overwrites in place:

examples/ch09/midstatement.tiko

```
'' midstatement.tiko -- mid on the left of an assignment overwrites
'' in place, without rebuilding the string.

dim as string s = "abcdef"
mid( s, 2, 3 ) = "XYZ"
print "mid( s, 2, 3 ) = ""XYZ"" -> ["; s; "]"

dim as string t = "abcdef"
mid( t, 2 ) = "XY"              '' no length: as much as will fit
print "mid( t, 2 )    = ""XY""   -> ["; t; "]"

'' It never changes the length. A replacement longer than the room
'' available is truncated rather than growing the string.
dim as string u = "abc"
mid( u, 2 ) = "ZZZZZZ"
print "overlong replacement   -> ["; u; "]  len ="; len( u )
```

examples/ch09/midstatement.out

```
mid( s, 2, 3 ) = "XYZ" -> [aXYZef]
mid( t, 2 )    = "XY"  -> [aXYdef]
overlong replacement   -> [aZZ]  len = 3
```

The `mid` **statement** never changes the length. A replacement longer than the room available is truncated, as the third case shows. That is what makes it cheap: nothing is reallocated. It works the same on a `ustring`, counting UTF-16 units.

Numbers and text

examples/ch09/numbers.tiko

```
'' numbers.tiko -- text to numbers and back.

print "str( 42 )     = ["; str( 42 ); "]"
print "str( -7 )     = ["; str( -7 ); "]"
print "str( 3.5 )    = ["; str( 3.5 ); "]"
print "print puts a space where the sign would be:"; 42

print "val( ""42"" )       ="; val( "42" )
print "val( ""42abc"" )    ="; val( "42abc" ); "  <- stops at 'a'"
print "val( ""abc"" )      ="; val( "abc" ); "  <- and a word gives 0"
print "valint( ""42"" )    ="; valint( "42" )
print "vallng( ""4200000000"" ) ="; vallng( "4200000000" )
print "valuint( ""&hFF"" ) ="; valuint( "&hFF" )

print "hex( 255 )    = ["; hex( 255 ); "]"
print "hex( 255, 4 ) = ["; hex( 255, 4 ); "]"
print "oct( 8 )      = ["; oct( 8 ); "]"
print "bin( 5 )      = ["; bin( 5 ); "]"

print using "pi to two places: ###.##"; 3.14159
print using "a whole number:   ###"; 42
```

examples/ch09/numbers.out

```
str( 42 )     = [42]
str( -7 )     = [-7]
str( 3.5 )    = [3.5]
print puts a space where the sign would be: 42
val( "42" )       = 42
val( "42abc" )    = 42  <- stops at 'a'
val( "abc" )      = 0  <- and a word gives 0
valint( "42" )    = 42
vallng( "4200000000" ) = 4200000000
```

```
valuint( "&hFF" ) =255
hex( 255 )    = [FF]
hex( 255, 4 ) = [00FF]
oct( 8 )      = [10]
bin( 5 )      = [101]
pi to two places:   3.14
a whole number:    42
```

`str` converts a number to text with no padding. `print` is different — it reserves a column for the sign, which is why printed positive numbers appear with a leading space throughout this book.

`hex`, `oct` and `bin` render a value in another base, and a second argument sets the width: `hex( 255, 4 )` is `00FF`.

`print using` lays out a line against a picture: `###.##` is a number with two decimals, `&` is a string. It is the quickest way to get a column to line up. For a line that mixes widths, bases and decimals, `strings.Format`, later in this chapter, says it in one place.

> **PITFALL**
> **`val( )` cannot fail.** It reads as far as it can and returns what it got, so `val( "42abc" )` is 42, and `val( "abc" )` is 0 — indistinguishable from the text `"0"`.
>
> That is fine for input you produced and dangerous for input you did not. When a program has to know whether the text *was* a number, `val` alone cannot tell it. `strings.IsNumeric( )` answers that question, and Chapter 18's `Outcome` type carries the answer and the failure together, which is the shape to reach for once parsing can go wrong in more than one way.

What `len` is actually counting

Here is the idea the rest of the chapter depends on. A string has three different lengths, and which one you get depends on the type holding it.

`examples/ch09/codeunits.tiko`

```
'' codeunits.tiko -- what len actually counts, in each type.
''
'' The text is built from explicit bytes so that nothing here depends
'' on how this file was saved: these ARE the UTF-8 bytes for the two
'' strings, listed one by one.

'' "cafe" with an e-acute: 63 61 66 C3 A9
dim as string cafeBytes = chr( &h63, &h61, &h66, &hC3, &hA9 )
```

```
'' "a", U+1F600 GRINNING FACE, "b": 61 F0 9F 98 80 62
dim as string faceBytes = chr( &h61, &hF0, &h9F, &h98, &h80, &h62 )

dim as ustring cafe = cafeBytes
dim as ustring face = faceBytes

print "cafe: as string (UTF-8 bytes) len ="; len( cafeBytes )
print "      as ustring (code units) len ="; len( cafe )
print "      code units:";
for i as int32 = 1 to len( cafe ) : print asc( cafe, i ); : next
print

print "face: as string (UTF-8 bytes) len ="; len( faceBytes )
print "      as ustring (code units) len ="; len( face )
print "      code units:";
for i as int32 = 1 to len( face ) : print asc( face, i ); : next
print
print "      -- 55357 and 56832 are ONE character, stored as a pair"

'' Converting back is lossless.
dim as string roundTrip = face
print "round trip back to string: len ="; len( roundTrip ); _
      "  equal = "; cbool( roundTrip = faceBytes )
```

examples/ch09/codeunits.out

```
cafe: as string (UTF-8 bytes) len = 5
      as ustring (code units) len = 4
      code units: 99 97 102 233
face: as string (UTF-8 bytes) len = 6
      as ustring (code units) len = 4
      code units: 97 55357 56832 98
      -- 55357 and 56832 are ONE character, stored as a pair
round trip back to string: len = 6  equal = true
```

The listing builds its text from explicit byte values, so nothing depends on how the file was saved — those really are the UTF-8 bytes.

Read the two cases:

- **`café`** is 4 characters. As UTF-8 bytes it is **5** — `é` needs two. As a `ustring` it is **4** code units. Here characters and code units agree.
- **`a` + U+1F600 + `b`** is 3 characters. As UTF-8 bytes it is **6**. As a `ustring` it is **4** code units — because U+1F600 is outside the Basic Multilingual Plane and UTF-16 stores it as a *surrogate pair*, two units that mean one character.

PITFALL

`len` on a `ustring` counts code units, not characters, and `mid` indexes them.

For the Basic Multilingual Plane — which is every European, Cyrillic, Greek, Hebrew, Arabic, CJK and Hangul character — those are the same number, which is why the bug hides. Emoji, historic scripts, many mathematical symbols and some rarer CJK are outside it, and there one character is two units.

So `mid( face, 2, 1 )` hands back **half a surrogate pair**: a value that is not a character and renders as the replacement character U+FFFD. Splitting a `ustring` at an arbitrary position can cut a character in two.

When the positions come from searching — `instr`, `Split`, a delimiter you put there yourself — you are safe, because those land on boundaries. When they come from arithmetic, such as "the first 20 characters", they are not.

Conversion between `string` and `ustring` is **UTF-8 in both directions, on every target**, and it is lossless: the round trip in the listing compares equal.

That is the rule worth memorising. A `ustring` assigned to a `string` is encoded as UTF-8; a `string` assigned to a `ustring` is decoded as UTF-8. There is no locale involved and no target dependence.

Bytes that are not UTF-8

examples/ch09/ansibytes.tiko

```
'' ansibytes.tiko -- a byte string is not automatically UTF-8.
''
'' Both of these hold "cafe" with an e-acute. One was saved by a
'' program that wrote Latin-1; the other is UTF-8. Converting each to
'' ustring shows what the difference costs.

dim as string latin1 = chr( &h63, &h61, &h66, &hE9 )
dim as string utf8   = chr( &h63, &h61, &h66, &hC3, &hA9 )

dim as ustring fromLatin1 = latin1
dim as ustring fromUtf8   = utf8

print "Latin-1 bytes:"; len( latin1 ); " -> ustring units:"; _
      len( fromLatin1 )
print "  code units:";
for i as int32 = 1 to len( fromLatin1 )
    print asc( fromLatin1, i );
next
print
```

```
print "UTF-8 bytes:  "; len( utf8 ); " -> ustring units:"; _
      len( fromUtf8 )
print "  code units:";
for i as int32 = 1 to len( fromUtf8 ) : print asc( fromUtf8, i ); : next
print
print
print "65533 is U+FFFD, the replacement character: the byte was not"
print "valid UTF-8, so the text it stood for is gone."
```

examples/ch09/ansibytes.out

```
Latin-1 bytes: 4 -> ustring units: 4
  code units: 99 97 102 65533
UTF-8 bytes:   5 -> ustring units: 4
  code units: 99 97 102 233

65533 is U+FFFD, the replacement character: the byte was not
valid UTF-8, so the text it stood for is gone.
```

Both of those `string`s hold the word *café*. One was written by something that used Latin-1, where `é` is the single byte `E9`. The other is UTF-8, where it is `C3 A9`.

Converting each to `ustring` decodes as UTF-8 — because that is the only rule — so the Latin-1 byte is not valid input and becomes **65533**, U+FFFD, the replacement character. The text it stood for is gone, and nothing was reported.

> **PITFALL**
> **A `string` does not know its encoding, and the conversion does not ask.**
>
> Bytes from a file written years ago, from a Windows API that returns the ANSI code page, or from a database column configured for Latin-1, are not UTF-8 — and assigning them to a `ustring` destroys every non-ASCII character in them, silently and unrecoverably.
>
> Convert at the boundary, with knowledge of what the bytes are, and keep everything inside the program as `ustring`. `utf_conv.bi` has the routines for encodings other than UTF-8. The one thing never to do is assume.

The standard library

examples/ch09/stdstrings.tiko

```
'' stdstrings.tiko -- what the standard library adds.
#import "std/strings"
```

```
dim as ustring s = "alpha,beta,gamma"

'' Split hands back an Array( of ustring ) -- the container from
'' Chapter 24. Here it is only asked for its count and its elements.
var parts = strings.Split( s, "," )
print "Split gave"; parts.Count( ); " parts:";
for i as int32 = 0 to parts.Count( ) - 1
    print " ["; parts[ i ]; "]";
next
print

print "Join       = ["; strings.Join( parts, " | " ); "]"
print "Replace    = ["; strings.Replace( s, "beta", "BETA" ); "]"
print "Between    = ["; strings.Between( "name(value)", "(", ")" ); "]"
print "StartsWith = "; strings.StartsWith( s, "alpha" )
print "EndsWith   = "; strings.EndsWith( s, "gamma" )
print "Contains   = "; strings.Contains( s, "bet" )
print "IsNumeric  = "; strings.IsNumeric( "42" ); _
      strings.IsNumeric( "4a" )
print "PadLeft    = ["; strings.PadLeft( "7", 4, "0" ); "]"
print "PadRight   = ["; strings.PadRight( "ab", 5, "." ); "]"
print "PadCenter  = ["; strings.PadCenter( "ab", 6, "." ); "]"
print "Repeat     = ["; strings.Repeat( 3, "ab" ); "]"
print "Reverse    = ["; strings.Reverse( "abc" ); "]"
print "Wrap       = ["; strings.Wrap( "hi", "<", ">" ); "]"
print "Escape     = ["; strings.Escape( !"a\tb" ); "]"
```

examples/ch09/stdstrings.out

```
Split gave 3 parts: [alpha] [beta] [gamma]
Join       = [alpha | beta | gamma]
Replace    = [alpha,BETA,gamma]
Between    = [value]
StartsWith = true
EndsWith   = true
Contains   = true
IsNumeric  = truefalse
PadLeft    = [0007]
PadRight   = [ab...]
PadCenter  = [..ab..]
Repeat     = [abababab]
Reverse    = [cba]
Wrap       = [<hi>]
Escape     = [a\tb]
```

`#import "std/strings"` brings in around forty-five operations that the language itself does not have. The ones above are the everyday set; the module also carries `Extract`, `Remain`, `Tally`, `InsertAt`, `DeleteAt`, `Shrink`, `VerifySet`, the `*Chars` family that works on sets of characters rather than substrings, and `Unescape`/`Unwrap` to undo their opposites.

`Split` hands back an `Array( of ustring )` — the container Chapter 24 covers. Here it is only asked for `.Count( )` and `[ i ]`, which is all you need to walk it.

`Join( Split( s, d ), d )` gives back `s`, for every `s` and every non-empty `d`. That round-trip property is worth knowing, because it tells you what `Split` does with the awkward cases: `"a,,c"` is three fields with an empty one in the middle, and `"a,"` is two.

Part of a string, without a copy

`mid`, `left`, `right` and `Split` all hand back new strings. Each one allocates and copies, and a program that takes a line apart word by word pays that once per word. Most of those words are looked at once and thrown away. A **slice** names part of a string where it already is:

examples/ch09/slices.tiko

```
'' slices.tiko -- naming part of a string without copying it.

dim as ustring s = "hello, world"

dim as uslice word = s[7 to 11]    '' units 7 to 11, both included
print "word           = ["; word; "]"
print "len( word )    = "; str( len( word ) )
print "word[0]        = "; str( word[0] )   '' a code unit
print "word[1 to 3]   = ["; word[1 to 3]; "]"
print "s[0 to 4]      = ["; s[0 to 4]; "]"
print "mid( s, 1, 5 ) = ["; mid( s, 1, 5 ); "]"

'' A slice compares, joins and prints like the text it names.
print "word = ""world"" is "; cbool( word = "world" )
print "<" & word & ">"

'' Assigning it to a ustring makes an owned copy.
dim as ustring copy = word
copy &= "!"
print "copy = ["; copy; "]  s = ["; s; "]"

'' A slice of a mutable string writes straight into it.
word[0] = asc( "W" )
```

```
print "s is now ["; s; "]"

'' An empty slice: hi is one less than lo.
print "s[3 to 2] has "; str( len( s[3 to 2] ) ); " units"
```

examples/ch09/slices.out

```
word           = [world]
len( word )    = 5
word[0]        = 119
word[1 to 3]   = [orl]
s[0 to 4]      = [hello]
mid( s, 1, 5 ) = [hello]
word = "world" is true
<world>
copy = [world!]  s = [hello, world]
s is now [hello, World]
s[3 to 2] has 0 units
```

`s[7 to 11]` is the units from 7 to 11, **both ends included, counting from 0** — the same counting as `s[i]`, which reads one unit. That is the one place in this chapter where positions start at 0, and it is why the listing prints `mid( s, 1, 5 )` beside `s[0 to 4]`: the same five characters, one counted from 1 and copied, the other counted from 0 and not.

There are three slice types, one for each width of text:

Type	Slices	Unit
`uslice`	`ustring`	a UTF-16 code unit
`slice`	`string`, `zstring`, a `zstring ptr`	a byte
`wslice`	`wstring`, a `wstring ptr`	the platform's wide character

A slice is small — a pointer, a length and one more word the runtime checks use — and it is never a copy. It prints, compares and joins with `&` as the text it names, and `len` on it is a single read. Indexing it gives a code unit, a number, as indexing a `ustring` does. Slicing a slice gives a smaller view of the same text.

A slice is as writable as the string it came from. `word[0] = asc( "W" )` changed `s`, because `word` *is* part of `s`. A slice of a `const` string, a literal or a temporary is a `const uslice`, and writing through it is refused. Going the other way — from `const` back to writable — is error `TKO1052`.

To keep the text, assign the slice to a `ustring`. That makes the copy, once, at the point where you decide the text is worth keeping.

`s[3 to 2]` is empty: a slice whose end is one before its start has no units. So `s[0 to len( s ) - 1]` is the whole of `s` even when `s` is empty. Anything else outside the string is runtime error 6 — checked whenever a slice is made, in every build — and a range the compiler can see is wrong, such as `5 to 2` or past the end of a `string * 4`, is `TKO1051` before the program runs.

Slices in the standard library

`std/strings` has a set of functions that take and return slices, so a whole parse can run without a single copy:

examples/ch09/slicestd.tiko

```
'' slicestd.tiko -- reading settings with no copies at all.
#import "std/strings"

dim as ustring config = !"  name = tiko \n\n version=0.1 \n  bad line"

for each line_ as const uslice in _
        strings.SplitSlices( config[0 to len( config ) - 1], !"\n" )
    dim as const uslice t = strings.TrimSpace( line_ )
    if len( t ) = 0 then continue for

    dim as int32 eq = strings.Find( t, "=" )
    if eq < 0 then
        print "skipped: ["; t; "]"
        continue for
    end if

    dim as const uslice key = strings.TrimSpace( t[0 to eq - 1] )
    dim as const uslice value = _
        strings.TrimSpace( t[eq + 1 to len( t ) - 1] )
    print "["; key; "] -> ["; value; "]"
next
```

examples/ch09/slicestd.out

```
[name] -> [tiko]
[version] -> [0.1]
skipped: [bad line]
```

`TrimSpace`, `TrimStart` and `TrimEnd` return a slice of their argument with the spaces, tabs, carriage returns and line feeds removed from the ends. `Find` returns the position of the first match, **counting from 0, or −1 when there is none** — slice counting,

not `instr`'s. `Equals`, `StartsWith` and `EndsWith` compare, with an optional `true` to ignore the case of A to Z.

`SplitSlices` is `Split` without the array. It hands the parts to a `for each` one at a time, each a slice of the original. An empty part is still a part, so the blank line in `config` arrives as an empty slice — that is what the `len( t ) = 0` test is for.

Every one of these takes a `const` slice, and any string can be passed for one. A string of the same width is viewed where it is; any other is converted once, for the call.

> **TIP**
> `SplitSlices( config[0 to len( config ) - 1], !"\n" )` passes a `uslice`, not the `ustring` itself. A `ustring` and a plain literal together leave the compiler two equally good choices — view the first and convert the second, or the other way round — and it refuses to guess (`TKO0660`). Slicing the whole string first settles it.

A slice cannot outlive its text

A slice points into memory it does not own. If that memory is freed or moved while the slice is still in use, the slice reads whatever is there now. The compiler keeps track of where every slice came from and refuses each way that could happen:

examples/ch09/sliceescape.tiko

```
'' sliceescape.tiko -- a slice of a local cannot leave its function.

function FirstWord( byval s as ustring ) as const uslice
    return s[0 to 2]      '' error: s is this call's own copy
end function

print FirstWord( "the end" )
```

```
sliceescape.tiko(4,13): error TKO1059: cannot return a slice of local 's'
     |
   4 |     return s[0 to 2]      '' error: s is this call's own copy
     |             ^
```

`s` is passed `byval`, so it belongs to the call and is gone when the call returns. Make it `byref s as const ustring` and the function compiles: the text then belongs to the caller, which still has it after the call.

The other rules follow from the same idea. A slice may be a local variable, a parameter or a result. It may not be `shared` or `static`, an array element, an element of a container, or allocated with `new` (`TKO1054`), because any of those can outlive the text with nothing to say when. A slice of a string declared in an inner block cannot be kept in a variable declared outside it (`TKO1057`). A slice of a temporary — `(a & b)`

`[0 to 3]` — can be printed or passed in the same statement, and cannot be kept at all (`TKO1058`).

The last rule is about the text changing underneath the slice:

examples/ch09/slicelock.tiko

```
'' slicelock.tiko -- a string cannot grow while a slice reads it.

dim as ustring s = "hello"
dim as uslice first3 = s[0 to 2]

s &= ", world"        '' error: may move the text first3 points into
print first3
```

```
slicelock.tiko(6,1): error TKO1062: 's' cannot be changed while slice
'first3' borrows from it (last used at line 7)
     |
   6 | s &= ", world"        '' error: may move the text first3 points
into
     | ^
```

Appending to a `ustring` may move its text to a bigger block of memory, which would leave `first3` pointing at the old one. So from the moment a slice is made until the last line that reads it, the string it came from cannot be assigned, appended to, swapped, read into, or passed `byref` to a procedure that might do any of those. Move the `print` above the `&=` and the program compiles: the slice is finished with before the string grows. Writing single units in place — `s[i] = x`, the `mid` statement — never moves the text, and stays allowed.

Types that hold a slice

A field of an ordinary type can outlive anything, so a slice field is refused there (`TKO1055`). A type declared `as view` may hold one, and is then held to the same rules as a slice:

examples/ch09/lexer.tiko

```
'' lexer.tiko -- a type that holds a slice is a view.

type Lexer as view
    src as const uslice      '' the text being read, not a copy
    pos as int32
end type

'' The next run of non-blank units, or an empty slice at the end.
function NextWord( byref lx as Lexer ) as const uslice
```

```
    while lx.pos < len( lx.src ) andalso lx.src[lx.pos] = 32
        lx.pos += 1
    wend
    dim as int32 start = lx.pos
    while lx.pos < len( lx.src ) andalso lx.src[lx.pos] <> 32
        lx.pos += 1
    wend
    return lx.src[start to lx.pos - 1]
end function

dim as ustring text = "  let total = price * 3  "
dim as Lexer lx
lx.src = text[0 to len( text ) - 1]

do
    dim as const uslice w = NextWord( lx )
    if len( w ) = 0 then exit do
    print "<"; w; ">";
loop
print
```

examples/ch09/lexer.out

```
<let><total><=><price><*><3>
```

`Lexer` walks the text without copying any of it, and `NextWord` returns a slice of the text the lexer holds — which is the caller's, so it may. A `Lexer` cannot be `shared`, put in an array or stored in an ordinary type, exactly as a slice cannot, and a type that holds a `Lexer` must say `as view` too.

What the compiler cannot see

The compiler follows a string through one procedure. It does not follow it into another, so it cannot know that `Note` appends to the same `logText` that `head` points into:

examples/ch09/slicecheck.tiko

```
'' slicecheck.tiko -- what -exx catches that the compiler cannot.
#cmdline "-exx"

dim shared as ustring logText

sub Note( byref msg as const ustring )
    logText &= msg & " -- long enough to move the text elsewhere"
end sub
```

```
sub Report( )
    logText = "start"
    dim as const uslice head = logText[0 to 4]
    Note( "grown" )      '' grows logText behind head's back
    print head           '' head now points at freed memory
end sub

Report( )
```

examples/ch09/slicecheck.out

```
Aborting due to runtime error 6 (out of bounds array access) at line 14
of slicecheck.tiko::REPORT()
```

Every slice of a `string` or `ustring` remembers which string it came from. Under `-exx`, each read of the slice first checks that its text is still inside that string, and stops the program with runtime error 6 when it is not. A `v[i]` past the end of a slice is caught the same way.

PITFALL

Without `-exx`, `slicecheck.tiko` does not stop. It prints whatever is in the memory `head` points at. On a good day that is still `start`, because nothing has reused the freed block yet. That is the worst outcome: the program looks right, and the bug waits for a longer log.

Build with `-exx` while you develop, as Chapter 1 recommends. It costs a few compares per read of a slice, which is the price of being told. A slice of a `zstring ptr` has no string to check against at all — the pointer is trusted as it is everywhere else.

Laying out a line

`str`, `hex` and `print using` each do one job. A line that wants a name padded to a width, a number in hexadecimal and a price to two decimals becomes three calls glued with `&`, and the shape of the line disappears into them. `strings.Format` takes the shape as one string — a **format string** — and the values to put in it:

examples/ch09/format.tiko

```
'' format.tiko -- text laid out by a format string.
#import "std/strings"

dim as ustring who = "Quinn"
```

```
dim as int32 n = 42
dim as float64 x = 3.14159265358979

'' {} takes the next value; {n} takes value n, counting from 0.
print strings.Format( "Hello, {}!", who )
print strings.Format( "{1} before {0}", "b", "a" )

'' Fill, alignment and width, after a ':'.
print strings.Format( "[{:>6}] [{:<6}] [{:^6}]", n, n, n )
print strings.Format( "[{:*^9}] [{:.3}]", "hi", who )

'' Integers in other bases, a sign, zero padding.
print strings.Format( "{:x} {:#x} {:08b} {:+}", 255, 255, 5, n )
print strings.Format( "[{:+06}] [{:c}]", n, 65 )

'' Floats: fixed, exponent, and the default -- the shortest text
'' that reads back as the same value.
print strings.Format( "{:.2f} {:e} {}", x, x, 0.1 + 0.2 )

print strings.Format( "{} {:d}", true, true )
print strings.Format( "{{}} is a literal pair of braces" )
```

examples/ch09/format.out

```
Hello, Quinn!
a before b
[    42] [42    ] [  42  ]
[***hi****] [Qui]
ff 0xff 00000101 +42
[+00042] [A]
3.14 3.141593e+00 0.30000000000000004
true 1
{} is a literal pair of braces
```

Each `{}` is replaced by the next value. `{0}`, `{1}` and so on name a value by position, counting from 0, so the same value can appear twice or the order can change; a format string uses one style or the other, never both. `{{` and `}}` are literal braces. Values left over are not an error.

After a `:` comes the **spec**, a run of optional fields in a fixed order:

Field	Meaning
fill and align	`<` left, `>` right, `^` centre, padded with the fill character — a space unless one is written before the alignment, as in `{:*^9}`
sign	`+` puts a sign on every number
`#`	`0x`, `0b` or `0` before a hexadecimal, binary or octal number
`0`	pads a number with zeros after its sign: `{:+06}`
width	the least number of characters
`.precision`	digits after the point for a float; the most characters taken from a string
type	`d`, `b`, `o`, `x`, `X` and `c` for integers; `f`, `e`, `g` and `a` for floats; `s` for text

Numbers align right and everything else left unless the spec says otherwise. A `boolean` is the word `true` or `false`, or `1` and `0` under `d`.

The spec language is C++'s `std::format`, so a format string written for one reads the same in the other.

A float with no type is printed as the shortest text that reads back as the same value. That is why `0.1 + 0.2` appears with its last digit, `0.30000000000000004`: the sum is not `0.3`, and `Format` does not hide that. Give it a precision — `{:.2f}` — when a reader wants two decimals.

A width can come from a value as well, with `{:{}}`. That is what makes a table line up when the widest entry is known only at run time:

examples/ch09/formattable.tiko

```
'' formattable.tiko -- columns that line up, whatever the values.
#import "std/strings"

dim as ustring item( 0 to 2 ) = { "tea", "biscuits", "jam" }
dim as int32 qty( 0 to 2 ) = { 2, 12, 1 }
dim as float64 price( 0 to 2 ) = { 3.5, 0.25, 4.75 }

'' The width can come from a value: {:{}} takes it from the next one.
dim as int32 w = 10
print strings.Format( "{:<{}}{:>5}{:>9}", "item", w, "qty", "each" )
for i as int32 = 0 to 2
    print strings.Format( "{:<{}}{:>5}{:>9.2f}", _
```

```
                            item( i ), w, qty( i ), price( i ) )
next
```

examples/ch09/formattable.out

```
item        qty     each
tea           2     3.50
biscuits     12     0.25
jam           1     4.75
```

Checked when it compiles

A format string written as a literal is read by the compiler, and checked against the values that follow it:

examples/ch09/formatcheck.tiko

```
'' formatcheck.tiko -- a literal format string is checked when the
'' program is compiled.
#import "std/strings"

dim as int32 done = 3
print strings.Format( "{} of {} done", done )   '' error: one value
```

```
formatcheck.tiko(6,30): error TKO0851: format string: no argument 1 (1
given)
     |
   6 | print strings.Format( "{} of {} done", done )   '' error: one
value
     |                               ^
```

The caret is on the field that has no value. Every way a format string can be wrong has its own id, `TKO0847` to `TKO0858` in Appendix B: a `{` with no `}`, a spec that does not parse, a type the value cannot take — `{:x}` on text, `{:f}` on an integer — and a precision on an integer, among others.

PITFALL

A format string held in a variable is checked only when it runs — and a bad one stops the program.

The compiler can read a literal. It cannot read a variable, so a format string that arrives in one — built by concatenation, loaded from a file — is not checked until `Format` meets it. `Format` then treats it as the bug it would have been in a literal: it writes the reason and the offset to the error output and stops the program with runtime error 1, which Chapter 18 explains.

> That is right for a format string the program wrote. It is wrong for one that came from outside, where a mistake is somebody else's and the program should carry on.

Format strings that are data

A format string read from a translation file or a user's settings is data, and a bad one is not a bug in the program. `strings.TryFormat` returns the text or the reason, and never stops anything:

examples/ch09/tryformat.tiko

```
'' tryformat.tiko -- a format string that is data, not code.
#import "std/strings"
#import "std/outcome"

sub Render( byref template as const ustring )
    var r = strings.TryFormat( template, "Quinn", 3 )
    if r.IsOk( ) then
        print "ok:      "; r.Value( )
    else
        print "refused: "; r.Failure( ).message; _
              " (offset"; r.Failure( ).offset; ")"
    end if
end sub

'' Templates as they might arrive from a translation file.
Render( "{} has {} messages" )
Render( "{1} messages for {0}" )
Render( "{} has {} messages, {}" )
Render( "{:d} has {} messages" )
```

examples/ch09/tryformat.out

```
ok:      Quinn has 3 messages
ok:      3 messages for Quinn
refused: no argument 2 (2 given) (offset 20)
refused: 'd' is no presentation type for text (offset 2)
```

The answer is a `Result` — Chapter 18's type for a value or the reason there isn't one. `IsOk( )` says which, `Value( )` is the text, and `Failure( )` carries a `message`, an `offset` into the format string counted from 0, and a `code` to test against. `TryFormat` takes a plain `ustring`, so even a literal passed to it is left for run time.

Reading a table

examples/ch09/csvreport.tiko

```
'' csvreport.tiko -- reading a table and printing a report.
#import "std/strings"

dim as ustring text = _
    "name,city,year" & chr( 10 ) & _
    "Ada,""London, UK"",1815" & chr( 10 ) & _
    "Grace,New York,1906" & chr( 10 ) & _
    "Barbara,""Wilmington, DE"",1925"

var rows = strings.SplitCsv( text )
print "rows read:"; rows.Count( ); " (including the header)"
print

'' Row 0 is the header. Its field names become the column labels.
var header = rows[ 0 ].fields

for r as int32 = 1 to rows.Count( ) - 1
    var f = rows[ r ].fields
    for c as int32 = 0 to f.Count( ) - 1
        print strings.PadRight( header[ c ], 6, " " ); ": "; f[ c ]
    next
    print
next

'' A quoted field keeps its comma. Writing one back out re-quotes it.
print "field 1 of row 1 = ["; rows[ 1 ].fields[ 1 ]; "]"
```

examples/ch09/csvreport.out

```
rows read: 4 (including the header)

name  : Ada
city  : London, UK
year  : 1815

name  : Grace
city  : New York
year  : 1906

name  : Barbara
city  : Wilmington, DE
year  : 1925

field 1 of row 1 = [London, UK]
```

CSV looks like a job for `Split` and is not. A field may be quoted, and a quoted field may contain the delimiter — `"London, UK"` is one field, not two. Anything that splits on commas gets that wrong, and gets it wrong on exactly the rows a test file does not contain.

`strings.SplitCsv( text )` parses a whole table and returns rows, each with a `.fields` array. `strings.SplitCsvRow( line )` parses a single record and returns the fields directly. Both handle quoting, embedded delimiters and embedded newlines. Going the other way, `strings.JoinCsvRow` quotes each field that needs it, and `strings.CsvWriter` writes rows to a file (Chapter 25).

A string as bytes

examples/ch09/binary.tiko

```
'' binary.tiko -- a string as a container of bytes.

'' mk* packs a number into its raw bytes; cv* reads it back.
dim as string packed = mkl( 70000 ) & mkd( 3.5 ) & mkshort( 300 )
print "packed bytes ="; len( packed )

print "cvl   ="; cvl( mid( packed, 1, 4 ) )
print "cvd   ="; cvd( mid( packed, 5, 8 ) )
print "cvshort ="; cvshort( mid( packed, 13, 2 ) )

'' Fixed-length strings can be aligned within their room.
dim as string * 10 slot
lset slot, "left"
print "lset  = ["; slot; "]"
rset slot, "right"
print "rset  = ["; slot; "]"

'' strptr and sadd reach the bytes themselves.
dim as string s = "hello"
print "first byte via sadd ="; peek( ubyte, sadd( s ) )
print "asc of it           ="; asc( s, 1 )
```

examples/ch09/binary.out

```
packed bytes = 14
cvl   = 70000
cvd   = 3.5
cvshort = 300
lset  = [left      ]
rset  = [     right]
```

```
first byte via sadd =104
asc of it           =104
```

The `mk*` family packs a number into its raw bytes and `cv*` reads it back: `mkl`/`cvl` for 32-bit, `mkshort`/`cvshort` for 16, `mkd`/`cvd` and `mks`/`cvs` for floating point, `mki`/`cvi` for the pointer-sized `integer`. This is how a record gets written to a binary file, which Chapter 25 comes back to.

`lset` and `rset` align text within a fixed-length string. `strptr` and `sadd` give the address of the characters themselves, which is what a C function needs — and what makes it your responsibility to keep the string alive while the pointer is in use.

Bytes as text: base64, hex and URLs

Some places take only text: a JSON field, an e-mail body, a URL, a configuration file. Bytes that are not text — a hash, a key, an image — have to be spelled in characters that survive the trip. `std/codec` has the three common spellings:

examples/ch09/bytesastext.tiko

```
#import "std/codec"
#import "std/containers"

'' Four bytes that no text format can carry as they are.
dim raw as containers.Array( of ubyte )
dim b( 0 to 3 ) as uint8 = { &h00, &hFB, &hFF, &h7F }
for i as int32 = 0 to 3
    raw.Push( b( i ) )
next

print "hex:     "; codec.HexEncode( raw )
print "base64:  "; codec.Base64Encode( raw )
print "url-safe "; codec.Base64Encode( raw, codec.B64_URL, false )

'' Text is encoded as its UTF-8 bytes, and decodes back the same.
dim word as ustring = "na" & wchr( &hEF ) & "ve"
dim enc as ustring = codec.Base64Encode( word )
print "naive:   "; enc
dim back as ustring = codec.Base64DecodeText( enc ).Value( )
print "same:    "; cbool( back = word )
```

examples/ch09/bytesastext.out

```
hex:     00fbff7f
base64:  APv/fw==
url-safe APv_fw
```

```
naive:   bmHDr3Zl
same:    true
```

Hex writes each byte as two digits: twice the size, and readable. **Base64** packs three bytes into four characters, a third bigger than the bytes; its URL-safe alphabet swaps `+` and `/` for `-` and `_`, and usually drops the `=` padding. **Base32** (`Base32Encode`) uses only capital letters and digits, for codes that people type. Each has an `…Encode`, a `…Decode` that fills a `containers.Array( of ubyte )`, and a `…DecodeText` that returns a `ustring` — text is encoded as its UTF-8 bytes, so `…DecodeText` gives back exactly the text that went in.

A decoder can be lenient or exact. These are exact:

examples/ch09/strictdecode.tiko

```
#import "std/codec"

'' A decoder that guesses hides corrupt data. This one refuses.
dim inputs( 0 to 3 ) as ustring = _
    { "aGVsbG8=", "aGVsbG8", "aGVs bG8=", "aGVsbG9=" }

for each t in inputs
    var r = codec.Base64DecodeText( t )
    if r.IsOk( ) then
        print t; " -> "; r.Value( )
    else
        print t; " -> "; r.Failure( ).Message( )
    end if
next
```

examples/ch09/strictdecode.out

```
aGVsbG8= -> hello
aGVsbG8 -> Base64DecodeText: invalid padding
aGVs bG8= -> Base64DecodeText: invalid character at 5
aGVsbG9= -> Base64DecodeText: non-zero trailing bits at 7
```

The second input lost its padding, the third gained a space, and the fourth has a final character whose unused low bits are not zero — three different damages, each refused with the position of the first bad character. A lenient decoder would have returned *something* for all three, and the fourth would have decoded to `hello`, so a corrupted value would have compared equal to the original. When the input is known to be loose, say so: `DECODE_IGNORE_WHITESPACE` and `DECODE_ALLOW_MISSING_PAD`, combined with `or`, relax exactly those two rules.

Text inside a URL needs a different treatment. **Percent-encoding** keeps letters, digits and `- . _ ~`, and writes every other byte of the UTF-8 as `%` and two hex digits:

examples/ch09/urlencode.tiko

```
#import "std/codec"

dim dish as ustring = "fish & chips"
print "component: "; codec.PercentEncode( dish )
print "form:      "; codec.PercentEncode( dish, codec.PCT_QUERY )
print "path:      "; _
    codec.PercentEncode( "menu/" & dish, codec.PCT_PATH )

var r = codec.PercentDecode( "fish+%26+chips", codec.PCT_QUERY )
print "decoded:   "; r.Value( )
```

examples/ch09/urlencode.out

```
component: fish%20%26%20chips
form:      fish+%26+chips
path:      menu/fish%20&%20chips
decoded:   fish & chips
```

The mode decides what else survives. `PCT_COMPONENT`, the default, is for one value placed inside a URL, and escapes `/` and `&` because they would end it. `PCT_PATH` keeps the path's own `/`. `PCT_QUERY` is the encoding an HTML form submits, with a space as `+`. Decode with the mode you encoded with: `+` means a space only in `PCT_QUERY`.

Working with letters

examples/ch09/caesar.tiko

```
'' caesar.tiko -- shifting letters, which is arithmetic on asc codes.

function Shift( byref text as ustring, byval by as int32 ) as ustring
    dim as ustring built = ""

    for i as int32 = 1 to len( text )
        dim as int32 c = asc( text, i )

        if c >= asc( "a" ) andalso c <= asc( "z" ) then
            c = asc( "a" ) + ( ( c - asc( "a" ) + by + 26 ) mod 26 )
        elseif c >= asc( "A" ) andalso c <= asc( "Z" ) then
            c = asc( "A" ) + ( ( c - asc( "A" ) + by + 26 ) mod 26 )
        end if

        built &= chr( c )
```

```
    next

    return built
end function

dim as ustring plain = "Attack at dawn!"
dim as ustring coded = Shift( plain, 3 )

print "plain   : "; plain
print "shifted : "; coded
print "back    : "; Shift( coded, -3 )
print "round trip holds: "; cbool( Shift( coded, -3 ) = plain )
```

examples/ch09/caesar.out

```
plain   : Attack at dawn!
shifted : Dwwdfn dw gdzq!
back    : Attack at dawn!
round trip holds: true
```

`asc` and `chr` move between a character and its number, which turns text into arithmetic. The `+ 26` before the `mod` is there because Chapter 5's rule still applies: `mod` takes the sign of its left operand, so shifting backwards past `a` would otherwise give a negative answer.

This works for ASCII letters and stops there. It is a cipher exercise, not a text-processing technique — `ucase` on a `ustring` already knows about every alphabet, and hand-rolled arithmetic does not.

A value with no printable form

```
print "X["; Box( 7 ); "]Y"      '' refused
```

A type that has no conversion to text has nothing for `print` to show, whether it arrives as a variable or as a temporary like `Box( 7 )`: both are refused with `TKO0871: a 'box' has no printable form`. Give the type an `operator cast( ) as ustring` — Chapter 13 — and it prints.

What it costs

- **`len` on a `ustring` is O(1).** So is `len` on a `string`. Neither walks the text.
- **`ustring` uses 2 bytes per code unit**, so ASCII text costs twice what it would as a `string`. That is the price of not having to think about encoding, and it is almost always worth paying.

- **`&` builds a new string.** Appending in a loop copies what is there each time; for a large result, build into one string and overwrite with the `mid` statement, or let the library do it.
- **A slice costs nothing to make** — no allocation, no copy — and `len` on it is O(1). Keeping its text, by assigning it to a `ustring`, is the copy you skipped; under `-exx` each read of a slice is also checked against its source.
- **Converting between `string` and `ustring` is a re-encode**, not a cast. Do it at the boundary, once, not inside a loop.
- **`instr` is a scan.** Finding n things in a long string costs n scans unless you carry the position forward with `instr( start, s, x )`.
- **`Format` reads its format string on every call**, and copies each text value it is given. In a loop over millions of lines that is measurable; for a report it is not.

Summary

- Five types: **`ustring` for text**, `string` for bytes, `zstring`/`wstring` for C, `string * n` fixed and **space-padded**.
- `wstring` is 2 bytes per character on Windows and 4 on Linux; `ustring` is UTF-16 everywhere.
- **A block string** opens with `"""` at the end of a line and closes at the first line starting with `"""`; **that closing line's indentation is the margin** removed from every line. `!"""` reads escapes, and every line break inside is `chr( 10 )`.
- **Positions start at 1, and "not found" is 0.**
- The `mid` statement overwrites in place and never changes the length, on `string` and `ustring` alike.
- `str` has no padding; `print` reserves a column for the sign. `hex`/`oct`/`bin` take a width.
- **`val( ) cannot fail`** — it returns 0 for text that is not a number. Use `IsNumeric`, or Chapter 18's `Outcome`, when it matters.
- **`len` on a `ustring` counts code units.** Outside the BMP one character is two units, so arithmetic positions can cut a character in half.
- **Converting between `string` and `ustring` is UTF-8 in both directions, on every target**, and lossless.
- **Bytes that are not UTF-8 become U+FFFD, silently.** Convert at the boundary, knowing what you have.
- `#import "std/strings"` adds ~45 operations including `Split`/`Join`, `Replace`, the `Pad*` family, `Between` and the CSV parsers. `Join( Split( s, d ), d )` is `s`.
- **`strings.Format` lays out a line from a format string**: `{}` or `{n}` for each value, then `:` and a spec — fill, alignment, sign, width, precision, type. A float with no type is the shortest text that reads back exactly.

- **A literal format string is checked when the program compiles** (`TKO0847` to `TKO0858`). One held in a variable is checked when it runs, and a bad one stops the program; for format strings that are data, `TryFormat` returns a `Result` instead.
- **`s[lo to hi]` is a slice**: part of `s`, counted from 0, both ends included, never copied. `uslice`, `slice` and `wslice` match `ustring`, `string` and `wstring`. A slice is as writable as its source; assigning it to a `ustring` keeps a copy.
- `strings.TrimSpace`, `Find` (0-based, −1 for none), `Equals` and `SplitSlices` work on slices without copying.
- **A slice cannot outlive its text.** It cannot be returned from the function that owns the text (`TKO1059`), stored where it could outlive it (`TKO1054`, `TKO1057`, `TKO1058`), or kept while its string grows (`TKO1062`). A type that holds one is declared `as view`.
- **`-exx` catches what crosses a procedure**: a slice read after its string moved stops with runtime error 6. Without it, the slice reads freed memory.
- **CSV is not a `Split` problem** — use `SplitCsv` or `SplitCsvRow`, which understand quoting.
- `#import "std/codec"`: `HexEncode`, `Base64Encode` (standard or URL-safe) and `Base32Encode` spell bytes as text; `…Decode` and `…DecodeText` are **strict** and report the position of the first bad character. `PercentEncode` makes text safe in a URL — component, path or form query.
- `mk*`/`cv*` pack numbers into bytes; `lset`/`rset` align in a fixed-length string; `strptr`/`sadd` reach the characters for C.
- **A value with no printable form is refused by `print`** (`TKO0871`), variable or temporary.

Exercises

1. Write `TitleCase`, which uppercases the first letter of each word and lowercases the rest, then check it against `strings.MCase` on a sentence, a shouted one, one with double spaces and an empty string. {solution="ex09-01″}
2. Parse a block of `key = value` lines into pairs, skipping blanks and lines with no `=`. A value may itself contain an `=`; make sure yours survives that. {solution="ex09-02″}
3. Take the `café` bytes from `codeunits.tiko` and print `len` of the `string` and of the `ustring`. Then do the same for a string of pure ASCII. When do the two numbers agree, and why?
4. `mid( face, 2, 1 )` on the emoji text returns half a character. Write a loop that walks a `ustring` and prints each *character*, treating a surrogate pair as one. What does it need to check?
5. Read a CSV line with `SplitCsvRow`, then split the same line on commas with `Split`, and print both field counts. Construct a line where they differ.

6. Print a multiplication table for 1 to 12 with `strings.Format`, every column right-aligned. Then make the column width depend on the largest product, so the table still lines up for 1 to

   99.
7. Rewrite `lexer.tiko`'s `NextWord` so that it also returns `=`, `*` and the other punctuation as words of their own, still without copying. Then move `print "<"; w; ">";` into a procedure that takes `byval w as const uslice`, and call it with a `ustring` and with a literal as well as with a slice.
8. Encode the bytes 0 to 255 in base64, then decode the result with one character changed in each position in turn. Does every change get refused? Which ones decode, and to what?

User-Defined Types and Layout

Part I worked with values one at a time: a number, a string, an array of numbers. Real data comes in groups — a point has an x and a y, a reading has a station and a temperature — and a program that keeps the parts in separate variables keeps them in step by hand, which means sometimes out of step.

A `type` puts the parts under one name. This chapter covers what that gives you, and then what it costs in memory, because a type is also a decision about where bytes go. That second half is what lets tiko talk to file formats, network protocols and C libraries without translation.

Chapter 12 turns these types into objects with methods. Everything here still applies then; it is the foundation, not a beginner's version.

Grouping values

examples/ch10/pointrect.tiko

```
'' pointrect.tiko -- a type groups values that belong together.

type Point
    x as int32
    y as int32
end type

type Rect
    topLeft as Point                  '' a type inside a type
    size    as Point = ( 10, 20 )     '' fields can start with a value
    title   as ustring = "untitled"
    visible as boolean = true
end type

dim as Rect r
print "a fresh Rect: size"; r.size.x; r.size.y; _
      ", title "; r.title; ", visible "; r.visible

r.topLeft.x = 5                       '' reach in with dots
r.title = "window"
print "after edits:  topLeft"; r.topLeft.x; r.topLeft.y; _
      ", title "; r.title

'' A literal in parentheses fills the fields in order.
dim as Point origin = ( 0, 0 )
```

```
dim as Point corner = ( 640, 480 )
print "corner:"; corner.x; corner.y; "  origin:"; origin.x; origin.y
```

examples/ch10/pointrect.out

```
a fresh Rect: size 10 20, title untitled, visible true
after edits:  topLeft 5 0, title window
corner: 640 480  origin: 0 0
```

A `type` lists its fields, each with a name and a type. Fields are reached with a dot, and a type can contain another type, so `r.topLeft.x` reads the way the data is shaped.

A field may carry an initialiser — `size as Point = ( 10, 20 )` — and every variable of the type starts with it. Fields without one start at zero, or empty for a string, exactly as variables do. A value in parentheses, `( 640, 480 )`, fills a type's fields in declaration order.

Parentheses, not braces. Braces are how an *array* is filled (Chapter 8), and a type is not an array:

examples/ch10/bracesudt.tiko

```
'' bracesudt.tiko -- a type is filled with parentheses. Braces are for
'' arrays, and a Point is not one.

type Point
    as int32 x, y
end type

dim as Point good = ( 3, 4 )
dim as Point bad = { 3, 4 }         '' refused
print good.x, bad.x
```

```
bracesudt.tiko(9,20): error TK00845: 'bad' is not an array, so it cannot
take a '{ ... }' initializer
     |
   9 | dim as Point bad = { 3, 4 }         '' refused
     |                    ^
```

fbc refuses it in the same place, with its error 63. Braces are for arrays, parentheses for types.

> **NOTE**
> `name` is a reserved word — the `name` statement renames a file — so it cannot name a variable or a procedure. A **field** may still be called `name`, and `b.name` reads it:

> a field's name is only read after `.` or `->`. Appendix A lists the reserved words and the few that cannot name a field.

A type is a value

examples/ch10/copying.tiko

```
'' copying.tiko -- assignment copies the whole value, text included.

type Account
    owner   as ustring
    balance as int64
end type

dim as Account a
a.owner = "Ada"
a.balance = 100

dim as Account b = a            '' a complete, independent copy
b.owner = "Grace"
b.balance = 250

print "a: "; a.owner; a.balance
print "b: "; b.owner; b.balance
```

examples/ch10/copying.out

```
a: Ada 100
b: Grace 250
```

Assigning one variable of a type to another copies **every field**, including the text: changing `b.owner` afterwards leaves `a.owner` alone. The two are independent values, the way two `int32`s are, and nothing is shared.

Comparison is a different matter:

examples/ch10/noequal.tiko

```
'' noequal.tiko -- a type can be copied, but not compared.
type Point
    x as int32
    y as int32
end type

dim as Point a = ( 1, 2 ), b = ( 1, 2 )
if a = b then print "equal"     '' refused: what would equal mean?
```

```
noequal.tiko(8,6): error TKO0720: no operator = compares this type
     |
   8 | if a = b then print "equal"     '' refused: what would equal mean?
     |      ^
```

The compiler will not guess what equality means for your type — whether two accounts with the same owner are equal, or only ones with the same balance too. Chapter 13 shows how to say.

Passing and returning

Chapter 7's table said that a type you define is passed by reference when no keyword is given. Here is the measurement:

examples/ch10/passing.tiko

```
'' passing.tiko -- a type you define is passed by reference unless
'' you say otherwise. Chapter 7 said so; this is the proof.

type Point
    x as int32
    y as int32
end type

sub NoKeyword( p as Point )          '' the default
    p.x = 99
end sub

sub Copied( byval p as Point )       '' a private copy
    p.x = 77
end sub

function Shifted( byval p as Point, byval dx as int32 ) as Point
    p.x += dx                        '' changes the copy ...
    return p                         '' ... and hands it back
end function

dim as Point a = ( 1, 2 )

NoKeyword( a )
print "after NoKeyword:"; a.x; "   <- the caller's Point changed"

a.x = 1
Copied( a )
print "after Copied:   "; a.x
```

```
dim as Point b = Shifted( a, 10 )
print "Shifted gave";  b.x; ", and a is still"; a.x
```

examples/ch10/passing.out

```
after NoKeyword: 99   <- the caller's Point changed
after Copied:    1
Shifted gave 11, and a is still 1
```

With no keyword, the procedure wrote straight into the caller's `Point`. With `byval` it worked on a copy. A function can return a type, and `Shifted` returns its modified copy while the original stays put.

The rule from Chapter 7 holds with more force here, because a type can be large: **write `byref` or `byval` on every parameter**. `byref ... as const` is the right shape for a type you only read — no copy, and the compiler stops you writing to it.

Where the bytes go

examples/ch10/layout.tiko

```
'' layout.tiko -- where the bytes go, and what decides it.

'' Each field sits at a multiple of its own size, so a 4-byte int32
'' after a 1-byte field leaves a 3-byte gap.
type Loose
    a as uint8
    b as int32
    c as uint8
end type

'' The same fields, largest first: the gaps disappear.
type Ordered
    b as int32
    a as uint8
    c as uint8
end type

'' field = n caps the alignment at n bytes.
type Packed field = 1
    a as uint8
    b as int32
    c as uint8
end type
```

```
print "Loose   sizeof"; sizeof( Loose ); _
      "  b at"; offsetof( Loose, b ); "  c at"; offsetof( Loose, c )
print "Ordered sizeof"; sizeof( Ordered ); _
      "  a at"; offsetof( Ordered, a ); "  c at"; offsetof( Ordered, c )
print "Packed  sizeof"; sizeof( Packed ); _
      "  b at"; offsetof( Packed, b ); "  c at"; offsetof( Packed, c )

'' field = packs the INSIDE of a type. Put that type inside another,
'' and the outer one still aligns it.
type Holder
    tag as uint8
    p   as Packed
end type
type PackedHolder field = 1
    tag as uint8
    p   as Packed
end type
print "Holder       p at"; offsetof( Holder, p ); _
      "  sizeof"; sizeof( Holder )
print "PackedHolder p at"; offsetof( PackedHolder, p ); _
      "  sizeof"; sizeof( PackedHolder )
```

examples/ch10/layout.out

```
Loose   sizeof 12  b at 4  c at 8
Ordered sizeof 8  a at 4  c at 5
Packed  sizeof 6  b at 1  c at 5
Holder       p at 4  sizeof 12
PackedHolder p at 1  sizeof 7
```

`sizeof` is a type's size in bytes, and `offsetof( T, field )` is where a field starts. Read the first three lines together.

`Loose` holds 6 bytes of data and takes 12. Each field is placed at a multiple of its own size, so the `int32` after a one-byte field starts at offset 4, leaving three unused bytes — and the whole type is rounded up to a multiple of its largest field so that an array of them keeps every element aligned. That padding is not waste for its own sake: the processor reads aligned values in one step and misaligned ones in two, or not at all.

`Ordered` has the same fields, largest first, and takes 8. **Ordering fields from largest to smallest is the cheapest optimisation in this book** — no code changes, and every array of the type gets smaller. Exercise 2 puts a number on it.

`field = 1` turns alignment off: `Packed` is 6 bytes with no gaps. Reach for it when the layout is dictated from outside — a file format, a network header, a C struct

declared with packing — and not otherwise, because the misaligned reads it creates are slower.

> **PITFALL**
> **`field =` packs the inside of a type, not the outside.** `Packed` has no gaps between its own fields, but put it inside `Holder` and it still starts at offset 4, with three bytes of padding in front of it — because `Holder` aligns its members normally.
>
> When a whole structure must match a byte-exact format, **every type in it needs its own `field = 1`**. `PackedHolder` has one, and its `p` lands at offset 1. Missing one level is the classic way to produce a header that is right in isolation and wrong in the file.

Seeing the layout

The compiler will print the layout it chose. From the directory holding the listing:

```
tiko --dump-udt layout.tiko
```

```
loose size 12 align 4
  loose off a 0
  loose off b 4
  loose off c 8
ordered size 8 align 4
  ordered off b 0
  ordered off a 4
  ordered off c 5
packed size 6 align 4
  packed off a 0
  packed off b 1
  packed off c 5
holder size 12 align 4
  holder off tag 0
  holder off p 4
packedholder size 7 align 4
  packedholder off tag 0
  packedholder off p 1
object size 8 align 8
  object off vptr 0
udts: 6
```

Names appear in lower case, because tiko's names are not case-sensitive. The `align` column is the alignment a type asks for **when it is placed inside another** — which

is why `packed` still says 4, and why the pitfall above happens. `object` is the built-in base type Chapter 14 introduces; it is listed whether or not you use it.

`--dump-udt` stops after printing, without building anything. This output was captured at the pin and is identical on both targets.

Several views of the same bytes

examples/ch10/unions.tiko

```
'' unions.tiko -- several views of the same bytes.

union Bits32
    asFloat as float32
    asInt   as uint32
    bytes( 0 to 3 ) as uint8
end union

dim as Bits32 u
u.asFloat = 1.0
print "sizeof( Bits32 ) ="; sizeof( Bits32 )
print "1.0 as float32 is the bit pattern &h"; hex( u.asInt, 8 )
print "its highest byte is "; u.bytes( 3 )

'' An anonymous union inside a type: one slot, two meanings, and a
'' field beside it that says which meaning is current.
type Value
    isFloat as boolean
    union
        i as int64
        f as float64
    end union
end type

dim as Value v
v.isFloat = true
v.f = 2.5
print "Value: sizeof"; sizeof( Value ); ", i and f both at offset"; _
      offsetof( Value, i ); offsetof( Value, f )
if v.isFloat then print "holding the float"; v.f
```

examples/ch10/unions.out

```
sizeof( Bits32 ) = 4
1.0 as float32 is the bit pattern &h3F800000
its highest byte is 63
```

```
Value: sizeof 16, i and f both at offset 8 8
holding the float 2.5
```

A `union` is declared like a type, but its fields all start at offset 0: they are different interpretations of one block of memory, and the union is as large as its largest field. Writing `asFloat` and reading `asInt` shows the IEEE 754 bit pattern of 1.0, which is the classic use — and also the only kind of use where reading a different field than the one last written is meaningful.

An **anonymous union** inside a type gives one slot two meanings. `Value` stores either an `int64` or a `float64` in the same eight bytes, and the `isFloat` field beside it records which one is current. That pairing is the whole discipline of a union: **something must say which field is live**, because the union cannot.

Fields narrower than a byte

examples/ch10/packetheader.tiko

```
'' packetheader.tiko -- fields narrower than a byte.

'' The first 32 bits of an IPv4 header, as bitfields. Each field
'' says how many bits it occupies; together they fill one uint32.
type Header
    version : 4  as uint32
    ihl     : 4  as uint32
    tos     : 8  as uint32
    length  : 16 as uint32
end type

dim as Header h
h.version = 4
h.ihl     = 5
h.tos     = 0
h.length  = 1500

print "sizeof( Header ) ="; sizeof( Header )
print "version "; h.version; "  ihl "; h.ihl; "  length "; h.length

'' The whole thing, read as one number. The first field declared
'' is in the LOWEST bits: version is the last hex digit.
dim as uint32 ptr whole = cast( uint32 ptr, @h )
print "as one uint32: &h"; hex( *whole, 8 )

'' A value too wide for its field is cut down to fit, silently.
```

```
h.version = 17
print "after version = 17, version is "; h.version
```

examples/ch10/packetheader.out

```
sizeof( Header ) = 4
version 4  ihl 5  length 1500
as one uint32: &h05DC0054
after version = 17, version is 1
```

`version : 4 as uint32` declares a **bitfield**: a field four bits wide, carved out of a `uint32`. Consecutive bitfields share their containing integer, so four of them describe the first 32 bits of an IPv4 header in exactly 4 bytes.

Reading the whole word back shows the allocation order: `&h05DC0054` has `version` (4) in the **lowest** four bits and `length` (1500, `&h5DC`) in the highest sixteen. The first field declared takes the least significant bits. That is the order the targets in this book use, and the order C compilers use on them — but it is not the order a network protocol transmits, which is why a real header parser reads bytes and shifts rather than overlaying a bitfield type on the wire.

`--dump-udt` reports bitfields at the byte offset of their container, so it cannot show bit positions; the listing measures them by reading the word back.

> **PITFALL**
> **A value too large for a bitfield is truncated, silently.** `version` is four bits, so it holds 0 to 15, and assigning 17 stores 17's low four bits — 1. No warning, no runtime error, even under `-exx`.
>
> Check the range before you store, wherever the value comes from outside.

Naming a type once

examples/ch10/withblock.tiko

```
'' withblock.tiko -- naming the thing once.

type Settings
    width   as int32
    height  as int32
    title   as ustring
    resizable as boolean
end type

dim as Settings s
```

```
with s
    .width     = 800
    .height    = 600
    .title     = "Report"
    .resizable = false
end with

print s.title; ":"; s.width; " x"; s.height; ", resizable "; s.resizable

'' An alias is another name for the same type, not a new type.
type Dimensions as Settings
dim as Dimensions d = s
print "through the alias: "; d.title; d.width
```

examples/ch10/withblock.out

```
Report: 800 x 600, resizable false
through the alias: Report 800
```

Inside `with s … end with`, a field written with a leading dot belongs to `s`. It is shorthand for filling in a type with many fields, and it reads well for exactly that. Inside a long block it reads less well, because the reader has to remember which variable the dots refer to.

`type Dimensions as Settings` makes a second name for an existing type. It is the same type, not a new one — a `Dimensions` and a `Settings` can be assigned to each other freely. Aliases are for giving a type a name that fits the context, or a shorter one, and Chapter 17 relies on them to name instantiated generics.

Types that refer to each other

examples/ch10/linked.tiko

```
'' linked.tiko -- types that point at each other.

'' A type may point at itself: the compiler only needs to know the
'' size of a pointer, which it always does.
type Node
    value as int32
    next_ as Node ptr
end type

dim as Node a, b, c
a.value = 1 : b.value = 2 : c.value = 3
```

```
a.next_ = @b : b.next_ = @c : c.next_ = nullptr

dim as Node ptr p = @a
print "the list:";
do while p <> nullptr
    print p->value;
    p = p->next_
loop
print

'' Two types that point at EACH OTHER need one of them named before
'' it is declared. A forward alias does that.
type ChildFwd as Child

type Parent
    firstChild as ChildFwd ptr
end type

type Child
    owner as Parent ptr
end type

dim as Parent mum
dim as Child kid
mum.firstChild = @kid
kid.owner = @mum
print "the child's owner is the parent: "; cbool( kid.owner = @mum )
```

examples/ch10/linked.out

```
the list: 1 2 3
the child's owner is the parent: true
```

A type can hold a **pointer** to its own kind, which is how linked structures are built. It cannot hold a value of its own kind — that would be infinitely large — but a pointer is 8 bytes on every target this book uses, so the compiler needs nothing more. Chapter 11 covers pointers properly; here they are only the arrows between records.

Two types that point at *each other* are a different problem. Whichever is written first mentions the other before it is declared, and the compiler refuses that with `TK00784: the type 'child' is used above its declaration`. A **forward alias** — `type ChildFwd as Child` — names the type before it exists, and a pointer to it can be used straight away.

A forward alias that is never completed is an **opaque type**:

examples/ch10/opaque.tiko

```
'' opaque.tiko -- a type known only by name.
type Handle as HandleImpl          '' never declared in this file

dim as Handle ptr h = nullptr      '' a pointer is fine
dim as Handle whole                '' a value is not: how big is it?
```

```
opaque.tiko(5,8): error TKO0810: 'handleimpl' is an incomplete type:
nothing declares it, so only a pointer to it may be used here
     |
   5 | dim as Handle whole                '' a value is not: how big is
it?
     |        ^
```

That refusal is the feature. A library can hand out `Handle ptr` values while keeping the contents of `HandleImpl` to itself, and callers cannot depend on what they cannot see. Chapter 29 meets the same pattern from C's side.

Arrays of records

examples/ch10/udtarray.tiko

```
'' udtarray.tiko -- an array whose elements are records.

type Reading
    station as ustring
    celsius as float64
end type

'' A small function that builds one record keeps the call sites to a
'' single readable line each.
function MakeReading( byref station as ustring, _
                      byval celsius as float64 ) as Reading
    dim as Reading r
    r.station = station
    r.celsius = celsius
    return r
end function

dim as Reading log_( 1 to 4 )
log_( 1 ) = MakeReading( "north", 11.5 )
log_( 2 ) = MakeReading( "south", 18.0 )
log_( 3 ) = MakeReading( "east",  14.25 )
log_( 4 ) = MakeReading( "west",  9.75 )
```

```
'' By index, when the position matters.
dim as int32 warmest = lbound( log_ )
for i as int32 = lbound( log_ ) to ubound( log_ )
    if log_( i ).celsius > log_( warmest ).celsius then warmest = i
next
print "warmest: "; log_( warmest ).station; log_( warmest ).celsius

'' for each byref, when it does not. Each record is used in place,
'' without being copied.
dim as float64 total = 0
for each byref r as Reading in log_
    total += r.celsius
next
print "mean:"; total / ( ubound( log_ ) - lbound( log_ ) + 1 )

'' byref also lets the loop change them.
for each byref r as Reading in log_
    r.celsius = r.celsius * 9 / 5 + 32
next
print "north in fahrenheit:"; log_( 1 ).celsius
```

examples/ch10/udtarray.out

```
warmest: south 18
mean: 13.375
north in fahrenheit: 52.7
```

An array of a type works like any array, and Chapter 8's advice carries over: index when the position matters, `for each byref` when it does not. For records, `byref` is more than a style choice, because the alternative copies every field — text included — on every pass.

`MakeReading` builds one record per call, which keeps each line of setup readable. Chapter 12 turns that function into a **constructor**, which is where it belongs.

> **NOTE**
> **A record that holds text is not built with `type( ... )`.** `log_( 2 ) = type( "south", 18.0 )` over a `Reading` holding a `ustring` is refused — `TK00929: type( ... ) cannot build a 'reading'` — because only a constructor can fill a var-len string. Build such records through a function like `MakeReading`, or — from Chapter 12 — with a constructor, which `type( ... )` then calls.

What it costs

- **A type costs exactly its `sizeof`.** There is no hidden header and no per-object allocation until Chapter 12 adds virtual methods, and even then it is one pointer.
- **Padding is real memory.** Ordering fields largest first is free and can save a third.
- **`field = 1` trades speed for exactness.** Use it where a format demands it, nowhere else.
- **Assignment copies everything**, and a `ustring` field is a separate allocation copied each time. Pass large types `byref`.
- **A union costs its largest member**, and the field that says which member is live.

Summary

- `type` groups named fields; a field may have an initialiser, and may itself be a type. `( a, b )` fills fields in order. **Braces are for arrays** — `{ a, b }` on a type is `TKO0845`
- **Assignment copies the whole value**, text included. **`=` does not compare types** — `TKO0720` — until Chapter 13 defines it.
- **With no keyword, a type is passed `byref`.** `byval` copies; a function may return a type.
- `sizeof` and `offsetof` measure layout. Fields align to their own size, and the type rounds up to its largest field — **order fields largest first**.
- `field = n` caps alignment — **for the inside of that type only**; nested types need their own.
- `tiko --dump-udt file.tiko` prints the layout the compiler chose.
- A `union` overlays its fields at offset 0; something beside it must record which one is live.
- **Bitfields** pack narrow fields into an integer, first field in the lowest bits, and **truncate an oversized value silently**.
- `with` shortens repeated access; `type A as B` is an alias for the same type.
- A type may point at itself; **types that point at each other need a forward alias**, and an alias never completed is an opaque type usable only through a pointer.
- Arrays of types: index for position, `for each` otherwise — **`for each byref` for records**, which the by-value form would copy field by field.
- **`type( ... ) cannot build a record holding text`** (`TKO0929`); build it through a function or a constructor.

Exercises

1. Declare a 16-byte record header — magic, version, flags, length, checksum — that must match a published binary format exactly, and have the program verify every offset against the spec instead of trusting it. {solution="ex10-01″}
2. Take a type with six fields of mixed sizes declared carelessly, reorder them largest first, and print both sizes and what a million of each would cost. {solution="ex10-02″}
3. Put a `field = 1` type inside an ordinary one and print the offset of the inner type. Then fix the outer type so the whole thing has no padding. How many `field =` did it take?
4. Write a `union` holding a `float64` and an array of eight `uint8`, store −0.0 and 0.0, and print the bytes of each. Which byte differs, and why does `=` on the two floats still say they are equal?
5. Build a three-element linked list of `Node`s and write a procedure that prints it given only the first node's address. What should the procedure do if it is handed `0`?

Pointers, References and Memory

Every variable so far has lived somewhere the compiler chose and lasted as long as the compiler decided. That is the right default, and most of a tiko program never needs anything else.

This chapter is about the rest: memory you ask for while the program runs, structures that link to each other, and addresses you hand to code that expects them — C libraries, the operating system, a buffer shared between two parts of a program. A **pointer** is the address of a value, and with it comes the responsibility the compiler was carrying for you: knowing that the address is valid, and giving the memory back.

Chapter 22 shows the tools that carry most of that responsibility again — destructors, `defer`, shared pointers. Learn the raw form first, because every one of those tools is built on it.

An address, and what lives at it

examples/ch11/basics.tiko

```
'' basics.tiko -- an address, and what lives at it.

dim as int32 n = 10
dim as int32 ptr p = @n          '' @ takes the address

print "*p ="; *p                 '' * reads what p points at
*p = 20                          '' ... or writes it
print "n is now"; n

print "varptr( n ) is the same address: "; cbool( varptr( n ) = p )

'' A pointer to a type reaches its fields with ->.
type Point
    x as int32
    y as int32
end type

dim as Point pt = ( 3, 4 )
dim as Point ptr pp = @pt
pp->x = 30
print "pt.x ="; pt.x; ", and (*pp).y ="; ( *pp ).y

'' nullptr is the null pointer: it points at nothing.
dim as int32 ptr nothing_ = nullptr
print "nothing_ is null: "; cbool( nothing_ = nullptr )
```

examples/ch11/basics.out

```
*p = 10
n is now 20
varptr( n ) is the same address: true
pt.x = 30, and (*pp).y = 4
nothing_ is null: true
```

Written	Means
`int32 ptr`	the type: an address of an `int32`
`@n` or `varptr( n )`	the address of `n`
`*p`	the value at `p` — readable and writable
`p->x`	field `x` of the type at `p`; the same as `( *p ).x`
`nullptr`	the null pointer: an address of nothing

Writing through `*p` changed `n`, because `p` holds `n`'s address — there is one value and two ways to reach it. That sharing is the whole point of a pointer, and the source of every pointer bug.

nullptr, not 0

`0` works as a null pointer too: `p = 0` and `if p = 0` are legal, and existing code is full of them. But `0` is an integer, and wherever the compiler has to choose a type for it, it chooses an integer's:

examples/ch11/nullptr.tiko

```
'' nullptr.tiko -- the null pointer, spelled as a pointer.

sub Show overload ( byval n as integer )
    print "  Show( integer )"
end sub

sub Show overload ( byval p as any ptr )
    print "  Show( any ptr )"
end sub

print "0 is an integer, so it picks the integer overload:"
Show( 0 )
print "nullptr is a pointer:"
Show( nullptr )

'' var takes the type of what it is given.
var a = 0
var b = nullptr
print "var from 0, then var from nullptr:"
```

```
Show( a )
Show( b )
```

examples/ch11/nullptr.out

```
0 is an integer, so it picks the integer overload:
  Show( integer )
nullptr is a pointer:
  Show( any ptr )
var from 0, then var from nullptr:
  Show( integer )
  Show( any ptr )
```

`nullptr` is an `any ptr` whose value is 0, so it converts to every pointer type — typed pointers, `zstring ptr`, procedure pointers — and an overload, `var` and a generic all see a pointer. Against two *pointer* overloads it matches both equally and the call is an error; `cptr( zstring ptr, nullptr )` names the one you mean. This book writes `nullptr` everywhere.

Arithmetic counts elements

examples/ch11/arithmetic.tiko

```
'' arithmetic.tiko -- pointer arithmetic counts elements, not bytes.

dim as int32 a( 0 to 4 ) = { 10, 20, 30, 40, 50 }
dim as int32 ptr p = @a( 0 )

print "p[2] ="; p[2]; ",  *( p + 3 ) ="; *( p + 3 )

'' How far does + 1 move? It depends on what the pointer points at.
dim as uint8 ptr b = cast( uint8 ptr, p )
dim as any ptr v = p

print "int32 ptr + 1 moves"; cast( integer, p + 1 ) - cast( integer, p )
print "uint8 ptr + 1 moves"; cast( integer, b + 1 ) - cast( integer, b )
print "any ptr   + 1 moves"; cast( integer, v + 1 ) - cast( integer, v )

'' Walking a run of elements with a pointer.
dim as int32 ptr last_ = @a( 4 )
dim as int32 ptr q = p
print "walked:";
do while q <= last_
    print *q;
    q += 1
```

```
loop
print
```

examples/ch11/arithmetic.out

```
p[2] = 30,  *( p + 3 ) = 40
int32 ptr + 1 moves 4
uint8 ptr + 1 moves 1
any ptr   + 1 moves 1
walked: 10 20 30 40 50
```

`p + 1` does not mean "one byte later". It means **the next element**, so on an `int32 ptr` it moves four bytes, on a `uint8 ptr` one, and on an `any ptr` — which has no element size — one. `p[2]` is `*( p + 2 )`, which is why a pointer to an array's first element can be indexed like the array.

PITFALL

`@v + 3` is not three bytes past `v`. If `v` is an `int32`, `@v` is an `int32 ptr`, and `+ 3` moves three *`int32`s* — twelve bytes, well outside `v`. Reading there returns whatever happens to be next in memory, and nothing reports it.

To step in bytes, step through a byte pointer: `cast( uint8 ptr, @v ) + 3`. The `peekpoke.tiko` listing later in this chapter does exactly that — and its first draft did not, which is how this pitfall got its example.

Comparison works as you would expect — `q <= last_` walks a run of elements — and subtracting two pointers into the same array gives the distance between them in elements. Comparing pointers into *different* blocks of memory tells you nothing useful.

Changing what a pointer points at

examples/ch11/conversions.tiko

```
'' conversions.tiko -- any ptr, and changing what a pointer points at.

dim as int32 n = 70000

'' any ptr holds an address of anything. Every pointer converts to
'' it, and it converts back to any pointer type, without a cast.
dim as any ptr raw = @n
dim as int32 ptr back = raw
print "through any ptr and back:"; *back
```

```
'' Between two typed pointers, say what you mean with cptr or cast.
dim as uint8 ptr bytes = cptr( uint8 ptr, @n )
print "the bytes of 70000:";
for i as int32 = 0 to 3 : print " "; bytes[i]; : next
print

dim as uint16 ptr halves = cast( uint16 ptr, @n )
print "as two uint16: "; halves[0]; " "; halves[1]
print "and 4464 + 1 * 65536 = 70000 -- the low half comes first"
```

examples/ch11/conversions.out

```
through any ptr and back: 70000
the bytes of 70000: 112 17 1 0
as two uint16: 4464 1
and 4464 + 1 * 65536 = 70000 -- the low half comes first
```

`any ptr` is an address with no type attached. Every pointer converts to it, and it converts back to any pointer type without a cast — which is what lets one procedure accept the address of anything, and what `allocate` returns.

Between two *typed* pointers the compiler wants you to say what you mean. `cptr( T ptr, p )` and `cast( T ptr, p )` reinterpret the address; without one, assigning a `float64 ptr` from an `int32 ptr` draws `TKO0830: suspicious pointer assignment` — a warning, because sometimes it is exactly what you want, and usually it is a bug.

The listing reads the four bytes of 70000 and then the two 16-bit halves. Both show the low part first: on every target in this book the least significant byte has the lowest address.

> **NOTE**
> **An `any ptr` cannot be dereferenced.** It has no element type, so there is nothing defined to read: `*a` and `a[ i ]` are refused with `TKO0933`. Convert to a typed pointer first: `*cptr( int32 ptr, a )`.

const on either side

examples/ch11/constptr.tiko

```
'' constptr.tiko -- const on either side of ptr.
dim as int32 n = 5, m = 6

dim as const int32 ptr readOnly = @n    '' can move; cannot write
dim as int32 const ptr fixed_ = @n      '' cannot move; can write
```

```
readOnly = @m                              '' fine
*fixed_ = 9                                '' fine

*readOnly = 1                              '' refused
fixed_ = @m                                '' refused
```

```
constptr.tiko(11,1): error TKO0617: cannot modify 'fixed_': it is a
constant
     |
  11 | fixed_ = @m                               '' refused
     | ^
constptr.tiko(10,1): error TKO0617: cannot modify '*readonly': it is a
constant
     |
  10 | *readOnly = 1                             '' refused
     | ^
```

`const int32 ptr` is a pointer *to* constant data: it may be pointed elsewhere, but not written through. `int32 const ptr` is a constant pointer: it always points at the same place, but what is there may change. Read the declaration from the right — "a const pointer to an int32", "a pointer to a const int32" — and the rule falls out.

Asking for memory

examples/ch11/buffer.tiko

```
'' buffer.tiko -- memory the program asks for, and gives back.

'' allocate returns raw memory: uninitialised, and yours to free.
dim as int32 ptr p = allocate( 4 * sizeof( int32 ) )
if p = nullptr then
    print "out of memory"
    end 1
end if

for i as int32 = 0 to 3 : p[i] = ( i + 1 ) * 10 : next
print "allocated:"; p[0]; p[1]; p[2]; p[3]

'' reallocate grows (or shrinks) a block, keeping its contents. It
'' may MOVE the block, so the old pointer must not be used again.
dim as int32 ptr grown = reallocate( p, 6 * sizeof( int32 ) )
if grown = nullptr then
    deallocate( p )             '' on failure the old block survives
    end 1
```

```
end if
p = grown
p[4] = 50 : p[5] = 60
print "reallocated:"; p[0]; p[3]; " and new"; p[4]; p[5]
deallocate( p )
p = nullptr                       '' so nothing can use it by mistake

'' callocate zeroes the memory it returns.
dim as int32 ptr z = callocate( 4, sizeof( int32 ) )
print "callocated:"; z[0]; z[1]; z[2]; z[3]

'' clear sets every BYTE to a value -- here 255, so each int32 is -1.
clear( *z, 255, 4 * sizeof( int32 ) )
print "after clear( 255 ): "; z[0]; " "; z[3]
deallocate( z )

deallocate( nullptr )             '' freeing null does nothing, safely
print "done"
```

examples/ch11/buffer.out

```
allocated: 10 20 30 40
reallocated: 10 40 and new 50 60
callocated: 0 0 0 0
after clear( 255 ): -1 -1
done
```

Call	Gives
`allocate( bytes )`	uninitialised memory, or `nullptr` if there is none
`callocate( count, size )`	the same, zeroed
`reallocate( p, bytes )`	the block resized, contents kept — **possibly at a new address**
`deallocate( p )`	the memory back; `deallocate( nullptr )` does nothing
`clear( dest, byte, count )`	`count` bytes at `dest` set to `byte`

Three habits are in that listing, and each prevents a class of bug:

- **Check for null.** Allocation can fail, and a null pointer used as a buffer fails much later and much less clearly.
- **Keep `reallocate`'s result in a new variable.** On failure it returns null *and leaves the old block alone*; writing the result straight over the only pointer you had leaks that block.

- **Set the pointer to `nullptr` after `deallocate`.** A freed pointer still holds an address, and using it is the worst kind of bug: it usually works.

`clear` sets **bytes**, so `clear( *z, 255, … )` makes each `int32` −1, not 255. It is for zeroing and for byte buffers, not for filling an array with a value.

Freeing on every way out

Those habits make each call correct. They do not make the procedure correct: a procedure with three `return`s after its `allocate` needs three `deallocate`s, and the fourth `return` someone adds later leaks. `defer`, from Chapter 6, writes the release once, beside the allocation, and runs it on every way out of the block:

`examples/ch11/deferfree.tiko`

```
'' deferfree.tiko -- defer pairs each allocate with its deallocate.

'' The sum of the squares 1, 4, 9 ... of the first n numbers, built in
'' a scratch buffer. Every return after the allocate frees it.
function SumOfSquares( byval n as int32 ) as int64
    if n < 1 then return 0              '' nothing allocated yet

    dim as int64 ptr squares = allocate( n * sizeof( int64 ) )
    defer deallocate( squares )         '' runs on every return below
    if squares = nullptr then return -1 '' freeing null does nothing

    for i as int32 = 0 to n - 1
        squares[i] = cast( int64, i + 1 ) * ( i + 1 )
    next
    dim as int64 total = 0
    for i as int32 = 0 to n - 1 : total += squares[i] : next
    return total
end function

'' A deferred statement reads its variables when it RUNS, not when it
'' is written: when reallocate moves the block, defer frees the new one.
function Collect( byval count as int32, byref slots as int32 ) as int32
    slots = 2
    dim as int32 ptr items = allocate( slots * sizeof( int32 ) )
    defer deallocate( items )
    if items = nullptr then return -1

    for i as int32 = 0 to count - 1
        if i = slots then
            dim as int32 wanted = slots * 2
            dim as int32 ptr grown = _
                reallocate( items, wanted * sizeof( int32 ) )
```

```
            if grown = nullptr then return -1 '' items still freed
            items = grown
            slots = wanted
        end if
        items[i] = i * i
    next

    dim as int32 total = 0
    for i as int32 = 0 to count - 1 : total += items[i] : next
    return total
end function

print "Sum of squares through a scratch buffer"
print "  n = 0:  " & SumOfSquares( 0 )
print "  n = 3:  " & SumOfSquares( 3 )
print "  n = 10: " & SumOfSquares( 10 )

dim as int32 slots
dim as int32 total = Collect( 5, slots )
print "Collect 5 squares, starting with 2 slots"
print "  total " & total & " in " & slots & " slots"
```

examples/ch11/deferfree.out

```
Sum of squares through a scratch buffer
  n = 0:  0
  n = 3:  14
  n = 10: 385
Collect 5 squares, starting with 2 slots
  total 30 in 8 slots
```

- **Write the `defer` on the line after the `allocate`**, before the check for null. `deallocate( nullptr )` does nothing, so the failure path is covered too, and the pair cannot drift apart.
- **A deferred statement reads its variables when it runs**, at the end of the block, not when the `defer` line is reached. That is why `Collect` can store `reallocate`'s result in `items` and still free the right block. It also means a block you free early by hand needs its pointer set to `nullptr` afterwards, or the `defer` frees it a second time.
- **Allocate and defer in the procedure's own body**, never inside an `if`. Chapter 6's pitfall applies with full force: the block is freed at `end if`, and the pointer still holds its address.

Typed allocation

examples/ch11/newdelete.tiko

```
'' newdelete.tiko -- typed allocation.

'' new knows the type: no sizeof, no cast, and the memory is zeroed.
dim as int32 ptr one = new int32( 42 )
print "new int32( 42 ):"; *one
delete one

dim as float64 ptr several = new float64[ 3 ]
several[0] = 1.5 : several[2] = 2.5
print "new float64[ 3 ]:"; several[0]; several[1]; several[2]
delete[] several                '' [] matches the [] in new

type Point
    x as int32
    y as int32
end type

dim as Point ptr pt = new Point
print "new Point starts at"; pt->x; pt->y
pt->x = 3 : pt->y = 4
print "and is now"; pt->x; pt->y
delete pt
```

examples/ch11/newdelete.out

```
new int32( 42 ): 42
new float64[ 3 ]: 1.5 0 2.5
new Point starts at 0 0
and is now 3 4
```

`new T` allocates one `T`, zeroed; `new T[ n ]` allocates `n` of them. No `sizeof`, no cast, and the pointer comes back typed. Pair them exactly: memory from `new` goes back with `delete`, memory from `new[]` with `delete[]`, and memory from `allocate` with `deallocate`. Mixing the families is undefined, and Chapter 12 shows the version where it visibly breaks — once a type has a destructor, `delete` runs it and `deallocate` does not.

`new` takes a value too. For a type with no constructor the arguments fill the fields in order, as `dim … = ( 3, 4 )` does, and any left out are zero:

examples/ch11/newargs.tiko

```
'' newargs.tiko -- new with a field list.
''
'' For a type with no constructor, the arguments to new fill its
```

```
'' fields in order, as they do in a dim. This prints 3 4.

type Point
    x as int32
    y as int32
end type

dim as Point ptr pt = new Point( 3, 4 )
print pt->x; pt->y
delete pt
```

examples/ch11/newargs.out

```
 3 4
```

`new int32( 42 )` works the same way for a single value. A type with a constructor takes its arguments through that instead — Chapter 12.

Structures that link

examples/ch11/linkednodes.tiko

```
'' linkednodes.tiko -- a list built on the heap, walked, and freed.

type Node
    value as int32
    next_ as Node ptr
end type

'' Push a value on the front. The new node points at the old head,
'' and becomes the head.
function Push( byval head as Node ptr, byval v as int32 ) as Node ptr
    dim as Node ptr n = new Node
    n->value = v
    n->next_ = head
    return n
end function

dim as Node ptr head = nullptr
for i as int32 = 1 to 5
    head = Push( head, i * i )
next

print "the list:";
dim as Node ptr p = head
do while p <> nullptr
```

```
    print p->value;
    p = p->next_
loop
print

'' Free every node. Read next_ BEFORE deleting the node it is in.
dim as int32 freed = 0
do while head <> nullptr
    dim as Node ptr after = head->next_
    delete head
    head = after
    freed += 1
loop
print "freed"; freed; " nodes"
```

examples/ch11/linkednodes.out

```
the list: 25 16 9 4 1
freed 5 nodes
```

A linked list is the canonical heap structure: each node allocated separately, each pointing at the next, the end marked by a null. `Push` puts a new node at the front in constant time, which is why the list prints in reverse order of insertion.

Freeing it shows the rule that matters for every linked structure: **read the link before you delete the node that holds it.** `head->next_` after `delete head` reads freed memory. It will often produce the right answer, which is exactly why it ships.

References

examples/ch11/references.tiko

```
'' references.tiko -- a second name for an existing variable.

'' A byref parameter is a reference the caller sets up.
sub Double_( byref n as int32 )
    n *= 2
end sub

dim as int32 a = 21
Double_( a )
print "after Double_:"; a

'' dim byref makes a reference inside a procedure or module. It must
'' be bound when it is declared, and it stays bound to that variable.
dim as int32 b = 5
```

```
dim byref as int32 alias_ = b
alias_ = 9
print "b ="; b; ", same address: "; cbool( @alias_ = @b )

'' A reference reads like a variable; a pointer reads like an
'' address. Same machine code, different promises.
dim as int32 ptr ptr_ = @b
*ptr_ = 11
print "b ="; b; " via a pointer, and"; alias_; " via the reference"
```

examples/ch11/references.out

```
after Double_: 42
b = 9, same address: true
b = 11 via a pointer, and 11 via the reference
```

A **reference** is a second name for an existing variable. `byref` parameters are references the caller sets up; `dim byref` makes one directly, and it must be bound when declared — there is no null reference and no rebinding.

Underneath, a reference and a pointer are the same machine address. The difference is the promise: a reference always refers to something, and is used like the variable it names. Prefer a reference whenever "nothing" is not a valid answer, and a pointer when it is.

The memory behind a string

examples/ch11/emptystrptr.tiko

```
'' emptystrptr.tiko -- an empty string has no characters to point at.

dim as ustring full = "hi"
dim as ustring empty_ = ""

print "strptr( full )   is null: "; cbool( strptr( full ) = nullptr )
print "strptr( empty_ ) is null: "; cbool( strptr( empty_ ) = nullptr )

'' Emptying a string by assignment releases its characters too.
dim as ustring was = "something"
was = ""
print "after was = """, strptr is null: "; _
      cbool( strptr( was ) = nullptr )
```

examples/ch11/emptystrptr.out

```

```

```
strptr( full )   is null: false
strptr( empty_ ) is null: true
after was = "", strptr is null: true
```

`strptr( s )` is the address of a string's characters, which is what a C function wants. A string owns that memory, and moves or frees it whenever it changes — so the pointer is valid only until the next modification of the string, and only while the string exists.

PITFALL

`strptr` of an empty string is null. An empty string owns no characters, so there is no address to give. Assigning `""` releases the characters a string had, so this applies to a string that was emptied, not only one that started empty.

A C function handed that pointer as "a string" will usually crash on it. Check for an empty string at the boundary and pass a pointer to a real empty literal instead — Chapter 29 shows the pattern.

Reading and writing raw memory

examples/ch11/peekpoke.tiko

```
'' peekpoke.tiko -- reading and writing a value of a stated type at
'' an address.

dim as int32 v = &h01020304

'' The lowest-addressed byte is the least significant: little-endian.
'' @v is an int32 ptr, so @v + 3 would be 12 bytes on. Step in bytes
'' through a uint8 ptr instead.
dim as uint8 ptr at = cast( uint8 ptr, @v )
print "byte 0 = "; peek( uint8, at ); _
      "  byte 3 = "; peek( uint8, at + 3 )

poke uint8, @v, &hFF                '' overwrite one byte
print "after poking byte 0: &h"; hex( v, 8 )

poke int32, @v, 7                   '' overwrite all four
print "after poking an int32:"; v
```

examples/ch11/peekpoke.out

```
```

```
byte 0 = 4  byte 3 = 1
after poking byte 0: &h010203FF
after poking an int32: 7
```

`peek( T, address )` reads a `T` at an address, and `poke T, address, value` writes one. They are the explicit spellings of `*cptr( T ptr, address )`, useful when the type at an address is decided by the code rather than by a declaration — parsing a binary format, for instance.

The output confirms what `conversions.tiko` showed: byte 0 of `&h01020304` is 4. These targets are little-endian.

When the pointer is null

examples/ch11/nullcheck.tiko

```
'' nullcheck.tiko -- following a null pointer, under -exx.
#cmdline "-exx"

dim as int32 ptr p = nullptr
print *p
print "not reached"
```

examples/ch11/nullcheck.out

```

Aborting due to runtime error 7 (null pointer access) at line 5 of
nullcheck.tiko::()
```

Under `-exx`, reading or writing through a null pointer with * stops the program with runtime error 7, and the process exits with status 7, instead of crashing somewhere further on. Without `-exx` the same program crashes with an access violation.

The same check covers `->` and indexing — `p->x` and `p[i]` through null stop with error 7 too. That is the common case: a null object pointer is reached through `->` far more often than through *.

examples/ch11/nullmember.tiko

```
'' nullmember.tiko -- a field through a null pointer, under -exx.
''
'' -exx checks p->x as it checks *p, and stops with the same runtime
'' error 7 that nullcheck.tiko shows.
#cmdline "-exx"

type Point
    x as int32
```

```
end type

dim as Point ptr p = nullptr
print p->x
print "not reached"
```

examples/ch11/nullmember.out

```
Aborting due to runtime error 7 (null pointer access) at line 12 of
nullmember.tiko::()
```

> **NOTE**
> In both listings the error is the only output. A `print` placed before the bad access can appear **after** the error message when output is redirected, because the message goes to the error stream immediately and the `print` waits in a buffer. Debugging with a print just before a crash needs that in mind.

What it costs

- **A pointer costs 8 bytes and one indirection** on every access. Following a chain of them — `a->b->c` — is a chain of dependent memory loads, which is slower than walking an array.
- **Every allocation has overhead** in time and bookkeeping. A thousand small `new`s cost more than one `new[]` of a thousand.
- **A linked list scatters its nodes** across memory, so walking it misses the cache where an array would not. Use one when you need constant-time insertion in the middle, not by default.
- **Memory you allocate is memory you must free.** Every exit path, including errors. `defer` covers the paths out of one procedure; Chapter 22 covers memory that outlives it.

Summary

- `T ptr` is an address; `@` takes one, `*` follows it, `->` reaches a field, and `nullptr` is null.
- **Write `nullptr`, not `0`.** Both are null, but `0` is an integer to an overload, `var` and a generic.
- **Pointer arithmetic counts elements**: `p + 1` moves `sizeof( T )` bytes. Step in bytes through a `uint8 ptr`.
- `any ptr` converts to and from every pointer type; between typed pointers use `cptr` or `cast`.
- **An `any ptr` is not dereferenced** (`TKO0933`); convert it to a typed pointer first.

- `const T ptr` cannot write through; `T const ptr` cannot be moved.
- `allocate`/`callocate`/`reallocate`/`deallocate`: **check for null, keep `reallocate`'s result separately, null the pointer after freeing.** `clear` sets bytes.
- **`defer deallocate( p ) on the line after the allocate`** frees on every way out. It reads `p` when it runs, so it follows a `reallocate`.
- `new`/`delete` and `new[]`/`delete[]` allocate typed and zeroed memory — **pair each family with its own release**. `new T( a, b )` fills a constructor-less type's fields in order.
- **Read a node's link before deleting the node.**
- References are bound once and never null; prefer them when "nothing" is not a valid answer.
- **`strptr` of an empty string is null**, and any `strptr` is valid only until the string changes.
- `peek`/`poke` read and write a stated type at an address. The targets are little-endian.
- **`-exx` traps `*p`, `p->x` and `p[i]` through null** with runtime error 7. Still test a pointer that may be null before you follow it.

Exercises

1. Write a growable `int32` buffer that doubles its capacity when it fills, using `reallocate`, and report each growth. Append twenty items and say how many times the buffer was copied. {solution="ex11-01″}
2. Reverse a singly linked list in place by changing the links, not the values. Make sure an empty list and a one-node list both survive. {solution="ex11-02″}
3. Print the address of each element of an `int64` array and of a `uint8` array. What is the difference between consecutive addresses in each, and why?
4. Allocate a block, free it, and then read through the old pointer. Run it several times. What does it print, and why is "it printed the right value" not evidence that the code is correct?
5. Write a procedure that takes a `string` and passes `strptr` of it to a procedure expecting a `zstring ptr`, which prints the text. What must it do when the string is empty?

Objects

Chapter 10's types held data and left every rule about that data to the code around them. An account balance could be set negative from anywhere; a buffer's length could disagree with its memory; a record could be built half-filled. Nothing in the type said otherwise, because the type could not say anything.

An **object** is a type that carries its rules with it. It has procedures of its own, it decides how it is created and destroyed, and it can keep its fields to itself so that the only way to change them is through code that keeps them consistent. None of that costs anything at run time: a method is an ordinary procedure with the object passed in, and an object is exactly as large as its fields.

A type with its own procedures

examples/ch12/account.tiko

```
'' account.tiko -- a type that looks after its own data.

type Account
    declare constructor( byref owner as ustring, _
                         byval opening as int64 )
    declare destructor( )
    declare sub Deposit( byval amount as int64 )
    declare function Withdraw( byval amount as int64 ) as boolean
    declare function Balance( ) as int64

    owner_   as ustring
    balance_ as int64
end type

'' Bodies live outside the type, named Type.Member.
constructor Account( byref owner as ustring, byval opening as int64 )
    this.owner_ = owner          '' this is the object being built
    balance_ = opening           '' ... and "this." may be left off
    print "  opened "; owner_
end constructor

destructor Account( )
    print "  closed "; owner_; " with"; balance_
end destructor

sub Account.Deposit( byval amount as int64 )
    balance_ += amount
end sub
```

```
function Account.Withdraw( byval amount as int64 ) as boolean
    if amount > balance_ then return false
    balance_ -= amount
    return true
end function

function Account.Balance( ) as int64
    return balance_
end function

scope
    dim as Account ada = Account( "Ada", 100 )
    dim as Account grace = Account( "Grace", 20 )

    ada.Deposit( 50 )
    print "Ada withdraws 500: "; ada.Withdraw( 500 )
    print "Ada withdraws 30:  "; ada.Withdraw( 30 )
    print "Ada's balance:"; ada.Balance( )
end scope                          '' both destructors run here
print "after the scope"
```

examples/ch12/account.out

```
  opened Ada
  opened Grace
Ada withdraws 500: false
Ada withdraws 30:  true
Ada's balance: 120
  closed Grace with 20
  closed Ada with 120
after the scope
```

A **method** is declared inside the type and defined outside it, named `Type.Member`. Inside, the object it was called on is `this`, and `this.` may be left off when nothing else has the same name — `balance_` and `this.balance_` are the same field.

The **constructor** runs when an object is created and the **destructor** when it is destroyed. For an object in a scope, that is at the end of the scope, in the reverse order of creation: Grace was opened after Ada and closed before her. That ordering is deliberate — something created later may depend on something created earlier, never the other way round.

The trailing underscore on the fields is a convention, not syntax. It marks the data the methods look after, and keeps names free for the methods that expose it.

Constructors

examples/ch12/constructors.tiko

```
'' constructors.tiko -- several ways in, one of them implicit.

type Temp
    declare constructor( )
    declare constructor( byval c as float64 )
    declare constructor( byref text as ustring )
    celsius as float64
end type

constructor Temp( )
    celsius = 20
    print "  default constructor"
end constructor

constructor Temp( byval c as float64 )
    celsius = c
    print "  from a float64"
end constructor

'' Delegation: hand the work to another constructor, then add to it.
constructor Temp( byref text as ustring )
    this.constructor( val( text ) )
    print "  ... reached through the text constructor"
end constructor

sub Show( byref t as const Temp )
    print "  Show sees"; t.celsius
end sub

print "dim as Temp a"
dim as Temp a

print "dim as Temp b = Temp( 5.0 )"
dim as Temp b = Temp( 5.0 )

print "dim as Temp c = Temp( ""31.5"" )"
dim as Temp c = Temp( "31.5" )

'' A one-argument constructor is also a CONVERSION: the compiler will
'' call it on its own whenever a float64 is offered where a Temp is
'' wanted.
print "dim as Temp d = 7.5"
```

```
dim as Temp d = 7.5
print "Show( 12.0 )"
Show( 12.0 )

print "values:"; a.celsius; b.celsius; c.celsius; d.celsius
```

examples/ch12/constructors.out

```
dim as Temp a
  default constructor
dim as Temp b = Temp( 5.0 )
  from a float64
dim as Temp c = Temp( "31.5" )
  from a float64
  ... reached through the text constructor
dim as Temp d = 7.5
  from a float64
Show( 12.0 )
  from a float64
  Show sees 12
values: 20 5 31.5 7.5
```

A type may have as many constructors as it needs, distinguished by their parameters like overloaded procedures. The one with no parameters is the **default constructor**, used by a plain `dim`.

One constructor may hand its work to another: `this.constructor( val( text ) )` runs the `float64` constructor and then carries on. Put shared setup in one constructor and delegate to it, rather than repeating it.

PITFALL

A one-parameter constructor is also an implicit conversion. Because `Temp` has a constructor taking a `float64`, the compiler will call it on its own whenever a `float64` appears where a `Temp` is wanted — `dim as Temp d = 7.5` and `Show( 12.0 )` both did, silently, in the listing.

That is convenient for a type that genuinely *is* a number with extra meaning, and surprising for one that merely takes a number to get started. A `Buffer( 1024 )` constructor makes `Show( 1024 )` compile for a procedure that wants a buffer. Nothing turns the conversion off, so give such a constructor a second parameter, or create the object through a named function instead. There is no `explicit` keyword to switch it off: `declare explicit constructor` is refused (`TKO0936`).

Copying an object that owns something

Chapter 10's types copied themselves correctly: every field was copied, strings included. That stops being true the moment a type holds a **pointer** to memory it owns. Copying the pointer gives two objects the same block — and two destructors that each free it.

Measured, with a `Buffer` that has a constructor and destructor but nothing else, after `dim as Buffer b = a`:

```
a.data_[0] = 99   same block: true
  freeing 31008
  freeing 31008
```

The write through `b` appeared in `a`, both destructors freed the same address — the number printed differs from run to run; that it is the same number twice does not — and the process then died — `0xC0000374`, heap corruption, on Windows, and `free(): double free` on Linux. fbc behaves identically: this is the language's rule, not a defect.

The fix is to say what copying means:

examples/ch12/copying.tiko

```
'' copying.tiko -- a type that owns memory must say how to copy it.

type Buffer
    declare constructor( byval n as int32 )
    declare constructor( byref other as const Buffer )    '' copy
    declare operator let( byref other as const Buffer )   '' assign
    declare destructor( )
    items as int32 ptr
    size  as int32
end type

constructor Buffer( byval n as int32 )
    items = callocate( n, sizeof( int32 ) )
    size = n
end constructor

'' Copy construction: a brand-new object, so there is nothing to free.
constructor Buffer( byref other as const Buffer )
    size = other.size
    items = allocate( size * sizeof( int32 ) )
    for i as int32 = 0 to size - 1 : items[i] = other.items[i] : next
    print "  copy constructor"
end constructor
```

```
'' Assignment: this object already owns memory. Free it first -- but
'' not when assigning an object to itself, or the source goes too.
operator Buffer.let( byref other as const Buffer )
    if @this = @other then
        print "  let: assigned to itself, nothing to do"
        exit operator
    end if
    deallocate( items )
    size = other.size
    items = allocate( size * sizeof( int32 ) )
    for i as int32 = 0 to size - 1 : items[i] = other.items[i] : next
    print "  let"
end operator

destructor Buffer( )
    deallocate( items )
end destructor

scope
    dim as Buffer a = Buffer( 4 )
    a.items[0] = 1

    dim as Buffer b = a           '' copy constructor
    b.items[0] = 99
    print "a[0] ="; a.items[0]; "  b[0] ="; b.items[0]; _
          "  separate memory: "; cbool( a.items <> b.items )

    dim as Buffer c = Buffer( 2 )
    c = a                         '' operator let
    print "c now has size"; c.size

    c = c                         '' must not destroy itself
end scope
print "every buffer freed exactly once"
```

examples/ch12/copying.out

```
  copy constructor
a[0] = 1  b[0] = 99  separate memory: true
  let
c now has size 4
  let: assigned to itself, nothing to do
every buffer freed exactly once
```

Two procedures are involved, because there are two ways an object gets a copy:

- The **copy constructor** builds a new object from an existing one — `dim as Buffer b = a`, or passing `byval`. There is nothing to clean up first, because the object did not exist.
- **`operator let`** assigns into an object that already exists — `c = a`. It must release what the object holds before taking the copy, and it must **do nothing when assigned to itself**, or it frees the very data it is about to copy.

PITFALL

The rule of three: a type that needs a destructor almost always needs a copy constructor and an `operator let` as well. The destructor is there because the type owns something; the copy operations decide what happens to that something when the object is duplicated. Write one without the others and every copy is a double free waiting for its scope to end.

The better answer, where it is available, is the **rule of zero**: build the type from fields that already copy themselves — strings, other well-behaved objects — so that it needs none of the three. The string builder later in this chapter does exactly that.

Properties

examples/ch12/properties.tiko

```
'' properties.tiko -- methods that read like fields.

type Thermostat
    declare property Celsius( ) as float64                '' get
    declare property Celsius( byval c as float64 )        '' set
    declare property Fahrenheit( ) as float64             '' get only
    declare property Reading( byval i as int32 ) as float64
    declare property Reading( byval i as int32, byval v as float64 )
  private:
    c_ as float64
    readings( 0 to 2 ) as float64
end type

property Thermostat.Celsius( ) as float64
    return c_
end property

'' The setter is where a type defends its own rules.
property Thermostat.Celsius( byval c as float64 )
    if c < -273.15 then c = -273.15
    c_ = c
```

```
end property

property Thermostat.Fahrenheit( ) as float64
    return c_ * 9 / 5 + 32
end property

property Thermostat.Reading( byval i as int32 ) as float64
    return readings( i )
end property

property Thermostat.Reading( byval i as int32, byval v as float64 )
    readings( i ) = v
end property

dim as Thermostat t
t.Celsius = 21.5
print "celsius"; t.Celsius; ", fahrenheit"; t.Fahrenheit

t.Celsius = -500
print "asked for -500, got "; t.Celsius

t.Reading( 1 ) = 3.5                '' an indexed property
print "reading 1 is"; t.Reading( 1 )
```

examples/ch12/properties.out

```
celsius 21.5, fahrenheit 70.7
asked for -500, got -273.15
reading 1 is 3.5
```

A **property** is a pair of methods that read and write like a field. `t.Celsius = 21.5` calls the setter; `t.Celsius` in an expression calls the getter. The point is the setter: it is where the type enforces its own rules, so asking for −500 °C gives absolute zero rather than an impossible temperature.

A property with only a getter is read-only — `Fahrenheit` is computed from the same field rather than stored, so it can never disagree with `Celsius`. A property may take an index, as `Reading( i )` does, which reads like an array whose elements the type controls.

A property with only a **setter** is write-only: reading it is refused, `TK00934`.

`t.Celsius += 2.5` works on a property with both accessors: it calls the getter, applies the operator and calls the setter. So does `swap`, and a property passed to a `byref` parameter is written back through its setter when the call returns.

Auto-properties

Most properties only store a value. An **auto-property** says so in its declaration, by ending it with its accessors; the compiler writes both over a hidden field:

examples/ch12/autoprops.tiko

```
type Person
     declare property Name as ustring get, set             '' read and
written like a field
    declare property Age as int32 = 18 get, private set '' anyone reads,
only Person writes
    declare property Id as int32 get, init               '' set once, by
the constructor
    declare constructor( byval newId as int32 )
    declare sub Birthday( )
end type

constructor Person( byval newId as int32 )
    Id = newId
end constructor

sub Person.Birthday( )
    Age += 1
end sub

dim as Person p = Person( 42 )
p.Name = "Ann"
p.Birthday( )
print p.Name; ", age"; p.Age; ", id"; p.Id
```

examples/ch12/autoprops.out

```
Ann, age 19, id 42
```

`get, set` behaves like a plain field. `get, private set` can be read anywhere but written only by the type's own code — the setter may be less visible than the getter, never more. `get, init` (or `get` alone) can be assigned only by a constructor, which makes an identifier that never changes after the object is built. `= 18` gives the hidden field its first value. An auto-property has no code of its own: as soon as the setter has a rule to keep, write an ordinary property.

Property blocks

A **property block** keeps the getter and setter together in one definition:

examples/ch12/propblock.tiko

```
type Thermostat
    declare property Celsius( ) as float64
    declare property Celsius( byval value as float64 )
    declare property Fahrenheit( ) as float64
  private:
    celsius_ as float64
end type

'' Both accessors in one block. In set, value is what was assigned;
'' in get, value is what will be returned.
property Thermostat.Celsius as float64
    get
        value = celsius_
    end get
    set
        if value < -273.15 then value = -273.15
        celsius_ = value
    end set
end property

property Thermostat.Fahrenheit as float64
    get
        return celsius_ * 9 / 5 + 32
    end get
end property

dim as Thermostat t
t.Celsius = 21.5
t.Celsius += 2.5                    '' get, add, set
print "celsius"; t.Celsius; ", fahrenheit"; t.Fahrenheit
t.Celsius = -500
print "asked for -500, got "; t.Celsius
```

examples/ch12/propblock.out

```
celsius 24, fahrenheit 75.2
asked for -500, got -273.15
```

Inside both sections, `value` is the value crossing the boundary: in `set` the value being assigned, which may be adjusted before it is stored; in `get` the result, which starts at the type's default and is returned at `end get`. `return` works too, and either section may be left out. `get` and `set` open a section only when they stand alone on their line, so `get #1, , buffer` in an ordinary getter is still the file statement.

A property may also take more than one index, be `static` (`Grid.Created`, belonging to the type), or live in a namespace with no object at all. These forms, and the auto-property and block above, are tiko's own: FreeBASIC has none of them (Appendix D).

Keeping fields private

Everything after `private:` in a type is reachable only from the type's own methods:

`examples/ch12/access.tiko`

```
'' access.tiko -- private means private.
type Thermostat
    declare property Celsius( ) as float64
  private:
    c_ as float64
end type

property Thermostat.Celsius( ) as float64
    return c_
end property

dim as Thermostat t
t.c_ = 5                            '' refused: c_ is private
```

```
access.tiko(13,2): error TKO0625: 'c_' is not accessible here
     |
  13 | t.c_ = 5                            '' refused: c_ is private
     |  ^
```

That refusal is what made the setter's rule real. Without it, `t.c_ = -500` would bypass the clamp and the type's guarantee would be a suggestion. `public:` reopens access, and `protected:` — private to everyone except types derived from this one — matters from Chapter 14.

Access applies to the calls the compiler makes for you as well as the ones you write:

`examples/ch12/privatector.tiko`

```
'' privatector.tiko -- access applies to the calls you never write.
type Fortress
    x as int32
  private:
    declare constructor( )
    declare destructor( )
end type
```

```
constructor Fortress( )
end constructor

destructor Fortress( )
end destructor

dim as Fortress f                '' needs both, and may call neither
```

```
privatector.tiko(15,17): error TKO0633: 'f' needs a constructor or
destructor that is not accessible here
     |
  15 | dim as Fortress f                '' needs both, and may call
neither
     |                  ^
privatector.tiko(15,17): error TKO0633: 'f' needs a constructor or
destructor that is not accessible here
     |
  15 | dim as Fortress f                '' needs both, and may call
neither
     |                  ^
```

A plain `dim` calls the constructor and, at the end of the scope, the destructor — so a type with both private cannot be declared from outside at all. The message appears once for each. Making a constructor private is how a type insists on being created through a function of its own choosing.

Members that belong to the type

examples/ch12/statics.tiko

```
'' statics.tiko -- members that belong to the type, and methods that
'' promise not to change the object.

type Ticket
    declare constructor( )
    declare static function Issued( ) as int32
    declare const function Describe( ) as ustring
    static issued_ as int32          '' one, shared by every Ticket
    number_ as int32                 '' one per Ticket
end type

dim as int32 Ticket.issued_ = 0      '' a static field is defined once

constructor Ticket( )
```

```
    issued_ += 1
    number_ = issued_
end constructor

'' A static method has no this: it is called on the type.
static function Ticket.Issued( ) as int32
    return issued_
end function

'' A const method may read the object but not change it.
const function Ticket.Describe( ) as ustring
    return "ticket #" & number_
end function

dim as Ticket a, b, c
print "issued so far:"; Ticket.Issued( )
print b.Describe( )
print c.Describe( )
```

examples/ch12/statics.out

```
issued so far: 3
ticket #2
ticket #3
```

A **`static` field** exists once, shared by every object of the type, and needs a definition outside the type — `dim as int32 Ticket.issued_ = 0` — in exactly one place. A **`static` method** has no `this` and is called on the type itself, `Ticket.Issued( )`.

A **`const` method** promises not to change the object. It may read fields and call other `const` methods, and the compiler holds it to the promise:

examples/ch12/constmethod.tiko

```
'' constmethod.tiko -- a const method that tries to change the object.
type Tally
    declare const sub Bump( )
    n as int32
end type

const sub Tally.Bump( )
    n += 1                              '' refused: Bump promised not to
end sub
```

```
constmethod.tiko(8,5): error TKO0664: 'n' cannot be written from a const
method
```

```
     |
   8 |     n += 1                                '' refused: Bump promised
not to
     |     ^
```

Objects on the heap

examples/ch12/heapobjects.tiko

```
'' heapobjects.tiko -- objects that do not live in a scope.

type Noisy
    declare constructor( )
    declare constructor( byval n as int32 )
    declare destructor( )
    n as int32
end type

constructor Noisy( )
    n = -1
    print "  default constructor"
end constructor

constructor Noisy( byval n as int32 )
    this.n = n
    print "  constructor"; n
end constructor

destructor Noisy( )
    print "  destructor"; n
end destructor

'' new runs the constructor; delete runs the destructor, then frees.
print "new and delete:"
dim as Noisy ptr one = new Noisy( 7 )
delete one

'' An array in a scope constructs every element, and destroys them
'' all when the scope ends.
print "an array in a scope:"
scope
    dim as Noisy pair( 1 to 2 )
    pair( 2 ).n = 20
end scope
```

```
'' Placement new builds an object in memory you already have, and you
'' must run the destructor yourself before freeing that memory.
print "placement new:"
dim as any ptr raw = allocate( sizeof( Noisy ) )
dim as Noisy ptr placed = new( raw ) Noisy( 42 )
print "  placed->n ="; placed->n
placed->destructor( )
deallocate( raw )
print "done"
```

examples/ch12/heapobjects.out

```
new and delete:
  constructor 7
  destructor 7
an array in a scope:
  default constructor
  default constructor
  destructor 20
  destructor-1
placement new:
  constructor 42
  placed->n = 42
  destructor 42
done
```

`new T( … )` allocates and runs the matching constructor; `delete` runs the destructor and then frees. An array of objects in a scope constructs every element, using the default constructor, and destroys them all — last first — when the scope ends.

Placement `new` — `new( address ) T( … )` — runs a constructor in memory you already have, without allocating. It is for memory pools and for objects that must live at a particular address. Because nothing was allocated, `delete` is wrong here: you call the destructor yourself, explicitly, and free the memory the way you got it.

`new T[ n ]` and `delete[]` do the same for an array on the heap: a constructor for each element, then a destructor for each, last first.

examples/ch12/deletearray.tiko

```
'' deletearray.tiko -- delete[] on an array of objects.
''
'' new[] runs a constructor for each element. delete[] must run the
'' destructor for each, or everything the objects own is leaked.

type Noisy
```

```
    declare constructor( )
    declare destructor( )
    n as int32
end type

constructor Noisy( )
    n = -1
    print "  constructor"
end constructor

destructor Noisy( )
    print "  destructor"; n
end destructor

dim as Noisy ptr three = new Noisy[ 3 ]
three[1].n = 10
delete[] three
print "done"
```

examples/ch12/deletearray.out

```
  constructor
  constructor
  constructor
  destructor-1
  destructor 10
  destructor-1
done
```

`delete[]` knows how many elements there are because, for a type with a destructor, `new[]` recorded the count just before the first one. That is one more reason the families must not be mixed: `delete` or `deallocate` on such memory starts at the wrong address.

A builder that needs none of the three

examples/ch12/stringbuilder.tiko

```
'' stringbuilder.tiko -- building text without copying it every time.
''
'' Joining with &= in a loop copies everything built so far on every
'' pass. A builder keeps spare room and copies only when it runs out.

type StringBuilder
    declare sub Add( byref piece as string )
```

```
    declare property Length( ) as int32
    declare property Capacity( ) as int32
    declare function Text( ) as string
  private:
    buf_ as string                '' the room, mostly unused
    len_ as int32                 '' how much of it holds text
end type

sub StringBuilder.Add( byref piece as string )
    dim as int32 n = len( piece )
    if n = 0 then exit sub

    '' Out of room: at least double it, so growth stays rare.
    if len_ + n > len( buf_ ) then
        dim as int32 grow = iif( len( buf_ ) = 0, 16, len( buf_ ) )
        do while len_ + n > len( buf_ ) + grow : grow *= 2 : loop
        buf_ &= space( grow )
    end if

    mid( buf_, len_ + 1, n ) = piece  '' overwrite in place: no copy
    len_ += n
end sub

property StringBuilder.Length( ) as int32
    return len_
end property

property StringBuilder.Capacity( ) as int32
    return len( buf_ )
end property

function StringBuilder.Text( ) as string
    return left( buf_, len_ )
end function

dim as StringBuilder sb
for i as int32 = 1 to 12
    sb.Add( "item" & i & ";" )
next
print "length"; sb.Length; ", capacity"; sb.Capacity
print left( sb.Text( ), 40 ); "..."

'' Every field copies itself correctly, so the type needs no copy
'' constructor, no operator let and no destructor of its own.
```

```
dim as StringBuilder copy_ = sb
copy_.Add( "more" )
print "original"; sb.Length; ", copy"; copy_.Length
```

examples/ch12/stringbuilder.out

```
length 75, capacity 128
item1;item2;item3;item4;item5;item6;item...
original 75, copy 79
```

Joining text with `&=` in a loop copies everything built so far on every pass. `StringBuilder` keeps spare room in its buffer, writes each piece into it with Chapter 9's `mid` statement — which overwrites in place and copies nothing — and grows by doubling, so growth is rare.

The copy at the end is the point. `StringBuilder` owns a buffer, and yet it has no copy constructor, no `operator let` and no destructor — because its buffer is a `string`, which already knows how to copy and free itself. That is the rule of zero in practice: let each field manage its own resource, and the type composed of them needs no management at all.

What it costs

- **A method call is a procedure call** with the object passed by reference. No lookup, no indirection — until Chapter 14 adds `virtual`, which adds one.
- **An object is the size of its fields.** Methods, properties and `static` fields add nothing to each instance.
- **A property is a call**, but a short one that the optimiser usually inlines. Use them freely.
- **A copy constructor runs on every `byval` pass and every return.** For a type that owns memory, that is an allocation each time — pass it `byref ... as const`.
- **Constructors and destructors run in scope order** — free at run time, but only correct if they are written to be.

Summary

- Methods are declared in the type and defined as `Type.Member`; `this` is the object, and `this.` is optional.
- Constructors run on creation, destructors at the end of the scope, **in reverse order of creation**.
- Constructors overload; `this.constructor( … )` delegates. **A one-parameter constructor is an implicit conversion**, with no way to switch it off.

- **Rule of three:** a type with a destructor needs a copy constructor and an `operator let`, and the `let` must survive self-assignment. **Rule of zero:** better still, build from fields that copy themselves.
- Properties are methods that read like fields; the setter enforces the type's rules. A property with only a setter cannot be read (`TKO0934`). An auto-property (`get, private set`, `get, init`) writes both accessors; a property block keeps them in one definition, with `value` crossing the boundary.
- `private:` fields are reachable only from the type's own methods; access is enforced on the constructor and destructor calls the compiler makes too.
- `static` fields are shared and defined once; `static` methods have no `this`.
- `const` methods may not change the object, and a `const` object may call only `const` methods (`TKO0935`).
- `new`/`delete` run constructor and destructor; placement `new` constructs in existing memory, and you call the destructor yourself. `new[]`/`delete[]` run them for every element.

Exercises

1. Write an `IntStack` that keeps its items in memory from `reallocate`, with a constructor, destructor, copy constructor and `operator let`. Show that a copied stack and an assigned stack are both independent of the original. Route both copy paths through one private helper. {solution="ex12-01″}
2. Write a `Temperature` that stores kelvin and exposes `Celsius` and `Fahrenheit` as read-write properties, with the absolute-zero check in exactly one place. {solution="ex12-02″}
3. Give `Account` a private constructor and a `static function Create( … ) as Account` that refuses a negative opening balance. What can callers no longer do, and why is that the point?
4. Remove the self-assignment check from `copying.tiko`'s `operator let` and run `c = c`. What happens, and which line of `let` is responsible?
5. `Temp` converts silently from `float64`. Write a procedure taking a `Temp`, call it with a plain number, and decide whether that call should have compiled. How would you redesign `Temp` if not?

Operator Overloading

Chapter 10 left a gap: two `Points` could be copied but not compared, because the compiler would not guess what equality meant. Chapter 12 gave types methods, so a vector could have an `Add` method — and code that reads `a.Add( b.Scale( 2 ) )` where a mathematician would write `a + 2b`.

An **operator overload** is a procedure that the compiler calls when it sees an operator applied to your type. It is ordinary code with an unusual name, and it costs exactly what the same code costs as a method. What it buys is notation: a type that behaves like a number can be written like one.

The chapter ends with the other side of that — what not to overload — because notation that lies about what it does is worse than a method name that tells the truth.

A value type that reads like a number

examples/ch13/vec2.tiko

```
'' vec2.tiko -- a value type that reads like a number.

type Vec2
    x as float64
    y as float64
    declare operator +=( byref o as Vec2 )       '' members: they
    declare operator cast( ) as ustring          '' change or convert
end type                                         '' this object

'' Binary operators are global: neither operand is "this".
operator + ( byref a as Vec2, byref b as Vec2 ) as Vec2
    return type( a.x + b.x, a.y + b.y )
end operator

operator - ( byref a as Vec2, byref b as Vec2 ) as Vec2
    return type( a.x - b.x, a.y - b.y )
end operator

operator - ( byref a as Vec2 ) as Vec2               '' unary minus
    return type( -a.x, -a.y )
end operator

'' Scaling needs both orders, or 2 * v will not compile.
operator * ( byref a as Vec2, byval k as float64 ) as Vec2
    return type( a.x * k, a.y * k )
end operator
```

```
operator * ( byval k as float64, byref a as Vec2 ) as Vec2
    return a * k
end operator

operator = ( byref a as Vec2, byref b as Vec2 ) as boolean
    return a.x = b.x andalso a.y = b.y
end operator

operator <> ( byref a as Vec2, byref b as Vec2 ) as boolean
    return not ( a = b )
end operator

operator abs ( byref a as Vec2 ) as float64          '' the length
    return sqr( a.x * a.x + a.y * a.y )
end operator

operator Vec2.+=( byref o as Vec2 )
    x += o.x
    y += o.y
end operator

operator Vec2.cast( ) as ustring
    return "(" & x & ", " & y & ")"
end operator

dim as Vec2 a = ( 1, 2 ), b = ( 3, 4 )

print "a + b     = "; a + b
print "b - a     = "; b - a
print "-a        = "; -a
print "2 * a     = "; 2 * a
print "a + b * 2 = "; a + b * 2; "   (* still binds first)"
print "abs( b )  ="; abs( b )
print "a = b: "; a = b; ",  a <> b: "; a <> b

a += b
print "a += b  -> "; a
```

examples/ch13/vec2.out

```
a + b     = (4, 6)
b - a     = (2, 2)
-a        = (-1, -2)
2 * a     = (2, 4)
```

```
a + b * 2 = (7, 10)   (* still binds first)
abs( b )  = 5
a = b: false,  a <> b: true
a += b  -> (4, 6)
```

An operator is defined like a function whose name is the operator: `operator + ( a, b ) as Vec2`. From then on, `a + b` on two `Vec2`s is a call to it.

A few things in that listing are deliberate:

- **Scaling needs both orders.** `a * 2` and `2 * a` are two different operators — the left operand's type differs — so the second is defined in terms of the first. Leave one out and the other order does not compile.
- **Unary minus is its own operator**, with one parameter.
- **`=` and `<>` are separate.** Defining one does not define the other, and they must agree, so `<>` is written as `not ( a = b )`.
- **`abs` is overloadable**, and for a vector the natural meaning is the length.
- **Precedence does not change.** `a + b * 2` still multiplies first, because the compiler parses the expression before it decides which procedures to call.

`operator cast( ) as ustring` is what makes `print a + b` work: `print` takes the vector's conversion to text. It is also what makes `"x" & v` work, an assignment to a `ustring`, and the string functions — `ucase( v )`, `left( v, 2 )` — treat the vector as its text.

Member or global

Some operators are declared inside the type and some outside it, and the compiler enforces which:

examples/ch13/wrongplace.tiko

```
'' wrongplace.tiko -- a binary operator declared as a member.
type Vec2
    x as float64
    y as float64
    declare operator +( byref o as Vec2 ) as Vec2   '' refused
end type
```

```
wrongplace.tiko(5,13): error TKO0790: operator '+' cannot be a member of
a type
     |
   5 |     declare operator +( byref o as Vec2 ) as Vec2   '' refused
     |             ^
```

The rule follows from what each operator acts on:

Kind	Where	Why
`+ - * / \ mod ^`, comparisons, `and or xor`, unary `- not`, `abs`	global	neither operand is privileged; either side may be another type
`+=` and the other compound assignments	member	they change the object on the left
`let`, `cast`	member	they assign *into* or convert *from* this object
`[]`, `@`	member	they reach into this object
`->`, `*` (dereference)	global	they take the object as their operand
`for`, `step`, `next`	member	the loop's counter is the object
`new`, `delete`, `new[]`, `delete[]`	member	they allocate for this type

`@` is the opposite case from `+`: as a global it is refused with `TKO0697: operator '@' must be a member of a type`.

A global operator is not a method, so it sees only the public parts of the type. That is usually right — an operator built from the type's public interface cannot break its invariants — and when it is not, the operator can call a method that does the work.

Conversions in both directions

`examples/ch13/fixedpoint.tiko`

```
'' fixedpoint.tiko -- exact decimals, stored as whole thousandths.
''
'' 0.1 + 0.2 is not 0.3 in floating point. Stored as an integer count
'' of thousandths, it is exactly 300.

type Fixed
    raw as int64                                '' thousandths
    declare operator let( byval v as float64 )
    declare operator cast( ) as float64
    declare operator cast( ) as string
end type

'' let from another type: assigning a float64 rounds it once, here.
operator Fixed.let( byval v as float64 )
    raw = cint( v * 1000 )
end operator
```

```
operator Fixed.cast( ) as float64
    return raw / 1000
end operator

operator Fixed.cast( ) as string
    dim as int64 whole = raw \ 1000
    dim as int64 part = abs( raw ) mod 1000
    dim as ustring sign = iif( raw < 0 andalso whole = 0, "-", "" )
    return sign & whole & "." & right( "00" & part, 3 )
end operator

operator + ( byref a as Fixed, byref b as Fixed ) as Fixed
    dim as Fixed r
    r.raw = a.raw + b.raw
    return r
end operator

operator * ( byref a as Fixed, byref b as Fixed ) as Fixed
    dim as Fixed r
    r.raw = ( a.raw * b.raw ) \ 1000
    return r
end operator

operator < ( byref a as Fixed, byref b as Fixed ) as boolean
    return a.raw < b.raw
end operator

operator = ( byref a as Fixed, byref b as Fixed ) as boolean
    return a.raw = b.raw
end operator

dim as Fixed p, q, sum, want
p = 0.1
q = 0.2
want = 0.3
sum = p + q
print "fixed:   0.1 + 0.2 = "; sum; "   equal to 0.3: "; sum = want

dim as float64 fp = 0.1 + 0.2
print "float64: 0.1 + 0.2 ="; fp; "   equal to 0.3: "; cbool( fp = 0.3 )

dim as Fixed price, qty
price = 19.99
```

```
qty = 3
print "19.99 * 3 = "; price * qty
print "as a float64:"; cast( float64, price * qty )
dim as Fixed twenty
twenty = 20
print "19.99 < 20: "; price < twenty
```

examples/ch13/fixedpoint.out

```
fixed:   0.1 + 0.2 = 0.300   equal to 0.3: true
float64: 0.1 + 0.2 = 0.3   equal to 0.3: false
19.99 * 3 = 59.970
as a float64: 59.97
19.99 < 20: true
```

`Fixed` stores an amount as a whole number of thousandths, so 0.1 + 0.2 is exactly 300 thousandths and compares equal to 0.3. The `float64` line beside it compares unequal — although **`print` shows it as 0.3**, because `print` rounds to fifteen significant digits. The difference is real and invisible; that is the whole case for a fixed-point type.

Two conversions make it convenient:

- **`operator let( byval v as float64 )`** lets a `float64` be assigned to a `Fixed`. The rounding happens once, there, and nowhere else.
- **`operator cast( ) as float64`** lets a `Fixed` be used where a `float64` is wanted. `cast( float64, price * qty )` calls it explicitly; passing a `Fixed` to a procedure that takes a `float64` calls it implicitly.

A type may have several `cast` operators — `Fixed` has two — distinguished by the type they produce.

The built-in maths functions take it too. `sqr`, `abs`, `int` and the rest accept a type with a numeric `cast`, convert, and work on the number:

examples/ch13/castmath.tiko

```
'' castmath.tiko -- a built-in maths function given a type with a cast.
''
'' Fixed converts itself to float64, so sqr( f ) is sqr( 2.5 ):
'' 1.58113883008419.

type Fixed
    raw as int64
    declare operator cast( ) as float64
end type

operator Fixed.cast( ) as float64
```

```
    return raw / 1000
end operator

dim as Fixed f
f.raw = 2500
dim as float64 r = sqr( f )
print "sqr( f ) ="; r
```

examples/ch13/castmath.out

```
sqr( f ) = 1.58113883008419
```

A type with no numeric `cast` — and no `operator sqr` of its own — is refused: `TKO0928`.

PITFALL

A `cast` operator is an implicit conversion, like a one-parameter constructor. Chapter 12's warning applies in the other direction: once `Fixed` converts to `float64`, it converts wherever a `float64` would be accepted, including places nobody intended.

Two conversions are worse than one. Give a type casts to both `float64` and `int64`, call a procedure overloaded for both, and the call is genuinely ambiguous — there is no reason to prefer either — so it is refused, `TKO0660`. Adding a second `cast` can therefore turn a line that compiled into one that does not.

Give a type the conversions its users genuinely need, and prefer a named method — `ToFloat( )` — for the rest.

Reaching into an object

examples/ch13/indexing.tiko

```
'' indexing.tiko -- [] , let from another type, and smart handles.

type Grid
    declare constructor( byval w as int32, byval h as int32 )
    declare destructor( )
    declare operator []( byval i as int32 ) byref as int32
    declare operator let( byval fill as int32 )
    declare operator @( ) as int32 ptr
    w as int32
    h as int32
  private:
    cells as int32 ptr
end type
```

```
constructor Grid( byval w as int32, byval h as int32 )
    this.w = w
    this.h = h
    cells = callocate( w * h, sizeof( int32 ) )
end constructor

destructor Grid( )
    deallocate( cells )
end destructor

'' Returning byref is what lets g[4] appear on the LEFT of =.
operator Grid.[]( byval i as int32 ) byref as int32
    return cells[i]
end operator

'' let from an int32: "g = 7" fills every cell.
operator Grid.let( byval fill as int32 )
    for i as int32 = 0 to w * h - 1 : cells[i] = fill : next
end operator

'' @g gives the first cell, which is what a C function would want.
operator Grid.@( ) as int32 ptr
    return cells
end operator

dim as Grid g = Grid( 3, 2 )
g = 7
g[4] = 42
print "g[0], g[4], g[5] ="; g[0]; g[4]; g[5]
print "via @g:"; ( @g )[4]

'' A handle that forwards -> and * to the Grid it refers to. These
'' two are global operators, taking the handle as their operand.
type GridHandle
    target as Grid ptr
end type

operator -> ( byref hnd as GridHandle ) byref as Grid
    return *hnd.target
end operator

operator * ( byref hnd as GridHandle ) byref as Grid
    return *hnd.target
```

```
end operator

dim as GridHandle hd
hd.target = @g        '' careful: this is Grid's own @, so a cell!
print "hd.target is the cells, not the Grid: "; _
      cbool( hd.target = cast( Grid ptr, @g ) )
hd.target = varptr( g )      '' varptr is never overloaded
print "through ->: w ="; hd->w; ",  through *: h ="; ( *hd ).h
```

examples/ch13/indexing.out

```
g[0], g[4], g[5] = 7 42 7
via @g: 42
hd.target is the cells, not the Grid: true
through ->: w = 3,  through *: h = 2
```

`operator []` returning **byref** is what lets `g[4]` appear on the left of an assignment: the caller receives the element itself, not a copy of it. A `[]` that returns by value gives a read-only index.

`operator let( byval fill as int32 )` is `let` from a different type: `g = 7` fills every cell. `let` from the type's own type is the copy assignment Chapter 12 covered; `let` from anything else is a conversion into the object.

`GridHandle` forwards `->` and `*` to the `Grid` it holds, which is the shape of every "smart" handle — a type that behaves like a pointer while doing something extra, such as counting references, which Chapter 21's `SharedPtr` does.

The middle of the listing is a warning about `@`. Because `Grid` overloads `@` to return its cells, `@g` no longer means the address of `g` — and assigning it to a `Grid ptr` hands the handle the cells instead of the grid. The compiler noticed:

```
indexing.tiko(61,1): warning TKO0830: suspicious pointer assignment: a
'grid ptr' from a 'int32 ptr'
```

`varptr( g )` is never overloaded, and always means the address of `g`, which is why the listing uses it for the real assignment.

A type that counts

examples/ch13/rangecounter.tiko

```
'' rangecounter.tiko -- a type that a for loop can count with.
''
'' for e as T = first to last [step s] calls five operators on T:
'' for (once, at the start), next (should the loop go on?) and step
```

```
'' (advance) -- each in a form with and without the step value.

type Every
    declare constructor( byval startAt as int32, byval stepBy as int32 )
    declare operator for( )
    declare operator for( byref stp as Every )
    declare operator step( )
    declare operator step( byref stp as Every )
    declare operator next( byref last as Every ) as integer
    declare operator next( byref last as Every, _
                           byref stp as Every ) as integer
    declare operator cast( ) as string
    value as int32
    by_ as int32
end type

constructor Every( byval startAt as int32, byval stepBy as int32 )
    value = startAt
    by_ = stepBy
end constructor

operator Every.for( )                   '' nothing to prepare
end operator

operator Every.for( byref stp as Every )
end operator

operator Every.step( )                  '' no step clause: own stride
    value += by_
end operator

operator Every.step( byref stp as Every )   '' step clause given
    value += stp.value
end operator

operator Every.next( byref last as Every ) as integer
    return value <= last.value
end operator

operator Every.next( byref last as Every, _
                     byref stp as Every ) as integer
    return value <= last.value
end operator
```

```
operator Every.cast( ) as string
    return str( value )
end operator

print "every third from 1 to 12:";
for e as Every = Every( 1, 3 ) to Every( 12, 0 )
    print " "; e;
next
print

print "0 to 20, step 5:";
for e as Every = Every( 0, 1 ) to Every( 20, 0 ) step Every( 5, 0 )
    print " "; e;
next
print
```

examples/ch13/rangecounter.out

```
every third from 1 to 12: 1 4 7 10
0 to 20, step 5: 0 5 10 15 20
```

`for e as T = first to last step s` works for any type that implements the counting protocol:

Operator	Called	Job
`for`	once, before the first pass	prepare; usually nothing
`next( last )`	before each pass	return non-zero to keep going
`step`	after each pass	advance

Each comes in two forms, with and without the step value, matching whether the `for` has a `step` clause. Nothing in the protocol assumes a direction or even a number — `next` decides when to stop — so a counter can walk dates, a range of keys or a countdown, as Exercise 2 does.

For walking the *elements* of a container rather than counting, Chapter 20's `for each` and its iterator protocol are the better tool.

Taking over allocation

examples/ch13/allocators.tiko

```
'' allocators.tiko -- a type that controls where its objects live.

type Tracked
    declare operator new( byval size as uinteger ) as any ptr
```

```
    declare operator delete( byval p as any ptr )
    n as int32
end type

dim shared as int32 live      '' a count every Tracked shares

'' new and delete are static members: there is no object yet in new,
'' and none left in delete. They allocate; they do not construct.
operator Tracked.new( byval size as uinteger ) as any ptr
    live += 1
    print "  allocating "; size; " bytes"
    return allocate( size )
end operator

operator Tracked.delete( byval p as any ptr )
    live -= 1
    print "  releasing"
    deallocate( p )
end operator

dim as Tracked ptr a = new Tracked
dim as Tracked ptr b = new Tracked
print "live objects:"; live
delete a
delete b
print "live objects:"; live
```

examples/ch13/allocators.out

```
  allocating 4 bytes
  allocating 4 bytes
live objects: 2
  releasing
  releasing
live objects: 0
```

`operator new` and `operator delete` replace the allocation step for one type. They do not construct or destroy — the constructor and destructor still run, before and after — they only decide where the bytes come from and go back to. That makes them the place for a memory pool, an allocation counter as here, or a check that every object was freed.

They are implicitly `static`: in `new` the object does not exist yet, and in `delete` it no longer does, so there is no `this` — and neither `this` nor a bare field may be used in either (`TKO0937`).

What not to overload

An operator is a promise about meaning that the reader will hold you to without reading the code. Break it and every expression using the type becomes a puzzle.

- **Keep the meaning.** `+` combines, `-` takes away, `=` is equality and nothing else. A `+` that appends to a log or a `*` that means "repeat" makes correct-looking code wrong.
- **Keep the algebra where it applies.** If `a + b` exists, `a - b`, `-a` and `a += b` should exist and agree with it. If `=` exists, `<>` must be its negation.
- **Keep the cost obvious.** `a + b` reads as cheap. If it allocates, copies a megabyte or touches a disk, give it a name that says so.
- **Do not overload `@`** unless the type genuinely is a handle to something else — the listing above shows what it does to every `@` in the program.
- **Do not overload `and`, `or` and `not` for logic.** The built-in `andalso` and `orelse` short-circuit; an overloaded `and` cannot, and a reader cannot tell which they are looking at.
- **When the operation has no single obvious meaning, use a method.** `Dot( a, b )` and `Cross( a, b )` are better than picking one of them to be `*`.

What it costs

- **An operator is a procedure call** with the same cost as the method it replaces, and short ones are inlined. Notation is free.
- **Returning a type by value copies it.** For a small value type like `Vec2` that is the right design; for a type that owns memory, each `a + b` constructs a temporary and runs its copy constructor.
- **Conversions run silently.** An implicit `cast` or `let` in a loop is a call per iteration that nothing in the source shows.

Summary

- `operator +` and its kind are procedures the compiler calls for your type. Precedence and associativity never change.
- **Binary and unary arithmetic, comparisons, `->` and `*` are global; compound assignment, `let`, `cast`, `[]`, `@`, the `for` protocol and `new`/`delete` are members** — the compiler enforces it (`TKO0790`, `TKO0697`).
- Define both orders of a mixed-type operator, and define `<>` alongside `=`.
- **`cast( ) as ustring` makes a type printable**, and usable with `&` and the string functions.
- `let` from another type and `cast` to another type are conversions in and out; the built-in maths functions use a numeric `cast`.
- `[]` returning `byref` gives an assignable index.

- **Overloading `@` changes the meaning of every `@`**; `varptr` is never overloaded.
- `for`, `step` and `next` make a type a `for` counter, in either direction.
- `operator new/delete` choose where memory comes from; construction and destruction still happen around them.
- **Overload only where the meaning is the obvious one**, and keep the algebra and the cost honest.

Exercises

1. Write `Money`, stored in whole cents, with `+`, `-`, multiplication by a count, `<`, `=` and a `cast` to text like `$15.30`. Add a thousand ten-cent amounts and check the total is exactly $100. Why is there no `Money * Money`? {solution="ex13-01″}
2. Write a `Countdown` type that a `for` loop counts *down* with, printing `liftoff` at zero. What happens when the start is below the end? {solution="ex13-02″}
3. Add `operator *` to `Vec2` as a dot product, then try to add a second `*` for the cross product. What stops you, and what does that tell you about overloading `*` here at all?
4. Give `Fixed` an `operator /`. What should `1 / 3` give, and where exactly does the rounding happen?
5. Remove `operator <>` from `vec2.tiko` and keep the `a <> b` line. What does the compiler say, and would you rather it had derived `<>` from `=` for you?

Inheritance and Polymorphism

A drawing program has circles, rectangles and squares, and somewhere in it a loop that has to ask each one for its area. Written with the tools of Chapter 12, every shape is its own type, and the loop needs a `select case` on some "kind" field to know which code to run. The next shape means editing that loop, and every other loop like it.

Inheritance lets one type start as another and add to it. **Polymorphism** lets a single call reach the right code for whatever object is really there, so the loop asks "what is your area?" and never needs to know what it is talking to.

tiko gives you two ways to build them, and this chapter is in two parts. **Part 1** is the classic model, the one C++ and FreeBASIC share: a base type with virtual methods, extended by every shape, and the three places it goes wrong: an object deleted through the wrong pointer, an object passed by value, and a relationship that was never really "is a". **Part 2** is **interfaces**: a contract that any type can take on, as many as it needs, without spending its one base on it. The two mix freely, and the chapter ends with how to choose.

Part 1: The classic model

A derived type starts as its base and adds to it, and a method the base marks `virtual` can be replaced by each derived type. Everything in this part works the same whether the type is declared with `type … extends` or, as Part 2 shows, with `class`.

Extending a type

examples/ch14/extends.tiko

```
'' extends.tiko -- a derived type starts as its base, and adds to it.

type Vehicle
    declare constructor( byval wheels as int32 = 4 )
    declare function Describe( ) as ustring
    wheels as int32
    speed as int32
end type

constructor Vehicle( byval wheels as int32 )
    this.wheels = wheels
end constructor

function Vehicle.Describe( ) as ustring
    return wheels & " wheels at " & speed & " km/h"
```

```
end function

type Bicycle extends Vehicle
    declare constructor( byval gears as int32 )
    declare function Describe( ) as ustring
    gears as int32
end type

constructor Bicycle( byval gears as int32 )
    base( 2 )                       '' build the Vehicle part first
    this.gears = gears
end constructor

function Bicycle.Describe( ) as ustring
    return base.Describe( ) & ", " & gears & " gears"
end function

dim as Bicycle bike = Bicycle( 21 )
bike.speed = 25                     '' a field it inherited
print bike.Describe( )

dim as Vehicle car
print car.Describe( )

print "sizes:"; sizeof( Vehicle ); sizeof( Bicycle )
print "offsets:"; offsetof( Bicycle, wheels ); _
    offsetof( Bicycle, speed ); offsetof( Bicycle, gears )
```

examples/ch14/extends.out

```
2 wheels at 25 km/h, 21 gears
4 wheels at 0 km/h
sizes: 8 12
offsets: 0 4 8
```

`type Bicycle extends Vehicle` makes a `Bicycle` a `Vehicle` with more fields. It has `wheels` and `speed` without declaring them, and the offsets show where they live: the base's fields come first, exactly as they are laid out in a `Vehicle`, and `gears` follows them. That layout is the point. A `Bicycle` *starts with* a complete `Vehicle`, so anything that works on a `Vehicle` can be handed one.

`base( 2 )` runs the base's constructor first. The `Vehicle` part of a `Bicycle` is built before the `Bicycle` constructor's own body runs, and `base( … )` chooses which `Vehicle` constructor builds it and with what. Written without a `base( … )` call, the base's default constructor runs instead.

`base.Describe( )` calls the base's version of a method from inside the derived one. `Bicycle`'s `Describe` has the same name as `Vehicle`'s and **hides** it: on a `Bicycle`, `Describe` means `Bicycle.Describe`. Reusing the base's text through `base.` means the two descriptions cannot drift apart.

A base without a default constructor

`Vehicle`'s constructor takes `wheels = 4` as a default, so it can be called with nothing at all. That default is load-bearing:

`examples/ch14/nodefault.tiko`

```
'' nodefault.tiko -- a base with no default constructor leaves the
'' derived type with no way to copy itself.

type Vehicle
    declare constructor( byval wheels as int32 )
    wheels as int32
end type

constructor Vehicle( byval wheels as int32 )
    this.wheels = wheels
end constructor

type Bicycle extends Vehicle
    declare constructor( )
    gears as int32
end type

constructor Bicycle( )
    base( 2 )
end constructor

dim as Bicycle bike
print bike.wheels
```

```
nodefault.tiko(13,1): error TKO0926: 'bicycle' needs a copy constructor:
its base has no constructor that takes no argument, so none can be made
for it
     |
  13 | type Bicycle extends Vehicle
     | ^
```

A derived type gets a copy constructor for free, and that free one works by building a fresh base part with the base's default constructor, then copying every field across. With no default constructor in the base, there is nothing to build the fresh part

with, so the derived type must write its own copy constructor. fbc refuses the same program with its error 188.

Give the base a constructor that takes no arguments, or default its parameters as `extends.tiko` does, and every type derived from it can be copied without writing anything.

`base( … )` passes whatever the base's constructors take, chosen the way any overloaded call is. Here it hands a label up, and the base's default value for it, `"unnamed"`, is what keeps `Sensor` copyable:

examples/ch14/basestring.tiko

```
'' basestring.tiko -- passing text up to the base's constructor.

type Labelled
    declare constructor( byref label as ustring = "unnamed" )
    label as ustring
end type

constructor Labelled( byref label as ustring )
    this.label = label
end constructor

type Sensor extends Labelled
    declare constructor( byref label as ustring, _
                         byval reading as float64 )
    reading as float64
end type

constructor Sensor( byref label as ustring, byval reading as float64 )
    base( label )
    this.reading = reading
end constructor

dim as Sensor s = Sensor( "boiler", 81.5 )
print s.label; " reads"; s.reading
```

examples/ch14/basestring.out

```
boiler reads 81.5
```

Calling through the base

A `Bicycle` can be handed to anything that takes a `Vehicle`. What happens to its methods when it is?

examples/ch14/hiding.tiko

```
'' hiding.tiko -- without virtual, the call is chosen by the type the
'' compiler sees, not by the object that is really there.

type Vehicle
    declare function Kind( ) as ustring
    wheels as int32
end type

function Vehicle.Kind( ) as ustring
    return "vehicle"
end function

type Bicycle extends Vehicle
    declare function Kind( ) as ustring
end type

function Bicycle.Kind( ) as ustring
    return "bicycle"
end function

sub Show( byref v as Vehicle )
    print "  Show sees a "; v.Kind( )
end sub

dim as Bicycle bike
print "directly: "; bike.Kind( )
Show( bike )
```

examples/ch14/hiding.out

```
directly: bicycle
  Show sees a vehicle
```

`Show` declares its parameter as a `Vehicle`, so inside `Show` the compiler knows only that it has a `Vehicle`, and `v.Kind( )` is compiled as a call to `Vehicle.Kind`. Which object arrives at run time does not enter into it. Hiding a method changes what the name means on the derived type; it changes nothing for code that sees the object as its base.

That is the right behaviour for most methods, and it costs nothing. But it is exactly wrong for the drawing program's loop, which sees every shape as "a shape" and needs each one's own answer.

Virtual methods

examples/ch14/shapes.tiko

```
'' shapes.tiko -- one loop over different shapes, each one answering
'' for itself.

type Shape extends Object
    declare abstract function Area( ) as float64
    declare virtual function Kind( ) as ustring
    declare function Describe( ) as ustring
end type

function Shape.Kind( ) as ustring
    return "shape"
end function

'' Not virtual, yet the calls inside it are.
function Shape.Describe( ) as ustring
    return Kind( ) & ", area " & Area( )
end function

type Circle extends Shape
    declare constructor( byval r as float64 )
    declare function Area( ) as float64 override
    declare function Kind( ) as ustring override
    r as float64
end type

constructor Circle( byval r as float64 )
    this.r = r
end constructor

function Circle.Area( ) as float64
    return 3.14159 * r * r
end function

function Circle.Kind( ) as ustring
    return "circle"
end function

type Rect extends Shape
    declare constructor( byval w as float64 = 1, _
                         byval h as float64 = 1 )
    declare function Area( ) as float64 override
    '' virtual AND override: a type derived from Rect may override
    '' it again.
    declare virtual function Kind( ) as ustring override
```

```
    w as float64
    h as float64
end type

constructor Rect( byval w as float64, byval h as float64 )
    this.w = w : this.h = h
end constructor

function Rect.Area( ) as float64
    return w * h
end function

function Rect.Kind( ) as ustring
    return "rectangle"
end function

type Square extends Rect
    declare constructor( byval side as float64 )
    declare function Kind( ) as ustring override
end type

constructor Square( byval side as float64 )
    base( side, side )
end constructor

function Square.Kind( ) as ustring
    return "square"
end function

dim as Circle c = Circle( 1 )
dim as Rect r = Rect( 2, 3 )
dim as Square s = Square( 2 )
dim as Shape ptr all( 0 to 2 ) = { @c, @r, @s }

for i as int32 = lbound( all ) to ubound( all )
    print all( i )->Describe( )
next
```

examples/ch14/shapes.out

```
circle, area 3.14159
rectangle, area 6
square, area 4
```

Four words do the work.

- **`extends Object.`** `Object` is the built-in root type. Extending it gives every object of the type one hidden field, a pointer to the type's table of virtual methods, and that pointer is how a call finds the right code at run time.
- **`virtual`** marks a method that derived types may replace. A call to it looks the method up in the object's table, so it reaches the version belonging to the object that is really there.
- **`override`** marks a method that replaces a virtual one. The compiler checks that there is something to replace: `override` on a method the base did not make virtual is `TKO0621`.
- **`abstract`** declares a virtual method with no body at all. `Shape` cannot say what its area is, so it says only that every shape has one.

`Shape.Describe` is not virtual, yet its output is different for every shape, because the two calls *inside* it are. That is a common and useful shape: the base writes the procedure once, and the derived types supply the parts that differ.

An override is not itself virtual. `Circle.Kind` replaces `Shape.Kind`, but a type derived from `Circle` could not override it again. `Rect` declares its `Kind` as `virtual … override`, which both replaces `Shape`'s and opens it to `Rect`'s own descendants. That is what lets `Square` override it in turn. `Square` keeps `Rect`'s `Area`, because a square's area is a rectangle's.

Leave `virtual` off, and the next level down cannot override:

examples/ch14/notvirtual.tiko

```
'' notvirtual.tiko -- an override is not itself virtual. A type
'' derived from Dog that writes Speak again hides it; it does not
'' override it.

type Animal extends Object
    declare virtual function Speak( ) as ustring
end type

function Animal.Speak( ) as ustring
    return "..."
end function

type Dog extends Animal
    declare function Speak( ) as ustring override
end type

function Dog.Speak( ) as ustring
    return "woof"
end function
```

```
type Puppy extends Dog
    declare function Speak( ) as ustring     '' hides Dog's
end type

function Puppy.Speak( ) as ustring
    return "yip"
end function

sub Listen( byref a as Animal )
    print "  through Animal: "; a.Speak( )
end sub

dim as Puppy pup
print "directly: "; pup.Speak( )
Listen( pup )
```

examples/ch14/notvirtual.out

```
directly: yip
  through Animal: woof
```

`Dog.Speak` is an override but not virtual, so `Puppy`'s `Speak` hides it without replacing it. Called on a `Puppy` directly it answers `yip`; called through an `Animal` it reaches `Dog`'s. Writing `override` on `Puppy.Speak` would not help: there is no open slot for it to fill, and the compiler says so (`TKO0621`). The fix is one word in `Dog`, `virtual … override`, exactly as `Rect` does in `shapes.tiko`.

C++ programmers expect the opposite, because there an override stays virtual all the way down. tiko follows fbc, where it does not, so code moves between the two compilers unchanged.

Abstract types

examples/ch14/abstract.tiko

```
'' abstract.tiko -- a type with an unwritten method cannot be created.

type Shape extends Object
    declare abstract function Area( ) as float64
end type

dim as Shape s
print s.Area( )
```

```
abstract.tiko(7,14): error TKO0620: 's' has an abstract type, which cannot
be instantiated
     |
   7 | dim as Shape s
     |              ^
```

A type with an abstract method is incomplete on purpose, so it cannot be created: there would be nothing to run when someone asked for its area. A derived type that fills in every abstract method can be created, and one that leaves any of them unwritten is abstract too, with the same error. What *can* exist is a `Shape ptr` or a `byref Shape` parameter, pointing at a complete type.

Seeing the table

`tiko --dump-vtable` prints each type's table of virtual methods. For `shapes.tiko`:

```
object vtable 0 slot(s)
shape vtable 2 slot(s)
  slot 0 <abstract>
  slot 1 _ZN5SHAPE4KINDEv
circle vtable 2 slot(s)
  slot 0 _ZN6CIRCLE4AREAEv
  slot 1 _ZN6CIRCLE4KINDEv
rect vtable 2 slot(s)
  slot 0 _ZN4RECT4AREAEv
  slot 1 _ZN4RECT4KINDEv
square vtable 2 slot(s)
  slot 0 _ZN4RECT4AREAEv
  slot 1 _ZN6SQUARE4KINDEv
vtables: 5
```

Every type in the family has the same two slots in the same order, `Area` then `Kind`. A derived type copies its base's table and replaces the entries it overrides, so a call through a `Shape` always looks in slot 0 for `Area`, whichever table it finds. `Shape`'s slot 0 is `<abstract>`, which is why a `Shape` cannot exist; `Square`'s slot 0 still names `Rect.Area`, which it inherited. The names are mangled the way C++ mangles them: `_ZN6SQUARE4KINDEv` is `SQUARE::KIND()`.

The dump is the place to look when a call reaches the wrong method. It shows what the compiler built, and comparing it with what you meant is usually enough to find the mistake.

Virtual destructors

An object created with `new` and deleted through a pointer to its base is destroyed by whichever destructor that pointer's type names:

examples/ch14/vdtor.tiko

```
'' vdtor.tiko -- delete through a base pointer runs the destructor the
'' pointer's type names, unless that destructor is virtual.

type Plain
    declare destructor( )
    id as int32
end type

destructor Plain( )
    print "  ~Plain"
end destructor

type PlainFile extends Plain
    declare destructor( )
    handle as int32
end type

destructor PlainFile( )
    print "  ~PlainFile: the file is closed"
end destructor

type Careful extends Object
    declare virtual destructor( )
end type

destructor Careful( )
    print "  ~Careful"
end destructor

type CarefulFile extends Careful
    declare destructor( )
    handle as int32
end type

destructor CarefulFile( )
    print "  ~CarefulFile: the file is closed"
end destructor

print "plain destructor:"
dim as Plain ptr p = new PlainFile
delete p

print "virtual destructor:"
```

```
dim as Careful ptr q = new CarefulFile
delete q
```

examples/ch14/vdtor.out

```
plain destructor:
  ~Plain
virtual destructor:
  ~CarefulFile: the file is closed
  ~Careful
```

PITFALL

`delete` through a base pointer skips the derived destructor unless the base's is virtual. `delete p` ran `~Plain` and nothing else. `PlainFile`'s destructor, the one that closes the file, never ran, and nothing said so: the program compiled cleanly, and the leaked handle surfaces somewhere else, later.

With `declare virtual destructor( )` in the base, `delete q` finds the right destructor through the table, runs it, and then runs the base's, in that order. **A type meant to be derived from and deleted through its base declares a virtual destructor**, even an empty one.

The rule from the previous section applies to destructors as well, and this is where it costs most. A destructor is virtual only if it says so, so a middle level that declares a plain one closes the chain below it:

examples/ch14/vdtormiddle.tiko

```
'' vdtormiddle.tiko -- 'virtual' is not inherited by a destructor that
'' does not say it. Each level that will be deleted through a base
'' pointer declares its destructor virtual.

type Resource extends Object
    declare virtual destructor( )
end type
destructor Resource( )
    print "  ~Resource"
end destructor

type File extends Resource
    declare destructor( )              '' not virtual
    handle as int32
end type
destructor File( )
```

```
    print "  ~File"
end destructor

type LogFile extends File
    declare destructor( )
    lines as int32
end type
destructor LogFile( )
    print "  ~LogFile: flushed"
end destructor

type SafeFile extends Resource
    declare virtual destructor( )      '' virtual again
    handle as int32
end type
destructor SafeFile( )
    print "  ~SafeFile"
end destructor

type SafeLog extends SafeFile
    declare destructor( )
    lines as int32
end type
destructor SafeLog( )
    print "  ~SafeLog: flushed"
end destructor

print "middle destructor not virtual:"
dim as Resource ptr r = new LogFile
delete r

print "middle destructor virtual:"
r = new SafeLog
delete r
```

examples/ch14/vdtormiddle.out

```
middle destructor not virtual:
  ~File
  ~Resource
middle destructor virtual:
  ~SafeLog: flushed
  ~SafeFile
  ~Resource
```

`LogFile`'s destructor never ran, although `Resource`'s is virtual and `LogFile` declares one of its own. `File` declared a destructor without `virtual`, and from there down nothing overrides. `SafeFile` says `virtual` again, and its descendant is destroyed properly. The same happens when the middle type declares no destructor at all: the one the compiler writes for it is not virtual. **Every level that may be deleted through a base pointer declares `virtual destructor`.**

Asking what an object really is

Sometimes the code holding an `Animal` needs something only a `Dog` has:

examples/ch14/isdown.tiko

```
'' isdown.tiko -- asking what an object really is, and treating it as
'' that type once the answer is yes.

type Animal extends Object
    declare virtual function Speak( ) as ustring
end type

function Animal.Speak( ) as ustring
    return "..."
end function

type Dog extends Animal
    declare virtual function Speak( ) as ustring override
    tricks as int32
end type

function Dog.Speak( ) as ustring
    return "woof"
end function

type Puppy extends Dog
    declare function Speak( ) as ustring override
end type

function Puppy.Speak( ) as ustring
    return "yip"
end function

sub Inspect( byref a as Animal )
    print a.Speak( );
    if a is Dog then
        '' a really is a Dog (or something derived from one)
        dim as Dog ptr d = cast( Dog ptr, @a )
        print ", knows"; d->tricks; " tricks";
```

```
    end if
    if a is Puppy then print ", and is a puppy";
    print
end sub

dim as Animal cat
dim as Dog rex
rex.tricks = 3
dim as Puppy pup
pup.tricks = 1

Inspect( cat )
Inspect( rex )
Inspect( pup )
```

examples/ch14/isdown.out

```
...
woof, knows 3 tricks
yip, knows 1 tricks, and is a puppy
```

`a is Dog` is true when the object is a `Dog` **or anything derived from one**, which is why the puppy passes both tests. Once it is true, `cast( Dog ptr, @a )` treats the same object as a `Dog`. The cast itself checks nothing, so always guard it with `is`.

`is` needs the type to extend `Object`, because the answer is read from the hidden table pointer, and it only accepts a type derived from the variable's own type: asking whether an `Animal` *is* an `Animal` is refused (`TKO0798`), and asking it of a type with no table is `TKO0799`. fbc refuses both too.

Use `is` sparingly. A chain of `if a is Dog … elseif a is Cat …` is the `select case` this chapter set out to remove, wearing different clothes. If the question is "what should this animal do", the answer is a virtual method.

Passing by value

examples/ch14/slicing.tiko

```
'' slicing.tiko -- a derived object passed by value arrives as a copy of
'' its base part, and only that.

type Animal extends Object
    declare virtual function Speak( ) as ustring
    legs as int32
end type

function Animal.Speak( ) as ustring
```

```
    return "..."
end function

type Dog extends Animal
    declare function Speak( ) as ustring override
    tricks as int32
end type

function Dog.Speak( ) as ustring
    return "woof"
end function

sub ByReference( byref a as Animal )
    print "byref:  "; a.Speak( ); ", a Dog? "; ( a is Dog )
end sub

sub ByValue( byval a as Animal )
    print "byval:  "; a.Speak( ); ", a Dog? "; ( a is Dog )
end sub

dim as Dog rex
rex.tricks = 3
ByReference( rex )
ByValue( rex )

dim as Animal copy = rex
print "copied: "; copy.Speak( )

'' The hidden field: one pointer, to the type's table of methods.
print "sizes:"; sizeof( Object ); sizeof( Animal ); sizeof( Dog )
```

examples/ch14/slicing.out

```
byref:  woof, a Dog? -1
byval:  ..., a Dog?  0
copied: ...
sizes: 8 16 24
```

PITFALL

Passing a derived object by value keeps only its base part. `ByValue` received a fresh `Animal`, made by copying the `Animal` part of `rex`, and that copy is an `Animal` through and through. It speaks `...`, it is not a `Dog`, and `tricks` is gone. The same happens on assignment: `dim as Animal copy = rex` is a plain `Animal`.

Nothing warns about this, because nothing is wrong with the program from the compiler's point of view: an `Animal` was asked for and an `Animal` was made. **Pass an object of a type with virtual methods `byref`, or by pointer**, never `byval`.

The last line of the output is the hidden field from the previous section made visible. `Object` is eight bytes, the table pointer and nothing else. `Animal` adds `legs` and rounds up to 16, and `Dog` adds `tricks` and rounds to 24.

Composition, and abstract types as contracts

Inheritance says "is a". A `Bicycle` is a `Vehicle`, and a `Square` is a `Rect`. When the relationship is "has a" or "uses a", inheritance is the wrong tool, even when it would save typing: a `Circle` that extends a `Point` to borrow its `x` and `y` can be passed to every procedure that expects a point, and none of them meant a circle. Give the `Circle` a `Point` field instead.

examples/ch14/pluginhost.tiko

```
'' pluginhost.tiko -- an abstract type as the contract, and a host that
'' is built from parts rather than derived from one.

'' The contract: anything that can transform a line of text.
type Plugin extends Object
    declare abstract function Name( ) as ustring
    declare abstract function Apply( byref text as ustring ) as ustring
    declare virtual destructor( )
end type

destructor Plugin( )
end destructor

type Shout extends Plugin
    declare function Name( ) as ustring override
    declare function Apply( byref text as ustring ) as ustring override
end type

function Shout.Name( ) as ustring
    return "shout"
end function

function Shout.Apply( byref text as ustring ) as ustring
    return ucase( text ) & "!"
end function
```

```
type Reverse extends Plugin
    declare function Name( ) as ustring override
    declare function Apply( byref text as ustring ) as ustring override
end type

function Reverse.Name( ) as ustring
    return "reverse"
end function

function Reverse.Apply( byref text as ustring ) as ustring
    dim as ustring result
    for i as int32 = len( text ) to 1 step -1
        result &= mid( text, i, 1 )
    next
    return result
end function

'' The host HAS plugins; it is not one. It owns them, so it deletes
'' them -- through Plugin ptr, which is why the destructor is virtual.
type Host
    declare destructor( )
    declare sub Add( byval p as Plugin ptr )
    declare function Run( byref text as ustring ) as ustring
    plugins( any ) as Plugin ptr
end type

destructor Host( )
    for i as int32 = lbound( plugins ) to ubound( plugins )
        delete plugins( i )
    next
end destructor

sub Host.Add( byval p as Plugin ptr )
    redim preserve plugins( 0 to ubound( plugins ) + 1 )
    plugins( ubound( plugins ) ) = p
end sub

function Host.Run( byref text as ustring ) as ustring
    dim as ustring t = text
    for i as int32 = lbound( plugins ) to ubound( plugins )
        t = plugins( i )->Apply( t )
        print "  after "; plugins( i )->Name( ); ": "; t
    next
    return t
```

```
end function

scope
    dim as Host h
    h.Add( new Reverse )
    h.Add( new Shout )
    print h.Run( "plugins" )
end scope
```

examples/ch14/pluginhost.out

```
  after reverse: snigulp
  after shout: SNIGULP!
SNIGULP!
```

`Plugin` is all contract: two abstract methods and a virtual destructor, with no data. Any type that extends it and fills in `Name` and `Apply` is a plugin, and `Host` can run it without knowing what it is. Written this way, the contract is an abstract type.

`Host` is the other half. It is not a plugin and does not extend one. It **has** plugins: it owns them, runs them in order, and deletes them in its destructor. The deletes go through `Plugin ptr`, which is why `Plugin`'s destructor is virtual. Adding a third plugin means writing one new type; `Host` does not change.

A type extends exactly one base, and that is where this model runs out. A `Plugin` that must also be something else, a second contract on the same type, a small value that should follow the contract without carrying a table pointer: none of them fits in one chain. Part 2 is the answer.

Part 2: Interfaces

Part 1's contracts are base types, and a type has one base. A `Drawing` that is both a shape and a document, a `Point` that should be printable without carrying a table pointer, a library type that already extends something else: none of them fits. An **interface** is a contract that is not a base. Any type can take on as many as it needs, and keep its own base if it has one.

A contract any type can take on

examples/ch14/ifaceshapes.tiko

```
'' ifaceshapes.tiko -- the shapes loop again, with an interface as the
'' contract and classes that implement it.

interface IShape
    function Area( ) as float64
```

```
    function Kind( ) as ustring
end interface

class Circle
    implements IShape
    declare constructor( byval r as float64 )
    declare function Area( ) as float64 implements IShape.Area
    declare function Kind( ) as ustring implements IShape.Kind
    r as float64
end class

constructor Circle( byval r as float64 )
    this.r = r
end constructor

function Circle.Area( ) as float64
    return 3.14159 * r * r
end function

function Circle.Kind( ) as ustring
    return "circle"
end function

class Rect
    implements IShape
    declare constructor( byval w as float64, byval h as float64 )
    declare function Area( ) as float64 implements IShape.Area
    declare function Kind( ) as ustring implements IShape.Kind
    w as float64
    h as float64
end class

constructor Rect( byval w as float64, byval h as float64 )
    this.w = w : this.h = h
end constructor

function Rect.Area( ) as float64
    return w * h
end function

function Rect.Kind( ) as ustring
    return "rectangle"
end function
```

```
'' Written once, against the contract only.
sub Describe( byval s as IShape )
    print s.Kind( ); ", area"; s.Area( )
end sub

dim as Circle c = Circle( 1 )
dim as Rect r = Rect( 2, 3 )
dim as IShape shapes( 0 to 1 )
shapes( 0 ) = c          '' an object converts to an interface it
shapes( 1 ) = @r         '' implements, and so does a pointer to one
for i as int32 = 0 to 1
    Describe( shapes( i ) )
next
print "an IShape value is"; sizeof( IShape ); " bytes"
```

examples/ch14/ifaceshapes.out

```
circle, area 3.14159
rectangle, area 6
an IShape value is 16 bytes
```

`interface IShape` lists what a shape can do and nothing else: no fields, no code, no constructor. Every member is public and abstract, so there is nothing to write on one but its signature. An interface is never created by itself.

`class Circle` is a type that always extends `Object`. It is the same as `type Circle extends Object`, written so that the object model is the default. Its body starts with the contracts it takes on, **`implements IShape`**, and then each method says which member of the contract it provides: `declare function Area( ) as float64 implements IShape.Area`. The mapping is explicit. A public method called `Area` does not become `IShape.Area` until it says so, and a member the class forgets is an error that names it:

examples/ch14/missing.tiko

```
'' missing.tiko -- a member of the contract with no method mapped
'' to it.

interface IShape
    function Area( ) as float64
    function Kind( ) as ustring
end interface

class Circle
    implements IShape
    declare function Area( ) as float64 implements IShape.Area
    r as float64
```

```
end class

function Circle.Area( ) as float64
    return 3.14159 * r * r
end function
```

```
missing.tiko(10,16): error TKO1000: 'circle' does not implement
'ishape.kind'; add 'implements ishape.kind' to one of its methods
     |
  10 |     implements IShape
     |                ^
missing.tiko(6,5): note: 'kind' is required by interface 'ishape',
declared here
     |
   6 |     function Kind( ) as ustring
     |     ^
```

An interface value is a reference. `dim as IShape shapes( 0 to 1 )` holds two of them, and each is the address of an object together with that object's table for `IShape`: sixteen bytes, as the last line says. A `Circle` converts to `IShape` implicitly, and so does a pointer to one. Members are called with `.`, never `->`, because the value is not a pointer you can follow. `Describe` is written against `IShape` alone, and runs unchanged for every type that implements it.

Two contracts at once

`examples/ch14/twocontracts.tiko`

```
'' twocontracts.tiko -- one type, two contracts, and a member name the
'' two contracts both use for different things.

interface IShape
    function Area( ) as float64
    function Name( ) as ustring          '' what the shape is called
end interface

interface IDocument
    function Name( ) as ustring          '' the file it is saved in
    sub Save( )
    sub Close( )
end interface

interface IResource
    sub Close( )
end interface
```

```
class Drawing
    implements IShape, IDocument, IResource
    declare function Area( ) as float64 implements IShape.Area
    declare function Title( ) as ustring implements IShape.Name
    declare function FileName( ) as ustring implements IDocument.Name
    declare sub Close( ) implements IDocument.Close, IResource.Close
    w as float64 = 4
    h as float64 = 2
private:
    declare sub WriteOut( ) implements IDocument.Save
end class

function Drawing.Area( ) as float64
    return w * h
end function

function Drawing.Title( ) as ustring
    return "floor plan"
end function

function Drawing.FileName( ) as ustring
    return "plan.txt"
end function

sub Drawing.Close( )
    print "closed"
end sub

sub Drawing.WriteOut( )
    print "saving "; FileName( )
end sub

dim as Drawing d
dim as IShape s = d
dim as IDocument doc = d
dim as IResource res = d
print s.Name( ); " / "; doc.Name( ); ", area"; s.Area( )
doc.Save( )                    '' reaches the private WriteOut
res.Close( )
```

examples/ch14/twocontracts.out

```
```

```
floor plan / plan.txt, area 8
saving plan.txt
closed
```

`Drawing` takes on three interfaces on one `implements` line. Part 1 could not express any of this:

- **Two contracts both have a `Name`, and they mean different things.** The shape's name is its title, the document's is a file name. They are different members, so `Drawing` maps each to its own method: `Title` implements `IShape.Name`, `FileName` implements `IDocument.Name`. A method's name does not have to match the member's.
- **One method can provide several members.** `Close` implements `IDocument.Close` and `IResource.Close` at once, because closing the drawing means the same thing to both.
- **A private method can implement a member.** `WriteOut` is under `private:`, so code outside `Drawing` cannot call it directly (`TKO0625`), yet `doc.Save( )` reaches it. Saving is something you do to a document, not to a drawing that happens to be one.

Small values

A type that implements an interface does not need a table pointer of its own. The table travels in the interface value instead, so a plain `type` can implement one:

examples/ch14/ifacevalue.tiko

```
'' ifacevalue.tiko -- a plain type implements an interface and stays
'' the size of its fields.

interface IShow
    function Show( ) as ustring
end interface

type Point
    implements IShow
    x as int32
    y as int32
    declare function Show( ) as ustring implements IShow.Show
end type

function Point.Show( ) as ustring
    return "(" & x & ", " & y & ")"
end function
```

```
'' The same contract written as an abstract base type.
type Showable extends Object
    declare abstract function Show( ) as ustring
end type

type OldPoint extends Showable
    x as int32
    y as int32
    declare function Show( ) as ustring override
end type

function OldPoint.Show( ) as ustring
    return "(" & x & ", " & y & ")"
end function

dim as Point p = ( 3, 4 )
dim as IShow v = p
print v.Show( )
print "Point:"; sizeof( Point ); " bytes"
print "OldPoint:"; sizeof( OldPoint ); " bytes"
```

examples/ch14/ifacevalue.out

```
(3, 4)
Point: 8 bytes
OldPoint: 16 bytes
```

`Point` is eight bytes, its two fields. `OldPoint` follows the same contract written as an abstract base type, and every one of them pays eight bytes more for its table pointer. An array of a million points is 8 MB one way and 16 MB the other. The interface's sixteen bytes are paid only where an interface value is actually held.

Asking an interface what it holds

examples/ch14/ifacequery.tiko

```
'' ifacequery.tiko -- asking what is behind an interface value.

interface IShow
    function Show( ) as ustring
end interface

interface ISize
    function Size( ) as int32
end interface
```

```
class Note
    implements IShow, ISize
    declare function Show( ) as ustring implements IShow.Show
    declare function Size( ) as int32 implements ISize.Size
    text as ustring = "buy milk"
end class

function Note.Show( ) as ustring
    return text
end function

function Note.Size( ) as int32
    return len( text )
end function

class Label
    implements IShow
    declare function Show( ) as ustring implements IShow.Show
end class

function Label.Show( ) as ustring
    return "label"
end function

sub Inspect( byval v as IShow )
    print v.Show( ); ":"
    '' A cross cast: does the same object ALSO implement ISize?
    dim as ISize sz = trycast( ISize, v )
    if sz <> nullptr then print "  size"; sz.Size( )
    '' A downcast: is it a Note?
    if v is Note then
        dim as Note ptr np = cast( Note ptr, v )
        print "  a note saying "; np->text
    end if
    print "  is ISize: "; cbool( v is ISize )
end sub

dim as Note n
dim as Label l
Inspect( n )
Inspect( l )
```

examples/ch14/ifacequery.out

```
```

```
buy milk:
  size 8
  a note saying buy milk
  is ISize: true
label:
  is ISize: false
```

`is` works with an interface on either side. `v is Note` asks whether the object behind `v` is a `Note`, or derives from one; `v is ISize` asks whether it implements `ISize`. Neither needs the object's type to be related to `IShow`.

`trycast` answers the same questions with a value. `trycast( ISize, v )` is a **cross cast**: the same object, seen through another interface it implements, or an empty interface value, equal to `nullptr`, when it implements no such thing. `trycast( Note ptr, v )` is the checked downcast. `cast( Note ptr, v )` is the unchecked one, which is why the listing asks `v is Note` first: a wrong `cast` is undefined behaviour, exactly as it is in Part 1.

Who owns the object

An interface value refers to an object. It does not own it, copy it or keep it alive. The compiler refuses the one case it can see, a temporary that would be gone by the next statement:

examples/ch14/temporary.tiko

```
'' temporary.tiko -- an interface value refers to an object, and a
'' temporary would be gone before the reference was used.

interface IShow
    function Show( ) as ustring
end interface

type Tag
    implements IShow
    text as ustring
    declare function Show( ) as ustring implements IShow.Show
end type

function Tag.Show( ) as ustring
    return text
end function

function MakeTag( ) as Tag
    dim as Tag t
    t.text = "new"
```

```
    return t
end function

dim as IShow v = MakeTag( )
print v.Show( )
```

```
temporary.tiko(24,14): error TKO1012: an interface reference cannot refer
to a temporary; store the object in a variable first
     |
  24 | dim as IShow v = MakeTag( )
     |              ^
```

`delete` through an interface is refused too (`TKO1013`): the interface does not know how the object was made, or whether it was made with `new` at all.

PITFALL

An interface value can outlive its object. Return an `IShape` that refers to a local `Circle`, or keep one in a list after the `Circle` it refers to has gone out of scope, and it compiles cleanly. The next call through it reads whatever is in that memory by then.

Keep the owning variable, or the owning pointer, as the object's own type, and make sure it lives at least as long as every interface value that refers to it. A host that holds plugins as interfaces is given them by code that owns them; it does not own them itself.

Classes and interfaces together

Interfaces do not replace class inheritance. They sit beside it, and a call through an interface honours every override below the class that implements it:

examples/ch14/ifaceclass.tiko

```
'' ifaceclass.tiko -- a class hierarchy behind an interface: the call
'' through the interface reaches the most derived override.

interface ISpeaker
    sub Speak( )
end interface

class Animal
    implements ISpeaker
    declare virtual sub Speak( ) implements ISpeaker.Speak
end class
```

```
sub Animal.Speak( )
    print "..."
end sub

class Dog
    inherits Animal
    declare virtual sub Speak( ) override
end class

sub Dog.Speak( )
    print "woof"
end sub

class Puppy
    inherits Dog
    declare sub Speak( ) override
end class

sub Puppy.Speak( )
    print "yip"
end sub

dim as Animal a
dim as Dog d
dim as Puppy p
dim as ISpeaker speakers( 0 to 2 )
speakers( 0 ) = a
speakers( 1 ) = d
speakers( 2 ) = p
for i as int32 = 0 to 2
    speakers( i ).Speak( )
next
```

examples/ch14/ifaceclass.out

```
...
woof
yip
```

`class Dog` names its base with an **`inherits`** line, the class's spelling of `extends`, at the top of its body. `Animal` implements `ISpeaker.Speak` with a *virtual* method, so an `ISpeaker` that refers to a `Dog` or a `Puppy` reaches their overrides. The rule from Part 1 still holds: `Dog` writes `virtual … override` so that `Puppy` can override it in turn.

Use this only when you want a base class that supplies code its subclasses replace. The default style is the one in the previous sections: each class implements the interfaces it needs, with no base class and no `virtual` anywhere, and shared code goes in a procedure that takes the interface.

Generic interfaces and constraints

An interface can take type parameters. Each instance, `IComparable( of Money )` or `IComparable( of Date )`, is an interface of its own. And a generic procedure or type can require its type argument to implement one, with `as` in its parameter list:

examples/ch14/ifacegeneric.tiko

```
'' ifacegeneric.tiko -- a generic interface, and a generic procedure
'' that requires one of its type argument.

interface IComparable( of T )
    function CompareTo( byref other as T ) as int32
end interface

class Money
    implements IComparable( of Money )
    declare constructor( byval cents as int32 = 0 )
    declare function CompareTo( byref other as Money ) as int32 _
        implements IComparable( of Money ).CompareTo
    cents as int32
end class

constructor Money( byval cents as int32 )
    this.cents = cents
end constructor

function Money.CompareTo( byref other as Money ) as int32
    return sgn( cents - other.cents )
end function

'' T must implement IComparable( of T ); the body calls T's own
'' CompareTo directly.
function Largest( of T as IComparable( of T ) )( a( ) as T ) as T
    dim as int32 best = lbound( a )
    for i as int32 = lbound( a ) + 1 to ubound( a )
        if a( i ).CompareTo( a( best ) ) > 0 then best = i
    next
    return a( best )
end function

dim as Money prices( 0 to 2 ) = _
```

```
    { Money( 250 ), Money( 1999 ), Money( 75 ) }
print "largest:"; Largest( prices( ) ).cents
```

examples/ch14/ifacegeneric.out

```
largest: 1999
```

`of T as IComparable( of T )` is a **constraint**: whatever `T` is, it must implement `IComparable` of itself. Several are written in braces, `of K as { IComparable( of K ), IShow }`. The constraint is checked once, where `Largest` is used, and a type argument that falls short is refused there:

examples/ch14/constraint.tiko

```
'' constraint.tiko -- a type argument that does not implement what the
'' parameter requires is refused where the generic is used.

interface IComparable( of T )
    function CompareTo( byref other as T ) as int32
end interface

function Largest( of T as IComparable( of T ) )( a( ) as T ) as T
    return a( lbound( a ) )
end function

dim as int32 counts( 0 to 2 ) = { 3, 1, 2 }
print Largest( counts( ) )
```

```
constraint.tiko(13,14): error TKO1022: type argument 'int32' does not
implement 'icomparable( of int32 )', which 'largest( of t )' requires
     |
  13 | print Largest( counts( ) )
     |              ^
```

Without the constraint, `Largest( counts( ) )` would fail inside the body, at `a( i ).CompareTo`, with an error about a member that `int32` does not have, and the reader would have to work out why from there. A constraint moves the error to the call that caused it.

A constraint does not change how the body runs. `Largest` is still compiled once for each type it is used with, and inside it `a( i ).CompareTo( … )` is an ordinary call to `Money.CompareTo`: no interface value and no table. Chapter 17 covers generics in full.

Choosing between them

You want	Use
A contract that unrelated types follow	an interface
A type to follow two contracts, or to keep its own base while following one	interfaces
A small value type that follows a contract	a plain `type` that implements an interface
A base class that supplies code its subclasses replace	`extends` (or `inherits`) with `virtual`
`is` and `delete` through the base, with the base owning nothing else	the classic model, with a virtual destructor
Generic code that needs something of `T`	a constraint, `of T as IFoo`
A layout that C or COM defines	the classic model: an abstract type that extends `Object`

The two mix freely. A class can inherit a base and implement interfaces, a `type` can extend a class, and a classic base can implement an interface with virtual methods that its descendants override.

What it costs

- **Every object of a type that extends `Object` carries one pointer**, eight bytes, plus whatever padding it causes. A million small objects pay eight megabytes for it.
- **A virtual call is an indirect call**: load the table pointer, load the slot, call through it. That is a few instructions, but it prevents the call from being inlined, which matters inside a hot loop over small methods.
- **A non-virtual method costs nothing extra**, inherited or not. Make methods virtual because a derived type needs to replace them, not by default.
- **`is` walks the chain of base types**, so it costs a little more the deeper the hierarchy.
- **Abstract types and `override` are compile-time only.** They cost nothing at run time.
- **An interface value is sixteen bytes**, the object's address and its table for that interface, and a call through it is one indirect call, the same as a virtual call. A class whose implementing method is virtual adds a short hop through its own table, so an override is reached.
- **Implementing an interface adds nothing to an object.** A plain `type` stays the size of its fields; the table is in the interface value.
- **`trycast` and `is` with an interface search a small table** at run time. Converting an object to an interface, or an interface to one it inherits, does not.

- **Mappings and constraints are compile-time only.** A constrained generic calls `T`'s methods directly.

Summary

- `type B extends A` starts `B` as a complete `A`, base fields first. `base( … )` chooses the base constructor, which runs before the derived body; `base.Method( )` calls the base's version.
- **A base with no default constructor forces every derived type to write a copy constructor** (`TKO0926`). Default the base constructor's parameters instead.
- Without `virtual`, a call is chosen by the type the compiler sees. A method of the same name on a derived type **hides** the base's; it does not replace it for code that sees the base.
- `extends Object` adds a table pointer. `virtual` methods are looked up in it at run time; `override` replaces one (`TKO0621` if there is nothing to replace); `abstract` declares one with no body. **An override is not virtual unless you also write `virtual`**, so a type derived from it hides the method instead of overriding it.
- A type with an abstract method cannot be created (`TKO0620`). `--dump-vtable` shows the table the compiler built.
- **Give a base that will be deleted through a pointer a virtual destructor**, or the derived destructor never runs, and **repeat `virtual` at every level below it**: a plain destructor in the middle stops the chain.
- `a is T` is true for `T` and anything derived from it; guard every `cast( T ptr, @a )` with it. Prefer a virtual method to a chain of `is` tests.
- **Passing by value slices** an object to its base part. Pass objects with virtual methods `byref`.
- Use inheritance for "is a" and composition for "has a". A type extends one base, so an abstract type as a contract uses that base up.
- An `interface` lists methods and properties with no data or code. A `type` or `class` takes it on with an `implements` line and maps each member with `implements IFoo.Member`, explicitly; a missing one is `TKO1000`. `class` is a type that always extends `Object`, with its base on an `inherits` line.
- A type may implement any number of interfaces. Members of the same name in two interfaces are different members; a method's name need not match its member's, one method may provide several, and a private method may provide one.
- An interface value is a 16-byte reference. It converts from any implementing object or pointer, is called with `.`, and **does not own its object**: temporaries are refused (`TKO1012`), `delete` through it is refused (`TKO1013`), and the object must outlive it.

- `is` and `trycast` take interfaces on either side; `trycast` answers null instead of guessing.
- A plain `type` can implement an interface without a table pointer of its own.
- Generic interfaces are interfaces per instance. A constraint, `of T as IFoo`, is checked where the generic is used (`TKO1022`) and costs nothing in the body.

Exercises

1. Add a `Triangle` with a base and a height to the shapes program, without changing `Shape`, `Circle` or the loop. What did you have to write, and what did you not? {solution="ex14-01″}
2. Write a `Counter` plugin whose `Apply` appends the length of the text and counts its calls, and whose destructor prints the count. Create it with `new`, use it through a `Plugin ptr`, and delete it through that pointer. Which line makes the destructor run? {solution="ex14-02″}
3. Remove `virtual` from `vdtor.tiko`'s `Careful` destructor. Predict the output, then run it.
4. Change `ByValue` in `slicing.tiko` to take its parameter `byref`. Which lines of output change, and why?
5. A `Stack` that extends `Array`, or a `Stack` that has an `Array` field: argue for one. What can a caller do to the first that a stack should not allow?
6. Add a `Triangle` to `ifaceshapes.tiko`. Then give `Rect` a second interface, `IScalable`, with `sub Scale( byval k as float64 )`, and scale a rectangle through it. Check with `is` that a triangle is not scalable. Which existing code did you change? {solution="ex14-03″}
7. Rewrite `pluginhost.tiko` with an `IPlugin` interface. Who deletes the plugins now, and why can the host no longer do it through the interface?

Modules and Packages

Sooner or later a program uses two libraries written by people who never met, and both of them define `Parse`. With `#include` there is nothing to be done about it. A header is text pasted into the file that includes it, so both `Parse`s land in the same scope and the program does not compile. That is not the only cost. A header's helpers are visible to everyone who includes it. Its code is compiled again by every file that includes it. And when two libraries each need setting up before `main` runs, the order they are set up in depends on how the program happened to be linked.

A **module** is tiko's answer to all four. It is a directory of source files that compiles as one unit, shows the world only what it marks `public`, keeps its names behind its own name, is built once and cached, and is initialised in a fixed order: after everything it uses, before anything that uses it. A **package** is a set of modules shipped together, with a small `tiko.toml` saying what it is called and what it depends on.

This chapter builds a module, then several, then a package. Everything in it is a tiko extension: `#include` still works exactly as Chapter 16 describes, and a program that never writes `#import` compiles as if modules did not exist.

What `#include` cannot do

Here are two small header libraries. `csv.ti` counts the fields in a line; `ini.ti` finds where a value starts. Each is reasonable on its own:

examples/ch15/headers/csv.ti

```
'' csv.ti -- a header library: its code is pasted into every includer.
function Parse( byval s as ustring ) as int32
    '' the number of fields in a comma-separated line
    var n = 1
    for i as int32 = 1 to len( s )
        if mid( s, i, 1 ) = "," then n += 1
    next
    return n
end function
```

examples/ch15/headers/ini.ti

```
'' ini.ti -- another header library, by another author.
function Parse( byval s as ustring ) as int32
    '' where the value starts in a "key=value" line
    return instr( s, "=" ) + 1
end function
```

A program that wants both cannot have them:

examples/ch15/clash.tiko

```
#include once "headers/csv.ti"
#include once "headers/ini.ti"

print Parse( "a,b,c" )
```

```
headers/ini.ti(2,1): error TKO0600: 'parse' is already declared in this
scope
     |
   2 | function Parse( byval s as ustring ) as int32
     | ^
headers/csv.ti(2,1): note: declared here
     |
   2 | function Parse( byval s as ustring ) as int32
     | ^
```

Neither author did anything wrong. The only fix available to them is to wrap their library in a `namespace`, and every library author has to remember to do it. The problem is the mechanism: `#include` puts every name a library declares into the scope of whoever includes it.

A module is a directory

```
hello.tiko
greet/
    greet.tiko
```

examples/ch15/greet/greet.tiko

```
#module "greet"

public function Hello( byval who as ustring ) as ustring
    return "Hello, " & Capitalise_( who ) & "!"
end function

function Capitalise_( byval s as ustring ) as ustring
    return ucase( left( s, 1 ) ) & mid( s, 2 )
end function
```

examples/ch15/hello.tiko

```
#import "./greet"

print greet.Hello( "world" )
```

examples/ch15/hello.out

```
Hello, World!
```

Three rules are at work.

1. **Every file of a module begins with `#module "name"`, and the name is the directory's name.** Comments may come first; any other directive is `TKO0360`. A name that does not match its directory is `TKO0361`.
2. **`#import` comes before any code.** The path is quoted, uses `/` on every platform, and `./greet` is relative to the file that imports it. An `#import` after any statement — even after a `#define` or a `#cmdline` — is `TKO0362`.
3. **The module's names are reached through the module's name.** The import binds `greet`, the last segment of the path, and the program calls `greet.Hello`.

Because a module's name becomes a name in the program, no segment of an import path may be a keyword: `#import "./sub/inner"` is `TKO0364`, `'sub' is a keyword`. Name modules for what they hold — `text`, `report` — not after statements.

There is no list of files on the command line and no build script. `tiko hello.tiko` reads the `#import`, finds `greet/`, compiles it, and links it in.

Who can see what

`Hello` is marked `public`; `Capitalise_` is not. Reaching for it from outside is refused, and the message names the rule:

examples/ch15/notpublic.tiko

```
#import "./greet"

print greet.Capitalise_( "world" )        '' refused
```

```
notpublic.tiko(3,13): error TKO0860: 'capitalise_' is not public in
module 'main.greet'
     |
   3 | print greet.Capitalise_( "world" )        '' refused
     |             ^
```

`main.greet` is the module's **chain**: its package's name followed by its path. A program with no `tiko.toml` is the package `main`. The chain is how tiko tells two modules called `greet` apart.

A module has three levels of visibility, and **the default is the middle one**:

You write	Visible to	Use it for
`public`	every file that imports the module	the module's API
nothing	every file of this module	helpers the module's files share
`private`	this module, and nothing reached from outside it	helpers nothing else may ever call

That default is the opposite of a classic program's, where a procedure is visible everywhere unless it says `private`. It is the right way round for a library: a helper you forgot to mark never becomes part of your API, so you are free to change it later.

The difference between the last two is what each one means to the linker. Measured on the object `tiko -c` writes for a module named `gen` with one procedure at each level:

Level	Name in the object	Linux (ELF)	Windows (COFF)
`public`	`_ZN4MAIN3GEN3APIEv`	global	external
module-private	`_ZN4MAIN3GEN7SHARED_EEv`	global, **hidden**	external
`private`	`_ZN4MAIN3GEN7SECRET_EEv`	local	local

Every name is spelled from the chain, so two packages' `Parse` procedures can never collide in the linker either. A module-private procedure is still an external symbol, because code compiled *outside* the module sometimes has to call it: a generic's body, and an exported macro's expansion, are compiled in the importer. A `private` one is local to the module's object and cannot be reached that way at all. The section on generics shows the difference.

A public procedure may not name a type that is not public. An importer should never be handed a value of a type it cannot name, so `public function Make( ) as Secret`, where `Secret` has no `public`, is refused: `TKO0868: 'secret' is module-private but appears in the public declaration of 'make'`. Every type that appears in a public signature is public too.

A module with several files

A module may be as many files as it needs, and all of them form one module. They see each other's non-`private` names freely, **in any order**:

examples/ch15/report/report.tiko

```
#module "report"
```

```
public function Row( byval label as ustring, _
                     byval value as int32 ) as ustring
    return PadRight_( label ) & Format_( value )
end function
```

examples/ch15/report/format.tiko

```
#module "report"

'' Both helpers are module-private: report.tiko calls them, and
'' nothing outside the module can.
function PadRight_( byval s as ustring ) as ustring
    return left( s & space( LABEL_WIDTH ), LABEL_WIDTH )
end function

function Format_( byval n as int32 ) as ustring
    return right( space( 6 ) & str( n ), 6 )
end function

const LABEL_WIDTH = 10
```

examples/ch15/several.tiko

```
#import "./report"

print report.Row( "apples", 12 )
print report.Row( "pears", 3075 )
```

examples/ch15/several.out

```
apples        12
pears       3075
```

`report.tiko` calls `PadRight_` and `Format_`, which are declared in another file. `PadRight_` uses `LABEL_WIDTH`, which is declared *below* it. None of that needs a `declare` line, and there is no internal header gluing the files together, because a module is compiled as a single unit and its top-level declarations are collected before any of them is checked.

Inside a procedure body nothing changes: a local variable is still declared before it is used. And code outside any module keeps declare-before-use for everything, as Chapter 7 described.

Macros do not cross between the files of a module. A `#define` in `report.tiko` is not visible in `format.tiko`; each file starts from the same clean slate. A macro several files need belongs in a small header that each of them includes.

Imports do not cross between the files either. An import belongs to the file that writes it, so each file says where its own names come from. A sibling file that writes `util.` without an `#import "../util"` of its own gets `TKO0743: 'util' is not declared`, with a note pointing at the file that did import it: `'util' is imported by another file of this module; an import belongs to the file that writes it.`

Importing

`#import` has two optional parts, and the imported names can be brought into scope with `using`:

examples/ch15/shapes/shapes.tiko

```
#module "shapes"

public const PI_ = 3.14159265358979

public type Circle
    r as float64
    declare function Area( ) as float64
end type

function Circle.Area( ) as float64
    return PI_ * r * r
end function

public function UnitCircle( ) as Circle
    return type( 1.0 )
end function
```

examples/ch15/importing.tiko

```
#import "./shapes" as sh
#import "./greet"
using greet

dim c as sh.Circle
c.r = 2
print c.Area( )

var u = sh.UnitCircle( )
print u.r, sh.PI_

print Hello( "using" )
```

examples/ch15/importing.out

```
 12.56637061435916
 1             3.14159265358979
Hello, Using!
```

- **`as sh` renames the module for this program.** Use it when two imports end in the same name, or when a name is long. The module is still `shapes`; only the spelling here changes.
- **Types are reached the same way as procedures**: `dim c as sh.Circle`. A value of a module's type behaves exactly as one of your own: `c.Area( )` needs no qualification, because a method belongs to its type.
- **`using greet` makes `greet`'s public names usable bare**, as `using` does for a namespace.

When two modules you are `using` both supply a name, a bare use of it is refused rather than guessed:

examples/ch15/ambiguous.tiko

```
#import "./tag"
#import "./label"
using tag
using label

print Kind( )                                '' refused
```

```
ambiguous.tiko(6,7): error TKO0919: 'kind' is ambiguous: more than one
USING in effect declares it; qualify it
     |
   6 | print Kind( )                                '' refused
     |       ^
```

Write `tag.Kind( )` or `label.Kind( )` and move on. That is the qualified name doing its job: the question "which `Kind`?" has an answer written in the source.

Passing an import on

An import is private to the module that writes it. A module can pass one on to its own importers by adding `public` to the `#import`:

examples/ch15/kit/kit.tiko

```
#module "kit"
#import "../greet" public       '' re-exported: importers see kit.greet
#import "../shapes"             '' kit's own business

public function Banner( ) as ustring
```

```
    return greet.Hello( "kit" ) & " pi=" & left( str( shapes.PI_ ), 4 )
end function
```

examples/ch15/reexport.tiko

```
#import "./kit"

print kit.Banner( )
print kit.greet.Hello( "again" )
```

examples/ch15/reexport.out

```
Hello, Kit! pi=3.14
Hello, Again!
```

`kit` re-exports `greet`, so an importer of `kit` reaches it as `kit.greet`. `shapes` is imported without `public`, so it stays `kit`'s business:

examples/ch15/noleak.tiko

```
#import "./kit"

print kit.shapes.PI_                        '' refused
```

```
noleak.tiko(3,11): error TKO0745: the namespace 'kit' has no member named
'shapes'
     |
   3 | print kit.shapes.PI_                        '' refused
     |           ^
```

That is what makes a module's dependencies its own to change. Swapping `shapes` for something else inside `kit` cannot break a program that imports `kit`, because no such program could have used it.

Packages

So far every module has sat beside the program that uses it. Once modules are meant for more than one program, they become a **package**: a directory tree of modules with a `tiko.toml` at its root.

```
acme/
    tiko.toml
    config/config.tiko
    text/pad/pad.tiko
app/
```

```
    tiko.toml
    main.tiko
```

examples/ch15/acme/tiko.toml

```
# The package acme: two modules, and one setting with a default.
[package]
name = "acme"
version = "0.3.0"

[defines]
ACME_COLUMNS = "8"
```

examples/ch15/acme/config/config.tiko

```
#module "config"

'' ACME_COLUMNS comes from the package's tiko.toml, or from the
'' program that depends on acme -- never from the program's code.
public function Columns( ) as int32
    return ACME_COLUMNS
end function
```

examples/ch15/acme/text/pad/pad.tiko

```
#module "pad"
#import "acme/config"

public function Cell( byval s as ustring ) as ustring
    return "|" & left( s & space( config.Columns( ) ), _
                       config.Columns( ) )
end function
```

`pad` imports `acme/config` by its full path. Inside a package, a path that starts with the package's own name means "this package". The module `pad`'s chain is `acme.text.pad`, and the program still calls it `pad`: the last segment.

The program says where to find `acme`:

examples/ch15/app/tiko.toml

```
[package]
name = "app"

[dependencies]
acme = { path = "../acme" }
```

examples/ch15/app/main.tiko

```
#import "acme/text/pad"

print pad.Cell( "tiko" ); pad.Cell( "modules" ); "|"
```

examples/ch15/app/main.out

```
|tiko    |modules |
```

A `tiko.toml` is optional — the programs before this section had none — and you add one for four things:

1. **A stable name.** Without one, a package is called `main`. With one, the name is written in the code and goes wherever the code goes.
2. **A root.** The directory holding `tiko.toml` is where the package starts. A relative import may not climb above it (`TKO0142`).
3. **Dependencies that travel with the project.** `acme = { path = "../acme" }` is in the repository, not in anyone's environment, so a fresh checkout builds with a bare `tiko main.tiko`.
4. **Settings for a library.** That is `[defines]`, and it needs a section of its own.

The file is a small subset of TOML: tables, strings, integers, booleans, inline tables and arrays of strings. Anything it does not recognise is `TKO0144`, naming the file and line. `version` is recorded for a future package manager and not otherwise used.

Where an import is looked for

For `#import "acme/text/pad"`, tiko tries these in order and takes the first that has it:

	Where	Example
1	a relative path, from the importing file	`#import "./greet"`
2	the current package, when the first segment is its name	inside acme: `#import "acme/config"`
3	a dependency in the current package's `tiko.toml`	`acme = { path = "../acme" }`
4	each `-i` directory, then each directory in `TIKO_PATH`	`TIKO_PATH=C:\tikolibs`
5	the toolchain's own `pkg/` directory	`std`

Nothing found is `TKO0140`, and the message lists every place it looked. `std` is special: it is always the toolchain's, so no project can replace the standard library by accident.

And one build may not find two different directories both claiming the same package name (`TKO0149`).

Settings that cross the boundary, and settings that do not

`acme`'s `config` module reads `ACME_COLUMNS`, which no source file defines. It comes from the package's `[defines]`, and a program that depends on `acme` can override it in its own manifest:

examples/ch15/tuned/tiko.toml

```
[package]
name = "tuned"

[dependencies]
acme = { path = "../acme" }

[dependencies.acme.defines]
ACME_COLUMNS = "4"
```

examples/ch15/tuned/main.out

```
|tiko|modu|
```

The same `main.tiko` now prints four-column cells. That is the only way in. **A module never sees the macros of the program that imports it** — not its `#defines`, and not its `-d` options either. This is deliberate, and it is what lets a module be compiled once and cached: if a program's macros could reach into a module, every program would need its own copy of it.

PITFALL

`-d` does not reach a module. A module that tests a symbol with `#ifdef` is compiled without any symbol the importing program defines:

examples/ch15/cfg/cfg.tiko

```
#module "cfg"

public function Mode( ) as ustring
#ifdef VERBOSE
    return "verbose"
#else
    return "quiet"
#endif
end function
```

examples/ch15/hermetic.tiko

```
#import "./cfg"
#cmdline "-d VERBOSE"

#ifdef VERBOSE
print "main: VERBOSE is defined"
#endif
print "cfg:  "; cfg.Mode( )
```

examples/ch15/hermetic.out

```
main: VERBOSE is defined
cfg:  quiet
```

Both lines compile and run, and the module quietly takes its `#else` branch. Nothing warns you, because from the module's side nothing is wrong. A setting a module is meant to take belongs in its package's `[defines]`, which a program overrides in `tiko.toml`; or, better, in an ordinary parameter, which the compiler checks.

Initialisation

A module may have top-level statements, shared variables with constructors, and `constructor` and `destructor` subs. tiko runs them in one order, the same on every target and whatever order the objects are linked in:

> **A module is initialised after every module it imports, and before any code that imports it. Shutdown is the exact reverse.**

examples/ch15/logger/logger.tiko

```
#module "logger"

type Sink
    lines as int32
    declare constructor( )
    declare destructor( )
end type

constructor Sink( )
    print "  logger: sink opened"
end constructor

destructor Sink( )
```

```
    print "  logger: sink closed after"; lines; " line(s)"
end destructor

dim shared sink_ as Sink
print "  logger: top-level code"

public sub Info( byval s as ustring )
    sink_.lines += 1
    print "  log: "; s
end sub
```

examples/ch15/store/store.tiko

```
#module "store"
#import "../logger"

logger.Info( "store opening" )

sub Close_( ) destructor
    logger.Info( "store closing" )
end sub

public function Count( ) as int32
    return 3
end function
```

examples/ch15/initorder.tiko

```
#import "./store"

sub Early( ) constructor 101
    print "constructor 101"
end sub

print "main: store has"; store.Count( ); " items"
```

examples/ch15/initorder.out

```
constructor 101
  logger: sink opened
  logger: top-level code
  log: store opening
main: store has 3 items
  log: store closing
  logger: sink closed after 2 line(s)
```

Read the output top to bottom. `logger` is set up completely — its shared `Sink` constructed, then its top-level code — before `store`'s top-level code calls `logger.Info`. `store` is set up before `main` runs. At the end, `store`'s destructor runs while `logger` is still there to log it, and only then is the `Sink` destroyed. Each module is initialised once, however many modules import it.

That is the guarantee classic code cannot make. Without modules, two libraries' setup code runs in the order the linker happened to place it, and that order differs between Windows and Linux.

Numbered constructors keep their meaning. `constructor 101` asks for a slot by number, and those slots run before any module is initialised — which is why `Early` printed first. A number must be from 101 to 65535 (`TKO0422`). Use one only for something that genuinely must precede everything, and prefer the import order, which says *why* one thing comes before another.

Macros and generics across modules

Exporting a macro

A module's macros are its own unless it exports them by name, after defining them:

examples/ch15/check/check.tiko

```
#module "check"

#macro Ensure( cond )
    if ( cond ) = 0 then Fail_( #cond, __FILE__, __LINE__ )
#endmacro
#export Ensure

#define MAX_ITEMS 10
#export MAX_ITEMS

sub Fail_( byval what as ustring, byval f as ustring, _
           byval ln as int32 )
    print "check failed: "; what; " ("; f; " line"; ln; ")"
end sub
```

examples/ch15/macros.tiko

```
#import "./check"

'' A Fail_ of this program's own, which the macro must not call.
sub Fail_( byval what as ustring, byval f as ustring, _
           byval ln as int32 )
    print "wrong Fail_"
```

```
end sub

var items = 12
check.Ensure( items <= check.MAX_ITEMS )
check.Ensure( items > 0 )
print "done"
```

examples/ch15/macros.out

```
check failed: items <= 10 (macros.tiko line 10)
done
```

- **An exported macro is used qualified**, as `check.Ensure( … )`. A bare `Ensure( … )` is not a macro at all, so importing a module can never capture a name of yours.
- **Its body means what it meant in the module.** `Fail_` in `Ensure`'s body is `check`'s `Fail_`, even though the program cannot name it and has a `Fail_` of its own. Names the module does not declare — `items`, the macro's argument — are left as the caller wrote them.
- `__FILE__` and `__LINE__` expand where the macro is used, as they always do. And `#cond` shows the argument after expansion, so `check.MAX_ITEMS` appears as `10`.
- **The module's own headers keep their macros too.** When a public declaration names a type from one of the module's headers, your program reads that header to get the type, but the header's macros stop there. A module that wants a header shared whole, macros and all, says `public #include`. Headers from outside the package, such as `windows.bi`, are read as if you had included them yourself.

Exporting a generic

examples/ch15/coll/coll.tiko

```
#module "coll"

public type Stack( of T )
    declare sub Push( byval v as T )
    declare function Pop( ) as T
    declare function Count( ) as int32
private:
    items( any ) as T
end type

sub Stack( of T ).Push( byval v as T )
    redim preserve items( ubound( items ) + 1 )
    items( ubound( items ) ) = v
```

```
    Counted_( )
end sub

function Stack( of T ).Pop( ) as T
    var v = items( ubound( items ) )
    if ubound( items ) = 0 then
        erase items
    else
        redim preserve items( ubound( items ) - 1 )
    end if
    return v
end function

function Stack( of T ).Count( ) as int32
    return ubound( items ) + 1
end function

'' Module-private, yet reached from the importer: Push's body is
'' instantiated there, and calls it.
sub Counted_( )
    pushes += 1
end sub

public dim shared pushes as int32
```

examples/ch15/generics.tiko

```
#import "./coll"

dim nums as coll.Stack( of int32 )
nums.Push( 1 )
nums.Push( 2 )
print nums.Pop( ); nums.Count( )

dim names as coll.Stack( of ustring )
names.Push( "ada" )
print names.Pop( ); names.Count( )

print "pushes:"; coll.pushes
```

examples/ch15/generics.out

```
 2 1
ada 0
pushes: 3
```

A generic is instantiated by the program that uses it, so its body travels with the module's interface and is compiled in the importer. That is why `Counted_` is module-private rather than `private`: `Push`'s body calls it from the program's own code, and a module-private name is an external symbol that code can reach. Make the helper `private` and the instantiation fails with `TKO0743: 'counted_' is not declared`, because a `private` name does not exist outside the module's object.

Building

`tiko main.tiko` does everything: it finds the imports, builds each module once, and links the result. Four options show or change what it did.

`--print-graph` lists the modules a program imports, in the order they are built, and stops. Each line is a chain, the directory it came from, and what it imports:

```
> tiko --print-graph main.tiko
acme.config    ../acme/config
acme.text.pad  ../acme/text/pad  [acme.config]
```

`-v` reports, per module, whether the build cache had it. Three builds of the two programs above:

```
module acme.config: miss (new)
module acme.text.pad: miss (new)

module acme.config: hit
module acme.text.pad: hit

module acme.config: miss (define ACME_COLUMNS=4)
module acme.text.pad: miss (define ACME_COLUMNS=4)
```

The cache is one directory shared by every project of yours: `%LOCALAPPDATA%\tiko\modcache` on Windows, `~/.cache/tiko/modcache` on Linux (or under `$XDG_CACHE_HOME`), and wherever `TIKO_MODCACHE` names if you set it. Nothing is written into the project, so there is nothing to add to `.gitignore`, and a module two projects build the same way is compiled once for both. A build with different settings is a different entry, as the third build shows. The key is the module's content, not file dates, so touching a file changes nothing.

What a change rebuilds, measured on `acme`:

You change	Rebuilt
a procedure's body in `pad`	`pad` only
something `public` in `config`	`config`, and `pad`, because it imports `config`
a setting in `[defines]`, or `-g`, `-exx`, the target	every module it affects

If you ever doubt the cache, `tiko --clean-module-cache` empties it, and `--no-module-cache` builds without reading or writing it. Nothing is lost but time.

`--emit-tki` prints a module's **interface**, the declarations an importer is compiled against, as ordinary tiko source:

```
> tiko --emit-tki thermo
'' tiko-interface 1
'' module main.thermo
namespace __tko_mod_main_thermo
declare function Round1_( byval x as float64 ) as float64
public declare function ToFahrenheit( byval c as float64 ) as float64
end namespace
```

(The real output carries a `#line` before each declaration, pointing back at the source.) The module-private `Round1_` is in it, without `public`: that is how a generic or macro body compiled in the importer can reach it, while the importer's own code cannot.

Objects, libraries and DLLs

The other output kinds work on a program that imports:

- `tiko -c -m main main.tiko -o out/main.o` compiles to objects, writing each module beside the output as its chain — `out/main.store.o`, `out/main.logger.o`. Linking `out/main.o` alone later finds the modules' objects through a record inside it.
- `tiko -lib` puts every module's object into the archive, so the library is complete on its own.
- `tiko -dll` links the modules in and exports only what the root marks `export`. A module's names never are.

`__TKO_MODULE__` is defined while a module is compiled and not otherwise, and `__TIKO_COMPILER__` is defined by tiko always: the test for "am I being compiled by tiko", for code that must also build elsewhere.

The standard library as a package

The standard library is the package `std`, shipped in the toolchain's `pkg/` directory. Its modules are `strings`, `fs`, `containers`, `chrono`, `tasks`, `outcome`, `numeric`, `money`, `logs`, `tui` and `memory`, and the library chapters (Part IV) use them throughout:

examples/ch15/stdlib.tiko

```
#import "std/strings"
#import "std/fs"

print strings.Replace( "a-b-c", "-", "+" )
print strings.StartsWith( "tiko-lang", "tiko" )
print strings.Reverse( "tiko" )

print fs.PathStem( "notes.txt" ); " "; fs.PathExtension( "notes.txt" )
print fs.PathFileName( "logs/today.txt" )
```

examples/ch15/stdlib.out

```
a+b+c
true
okit
notes .txt
today.txt
```

These are ordinary modules, built the way this chapter describes: compiled once, cached, and initialised in import order. Everything in the chapter applies to them:

- **`using strings` makes `Replace` a bare name**, as for any module. Two `using`s that supply the same name – `strings.Contains` and `containers.Contains` – are `TKO0919` when the name is used bare; qualify it.
- **Their macros are exported**, and used qualified: `logs.LogInfo( "started" )`.
- **A statement may need one.** `spawn` and `select channel` lower to `std/tasks`, so a file that uses them imports it (`TKO0423` otherwise), and builds with `-mt`.

Converting a header library into a module

Most existing libraries are headers. Moving one to a module is mostly mechanical, and the result is easier to use and cheaper to build. Here is a header library, the classic way:

examples/ch15/classic/thermo.ti

```
'' thermo.ti -- a header library, the classic way.
#pragma once

namespace Thermo
```

```
private function Round1_( byval x as float64 ) as float64
    return int( x * 10 + 0.5 ) / 10
end function

private function ToFahrenheit( byval c as float64 ) as float64
    return Round1_( c * 9 / 5 + 32 )
end function

end namespace
```

examples/ch15/classic.tiko

```
#include once "classic/thermo.ti"

print Thermo.ToFahrenheit( 21.5 )
```

And the same library as a module, with a program using it:

examples/ch15/thermo/thermo.tiko

```
'' thermo -- the same library, as a module.
#module "thermo"

function Round1_( byval x as float64 ) as float64
    return int( x * 10 + 0.5 ) / 10
end function

public function ToFahrenheit( byval c as float64 ) as float64
    return Round1_( c * 9 / 5 + 32 )
end function
```

examples/ch15/converted.tiko

```
#import "./thermo"

print thermo.ToFahrenheit( 21.5 )
```

Both programs print the same line:

examples/ch15/converted.out

```
 70.7
```

The recipe, file by file:

Step	Header library	Module
Place	`thermo.ti`	`thermo/thermo.tiko`, beginning `#module "thermo"`
Guard	`#pragma once`	gone: a module file is never included
Namespace	`namespace Thermo … end namespace`	gone: the module *is* the namespace
API	`private function ToFahrenheit` (every includer gets a copy)	`public function ToFahrenheit`
Helpers	`private function Round1_`	no keyword: module-private
Library-internal includes	`#include once "other.ti"`	`#import` of the module that holds it
API macros	`#define`	`#define` then `#export`; or, better, a procedure
Configuration macros	"define this before you include me"	`[defines]` in the package's `tiko.toml`

Two steps need judgement. **Headers that include each other** cannot become two modules, because imports may not form a cycle (`TKO0141`, which names the cycle); merge them into one module, or move what they share into a third. And **a system header your public signatures mention** — a Windows handle type, say — stays an `#include` in the module. The module's interface records that it needs the header, and including it again in the importer is harmless: it is the same file, read once.

What it costs

- **A module is compiled once and cached**, so the first build of a program that imports a large module is slow and every later build is not. A header library is compiled again by every file that includes it, every time.
- **Changing a public declaration rebuilds every module that imports it**, directly or through others. Changing a body rebuilds one module. Keep public interfaces stable and the rebuild stays small.
- **Calls across a module boundary are ordinary calls.** A module is not a separate binary at run time; the linker joins everything into one executable, as before.
- **Generics are instantiated in the importer**, so each program that uses `coll.Stack( of int32 )` compiles that instance itself, and a program that instantiates many pays in compile time and binary size exactly as Chapter 17 described.

- **The cache costs disk space** and is never cleaned on its own. `--clean-module-cache` empties it.

Summary

- `#include` pastes a library into its includer: two libraries' names collide (`TKO0600`), helpers are visible to all, and the code is compiled again by every includer.
- **A module is a directory**; every file begins `#module "name"`, matching the directory (`TKO0360`, `TKO0361`). `#import "./greet"` comes before any code (`TKO0362`) and binds `greet`.
- **Module-private is the default.** `public` exports; `private` keeps a name inside the module's object. Names are spelled from the module's chain, so they never collide in the linker either. Reaching a module-private name is `TKO0860`.
- **Inside a module, declaration order does not matter**, across files or within one. Macros do not cross between its files.
- `#import … as x` renames; `using` makes public names bare, and a name two `using`s supply is `TKO0919`. `#import … public` re-exports; otherwise imports never leak (`TKO0745`).
- **A package is a directory tree with a `tiko.toml`**: a name, a root, `path` dependencies, and `[defines]` a program overrides through `[dependencies.NAME.defines]`.
- **A module never sees its importer's macros or `-d` options**, which is what makes it cacheable, and which silently takes a module's `#else` branch if you expect otherwise.
- **Modules initialise after what they import and finalise in reverse**, once each, the same on every target. Numbered constructors run first.
- `#export NAME` shares a macro, used qualified and hygienic. An exported generic is instantiated in the importer, so its helpers must be module-private, not `private`.
- `--print-graph`, `-v`, `--no-module-cache`, `--clean-module-cache` and `--emit-tki` show and control the build; `-c`, `-lib` and `-dll` work on programs that import.
- `std` is a package in the toolchain, and its modules are ordinary modules.
- Converting a header library: add `#module`, drop the guard and the namespace, mark the API `public`, unmark the helpers, and turn configuration macros into `[defines]`.

Exercises

1. Write a module `temp` with public `CToF` and `FToC` and a module-private rounding helper, and a program that prints a table from −10 to 30 degrees in steps of 10, converting each value there and back. {solution="ex15-01″}
2. Write a module `counter` whose top-level code prints that it is initialising and whose `destructor` sub prints that it is done, with a public shared `hits`. Import

it from two modules, `left` and `right`, that each add one to `hits`, and import all three from a program. Predict how often `counter` initialises, then run it. {solution="ex15-02″}

3. Move `report`'s `LABEL_WIDTH` into a third file of the module. What did you have to change elsewhere, and why?
4. Make `acme`'s column count a parameter of `pad.Cell` instead of a `[defines]` setting. What does the program gain, and what does the package lose?
5. Take a header library of your own and convert it with the recipe. Which step needed a decision rather than an edit?

Namespaces, Headers and the Preprocessor

A program outgrows one file long before it outgrows one author. The geometry goes in one file, the measuring in another, the program that uses them in a third, and each has to know enough about the others to call them. Chapter 15 builds that out of modules, which is what new code should do. This chapter is the machinery underneath: files compiled one at a time and joined by the linker, headers that tell each file what the others provide, namespaces that keep their names apart, and the **preprocessor**, which rewrites the source before the compiler reads a word of it.

You need all of it anyway. Every binding to a C library is a header. Every program that must behave differently on Windows and Linux says so with the preprocessor. And a module is itself built from these parts, so the rules here are the ones that hold underneath Chapter 15.

A program in several files

Here is a small program in three source files. The part they share goes in a **header**, a `.ti` file that declares what exists without defining it:

examples/ch16/geometry.ti

```
'' geometry.ti -- what the three files of the program share.
#pragma once

type Point
    x as float64
    y as float64
end type

'' Defined in shapes.tiko.
declare function Distance( byval a as Point, _
                           byval b as Point ) as float64
declare function Perimeter( pts() as Point ) as float64

'' Defined in measure.tiko, read everywhere.
extern measured as int32
```

`declare` promises a procedure that some other file defines; `extern` promises a variable. The two source files keep those promises. `shapes.tiko` defines the procedures, and `measure.tiko` owns the counter:

examples/ch16/shapes.tiko

```
'' shapes.tiko -- distances and perimeters.
#include once "geometry.ti"

private function Square( byval v as float64 ) as float64
    return v * v
end function

function Distance( byval a as Point, byval b as Point ) as float64
    measured += 1
    return sqr( Square( a.x - b.x ) + Square( a.y - b.y ) )
end function

function Perimeter( pts() as Point ) as float64
    dim total as float64 = 0
    for i as int32 = lbound( pts ) to ubound( pts )
        var j = iif( i = ubound( pts ), lbound( pts ), i + 1 )
        total += Distance( pts( i ), pts( j ) )
    next
    return total
end function
```

examples/ch16/measure.tiko

```
'' measure.tiko -- owns the counter that geometry.ti declares.
#include once "geometry.ti"

dim shared measured as int32
```

The main file includes the same header and uses all of it:

examples/ch16/geometry.tiko

```
'' geometry.tiko -- the main module.
#include once "geometry.ti"

dim box( 0 to 3 ) as Point = { ( 0, 0 ), ( 4, 0 ), ( 4, 3 ), ( 0, 3 ) }

print "diagonal: "; Distance( box( 0 ), box( 2 ) )
print "perimeter:"; Perimeter( box() )
print "distances measured:"; measured
```

examples/ch16/geometry.out

```
diagonal:  5
perimeter: 14
distances measured: 5
```

Build it by naming every source file; the first one named is the program:

```
tiko geometry.tiko shapes.tiko measure.tiko -o geometry
```

Each `.tiko` file is compiled **on its own**, as if the others did not exist. That is why each one includes the header: a file knows only what it has been told. The linker then joins the compiled files and connects every `declare` and `extern` to the one definition behind it. `dim shared measured` in `measure.tiko` is that definition; `extern measured` in the header lets the other two files use it.

`#include` pastes a file's text in place. `#include once` pastes it only the first time, and `#pragma once` at the top of a header says the same thing from the inside, so the header is safe however many times it is reached. `geometry.ti` says both ways, which costs nothing.

Where `#include` looks

examples/ch16/platform.tiko

```
'' platform.tiko -- one source, two platforms.
#include once "include/platform.ti"

print "platform: "; PLATFORM
print "pointers:"; POINTER_BITS; " bits"
print "a path:   logs"; PATH_SEP; "today.txt"
#ifdef __FB_MSVC__
print "target:   windows-x86_64"
#endif
```

examples/ch16/include/platform.ti

```
'' platform.ti -- what this program needs to know about where it runs.
#pragma once
#include once "separator.ti"

#if defined( __FB_WIN32__ )
    const PLATFORM = "Windows"
#elseif defined( __FB_LINUX__ )
    const PLATFORM = "Linux"
#elseif defined( __FB_DARWIN__ )
    const PLATFORM = "macOS"
#else
    #error "platform.ti: this platform is not supported"
#endif

#ifdef __FB_64BIT__
    const POINTER_BITS = 64
```

```
#else
    const POINTER_BITS = 32
#endif
```

examples/ch16/include/separator.ti

```
'' separator.ti -- found beside platform.ti, which includes it.
#pragma once

#ifdef __FB_WIN32__
    const PATH_SEP = "\"
#else
    const PATH_SEP = "/"
#endif
```

`platform.tiko` names `include/platform.ti`, a path relative to itself. `platform.ti` then includes `separator.ti` with no directory at all, and finds it beside `platform.ti`: **a name is looked for first beside the file that includes it**, not beside the program's first file. That is what lets a header bring its own helpers along without knowing where it will be used from.

When the file is not there, tiko tries the directory the compiler was run in, then each directory given with `-i`, in the order they were given, and finally its own `inc/`, which holds the C bindings and the classic headers that ship with the toolchain. The first place that has the file wins. So a header of your own in the working directory, named like one of the toolchain's, is the one your files get when they include that name, in place of the toolchain's. Give your headers names of their own, or include them by a path relative to the file that needs them.

Headers or modules?

Chapter 15 opens with the two problems headers cannot solve: every name a header declares lands in the scope of whoever includes it, and its code is compiled again by every file that does. For your own code, a module is the better tool. Headers remain the right one for a binding to a C library, for macros meant to be shared between files, and for existing code that is already written as headers.

Linkage

Two files compiled on their own can both define a name, and neither compile notices. The linker does:

examples/ch16/clash.tiko

```
'' clash.tiko -- two files, each with a sub called Init.
sub Init( )
```

```
    print "report ready"
end sub

Init( )
```

examples/ch16/clashother.tiko

```
'' clashother.tiko -- the other file.
sub Init( )
    print "export ready"
end sub
```

```
lld-link: error: duplicate symbol: INIT
clang: error: linker command failed with exit code 1 (use -v to see
invocation)
tiko-lang: error TKO0116: link failed (exit 1)
```

(The linker's two `>>> defined at` lines, which name the temporary object files, are left out.)

A procedure is **public** by default: its name is visible to the linker, so every other file can call it, and two files cannot both have one. **`private`** keeps a procedure inside its own file. Make one of the two `Init`s private and the program links. The file that owns it reaches it through a public procedure:

examples/ch16/privatefixother.tiko

```
'' privatefixother.tiko -- its Init is its own business.
private sub Init( )
    print "export ready"
end sub

sub StartExport( )
    Init( )
end sub
```

examples/ch16/privatefix.tiko

```
'' privatefix.tiko -- two subs called Init, and the program links.
declare sub StartExport( )

sub Init( )
    print "report ready"
end sub
```

```
Init( )
StartExport( )
```

examples/ch16/privatefix.out

```
report ready
export ready
```

Make helpers `private` as a habit. A public helper is a name another file can collide with, and an entry point nobody meant to offer.

Private by default, for a region

Marking every helper costs a word on each one. `#pragma push( private, true )` turns the default around until the matching `#pragma pop( private )`. Inside the region a procedure is private unless it says `public`:

examples/ch16/pragmaother.tiko

```
'' pragmaother.tiko -- one public entry point over private helpers.
#pragma push( private, true )

sub Validate( byval n as int32 )
    print "validating"; n
end sub

sub Store( byval n as int32 )
    print "storing"; n
end sub

'' Inside the region, but it says so: visible to the program.
public sub Flush( )
    print "flushing"
end sub

#pragma pop( private )

sub Save( byval n as int32 )
    Validate( n )
    Store( n )
end sub
```

examples/ch16/pragma.tiko

```
'' pragma.tiko -- Save and Flush are all this file can reach.
declare sub Save( byval n as int32 )
declare sub Flush( )
```

```
Save( 42 )
Flush( )
```

examples/ch16/pragma.out

```
validating 42
storing 42
flushing
```

`pragma.tiko` can reach `Save`, which follows the region, and `Flush`, which says `public` inside it. Declaring `Validate` there and calling it would fail at the link: no file offers a public `Validate`.

Shared variables

You have seen one way to share a variable: `dim shared` in exactly one file and `extern` in the header. The other is **`common`**. Every file that declares the same `common shared` variable gets the same storage, and no file has to be the owner:

examples/ch16/tally.tiko

```
'' tally.tiko -- both files declare the same common variable.
common shared hits as int32
declare sub Hit( )

Hit( )
Hit( )
Hit( )
print "hits:"; hits
```

examples/ch16/tallyother.tiko

```
'' tallyother.tiko -- the same declaration, the same storage.
common shared hits as int32

sub Hit( )
    hits += 1
end sub
```

examples/ch16/tally.out

```
hits: 3
```

Prefer `extern`. With `extern`, one file says where the variable lives and the header says what it is, so a mismatch is caught when the files are compiled. `common` makes every file restate the declaration, and nothing checks that they agree: declared `int32` in one file and `float64` in the other, the program still links, and counts wrong.

Startup and shutdown

A program sometimes has work to do before its first statement, and after its last. A sub marked **`constructor`** runs before the main program; one marked **`destructor`** runs after it. A number after the keyword is a **priority**:

examples/ch16/startup.tiko

```
'' startup.tiko -- the main module, with the log's setup and teardown.
private sub OpenLog( ) constructor 101
    print "101: open the log"
end sub

private sub CloseLog( ) destructor 101
    print "101: close the log"
end sub

print "main runs"
```

examples/ch16/startupconfig.tiko

```
'' startupconfig.tiko -- the configuration, which needs the log.
private sub LoadConfig( ) constructor 102
    print "102: load the config"
end sub

private sub SaveConfig( ) destructor 102
    print "102: save the config"
end sub

'' Top-level code in a file that is not the main module.
print "startupconfig.tiko: top-level code"
```

examples/ch16/startup.out

```
101: open the log
102: load the config
startupconfig.tiko: top-level code
main runs
102: save the config
101: close the log
```

Constructors run lowest priority first, and destructors run in the opposite order, so whatever started first is shut down last: the configuration is saved before the log it writes to is closed. Give anything that depends on something else a higher number than the thing it depends on.

Look at where `startupconfig.tiko`'s own top-level code ran: before the main program, after the constructors. Only one file is the **main module**, whose top-level code is the program. Every other file's top-level code runs as part of startup. The main module is the first file named, unless `-m` names another:

examples/ch16/whichmain.tiko

```
'' whichmain.tiko -- compiled first, but not the main module.
#cmdline "-m whichmainother"

print "whichmain.tiko: top-level code"
```

examples/ch16/whichmainother.tiko

```
'' whichmainother.tiko -- named by -m, so its code is the program.
print "whichmainother.tiko: the main program"
```

examples/ch16/whichmain.out

```
whichmain.tiko: top-level code
whichmainother.tiko: the main program
```

`#cmdline "-m whichmainother"` puts the option in the source, so the listing builds the same way whoever compiles it.

Namespaces

A `private` procedure is hidden from other files, but that does not help when two parts of the *same* program both want a function called `Area`. A **namespace** gives names a family name:

examples/ch16/namespaces.tiko

```
namespace Geo
    const ORIGIN = 0

    function Area( byval w as float64, byval h as float64 ) as float64
        return w * h
    end function

    namespace Units
        const CM_PER_M = 100
    end namespace
end namespace

'' A namespace can be reopened, here or in another file.
namespace Geo
    function Perimeter( byval w as float64, _
```

```
                        byval h as float64 ) as float64
        return 2 * ( w + h )
    end function
end namespace

print "area, perimeter and centimetres per metre:"
print Geo.Area( 2, 3 ), Geo.Perimeter( 2, 3 )
print Geo.Units.CM_PER_M

sub Report( )
    using Geo
    print Area( 4, 5 ), Units.CM_PER_M
end sub

Report( )
```

examples/ch16/namespaces.out

```
area, perimeter and centimetres per metre:
 6             10
 100
 20            100
```

- **Outside the namespace, a member is written qualified**: `Geo.Area`, `Geo.Units.CM_PER_M`.
- **Namespaces nest**, and the path is written out in full.
- **A namespace can be reopened.** The second `namespace Geo` adds `Perimeter` to the same namespace, and another file can do the same.
- **`using Geo`** makes Geo's members usable without the prefix, from the `using` to the end of the block that contains it: here, the sub `Report`. Written at the top of a file, it lasts to the end of the file.

How a name is found

`using` makes a namespace's names *available*. It does not make them *preferred*:

examples/ch16/lookup.tiko

```
namespace Defaults
    const LIMIT = 20
    function Label( ) as ustring
        return "from Defaults"
    end function
end namespace

const LIMIT = 10
```

```
function Label( ) as ustring
    return "from the module"
end function

sub Show( )
    using Defaults
    print LIMIT, Label( )
end sub

print "inside Show, under using Defaults:"
Show( )
print "qualified:"
print Defaults.LIMIT, Defaults.Label( )
```

examples/ch16/lookup.out

```
inside Show, under using Defaults:
 10           from the module
qualified:
 20           from Defaults
```

Inside `Show`, `LIMIT` and `Label` are both declared twice: once in `Defaults`, which `using` brings in, and once at module level. **A name declared on the way out — in the procedure, then in the enclosing scopes, then the module — beats every `using`.** The namespace's members are reached only where nothing closer has the name. To mean the namespace's, qualify it.

When two `using`s both supply a name and nothing closer declares it, tiko does not pick one:

examples/ch16/ambiguous.tiko

```
namespace Report
    function Title( ) as ustring
        return "Report"
    end function
end namespace

namespace Archive
    function Title( ) as ustring
        return "Archive"
    end function
end namespace

using Report, Archive
print Title( )
```

```
ambiguous.tiko(14,7): error TKO0919: 'title' is ambiguous: more than one
USING in effect declares it; qualify it
     |
  14 | print Title( )
     |       ^
```

Macros

The preprocessor reads the source before the compiler does, and **macros** are its main tool. A macro is a name for a piece of text, and the preprocessor replaces every use of the name with the text:

examples/ch16/macros.tiko

```
#define LIMIT 100
#define SQUARE( x ) ( ( x ) * ( x ) )
#define AS_TEXT( x ) #x
#define PASTE( prefix, n ) prefix##n

#macro SWAP_VALUES( a, b )
    scope
        var t_ = a
        a = b
        b = t_
    end scope
#endmacro

dim as int32 width1 = 3, width2 = 8
print "limit:"; LIMIT
print "square of 1 + 2:"; SQUARE( 1 + 2 )
print "as text: "; AS_TEXT( width1 + width2 )
print "pasted:"; PASTE( width, 2 )
SWAP_VALUES( width1, width2 )
print "after the swap:"; width1; width2
```

examples/ch16/macros.out

```
limit: 100
square of 1 + 2: 9
as text: width1 + width2
pasted: 8
after the swap: 8 3
```

- **`#define LIMIT 100`** is an object-like macro: `LIMIT` becomes `100` wherever it appears.
- **`#define SQUARE( x ) ...`** takes arguments. The text of each argument is substituted for the parameter, unevaluated.

- **`#x`** turns an argument into a string literal, so `AS_TEXT( width1 + width2 )` is the text `width1 + width2`, not the sum.
- **`prefix##n`** pastes two pieces into one token: `PASTE( width, 2 )` becomes the identifier `width2`.
- **`#macro ... #endmacro`** defines a macro that spans several lines, which is how a macro expands to whole statements. `SWAP_VALUES` wraps its temporary in a `scope` so that two uses in one procedure do not both declare `t_`.

A macro cannot take the name of a reserved word: `#define print 1` is `TKO0371`, "'print' cannot be defined as a macro: it is a reserved word".

PITFALL

A macro substitutes text, not values. The parentheses in `SQUARE` are not decoration:

examples/ch16/badsquare.tiko

```
#define BAD_SQUARE( x ) x * x
#define SQUARE( x ) ( ( x ) * ( x ) )

print "BAD_SQUARE( 1 + 2 ) ="; BAD_SQUARE( 1 + 2 )
print "SQUARE( 1 + 2 )     ="; SQUARE( 1 + 2 )
print "100 / SQUARE( 5 )   ="; 100 / SQUARE( 5 )
```

examples/ch16/badsquare.out

```
BAD_SQUARE( 1 + 2 ) = 5
SQUARE( 1 + 2 )     = 9
100 / SQUARE( 5 )   = 4
```

`BAD_SQUARE( 1 + 2 )` becomes `1 + 2 * 1 + 2`, which is 5. Wrap every parameter in parentheses, and the whole body too: without the outer pair, `100 / SQUARE( 5 )` would divide by the first `5` and multiply by the second. A function has neither problem, and should be the first thing you reach for. Use a macro for what a function cannot do.

What a function cannot do is generate code. This macro writes an `enum` and a function that names its values, from one list:

examples/ch16/enumnames.tiko

```
'' One list of names, written once: the enum and its spelling.
#macro DEFINE_ENUM( name, a, b, c )
    enum name
        a
        b
```

```
        c
    end enum

    function name##_Name( byval v as name ) as ustring
        select case v
        case a : return #a
        case b : return #b
        case c : return #c
        end select
        return "?"
    end function
#endmacro

DEFINE_ENUM( Colour, RED, GREEN, BLUE )
DEFINE_ENUM( Size, SMALL, MEDIUM, LARGE )

print Colour_Name( GREEN ); " "; Size_Name( LARGE )
for c as Colour = RED to BLUE
    print c; " "; Colour_Name( c )
next
```

examples/ch16/enumnames.out

```
GREEN LARGE
 0 RED
 1 GREEN
 2 BLUE
```

Each value is written once, so the names can never disagree with the enum. `name##_Name` builds the function's name from the enum's, and `#a` turns each value into its own spelling.

The built-in macros

A family of built-in macros works on macro arguments themselves:

examples/ch16/builtins.tiko

```
#define SIZES 8, 16, 32, 64

print "count:"; __FB_ARG_COUNT__( SIZES )
print "third:"; __FB_ARG_EXTRACT__( 2, SIZES )
#define PAIR key = value
print "left of =: "; __FB_QUOTE__( __FB_ARG_LEFTOF__( PAIR, = ) )
print "right of =: "; __FB_QUOTE__( __FB_ARG_RIGHTOF__( PAIR, = ) )
print "evaluated:"; __FB_EVAL__( 6 * 7 )
```

```
dim as int32 total32 = 32
print "joined:"; __FB_JOIN__( total, 32 )
__FB_UNQUOTE__( "print ""unquoted into code""" )

'' A name no other use of the macro can collide with.
#macro TIMES_TWO( v )
    __FB_UNIQUEID_PUSH__( tmp )
    var __FB_UNIQUEID__( tmp ) = v
    print "twice"; v; " is"; __FB_UNIQUEID__( tmp ) * 2
    __FB_UNIQUEID_POP__( tmp )
#endmacro
TIMES_TWO( 5 )
TIMES_TWO( 9 )
```

examples/ch16/builtins.out

```
count: 4
third: 32
left of =: key
right of =: value
evaluated: 42
joined: 32
unquoted into code
twice 5 is 10
twice 9 is 18
```

Macro	Gives
`__FB_ARG_COUNT__( ... )`	how many comma-separated arguments it was given
`__FB_ARG_EXTRACT__( n, ... )`	the argument at position `n`, counting from 0
`__FB_ARG_LEFTOF__( arg, sep )`, `__FB_ARG_RIGHTOF__( arg, sep )`	the text on either side of `sep`
`__FB_QUOTE__( text )`, `__FB_UNQUOTE__( "text" )`	text as a string literal, and a string literal as text
`__FB_EVAL__( expr )`	the constant expression's value, worked out by the preprocessor
`__FB_JOIN__( a, b )`	`a` and `b` pasted into one token, like `##`
`__FB_UNIQUEID_PUSH__( id )`, `__FB_UNIQUEID__( id )`, `__FB_UNIQUEID_POP__( id )`	a name no other expansion can produce

The last row is what makes a macro safe to use twice. `TIMES_TWO` needs a temporary, and a fixed name would be declared twice the second time the macro is used. `__FB_UNIQUEID__( tmp )` names a fresh one for each expansion.

Conditional compilation

`#if`, `#ifdef`, `#elseif`, `#else` and `#endif` choose which lines are compiled at all. A line in a branch not taken is never compiled, so it can call things that do not exist on that platform. `platform.ti`, above, is the usual use: one source, a different constant on each platform. The platform symbols tiko defines:

Symbol	Defined on
`__FB_WIN32__`	Windows
`__FB_LINUX__`, `__FB_UNIX__`	Linux
`__FB_64BIT__`	every 64-bit target
`__FB_MSVC__`	the `windows-x86_64` target
`__TIKO_VERSION__`	always: tiko's version, as a string

You define your own on the command line: `-d NAME=value` is `#define NAME value` before the first line of every file.

PITFALL

`#ifdef` asks whether a name is defined, not what it is.

examples/ch16/ifdeftrap.tiko

```
'' TRACE is 0 to turn tracing off -- or so the author thought.
#define TRACE 0

#ifdef TRACE
print "#ifdef TRACE: tracing is on"
#endif

#if TRACE
print "#if TRACE: tracing is on"
#else
print "#if TRACE: tracing is off"
#endif
```

examples/ch16/ifdeftrap.out

```
#ifdef TRACE: tracing is on
#if TRACE: tracing is off
```

`TRACE` is defined, as `0`, so every `#ifdef TRACE` in the program is true and the tracing it was meant to turn off is compiled in. Choose one convention per switch and keep to it: either a switch is defined or not, and you test it with `#ifdef`, or it always has a value, and you test it with `#if`. `#if defined( NAME ) andalso NAME <> 0` covers a name that might be either.

Compile-time checks

The preprocessor can also ask questions about the program, and stop the build when the answer is wrong:

examples/ch16/checks.tiko

```
'' Questions answered while compiling, not while running.
dim as int32 count = 0
dim as int64 total = 0

#if defined( __FB_64BIT__ ) andalso defined( __TIKO_VERSION__ )
print "a 64-bit build, by tiko"
#endif

#if typeof( count ) = typeof( int32 )
print "count is an int32"
#endif
#if typeof( total ) <> typeof( count )
print "total is not the same type as count"
#endif

'' The build stops here if the record's layout ever changes.
type Header
    magic as uint32
    size as uint32
end type
#assert sizeof( Header ) = 8
print "Header is"; sizeof( Header ); " bytes"
```

examples/ch16/checks.out

```
a 64-bit build, by tiko
count is an int32
total is not the same type as count
Header is 8 bytes
```

- **`defined( NAME )`** is true when `NAME` has been declared — as a macro, or as anything else above the line — and it combines with `andalso`, `orelse` and `not`.

- **`#if typeof( a ) = typeof( b )`** compares two types exactly: `int32` is not `int64`, and a pointer is not what it points to.
- **`#assert`** stops the build when its condition is false. `Header` is a record written to files; if a field is ever added, the build stops at the `#assert`, before a file in the old layout is misread.

`#error` stops the build with a message of your own:

examples/ch16/unsupported.tiko

```
'' This program needs a configuration chosen on the command line.
#ifndef CONFIG
    #error "build with -d CONFIG=1 (development) or 2 (release)"
#else
    print "configuration"; CONFIG
#endif
```

```
unsupported.tiko(3,5): error TKO0309: #error: "build with -d CONFIG=1
(development) or 2 (release)"
     |
   3 |     #error "build with -d CONFIG=1 (development) or 2 (release)"
     |     ^
```

Built with `-d CONFIG=1`, the same file prints `configuration 1`. The `print` sits in the `#else` branch for a reason: `#error` reports, and the compiler carries on reading the rest of the file for more errors, so a `print` that used `CONFIG` after the `#endif` would add a second error, about a name that is not declared. Keep the code that needs the setting in the branch that has it.

`#print` is the gentle form. `#print the build is starting` reports its text as a note, `TKO0315`, and the build goes on.

Source information and pragmas

The compiler knows where every line came from, and tells you:

examples/ch16/sourceinfo.tiko

```
sub Report( )
    print "in "; __FUNCTION__; ", line"; __LINE__
end sub

print "line:"; __LINE__
Report( )
print "an ISO date has 10 characters:"; len( __DATE_ISO__ )
#line 1000 "generated.tiko"
print "now:"; __LINE__; " of "; __FILE__
```

```
#ifdef __TIKO_VERSION__
print "compiled by tiko"
#endif
```

examples/ch16/sourceinfo.out

```
line: 5
in REPORT, line 2
an ISO date has 10 characters: 10
now: 1001 of generated.tiko
compiled by tiko
```

- **__LINE__** is the line it is written on, and **__FILE__** the file's name.
- **__FUNCTION__** is the enclosing procedure's name, in upper case.
- **__DATE_ISO__** is the date of the build as `yyyy-mm-dd`, which is why the listing prints only its length: the date changes every day, and an expected output cannot.
- **`#line 1000 "generated.tiko"`** tells the compiler that the *next* line is line 1000 of `generated.tiko`. A program that writes tiko source for another program uses it, so errors point at the file a person edits rather than the generated one.

Diagnostics use the same information, so after a `#line` they point where it says.

Pragmas are instructions to the compiler. You have met `#pragma once` and `#pragma push( private, true )`. `#pragma reserve` keeps a name from being declared at all, which lets a library claim a name now and fill it in later:

examples/ch16/reserved.tiko

```
'' Keep a name free for a later version of the library.
#pragma reserve Checksum

function Checksum( byval s as ustring ) as uint32
    return len( s )
end function
```

```
reserved.tiko(4,1): error TKO0769: 'checksum' is reserved by '#pragma
reserve' and cannot be declared
     |
   4 | function Checksum( byval s as ustring ) as uint32
     | ^
```

`#lang "fb"` names the language a file is written in. It is the only one tiko accepts, so you never need to write it; any other is `TKO0317`.

What it costs

- **A header is compiled again by every file that includes it.** That is nothing for a few declarations and a great deal for a large C binding included by fifty files. Modules exist to pay that cost once (Chapter 15).
- **Linking is where separately compiled files meet**, so a clash or a missing definition is found last, after every file has compiled, and reported by the linker in the linker's terms.
- **A macro has no type and no scope.** It is checked only after it is expanded, so a mistake in a macro is reported at every use, in terms of the expanded text.
- **A branch of conditional compilation is compiled only on the platform that takes it.** Code for Linux that is never built on Linux is code nobody has compiled. Build every target you claim to support.
- **Namespaces cost nothing when the program runs.** A namespace changes the name the linker sees, not the code.

Summary

- A program can be several files, each compiled on its own and joined by the linker. A header (`.ti`) declares what the files share: `declare` for procedures, `extern` for variables. Exactly one file defines each.
- `#include once` and `#pragma once` make a header safe to reach more than once. A header is looked for first beside the file that includes it, then in the working directory, then in each `-i` directory in turn, then in tiko's `inc/`.
- Procedures are public by default. `private` keeps one in its file, and `#pragma push( private, true )` … `#pragma pop( private )` makes private the default for a region.
- `dim shared` in one file with `extern` in the header shares a variable; `common shared` makes every declaring file share one, unchecked.
- `constructor` and `destructor` subs run before and after the main program, lowest priority first and last respectively. Other files' top-level code runs during startup; `-m` chooses the main module.
- A namespace qualifies names. It nests and can be reopened; `using` makes its members available to the end of the enclosing block, but any closer declaration of the name wins, and two `using`s supplying the same name is `TKO0919`.
- Macros substitute text: parenthesise parameters and bodies, use `#` to quote, `##` to paste, and `#macro` for whole statements. Prefer a function unless the macro generates code.
- `#if` tests a value and `#ifdef` tests a definition. Platform symbols include `__FB_WIN32__`, `__FB_LINUX__`, `__FB_64BIT__` and `__FB_MSVC__`; `-d` defines your own.
- `defined( )`, `#if typeof`, `#assert` and `#error` check the program while it compiles. `#error` reports and the compiler reads on.

- `__FILE__`, `__LINE__`, `__FUNCTION__` and `__DATE_ISO__` describe the source; `#line` renumbers it. `#pragma reserve` claims a name.

Exercises

1. Write `CUBE( x )` so that `CUBE( 1 + 1 )` is 8 and `100 \ CUBE( 2 )` is 12. Then call it with a function that prints each time it runs. How many times does it run, and why? {solution="ex16-01″}
2. Write a `#macro DEFINE_COUNTER( name )` that defines a function `Next<name>` returning 1, 2, 3 … on successive calls, so that `DEFINE_COUNTER( Order )` gives `NextOrder( )`. Define two counters and show they count separately. {solution="ex16-02″}
3. Write a program that prints the name of the environment variable that holds the temporary directory: `TEMP` on Windows, `TMPDIR` on Linux. Make it refuse to build anywhere else. {solution="ex16-03″}
4. Put a function `Area` in a namespace `Disc` (one radius) and another in a namespace `Rect` (width and height). Call the first without its prefix from a sub, and the second qualified. {solution="ex16-04″}
5. Take `clash.tiko` and `clashother.tiko` and make them link with `#pragma push( private, true )` instead of the word `private`. Which file did you change, and why that one?

Generics

A `Stack` of integers and a `Stack` of names are the same code twice. Before generics you had three ways to avoid writing it twice, and all of them cost something: a macro, which the compiler checks only after it has been pasted somewhere; a container of `any ptr`, which forgets what it holds and makes you remember; or copy and paste, which works until the two copies drift.

A generic is the fourth way. You write the code once with a placeholder for the type, and the compiler produces a real, separately checked version for each type you ask for. The code inside a generic is ordinary tiko — the same parser, the same type checking, the same errors — and what comes out is as fast and as strongly typed as the version you would have written by hand.

A type with a parameter

`of T` after the type's name declares a type parameter, and `T` is then usable anywhere a type can appear:

examples/ch17/box.tiko

```
'' box.tiko -- a generic type, and one type per argument list.
type Box( of T )
    as T value
end type

dim b as Box( of int32 )
dim s as Box( of ustring )
dim n as Box( of Box( of int32 ) )    '' nesting is fine

b.value = 42
s.value = "hello"
n.value.value = 7

print b.value
print s.value
print n.value.value
```

examples/ch17/box.out

```
 42
hello
 7
```

`Box( of int32 )` and `Box( of ustring )` are **two distinct types**, as unrelated as any two types you wrote by hand. One will not assign to the other and one will not

pass where the other is expected. Nesting is ordinary too: `Box( of Box( of int32 ) )` is a box holding a box.

Each distinct argument list produces one instantiation, and writing `Box( of int32 )` in two different files produces the same type in both — they link.

A procedure with a parameter

Procedures take type parameters the same way, and you usually do not have to state them:

examples/ch17/minmax.tiko

```
'' minmax.tiko -- a generic procedure, inferred and given explicitly.
function Larger( of T )( byval a as T, byval b as T ) as T
    if a > b then return a
    return b
end function

print Larger( 3, 9 )                    '' T inferred from the arguments
print Larger( of float64 )( 1.5, 2.5 )  '' or stated
print Larger( "apple", "pear" )
```

examples/ch17/minmax.out

```
 9
 2.5
pear
```

The first call says nothing about `T`; the compiler infers `int32` from the arguments. The second states it, `Larger( of float64 )( … )`, which you need when the arguments would infer something else — here, two literals that would otherwise have been integers.

Inference reads the arguments, and nothing else. It cannot work backwards from what you do with the result, so a generic whose type parameter appears only in the return type must always be given its argument explicitly.

Members, and bodies written out of line

A generic type can declare the full range of members — methods, properties, constructors, operators — and their bodies can live away from the declaration, which is how you keep a long type readable. The body repeats the `( of T )`:

examples/ch17/stack.tiko

```
'' stack.tiko -- a generic type with members, defined out of line.
type Stack( of T )
```

```
    public:
        declare sub Push( byval item as T )
        declare function Pop( ) as T
        declare function IsEmpty( ) as boolean
        declare property Count( ) as int32
    private:
        as T items( any )
        as int32 count_
end type

sub Stack( of T ).Push( byval item as T )
    if this.count_ > ubound( this.items ) then
        redim preserve this.items( this.count_ * 2 + 3 )
    end if
    this.items( this.count_ ) = item
    this.count_ += 1
end sub

function Stack( of T ).Pop( ) as T
    this.count_ -= 1
    return this.items( this.count_ )
end function

function Stack( of T ).IsEmpty( ) as boolean
    return this.count_ = 0
end function

property Stack( of T ).Count( ) as int32
    return this.count_
end property

dim words as Stack( of ustring )
words.Push( "first" )
words.Push( "second" )
print "count: " & words.Count
print words.Pop( )
print words.Pop( )
print "empty: " & words.IsEmpty( )
```

examples/ch17/stack.out

```
count: 2
second
first
empty: true
```

Two details in that listing are worth naming. `redim preserve this.items( … )` grows the storage, so `T` must be a type an array can hold — which every type is. And `declaration order does not matter`: the member bodies come after the type, and a generic may be used before its bodies are written, because the body is not compiled until something asks for an instantiation.

More than one parameter is the obvious extension:

examples/ch17/pair.tiko

```
'' pair.tiko -- two type parameters, and an out-of-line member.
type Pair( of K, V )
    declare function Describe( ) as ustring
    as K key
    as V value
end type

function Pair( of K, V ).Describe( ) as ustring
    return "(" & this.key & ", " & this.value & ")"
end function

dim score as Pair( of ustring, int32 )
score.key = "gold"
score.value = 3
print score.Describe( )

dim ratio as Pair( of ustring, float64 )
ratio.key = "pi"
ratio.value = 3.14
print ratio.Describe( )
```

examples/ch17/pair.out

```
(gold, 3)
(pi, 3.14)
```

Inheritance and virtual methods

An instantiation is an ordinary type, so it can extend another type, and it can override virtual methods. Dispatch through a base pointer reaches the instantiation's override:

examples/ch17/virtual.tiko

```
type Base_ extends Object
    declare virtual function Name( ) as ustring
end type
```

```
function Base_.Name( ) as ustring
    return "base"
end function

type Holder( of T ) extends Base_
    declare virtual function Name( ) as ustring
    as T v
end type
function Holder( of T ).Name( ) as ustring
    return "holder"
end function

dim h as Holder( of int32 )
dim p as Base_ ptr = @h
print p->Name( )          '' virtual dispatch into an instantiation
```

examples/ch17/virtual.out

```
holder
```

This is what makes a generic usable inside an object-oriented design rather than beside one: a `Holder( of int32 )` is a `Base_`, and code that knows only about `Base_` can hold one.

A type parameter promises nothing

This is the part of generics that surprises people coming from languages with constraints, and it shapes how you design with them.

`of T` does not say what `T` can do. It is a blank. The compiler therefore cannot check the body of a generic when it reads it — it has no idea whether `T` has a `>` operator — and instead checks the body **once per instantiation**, against the real type. A demand the type cannot meet is an error at that point:

examples/ch17/unconstrained.tiko

```
'' unconstrained.tiko -- a type parameter promises nothing, so the
'' body's demands are checked only once a type goes in. Fails to build.
type Ranked( of T )
    declare function Best( byval a as T, byval b as T ) as T
    as T dummy
end type

function Ranked( of T ).Best( byval a as T, byval b as T ) as T
    if a > b then return a        '' needs '>' on T
    return b
```

```
end function

type Opaque_
    as int32 x
end type

dim r as Ranked( of Opaque_ )     '' Opaque_ has no '>'
```

That program does not build:

```
unconstrained.tiko(9,10): error TKO0720: no operator > compares this type
     |
   9 |     if a > b then return a        '' needs '>' on T
     |          ^
unconstrained.tiko(17,10):  note:  in  instantiation  of  'ranked(  of
opaque_ )'
     |
  17 | dim r as Ranked( of Opaque_ )     '' Opaque_ has no '>'
     |          ^
```

PITFALL

The error is in the generic's body; the cause is usually the instantiation. The first line points at `Ranked`'s `>`, which is correct code. The mistake is line 17, where `Opaque_`, a type with no `>`, was handed to it. Read only the first line and you start editing a generic that was never wrong.

The `note: in instantiation of` line underneath names the instantiation and the line that wrote it. Read to the end of an error before you edit. And comment a generic's demands where a reader will see them, "`T` needs `>`" above the type, so the next person reaches line 17 first.

The upside of the same rule is that a generic costs nothing to declare. `Stack( of T )` works for every type that can be copied, and you never had to enumerate what those are.

Making your own type fit a generic's demands

The standard containers state their demands in their documentation rather than in the language. `Map( of K, V )` needs two things of a key: that `K = K` compiles, and that the key can hash itself. You supply both:

`examples/ch17/hashkey.tiko`

```
#import "std/containers"
using containers

'' The hash contract: extend Hashable and override its HashOf( ).
type Point_ extends Hashable
    as int32 x, y
    declare constructor( byval x as int32 = 0, byval y as int32 = 0 )
    declare function HashOf( ) as ulongint override
end type

constructor Point_( byval x as int32, byval y as int32 )
    this.x = x : this.y = y
end constructor

function Point_.HashOf( ) as ulongint
    return containers.HashOf( x ) * 31 + containers.HashOf( y )
end function

operator = ( byref a as Point_, byref b as Point_ ) as boolean
    return ( a.x = b.x ) andalso ( a.y = b.y )
end operator

dim seen as Map( of Point_, ustring )
dim as Point_ a = Point_( 1, 2 ), b = Point_( 1, 2 )
seen[ a ] = "first"
print "keys: " & seen.Count & "   lookup: " & seen[ b ]
```

examples/ch17/hashkey.out

```
keys: 1   lookup: first
```

The key **extends `Hashable`** and overrides its `HashOf( )`, because the container's code lives in the `std/containers` module: a free `HashOf` overload of your own would be invisible to it, and a module's namespace cannot be re-opened to add one. The built-in types are hashable already, and `containers.HashOf( x )` is how the override hashes its fields.

> **NOTE**
> `operator =` for a key type is not optional. A hash gets you to the right bucket; equality decides which entry in that bucket is yours, and a hash that agrees while equality disagrees produces a map that stores duplicates and finds nothing.

Filling a container from a list

A container starts empty, and filling it used to take one statement per element. A brace list says the same thing at the declaration, the way it does for an array:

examples/ch17/fill.tiko

```
'' fill.tiko -- a brace list fills a container: one call per element,
'' in order, after the container is constructed.
#import "std/containers"

'' Your own generic takes part by having an Add, a PushBack or a Push.
type Bag( of T )
    declare sub Add( byval item as T )
    as T items( any )
end type

sub Bag( of T ).Add( byval item as T )
    redim preserve this.items( ubound( this.items ) + 1 )
    this.items( ubound( this.items ) ) = item
end sub

dim primes as containers.Array( of int32 ) = { 2, 3, 5, 7 }
dim queue as containers.LinkedList( of ustring ) = { "a", "b", "c" }
dim ages as containers.Map( of ustring, int32 ) = _
    { { "Ada", 36 }, { "Grace", 85 } }
dim bag as Bag( of float64 ) = { 0.5, 1.5 }

print "primes:"; primes.Count( ); ", last"; primes[3]
for each s as ustring in queue
    print s;
next
print
print "Grace is"; ages["Grace"]
print "bag holds"; ubound( bag.items ) + 1; ", last"; bag.items( 1 )
```

examples/ch17/fill.out

```
primes: 4, last 7
abc
Grace is 85
bag holds 2, last 1.5
```

Each element becomes one call, in order, after the container is constructed. `primes` gets four calls to `Push`, `queue` three to `PushBack`, and `ages` two to `Add`. An element that is itself a brace list supplies the *arguments* of one call, which is how `{ "Ada", 36 }`

becomes `ages.Add( "Ada", 36 )`. They are ordinary calls, so the ordinary conversions apply: `"Ada"` is a string literal and arrives as the `ustring` key the map wants.

The demand is the same kind this chapter has been describing: **a shape, not a declaration.** A brace list fills a type that needs constructing and has a member called `Add`, `PushBack` or `Push`, looked for in that order. Nothing says "I am a collection". `Bag( of T )` takes part because it has an `Add`, and because its dynamic array is a field that needs constructing.

A type that needs constructing but has none of the three methods is refused:

examples/ch17/fillnone.tiko

```
'' fillnone.tiko -- a type that needs constructing but has no Add,
'' PushBack or Push cannot take a brace list.

type Counter
    declare constructor( )
    as int32 n
end type

constructor Counter( )
    this.n = 100
end constructor

dim c as Counter = { 1, 2 }         '' refused
print c.n
```

```
fillnone.tiko(13,20): error TKO0843: 'c' cannot take an initializer list:
its type has no Add, PushBack or Push method
     |
  13 | dim c as Counter = { 1, 2 }         '' refused
     |                    ^
```

Overloading, and when a second generic is unreachable

Two generic procedures may share a name when they take different numbers of arguments, and the call picks the right one — the usual overloading rules apply to the instantiations.

Two limits are worth knowing before you lean on it:

- **A second generic with a different number of *type* parameters does not resolve.** Declaring both `Make( of T )( a as T )` and `Make( of K, V )( k as K, v as V )` compiles, but the two-argument call fails with `TKO0632: no overload of 'make<8>' matches these arguments`.

- **An unconstrained `of T` parameter matches everything**, so an overload that differs only in parameter type is unreachable. Given `Widen( of T )( a as T )` and `Widen( of T )( a as T ptr )`, a pointer argument still selects the first — `T` binds to the pointer type. Distinguish overloads by arity, or give them different names.

Instantiation has a depth limit

A generic that instantiates itself, directly or through another generic, would otherwise recurse until memory ran out. The compiler stops at 64 levels:

examples/ch17/toodeep.tiko

```
type Wrap( of T )
    as T v
end type
'' each instantiation asks for another, one level deeper
type Deep( of T )
    as Wrap( of Deep( of Wrap( of T ) ) ) w
end type
dim d as Deep( of int32 )
print sizeof( d )
```

```
toodeep.tiko(6,26): error TKO0829: generic instantiation of 'wrap' is too
deep
(the limit is 64; -maxinstdepth raises it)
```

Almost always this means a mistake — a type whose definition mentions itself without a pointer or an indirection to stop the recursion. When the depth is genuinely needed, `-maxinstdepth` raises the ceiling.

What it costs

Each distinct argument list produces its own code, so `Array( of int32 )` and `Array( of ustring )` are two separate compiled copies. That is the deal generics make: you pay in binary size for code that runs with no indirection, no boxing and no run-time type checks. A handful of instantiations is nothing; a generic instantiated over forty types in a large program is worth noticing.

Compared with the alternatives, a generic is checked where an `any ptr` container is not, and it produces real diagnostics where a macro produces diagnostics about the expansion.

Summary

- `type Box( of T )` and `function Larger( of T )( … )` declare type parameters; each distinct argument list is a separate, fully checked type or procedure.

- Type arguments are inferred from a call's arguments, and stated explicitly when they are not — inference never works backwards from the return type.
- Members may be defined out of line, repeating the `( of T )`. Declaration order does not matter.
- An instantiation is an ordinary type: it can extend a base and override virtual methods.
- **A type parameter promises nothing.** The body is checked per instantiation; the error lands in the body, and a `note: in instantiation of` names the instantiation that caused it.
- A container states its demands in its docs: `Map`'s key needs `=`, and a `HashOf( )` override from extending `containers.Hashable`.
- **A brace list fills a container**: one `Add`, `PushBack` or `Push` per element, after construction, and `{ k, v }` supplies one call's arguments. Any type that needs constructing and has one of those methods takes part.
- Instantiation stops at 64 levels deep; `-maxinstdepth` raises it.

Exercises

1. Write `Queue( of T )` — first in, first out — with `Enqueue`, `Dequeue` and a `Count` property, and use it with two different element types. {solution="ex17-01″}
2. Write `Swap_( of T )( byref a as T, byref b as T )`. Then try calling it with an `int32` and a `float64` in the same call, and read the error: what does it tell you about how inference decides `T`? {solution="ex17-02″}
3. Make a `Version_` type with `major` and `minor` fields usable as a `Map` key, so that two separately built values of the same version find the same entry. {solution="ex17-03″}
4. Take the `Ranked( of T )` listing above and make it compile by giving `Opaque_` the operator it needs. Which line did you have to change — the generic, or the type you put in it?

Errors: Optional, Result and Runtime Checks

Every program meets things that go wrong: a file that is not there, a line that does not parse, a divisor that turns out to be zero. The question is never whether failure happens but where the program finds out about it, and whether the code that finds out is the code that can do something about it.

tiko gives you two kinds of answer. **Values that say they might have failed** — `Optional` and `Result` from the standard library — put the possibility of failure into a function's type, so the caller cannot use the answer without first deciding what to do when there is none. **Runtime errors** — the `error` statement, `on error goto`, and the checks `-exx` adds — stop the normal flow and jump to a handler. The first kind is for failures a caller is expected to handle. The second is for the ones it is not.

This chapter covers both, then the checks the compiler can add for you and the assertions you can add yourself.

A value that means "nothing"

The oldest way to report failure is a value that cannot otherwise happen:

examples/ch18/sentinel.tiko

```
'' The coldest reading, or -1 when there are none.
function Coldest( temps() as int32 ) as int32
    if ubound( temps ) < lbound( temps ) then return -1
    var lowest = temps( lbound( temps ) )
    for i as int32 = lbound( temps ) + 1 to ubound( temps )
        if temps( i ) < lowest then lowest = temps( i )
    next
    return lowest
end function

dim none() as int32
dim winter( 0 to 2 ) as int32 = { 3, -1, 2 }

print "no readings:"; Coldest( none() )
print "a cold day: "; Coldest( winter() )
```

examples/ch18/sentinel.out

```
no readings:-1
a cold day: -1
```

PITFALL

A sentinel is only safe while it cannot be a real answer. `Coldest` returns −1 to mean "there were no readings", and −1 is also a perfectly good temperature. Both calls print −1; one means "nothing", the other means "minus one degree", and the caller has no way to tell them apart.

It compiles, it runs, and it is wrong exactly when the data reaches the sentinel — the case a test is least likely to cover. The fix is not a cleverer sentinel (−9999 is a legal reading somewhere); it is a return type that has room for "no answer" beside every possible answer.

Optional

`Optional( of T )` is a `T` that might not be there. It lives in the standard library's `outcome` module:

examples/ch18/optional.tiko

```
#import "std/outcome"

'' Where v is in a, if it is anywhere.
function FindIndex( a() as int32, byval v as int32 ) _
    as outcome.Optional( of int32 )
    for i as int32 = lbound( a ) to ubound( a )
        if a( i ) = v then return outcome.Some( i )
    next
    return outcome.None( of int32 )( )
end function

dim xs( 0 to 3 ) as int32 = { 4, 8, 15, 16 }

var found = FindIndex( xs(), 15 )
if found.HasValue( ) then print "15 is at"; found.Value( )

var missing = FindIndex( xs(), 99 )
print "99 is at "; str( missing.ValueOr( -1 ) ); ", which means nowhere"

dim nothing as outcome.Optional( of ustring )
print "a new Optional has a value: "; nothing.HasValue( )
```

examples/ch18/optional.out

```
15 is at 2
99 is at -1, which means nowhere
a new Optional has a value: false
```

Member or procedure	Does
`outcome.Some( v )`	an `Optional` holding `v`
`outcome.None( of T )( )`	an `Optional` holding nothing
`o.HasValue( )`	whether there is a value
`o.Value( )`	the value — **a runtime error if there is none**
`o.ValueOr( d )`	the value, or `d` if there is none; never raises

`FindIndex` can now say "15 is at index 2" and "99 is not here" without inventing an index that means nothing. The caller has to ask: there is no way to get an `int32` out of the result without choosing between `HasValue`, `Value` and `ValueOr`.

A few details matter in practice:

- **A new `Optional` is empty.** `dim nothing as outcome.Optional( of ustring )` holds no value, and there is no separate "uninitialised" state to get wrong.
- **`Some` infers `T` from its argument, and a literal is an `integer`.** `outcome.Some( 42 )` is an `Optional( of integer )`, which does not convert to an `Optional( of int32 )`. Write `outcome.Some( of int32 )( 42 )` when the type has to be exact. `None` has no argument to infer from, so its type is always written.
- **`Value( )` on an empty `Optional` is a bug at the call site**, not a recoverable condition: it raises runtime error 1, "illegal function call". Ask `HasValue` first, or use `ValueOr`. The section on runtime errors shows the error being caught.

Result

`Optional` says *that* there is no answer. Often the caller also needs to know *why*, and that is `Result( of T, E )`: a `T` on success, or an `E` that explains the failure.

examples/ch18/result.tiko

```
#import "std/outcome"

type DigitResult as outcome.Result( of int32, ustring )

function ParseDigit( byval s as ustring ) as DigitResult
    if ( len( s ) = 1 ) andalso ( s >= "0" ) andalso ( s <= "9" ) then
        return outcome.Ok( of int32, ustring )( asc( s ) - 48 )
    end if
    var why = "'" & s & "' is not a digit"
    return outcome.Fail( of int32, ustring )( why )
end function

dim inputs( 0 to 2 ) as ustring = { "7", "x", "42" }
for each s as ustring in inputs
```

```
    var r = ParseDigit( s )
    if r.IsOk( ) then
        print s; " ->"; r.Value( )
    else
        print s; " -> "; r.Failure( )
    end if
next

dim blank as DigitResult
print "a new Result succeeded: "; blank.IsOk( )
```

examples/ch18/result.out

```
7 -> 7
x -> 'x' is not a digit
42 -> '42' is not a digit
a new Result succeeded: false
```

Member or procedure	Does
`outcome.Ok( of T, E )( v )`	a success holding `v`
`outcome.Fail( of T, E )( f )`	a failure holding `f`
`r.IsOk( )`	whether it succeeded
`r.Value( )`	the value — a runtime error if it failed
`r.Failure( )`	the reason — a runtime error if it succeeded
`r.ValueOr( d )`	the value, or `d` on failure

`Ok` and `Fail` each mention only one of `T` and `E` in their parameters, so the compiler cannot work out the other: **both type arguments are always written**. That is why the result type is usually named once with a `type … as`, here `DigitResult`, and the long spelling appears only where a value is built.

A new `Result` is a failure. `dim blank as DigitResult` reports `IsOk( )` false, and its `Failure( )` is a default-constructed `E`. That is the safe default — code that forgets to set a `Result` cannot accidentally claim success — and it has a consequence for your own error types: `E` needs a constructor that takes no arguments, because every `Result` builds one.

Passing a failure on

A function that calls another fallible function usually wants to stop at the first failure and pass it up, adding what it knows:

examples/ch18/checkeddivide.tiko

```
#import "std/outcome"
```

```
type Quotient as outcome.Result( of float64, ustring )

function Divide( byval a as float64, byval b as float64 ) as Quotient
    if b = 0 then
        return outcome.Fail( of float64, ustring )( "division by zero" )
    end if
    return outcome.Ok( of float64, ustring )( a / b )
end function

'' The average of a total over a count, which can fail the same way.
function Average( byval total as float64, byval n as int32 ) as Quotient
    var q = Divide( total, n )
    if q.IsOk( ) = false then
        var why = "no average: " & q.Failure( )
        return outcome.Fail( of float64, ustring )( why )
    end if
    return q
end function

print Average( 10, 4 ).ValueOr( 0 )
var bad = Average( 10, 0 )
print bad.IsOk( ); " "; bad.Failure( )
```

examples/ch18/checkeddivide.out

```
 2.5
false no average: division by zero
```

`Average` asks `IsOk( )`, and on failure builds its own failure that wraps the reason it was given. On success it returns `Divide`'s result unchanged, because it is already the right type. Each layer either handles a failure or passes it on, and either way it says so in its code.

An error type of your own

A `ustring` is enough to tell a person what went wrong. A program that wants to *react* to the failure — retry, skip the line, ask again — needs something it can branch on. The standard library's `fs.FsError` is the model: a code, the operation and path involved, and a message. A configuration parser built the same way:

examples/ch18/parseconfig.tiko

```
#import "std/outcome"

'' WHY A LINE WAS REFUSED, modelled on fs.FsError: a code to branch
```

```
'' on, the line it happened on, and a message for people.
enum ConfigCode
    CFG_OK
    CFG_NO_EQUALS
    CFG_EMPTY_KEY
    CFG_NOT_A_NUMBER
end enum

type ConfigError
    code as ConfigCode
    lineNo as int32
    declare constructor( )
    declare constructor( byval c as ConfigCode, byval n as int32 )
    declare function Message( ) as ustring
end type

constructor ConfigError( )
end constructor

constructor ConfigError( byval c as ConfigCode, byval n as int32 )
    code = c : lineNo = n
end constructor

function ConfigError.Message( ) as ustring
    dim as ustring what
    select case code
    case CFG_NO_EQUALS    : what = "expected key=value"
    case CFG_EMPTY_KEY    : what = "the key is empty"
    case CFG_NOT_A_NUMBER : what = "the value is not a number"
    end select
    return "line " & str( lineNo ) & ": " & what
end function

type Setting
    key as ustring
    value as int32
end type

type SettingResult as outcome.Result( of Setting, ConfigError )

function Refuse( byval c as ConfigCode, byval n as int32 ) _
    as SettingResult
    var e = ConfigError( c, n )
    return outcome.Fail( of Setting, ConfigError )( e )
```

```
end function

function ParseLine( byval text as ustring, byval n as int32 ) _
    as SettingResult
    var eq = instr( text, "=" )
    if eq = 0 then return Refuse( CFG_NO_EQUALS, n )
    dim s as Setting
    s.key = trim( left( text, eq - 1 ) )
    if len( s.key ) = 0 then return Refuse( CFG_EMPTY_KEY, n )
    var v = trim( mid( text, eq + 1 ) )
    if ( len( v ) = 0 ) orelse ( str( valint( v ) ) <> v ) then
        return Refuse( CFG_NOT_A_NUMBER, n )
    end if
    s.value = valint( v )
    return outcome.Ok( of Setting, ConfigError )( s )
end function

'' Every line, or the first failure: the error travels up unchanged.
function ParseConfig( lines() as ustring, settings() as Setting ) _
    as outcome.Result( of int32, ConfigError )
    for i as int32 = lbound( lines ) to ubound( lines )
        var r = ParseLine( lines( i ), i + 1 )
        if r.IsOk( ) = false then
            return outcome.Fail( of int32, ConfigError )( r.Failure( ) )
        end if
        redim preserve settings( ubound( settings ) + 1 )
        settings( ubound( settings ) ) = r.Value( )
    next
    return outcome.Ok( of int32, ConfigError )( ubound( settings ) + 1 )
end function

sub Load( lines() as ustring )
    dim settings() as Setting
    var r = ParseConfig( lines(), settings() )
    if r.IsOk( ) then
        print "loaded"; r.Value( ); " settings:";
        for each s as Setting in settings
            print " "; s.key; "="; str( s.value );
        next
        print
    else
        print "refused, "; r.Failure( ).Message( )
    end if
end sub
```

```
dim good( 0 to 2 ) as ustring = _
    { "width = 80", "height=24", "tabs = 4" }
dim bad( 0 to 2 ) as ustring = _
    { "width = 80", "height=tall", "tabs = 4" }
Load( good() )
Load( bad() )
```

examples/ch18/parseconfig.out

```
loaded 3 settings: width=80 height=24 tabs=4
refused, line 2: the value is not a number
```

`ConfigError` carries a `ConfigCode` for programs and a `Message( )` for people, and it has the constructor with no arguments that `Result` needs. `ParseLine` fails in three distinct ways, and `ParseConfig` stops at the first failing line and passes its error up unchanged. The caller sees exactly which line failed and why, and nothing below it had to print anything.

The standard library's own errors work the same way. Reading a file that is not there:

examples/ch18/fserror.tiko

```
#import "std/fs"

var r = fs.ReadAllText( "no-such-file.txt" )
if r.IsOk( ) = false then
    var e = r.Failure( )
    if e.Code( ) = fs.FS_NOT_FOUND then print "not found"
    print e.Operation( ); " "; e.Path( ); ": "; e.Message( )
end if
```

examples/ch18/fserror.out

```
not found
ReadAllText no-such-file.txt: no such file or directory
```

`ReadAllText` returns a `Result` whose failure is an `fs.FsError`. `Code( )` is what the program branches on — `fs.FS_NOT_FOUND` here, the same on Windows and Linux — and the text is for the person reading the output.

Statements that return an error code

Some of the runtime's oldest statements have a **function form** that returns an error number instead of raising a runtime error. `open` and `close` are the ones you will meet most:

examples/ch18/iocodes.tiko

```
dim as int32 f = freefile

'' The function form of OPEN returns an error code; it does not raise.
print "missing:"; open( "no-such-file.txt" for input as #f )
print "present:"; open( "notes.txt" for input as #f )
print "close:"; close( #f )
print "again:"; close( #f )
```

examples/ch18/iocodes.out

```
missing: 2
present: 0
close: 0
again: 1
```

0 means success. Any other value is a runtime error number: 2 is “file not found”, and closing a file number that is no longer open is 1, “illegal function call”.

Written as a statement — `open "notes.txt" for input as #f` — the same failure is **silent** unless the program was compiled with `-e`, and then it stops the program with runtime error 2. A missing file that nothing reports is the worst outcome of the three, which is the reason to use the function form whenever a missing file is an expected event rather than a bug.

Runtime errors

A **runtime error** is a numbered failure that stops the normal flow of the program. The runtime raises them itself — a file that cannot be opened, an `Optional` with no value — and your code can raise one with `error n`. If nothing handles it, the program prints the error and ends with that number as its exit code. To handle one, install a handler with `on error goto`:

examples/ch18/runtimeerrors.tiko

```
#cmdline "-exx"

function Ratio( byval a as int32, byval b as int32 ) as int32
    if b = 0 then error 11            '' 11: division by zero
    return a \ b
end function

sub Report( byval a as int32, byval b as int32 )
    on local error goto failed
    dim as int32 r = Ratio( a, b )
    print a; " /"; b; " ="; r
    exit sub
```

```
failed:
    '' Copy the error out first: the next runtime call may clear it.
    dim as int32 code = err, where_ = erl
    print "error"; code; " at line"; where_; " in "; *erfn( )
end sub

Report( 12, 4 )
Report( 1, 0 )
```

examples/ch18/runtimeerrors.out

```
 12 / 4 = 3
error 11 at line 4 in RATIO
```

`on local error goto failed` installs a handler for the rest of `Report`. When `Ratio` raises error 11, control jumps to `failed:`, where four functions describe what happened:

Function	Gives
`err`	the error number
`erl`	the line it happened on
`erfn( )`	the procedure it happened in, as a `zstring ptr`
`ermn( )`	the module (source file) it happened in, as a `zstring ptr`

`erfn` and `ermn` need the error-location information `-exx` compiles in; without it `erfn( )` is empty. And **read `err` before doing anything else.** The next call into the runtime may clear it: `print "error"; err` printed 0 in testing, because printing the text came first. Copying it to a variable on the handler's first line is the reliable form.

An empty `Optional` raises in exactly this way:

examples/ch18/emptyvalue.tiko

```
#import "std/outcome"

sub ShowValue( byval o as outcome.Optional( of int32 ) )
    on local error goto empty
    dim as int32 v = o.Value( )
    print "value:"; v
    exit sub
empty:
    dim as int32 code = err
```

```
    print "no value: runtime error"; code
end sub

ShowValue( outcome.Some( of int32 )( 42 ) )
ShowValue( outcome.None( of int32 )( ) )
```

examples/ch18/emptyvalue.out

```
value: 42
no value: runtime error 1
```

What the jump skips

PITFALL

When an error jumps out of a procedure, that procedure's destructors and `defer`s do not run.

examples/ch18/escape.tiko

```
type Connection
    handle as int32
    declare constructor( )
    declare destructor( )
end type

constructor Connection( )
    print "  connection opened"
end constructor

destructor Connection( )
    print "  connection closed"
end destructor

sub Transfer( )
    dim c as Connection
    defer print "  transfer finished"
    error 5
end sub

on error goto failed
Transfer( )
end

failed:
```

```
dim as int32 code = err
print "handler: error"; code
```

examples/ch18/escape.out

```
  connection opened
handler: error 5
```

`Transfer` opens a `Connection` and schedules a `defer`. The error leaves `Transfer` for the handler at module level, and neither runs: the connection is never closed, and nothing reports it. A handler *inside* the procedure is different — `Report` above returns normally from its own handler, and its locals are cleaned up as usual. The danger is the error that crosses a procedure boundary on its way to a handler further out. Keep handlers next to the resources they protect, or report failures with a `Result`, which unwinds like any other return.

COMING FROM FREEBASIC

tiko has no `resume`. In a handler you can inspect the error, clean up and leave; you cannot jump back into the code that failed. `resume` is refused as `TKO0409: 'RESUME' is not available in the fb dialect`. A loop around a `Result` does what a `resume` would have, and says so where it happens.

Runtime checking

Some mistakes can only be caught while the program runs: an array index out of range, a null pointer followed. The compiler adds checks for them when asked, and three options control how many:

Option	Adds
`-e`	error checking: runtime errors report and end the program, and handlers work
`-ex`	the same, plus the support `resume` would need in FreeBASIC
`-exx`	the same, plus array bounds, null pointers and the error's location

Chapters 8 and 11 showed what `-exx` catches. A program can ask how it was compiled through `__FB_ERR__`, a bit mask:

examples/ch18/errflags.tiko

```
#cmdline "-exx"
```

```
'' Which runtime checks this program was compiled with.
print "__FB_ERR__ ="; __FB_ERR__
print "error checking:  "; iif( __FB_ERR__ and 1, "on", "off" )
print "array bounds:    "; iif( __FB_ERR__ and 8, "on", "off" )
print "null pointers:   "; iif( __FB_ERR__ and 16, "on", "off" )
```

examples/ch18/errflags.out

```
__FB_ERR__ = 1823
error checking:  on
array bounds:    on
null pointers:   on
```

The values are 0 with no option, 513 with `-e`, 515 with `-ex` and 1823 with `-exx`. Bit 1 is error checking, 8 array bounds and 16 null pointers; the rest are documented with the compiler's options.

Each check also has an option of its own, for a build that wants one check and not the rest: `-earray` (8), `-enullptr` (16), `-elocation`, `-eassert`, `-edebug`, `-edebuginfo`, `-eunwind` and `-earraydims`, each setting only its own bit. `-g` sets 32, 64 and 128 — assertions, debug information and `__FB_DEBUG__` — so `__FB_ERR__` is 224 with `-g` alone and 2047 with `-g -exx`.

Checks cost speed. `-exx` adds a comparison to every array access and a test to every pointer dereference. Use it while developing and testing, and decide deliberately whether a release build keeps it.

Assertions

An **assertion** states something that must be true, and stops the program if it is not. tiko has two kinds.

`#assert` is checked by the compiler, against constants:

examples/ch18/staticassert.tiko

```
const SLOTS = 12

#assert SLOTS mod 4 = 0          '' holds
#assert SLOTS <= 8               '' refused

print "built"
```

```
staticassert.tiko(4,1): error TKO0604: #assert condition failed
     |
```

```
   4 | #assert SLOTS <= 8                '' refused
     | ^
```

It costs nothing at run time and catches a configuration mistake before the program exists.

`assert` is checked while the program runs, and **only in a build with `-g`**. Without `-g` it compiles to nothing at all, so it can stay in code that ships:

examples/ch18/assertfail.tiko

```
#cmdline "-g"

function Average( byval total as int32, byval n as int32 ) as int32
    assert( n > 0 )
    return total \ n
end function

dim as int32 a = Average( 10, 0 )
```

examples/ch18/assertfail.out

```
assertfail.tiko(4): assertion failed at AVERAGE: n > 0
```

The failure names the file, the line, the procedure and the condition as written, and the program exits with status 1. `assertwarn` does the same and carries on.

An assertion is for a mistake in the program — a caller passing 0 where the documentation says "never 0". A condition the program should expect, like a missing file, is a `Result`.

What it costs

- **An `Optional( of T )` is a `T` plus a flag**, and the `T` is default-constructed even when there is no value. A `Result( of T, E )` holds both a `T` and an `E`. For small types this is nothing; for a large one, return it through a parameter or a pointer instead.
- **Checking a `Result` is a branch** at every call. That is the point: the cost is paid where the decision is made.
- **`on error goto` costs nothing until an error happens**, and then skips destructors on its way out of a procedure, which is a cost of a different kind.
- **`-exx` slows every array access and pointer dereference.** `assert` costs nothing without `-g`, and a comparison with it.

Summary

- A sentinel return value is ambiguous whenever it can also be a real answer.

- `Optional( of T )` holds a value or nothing: `HasValue`, `Value` (raises when empty), `ValueOr`. `Some( v )` and `None( of T )( )` build one; a new one is empty.
- `Result( of T, E )` holds a value or a reason: `IsOk`, `Value`, `Failure`, `ValueOr`. `Ok` and `Fail` need both type arguments, so name the type with `type … as`. **A new `Result` is a failure**, and `E` needs a default constructor.
- Pass failures up by returning them; give the error a code to branch on, as `fs.FsError` does.
- `open( … )` and `close( … )` used as functions return an error number instead of raising.
- `error n` raises a runtime error; `on [local] error goto` handles it; `err`, `erl`, `erfn`, `ermn` describe it. Copy `err` first.
- **An error that leaves a procedure for an outer handler skips that procedure's destructors and `defers`.** tiko has no `resume`.
- `-e`, `-ex` and `-exx` add runtime checks; `__FB_ERR__` says which were used.
- `#assert` checks constants at compile time (`TKO0604`); `assert` checks at run time under `-g`.

Exercises

1. Write `ToInt( s )` returning an `Optional( of int32 )` that accepts digits with an optional leading minus and refuses anything else. Use it to total the valid entries of a list and count the refused ones. {solution="ex18-01″}
2. Write `Divide` and `Root`, each returning a `Result( of float64, ustring )`, and a `RootOfRatio( a, b )` that stops at the first failure and passes it on. Run it over a success and one failure of each kind. {solution="ex18-02″}
3. Give `ConfigError` a `CFG_DUPLICATE` code, and make `ParseConfig` refuse a key that appears twice. Which function has to know about the other lines, and why?
4. Move the handler in `escape.tiko` into `Transfer`. What changes in the output, and why?
5. Compile `runtimeerrors.tiko` with `-e` instead of `-exx`. Which part of the handler's output changes?

Testing, Debugging and Tooling

A program that compiles is a program the compiler could read. Whether it does what you meant is a separate question, and the sooner you find out the cheaper the answer is. A failed test costs a minute. A wrong number found in a report a month later costs the month.

This chapter is about the tools that shorten that distance. **Tests** check behaviour you have decided on, every time you build. **Tracing** and a **debugger** show you what a program did when a test, or a user, finds it doing something else. And the compiler's own options — its warnings, its diagnostics file and its ways of showing you the code it generated — are tools too.

Unit tests with fbcunit

tiko ships a unit-test library, fbcunit. A test is a block of code that checks something with an assertion; tests are grouped into suites; one call runs them all and prints a summary. No module fronts the library, so it is reached with `#include`:

examples/ch19/firsttests.tiko

```
'' A stack of int32, and the tests that pin its behaviour down.
#include once "fbcunit.bi"

type Stack
    declare sub Push( byval v as int32 )
    declare function Pop( ) as int32
    declare function Count( ) as int32
private:
    dim items( any ) as int32
end type

sub Stack.Push( byval v as int32 )
    redim preserve items( ubound( items ) + 1 )
    items( ubound( items ) ) = v
end sub

function Stack.Pop( ) as int32
    var top = items( ubound( items ) )
    if ubound( items ) = 0 then
        erase items
    else
        redim preserve items( ubound( items ) - 1 )
    end if
    return top
```

```
end function

function Stack.Count( ) as int32
    return ubound( items ) + 1
end function

SUITE( lifo )
    TEST( starts_empty )
        dim s as Stack
        CU_ASSERT_EQUAL( s.Count( ), 0 )
    END_TEST

    TEST( pops_in_reverse )
        dim s as Stack
        s.Push( 1 ) : s.Push( 2 ) : s.Push( 3 )
        CU_ASSERT_EQUAL( s.Pop( ), 3 )
        CU_ASSERT_EQUAL( s.Pop( ), 2 )
        CU_ASSERT_EQUAL( s.Count( ), 1 )
    END_TEST

    TEST( counts_pushes )
        dim s as Stack
        s.Push( 7 )
        CU_ASSERT_EQUAL( s.Count( ), 2 )   '' wrong on purpose
    END_TEST
END_SUITE

end iif( fbcu.run_tests( ), 0, 1 )
```

examples/ch19/firsttests.out

```
             firsttests.tiko(48)  :  error  :  TESTS.LIFO.COUNTS_PUSHES
CU_ASSERT_EQUAL(s.Count( ),2)

SUMMARY

 Asserts    Passed    Failed  Suite                                  Tests
--------   --------   --------   ------------------------------------
--------
       5         4         1  lifo                                     3
--------   --------   --------   ------------------------------------
--------
       5         4         1  Total                                    3
```

`SUITE( lifo )` … `END_SUITE` groups the tests; each `TEST( name )` … `END_TEST` is one test with a fresh `Stack` of its own. `fbcu.run_tests( )` runs every test, prints each failed assertion with its file, line and the assertion as written, then the summary, and returns `true` only if everything passed. The last line turns that into the program's exit code — 1 here, because the third test is wrong on purpose — which is what a build script or a CI job checks.

The assertions you will use most:

Macro	Passes when
`CU_ASSERT( c )`	`c` is true
`CU_ASSERT_EQUAL( a, b )`	`a = b`
`CU_ASSERT_NOT_EQUAL( a, b )`	`a <> b`
`CU_ASSERT_TRUE( c )`, `CU_ASSERT_FALSE( c )`	`c` is true, or false
`CU_ASSERT_DOUBLE_EQUAL( a, e, g )`	`a` is within `g` of `e`
`CU_FAIL( msg )`	never: marks code that must not be reached

A failed assertion does not stop its test: the test carries on, and every failed assertion is listed before the summary. That is why one `TEST` can hold several related checks, as `pops_in_reverse` does.

> **NOTE**
> **A suite is a namespace,** so it cannot share a name with something it sits beside. `SUITE( stack )` next to a `Stack` type is refused, and the message lands on the type's methods — `TKO0669: 'stack.push' adds a new name to a namespace from outside it` — rather than on the suite. Name a suite after the behaviour it checks, not after the type.

Tests in a multi-module project

In a real project the code under test lives in a module, and the tests are a separate program that imports it. Here is the stack as Chapter 15's kind of module, `stacks/stacks.tiko`:

examples/ch19/stacks/stacks.tiko

```
#module "stacks"

'' A last-in, first-out stack of int32.
public type Stack
    declare sub Push( byval v as int32 )
    declare function Pop( ) as int32
```

```
    declare function Peek( ) as int32
    declare function Count( ) as int32
private:
    dim items( any ) as int32
end type

sub Stack.Push( byval v as int32 )
    redim preserve items( ubound( items ) + 1 )
    items( ubound( items ) ) = v
end sub

function Stack.Pop( ) as int32
    var top = items( ubound( items ) )
    if ubound( items ) = 0 then
        erase items
    else
        redim preserve items( ubound( items ) - 1 )
    end if
    return top
end function

function Stack.Peek( ) as int32
    return items( ubound( items ) )
end function

function Stack.Count( ) as int32
    return ubound( items ) + 1
end function
```

and its tests:

examples/ch19/stacktests.tiko

```
'' The stacks module's tests: a program of their own.
#import "./stacks"
#include once "fbcunit.bi"

SUITE( lifo )
    TEST( starts_empty )
        dim s as stacks.Stack
        CU_ASSERT_EQUAL( s.Count( ), 0 )
    END_TEST

    TEST( pops_in_reverse )
        dim s as stacks.Stack
```

```
        s.Push( 1 ) : s.Push( 2 ) : s.Push( 3 )
        CU_ASSERT_EQUAL( s.Pop( ), 3 )
        CU_ASSERT_EQUAL( s.Pop( ), 2 )
        CU_ASSERT_EQUAL( s.Count( ), 1 )
    END_TEST
END_SUITE

SUITE( peeking )
    TEST( leaves_the_top )
        dim s as stacks.Stack
        s.Push( 7 )
        CU_ASSERT_EQUAL( s.Peek( ), 7 )
        CU_ASSERT_EQUAL( s.Count( ), 1 )
    END_TEST
END_SUITE

end iif( fbcu.run_tests( ), 0, 1 )
```

examples/ch19/stacktests.out

```

SUMMARY

 Asserts    Passed    Failed  Suite                                  Tests
--------   --------   --------   ------------------------------------
--------
       4         4         0  lifo                                     2
       2         2         0  peeking                                  1
--------   --------   --------   ------------------------------------
--------
       6         6         0  Total                                    3
```

The test program sees the module exactly as any other client does: `stacks.Stack` and its public members, and nothing `private`. That is a feature. Tests written against the public surface keep passing while you rearrange the inside, and they fail when you change what callers depend on — which is the change you wanted to hear about.

PITFALL

Suites do not run in the same order on every platform. On Windows the same program prints its summary the other way round:

examples/ch19/stacktests.windows-x86_64.out

```

SUMMARY
```

```
 Asserts   Passed   Failed  Suite                                  Tests
--------  --------  --------  ------------------------------------
--------
      2         2         0  peeking                                    1
      4         4         0  lifo                                       2
--------  --------  --------  ------------------------------------
--------
      6         6         0  Total                                      3
```

Each `SUITE` registers itself with a module constructor, and module constructors run in the order each platform's start-up code walks them — declaration order on Linux, the reverse on Windows, exactly as fbc does it. Nothing is wrong while every test stands alone. A test that quietly relies on state an earlier suite left behind — a file it wrote, a global it set — passes on one platform and fails on the other. Give every test its own setup.

Tracing

Sometimes the quickest way to see what a program did is to have it tell you. Three predefined names make that useful: **`__FILE__`** is the source file being compiled, **`__LINE__`** the line, and **`__FUNCTION__`** the procedure — in capitals, and `__FB_MAINPROC__` at module level. Wrapped in a macro, they report where the macro was *used*, not where it was defined:

examples/ch19/trace.ti

```
'' trace.ti -- TRACE( msg ) reports where it was called from, on
'' stderr. Without TRACING defined it compiles to nothing.
#pragma once

#ifdef TRACING
    #define TRACE( msg ) _
        TraceAt( __FILE__, __LINE__, __FUNCTION__, msg )
#else
    #define TRACE( msg )
#endif

sub TraceAt( byval file as ustring, byval lineNo as int32, _
             byval proc as ustring, byval msg as ustring )
    var f = freefile( )
    open err for output as #f
    print #f, file & "(" & lineNo & ") " & proc & ": " & msg
```

```
    close #f
end sub
```

examples/ch19/tracing.tiko

```
#define TRACING
#include once "trace.ti"

function Mean( values( ) as float64 ) as float64
    var n = ubound( values ) - lbound( values ) + 1
    TRACE( "count " & n )
    dim total as float64
    for i as int32 = lbound( values ) to ubound( values )
        total += values( i )
    next
    TRACE( "total " & total )
    return total / n
end function

dim readings( 0 to 2 ) as float64 = { 3.5, 4.0, 5.5 }
var m = Mean( readings( ) )
TRACE( "mean " & m )
```

examples/ch19/tracing.out

```
tracing.tiko(6) MEAN: count 3
tracing.tiko(11) MEAN: total 13
tracing.tiko(17) __FB_MAINPROC__: mean 4.333333333333333
```

Two decisions in `trace.ti` are worth copying:

- **Trace output goes to stderr.** `open err` opens the standard error stream as a file number. The program's real output stays on stdout, so `program > result.txt` still captures exactly the result, and the trace stays on the screen or goes to a file of its own with `2> trace.txt`.
- **Tracing is switched off by the preprocessor, not by an `if`.** Remove the `#define TRACING` line and `TRACE( … )` expands to nothing: the calls are gone from the generated code, and so is the cost of building their messages. The `TraceAt` procedure itself is still compiled; it is never called.

> **NOTE**
> Do not mix trace lines and ordinary output in one stream and expect their order to hold. stdout and stderr are separate streams with separate buffers; when both go to the same place, how the lines interleave depends on the platform and on

> what is reading them. That is why everything `tracing.tiko` prints goes through `TRACE`.

Debugging with -g

Tracing answers questions you thought to ask before you ran the program. A debugger answers the ones you think of while it is stopped. Compile with **`-g`** and tiko writes **DWARF** debug information into the executable: line tables, procedures, and every variable with its type. Any DWARF debugger can then stop the program and show you its state. The one that comes with the LLVM install tiko already uses is **lldb**.

examples/ch19/readings.tiko

```
'' A small program to stop inside: build it with -g.
type Reading
    sensor as ustring
    value as float64
end type

function Total( rs( ) as Reading ) as float64
    dim sum as float64
    for i as int32 = lbound( rs ) to ubound( rs )
        sum += rs( i ).value
    next
    return sum
end function

dim rs( 1 to 3 ) as Reading
rs( 1 ).sensor = "north" : rs( 1 ).value = 3.5
rs( 2 ).sensor = "south" : rs( 2 ).value = 4.0
rs( 3 ).sensor = "east"  : rs( 3 ).value = 5.5
dim label as ustring = "total"
print label; Total( rs( ) )
```

examples/ch19/readings.out

```
total 13
```

Build it with `tiko -g readings.tiko`, then run it under lldb, stopping at line 10:

```
(lldb) breakpoint set -f readings.tiko -l 10
(lldb) run
* thread #1, stop reason = breakpoint 1.1
    frame #0: readings.exe`Total(rs=0x0000005c4975f7e0) at
readings.tiko:10
```

```
   9        for i as int32 = lbound( rs ) to ubound( rs )
-> 10           sum += rs( i ).value
   11       next
(lldb) frame variable sum i
(double) sum = 0
(long) i = 1
(lldb) up
frame #1: readings.exe`main at readings.tiko:20
-> 20   print label; Total( rs( ) )
(lldb) frame variable label
(ustring) label = (data = u"total", len = 5, size = 16)
(lldb) frame variable rs
(Reading(1 to 3)) rs = {
  [0] = {
    sensor = (data = u"north", len = 5, size = 18)
    value = 3.5
  }
  [1] = {
    sensor = (data = u"south", len = 5, size = 18)
    value = 4
  }
  [2] = {
    sensor = (data = u"east", len = 4, size = 18)
    value = 5.5
  }
}
```

The breakpoint is set by file and line, the stop shows the source around it, and `up` moves to the caller. Variables keep their types: a record shows its fields, a `ustring` its text and length, and the array its declared bounds, `Reading(1 to 3)`. lldb spells some types the way C would — an `int32` is a `long` and a `float64` a `double` — because that is the vocabulary it has.

Two kinds of value read less well. An **array parameter**, such as `rs` inside `Total`, is passed as a descriptor, and lldb shows the descriptor's fields — the data pointer, the element size, and eight dimension slots of which only the first `dimensions` mean anything. And a `ustring` shows its bookkeeping beside its text.

PITFALL

lldb's `[ ]` counts from zero, whatever bounds the array was declared with.

```
(lldb) p rs[1]
(Reading) {
  sensor = (data = u"south", len = 5, size = 18)
  value = 4
}
```

`rs` is declared `1 to 3`, so `rs( 1 )` in the program is `"north"` — but lldb's `rs[1]` is the *second* element, `"south"`. lldb reads the expression as C, and C arrays start at zero. The listing labels the elements `[0]` to `[2]` for the same reason. Subtract the lower bound before you index, or read the element out of the full listing.

ON WINDOWS

On windows-x86_64 the DWARF is placed inside the `.exe`; tiko asks the linker for that itself under `-g` (`-debug:dwarf`). No `.pdb` is written and no CodeView is emitted, so Visual Studio and WinDbg cannot read a tiko program's variables. lldb, and `lldb-dap` for editors, are in the same LLVM `bin` directory as clang.

ON LINUX

gdb, which Ubuntu installs by default, reads the same DWARF. `break readings.tiko:10`, `run` and `info locals` stop in `Total` and show `sum` and `i`; `print rs` in the caller shows the three records with their strings. Unlike lldb, gdb honours the declared lower bound: its `print rs[1]` is the element the program calls `rs( 1 )`, `"north"`.

The same session needs no commands in tkEdit (Chapter 2). Put the cursor on line 10, press `F9` for a breakpoint and `F6` to build with `-g` and run. The program stops in `Total`, and the Locals tab of the output pane lists `rs`, `sum` and `i`. There `rs` is `(1 to 3)` and opens to `rs(1)`, `rs(2)` and `rs(3)`, each element named by the subscript the program uses: tkEdit reads the array descriptor, so the zero-based pitfall above does not apply. A string shows its text, `u"north"`, without the bookkeeping, and a watch takes the program's spelling, `rs(2).sensor`.

Warnings and the diagnostics workflow

An error stops the build. A **warning** does not: the program is built, and it runs. That is exactly what makes warnings worth your attention.

PITFALL

A warning is often a wrong answer the compiler noticed first.

examples/ch19/ptrwarn.tiko

```
'' It compiles. It prints the wrong numbers. The compiler said so.
sub Fill( byval p as int32 ptr, byval n as int32 )
    for i as int32 = 0 to n - 1
        p[i] = i * i
    next
end sub

dim squares( 0 to 3 ) as int64
Fill( @squares( 0 ), 4 )
print squares( 0 ); squares( 1 ); squares( 2 ); squares( 3 )
```

```
ptrwarn.tiko(9,7): warning TKO0833: passing different pointer types:
a 'int64 ptr' for a 'int32 ptr' parameter
     |
   9 | Fill( @squares( 0 ), 4 )
     |       ^
```

examples/ch19/ptrwarn.out

```
 4294967296 38654705668 0 0
```

`Fill` writes four `int32`s, sixteen bytes. `squares` is four `int64`s. The first two squares land in `squares( 0 )` and `squares( 1 )` as halves of 64-bit numbers, and the last two elements are never touched. The build succeeded and the exit code is 0. Only the warning said anything.

Three options decide what warnings do to a build:

Option	Effect
`-Werror`	every warning becomes an error, and the build fails
`-Wno-TKOnnnn`	that one diagnostic is not reported
`--diagnostics-format gnu`	the location as `file:line:col:`, the form many tools parse

With `-Werror` the same program stops at `error TKO0833` and the exit code is 1. Turn it on in any build where you would rather fix a warning than discover it. Use `-Wno-` for a single, understood warning, by id, and never wider: every diagnostic has a permanent number (Appendix B), so the option says exactly what it silences.

The diagnostics file

The console text is for a person. A tool that needs the diagnostics — an editor, a build script, a CI job — asks for **`--diagnostics-file path`**, and tiko writes every diagnostic of the run there as well, one tab-separated record per line:

```
tiko-diagnostics    1
warning TKO0833 C:\work\ptrwarn.tiko    9   7   9   8   passing different
pointer types: a 'int64 ptr' for a 'int32 ptr' parameter
end 0   1   0
```

The first line names the format and its version. Each record is the kind — after `-Werror` and `-Wno-` have been applied — the id, the absolute path, the start and end of the range (line and column, the end exclusive), and the message. The `end` record carries the error count, the warning count and the exit code; a file with no `end` record means the compiler did not finish. The console output is unchanged. This is what tkEdit reads after every build, to list the errors and underline them in the source.

Looking inside a build

When you need to know what the compiler made of your code, ask it. One listing to look at:

examples/ch19/layout.tiko

```
'' Three fields, six bytes of data, twelve bytes of record.
type Packet
    kind as uint8
    length as uint32
    flags as uint8
end type

function Square( byval x as int32 ) as int32
    return x * x
end function

print sizeof( Packet ); Square( 9 )
```

examples/ch19/layout.out

```
 12 81
```

`--dump-udt` prints each record's size, alignment and field offsets, and stops:

```
packet size 12 align 4
  packet off kind 0
```

```
  packet off length 4
  packet off flags 8
```

Six bytes of fields take twelve bytes of record. `length` is aligned to 4, so three bytes of padding follow `kind`, and the record's size is rounded up to its alignment after `flags`. Chapter 9's advice — order fields from largest to smallest — is the fix, and `--dump-udt` is how you check it worked.

`--emit-ir` writes the LLVM IR tiko hands to clang, and stops. Without `-o` it goes to stdout:

```
define i32 @"SQUARE"(i32 %p0) nounwind uwtable(sync) {
L0:
  %s0 = alloca i32, align 4
  store i32 0, ptr %s0, align 4
  %s1 = alloca i32, align 4
  store i32 %p0, ptr %s1, align 4
  %t4 = load i32, ptr %s1, align 4
  %t5 = load i32, ptr %s1, align 4
  %t6 = sext i32 %t4 to i64
  %t7 = sext i32 %t5 to i64
  %t8 = mul i64 %t6, %t7
  %t9 = trunc i64 %t8 to i32
  ret i32 %t9
}
```

`--emit-asm` writes the machine code as assembly, to `layout.s` unless `-o` names another file, and stops. With `-O2` the IR above becomes three instructions:

```
SQUARE:
    movl    %ecx, %eax
    imull   %ecx, %eax
    retq
```

The IR is written plainly — every parameter goes to memory and comes back, the multiplication is done at 64 bits and narrowed — and the optimiser removes all of it. Read the IR to see what your code *means*; read optimised assembly to see what it *costs*.

Option	Shows
`--dump-udt`	records' sizes, alignment and field offsets
`--dump-vtable`	each type's virtual method table (Chapter 14)
`--dump-mangle`	the symbol name each procedure is emitted under
`--emit-ir`	the LLVM IR, then stops
`--emit-asm`	the assembly, then stops
`--keep-temps`	builds normally and keeps the `.ll` and `.o` files beside the output
`-v`	the clang and linker command lines tiko runs

The full list, with the front end's own `--dump-*` options, is Appendix C.

What it costs

- **Tests are code you maintain.** They pay for themselves when they check behaviour callers depend on, and cost you when they pin down details nobody relies on. Test through the public surface.
- **Tracing costs nothing when it is compiled out**, and a call, a string build and a write per `TRACE` when it is in.
- **`-g` makes the executable larger**: the debug information is stored beside the code. Keep it in every development build. Optimisation works against it — variables live in registers and lines are reordered — so debug an unoptimised build.
- **`-Werror` costs a little friction**, paid when the warning appears rather than when the bug it predicted does.

Summary

- fbcunit: `#include once "fbcunit.bi"`, `SUITE`/`TEST` blocks, `CU_ASSERT*` macros, and `fbcu.run_tests( )`, whose result becomes the exit code. A suite is a namespace; name it after a behaviour.
- Test a module from a separate program that imports it, through its public surface.
- **Suites run in a platform-dependent order**; no test may depend on another.
- `__FILE__`, `__LINE__` and `__FUNCTION__` inside a macro report the place of use. Trace to stderr with `open err`, and switch tracing off with the preprocessor.
- `-g` writes DWARF with variables and types. lldb shows locals, records, strings and fixed arrays; **its `[ ]` counts from zero**. On Windows the DWARF is inside the `.exe`; there is no `.pdb`.

- tkEdit's debugger (`F9`, `F6`) reads the same DWARF and names elements by the program's own subscripts.
- A warning can be a wrong answer: `-Werror` fails the build on one, `-Wno-TKOnnnn` silences one by id, `--diagnostics-file` gives tools the records.
- `--dump-udt`, `--emit-ir`, `--emit-asm`, `--keep-temps` and `-v` show what the compiler made of your code.

Exercises

1. Write `Clamp( v, lo, hi )` and an fbcunit suite for it with three tests: a value inside the range, values on both edges, and values outside on both sides. {solution="ex19-01″}
2. Add `TRACEVAL( x )` to `trace.ti`'s family: it prints the expression as written, `=`, and its value. (`#x` turns a macro argument into a string.) {solution="ex19-02″}
3. Make `stacktests.tiko` fail on one platform only, by letting the `peeking` suite rely on something `lifo` does. Then remove the dependency.
4. Reorder `Packet`'s fields so that `--dump-udt` reports a size of 8. Why not 6?
5. Run `readings.tiko` under lldb and print the element `rs( 3 )` the program means. Which expression did you need?

Tuples, Iterators and For Each

Two small problems turn up in almost every program. A function has two answers to give — a minimum and a maximum, a quotient and a remainder — and a procedure returns one value. And a loop over a collection needs an index it does not care about, with a lower bound and an upper bound to get wrong.

This chapter solves both. A **tuple** is a value made of other values, with no type to declare first: a function can return `(lo, hi)` and the caller can take it apart in one line. **`for each`** walks arrays, strings and collections without an index, and the **iterator protocol** lets your own types be walked the same way — including sequences that are computed as they are walked and never stored.

Returning more than one value

examples/ch20/minmax.tiko

```
'' One function, two answers, no out-parameters.
function MinMax( values( ) as int32 ) as (lo as int32, hi as int32)
    var lo = values( lbound( values ) ), hi = lo
    for each v in values
        if v < lo then lo = v
        if v > hi then hi = v
    next
    return (lo, hi)
end function

dim temps( 0 to 4 ) as int32 = { 12, -3, 25, 7, 0 }

var (lo, hi) = MinMax( temps( ) )          '' two new variables
print "range"; lo; " to"; hi

var r = MinMax( temps( ) )                 '' one tuple
print "by name:"; r.lo; r.hi
print "by position:"; r._1; r._2
```

examples/ch20/minmax.out

```
range-3 to 25
by name:-3 25
by position:-3 25
```

`MinMax` returns `(lo as int32, hi as int32)`, a **tuple type**: a parenthesised list of two or more types, written wherever a type can go. `return (lo, hi)` builds a value of it. The caller has two ways to receive it:

- **Destructure it.** `var (lo, hi) = MinMax( … )` declares two variables and gives each one element, in order.
- **Keep it whole.** `var r = MinMax( … )` holds the tuple, and its elements are reached by the names the function's declaration gave them, `r.lo` and `r.hi`, or by position, `r._1` and `r._2`. The positional names `_1` … `_N` always work, named or not.

A tuple has at least two elements. `(x)` is an expression in parentheses, as it always was, and there is no one-element tuple to write.

Tuple types

A tuple type can be written anywhere a type can: a variable, a parameter, a field, an array element, a function result. Elements nest, `((int32, int32), ustring)`, and a typedef gives a tuple type a name when it is used often: `type Point as (x as int32, y as int32)`.

Underneath, a tuple is an ordinary record with the same size, alignment and field offsets as the equivalent `type`. A `ustring` element is constructed, copied and destroyed exactly as a `ustring` field is.

Identity is structural

Two tuple types are **the same type** when they have the same element types in the same order. What the elements are called, and whether the type was spelled through a typedef, does not enter:

examples/ch20/tupleidentity.tiko

```
'' Two tuple types are the same when their element types match,
'' in order. Names and spellings do not enter.
type Point as (x as int32, y as int32)

dim p as Point = (3, 4)
dim span as (lo as int32, hi as int32) = p   '' the same type
dim plain as (int32, int32) = span

print span.lo; span.hi; plain._1; plain._2
print p = plain; p <> plain                  '' element by element
plain._2 = 5
print p = plain; p <> plain
print sizeof( Point ); sizeof( (int32, int32, int64) )
```

examples/ch20/tupleidentity.out

```
 3 4 3 4
-1 0
 0-1
 8 16
```

`Point`, `(lo as int32, hi as int32)` and `(int32, int32)` are one type, so each assigns to the others without a conversion. Each variable is read with the names of its own declaration: `span` has `lo` and `hi`, `plain` only `_1` and `_2`. `=` and `<>` compare element by element and give −1 or 0, as any comparison does. `sizeof` takes a tuple type directly or through its typedef.

PITFALL

Element names are documentation for the reader, not a contract the compiler checks.

examples/ch20/swapnames.tiko

```
function ScreenSize( ) as (w as int32, h as int32)
    return (1920, 1080)
end function

'' Names are not part of the type: this copies by position.
dim s as (h as int32, w as int32) = ScreenSize( )
print "width"; s.w; " height"; s.h

'' Destructuring is by position too.
var (h, w) = ScreenSize( )
print "width"; w; " height"; h
```

examples/ch20/swapnames.out

```
width 1080 height 1920
width 1080 height 1920
```

`ScreenSize` returns `(w, h)`; `s` is declared `(h, w)`. Both are `(int32, int32)`, so the assignment copies by position — the width into the element `s` calls `h` — and both lines print the dimensions the wrong way round. Destructuring is positional in the same way: the names in `var (h, w)` are new variables, not a request for the elements called `h` and `w`.

A tuple groups values *by position at a call boundary*. When the meaning of each element matters — a width that must never be taken for a height — declare a `type` with named fields: two different `type`s never convert into each other.

Tuple values

A tuple literal is **built at the type it is assigned to**. Each element is converted as it is stored, in the four places where the target type is known: a declaration, an assignment, a `return` and an argument.

examples/ch20/tuplevalues.tiko

```
'' A tuple literal is built at the type it is assigned to.
dim reading as (float64, ustring) = (21, "north")
print reading._1; " "; reading._2

function Origin( ) as (x as float64, y as float64)
    return (0, 0)                      '' each 0 becomes a float64
end function
var o = Origin( )
print o.x; o.y

'' Assigning to a list of places: the right side is read first.
dim a as int32 = 1, b as int32 = 2
(a, b) = (b, a)
print a; b

dim slots( 0 to 2 ) as int32
type Cell
    row as int32
    col as int32
end type
dim c as Cell
(c.row, c.col, slots( 2 )) = (4, 7, 9)
print c.row; c.col; slots( 2 )

'' Tuples nest.
dim span as ((int32, int32), ustring) = ((9, 17), "office hours")
print span._2; ":"; span._1._1; " to"; span._1._2
```

examples/ch20/tuplevalues.out

```
 21 north
 0 0
 2 1
 4 7 9
office hours: 9 to 17
```

`(21, "north")` becomes a `float64` and a `ustring`; `return (0, 0)` builds two `float64`s. The assigning form `(a, b) = …` takes any list of places — variables, fields, array elements — and **reads the right-hand side completely before writing anything**, which is why `(a, b) = (b, a)` swaps rather than copying one value into both. The number of places must match the number of elements exactly: `var (x, y, z) = (1, 2)` is `TKO0825: the tuple has 2 elements, and 3 are written`.

What tuples do not do

A literal adapts to its target, but **two tuple values of different types never convert**:

examples/ch20/tuplemismatch.tiko

```
dim whole as (int32, int32) = (1, 2)
dim exact as (float64, float64) = whole    '' refused
```

```
tuplemismatch.tiko(2,35): error TKO0738: no conversion between these two
types
```

The rest of the limits, each one measured:

Written	Result
`a = b` with `a` an `(int32, int32)` and `b` a `(float64, float64)`	`TKO0720: no operator = compares this type`
`a = (1, 2)` with `a` an `(int32, int32)`	`TKO0720` too: the literal's numbers are `integer`s, and a comparison has no target type to build it at
`a < b`, and `<=`, `>`, `>=`	`TKO0720`: tuples have equality, not order
`print a`	`TKO0871: a '(int32, int32)' has no printable form`

The missing ordering is deliberate. "Compare the first elements, then the second" is only sometimes what a program means, and a sort that silently used it the rest of the time would be wrong without a word. Write the comparison you mean. To compare against fixed values, declare them at the tuple's type first: `dim target as (int32, int32) = (1, 2)`.

The shapes of for each

`for each` walks a collection without an index. The loop declares its own variable, which exists only inside the loop body:

examples/ch20/foreachshapes.tiko

```
#import "std/containers"

'' An array, whatever its bounds. The variable is a copy.
dim scores( 1 to 4 ) as int32 = { 70, 85, 90, 62 }
for each s in scores
    print s;
next
print
```

```
'' byref binds to the element itself.
for each byref s in scores
    s += 5
next

'' 'as' gives the element a type; the usual conversions apply.
for each s as float64 in scores
    print s / 10;
next
print

'' exit for and continue for work as in any for loop.
for each s in scores
    if s < 80 then continue for
    if s > 94 then exit for
    print "pass"; s
next

'' Library collections are walked the same way.
dim queue as containers.LinkedList( of ustring ) = _
    { "build", "test", "ship" }
for each task in queue
    print task; " ";
next
print
```

examples/ch20/foreachshapes.out

```
 70 85 90 62
 7.5 9 9.5 6.7
pass 90
build test ship
```

Form	The loop variable is
`for each x in c`	a copy of each element, its type inferred
`for each x as E in c`	a copy converted to `E`, as `dim x as E = …` would
`for each byref x in c`	the element itself; writing to `x` writes to the collection
`for each byref x as E in c`	the same, with the type stated

The loop ends with a plain `next`. Naming the variable after it is `TKO0819: 'next' cannot name a FOR EACH variable`, because the variable is not a counter and does not exist after the loop. `exit for` and `continue for` work as they do in any `for` loop.

An array is walked in index order from its lower bound, whatever that bound is. A multi-dimensional array is refused with `TKO0608`: there is no order that would be right for every program, so you choose one with ordinary `for` loops.

The collection expression is evaluated **once**, before the first pass. Over an array, `for each` compiles to the same counted loop you would have written by hand.

Strings

A `ustring` cannot be walked:

examples/ch20/ustringwalk.tiko

```
dim word as ustring = "tiko"
for each ch in word
    print ch
next
```

```
ustringwalk.tiko(2,1): error TKO0688: 'for each' cannot walk a ustring;
index it with 'for' and 'mid' instead
```

A `string` can — and that is the trap.

PITFALL

`for each` over a `string` yields bytes, not characters.

examples/ch20/stringbytes.tiko

```
'' A string is walked one byte at a time.
dim word as string = "caf" & chr( &hC3, &hA9 )  '' e-acute in UTF-8
var n = 0
for each b in word
    print b;
    n += 1
next
print
print n; " steps for"; len( word ); " bytes"
```

examples/ch20/stringbytes.out

```
9997102195169
 5 steps for 5 bytes
```

The word has four letters and the loop runs five times: `é` is two bytes in UTF-8, 195 and 169. Each `b` is a `ubyte`, so the loop is right for parsing bytes and wrong

> for anything a reader would call a letter, and no diagnostic says so. For text, keep the `ustring` and index it with `for` and `mid`, as the `TKO0688` message says.

Binding a string's bytes by reference is refused: `for each byref b in word` is `TKO0941: 'for each byref' cannot walk a string: a character has no stable address`.

Taking pairs apart

When a collection's elements are tuples, `for each` can destructure each one into names:

examples/ch20/pairs.tiko

```
#import "std/containers"

dim ages as containers.OrderedMap( of ustring, int32 )
ages.Put( "grace", 85 )
ages.Put( "ada", 36 )
ages.Put( "edsger", 72 )

'' Pairs( ) yields tuples, which for each takes apart.
for each (who, age) in ages.Pairs( )
    print who; " is"; age
next

'' The map itself yields named key/value pairs.
for each kv in ages
    print kv.key; "="; kv.value; " ";
next
print
```

examples/ch20/pairs.out

```
ada is 36
edsger is 72
grace is 85
ada= 36 edsger= 72 grace= 85
```

`Pairs( )` yields `(key, value)` tuples, and `for each (who, age)` declares one variable per element. Iterating the map itself yields a pair record with named fields, `key` and `value`. An `OrderedMap` walks its keys in sorted order; a plain `containers.Map` walks them in an order you must not depend on.

The name list has its own rules:

Written	Diagnostic
`for each (k, v) as int32 in …`	`TKO0410`: each name is typed by its own element, so there is no `as`
`for each byref (k, v) in …`	`TKO0411`: bind the whole element `byref` instead and reach through `_1`, `_2`
`for each (k) in …`	`TKO0412`: a name list needs at least two names

The names are copies. Destructuring costs one copy of each element.

The iterator protocol

`for each` walks your own types too, and there is nothing to inherit to make that happen. A type is **iterable** when it has a `GetIterator( )` function, and what that returns is an **iterator** when it has three members:

Member	Job
`function GetIterator( ) as I`	on the collection: hand out a new iterator `I`
`function IsValid( ) as boolean`	on the iterator: is there a current element?
`function Value( ) as E`	the current element; `E` is the loop variable's type
`sub MoveNext( )`	advance to the next element

examples/ch20/countdown.tiko

```
'' The smallest iterable: a collection and its iterator.
type CountdownIterator
    n as int32
    declare function IsValid( ) as boolean
    declare function Value( ) as int32
    declare sub MoveNext( )
end type

function CountdownIterator.IsValid( ) as boolean
    return n > 0                    '' still on an element?
end function

function CountdownIterator.Value( ) as int32
    return n                        '' the current element
end function

sub CountdownIterator.MoveNext( )
```

```
    n -= 1                          '' advance
end sub

type Countdown
    from as int32
    declare function GetIterator( ) as CountdownIterator
end type

function Countdown.GetIterator( ) as CountdownIterator
    print "(GetIterator)"
    return type( from )             '' already on the first element
end function

dim launch as Countdown = type( 5 )
for each t in launch
    print t;
next
print " lift-off"
```

examples/ch20/countdown.out

```
(GetIterator)
 5 4 3 2 1 lift-off
```

The loop calls `GetIterator( )` once, then, for as long as `IsValid( )` is true, binds `Value( )` to the loop variable, runs the body and calls `MoveNext( )`. Three rules make an iterator behave:

- **A new iterator is already on the first element**, or `IsValid( )` is false at once for an empty collection. There is no "before the first element" state and no initial `MoveNext( )`.
- **`IsValid( )` is a question, not an action.** Asking twice without a `MoveNext( )` between gives the same answer, and `Value( )` may be read more than once for the same element.
- **`Value( )` decides what `byref` can do.** Declared `byref as E`, it hands out the element itself and a `for each byref` loop can write to it. Declared by value, it hands out a copy.

The protocol is matched by name, at compile time. There is no interface and no virtual call: the loop calls the three members directly, exactly as if you had written the calls yourself. An iterator is an ordinary value returned by value — usually a pointer and a count — and if it holds a resource, its destructor runs when the loop ends, by `exit for` or otherwise.

Near misses

Matching by name has one cost: a misspelling does not produce a "you meant" message from a type system — it produces a type that does not match. tiko says which member is missing:

`examples/ch20/nearmiss.tiko`

```
type Ticker
    n as int32
    declare function GetIterator( ) as Ticker
    declare function IsValid( ) as boolean
    declare function Value( ) as int32
    declare sub MoveNxt( )
end type

function Ticker.GetIterator( ) as Ticker
    return this
end function
function Ticker.IsValid( ) as boolean
    return n > 0
end function
function Ticker.Value( ) as int32
    return n
end function
sub Ticker.MoveNxt( )
    n -= 1
end sub

dim t as Ticker = type( 3 )
for each x in t
    print x
next
```

```
nearmiss.tiko(23,1): error TKO0607: 'ticker' is not an iterator: it has
no 'MoveNext( )'
```

A collection with no `GetIterator( )` at all gets `TKO0606: 'bag' cannot be walked by 'for each': it has no GetIterator( )`. Both point at the loop, and both name the member to add.

`Ticker` is also its own iterator: its `GetIterator( )` returns a copy of itself. That is a reasonable shape for a one-shot sequence. It is a poor one for a collection, because the "collection" and its cursor are then the same object.

Writing an iterable generic collection

With generics, the iterator is written once for every element type. A ring buffer keeps the newest N values and forgets the oldest:

examples/ch20/ring.tiko

```
'' A ring buffer keeps the newest N values; for each walks them
'' oldest first. Value( ) is byref, so byref loops can write.
type RingIterator( of T )
    buf as T ptr
    cap as int32
    at as int32
    left as int32
    declare function IsValid( ) as boolean
    declare function Value( ) byref as T
    declare sub MoveNext( )
end type

function RingIterator( of T ).IsValid( ) as boolean
    return left > 0
end function

function RingIterator( of T ).Value( ) byref as T
    return buf[at]
end function

sub RingIterator( of T ).MoveNext( )
    at = ( at + 1 ) mod cap
    left -= 1
end sub

type Ring( of T )
    declare constructor( byval capacity as int32 )
    declare sub Add( byval v as T )
    declare function GetIterator( ) as RingIterator( of T )
private:
    dim items( any ) as T
    dim head as int32               '' the next slot to write
    dim count as int32
end type

constructor Ring( of T )( byval capacity as int32 )
    redim items( 0 to capacity - 1 )
end constructor
```

```
sub Ring( of T ).Add( byval v as T )
    var cap = ubound( items ) + 1
    items( head ) = v
    head = ( head + 1 ) mod cap
    if count < cap then count += 1
end sub

function Ring( of T ).GetIterator( ) as RingIterator( of T )
    dim it as RingIterator( of T )
    it.cap = ubound( items ) + 1
    it.buf = @items( 0 )
    it.at = ( head - count + it.cap ) mod it.cap
    it.left = count
    return it
end function

dim words( 0 to 4 ) as ustring = _
    { "alpha", "beta", "gamma", "delta", "epsilon" }
var recent = Ring( of ustring )( 3 )
for each w in words
    recent.Add( w )
next
for each w in recent
    print w; " ";
next
print

var nums = Ring( of int32 )( 4 )
nums.Add( 1 ) : nums.Add( 2 )
for each byref n in nums
    n *= 100
next
for each n in nums
    print n;
next
print

var none = Ring( of int32 )( 2 )
for each n in none
    print "never printed"
next
print "an empty ring walks zero times"
```

examples/ch20/ring.out

```
gamma delta epsilon
 100 200
an empty ring walks zero times
```

`Ring( of T )`'s `GetIterator( )` returns a `RingIterator( of T )` — an instance of a *different* generic, which is the usual shape and the one the standard containers use. The iterator holds a pointer to the storage, where to start, and how many elements are left; it knows nothing about how they got there. Because `Value( )` is `byref as T`, the second loop multiplies the stored numbers in place. And the empty ring needs no special case: its iterator starts with nothing left, so the loop body never runs.

Laziness

An iterator does not have to walk anything stored. `Value( )` can compute the element, and `MoveNext( )` can compute the next one:

examples/ch20/fibseq.tiko

```
'' A sequence that is computed as it is walked, never stored.
type FibIterator
    a as int64
    b as int64
    limit as int64
    declare function IsValid( ) as boolean
    declare function Value( ) as int64
    declare sub MoveNext( )
end type

function FibIterator.IsValid( ) as boolean
    return a <= limit
end function

function FibIterator.Value( ) as int64
    return a
end function

sub FibIterator.MoveNext( )
    var nxt = a + b
    a = b
    b = nxt
end sub

type Fibonacci
    upTo as int64
    declare function GetIterator( ) as FibIterator
```

```
end type

function Fibonacci.GetIterator( ) as FibIterator
    return type( 0, 1, upTo )
end function

dim small as Fibonacci = type( 100 )
for each f in small
    print f;
next
print

'' Effectively endless: only the values the loop asks for exist.
dim all as Fibonacci = type( 9223372036854775807 )
var steps = 0
for each f in all
    steps += 1
    if f > 1000000 then
        print "first above a million:"; f; ", term"; steps
        exit for
    end if
next
```

examples/ch20/fibseq.out

```
 0 1 1 2 3 5 8 13 21 34 55 89
first above a million: 1346269, term 32
```

`Fibonacci` holds one number, its upper limit. Its iterator holds the two most recent terms, and each `MoveNext( )` computes the next. Nothing is stored, so a sequence bounded only by the largest `int64` costs the same to set up as one that stops at 100 — the second loop asks for 32 terms, `exit for` ends it, and the rest are never computed. This is a **lazy sequence**: the work is done when an element is asked for, and only for the elements that are.

What invalidates a loop

A loop and the collection it walks share the collection's memory, and changing the collection's *shape* during the loop breaks that sharing.

examples/ch20/grow.tiko

```
'' The loop reads an array's bounds once, before the first pass.
dim queue( any ) as int32
redim queue( 0 to 2 )
queue( 0 ) = 1 : queue( 1 ) = 2 : queue( 2 ) = 3
```

```
for each q in queue
    print q;
    if q = 2 then
        redim preserve queue( 0 to 3 )   '' a fourth element...
        queue( 3 ) = 4
    end if
next
print
print "elements now:"; ubound( queue ) + 1   '' ...never visited
```

examples/ch20/grow.out

```
 1 2 3
elements now: 4
```

Over an array, `for each` reads the bounds **once**, before the first pass, exactly as the counted loop it compiles to would. Growing the array inside the loop does not extend the walk: the fourth element exists afterwards and is never visited. The same rule cuts the other way: shrink the array inside its loop and the loop still runs to the bound it read, past the elements that are left. And resizing moves the elements, so a `byref` loop variable bound before the resize refers to where an element used to be.

The rule for collections is the same and stricter. **Changing a collection's structure — adding or removing elements — while an iterator walks it invalidates the iterator**; the iterator may hold a pointer into storage that has since moved. Changing an element's *value* through a `byref` binding is fine; that is what `byref` is for. To add or remove while walking, collect the changes in a second collection and apply them after the loop.

What it costs

- **A tuple is a record.** Returning one costs what returning the equivalent `type` costs, and a tuple of two `int32`s is eight bytes. Destructuring copies each element once.
- **`for each` over an array is the counted loop**, with the same code at `-O2`.
- **An iterator costs three direct calls per element**, which the optimiser can inline: there is no virtual dispatch, because the protocol is matched at compile time.
- **A by-value `Value( )` copies each element.** For a `ustring` or a large record, that is an allocation or a copy per pass; a `byref Value( )` avoids it.
- **A lazy sequence stores nothing**, and recomputes everything each time it is walked.

Summary

- A tuple type is `(T1, T2, …)`, at least two elements, written wherever a type can go. A literal `(a, b)` is built at its target's type.
- Elements are reached as `_1` … `_N`, or by the names in the declaration the value came from. `var (a, b) = …` declares; `(x, y) = …` assigns, reading the right-hand side first.
- **Identity is structural**: same element types, same order, same type. Names are not checked, so a tuple is the wrong tool where meaning must be enforced.
- Different tuple types never convert. Tuples have `=` and `<>`, no ordering, and no `print`.
- `for each x in c`, with `as E` and `byref`; `next` alone; `exit for` and `continue for` work. Arrays in index order; a multi-dimensional array is refused.
- **A `string` is walked as bytes**; a `ustring` cannot be walked.
- `for each (k, v)` destructures tuple elements, with no `as` and no `byref`.
- The iterator protocol: `GetIterator( )`, then `IsValid( )`, `Value( )`, `MoveNext( )`, matched by name. A new iterator is on the first element; a `byref Value( )` permits `byref` loops. `TKO0606` and `TKO0607` name the missing member.
- An iterator can compute its elements: a lazy sequence.
- Bounds are read once; changing a collection's structure invalidates its iterators.

Exercises

1. Write `DivMod( a, b )` returning the quotient and remainder as one tuple, and use it both destructured and whole. {solution="ex20-01″}
2. Write a `Range` type holding `from`, `upTo` and `stepBy`, walkable by `for each` with a positive or negative step, and empty when the step points away from the limit. {solution="ex20-02″}
3. Give `Ring( of T )` a `Count( )` member and test the iterator on a ring that has wrapped round more than once. Which field of the iterator does the wrapping?
4. Change `fibseq.tiko`'s `Value( )` to `byref as int64` and write to the loop variable in a `byref` loop. What changes, and why is it a poor idea for a computed sequence?
5. Rewrite the `swapnames.tiko` example with a `type Size` holding `w` and `h`. Which line now refuses to compile, and what does it say?

Lambdas and Closures

A sort needs to know which of two elements comes first. A callback needs to know what to do when it is called. Both are a few lines of code that only make sense at one place in the program, and until now each of them had to be a named procedure written somewhere else, with any state it needed passed in by hand or left in a global.

A **lambda** is a procedure written where it is used, as an expression. One that uses only its parameters is an ordinary procedure with an address. One that uses variables from around it — **captures** them — becomes a **closure**: a small object that carries those variables, or references to them, along with the code. This chapter covers both, the rule that every capture states how it is held, and where the two kinds can and cannot go.

Lambdas in expressions

examples/ch21/lambdas.tiko

```
'' A lambda is a procedure written where it is used.
type Transform as function( byval x as int32 ) as int32

sub Apply( values( ) as int32, byval f as Transform )
    for i as int32 = lbound( values ) to ubound( values )
        values( i ) = f( values( i ) )
    next
end sub

'' Stored in a procedure-pointer variable...
dim twice as Transform = function( byval x as int32 ) as int32
    return x * 2
end function
print twice( 21 )

'' ...or in a var, whose type it gives; the body can be one line.
var square = function( byval x as int32 ) as int32 : return x * x : _
    end function
print square( 7 )

'' ...or passed straight to the procedure that wants it.
dim readings( 0 to 2 ) as int32 = { 1, 2, 3 }
Apply( readings( ), function( byval x as int32 ) as int32
    return x + 10
end function )
print readings( 0 ); readings( 1 ); readings( 2 )
```

```
var greet = sub( byval who as ustring )
    print "hello, "; who
end sub
greet( "tiko" )
```

examples/ch21/lambdas.out

```
 42
 49
 11 12 13
hello, tiko
```

A lambda is spelled like a procedure without a name: `function( parameters ) as Type … end function`, or `sub( parameters ) … end sub` when there is nothing to return. It can appear wherever an expression can:

- **assigned to a procedure-pointer variable** whose type it matches, like `twice`;
- **given to `var`**, which takes its type from it, like `square` and `greet`;
- **passed as an argument**, like the one `Apply` receives.

The body is ordinary statements. Because `:` separates statements, a short body fits on one line: `function( byval x as int32 ) as int32 : return x * x : end function`. A lambda is called like any procedure: `square( 7 )`.

A lambda that captures nothing is a procedure pointer

A lambda that uses only its own parameters and locals is compiled to an ordinary procedure, and the expression's value is that procedure's address. It is interchangeable with `@SomeProcedure` and converts to any procedure-pointer type with the same signature — which is exactly what a C library callback wants:

examples/ch21/qsortcallback.tiko

```
'' The C library's qsort takes a comparer by address. A lambda that
'' captures nothing is exactly that: a procedure and its address.
#include once "crt/stdlib.bi"

dim scores( 0 to 5 ) as int32 = { 42, 7, 19, 88, 3, 56 }

qsort( @scores( 0 ), 6, sizeof( int32 ), _
    function cdecl( byval a as const any ptr, _
                    byval b as const any ptr ) as long
        return *cptr( int32 ptr, a ) - *cptr( int32 ptr, b )
    end function )

for each s in scores
```

```
    print s;
next
print
```

examples/ch21/qsortcallback.out

```
 3 7 19 42 56 88
```

`qsort` takes the address of a comparison function that returns a negative number, zero or a positive number. The lambda is that function, written at the one place it is needed. A calling convention goes after the keyword, as on a declared procedure: `function cdecl( … )`. On both of tiko's targets the C convention is the only one in use, so `cdecl` here states what the C library expects rather than changing the code; write it anyway, because it is part of the contract the callback signs.

Capturing variables

A lambda that needs a variable from the code around it lists it in square brackets after the keyword, and says **how** it holds it:

examples/ch21/captures.tiko

```
#import "std/containers"

dim prices as containers.Array( of float64 ) = { 4.5, 12.0, 7.25 }

'' byref: the lambda works on the caller's own variable.
dim total as float64
containers.ForEach( prices, sub[ byref total ]( byref p as float64 )
    total += p
end sub )
print "total"; total

'' byval: the lambda keeps its own copy. Mixed modes are fine.
dim as float64 rate = 0.25
dim as int32 taxed = 0
containers.ForEach( prices, _
    sub[ byval rate, byref taxed ]( byref p as float64 )
        p += p * rate               '' writes the array's element
        taxed += 1
    end sub )
print taxed; " prices taxed:"; prices[0]; prices[1]; prices[2]
```

examples/ch21/captures.out

```

```

```
total 23.75
 3 prices taxed: 5.625 15 9.0625
```

Capture	The closure holds	Changes the lambda makes
`byref total`	the address of the caller's `total`	reach the caller's variable
`byval rate`	its own copy of `rate`, taken when the lambda is evaluated	stay in the copy

The first `ForEach` adds every price into the caller's `total`. The second captures `rate` by value and `taxed` by reference, and mixes the two freely. Its parameter `p` is `byref`, so `p += p * rate` changes the array's elements themselves; that is the parameter's doing, not the capture's.

There is no default mode. A capture written without `byval` or `byref` is refused:

examples/ch21/nomode.tiko

```
dim as int32 n = 1
var show = sub[ n ]( )
    print n
end sub
show( )
```

```
nomode.tiko(2,17): error TKO0413: every lambda capture needs an explicit
'byval' or 'byref'
```

The mode is not a detail to infer. `byref` means the closure holds an address, and an address is only good while the variable exists; `byval` means the closure is self-contained. One word at the capture puts that decision where the next reader will look for it. An empty list, `sub[ ]( … )`, captures nothing and means exactly what `sub( … )` means.

PITFALL

A `byval` capture is a snapshot, taken when the lambda is evaluated — not when it is called.

examples/ch21/snapshot.tiko

```
dim as int32 bias = 10

var addBias = function[ byval bias ]( byval x as int32 ) as int32
```

```
    return x + bias
end function
var addBiasNow = function[ byref bias ]( byval x as int32 ) as int32
    return x + bias
end function

bias = 1000
print addBias( 5 )            '' the copy taken when the lambda was
made
print addBiasNow( 5 )       '' the variable as it is now
```

examples/ch21/snapshot.out

```
 15
 1005
```

`bias` is 1000 by the time either lambda runs. `addBias` copied it when the `var` line ran, so it still adds 10; `addBiasNow` holds the variable's address and sees 1000. Both are correct programs, and only one of them is the program you meant. A lambda written early and called late — a handler, a comparer kept for later — is where this bites. Capture `byref` when the lambda must see the variable as it is at the call, and `byval` when it must not.

A `byval` capture of a container is a copy of the whole container, made when the lambda is evaluated. A lambda that pushes onto it changes its copy, and the caller's container keeps its old contents.

A capture must name a variable that exists. In `sub[ byref totl ]( … )`, where only `total` exists, the capture itself is `TKO0743: 'totl' is not declared`, and a use of `total` in the body is reported too, because the capture did not bring it in.

A closure is not a procedure pointer

A lambda that captures something cannot become a procedure pointer: a pointer is an address and nothing else, and a closure has state that the address has nowhere to keep.

examples/ch21/notaptr.tiko

```
type Scale as function( byval x as int32 ) as int32
dim as int32 factor = 3
dim times3 as Scale = _
    function[ byval factor ]( byval x as int32 ) as int32
    return x * factor
end function
```

```
notaptr.tiko(3,5): error TKO0738: no conversion between these two types
```

A closure reaches a procedure through a **generic parameter** instead. `ForEach` and `SortBy` take their callable as `byref f as F`, with `F` a type parameter, so they accept a closure and a plain lambda alike — each call instantiates the procedure for the exact type it was given. Your own procedures that take callbacks should do the same; `Repeat` below is one.

Closures carry state

examples/ch21/counter.tiko

```
'' A closure carries its own state. Each one is a separate object.
sub Repeat( of F )( byval times as int32, byref action as F )
    for i as int32 = 1 to times
        action( )
    next
end sub

dim as int32 clicks = 0, pings = 0
var click = sub[ byref clicks ]( ) : clicks += 1 : end sub
var ping = sub[ byref pings ]( ) : pings += 1 : end sub

Repeat( 3, click )
Repeat( 2, ping )
click( )
print "clicks"; clicks; ", pings"; pings

'' A closure is a value: copying one copies its captures.
var again = click
again( )
print "clicks"; clicks
```

examples/ch21/counter.out

```
clicks 4, pings 2
clicks 5
```

`click` and `ping` are two closures, each holding the address of its own counter, and each call increments the counter it holds. `Repeat( of F )` calls whatever it is given, the number of times it is told; it neither knows nor cares that `action` carries state.

A closure is an ordinary value. `var again = click` copies it — and copying a closure copies its captures. `again` holds the same address `click` does, so calling `again` increments `clicks`. A `byval` capture would have been copied instead, and the two closures would then count separately.

How long a closure lives

The closure object lives in the scope where the lambda was written, like any local variable, and is destroyed with it. There is no heap allocation and nothing to free.

A `byref` capture holds the address of a variable in some frame, and is only valid while that frame is running. tiko's closures are built so that the usual way of breaking that rule cannot be written: a closure's type has no name, so no function can declare that it returns one. Closures travel *down*, into the procedures they are passed to, which return before the frame that made the closure does. They do not travel *up*, out of the frame whose variables they reference.

Two things remain your responsibility. A pointer to a closure, taken with `@`, carries no such protection: it can be stored, and it is invalid once the closure's scope ends. And the same holds for anything a closure is copied into that you keep beyond that scope.

Comparers and predicates

The standard containers take callables wherever a decision belongs to the caller: `SortBy` and `StableSortBy` take a **comparer**, `ForEach` takes an **action**, and an action that tests and counts is a **predicate** in all but name:

examples/ch21/sortpeople.tiko

```
#import "std/containers"

type Person
    name as ustring
    age as int32
end type

sub Show( byref people as containers.Array( of Person ) )
    for each p in people
        print p.name; p.age; "  ";
    next
    print
end sub

dim people as containers.Array( of Person )
dim p as Person
p.name = "grace"  : p.age = 85 : people.Push( p )
p.name = "ada"    : p.age = 36 : people.Push( p )
p.name = "edsger" : p.age = 72 : people.Push( p )
p.name = "alan"   : p.age = 41 : people.Push( p )

'' The comparer answers one question: does a come before b?
containers.SortBy( people, _
```

```
    function( byref a as Person, byref b as Person ) as boolean
        return a.age < b.age
    end function )
Show( people )

'' A capturing comparer: the direction comes from outside.
dim as boolean oldestFirst = true
containers.SortBy( people, _
    function[ byval oldestFirst ]( byref a as Person, _
                                   byref b as Person ) as boolean
        if oldestFirst then return a.age > b.age
        return a.age < b.age
    end function )
Show( people )

'' A predicate: how many are over forty?
dim as int32 over40 = 0
containers.ForEach( people, sub[ byref over40 ]( byref q as Person )
    if q.age > 40 then over40 += 1
end sub )
print over40; " over forty"
```

examples/ch21/sortpeople.out

```
ada 36  alan 41  edsger 72  grace 85
grace 85  edsger 72  alan 41  ada 36
 3 over forty
```

A comparer answers one question — "does `a` come before `b`?" — with `true` or `false`, and the sort does the rest. It must answer consistently: if `a` comes before `b`, then `b` must not come before `a`, and nothing comes before itself. Returning `true` for equal elements breaks that, and a sort given such a comparer may put elements in any order.

The second comparer captures `oldestFirst`, so the direction is decided by the code around it rather than written into the lambda. That is the everyday use of a capture: a fixed piece of logic with one setting supplied from outside.

Lambdas inside generics

A lambda written inside a generic procedure is instantiated with it, so it can use the generic's type parameter:

examples/ch21/ingeneric.tiko

```
#import "std/containers"
```

```
'' A lambda that captures nothing may use T freely.
sub SortDown( of T )( byref xs as containers.Array( of T ) )
    containers.SortBy( xs, function( byref a as T, byref b as T ) _
            as boolean
        return a > b
    end function )
end sub

'' A capturing lambda may capture a T, but not take one as a parameter:
'' give it a concrete parameter type instead.
function CountAbove( of T )( byref xs as containers.Array( of T ), _
                             byval limit as T ) as int32
    dim as int32 n = 0
    var check = sub[ byref n, byval limit ]( byval v as float64 )
        if v > limit then n += 1
    end sub
    for i as int32 = 0 to xs.Count( ) - 1
        check( xs[i] )
    next
    return n
end function

dim marks as containers.Array( of int32 ) = { 55, 91, 70, 38 }
SortDown( marks )
for each m in marks
    print m;
next
print
print CountAbove( of int32 )( marks, 60 ); " above 60"
```

examples/ch21/ingeneric.out

```
 91 70 55 38
 2 above 60
```

`SortDown`'s comparer captures nothing, so its parameters may be of type `T`. `CountAbove`'s lambda captures a `T` — `limit` — and that is allowed too. What a **capturing** lambda cannot do is take a parameter of type `T`:

examples/ch21/genericlimit.tiko

```
#import "std/containers"

function CountAbove( of T )( byref xs as containers.Array( of T ), _
                             byval limit as T ) as int32
```

```
    dim as int32 n = 0
    containers.ForEach( xs, sub[ byref n, byval limit ]( byref x as T )
        if x > limit then n += 1
    end sub )
    return n
end function

dim marks as containers.Array( of int32 ) = { 55, 91, 70, 38 }
print CountAbove( of int32 )( marks, 60 )
```

```
genericlimit.tiko(3,25): error TKO0921: the capturing lambda's parameter
'x' cannot be of the generic's type parameter 't'
```

A capturing lambda is itself built as a generic, over the types of its captures, and its parameter list cannot also refer to the enclosing generic's `T`. The workaround is the one `CountAbove` uses: give the lambda's parameter a concrete type the elements convert to — `float64` for any number — or call it from an ordinary loop over the collection.

What it costs

- **A lambda that captures nothing costs what a named procedure costs**, and calling it through a pointer costs what any indirect call costs.
- **A closure is a small record** — one address per `byref` capture, one copy per `byval` capture — on the stack of the scope that wrote it. Making one allocates nothing.
- **A `byval` capture copies when the lambda is evaluated.** For a `ustring` or a container, that is an allocation and a copy every time the lambda expression runs; capture `byref` when the copy is not the point.
- **Each generic call site is instantiated for the closure's type.** Passing ten different closures to `Repeat` makes ten copies of `Repeat`, as passing ten different types to any generic does. That is also why the call is direct and can be inlined.

Summary

- A lambda is `function( … ) as T … end function` or `sub( … ) … end sub`, written as an expression; `var` takes its type, and `:` fits a short body on one line.
- A lambda that captures nothing is a procedure and its address: it converts to a matching procedure-pointer type and serves as a C callback. A calling convention follows the keyword.

- `sub[ byref a, byval b ]( … )` captures. **Every capture says `byref` or `byval`** (`TKO0413`). `byref` holds the variable's address; `byval` copies it **when the lambda is evaluated**.
- A capturing lambda is not a procedure pointer (`TKO0738`). It reaches procedures through a generic `of F` parameter, as `ForEach` and `SortBy` take it.
- A closure is a value; copying it copies its captures. It lives in the scope that wrote it, and because its type has no name, no function can return one.
- A comparer answers "does `a` come before `b`?" consistently; capture its settings.
- Inside a generic, a capturing lambda may capture a `T` but not take one as a parameter (`TKO0921`).

Exercises

1. Count the words in a `containers.Array( of ustring )` that are at least `minLen` letters long, with `ForEach` and one closure. Which capture is `byref`, which `byval`, and why? {solution="ex21-01″}
2. Sort names by length, and names of equal length alphabetically, with one comparer. {solution="ex21-02″}
3. Change `snapshot.tiko` so that both lambdas print 1005 without changing their capture modes. What did you have to move?
4. Write `Times( of F )( byval n as int32, byref f as F )` that calls `f( i )` with the counter, and use it with a closure that sums the squares of 1 to 10.
5. Give `sortpeople.tiko` a comparer that returns `true` for equal ages too. Run it a few times with more people of the same age. What does the order of equal elements depend on now?

Resource Management

A program borrows things from the system it runs on: memory, open files, sockets, locks, window handles. Each one has to be given back, exactly once, after the last use — and the code paths between taking and giving back multiply with every `if`, every early `return` and every runtime error.

tiko's answer is **deterministic destruction**: an object is destroyed at a point you can read off the source, when the block that owns it ends. This chapter builds on that rule. It recaps the rule, then shows how to choose between `defer` and a type with a destructor, how to write a handle type that owns one resource, and how to share ownership of an object between several owners with `SharedPtr` and `WeakPtr`.

The rule

Chapter 6 introduced `defer` and Chapter 12 destructors. They follow one rule:

examples/ch22/cleanuporder.tiko

```
'' Destructors and defers share one rule: when a block ends, everything
'' it set up is undone, newest first.
type Resource
    tag as ustring
    declare constructor( byref t as ustring )
    declare destructor( )
end type

constructor Resource( byref t as ustring )
    tag = t
    print "open "; tag
end constructor

destructor Resource( )
    print "close "; tag
end destructor

sub Work( )
    dim a as Resource = Resource( "a" )
    defer print "deferred 1"
    dim b as Resource = Resource( "b" )
    defer print "deferred 2"
    print "working"
end sub

Work( )
```

examples/ch22/cleanuporder.out

```
open a
open b
working
deferred 2
close b
deferred 1
close a
```

When a block ends, everything it set up is undone **in reverse order** — destructors and deferred statements interleaved, newest first. `b` was opened after `deferred 1` was scheduled, so it is closed before that statement runs. The order is the order you wrote things in, reversed; nothing depends on a collector, a timer or the order objects happen to be found in memory.

Every way out of a block follows the rule: falling off its end, `exit sub`, `return`, `exit for`. The one exception is the runtime error that leaves a procedure for a handler further out, which skips that procedure's cleanup — Chapter 18 showed it, and it is the reason to keep handlers close to the resources they protect.

defer, or a type with a destructor

Both run cleanup at the end of a block. They differ in where the knowledge of the cleanup lives.

- **`defer` keeps it at the call site.** `open … as #f` followed by `defer close #f` is right when the resource is used in one procedure and nowhere else: the release is one line, next to the acquisition, and there is nothing to name or declare.
- **A destructor keeps it in a type.** When the same resource is acquired in many places, passed between procedures or stored in other objects, writing `defer` at every acquisition is the same line repeated — and a line someone will forget. A type whose destructor releases the resource makes the release impossible to forget, because there is nothing to write.

The rule of thumb: **`defer` for a one-off, a type for a resource your program has more than one of.**

Writing a handle type

A handle type owns one resource for exactly as long as it lives. A file is the usual example:

examples/ch22/filehandle.tiko

```
'' A handle type: the file is open exactly as long as the object lives.
type TextWriter
```

```
    declare constructor( byref path as ustring )
    declare destructor( )
    declare sub WriteLine( byref text as ustring )
private:
    declare constructor( byref other as TextWriter )   '' no copies
    declare operator let( byref other as TextWriter )
    dim channel as int32
end type

constructor TextWriter( byref path as ustring )
    channel = freefile( )
    open path for output as #channel
end constructor

destructor TextWriter( )
    close #channel
    print "(closed)"
end destructor

sub TextWriter.WriteLine( byref text as ustring )
    print #channel, text
end sub

sub SaveReport( byref path as ustring )
    dim w as TextWriter = TextWriter( path )
    w.WriteLine( "rows: 3" )
    w.WriteLine( "status: ok" )
end sub                      '' w is closed here, on every way out

SaveReport( "report.txt" )

dim as int32 f = freefile( )
open "report.txt" for input as #f
dim as ustring lineText
do until eof( f )
    line input #f, lineText
    print "read: "; lineText
loop
close #f
kill "report.txt"
```

examples/ch22/filehandle.out

```
```

```
(closed)
read: rows: 3
read: status: ok
```

The constructor opens the file and the destructor closes it, so `SaveReport` contains no cleanup code at all and still closes the file on every way out. The program then reads the file back — which it can only do because `SaveReport` has already closed it.

The two `private` declarations matter as much as the destructor. A `TextWriter` that could be copied would give two objects the same channel, and two destructors would close it twice — the double-release from Chapter 12, wearing a file instead of memory. Declaring the copy constructor and `operator let` private, and never defining them, means there is only ever one owner:

examples/ch22/nocopy.tiko

```
type TextWriter
    declare constructor( byref path as ustring )
private:
    declare constructor( byref other as TextWriter )
    dim channel as int32
end type

constructor TextWriter( byref path as ustring )
end constructor

dim a as TextWriter = TextWriter( "a.txt" )
dim b as TextWriter = a                          '' a second owner? refused
```

```
nocopy.tiko(12,5): error TKO0633: 'b' needs a constructor or destructor
that is not accessible here
```

This is the third answer to Chapter 12's rule of three. A type that owns something can **copy it properly**, be **built from parts that copy themselves**, or **refuse to be copied**. For a handle to something the system gave you once — a file, a socket, a lock — refusing is usually right.

Shared ownership: SharedPtr

Sometimes one object really does have several owners — a session that several connections use, a texture several sprites draw — and none of them knows when the others are finished. `SharedPtr( of T )` answers that with a **count of owners**: each copy of the pointer adds one, each copy that goes away subtracts one, and the object is destroyed when the count reaches zero.

> **NOTE**
> **Smart pointers are the `std/memory` module**: `#import "std/memory"`, and its names are written `memory.SharedPtr`, `memory.MakeShared` and so on (or bare, after `using memory`).

examples/ch22/shared.tiko

```
'' One object, several owners; the last owner to go destroys it.

#import "std/memory"
type Session
    user as ustring
    id as int32
    declare destructor( )
end type

destructor Session( )
    print "session"; id; " closed"
end destructor

dim s as memory.SharedPtr( of Session ) = memory.MakeShared( of Session )
( )
s->user = "ada"
s->id = 1815

scope
    dim t as memory.SharedPtr( of Session ) = s     '' a second owner
    print "owners:"; t.UseCount( )
    t->id = 1816
end scope                                          '' t goes; s remains
print s->user; s->id; ", owners:"; s.UseCount( )

s.Reset( )                                       '' the last owner lets go
print "after reset, null: "; s.IsNull( )
```

examples/ch22/shared.out

```
owners: 2
ada 1816, owners: 1
session 1816 closed
after reset, null: true
```

Member	Does
`memory.MakeShared( of T )( )`	a new `T` on the heap, owned by the returned pointer
`p->field`, `*p`	the object, as through a raw pointer — no check for null
`p.Ref( )`	the object, **checked**: a runtime error on an empty pointer
`p.UseCount( )`	how many `SharedPtr`s own the object
`p.Reset( )`	this pointer stops owning; the object goes if it was the last owner
`p.IsNull( )`	whether this pointer owns nothing
`p.Get( )`	the raw `T ptr`, for code that needs one

A pointer can also adopt an object that already exists: `memory.SharedPtr( of T )( new T( … ) )`. After that, the `SharedPtr` owns it — never `delete` it yourself.

Passing one around

examples/ch22/passing.tiko

```
#import "std/memory"
type Config
    name as ustring
end type

sub Show( byref c as memory.SharedPtr( of Config ) )
    print "byref: owners"; c.UseCount( )
end sub

sub Keep( byval c as memory.SharedPtr( of Config ) )
    print "byval: owners"; c.UseCount( )
end sub

dim cfg as memory.SharedPtr( of Config ) = memory.MakeShared( of Config )
( )
Show( cfg )
Keep( cfg )
print "after:   owners"; cfg.UseCount( )

'' Ref( ) is the checked way in: on an empty handle it raises.
on local error goto failed
dim nothing as memory.SharedPtr( of Config )
print nothing.Ref( ).name
```

```
end
failed:
dim as int32 code = err
print "runtime error"; code
```

examples/ch22/passing.out

```
byref: owners 1
byval: owners 2
after:   owners 1
runtime error 1
```

Passing a `SharedPtr` `byval` makes a copy — a real new owner — and costs an increment on the way in and a decrement on the way out. A procedure that only uses the object should take it `byref`. Take it `byval` only when the procedure keeps a copy that outlives the call.

The second half of the listing is `Ref( )` on an empty pointer: runtime error 1, "illegal function call", which an `on error` handler can catch. `->` and `*` do not check, exactly as a raw pointer does not; use `Ref( )` where an empty pointer is possible and you want it caught.

`->` reaches a field, so it needs `T` to be a type with fields. Over a `SharedPtr( of int32 )`, `*count` works and `count->value` does not:

examples/ch22/scalararrow.tiko

```
#import "std/memory"
dim count as memory.SharedPtr( of int32 ) = memory.MakeShared( of int32 )
( )
*count = 3                    '' fine: * works for any T
print count->value            '' -> needs a type with fields
```

```
scalararrow.tiko(5,14): error TKO0739: 'sharedptr' has no member named
'value'
```

Cycles, and WeakPtr

Reference counting has one failure it cannot detect on its own: two objects that own each other.

examples/ch22/treeparents.tiko

```
#import "std/memory"
#import "std/containers"
```

```
'' Children own each other upward AND downward: a cycle.
type LeakyNode
    name as ustring
    kids as containers.Array( of memory.SharedPtr( of LeakyNode ) )
    parent as memory.SharedPtr( of LeakyNode )
    declare destructor( )
end type
destructor LeakyNode( )
    print "freed "; name
end destructor

'' The parent link observes instead of owning.
type Node
    name as ustring
    kids as containers.Array( of memory.SharedPtr( of Node ) )
    parent as memory.WeakPtr( of Node )
    declare destructor( )
end type
destructor Node( )
    print "freed "; name
end destructor

scope
    var root = memory.MakeShared( of LeakyNode )( )
    root->name = "leaky root"
    var kid = memory.MakeShared( of LeakyNode )( )
    kid->name = "leaky kid"
    kid->parent = root
    root->kids.Push( kid )
end scope
print "-- leaky tree out of scope"

scope
    var root = memory.MakeShared( of Node )( )
    root->name = "root"
    var kid = memory.MakeShared( of Node )( )
    kid->name = "kid"
    kid->parent = root
    root->kids.Push( kid )

    var up = kid->parent.Lock( )        '' a real owner, or null
    if up.IsNull( ) = false then print "kid's parent: "; up->name
end scope
print "-- tree out of scope"
```

examples/ch22/treeparents.out

```
-- leaky tree out of scope
kid's parent: root
freed root
freed kid
-- tree out of scope
```

In the first tree, the root owns the kid through `kids` and the kid owns the root through `parent`. When the scope ends, each local `SharedPtr` lets go — and each node still has one owner: the other node. Neither count reaches zero, neither destructor runs, and nothing reports it. The memory is leaked for the rest of the program.

The second tree makes the **back link weak**. A `WeakPtr( of T )` observes an object without owning it: it does not count, so the kid's `parent` no longer keeps the root alive. When the scope ends, the root's count reaches zero, the root is destroyed, its `kids` array goes with it, and the kid follows.

A `WeakPtr` is read through `Lock( )`, which returns a real `SharedPtr` — an owner for as long as you hold it — or an empty one if the object has already gone. There is no way to reach the object through a `WeakPtr` without that check, and that is the point: between "is it still there?" and "use it", the last owner could let go, and `Lock( )` answers both questions at once.

The rule for designing with them: **ownership flows one way.** Pick the direction that owns — parent to child, container to element, window to control — make every link that way a `SharedPtr`, and make every link back a `WeakPtr`.

SpCast and the virtual destructor

A `SharedPtr( of Circle )` does not convert to a `SharedPtr( of Shape )` on its own. The conversion is written out, with both types named:

```
var s = memory.SpCast( of Circle, Shape )( c )
```

Both pointers now own the same `Circle`, and the count is shared.

> **PITFALL**
> **The last owner destroys the object as its own `T`.** If that last owner is a `SharedPtr( of Shape )`, the object is destroyed as a `Shape`:

examples/ch22/spcast.tiko

```
#import "std/memory"
type Shape extends object
```

```
    declare destructor( )               '' not virtual
end type
destructor Shape( )
    print "shape gone"
end destructor

type Circle extends Shape
    r as float64
    declare destructor( )
end type
destructor Circle( )
    print "circle gone"
end destructor

scope
    var c = memory.MakeShared( of Circle )( )
    var s = memory.SpCast( of Circle, Shape )( c )   '' derived to
base
    c.Reset( )
    print "the Shape handle is the last owner"
end scope
```

examples/ch22/spcast.out

```
the Shape handle is the last owner
shape gone
```

`Circle`'s destructor never runs. Here it only prints; in a real type, it is where a file is closed or a buffer freed, and nothing reports that it was skipped. The fix is the one Chapter 14 gave for deleting through a base pointer — **make the base's destructor `virtual`**:

examples/ch22/spcastvirtual.tiko

```

#import "std/memory"
type Shape extends object
    declare virtual destructor( )
end type
destructor Shape( )
    print "shape gone"
end destructor

type Circle extends Shape
```

```
    r as float64
    declare destructor( )
end type
destructor Circle( )
    print "circle gone"
end destructor

scope
    var c = memory.MakeShared( of Circle )( )
    var s = memory.SpCast( of Circle, Shape )( c )   '' derived to
base
    c.Reset( )
    print "the Shape handle is the last owner"
end scope
```

examples/ch22/spcastvirtual.out

```
the Shape handle is the last owner
circle gone
shape gone
```

Any type you intend to share through a pointer to its base needs a virtual destructor. The compiler cannot check that for you: it is a contract of `SpCast`.

Threads

Under `-mt`, the counts inside a `SharedPtr` are updated with atomic operations, so several threads may copy and drop pointers to the same object at the same time. Without `-mt` there are no threads, and the counts are plain arithmetic. That guarantee covers **the count, never the object**: two threads that change the object itself still need a lock of their own. Chapter 23 covers threads and locks.

What it costs

- **A handle type costs nothing beyond its resource.** Its destructor is a call at the end of the block, the same call you would have written by hand.
- **`defer` costs nothing at run time**; the deferred statement is placed on each way out.
- **A `SharedPtr` is two allocations** — the object and a small block holding the counts — and each copy is an increment and, later, a decrement. Under `-mt` those are atomic operations, which are slower again. Pass `byref` where the copy is not needed.

- **A `WeakPtr` keeps the count block alive**, though not the object, until the last weak pointer goes.

Summary

- When a block ends, its destructors and deferred statements run in reverse order of setup, on every way out except an error that leaves for an outer handler.
- `defer` for a one-off resource in one procedure; a type with a destructor for a resource the program has many of or passes around.
- A handle type acquires in its constructor, releases in its destructor, and usually refuses to be copied: a private copy constructor and `operator let` (`TKO0633` at any copy).
- `#import "std/memory"` — `memory.MakeShared`, `->`, `*`, `Ref( )` (checked), `UseCount( )`, `Reset( )`. Pass `byref` unless the callee keeps a copy.
- **Two objects that own each other are never freed.** Make back links `WeakPtr`s and read them with `Lock( )`.
- `memory.SpCast( of Derived, Base )` converts explicitly; **the base needs a virtual destructor**, or the derived part is never destroyed.
- Under `-mt` the counts are atomic; the object itself is not protected.

Exercises

1. Write a `ScratchFile` type that creates a file in its constructor and deletes it in its destructor, and cannot be copied. Show that the file exists inside a `scope` and not after it. {solution="ex22-01″}
2. Keep a `WeakPtr( of Image )` as a cache entry. Show a hit while an owner exists, and an empty `Lock( )` and `Expired( )` after the last owner has gone. {solution="ex22-02″}
3. Rewrite `filehandle.tiko`'s `SaveReport` with `open` and `defer` and no type. Which version would you rather have if ten procedures wrote reports?
4. In `treeparents.tiko`, make the root's `kids` weak instead of the kid's `parent`. What is freed when, and why is that wrong for a tree?
5. Add a third level to the tree and print each node's path to the root by following `parent` links with `Lock( )`.

Concurrency

A program that does one thing at a time wastes the machine it runs on. Downloads wait on the network, a scan waits on the disk, a calculation could use every core — and a window has to keep answering the user while all of that happens. Doing several things at once is how a program stops waiting on itself.

It is also the easiest way to write a program that is right on every run but one. Two threads that change the same variable without coordinating lose updates; a thread that waits for a message nobody sends waits forever; and none of it shows up until the timing is unlucky.

tiko gives you two levels. **OS threads**, with mutexes and condition variables, are the foundation: explicit, flexible and easy to get wrong. **Goroutines and channels**, from the `std/tasks` module, are built on top: you start work with `spawn`, and the pieces talk by sending values rather than by sharing variables. This chapter covers both, then the tools for coordinating them, and the shared state that remains your problem either way.

The thread-safe runtime

Every program that runs more than one thread must be built with **`-mt`**, which links the runtime that is safe to call from several threads at once. Put it in the program, where it cannot be forgotten:

```
#import "std/tasks"
#cmdline "-mt"
```

`#cmdline` comes after any `#import` lines: an `#import` must come before everything else in the file (`TKO0362`). A program can ask how it was built: `__FB_MT__` is −1 under `-mt` and 0 without it.

> **PITFALL**
> **`std/tasks` refuses a build without `-mt`, and the refusal matters.** `#import "std/tasks"` without `-mt` stops at once: `TKO0309: #error: "std/tasks needs the multithreaded runtime: build with -mt"`. Before the library was a module its headers could be included directly, unchecked, and built without `-mt` the goroutine program in the next section printed its correct result and then, run three times, crashed once with a segmentation fault and hung twice at exit. That is the worst kind of failure: the output looks right.
>
> Every listing in this chapter carries `#cmdline "-mt"`. Do the same in yours.

OS threads

Threads and a mutex

`threadcreate( @Proc, arg )` starts a new operating-system thread running `Proc( arg )`, where `Proc` takes one `byval … as any ptr`. `threadwait( t )` waits for it to finish. A **mutex** makes a section of code run in one thread at a time:

examples/ch23/threads.tiko

```
#cmdline "-mt"
'' Four OS threads add to one total. The mutex makes each += whole.
type Counter
    lock as any ptr
    total as int64
end type

sub AddMany( byval p as any ptr )
    var c = cptr( Counter ptr, p )
    for i as int32 = 1 to 100000
        mutexlock( c->lock )
        c->total += 1
        mutexunlock( c->lock )
    next
end sub

dim c as Counter
c.lock = mutexcreate( )

dim workers( 0 to 3 ) as any ptr
for i as int32 = 0 to 3
    workers( i ) = threadcreate( @AddMany, @c )
next
for i as int32 = 0 to 3
    threadwait( workers( i ) )            '' join: wait for it to end
next
mutexdestroy( c.lock )

print "total:"; c.total
```

examples/ch23/threads.out

```
total: 400000
```

`c->total += 1` is three steps — read, add, write back. Two threads can read the same value, both add one and both write, and one increment is lost. Inside `mutexlock` … `mutexunlock` only one thread runs those steps at a time, so the total is exact. The

same program with the two mutex lines removed printed 149321, 188315 and 133439 on three runs: more than half the work lost, and a different amount each time.

The argument is an `any ptr`, so everything a thread needs travels in one record whose address is passed — here the counter and its lock together. A mutex is created with `mutexcreate( )` and must be destroyed with `mutexdestroy( )` when no thread can use it any longer.

Waiting for a condition

A **condition variable** lets a thread sleep until another thread tells it something has changed:

examples/ch23/condvar.tiko

```
#cmdline "-mt"
'' One thread waits until another says the data is ready.
type Mailbox
    lock as any ptr
    ready as any ptr          '' the condition
    full as boolean
    message as ustring
end type

sub Deliver( byval p as any ptr )
    var box = cptr( Mailbox ptr, p )
    mutexlock( box->lock )
    box->message = "parcel"
    box->full = true
    condsignal( box->ready )
    mutexunlock( box->lock )
end sub

dim box as Mailbox
box.lock = mutexcreate( )
box.ready = condcreate( )

var courier = threadcreate( @Deliver, @box )
mutexlock( box.lock )
do until box.full                 '' re-test: a wake-up is only a hint
    condwait( box.ready, box.lock )
loop
print "received: "; box.message
mutexunlock( box.lock )

threadwait( courier )
```

```
conddestroy( box.ready )
mutexdestroy( box.lock )
```

examples/ch23/condvar.out

```
received: parcel
```

`condwait( cond, mutex )` releases the mutex while it sleeps and takes it back before it returns, so the flag it is waiting on can be changed by the other thread in the meantime. Always test the condition **in a loop**: a wake-up says only that something may have changed — the thread can be woken with nothing to do, or after another thread has already consumed the change.

threadcall

`threadcall Proc( args )` is meant to start a thread for an ordinary procedure, with ordinary arguments. In this version it does not: it compiles with a warning, `TKO0797: 'threadcall' is not supported: 'hello' will not run, and the thread handle is 0`, and the procedure never runs. Use `threadcreate` with a record of arguments, or — better — `spawn`.

Goroutines

spawn

A **goroutine** is a procedure call that runs alongside the rest of the program. `spawn` starts one:

examples/ch23/spawnsum.tiko

```
#import "std/tasks"
#cmdline "-mt"

'' A goroutine returns nothing: the channel is its way back.
sub Square( byval n as int32, _
            byval results as tasks.SendChan( of int32 ) )
    results.Send( n * n )
end sub

var results = tasks.Channel( of int32 )( 8 )
for i as int32 = 1 to 5
    spawn Square( i, results )        '' i is copied, here
next

dim as int32 total = 0, v
for i as int32 = 1 to 5
    results.Receive( v )
```

```
    total += v
next
print "sum of squares 1..5:"; total
```

examples/ch23/spawnsum.out

```
sum of squares 1..5: 55
```

`spawn Square( i, results )` returns at once; `Square` runs later, on one of the runtime's worker threads. A goroutine has no return value, so `Square` sends its answer down a **channel**, and the main program receives five answers. They may arrive in any order, which is why the program adds them up rather than printing them.

A goroutine is a work item on a pool of threads, not a thread of its own. Starting one is a queue push, so thousands are fine; the pool grows when goroutines block in system calls, up to 256 workers. And **nothing waits for a goroutine at program exit**: one still running when the program ends is abandoned. A channel, as here, or the `WaitGroup` below is how you wait.

Arguments are copied

The arguments to a spawned procedure are evaluated and **copied at the `spawn` statement**:

examples/ch23/spawncopies.tiko

```
#import "std/tasks"
#import "std/containers"
#cmdline "-mt"

sub Report( byval id as int32, byval label as ustring, _
            byval out_ as tasks.SendChan( of ustring ) )
    out_.Send( label & " #" & id )
end sub

dim results as tasks.Channel( of ustring ) = _
    tasks.Channel( of ustring )( 8 )
dim as ustring label = "job"
for i as int32 = 1 to 4
    spawn Report( i, label, results )   '' its own i and label
next
label = "changed"                         '' too late to matter

dim got as containers.Array( of ustring )
dim as ustring s
for i as int32 = 1 to 4
    results.Receive( s )
```

```
    got.Push( s )
next
containers.Sort( got )                    '' arrival order varies
for each g in got
    print g
next
```

examples/ch23/spawncopies.out

```
job #1
job #2
job #3
job #4
```

Each goroutine gets the `i` and the `label` that existed when it was spawned. Changing `label` afterwards changes nothing they see, however late they run. The results arrive in whatever order the goroutines finish, so the program sorts them before printing.

Copying is what makes this safe, and it rules out a `byref` parameter: the goroutine would hold a reference into the spawning procedure's frame, which may be gone by the time it runs.

examples/ch23/spawnbyref.tiko

```
#import "std/tasks"
#cmdline "-mt"

sub Fill( byref total as int32 )
    total = 42
end sub

dim as int32 answer
spawn Fill( answer )
```

```
spawnbyref.tiko(9,1): error TKO0812: 'spawn fill' cannot pass its byref
parameter 'total'; a string or UDT parameter is byref unless it is written
'byval'
```

Note the second half of the message: a `ustring` or record parameter written without either keyword is `byref`, so a procedure you mean to spawn writes `byval` on every parameter. When the goroutine must change something of yours, pass a pointer and own that decision.

`spawn` takes a procedure declared at module level — a `sub`, or a `function` whose result is discarded. A method, a lambda and a procedure pointer cannot be spawned.

Anything else is refused: with `var greet = sub( ) … end sub`, `spawn greet( )` is `TKO0943: 'spawn' needs a sub or function declared at module level, and 'greet' is not one.`

Channels

A `Channel( of T )` is a typed queue between goroutines. It is created with a **capacity**:

examples/ch23/channels.tiko

```
#import "std/tasks"
#cmdline "-mt"

dim box as tasks.Channel( of int32 ) = tasks.Channel( of int32 )( 2 )

'' Try* never blocks: the third send finds the buffer full.
print box.TrySend( 10 ); box.TrySend( 20 ); box.TrySend( 30 )
print "count"; box.Count( ); ", capacity"; box.Capacity( )

box.Close( )                        '' no more sends; what is queued stays
for each v in box                    '' drains, then ends because closed
    print v;
next
print

dim as int32 x = -1
print "receive after drain: "; box.Receive( x ); ", x still"; x

dim quiet as tasks.Channel( of int32 ) = tasks.Channel( of int32 )( 1 )
print "receive with a 50 ms limit: "; quiet.ReceiveFor( x, 50 )
```

examples/ch23/channels.out

```
truetruefalse
count 2, capacity 2
 10 20
receive after drain: false, x still-1
receive with a 50 ms limit: false
```

Operation	Does
`Send( v )`	waits until there is room, then queues `v`
`Receive( v )`	waits for a value; `false` once the channel is closed **and** empty
`TrySend( v )`, `TryReceive( v )`	never wait; `false` if they would have to
`SendFor( v, ms )`, `ReceiveFor( v, ms )`	wait at most `ms` milliseconds
`Close( )`	no more sends; queued values remain to be received
`for each v in ch`	receives until the channel is closed and drained
`Count( )`, `Capacity( )`, `IsClosed( )`, `IsNil( )`	snapshots, for reporting — not for deciding

A capacity of **0** is a **rendezvous**: `Send` waits until a receiver takes the value, so a send is also a meeting point. A capacity of N buffers N values, and `Send` waits only when the buffer is full — which is what stops a fast producer from running ahead of a slow consumer without limit.

`Receive` returning `false` at the end is not an error; it is how a consumer learns that the producer has finished, and the `for each` form is built on it. `x` keeps its old value when that happens. Sending to a closed channel, or closing one twice, is runtime error 1: it means two parts of the program disagree about who produces.

A channel is a **handle**: copying it gives a second handle to the same queue, which is how one channel is shared between goroutines. A channel declared without a capacity is **nil**, and every send and receive on it waits forever — `IsNil( )` exists to check your own setup.

Pipelines, and channels with one end

Channels chain goroutines into a **pipeline**, each stage reading from one channel and writing to the next:

examples/ch23/pipeline.tiko

```
#import "std/tasks"
#cmdline "-mt"

'' Two stages, each a goroutine, joined by channels; the main program is
'' the third. Each stage closes its output when its input runs dry.
sub Generate( byval upTo as int32, _
              byval out_ as tasks.SendChan( of int32 ) )
    for i as int32 = 1 to upTo
```

```
        out_.Send( i )
    next
    out_.Close( )
end sub

sub Square( byval in_ as tasks.RecvChan( of int32 ), _
            byval out_ as tasks.SendChan( of int32 ) )
    dim as int32 v
    do while in_.Receive( v )
        out_.Send( v * v )
    loop
    out_.Close( )
end sub

var numbers = tasks.Channel( of int32 )( 0 )      '' rendezvous
var squares = tasks.Channel( of int32 )( 0 )

spawn Generate( 6, numbers )
spawn Square( numbers, squares )

for each s in squares
    print s;
next
print
```

examples/ch23/pipeline.out

```
 1 4 9 16 25 36
```

`Generate` closes `numbers` when it has sent everything; `Square` sees `Receive` return `false`, closes `squares` in turn; and the main program's `for each` ends. The close flows down the pipeline, and no stage needs to know how many values there are.

The parameter types say which end each stage holds. A `SendChan( of T )` can send and close; a `RecvChan( of T )` can only receive. A `Channel` converts to either when it is passed. A receiver that tries to close its input is refused, because closing is the producer's decision:

examples/ch23/recvclose.tiko

```
#import "std/tasks"
#cmdline "-mt"

sub Consume( byval inbox as tasks.RecvChan( of int32 ) )
    inbox.Close( )            '' only the sending side may close
end sub
```

```
recvclose.tiko(5,11): error TKO0739: 'recvchan' has no member named
'close'
```

The one-ended types state intent at the boundary where a reader looks for it. They are not a lock: the code that created the `Channel` still holds both ends.

Waiting on several channels: select channel

`select channel` waits until one of several channel operations can proceed, and runs that case. It is part of the language, and its runtime support is in `std/tasks`, which the file imports already (`TKO0423` says so when it does not):

examples/ch23/selecttimeout.tiko

```
#import "std/tasks"
#cmdline "-mt"

sub SlowAnswer( byval delayMs as int32, _
                byval out_ as tasks.SendChan( of int32 ) )
    sleep delayMs, 1
    out_.Send( 42 )
end sub

dim answers as tasks.Channel( of int32 ) = _
    tasks.Channel( of int32 )( 1 )
dim as int32 v

'' case else: do not wait at all.
select channel
    case receive v from answers : print "got"; v
    case else                   : print "nothing ready yet"
end select

'' case timeout: wait, but only so long.
spawn SlowAnswer( 2000, answers )
select channel
    case receive v from answers : print "got"; v
    case timeout 100            : print "gave up after 100 ms"
end select

'' This time the answer arrives while we wait.
select channel
    case receive v from answers : print "got"; v
    case timeout 10000          : print "gave up"
end select
```

examples/ch23/selecttimeout.out

```
nothing ready yet
gave up after 100 ms
got 42
```

Case	Fires when
`case receive v from ch`	a value can be received into `v` (declared beforehand)
`case receive from ch`	a value can be received; it is discarded
`case send v to ch`	`v` can be sent
`case timeout ms`	nothing else became ready within `ms` milliseconds
`case else`	nothing is ready right now — the `select` does not wait at all

A `select` may have one `timeout` or one `else`, not both — writing both is `TKO0942` — because `else` would fire first every time, and the timeout could never be reached. When several cases are ready at once, one is chosen in a varying order, so a channel that is always ready cannot starve the others. A closed channel counts as ready to receive, and a nil channel never does — assigning a nil channel to a variable is how a case is switched off.

Coordinating goroutines

WaitGroup

A `WaitGroup` counts goroutines in flight. `Add( )` raises the count, each goroutine calls `Done( )` as it finishes, and `Wait( )` returns when the count reaches zero:

examples/ch23/workerpool.tiko

```
#import "std/tasks"
#cmdline "-mt"

sub Worker( byval jobs as tasks.RecvChan( of int32 ), _
            byval results as tasks.SendChan( of int32 ), _
            byval wg as tasks.WaitGroup ptr )
    dim as int32 job
    do while jobs.Receive( job )      '' until the jobs channel closes
        results.Send( job * 2 )
    loop
    wg->Done( )
end sub
```

```
sub CloseWhenDone( byval wg as tasks.WaitGroup ptr, _
                   byval results as tasks.Channel( of int32 ) )
    wg->Wait( )
    results.Close( )
end sub

var jobs = tasks.Channel( of int32 )( 16 )
var results = tasks.Channel( of int32 )( 16 )
dim wg as tasks.WaitGroup

for w as int32 = 1 to 3
    wg.Add( )                          '' before the spawn, never inside
    spawn Worker( jobs, results, @wg )
next
spawn CloseWhenDone( @wg, results )

for j as int32 = 1 to 10
    jobs.Send( j )
next
jobs.Close( )

dim as int32 total = 0, count = 0
for each r in results
    total += r
    count += 1
next
print count; " results, total"; total
```

examples/ch23/workerpool.out

```
 10 results, total 110
```

This is a **worker pool**: three workers share one `jobs` channel, so each job goes to whichever worker is free. The main program does not know which worker finishes last, so a small goroutine, `CloseWhenDone`, waits on the group and then closes `results` — and the main program's `for each` ends exactly when the last result is in.

Two rules. **`Add` before `spawn`**, never inside the goroutine: otherwise `Wait` can run before the goroutine has counted itself, and return early. And a `WaitGroup` cannot be copied — a copy would be a second counter — so it is passed by pointer.

Context and cancellation

A **context** asks goroutines to stop. `WithCancel` makes one that `Cancel( )` cancels; `WithTimeout` makes one that cancels itself after a number of milliseconds:

examples/ch23/cancel.tiko

```
#import "std/tasks"
#cmdline "-mt"

'' Works until its context says stop, then reports how it ended.
sub Poll( byval ctx as tasks.Context, _
          byval done_ as tasks.SendChan( of ustring ) )
    do
        select channel
            case receive from ctx.Done( ) : exit do
            case timeout 5                : '' one round of work
        end select
    loop
    done_.Send( "stopped: " & ctx.Err( ) )
end sub

dim finished as tasks.Channel( of ustring ) = _
    tasks.Channel( of ustring )( 1 )
dim as ustring why

'' Cancelled by hand.
dim byHand as tasks.CancelContext = tasks.WithCancel( tasks.Context( ) )
spawn Poll( byHand, finished )
sleep 50, 1
byHand.Cancel( )
finished.Receive( why )
print why

'' Cancelled by its own deadline.
dim byClock as tasks.CancelContext = _
    tasks.WithTimeout( tasks.Context( ), 50 )
spawn Poll( byClock, finished )
finished.Receive( why )
print why
```

examples/ch23/cancel.out

```
stopped: cancelled
stopped: deadline exceeded
```

`ctx.Done( )` is a channel that is **closed** when the context is cancelled — and a closed channel is always ready to receive, so cancellation becomes one more case in a `select`. `Err( )` says why: `"cancelled"` or `"deadline exceeded"`. Cancelling a context cancels every context derived from it, and `Cancel` may be called any number of times.

Cancellation is a request. Nothing is ever stopped by force: a goroutine that never looks at its context runs to the end, and one blocked in a system call cannot see the request until the call returns. That is deliberate — stopping a thread from outside would skip its destructors and its `defer`s, the cleanup the language promises.

`tasks.InterruptContext( )` is a context that **Ctrl-C** cancels (Chapter 25's `os.CatchInterrupt`, started for you). Derive from it and every goroutine that watches its context stops on Ctrl-C with no code of its own:

examples/ch23/interruptctx.tiko

```
#import "std/tasks"
#cmdline "-mt"

'' Works until its context says stop.
sub Work( byval ctx as tasks.Context, byval wg as tasks.WaitGroup ptr )
    while ctx.IsCancelled( ) = false
        sleep 5
    wend
    print "stopped: "; ctx.Err( )
    wg->Done( )
end sub

'' Cancelled by Ctrl-C, or after 100 ms, whichever comes first.
dim           ctx           as           tasks.CancelContext           =
tasks.WithTimeout( tasks.InterruptContext( ), 100 )
dim wg as tasks.WaitGroup
wg.Add( )
spawn Work( ctx, @wg )
wg.Wait( )
```

examples/ch23/interruptctx.out

```
stopped: deadline exceeded
```

Run without a key press it ends with `deadline exceeded`; press Ctrl-C within the 100 ms and it says `cancelled`. Every call returns the same context, so a library can ask for it too.

Shared state

Channels move values between goroutines; they do not protect anything the goroutines share. When two threads or goroutines do touch the same variable, it needs a mutex — or, for a counter, an **atomic** operation:

examples/ch23/atomic.tiko

```
#cmdline "-mt"
#include once "fbatomic.bi"

'' No mutex: each add is one indivisible step.
sub Bump( byval p as any ptr )
    for i as int32 = 1 to 100000
        FB.AtomicFetchAddLong( cptr( long ptr, p ), 1 )
    next
end sub

dim as long hits
dim workers( 0 to 3 ) as any ptr
for i as int32 = 0 to 3
    workers( i ) = threadcreate( @Bump, @hits )
next
for i as int32 = 0 to 3
    threadwait( workers( i ) )
next
print "hits:"; FB.AtomicLoadLong( @hits )
```

examples/ch23/atomic.out

```
hits: 400000
```

`fbatomic.bi` provides three operations on a 32-bit `long`: `FB.AtomicFetchAddLong` adds and returns the old value in one indivisible step, `FB.AtomicCasLong` stores a value only if the current one is what you expected, and `FB.AtomicLoadLong` reads with the ordering a counter written by another thread needs. Each is a full barrier, and each is a call into the runtime.

Some state is not shared at all, and that can be a surprise too.

PITFALL

Some of the runtime's own state belongs to the thread, and a goroutine runs on whichever thread is free.

examples/ch23/dirstate.tiko

```
#import "std/tasks"
#cmdline "-mt"

function Found( byref entry as ustring ) as ustring
    return iif( len( entry ) > 0, "a file", "nothing" )
end function
```

```
sub NextName( byval out_ as tasks.SendChan( of ustring ) )
    out_.Send( "[" & dir( ) & "]" )      '' continue... which listing?
end sub

open "part1.dat" for output as #1 : close #1
open "part2.dat" for output as #1 : close #1

print "main's first match:  "; Found( dir( "part*.dat" ) )

dim names as tasks.Channel( of ustring ) = _
    tasks.Channel( of ustring )( 1 )
spawn NextName( names )
dim as ustring fromGoroutine
names.Receive( fromGoroutine )
print "goroutine's next:    "; fromGoroutine

print "main's next match:   "; Found( dir( ) )

kill "part1.dat" : kill "part2.dat"
```

examples/ch23/dirstate.out

```
main's first match:  a file
goroutine's next:    []
main's next match:   a file
```

`dir( pattern )` starts a directory listing and `dir( )` continues it — **on the same thread**. The goroutine's `dir( )` continues nothing, because the listing was started on the main thread, and returns an empty string. The main thread's listing is untouched. The same holds for `err` and the other runtime state that a statement leaves behind for the next one. Keep a sequence of calls that depend on each other inside one goroutine.

Threads or goroutines?

- **Reach for goroutines and channels first.** They copy arguments, pass results back by value, and make the order of events visible in the code. Most concurrent programs — fan-out work, pipelines, timeouts — are shorter and easier to get right this way.
- **Use an OS thread for something that must own its thread**: a loop that blocks for a long time, code that keeps per-thread state, or anything that belongs to one thread for its whole life.

- **Never touch a window from a goroutine.** A window belongs to the thread that created it, and a goroutine runs on an arbitrary worker. Compute in the goroutine, send the result through a channel, and let the window's own thread take it with `TryReceive`, which never blocks the message loop.

What it costs

- **`-mt` makes every runtime call that touches shared state take a lock**, so a single-threaded program built with it is somewhat slower. Use it where threads are.
- **A goroutine costs a queue push and a copy of its arguments**; a thread costs an operating-system thread. Thousands of goroutines are fine; millions are not.
- **Every channel operation copies the value**, and a buffered send copies twice. For a large record, send a pointer and decide who owns it.
- **A mutex costs little when nobody else holds it** and a great deal when everyone does. A lock around the inner loop of four threads, as in `threads.tiko`, makes them take turns.

Summary

- Build every multi-threaded program with `-mt` — `#cmdline "-mt"` after the imports. Without it the program compiles and then crashes or hangs.
- `threadcreate`/`threadwait` start and join OS threads; `mutexlock`/`mutexunlock` make a section exclusive; `condwait` sleeps on a condition, always in a loop. `threadcall` does not run (`TKO0797`).
- `spawn Proc( args )` runs a module-level procedure as a goroutine; its arguments are copied at the spawn, and a `byref` parameter is refused (`TKO0812`). Nothing waits for goroutines at exit.
- `Channel( of T )( capacity )`: 0 is a rendezvous. `Receive` is `false` at the end of a closed stream; `for each` drains one. `SendChan`/`RecvChan` give one end each.
- `select channel` waits on several operations, with `timeout` or `else`; like `spawn`, it needs `#import "std/tasks"` in the file (`TKO0423`).
- `WaitGroup`: `Add` before `spawn`, `Done` in the goroutine, `Wait` to join. `Context`: cooperative cancellation through a `Done( )` channel, by hand, by deadline, or by Ctrl-C (`InterruptContext`).
- Shared variables need a mutex or an atomic; per-thread runtime state such as `dir( )`'s listing does not follow a goroutine.

Exercises

1. Sum the integers 1 to 1,000,000 in four goroutines, one quarter each, and combine the partial sums through a channel. {solution="ex23-01″}

2. Two producers send three numbers each into **one** channel. Use a `WaitGroup` and a closing goroutine so that the consumer's `for each` ends exactly when both have finished. {solution="ex23-02″}
3. Add a third stage to `pipeline.tiko` that keeps only the even squares.
4. Give `workerpool.tiko`'s workers a `Context`, and cancel it after the first five results. What must the workers do for the cancellation to take effect?
5. Remove the mutex calls from `threads.tiko` and run it several times. Then replace the increment with `FB.AtomicFetchAddLong`. What changes, and what does each version cost?

Containers and Sorting

A fixed array holds a number of elements decided when it is declared. Most real collections are not like that: a list of orders grows as orders arrive, a word count gains a key for every new word, a set of visited cells answers "have I been here?" a million times. Writing those from arrays and `redim preserve` each time is how programs get slow and wrong.

The standard library's `std/containers` module has nine collections, each built for one kind of access, plus the procedures that sort, search and rearrange them. They are ordinary generic types, written in tiko — nothing about them is built into the compiler — and they share one shape: construct one, fill it, walk it with `for each`.

```
#import "std/containers"
```

Arrays that grow: Array(of T)

examples/ch24/arraybasics.tiko

```
#import "std/containers"

sub Show( byref label as ustring, _
          byref xs as containers.Array( of int32 ) )
    print label; ":";
    for each x in xs
        print x;
    next
    print "   (count"; xs.Count( ); ", capacity"; xs.Capacity( ); ")"
end sub

dim xs as containers.Array( of int32 ) = { 5, 3, 9 }
Show( "start", xs )

xs.Push( 7 )                        '' append
Show( "push 7", xs )
print "pop gives"; xs.Pop( ); ", xs[0] is"; xs[0]   '' indices from 0

xs.Insert( 0, 1 )                   '' shifts everything after it
Show( "insert at 0", xs )
xs.Remove( 1 )                      '' keeps the order
Show( "remove at 1", xs )
xs.RemoveSwap( 0 )                 '' moves the last element into the gap
Show( "removeswap 0", xs )
```

```
xs.Reserve( 100 )                  '' room for 100 without regrowing
Show( "reserve 100", xs )
xs.Clear( )                        '' drops elements, keeps capacity
Show( "clear", xs )
xs.Shrink( )                       '' gives the capacity back
Show( "shrink", xs )
```

examples/ch24/arraybasics.out

```
start: 5 3 9   (count 3, capacity 8)
push 7: 5 3 9 7   (count 4, capacity 8)
pop gives 7, xs[0] is 5
insert at 0: 1 5 3 9   (count 4, capacity 8)
remove at 1: 1 3 9   (count 3, capacity 8)
removeswap 0: 9 3   (count 2, capacity 8)
reserve 100: 9 3   (count 2, capacity 128)
clear:   (count 0, capacity 128)
shrink:   (count 0, capacity 0)
```

An `Array( of T )` keeps a **count** — how many elements there are — and a **capacity** — how many fit before it must grow. `Push` appends; when the capacity runs out it doubles, so a long run of pushes reallocates only a handful of times and each push costs, on average, a constant amount. That is the difference from `redim preserve`, which reallocates on every call and makes a loop of appends quadratic.

Member	Does	Cost
`xs[i]`	the element at `i`, which can be assigned	O(1)
`Push( v )`, `Pop( )`	add at the end; remove and return the last	O(1) on average
`Insert( i, v )`, `Remove( i )`	add or remove at `i`, keeping the order	O(n)
`RemoveSwap( i )`	remove at `i` by moving the last element into the gap	O(1)
`Count( )`, `Capacity( )`, `IsEmpty( )`	sizes	O(1)
`Reserve( n )`, `Shrink( )`	grow capacity ahead of time; release the spare	O(n)
`Clear( )`	remove every element, **keeping** the capacity	O(n)

Indices start at 0, whatever an ordinary array's `dim a( 1 to 10 )` habit says: `xs[0]` is the first element and `xs[xs.Count( ) - 1]` the last. `RemoveSwap` is the fast

way to remove when the order does not matter. `Reserve` is worth calling when you know how many elements are coming: one allocation instead of several.

Copying an `Array` copies every element — `dim b as … = a`, assignment, and a `byval` parameter all make an independent array. Pass containers `byref` unless a copy is what you want.

Searching

examples/ch24/indexof.tiko

```
#import "std/containers"

dim scores as containers.Array( of int32 ) = { 70, 85, 90 }

'' Searching is a free procedure, not a member.
dim as int32 wanted = 85, missing = 60
print containers.IndexOf( scores, wanted )
print containers.IndexOf( scores, missing )
print containers.Contains( scores, wanted )
```

examples/ch24/indexof.out

```
 1
-1
true
```

`IndexOf` returns the first index holding the value, or −1; `Contains` says whether there is one. Both are **free procedures**, not members, and that is deliberate. They need `=` on the element type, and every member of a generic type is compiled for every element type it is used with — so a member `IndexOf` would make an `Array` of any type without `=` impossible to declare. As free procedures, the requirement lands only on the calls that make it.

Their element type is inferred from the arguments, exactly, with no conversions:

examples/ch24/indexofliteral.tiko

```
#import "std/containers"

dim scores as containers.Array( of int32 ) = { 70, 85, 90 }
print containers.IndexOf( scores, 85 )   '' 85 is an integer, not int32
```

```
indexofliteral.tiko(4,25): error TKO0809: cannot infer the type arguments
of 'indexof'; write them, as 'indexof( of ... )( ... )'
```

`85` is an `integer` and the array holds `int32`s, so no single `T` fits both. Pass a variable of the element type, as `indexof.tiko` does, or write the type: `containers.IndexOf( of int32 )( scores, 85 )`.

Searching a sorted array

`IndexOf` looks at every element until it finds one. When the array is **sorted**, a binary search halves the range at each step instead — about 20 looks for a million elements:

examples/ch24/bsearch.tiko

```
#import "std/containers"

'' A sorted array answers "where?" by halving, not by scanning.
dim xs as containers.Array( of int32 ) = { 12, 15, 15, 15, 22, 30 }
dim as int32 v = 15, hit = 22, miss = 20

print "first 15 at"; containers.LowerBound( xs, v )
print "after the 15s:"; containers.UpperBound( xs, v )
print "how many 15s:"; _
    containers.UpperBound( xs, v ) - containers.LowerBound( xs, v )

print "22 found at"; containers.BinarySearch( xs, hit )
dim r as int32 = containers.BinarySearch( xs, miss )
print "20 gives "; r; ", so it belongs at"; -( r + 1 )
```

examples/ch24/bsearch.out

```
first 15 at 1
after the 15s: 4
how many 15s: 3
22 found at 4
20 gives -5, so it belongs at 4
```

Procedure	Returns
`LowerBound( xs, v )`	the first index whose element is not less than `v`: where `v` would go, before any equal elements
`UpperBound( xs, v )`	the first index whose element is greater than `v`: just past the last equal one
`BinarySearch( xs, v )`	the index of the first element equal to `v`; when there is none, `-( insertion point ) - 1`
`PartitionPoint( xs, pred )`	the first index where `pred` is `false`, for an array where every `true` comes first

`UpperBound` minus `LowerBound` counts the copies of a value. A miss in `BinarySearch` is always negative and still says where the value belongs: `-( r + 1 )` turns it back into the index at which an `Insert` keeps the array sorted.

All four **require the array sorted by the same order they search with**; on an unsorted array they return some index without complaint. They infer `T` from both arguments exactly, as `IndexOf` does, which is why the listing names `v`, `hit` and `miss` as `int32` variables. Each has a `...By` twin taking a comparer, for an array sorted with `SortBy`.

Hashed containers

Map(of K, V)

A `Map` stores values under keys, and finds the value for a key in constant time on average:

examples/ch24/wordcount.tiko

```
#import "std/containers"

dim text( 0 to 8 ) as ustring = _
    { "to", "be", "or", "not", "to", "be", "that", "is", "it" }

dim counts as containers.Map( of ustring, int32 )
for each word in text
    counts[ word ] += 1           '' a missing key starts at 0
next
print counts.Count( ); " distinct words"

'' TryGet reads without inserting.
dim as int32 n
if counts.TryGet( "be", n ) then print "be:"; n
if counts.TryGet( "maybe", n ) = false then print "maybe: absent"

'' A Map has no order; sort the keys to print them in one.
var keys = counts.Keys( )
containers.Sort( keys )
for each k in keys
    print k; "="; counts[ k ]; " ";
next
print
```

examples/ch24/wordcount.out

```
 7 distinct words
be: 2
```

```
maybe: absent
be= 2 is= 1 it= 1 not= 1 or= 1 that= 1 to= 2
```

Member	Does
`m[k]`	the value for `k`, which can be assigned — **inserting a default value if `k` is absent**
`TryGet( k, v )`	`true` and the value in `v` if `k` is present; inserts nothing
`Contains( k )`	whether `k` is present; inserts nothing
`Add( k, v )`	inserts only if `k` is absent
`Put( k, v )`	inserts, or replaces the value of an existing key
`Remove( k )`	`true` if something was removed
`Keys( )`, `Values( )`	the keys or values, copied into an `Array`
`Pairs( )`	`(key, value)` tuples for `for each`

`counts[ word ] += 1` works on the first sighting of a word because a missing key is inserted with the value type's default, 0, before the `+= 1`. That is the reason the indexer inserts. The cost of that convenience is the next listing.

A `Map` keeps **no order**: `for each` visits its entries in whatever order the table holds them, and that order changes as the map grows. To print in a stable order, take `Keys( )` and sort it, as `wordcount.tiko` does — or use an `OrderedMap`, below.

PITFALL

Reading a `Map` with `[ ]` inserts every key it does not find.

examples/ch24/mapgrows.tiko

```
#import "std/containers"

dim stock as containers.Map( of ustring, int32 )
stock[ "apples" ] = 12
stock[ "pears" ] = 0

'' Reading through [ ] looks harmless...
if stock[ "plums" ] = 0 then print "no plums"
if stock[ "kiwis" ] = 0 then print "no kiwis"

'' ...but each miss inserted a zero.
print "items in stock map:"; stock.Count( )
```

examples/ch24/mapgrows.out

```
no plums
no kiwis
items in stock map: 4
```

Two items were stocked; the two `if`s asked about two more, and each question inserted a zero. The program printed the right answers and quietly made the map hold four entries — so a later `Count( )`, a `for each` over the stock, or a report of "every fruit we carry" is now wrong. Use `[ ]` to **write**, and `TryGet` or `Contains` to **read**.

`Add` and `Put` differ in what happens when the key is already there:

examples/ch24/addput.tiko

```
#import "std/containers"

dim ages as containers.Map( of ustring, int32 )
print ages.Add( "ada", 36 )       '' true: inserted
print ages.Add( "ada", 99 )       '' false: already there, unchanged
print ages[ "ada" ]
print ages.Put( "ada", 37 )       '' false: was there, now replaced
print ages[ "ada" ]
print ages.Remove( "ada" ); ages.Count( )
```

examples/ch24/addput.out

```
true
false
 36
false
 37
true 0
```

`Add` refuses to replace and says so; `Put` replaces and tells you whether the key was new.

Set(of T)

A `Set` holds each value at most once and answers "is it in?" in constant time on average:

examples/ch24/setops.tiko

```
#import "std/containers"

sub Show( byref label as ustring, _
          byref s as containers.Set( of int32 ) )
    var items = s.Items( )            '' a Set has no order either
```

```
    containers.Sort( items )
    print label; ":";
    for each x in items
        print x;
    next
    print
end sub

dim evens as containers.Set( of int32 ) = { 2, 4, 6, 8 }
dim small as containers.Set( of int32 ) = { 1, 2, 3, 4 }

print "add 4 again: "; evens.Add( 4 )     '' false: already present

dim both as containers.Set( of int32 ) = evens
both.IntersectWith( small )
Show( "evens and small", both )

dim either as containers.Set( of int32 ) = evens
either.UnionWith( small )
Show( "evens or small", either )

dim onlyEven as containers.Set( of int32 ) = evens
onlyEven.ExceptWith( small )
Show( "evens not small", onlyEven )
```

examples/ch24/setops.out

```
add 4 again: false
evens and small: 2 4
evens or small: 1 2 3 4 6 8
evens not small: 6 8
```

`Add` returns `false` for a value already present. `IntersectWith`, `UnionWith` and `ExceptWith` change the set they are called on, so the listing copies `evens` first — copying a `Set`, like every container here, copies its contents. Like a `Map`, a `Set` has no order; `Items( )` gives an `Array` to sort.

Keys of your own

Every built-in type can be a key. A record type needs two things: `=`, to say when two keys are the same, and a **hash** — a number computed from the key, which decides where in the table it is stored:

examples/ch24/customkey.tiko

```
#import "std/containers"
```

```
'' A key extends Hashable and overrides HashOf( ). Hash only what
'' '=' compares: equal keys must hash equally.
type Cell extends containers.Hashable
    row as int32
    col as int32
    declare constructor( byval row as int32 = 0, byval col as int32 = 0 )
    declare function HashOf( ) as uint64 override
end type

constructor Cell( byval row as int32, byval col as int32 )
    this.row = row : this.col = col
end constructor

function Cell.HashOf( ) as uint64
        return  containers.HashInt(  containers.HashOf(  row  )  )  xor
containers.HashOf( col )
end function

'' ...and '='.
operator = ( byref a as Cell, byref b as Cell ) as boolean
    return ( a.row = b.row ) andalso ( a.col = b.col )
end operator

dim terrain as containers.Map( of Cell, ustring )
dim spot as Cell = Cell( 2, 3 )
terrain[ spot ] = "forest"

dim probe as Cell = Cell( 2, 3 )            '' a different variable
dim found as ustring
if terrain.TryGet( probe, found ) then print "( 2, 3 ) is "; found
```

examples/ch24/customkey.out

```
( 2, 3 ) is forest
```

`probe` is a different variable from `spot`, with the same contents, and the map finds it: the two are equal, so they hash equally and land in the same place. The hash is the key's own `HashOf( )`, an override of the abstract one in `containers.Hashable`, which every key type extends; the module's `containers.HashOf` already has an overload for every built-in type, which is what `containers.HashOf( row )` calls. `containers.HashInt` scrambles the row's hash before the two are combined with `xor`, so that the row and the column are not interchangeable: `( 2, 3 )` and `( 3, 2 )` get different hashes.

One rule makes it work: **equal keys must produce equal hashes.** Compute the hash from exactly the fields that `=` compares. A hash that included a field `=` ignores would put two equal keys in different places, and the map would lose track of entries without any error.

Ordered containers

`OrderedMap` and `OrderedSet` keep their contents **sorted**. They are trees rather than tables, so each operation costs O(log n) rather than O(1) — and in exchange they can answer questions about order that a hashed container cannot:

examples/ch24/leaderboard.tiko

```
#import "std/containers"

'' An OrderedMap keeps its keys sorted: here, scores.
dim board as containers.OrderedMap( of int32, ustring )
board.Put( 120, "ada" )
board.Put( 95, "alan" )
board.Put( 140, "grace" )
board.Put( 110, "edsger" )

for each (score, who) in board.Pairs( )      '' always in key order
    print score; " "; who
next

dim as int32 k
dim as ustring who
if board.TryFirst( k, who ) then print "lowest:"; k; " "; who
if board.TryLast( k, who ) then print "highest:"; k; " "; who

'' The neighbours of a value that is not a key.
if board.TryFloor( 130, k ) then print "best at or below 130:"; k
if board.TryCeiling( 130, k ) then print "first at or above 130:"; k
print "anything at or above 200? "; board.TryCeiling( 200, k )
```

examples/ch24/leaderboard.out

```
 95 alan
 110 edsger
 120 ada
 140 grace
lowest: 95 alan
highest: 140 grace
best at or below 130: 120
```

```
first at or above 130: 140
anything at or above 200? false
```

`for each` visits the keys from smallest to largest, every time. `TryFirst` and `TryLast` give the smallest and largest entries. `TryFloor( k, … )` finds the largest key **at or below** `k`, and `TryCeiling` the smallest **at or above** — the nearest neighbours of a value that need not be a key at all, which is what a schedule, a price ladder or a leaderboard asks. Each returns `false` when there is no such key.

An ordered container needs only `<` on its key type, not a hash. Two keys are treated as the same when neither is less than the other. `OrderedSet( of T )` is the same tree with no values: `Add`, `Contains`, `Remove`, the set operations, and the four `Try` neighbours.

Reach for an ordered container when you need the order, not by default: the hashed ones are faster for everything else.

LinkedList(of T)

examples/ch24/queue.tiko

```
#import "std/containers"

'' A LinkedList is a queue at both ends.
dim jobs as containers.LinkedList( of ustring )
jobs.PushBack( "build" )
jobs.PushBack( "test" )
jobs.PushFront( "fetch" )              '' jumps the queue

print "next: "; jobs.Front( ); ", last: "; jobs.Back( )
do until jobs.IsEmpty( )
    print "doing "; jobs.PopFront( )
loop
print "empty: "; jobs.IsEmpty( )
```

examples/ch24/queue.out

```
next: fetch, last: test
doing fetch
doing build
doing test
empty: true
```

A `LinkedList` adds and removes at **both ends** in constant time: `PushBack`, `PushFront`, `PopBack`, `PopFront`, with `Front( )` and `Back( )` to look without removing. That makes it a queue, a stack, or both at once.

Popping an empty list returns the element type's default value — 0, or an empty string — rather than an error, so test `IsEmpty( )` first when the difference matters, as the loop above does. A `LinkedList` has no index: reaching the middle means walking to it.

Deque(of T)

A `Deque` — say "deck", short for double-ended queue — does what a `LinkedList` does at both ends, and adds what it cannot: `d[ i ]` reaches any element in constant time.

examples/ch24/window.tiko

```
#import "std/containers"

'' The last three readings: in at the back, out at the front.
dim recent as containers.Deque( of int32 )
dim readings( 0 to 5 ) as int32 = { 20, 22, 21, 25, 30, 28 }

for i as int32 = 0 to 5
    recent.PushBack( readings( i ) )
    if recent.Count( ) > 3 then recent.PopFront( )
    dim total as int32 = 0
    for each r in recent
        total += r
    next
    print "reading"; readings( i ); ": oldest"; recent[ 0 ]; _
        ", average"; total \ recent.Count( )
next

recent.PushFront( 99 )
print "front"; recent.Front( ); ", back"; recent.Back( )
```

examples/ch24/window.out

```
reading 20: oldest 20, average 20
reading 22: oldest 20, average 21
reading 21: oldest 20, average 21
reading 25: oldest 22, average 22
reading 30: oldest 21, average 25
reading 28: oldest 25, average 27
front 99, back 28
```

The listing keeps a **sliding window**: each reading goes in at the back, and once there are more than three the oldest comes off the front. `recent[ 0 ]` is always the front, and `for each` walks front to back.

A `Deque` keeps its elements in one block of memory used as a ring: the front can sit anywhere in the block, and the elements wrap round its end. When the block fills, its size doubles. So `PushBack`, `PushFront`, `PopBack` and `PopFront` each cost a constant amount on average, nothing is allocated per element, and a walk reads one contiguous block, as it does for an `Array`. Prefer it to a `LinkedList` for queues; keep the `LinkedList` for the one thing the ring cannot give, an iterator that stays valid while you add elsewhere.

The `Pop` forms return a default value on an empty deque, as `LinkedList`'s do; `TryPopFront( v )` and `TryPopBack( v )` return `false` instead. `d[ i ]`, `Front( )` and `Back( )` need an index from 0 to `Count( ) - 1`, and the deque does not grow to make a bad one valid.

PriorityQueue(of T)

A queue serves in arrival order. A **priority queue** serves whatever matters most first, however late it arrived:

examples/ch24/jobs.tiko

```
#import "std/containers"

type Job
    priority as int32
    name as ustring
end type

'' Does a leave the queue before b?  Lower number = more urgent.
type JobFirst as function( byref a as Job, byref b as Job ) as boolean

dim urgent as JobFirst = function( byref a as Job, _
                                   byref b as Job ) as boolean
    return a.priority < b.priority
end function

dim queue as containers.PriorityQueueBy( of Job, JobFirst ) = _
    containers.PriorityQueueBy( of Job, JobFirst )( urgent )

dim j as Job
j.priority = 3 : j.name = "tidy logs"    : queue.Push( j )
j.priority = 1 : j.name = "restart db"   : queue.Push( j )
j.priority = 2 : j.name = "rotate keys"  : queue.Push( j )
print "next up: "; queue.Peek( ).name

j.priority = 0 : j.name = "page on-call" : queue.Push( j )
```

```
do until queue.IsEmpty( )
    j = queue.Pop( )
    print j.priority; " "; j.name
loop
```

examples/ch24/jobs.out

```
next up: restart db
 0 page on-call
 1 restart db
 2 rotate keys
 3 tidy logs
```

`PriorityQueue( of T )` hands back its **smallest** element first, by `T`'s own `<`. For any other order — largest first, or by one field of a record, as here — `PriorityQueueBy( of T, TLess )` takes a comparer when it is constructed: `less( a, b )` is `true` when `a` must leave before `b`, the question a `SortBy` comparer answers. The second type argument is the comparer's type. A plain lambda or `@MyProc` has a procedure-pointer type, which the listing names `JobFirst`; a capturing lambda's type is written `typeof( name )`.

`Push` and `Pop` cost O(log n), and `Peek` looks at the next element without removing it. `MakeHeap( xs )` builds a queue from an `Array` in O(n), cheaper than pushing the elements one at a time. Two elements that compare equal leave in no particular order: give records a sequence number and compare on it second when the order among equals matters.

PITFALL
`for each` over a priority queue is not priority order.

examples/ch24/heaporder.tiko

```
#import "std/containers"

dim q as containers.PriorityQueue( of int32 )
dim nums( 0 to 4 ) as int32 = { 5, 1, 4, 2, 3 }
for i as int32 = 0 to 4
    q.Push( nums( i ) )
next

print "for each:";
for each n in q
    print n;
next
print
```

```
print "popping:";
do until q.IsEmpty( )
    print q.Pop( );
loop
print
```

examples/ch24/heaporder.out

```
for each: 1 2 4 5 3
popping: 1 2 3 4 5
```

The queue is a **binary heap**: an array arranged so that each element is no greater than the two below it. That arrangement is what makes `Push` and `Pop` cheap, and it is not sorted — so a loop that prints the queue, or takes "the first three" from it, gets heap order, which looks almost right. Take elements in priority order with `Pop`, or call `ToSortedArray( )`, which sorts a copy and leaves the queue as it was.

BitSet

A `BitSet` is a set of small whole numbers stored one bit each, so a million flags fit in 125 KB and testing one costs a shift and a mask:

examples/ch24/sieve.tiko

```
#import "std/containers"

const LIMIT = 50

'' Sieve of Eratosthenes: one bit per number below LIMIT.
dim composite as containers.BitSet = containers.BitSet( LIMIT )
for n as int32 = 2 to LIMIT - 1
    if composite.Test( n ) = false then
        for m as int32 = n * n to LIMIT - 1 step n
            composite.Set( m )
        next
    end if
next

dim primes as containers.BitSet
for n as int32 = 2 to LIMIT - 1
    if composite.Test( n ) = false then primes.Set( n )
next
print "primes below"; LIMIT; ":"; primes.CountOnes( )
```

```
for each p in primes
    print p;
next
print

'' Set algebra, one machine word at a time.
dim odd as containers.BitSet
for n as int32 = 1 to LIMIT - 1 step 2
    odd.Set( n )
next
dim even as containers.BitSet = primes
even.DifferenceWith( odd )
print "even primes:"; even.CountOnes( ); _
    ", the first is"; even.FirstSet( )
```

examples/ch24/sieve.out

```
primes below 50: 15
 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
even primes: 1, the first is 2
```

`Set( i )` grows the set when `i` is beyond its `Length( )`, and `Test` of a bit beyond the length is `false`, so a `BitSet` can start empty; `BitSet( n )` starts with `n` clear bits instead. `Length( )` counts addressable bits, `CountOnes( )` the ones that are set. `for each` visits the set bits in ascending order; `FirstSet( from )` and `NextClear( from )` find the next set or clear bit.

`UnionWith`, `IntersectWith`, `DifferenceWith` and `SymmetricDifferenceWith` work 64 bits at a time and change the set they are called on — which is why the listing copies `primes` before taking the odd numbers away. Compare two sets with `a.Equals( b )`: it compares the bits that are set, so sets of different lengths can be equal.

Filling a container from a list

Every container but `BitSet` takes a brace list when it is declared, one entry per element:

examples/ch24/initializers.tiko

```
#import "std/containers"

'' A brace list fills a container as it is declared.
dim primes as containers.Array( of int32 ) = { 2, 3, 5, 7 }
dim tags as containers.Set( of ustring ) = { "new", "sale" }
dim route as containers.LinkedList( of ustring ) = _
    { "home", "shop", "home" }
```

```
'' A map takes one { key, value } pair per entry.
dim ages as containers.OrderedMap( of ustring, int32 ) = _
    { { "ada", 36 }, { "alan", 41 }, { "grace", 85 } }

print primes.Count( ); tags.Count( ); route.Count( ); ages.Count( )
for each (who, age) in ages.Pairs( )
    print who; age
next
```

examples/ch24/initializers.out

```
 4 2 3 3
ada 36
alan 41
grace 85
```

A map's entries are `{ key, value }` pairs inside the outer braces. The list is a convenience for starting values; the container is an ordinary one afterwards.

Sorting

examples/ch24/sorting.tiko

```
#import "std/containers"

type Player
    name as ustring
    team as ustring
    points as int32
end type

sub Show( byref ps as containers.Array( of Player ) )
    for each p in ps
        print p.team; " "; p.name; p.points
    next
    print "--"
end sub

dim ps as containers.Array( of Player )
dim p as Player
p.name = "ada"  : p.team = "red"  : p.points = 31 : ps.Push( p )
p.name = "bob"  : p.team = "blue" : p.points = 27 : ps.Push( p )
p.name = "cy"   : p.team = "red"  : p.points = 27 : ps.Push( p )
p.name = "dee"  : p.team = "blue" : p.points = 35 : ps.Push( p )
```

```
'' Best first: a comparer says whether a goes before b.
containers.SortBy( ps, function( byref a as Player, _
                                 byref b as Player ) as boolean
    return a.points > b.points
end function )
Show( ps )

'' Then by team, STABLY: within a team, best-first is kept.
containers.StableSortBy( ps, function( byref a as Player, _
                                       byref b as Player ) as boolean
    return a.team < b.team
end function )
Show( ps )

dim nums as containers.Array( of int32 ) = { 9, 4, 7 }
print containers.IsSorted( nums );
containers.Sort( nums )
print containers.IsSorted( nums ); nums[0]; nums[1]; nums[2]
```

examples/ch24/sorting.out

```
blue dee 35
red ada 31
blue bob 27
red cy 27
--
blue dee 35
blue bob 27
red ada 31
red cy 27
--
falsetrue 4 7 9
```

Procedure	Sorts	Stable	Order by
`Sort( xs )`	an `Array`, in place	no	`<` on the element
`SortBy( xs, less )`	an `Array`, in place	no	a comparer
`StableSort( xs )`, `StableSortBy( xs, less )`	an `Array`, in place	**yes**	`<`, or a comparer
`SortList( l )`, `SortListBy( l, less )`	a `LinkedList`	yes	`<`, or a comparer
`IsSorted( xs )`, `IsSortedBy( xs, less )`	—	—	answers whether it is

A **comparer** answers "does `a` go before `b`?", as Chapter 21 described. A **stable** sort keeps elements that compare equal in the order they arrived. The listing sorts by points first, then by team stably — so within each team, the best-first order from the first sort survives. With the unstable `SortBy` for the second step, the order within a team would be whatever the algorithm left.

`Sort` is an introsort: O(n log n) in the worst case, in place, not stable. `StableSort` is a merge sort: O(n log n) too, stable, and it needs a scratch buffer as large as the array. Both move elements by swapping, so sorting an array of strings or records copies no element.

Rearranging in place

Often a full sort answers more than the question asked. The three fastest times, or the median, need only part of the order:

examples/ch24/rearrange.tiko

```
#import "std/containers"

dim times as containers.Array( of int32 ) = _
    { 41, 17, 88, 23, 17, 65, 9, 52, 23 }

'' The three fastest, in order; the rest are left unsorted.
dim fast as containers.Array( of int32 ) = times
containers.PartialSort( fast, 3 )
print "fastest:"; fast[ 0 ]; fast[ 1 ]; fast[ 2 ]

'' The median, without sorting everything.
dim middle as containers.Array( of int32 ) = times
dim k as int32 = middle.Count( ) \ 2
```

```
containers.NthElement( middle, k )
print "median:"; middle[ k ]

'' Distinct values: sort, then drop neighbours that repeat.
dim distinct as containers.Array( of int32 ) = times
containers.Sort( distinct )
print "repeats removed:"; containers.Dedup( distinct )
print "distinct values:"; distinct.Count( )

'' A turn order that moves along one seat each round.
dim seats as containers.Array( of ustring ) = { "ana", "ben", "cai" }
for r as int32 = 1 to 3
    print "round"; r; ": "; seats[ 0 ]; " starts"
    containers.RotateArray( seats, 1 )
next
```

`examples/ch24/rearrange.out`

```
fastest: 9 17 17
median: 23
repeats removed: 2
distinct values: 7
round 1: ana starts
round 2: ben starts
round 3: cai starts
```

Procedure	Leaves the array	Cost
`PartialSort( xs, k )`	the `k` smallest at the front, sorted; the rest in no order	O(n log k)
`NthElement( xs, k )`	at `xs[k]` what a full sort would put there; nothing greater before it, nothing less after	O(n) on average
`Dedup( xs )`	without **consecutive** repeats; returns how many it removed	O(n)
`ReverseArray( xs )`, `RotateArray( xs, k )`	reversed; rotated left by `k`	O(n)
`NextPermutation( xs )`	the next ordering in sorted order; `false` after the last	O(n)
`MinIndex( xs )`, `MaxIndex( xs )`	unchanged; the index of the smallest or largest, or −1	O(n)

`Dedup` removes runs, so `Sort` first when every repeat should go. Each procedure that compares has a `...By` twin taking a comparer, and every move is a swap, as in the sorts.

`ReverseArray` and `RotateArray` carry the type in their names because `std/strings` has its own `Reverse` and `std/numeric` its own `Rotate`: a program that brings in both modules with `using` would find the bare names ambiguous.

Choosing a container

Container	Find by key or value	Add	Remove	Order	Use it for
`Array`	O(n) search, O(1) by index	O(1) at the end	O(n), or O(1) with `RemoveSwap`	insertion	most lists
`Map`	O(1) average	O(1) average	O(1) average	none	lookup by key
`Set`	O(1) average	O(1) average	O(1) average	none	membership, no duplicates
`OrderedMap`	O(log n)	O(log n)	O(log n)	sorted by key	sorted lookup, nearest key
`OrderedSet`	O(log n)	O(log n)	O(log n)	sorted	sorted unique values
`LinkedList`	O(n)	O(1) at either end	O(1) at either end	insertion	iterators that survive adds
`Deque`	O(n) search, O(1) by index	O(1) at either end	O(1) at either end	insertion	queues, sliding windows
`PriorityQueue`	O(1), the smallest only	O(log n)	O(log n), the smallest only	by priority, through `Pop`	scheduling, “most urgent next”
`BitSet`	O(1)	O(1)	O(1)	ascending	flags, sets of small numbers

When in doubt, start with an `Array`. It is the most compact and the fastest to walk, and it is the right answer more often than its O(n) search suggests: for a few dozen elements, a linear scan over contiguous memory is cheap, and can cost less than hashing the key.

What invalidates a loop

Chapter 20's rule holds here, container by container. **Changing a container's structure while a `for each` walks it is not allowed**, and nothing detects it:

- **`Array`**: `Push`, `Insert`, `Remove`, `RemoveSwap`, `Clear`, `Reserve`, `Shrink` and a sort may move every element. Assigning to `xs[i]`, or through a `for each byref` variable, does not.
- **`Map`, `Set`**: adding can rehash the whole table, and removing invalidates the walk. A `Set` element must never be changed in place, even through `byref`: its hash changes, and it can no longer be found.
- **`OrderedMap`, `OrderedSet`**: adding or removing changes the tree under the loop.
- **`LinkedList`**: adding at either end leaves a walk valid; removing the node being visited does not.
- **`Deque`, `PriorityQueue`**: any push or pop may move every element.
- **`BitSet`**: setting a bit at or beyond `Length( )` grows the set under the loop.

Collect the changes in a second container during the loop, and apply them after it.

What it costs

- **Every container copies its contents when it is copied.** A `byval` parameter or an assignment of a large container is a large copy; pass `byref`.
- **A hashed container trades memory for speed**: its table has spare room so that lookups stay constant-time.
- **An ordered container allocates per element**, a node at a time, and pays O(log n) for order.
- **`Keys( )`, `Values( )` and `Items( )` build a new `Array`** every time they are called.

Summary

- `#import "std/containers"`. `Array`, `Map`, `Set`, `OrderedMap`, `OrderedSet`, `LinkedList`.
- `Array`: zero-based, `Push`/`Pop` amortised O(1), `Insert`/`Remove` O(n), `RemoveSwap` O(1), `Reserve` to pre-size. `IndexOf` and `Contains` are free procedures, and infer `T` exactly — a bare literal does not match an `int32` array (`TKO0809`).
- `Map`: **`m[k]` inserts on a miss** — read with `TryGet`/`Contains`. `Add` will not replace; `Put` will. No order: sort `Keys( )` to print.
- `Set`: unique values, with `IntersectWith`, `UnionWith`, `ExceptWith`.
- A custom key needs `=`, and extends `containers.Hashable` with a `HashOf( )` override; equal keys must hash equally.

- `OrderedMap`/`OrderedSet`: sorted, O(log n), `TryFirst`/`TryLast`/`TryFloor`/`TryCeiling`; need only `<`.
- `LinkedList`: O(1) at both ends; popping an empty list returns a default value.
- `Deque`: O(1) at both ends **and** by index — the queue to reach for first.
- `PriorityQueue`: smallest first, or any order with `PriorityQueueBy` and a comparer. `for each` is heap order, not priority order: use `Pop` or `ToSortedArray( )`.
- `BitSet`: one bit per number, grows on `Set`, set algebra a word at a time, `Equals` to compare.
- On a sorted array, `LowerBound`, `UpperBound` and `BinarySearch` find in O(log n). `PartialSort` and `NthElement` answer "the k smallest" and "the median" without a full sort.
- Brace lists fill any container but `BitSet` at its declaration; a map takes `{ key, value }` pairs.
- `Sort`/`SortBy` are fast and unstable; `StableSort`/`StableSortBy` keep equal elements in order.
- Do not change a container's structure inside a `for each` over it.

Exercises

1. Count the words of a text in a `Map` and print the three most frequent, ties broken alphabetically so the output never varies. {solution="ex24-01″}
2. Keep the booked appointment times in an `OrderedSet`. For each requested time, report whether it is booked, and if not, how long it is free until. {solution="ex24-02″}
3. Rewrite `mapgrows.tiko` so it prints the same messages and the map still holds two entries.
4. Remove every odd number from an `Array( of int32 )` without changing the array while walking it. Then do it again with `RemoveSwap`, walking from the end. Which keeps the order?
5. Give `Cell` a third field, `label`, that `=` ignores. What must `HashOf` still use, and what goes wrong if it also hashes `label`?
6. Change `window.tiko` to keep the last five readings and also print the largest reading in the window each time. `MaxIndex` takes an `Array`, not a `Deque`: decide whether to copy the window or walk it.
7. Give `Job` a sequence number and make `JobFirst` break ties with it, so two jobs of equal priority leave in the order they arrived.

Files, Processes and the Environment

Almost every useful program reads something it did not make and leaves something behind: a configuration file, a log, a folder of results, an exit code a build script checks. This chapter is about that boundary — the files a program reads and writes, the directories around them, the console it talks through, the other programs it starts, and the machines it talks to over the network.

Files are the **`std/fs` module**: an open `File` and the streams layered on it, a path algebra, a single `Stat` call for every fact about a file, directory walks with `for each`, whole-file reads and writes, and the directories a program keeps its own files in. Configuration files in TOML and INI are **`std/config`**; addresses, sockets and URLs are **`std/net`**. Every operation that can fail returns a `Result` (Chapter 18) rather than raising a runtime error, so a missing file is a value the program looks at, not a crash.

```
#import "std/fs"
```

Files and streams

Text files

A `File` is opened in place and **closes itself** when it goes out of scope. A `TextWriter` writes text to it, and `ReadLines` reads a file back a line at a time with `for each`:

examples/ch25/textfile.tiko

```
#import "std/fs"

'' Write a text file, add to it, read it back a line at a time.
scope
    dim f as fs.File
    f.Create( "notes.txt" )               '' creates, or empties, the
file
    dim w as fs.TextWriter = fs.TextWriter( f )
    w.WriteLine( "first line" )
    w.WriteLine( "second line" )
end scope                                 '' the file closes here

scope
    dim f as fs.File
    f.OpenAppend( "notes.txt" )           '' keeps what is there
    dim w as fs.TextWriter = fs.TextWriter( f )
    w.WriteLine( "third line" )
end scope
```

```
dim as int32 n = 0
for each ln in fs.ReadLines( "notes.txt" )
    n += 1
    print n; ": "; ln.Value( )
next
fs.RemoveFile( "notes.txt" )
```

examples/ch25/textfile.out

```
 1: first line
 2: second line
 3: third line
```

`Create` makes the file, or empties one that is there; `OpenAppend` keeps what is there and writes at the end; `OpenRead` opens for reading only. The `scope` blocks are where each `File` ends: its destructor closes the handle, and the `TextWriter` declared after it is destroyed first, so nothing is left unwritten. A `File` cannot be copied — pass it `byref`.

`WriteLine` ends each line the way the platform does: carriage return and line feed on Windows, line feed alone on Linux. Set the writer's `NewLine` to `chr( 10 )` for files that must be the same bytes everywhere. `ReadLines` accepts either ending, removes it, and yields one `Result( of ustring, FsError )` per line — `ln.Value( )` is the text. It holds one line in memory at a time, whatever the size of the file, and leaving the loop early closes the file.

Text is UTF-8 unless you say otherwise, and a `TextReader` and `ReadLines` also honour a byte-order mark at the start of a file, for UTF-8 and UTF-16 text written by other programs.

Text in other encodings

Not every file is UTF-8. Windows tools save UTF-16, and older programs save in a **code page** such as Windows-1252, one byte per character, in which the same letters are different bytes. Every text call in std/fs takes an encoding as an extra, last argument:

examples/ch25/encodings.tiko

```
#import "std/fs"

'' Two programs saved the same name: one in Windows-1252, one in UTF-16.
dim as ustring name_ = "M" & wchr( &hFC ) & "ller"       '' u with an
umlaut
fs.WriteAllText( "legacy.txt", name_, fs.CodePage( 1252 ) )
fs.WriteAllText( "unicode.txt", name_, fs.Utf16LE( ) )
```

```
'' Read both without knowing which is which: a BOM, else UTF-8 if the
'' bytes are valid UTF-8, else the fallback named here.
dim as fs.TextDetect guess = fs.TextDetect( fs.CodePage( 1252 ) )
dim as fs.TextEncoding found
dim as ustring a = fs.ReadAllText( "legacy.txt", found, guess ).Value( )
print "legacy.txt:  "; found.Name( )
dim as ustring u = fs.ReadAllText( "unicode.txt", found, guess ).Value( )
print "unicode.txt: "; found.Name( )
print "the same text: "; iif( a = u, "yes", "no" )

'' Windows-1252 has no arrow, so writing one is refused, and the
'' file is left as it was.
var r = fs.WriteAllText( "legacy.txt", name_ & wchr( &h2192 ), _
                         fs.CodePage( 1252 ) )
print "an arrow in 1252: "; iif( r.IsOk( ), "written", "refused" )
dim as ustring now_ = _
    fs.ReadAllText( "legacy.txt", fs.CodePage( 1252 ) ).Value( )
print "legacy.txt unchanged: "; iif( now_ = name_, "yes", "no" )

fs.RemoveFile( "legacy.txt" )
fs.RemoveFile( "unicode.txt" )
```

examples/ch25/encodings.out

```
legacy.txt:  windows-1252
unicode.txt: utf-16le
the same text: yes
an arrow in 1252: refused
legacy.txt unchanged: yes
```

An encoding is a `TextEncoding`. `Utf8( )` is the default; `Utf16LE( )` and `Utf16BE( )` write a byte-order mark unless passed `false`; `CodePage( n )` is a Windows code page by number, and `Ansi( )` the one the machine is set to, on Windows only; `BinaryMode( )` reads byte n as character n, so any file reads and writes back unchanged. `ReadAllText`, `ReadAllLines`, `WriteAllText`, `AppendAllText`, `WriteAtomic`, `ReadLines`, `TextReader` and `TextWriter` all take one, and so do `Stream.WriteText` and `MemoryStream.ToText` for bytes in memory.

A read with an encoding **believes it**: a byte-order mark that matches is skipped, and any other bytes are text. Bytes that are not valid in the encoding become U+FFFD, or fail under `TEXT_STRICT`. When the encoding is not known, pass a `TextDetect` instead, as the listing does: a byte-order mark decides, else UTF-8 if the bytes are valid

UTF-8, else the fallback it names, and `found` says which was used. A file that starts with a UTF-32 mark is refused with `FS_UNSUPPORTED`, never misread.

A write is **strict by default**. A character the encoding cannot hold fails with `FS_INVALID_DATA` before the file is opened, so the file keeps what it had. Pass `TEXT_REPLACE` to write a `?` in its place instead; the write then succeeds and its value is `false`, meaning "written, but not exactly". `AppendAllText` with an encoding refuses to add UTF-16 to a file that starts with a UTF-8 mark, or the other way round.

Opening a file

Every open returns a `Result( of boolean, FsError )`, and the error says why:

examples/ch25/openfile.tiko

```
#import "std/fs"

'' Every way of opening a file can fail, and says why.
dim f as fs.File
var r = f.OpenRead( "missing.txt" )
print "open missing: "; r.IsOk( ); ", not found: "; _
      cbool( r.Failure( ).Code( ) = fs.FS_NOT_FOUND )

'' CreateNew refuses a file that is already there.
dim o as fs.OpenOptions
o.Write = true
o.CreateNew = true
print "first create: "; f.Open( "once.txt", o ).IsOk( )
f.Close( )
var again = f.Open( "once.txt", o )
print "second create: "; again.IsOk( ); ", exists: "; _
      cbool( again.Failure( ).Code( ) = fs.FS_ALREADY_EXISTS )

'' A File refuses what it was not opened for.
f.OpenRead( "once.txt" )
var w = f.WriteUtf8( "x" )
print "write to a read-only File: "; w.IsOk( )
f.Close( )
fs.RemoveFile( "once.txt" )
```

examples/ch25/openfile.out

```
open missing: false, not found: true
first create: true
second create: false, exists: true
write to a read-only File: false
```

Method	Opens for	An existing file is	A missing file
`OpenRead( p )`	reading	read	fails, `FS_NOT_FOUND`
`Create( p )`	writing	**emptied**	created
`CreateNew( p )`	writing	refused, `FS_ALREADY_EXISTS`	created
`OpenAppend( p )`	writing at the end	kept	created
`Open( p, options )`	what `OpenOptions` says	as the options say	as the options say

An `OpenOptions` names what the program needs: `Read`, `Write`, `Append`, `Truncate`, `Create` and `CreateNew`. A combination that makes no sense — `Append` with `Truncate`, `CreateNew` without `Write` — is refused before anything is opened. An operation the `File` was not opened for fails too: writing to a file opened for reading is a failure, not a silent nothing.

Other programs may still read, write, rename and delete a file you have open; on Windows the `ShareRead`, `ShareWrite` and `ShareDelete` options narrow that. A handle is **not** passed to the programs this one starts unless `Inheritable` is set.

Fields in a line

The simplest file format a program can save and read again is one record per line with its fields separated by commas:

examples/ch25/fields.tiko

```
#import "std/fs"
#import "std/strings"

'' One record per line, fields separated by commas. A field that holds
'' a comma is quoted, and SplitCsvRow takes the quotes off again.
dim f as fs.File
f.Create( "people.csv" )
scope
    dim w as fs.TextWriter = fs.TextWriter( f )
    w.WriteLine( "ada,36,1.5" )
    w.WriteLine( """hopper, grace"",85,2" )
end scope
f.Close( )

for each ln in fs.ReadLines( "people.csv" )
    var cols = strings.SplitCsvRow( ln.Value( ) )
    dim as ustring who = cols[0]
    dim as int32 age = valint( cols[1] )
```

```
    dim as float64 rating = val( cols[2] )
    print who; " |"; age; " |"; rating
next
fs.RemoveFile( "people.csv" )
```

examples/ch25/fields.out

```
ada | 36 | 1.5
hopper, grace | 85 | 2
```

A field that contains a comma is written inside double quotes, and `strings.SplitCsvRow` splits a line into its fields with the quotes removed, so `hopper, grace` survives the round trip. `valint` and `val` turn the text fields back into numbers. For anything that other programs must read, prefer a format both sides agree on.

Records at a position

A file of fixed-size records can be read and written **by number**, without touching the records around the one you want. A `RecordFile( of T )` is such a file, of records of the type `T`:

examples/ch25/records.tiko

```
#import "std/fs"

'' A file of fixed-size records, read and written by number.
type Item
    id as int32
    qty as int32
    label as string * 8       '' fixed width: every record the same
end type

dim stock as fs.RecordFile( of Item )
stock.Open( "stock.dat" )                 '' read and write; created if
missing

dim it as Item
for n as int32 = 1 to 3
    it.id = n
    it.qty = n * 10
    it.label = "part" & n
    stock.Append( it )                    '' becomes record n
next
print "records:"; stock.Count( ).Value( ); _
      ", each"; stock.RecordLength( ); " bytes"

stock.GetRecord( 2, it )
```

```
print "record 2:"; it.qty; " "; rtrim( it.label )
it.qty -= 5
stock.PutRecord( 2, it )
stock.GetRecord( 2, it )
print "record 2 now:"; it.qty
print "record 4 there? "; stock.GetRecord( 4, it ).Value( )
stock.Close( )
fs.RemoveFile( "stock.dat" )
```

examples/ch25/records.out

```
records: 3, each 16 bytes
record 2: 20 part2
record 2 now: 15
record 4 there? false
```

`Open` opens the file for reading and writing and creates it if it is missing; `Open( p, options )` takes `OpenOptions` instead. Records are numbered from 1, and record `n` begins `( n - 1 ) * sizeof( Item )` bytes into the file. `Append` writes after the last record and returns the new record's number, `GetRecord` and `PutRecord` read and write record `n`, `Count( )` is the number of whole records, and `for each r in stock` walks them, one `Result` per record. Reading a record that is not all there — past the end, or a short last record — is not a failure: the value is `false` and the record is left untouched. Writing past the end grows the file, and the records in between read as zeros.

Every field must have a fixed size for this to work, which is why `label` is a `string * 8` rather than a `ustring`: a variable-length string is a pointer, and a pointer written to a file means nothing when it is read back. Nothing checks this for you. The bytes are the record's bytes in memory, padding included; `field = 1` removes the padding.

Underneath are `File.ReadAt( position, buffer, count )` and `WriteAt`, which read and write bytes at any byte position without moving the file's position, and may be called from several threads on one `File`. Use them for a file that mixes records with other data, and `ReadRecord( of T )`, `WriteRecord( of T )` and `RecordCount( of T )` for records in a `File` you have opened yourself.

Binary formats

A record of fixed-size fields is one kind of binary file. Most binary formats — images, archives, network messages — are laid out byte by byte instead, with each number in a stated width and **byte order**:

examples/ch25/binformat.tiko

```
```

```
#import "std/codec"
#import "std/containers"

'' A small binary format: a magic number, a version, then a count
'' and the values, each as short as it can be.
dim buf as containers.Array( of ubyte )
codec.AppendU32BE( buf, &h544B4F31 )            '' "TKO1"
codec.AppendU16LE( buf, 2 )
dim values( 0 to 3 ) as int64 = { 5, 300, -2, 1000000 }
codec.AppendUVarint( buf, 4 )
for i as int32 = 0 to 3
    codec.AppendVarint( buf, values( i ) )
next
print buf.Count( ); " bytes: "; codec.HexEncode( buf, true )

'' Reading it back, field by field.
print "magic   "; hex( codec.GetU32BE( buf, 0 ) )
print "version "; codec.GetU16LE( buf, 4 )
dim at as int32 = 6
dim n as uint64 = codec.ReadUVarint( buf, at ).Value( )
dim shown as ustring = "values"
for i as int32 = 1 to n
    shown &= " " & codec.ReadVarint( buf, at ).Value( )
next
print shown
print "read up to byte"; at
```

examples/ch25/binformat.out

```
 14 bytes: 544B4F310200040AD8040380897A
magic   544B4F31
version 2
values 5 300 -2 1000000
read up to byte 14
```

`codec.AppendU32BE` appends a 32-bit number **big-endian**, most significant byte first — the order a person reads, and the one network protocols use. `AppendU16LE` writes **little-endian**, the order x86 processors keep in memory and most file formats on Windows use. The format decides; the code must match it field by field, which is why every name carries its width and order: `Get`, `Put` and `Append`, for `U16` to `U64`, `I16` to `I64`, `F32` and `F64`, each `LE` or `BE`. A `Get` past the end of the buffer is a run-time error, the same as indexing an array past its end.

The values are **varints**: each byte carries seven bits and a flag saying whether another byte follows, so 5 takes one byte and a million three. `AppendVarint` first folds the sign

in, so that −2 stays as short as 2. `ReadUVarint( buf, at )` and `ReadVarint` read one value at `at` and move `at` past it, which is what lets the loop walk the list without knowing where each value ends.

The buffer is a `containers.Array( of ubyte )`, so `fs.WriteAllBytes` and `fs.ReadAllBytes` move it to and from a file whole, and `MemoryStream` and `File.Write` take its bytes as they are.

Streams

A `File` is a **`Stream`**: something that bytes can be read from, written to, or both. A `MemoryStream` is another — a growable buffer in memory — and everything that works on one works on the other:

examples/ch25/streams.tiko

```
#import "std/fs"

'' Build the text in memory, then copy it to a file in one step.
dim m as fs.MemoryStream
scope
    dim w as fs.TextWriter = fs.TextWriter( m )
    w.NewLine = chr( 10 )
    for i as int32 = 1 to 3
        w.WriteLine( "line " & i )
    next
end scope
print "in memory:"; m.Length( ); " bytes"

m.Seek( 0 )                                '' Copy starts at the position
dim f as fs.File
f.Create( "out.txt" )
print "copied:"; fs.Copy( m, f ).Value( ); " bytes"
f.Close( )

'' Read it back through a TextReader, which works on any stream.
f.OpenRead( "out.txt" )
dim r as fs.TextReader = fs.TextReader( f )
dim as ustring ln
do while r.ReadLine( ln ).ValueOr( false )
    print "["; ln; "]"
loop
f.Close( )
fs.RemoveFile( "out.txt" )
```

examples/ch25/streams.out

```
in memory: 21 bytes
copied: 21 bytes
[line 1]
[line 2]
[line 3]
```

`fs.Copy( source, destination )` copies from the source's current position to its end, which is why the listing seeks the `MemoryStream` back to 0 first. A `TextReader` reads lines from any stream, here the `File`; `ReadLine` returns `false` when there are no more.

Type	Is
`File`	a file on disk; **unbuffered**: each `Read` and `Write` is one call to the system
`MemoryStream`	bytes in memory; `Length( )`, `ToUtf8Text( )`
`BufferedReader`, `BufferedWriter`	a buffer over another stream, so many small reads or writes become a few large ones
`TextReader`, `TextWriter`	text over another stream: `ReadLine`, `WriteLine`

A wrapper keeps a pointer to the stream it was given, so that stream must outlive it — declare the stream first. `Read` may return fewer bytes than asked, and `Ok( 0 )` means the end; `ReadExact`, `WriteAll` and `Copy` are the loops that make short reads and writes invisible.

A stream of your own extends `fs.Stream` and overrides the operations it supports:

examples/ch25/counting.tiko

```
#import "std/fs"
#import "std/outcome"
using outcome

'' A stream of your own: it keeps nothing, and counts what passes.
type Counter extends fs.Stream
    as int64 bytes = 0
    as int64 lines = 0
    declare const function CanWrite( ) as boolean override
    declare function Write( byval buf as const ubyte ptr, _
                            byval n as longint ) as fs.FsResultI64
override
end type
```

```
const function Counter.CanWrite( ) as boolean
    return true
end function

function Counter.Write( byval buf as const ubyte ptr, _
                        byval n as longint ) as fs.FsResultI64
    if ( n < 0 ) orelse ( ( buf = nullptr ) andalso ( n > 0 ) ) then
        return fs.FsFail( of longint )( fs.FS_INVALID_ARGUMENT, "Write",
"" )
    end if
    for i as longint = 0 to n - 1
        if buf[i] = 10 then this.lines += 1
    next
    this.bytes += n
    return Ok( of longint, fs.FsError )( n )
end function

dim c as Counter
dim w as fs.TextWriter = fs.TextWriter( c )
w.NewLine = chr( 10 )
w.WriteLine( "one" )
w.WriteLine( "two" )
w.Write( "three" )
print c.lines; " lines,"; c.bytes; " bytes"
print "can read: "; c.CanRead( )
```

examples/ch25/counting.out

```
 2 lines, 13 bytes
can read: false
```

`Read`, `Write`, `Seek` and `Flush` are virtual and return a `Result`. `Counter` overrides `Write` and says so with `CanWrite`; the operations it does not override fail with `FS_UNSUPPORTED`, and `CanRead( )` answers `false`. Anything written for a `Stream` — `TextWriter`, `Copy`, a buffer — now works with it.

Paths, metadata and directories

Paths

Paths are strings, and `std/fs` has the functions to take them apart and build them without touching the disk:

examples/ch25/paths.tiko

```
```

```
#import "std/fs"

'' The path algebra: functions over strings, no file system involved.
print fs.PathJoin( "data", "2026", "report.txt" )
print fs.PathFileName( "logs/2026/report.tar.gz" )
print fs.PathStem( "report.tar.gz" ); "  "; _
      fs.PathExtension( "report.tar.gz" )
print fs.PathStem( ".gitignore" ); "  ["; _
      fs.PathExtension( ".gitignore" ); "]"
print fs.PathParent( "logs/2026/report.txt" )
print fs.PathNormalize( "logs/./2026/../report.txt" )
print fs.PathChangeExtension( "report.txt", ".md" )
```

On Windows:

`examples/ch25/paths.windows-x86_64.out`

```
data\2026\report.txt
report.tar.gz
report.tar  .gz
.gitignore  []
logs/2026
logs\report.txt
report.md
```

On Linux:

`examples/ch25/paths.out`

```
data/2026/report.txt
report.tar.gz
report.tar  .gz
.gitignore  []
logs/2026
logs/report.txt
report.md
```

Function	Gives
`PathJoin( a, b, … )`	the parts joined with the platform's separator
`PathParent( p )`, `PathFileName( p )`	everything before the last separator; the last part
`PathStem( p )`, `PathExtension( p )`	the file name without, and only, its last extension
`PathNormalize( p )`	`.` and `..` resolved, separators made uniform
`PathChangeExtension( p, ext )`	`p` with a different extension
`PathIsAbsolute( p )`, `PathStartsWith( p, prefix )`	tests
`PathRelativeTo( p, base )`	`p` written relative to `base`

The separator is `\` on Windows and `/` on Linux and macOS, and the functions use the platform's own when they build a path. A name that begins with a dot, `.gitignore`, has no extension: its stem is the whole name.

PITFALL

`PathJoin` with an absolute second part throws the first part away.

examples/ch25/pathescape.tiko

```
#import "std/fs"

'' Build a path inside "uploads" from a name somebody else chose.
dim as ustring folder = "uploads"
dim names( 0 to 2 ) as ustring = _
    { "photo.png", "/etc/passwd", "C:\Windows\win.ini" }

for each n in names
    var full = fs.PathJoin( folder, n )
   var inside = fs.PathStartsWith( fs.PathNormalize( full ), folder )
    print n; " -> "; full; "   inside: "; inside
next
```

On Windows:

examples/ch25/pathescape.windows-x86_64.out

```
photo.png -> uploads\photo.png   inside: true
/etc/passwd -> /etc/passwd   inside: false
C:\Windows\win.ini -> C:\Windows\win.ini   inside: false
```

On Linux:

examples/ch25/pathescape.out

```
photo.png -> uploads/photo.png   inside: true
/etc/passwd -> /etc/passwd   inside: false
C:\Windows\win.ini -> uploads/C:\Windows\win.ini   inside: true
```

Joining `"uploads"` with a name a user supplied looks like it keeps the file inside `uploads`. It does not: an absolute name replaces the folder, and the program reads or writes wherever the name points. Which names count as absolute differs by platform — on Linux the Windows-style name is a harmless file called `C:\Windows\win.ini` inside `uploads`; on Windows it escapes. Check containment after joining, with `PathStartsWith` on the normalized result, before using a path built from somebody else's input.

Facts about a file: Stat

examples/ch25/stat.tiko

```
#import "std/fs"

fs.WriteAllText( "hello.txt", "hello" & chr( 10 ) )

'' One call, every fact about the file.
var st = fs.Stat( "hello.txt" )
if st.IsOk( ) then
    var info = st.Value( )
    print "size:"; info.Size( ); ", file: "; info.IsFile( ); _
          ", directory: "; info.IsDirectory( )
end if

'' A file that is not there is a failure with a reason, not a zero.
var gone = fs.Stat( "no-such-file.txt" )
if gone.IsOk( ) = false then
    var code = gone.Failure( ).Code( )
    print "missing file: "; _
          iif( code = fs.FS_NOT_FOUND, "not found", "other" )
end if

'' The yes/no predicates fold every failure into false.
print fs.Exists( "hello.txt" ); fs.IsFile( "hello.txt" ); _
      fs.IsDir( "hello.txt" )
fs.RemoveFile( "hello.txt" )
print fs.Exists( "hello.txt" )
```

examples/ch25/stat.out

```
size:6, file: true, directory: false
missing file: not found
truetruefalse
false
```

`fs.Stat( path )` returns a `Result` holding a `FileInfo` — size, kind, times, read-only and hidden flags — gathered in one call. A path that is not there is a **failure** whose `FsError` says `FS_NOT_FOUND`, which is a different answer from `FS_ACCESS_DENIED`. `Exists`, `IsFile` and `IsDir` answer yes or no and fold every failure into `false`: convenient in an `if`, and unable to tell "absent" from "not allowed to look". Use `Stat` when the difference matters.

Every `std/fs` operation that can fail returns a `Result` in the same way. `Value( )` on a failure is a runtime error, so test `IsOk( )` first — the habit from Chapter 18.

Directories: walking, globbing, copying

examples/ch25/copytree.tiko

```
#import "std/fs"
#import "std/containers"

'' A small tree in a fresh temporary directory.
var made = fs.CreateTempDir( "book" )
if made.IsOk( ) = false then end 1
var root = made.Value( )
fs.CreateDirAll( fs.PathJoin( root, "src", "lib" ) )
fs.WriteAllText( fs.PathJoin( root, "readme.txt" ), "read me" )
fs.WriteAllText( fs.PathJoin( root, "src", "main.tiko" ), "print 1" )
fs.WriteAllText( fs.PathJoin( root, "src", "lib", "util.tiko" ), "" )
fs.WriteAllText( fs.PathJoin( root, "src", "lib", "todo.txt" ), "" )

'' Every entry below root, at any depth. Each element is a Result.
dim found as containers.Array( of ustring )
for each r in fs.WalkDir( root )
    if r.IsOk( ) then
        var e = r.Value( )
        found.Push( e.Depth( ) & " " & e.Name( ) & _
                    iif( e.IsDirectory( ), "/", "" ) )
    end if
next
containers.Sort( found )              '' directory order is not defined
for each s in found
    print s
next
```

```
'' Every .tiko file, at any depth.
dim hits as containers.Array( of ustring )
fs.Glob( fs.PathJoin( root, "**", "*.tiko" ), hits )
containers.Sort( hits )
for each h in hits
    print "glob: "; fs.PathFileName( h )
next

print "copied:"; fs.CopyDir( fs.PathJoin( root, "src" ), _
                            fs.PathJoin( root, "backup" ) ).Value( )
print "removed:"; fs.RemoveDirAll( root ).Value( ); _
      ", still there: "; fs.Exists( root )
```

examples/ch25/copytree.out

```
0 readme.txt
0 src/
1 lib/
1 main.tiko
2 todo.txt
2 util.tiko
glob: util.tiko
glob: main.tiko
copied: 5
removed: 12, still there: false
```

`CreateTempDir` creates a directory with a fresh, unpredictable name and returns its path — the safe way to get a scratch area. `ReadDir( p )` walks one directory; `WalkDir( p )` walks a whole tree, a directory before its contents, with `Depth( )` counting from 0 for the root's own children. Each element is a `Result`, because a walk can fail part-way — a directory it may not open is reported as a failed element, and the walk carries on. Directory order is whatever the file system gives, so the listing sorts before printing.

`Glob( pattern, results )` finds the paths that match a pattern: `*` and `?` within one part of a path, and `**` for any number of directories, including none. `CopyDir` copies a tree and `RemoveDirAll` deletes one; each returns how many objects it handled. `RemoveDirAll` deletes a link inside the tree as a link, without following it, and it opens each directory relative to the one it is already in, so even another program swapping a directory for a link while the delete runs cannot send it outside the tree. Given a path that is itself a link, it refuses with `FS_NOT_A_DIRECTORY` and deletes nothing.

Whole files, written atomically

examples/ch25/wholefile.tiko

```
#import "std/fs"
#import "std/containers"

'' Replace the whole file in one step: a reader sees the old file or
'' the new one, never half of either.
var ok = fs.WriteAtomic( "app.ini", _
    "[main]" & chr( 10 ) & "mode=fast" & chr( 10 ) )
print "written: "; ok.IsOk( )

dim lines as containers.Array( of ustring )
var n = fs.ReadAllLines( "app.ini", lines )
print n.Value( ); " lines: "; lines[0]; " / "; lines[1]
print "as text: "; fs.ReadAllText( "app.ini" ).Value( );

var missing = fs.ReadAllText( "nowhere/app.ini" )
if missing.IsOk( ) = false then
    var why = missing.Failure( )
    print "could not read: "; _
          iif( why.Code( ) = fs.FS_NOT_FOUND, "not found", "other" )
end if
fs.RemoveFile( "app.ini" )
```

examples/ch25/wholefile.out

```
written: true
 2 lines: [main] / mode=fast
as text: [main]
mode=fast
could not read: not found
```

`ReadAllText` reads a file into one `ustring`; `ReadAllLines` splits it into an `Array`, removing the line endings of either platform; `WriteAllText` writes one. Text is UTF-8 unless the file starts with a byte-order mark that says otherwise.

`WriteAtomic` writes the new content to a temporary file **in the same directory** and then replaces the target with it. A program reading the file at the same moment sees the old contents or the new, never half of each, and a crash during the write leaves the old file intact. That is the right way to save a configuration or a document. What it does not promise is survival of a power failure at that instant, which needs the operating system's cooperation beyond any library.

Mapped files, locks and temporary files

examples/ch25/advanced.tiko

```
#import "std/fs"

fs.WriteAllText( "word.txt", "tiko" )

'' Map the file into memory and change a byte in place.
dim m as fs.MemoryMappedFile
if m.Open( "word.txt", true ).IsOk( ) then
    print "mapped"; m.Size( ); " bytes"
    m.Data( )[0] = asc( "T" )
    m.Flush( )
    m.Close( )                        '' Data( ) is invalid after this
end if
print fs.ReadAllText( "word.txt" ).Value( )

'' An advisory lock: it binds only programs that also ask for it.
dim guard as fs.FileLock
print "locked: "; guard.Acquire( "word.txt" ).IsOk( ); guard.IsHeld( )
guard.Release( )
print "held after release: "; guard.IsHeld( )
fs.RemoveFile( "word.txt" )

'' A temporary file is created for you, under a fresh name.
var tmp = fs.CreateTempFile( "book" )
print "temp file exists: "; fs.IsFile( tmp.Value( ) )
fs.RemoveFile( tmp.Value( ) )         '' nothing removes it for you
```

examples/ch25/advanced.out

```
mapped4 bytes
Tiko
locked: truetrue
held after release: false
temp file exists: true
```

A `MemoryMappedFile` makes a file's bytes appear in memory: `Data( )` is a pointer to them, and a write through it changes the file. It is the fastest way to read a large file at random, and it is a raw pointer — invalid after `Close( )`, and a fault if another program shortens the file while it is mapped.

A `FileLock` keeps other cooperating programs away from a file while it is held. It is **advisory**: on Linux it binds only programs that also ask for the lock, and the library promises no more than that on Windows either. `CreateTempFile` creates a file under a fresh name and returns the path; nothing deletes it for you.

`std/fs` also has a `FileWatcher`, which reports files created, changed, renamed and deleted under a directory when the program calls its `Poll` method. Its events depend

on the platform and on timing — a single save can arrive as several events — so treat each batch as a reason to look again rather than as an exact log.

Where a program keeps its files

examples/ch25/userdirs.tiko

```
#import "std/fs"

'' Where a program keeps its files. Nothing is created: each call says
'' where the directory should be, and the program adds its own name.
var cfg = fs.ConfigDir( )
if cfg.IsOk( ) then
    var mine = fs.PathJoin( cfg.Value( ), "myapp", "settings.ini" )
    print "settings under the config dir: "; fs.PathIsAbsolute( mine )
end if
print "cache dir found: "; fs.CacheDir( ).IsOk( )

'' The program's own file, however it was started.
var exe = fs.ExecutablePath( ).ValueOr( "" )
print "running: "; fs.PathStem( exe )

'' ~ means the home directory, and only at the start.
var home = fs.HomeDir( ).ValueOr( "" )
print cbool( fs.ExpandHome( "~/notes.txt" ).ValueOr( "" ) = _
             fs.PathJoin( home, "notes.txt" ) )
print fs.ExpandHome( "a/~/b" ).ValueOr( "" )
```

examples/ch25/userdirs.out

```
settings under the config dir: true
cache dir found: true
running: userdirs
true
a/~/b
```

Function	Windows	Linux	macOS
`HomeDir( )`	the user's profile folder	`$HOME`, else the user's account entry	`$HOME`, else the user's account entry
`ConfigDir( )`	`AppData\Roaming`	`$XDG_CONFIG_HOME`, else `~/.config`	`$XDG_CONFIG_HOME`, else `~/Library/Application Support`
`DataDir( )`	`AppData\Local`	`$XDG_DATA_HOME`, else `~/.local/share`	`$XDG_DATA_HOME`, else `~/Library/Application Support`
`CacheDir( )`	`AppData\Local`	`$XDG_CACHE_HOME`, else `~/.cache`	`$XDG_CACHE_HOME`, else `~/Library/Caches`
`StateDir( )`	`AppData\Local`	`$XDG_STATE_HOME`, else `~/.local/state`	`$XDG_STATE_HOME`, else `~/Library/Application Support`
`ExecutablePath( )`	the program's own file	the program's own file	the program's own file

Nothing is created and no program name is added. Each call says where the directory should be; join your own program's name, and `CreateDirAll` it before writing there. On Linux and macOS an `XDG_` variable counts only when it holds an absolute path. `ExecutablePath( )` is the full path of the running program however it was started, which `command( 0 )` is not — the way to find a file installed beside the program. `ExpandHome` replaces a leading `~` with the home directory and leaves every other path as it is.

Checksums and digests

A **digest** is a short, fixed-size value computed from any amount of data: change one byte and it changes. `std/hash` computes them:

examples/ch25/digest.tiko

```
#import "std/hash"
#import "std/fs"
#import "std/codec"

'' One call: the algorithm, then the data.
print "sha256: "; hash.DigestHex( hash.HASH_SHA256, "abc" )
print "crc32:  "; hash.DigestHex( hash.HASH_CRC32, "123456789" )
```

```
'' In pieces: the same answer as all at once.
dim h as hash.Sha256
h.UpdateText( "a" )
h.UpdateText( "bc" )
print      "pieces:      ";      cbool(      h.FinishHex(      )      =
hash.DigestHex( hash.HASH_SHA256, "abc" ) )

'' A file, read in blocks however large it is.
dim scratch as ustring = fs.CreateTempDir( ).Value( )
dim path as ustring = fs.PathJoin( scratch, "data.txt" )
fs.WriteAllText( path, "abc" )
print "file:   "; codec.HexEncode( hash.DigestFile( hash.HASH_SHA256,
path ).Value( ) )
fs.RemoveDirAll( scratch )

'' A keyed hash: only someone with the key can produce it.
print "hmac:   "; hash.HmacHex( hash.HASH_SHA256, "key", _
    "The quick brown fox jumps over the lazy dog" )
```

examples/ch25/digest.out

```
sha256:
ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
crc32:  cbf43926
pieces: true
file:   ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
hmac:   f7bc83f430538424b13298e6aa6fb143ef4d59a14946175997479dbc2d1a3cd8
```

`DigestHex( algorithm, data )` is the one-call form. Every algorithm is also a type that takes its data in pieces — `UpdateText`, `UpdateBytes`, or `Update( pointer, count )` — and gives the same answer as the one call, so a file of any size is hashed a block at a time; `DigestFile` does that for you. `HmacHex` mixes in a key, so only someone holding the key can produce the value: it is how a server checks that a message came from someone it trusts.

Algorithms	Use them for
`HASH_CRC32`, `HASH_CRC32C`, `HASH_ADLER32`	catching accidental corruption: zip, PNG, network frames
`HASH_FNV1A32`, `HASH_FNV1A64`, `HASH_XXHASH64`	fast hashing of keys and large data
`HASH_SHA256`, `HASH_SHA224`, `HASH_SHA512`, `HASH_SHA384`	**anything someone might tamper with**: downloads, signatures, content addresses
`HASH_MD5`, `HASH_SHA1`	only to match a format or protocol that already uses them

The first two rows are not security tools: anyone can make two different files with the same CRC or FNV value on purpose, and MD5 and SHA-1 have the same weakness. To compare two MACs, use `hash.ConstantTimeEquals`, which takes the same time whatever the first differing byte is.

Configuration files and CSV

A program that people install needs settings they can edit, and a program that produces data is often asked for a spreadsheet. `std/config` reads and writes the two configuration formats most programs meet, **TOML** and **INI**; the CSV writer sits in `std/strings`, beside `SplitCsv`.

Reading TOML

TOML is a configuration format with real types: strings, integers, floats, booleans, dates and times, arrays, and tables that nest. `config.ParseToml` turns the text into a tree of `TomlValue`s, and `config.LoadToml( path )` does the same for a file:

examples/ch25/tomlconfig.tiko

```
#import "std/config"

dim doc as ustring = """
    # the service's settings
    title = "inventory"
    debug = false

    [server]
    host = "0.0.0.0"
    port = 8080
    started = 2026-10-19T09:30:00Z

    [[users]]
    login = "ada"
```

```
    roles = [ "admin", "dev" ]

    [[users]]
    login = "lin"
    roles = [ "dev" ]
    """

dim r as config.ConfigResultToml = config.ParseToml( doc )
if r.IsOk( ) = false then
    print r.Failure( ).Message( )
    end 1
end if
dim cfg as config.TomlValue = r.Value( )

'' The value at a path, or an empty TomlValue when there is none,
'' whose AsInt( ), AsString( ) and the rest are then empty too.
function Setting( byref t as config.TomlValue, byref p as ustring ) _
        as config.TomlValue
    return t.Path( p ).ValueOr( config.TomlValue( ) )
end function

print "title:   " & Setting( cfg, "title" ).AsString( ).ValueOr( "?" )
print "port:    " & Setting( cfg, "server.port" ).AsInt( ).ValueOr( 80 )
print "debug:   " & Setting( cfg, "debug" ).AsBool( ).ValueOr( true )
print "wait:    " & Setting( cfg, "server.wait" ).AsInt( ).ValueOr( 30 )

dim started as config.TomlValue = Setting( cfg, "server.started" )
print "started: " & started.AsDateTime( ).Value( ).ToIsoString( )

'' [[users]] is an array of tables: index it, or walk it with Count.
dim users as config.TomlValue = Setting( cfg, "users" )
for i as int32 = 0 to users.Count( ) - 1
    print Setting( users[i], "login" ).AsString( ).Value( ) _
        & ": " & Setting( users[i], "roles" ).Count( ) & " role(s)"
next
print "second user's first role: " _
    & Setting( cfg, "users.1.roles.0" ).AsString( ).ValueOr( "?" )
```

examples/ch25/tomlconfig.out

```
title:   inventory
port:    8080
debug:   false
wait:    30
started: 2026-10-19T09:30:00Z
```

```
ada: 2 role(s)
lin: 1 role(s)
second user's first role: dev
```

Every lookup answers a value or nothing, never a crash. `Path( "server.port" )` follows keys separated by dots, with a number indexing an array (`"users.1.roles.0"`), and returns an `Optional`; `AsInt( )`, `AsString( )`, `AsBool( )`, `AsFloat( )` return one too, empty when the value is of another kind. `Setting` folds the two steps into one, so a missing key such as `server.wait` falls through to the default the program chose. A default-constructed `TomlValue` has the kind `TOML_NONE`, and every `As…` on it is empty.

TOML	`Kind( )`	Read with
`"text"`, `'text'`, `"""…"""`	`TOML_STRING`	`AsString( )`
`42`, `0xFF`, `1_000`	`TOML_INT`	`AsInt( )`, a 64-bit `longint`
`3.14`, `5e+22`, `inf`, `nan`	`TOML_FLOAT`	`AsFloat( )`, which also reads an integer
`true`, `false`	`TOML_BOOL`	`AsBool( )`
`2026-10-19T09:30:00Z`	`TOML_OFFSET_DATETIME`	`AsDateTime( )`, a `chrono.DateTime` with an offset
`2026-10-19T09:30:00`	`TOML_LOCAL_DATETIME`	`AsDateTime( )`, without one
`2026-10-19`, `09:30:00`	`TOML_LOCAL_DATE`, `TOML_LOCAL_TIME`	`AsLocalDate( )`, `AsLocalTime( )`
`[ 1, 2 ]`, `[[users]]`	`TOML_ARRAY`	`Count( )` and `[ i ]`
`[server]`, `{ x = 1 }`	`TOML_TABLE`	`Get`, `Find`, `Path`, `Keys`, `KeyAt( i )`

A table keeps its keys **in the order the file wrote them**, so `Keys( )` and `KeyAt( i )` walk it the way a reader of the file would. `[[users]]` is an *array of tables*: each `[[users]]` header starts one more table in the `users` array.

When a document is wrong

`ParseToml` refuses a document that breaks the format's rules, and the `ConfigError` says where:

examples/ch25/tomlerrors.tiko

```
#import "std/config"
```

```
dim docs( 0 to 3 ) as ustring = { _
    !"port = 80\nport = 8080", _
    !"[server]\nhost = \"a\"\n[server]", _
    !"[fruit]\napple.color = \"red\"\n[fruit.apple]", _
    "size = 9223372036854775808" }

print "Rejected documents:"
for i as int32 = 0 to 3
    dim r as config.ConfigResultToml = config.ParseToml( docs( i ) )
    print r.Failure( ).Message( )
next
```

examples/ch25/tomlerrors.out

```
Rejected documents:
line 2, column 1: duplicate key: 'port'
line 3, column 1: duplicate key: table 'server' already defined
line 3, column 1: duplicate key: table 'fruit.apple' already defined
line 1, column 8: number out of range: 9223372036854775808
```

A key, or a table, is defined once. The third document is the subtle one: `apple.color` under `[fruit]` *defines* the table `fruit.apple` through its dotted key, so a later `[fruit.apple]` header is a second definition. An integer that does not fit in 64 bits is an error rather than a float that has quietly lost digits. `Code( )` gives the reason as a `ConfigErrorCode` — `CFG_SYNTAX`, `CFG_DUPLICATE_KEY`, `CFG_TYPE_REDEFINED`, `CFG_INVALID_ESCAPE`, `CFG_NUMBER_RANGE`, `CFG_INVALID_DATETIME` — and `Line( )` and `Column( )` the position, so a program can point its user at the line to fix.

Changing and writing TOML

The tree can be changed and written back:

examples/ch25/tomlwrite.tiko

```
#import "std/config"

dim r as config.ConfigResultToml = config.ParseToml( """
    # edited by the program below
    [server]
    port = 8080
    title = "inventory"
    """ )
dim cfg as config.TomlValue = r.Value( )

'' Change a value, add a key and a whole table.
dim server as config.TomlValue ptr = cfg.Find( "server" )
```

```
server->SetKey( "port", config.TomlValue.FromInt( 9090 ) )
server->SetKey( "tls", config.TomlValue.FromBool( true ) )

dim logging as config.TomlValue = config.TomlValue.NewTable( )
logging.SetKey( "level", config.TomlValue.FromString( "info" ) )
dim files as config.TomlValue = config.TomlValue.NewArray( )
files.Push( config.TomlValue.FromString( "app.log" ) )
files.Push( config.TomlValue.FromString( "audit.log" ) )
logging.SetKey( "files", files )
cfg.SetKey( "logging", logging )

print config.ToTomlText( cfg ).Value( );
```

examples/ch25/tomlwrite.out

```
[server]
port = 9090
title = "inventory"
tls = true

[logging]
level = "info"
files = ["app.log", "audit.log"]
```

`Find( key )` returns a pointer into the tree, so `server->SetKey` changes the table in place; `Get( key )` returns a copy. `SetKey` replaces a key's value or adds the key at the end, `Push` appends to an array, and `RemoveKey` deletes. `FromInt`, `FromString`, `FromBool`, `FromFloat`, `FromDateTime`, `FromLocalDate`, `FromLocalTime`, `NewTable` and `NewArray` make the values.

`config.ToTomlText( root )` writes the tree out: each table's plain keys first, then its sub-tables as `[section]` headers, in the order they were added. **The comment is gone.** The tree holds values, not the text they came from, so a program that rewrites a hand-edited file loses its comments. Write settings the program owns to a file of their own, and leave the user's file to the user.

INI files

INI is older and simpler: sections of `key = value` lines, every value text.

examples/ch25/inifile.tiko

```
#import "std/config"

dim r as config.ConfigResultIni = config.ParseIni( """
    ; written by hand
    [Database]
```

```
    Host = db.example.org
    port: 5432
    motd = "  spaces kept  "
    """ )
dim ini as config.IniFile = r.Value( )

print "Settings:"
'' Section and key names ignore case.
print ini.Get( "database", "host" ).ValueOr( "?" )
print "[" & ini.Get( "Database", "motd" ).ValueOr( "" ) & "]"
print ini.Get( "Database", "timeout" ).ValueOr( "30" )   '' a default

ini.SetValue( "Database", "port", "5433" )
ini.SetValue( "Cache", "size", "64" )
print ini.ToText( );
```

examples/ch25/inifile.out

```
Settings:
db.example.org
[  spaces kept  ]
30
[Database]
Host = db.example.org
port = 5433
motd = "  spaces kept  "

[Cache]
size = 64
```

Names ignore case unless `IniOptions.CaseSensitiveKeys` says otherwise, so `Get( "database", "host" )` finds `Host` under `[Database]`. `key: value` works as well as `key = value`. A line beginning with `;` or `#` is a comment, but a `;` later in a line belongs to the value. Spaces around a value are dropped unless the value is in double quotes, and the quotes themselves are removed. Keys before the first `[section]` belong to the section `""`.

`SetValue` replaces a value or adds the key, and the section, at the end; `Remove( section, key )` deletes a key and `Remove( section )` a whole section. `ToText( )` writes the file back, quoting the value that needed its quotes. A repeated key replaces the first one, which is what most INI readers do; `IniOptions.Duplicates = config.INI_DUP_ERROR` makes it an error instead. `config.LoadIni( path )` and `config.SaveIni( ini, path )` read and write the file itself.

Writing CSV

`strings.JoinCsvRow` turns an `Array( of ustring )` into one line of CSV, and `CsvWriter` writes rows to any `fs.Stream`:

examples/ch25/csvwrite.tiko

```
#import "std/strings"
#import "std/containers"
#import "std/fs"
using containers

dim header as Array( of ustring ) = { "item", "note", "qty" }
dim row1 as Array( of ustring ) = { "bolt", "M6, steel", "200" }
dim row2 as Array( of ustring ) = { "nut", !"says \"hex\"", "150" }

print strings.JoinCsvRow( row1 )

'' Any stream will do: a file, a pipe, or memory, as here.
dim buffer as fs.MemoryStream
dim w as strings.CsvWriter = strings.CsvWriter( buffer, ",", false )
w.WriteRow( header )
w.WriteRow( row1 )
w.WriteRow( row2 )
print "written: "; w.Finish( ).IsOk( )
print buffer.ToUtf8Text( );

'' SplitCsv reads back exactly what was written.
dim text as ustring = buffer.ToUtf8Text( )
dim rows as Array( of strings.CsvURow ) = strings.SplitCsv( text )
print rows.Count( ) & " rows; row 2, field 1: " & rows[2].fields[1]
```

examples/ch25/csvwrite.out

```
bolt,"M6, steel",200
written: true
item,note,qty
bolt,"M6, steel",200
nut,"says ""hex""",150
3 rows; row 2, field 1: says "hex"
```

A field is quoted only when it has to be: when it holds the separator, a double quote or a line break, or starts or ends with a space. A quote inside a field is doubled. That is the standard CSV form (RFC 4180), and `SplitCsv` reads it back field for field, as the last line shows. Records end with CR LF, the standard's line ending, unless the third argument is `false`; the listing uses LF so that its output prints cleanly. `CsvWriter` keeps the first failure: later rows are not written, and `Finish( )` reports it.

Coming from the file statements

tiko still compiles the file statements of the BASIC family — `open`, `print #`, `line input #`, `get #`, `put #`, `kill`, `mkdir` and the rest — so older programs keep working. New code uses `std/fs`, where a failure is a value rather than a runtime error or a silent nothing:

Statement	Does	std/fs
`open … for output` / `append` / `input`	open a text file	`File.Create`, `OpenAppend`, `OpenRead`
`print #`, `line input #`, `eof`	write and read lines	`TextWriter.WriteLine`, `ReadLines`, `TextReader.ReadLine`
`write #`, `input #`	comma-separated fields	`WriteLine` of the fields, `strings.SplitCsvRow`
`open … for random`, `get #`, `put #`	fixed-size records	`RecordFile`: `GetRecord`, `PutRecord`, `Append`
`open … encoding "utf-16"`	text in another encoding	an encoding argument: `ReadAllText( p, Utf16LE( ) )`
`lof( f )`, `seek`	length, position	`File.Size( )`, `Seek`
`mkdir p`, `rmdir p`	create or remove a directory	`CreateDir`, `CreateDirAll`, `RemoveDir`, `RemoveDirAll`
`chdir p`, `curdir( )`	change or read the current directory	`SetCurrentDir`, `CurrentDir`
`kill p`	delete a file	`RemoveFile`
`name old as new`	rename	`MoveFile` (which also crosses drives)
`dir( pattern )`, `dir( )`	first match, then next match	`ReadDir`, `Glob`
`exepath( )`	the folder the program is in	`PathParent( ExecutablePath( ).Value`

`dir( )` keeps one listing per thread, so two listings cannot be interleaved and a goroutine cannot continue the main program's (Chapter 23). `ReadDir` has neither limit.

The console as a device

The console is something a program can draw on — but only when there is a console. When the output is redirected to a file or a pipe, colour and cursor movement become stray control bytes in someone else's data:

examples/ch25/console.tiko

```
#include once "fbio.bi"

'' Colour and cursor positions are for a person at a console. When the
'' output goes to a file or a pipe, print plain text instead.
if IsRedirected( ) then
    print "report (plain): 3 warnings"
else
    cls
    color 14
    locate 2, 4
    print "report: 3 warnings"
    color 7
end if
```

examples/ch25/console.out

```
report (plain): 3 warnings
```

`IsRedirected( )`, from `#include once "fbio.bi"`, is true when standard output is not a console; `IsRedirected( true )` asks the same about standard input. The listing's output above is the redirected branch, because that is how the book's harness runs it. At a console, `cls` clears the screen, `color` sets the text colour, `locate row, column` moves the cursor, and `width` sets the console's size in characters.

For keyboard input, `inkey( )` returns the key waiting to be read, or an empty string at once if there is none — the right call inside a loop that must keep working. `getkey( )` waits for a key and returns its code. `sleep ms` pauses for that many milliseconds. `open cons` and `open err` open the console's output and its error stream as file numbers, for code written against `print #` (Chapter 19 traced to `open err`).

Processes and the environment

Arguments and environment variables

examples/ch25/envreport.tiko

```
'' command( n ) is the n-th argument; command( ) is all of them.
dim as int32 i = 1
do while len( command( i ) ) > 0
    print i; ": "; command( i )
    i += 1
loop
print "all: "; command( )

'' Environment variables: set one, read it, read one that is not set.
```

```
setenviron "REPORT_MODE=draft"
print "REPORT_MODE="; environ( "REPORT_MODE" )
print "UNSET_VARIABLE=["; environ( "UNSET_VARIABLE_XYZ" ); "]"
```

Run as `envreport alpha "two words" 3`:

examples/ch25/envreport.out

```
 1: alpha
 2: two words
 3: 3
all: alpha two words 3
REPORT_MODE=draft
UNSET_VARIABLE=[]
```

`command( n )` is the `n`-th argument, with quotes already removed, and an empty string past the last one; `command( 0 )` is the program's own path, and `command( )` all the arguments as one string. `environ( name )` reads an environment variable — an empty string if it is not set — and `setenviron "NAME=value"` sets one for this process and the programs it starts.

In the main module, `__FB_ARGC__` is the number of arguments counting the program itself, and `__FB_ARGV__` the C array of them, a `zstring ptr ptr` — the form a C library that parses its own command line expects. They belong to the main module's top level, as in fbc: inside a procedure they are `TKO0743: '__fb_argc__' is not declared`, and another module's top level reads 0.

The std/os module

`command` and `environ` are short to write, but each loses something: `command( n )` cannot show an empty argument, because an empty string already means "past the last one", and `environ` cannot tell a variable that is not set from one set to nothing. `std/os` keeps both apart:

examples/ch25/osenv.tiko

```
#import "std/os"
#import "std/containers"
#import "std/outcome"

'' Every argument, split as the shell passed it; an empty one is kept.
dim args as containers.Array( of ustring ) = os.Args( )
for i as int32 = 1 to args.Count( ) - 1
    print i; ": ["; args[ i ]; "]"
next

'' GetEnv tells a variable that is not set from one set to nothing.
```

```
os.SetEnv( "REPORT_MODE", "" )
dim m as outcome.Optional( of ustring ) = os.GetEnv( "REPORT_MODE" )
print "REPORT_MODE set? "; m.HasValue( ); ", value ["; m.ValueOr( "?" );
"]"
print              "UNSET_VARIABLE_XYZ              set?              ";
os.GetEnv( "UNSET_VARIABLE_XYZ" ).HasValue( )

'' $NAME and ${NAME} inside a string.
os.SetEnv( "REPORT_MODE", "draft" )
print os.ExpandEnv( "mode: $REPORT_MODE, file: ${REPORT_MODE}.txt" )

'' A refused name is a failure value, not a crash.
dim r as os.OsResultBool = os.SetEnv( "A=B", "x" )
print r.Failure( ).Message( )

'' Facts about the machine.
print "cpus > 0? "; cbool( os.CpuCount( ) > 0 ); ", page: "; os.PageSize( )
```

Run as `osenv alpha "two words" ""`:

`examples/ch25/osenv.out`

```
 1: [alpha]
 2: [two words]
 3: []
REPORT_MODE set? true, value []
UNSET_VARIABLE_XYZ set? false
mode: draft, file: draft.txt
SetEnv 'A=B': invalid argument
cpus > 0? true, page:  4096
```

`os.Args( )` is every argument in an `Array( of ustring )`, the program's own path first, Unicode on both systems. `os.GetEnv` returns an `Optional`: empty when the variable is not set, holding `""` when it is set to nothing. `SetEnv` and `UnsetEnv` return a `Result`, and refuse a name that is empty or contains `=` with an `OsError` rather than passing it to the system. `ExpandEnv` replaces `$NAME` and `${NAME}` — and on Windows `%NAME%` — with their values. `Environment( )` is a sorted snapshot of every variable, as an `OrderedMap`.

The module also answers questions about the machine: `OS_NAME` and `ARCH_NAME` (constants), `OsVersion( )`, `Hostname( )`, `UserName( )`, `ProcessId( )`, `PageSize( )`, `TotalMemory( )`, `IsTerminal( fd )`, and `CpuCount( )` — the processors this process may use, which inside a container or under an affinity mask is fewer than the machine has. That is the number to size a pool of workers by.

Set the environment before starting threads. On Linux and macOS, changing a variable while another thread reads the environment is not safe, whichever function does it.

Running other programs

examples/ch25/pipe.tiko

```
'' Run a command through the shell, and read its output through a pipe.
var code = shell( "exit 3" )
print "shell returned"; code

var f = freefile( )
open pipe "echo from the child" for input as #f
dim as ustring got
line input #f, got
close #f
print "read: "; trim( got )
```

examples/ch25/pipe.out

```
shell returned 3
read: from the child
```

Statement	Runs	Returns
`shell( "command" )`	a command line, through the system shell	its exit code
`exec( program, args )`	a program, directly	its exit code
`chain( program )`	a program	its exit code
`run program`	a program	nothing: the current program ends
`open pipe "command" for input as #f`	a command whose output the program reads	—

`shell` goes through `cmd.exe` on Windows and `/bin/sh` on Linux and macOS, so the command line is in that shell's language; `exec` starts a program with no shell in between, which is safer when an argument came from a user. `open pipe … for input` reads what the command prints, a line at a time, as if it were a file; `for output` sends lines to its input instead.

Exit codes

examples/ch25/exitcode.tiko

```
'' The exit code is what a script or a build tool checks.
dim as int32 errors = 4
```

```
print errors; " errors"
if errors > 0 then end errors     '' the process exits with status 4
print "never printed"
```

examples/ch25/exitcode.out

```
 4 errors
```

`end n` ends the program with exit status `n`. Zero means success to every shell and build tool; anything else is a failure whose number the caller can test — `$?` in a Unix shell, `%ERRORLEVEL%` in `cmd.exe`, or the return of `shell` and `exec` in another tiko program. A program that can fail should say so through its exit code, not only through what it prints.

`os.ExitWith( n )` from `std/os` is `end n` under a name a procedure can pass around, and `os.Abort( )` ends the program at once, skipping every destructor and every buffered write.

Stopping cleanly on Ctrl-C

Ctrl-C normally ends a console program wherever it happens to be — halfway through writing a file, with a temporary directory left behind. `os.CatchInterrupt( )` turns it into a flag instead:

examples/ch25/interrupt.tiko

```
#import "std/os"

'' From here on, Ctrl-C sets a flag instead of ending the program.
os.CatchInterrupt( )

dim as int32 done_ = 0
while os.Interrupted( ) = false andalso done_ < 20
    sleep 5                          '' one unit of work
    done_ += 1
wend

if os.Interrupted( ) then
    print "stopped early after"; done_; " steps"
    os.ExitWith( 130 )
end if
print "finished all"; done_; " steps"
```

Left alone, it prints:

examples/ch25/interrupt.out

```
finished all 20 steps
```

Press Ctrl-C while it runs and the loop ends at the next check, prints `stopped early after` and exits with 130, the code a shell reports for a program stopped by Ctrl-C. The program decides where it is safe to stop; nothing of yours runs at the moment the key is pressed.

`os.Interrupted( )` asks without waiting, `os.WaitInterrupt( ms )` waits, and `os.LastInterrupt( )` says which kind arrived: `INT_CTRL_C`, `INT_BREAK` (Ctrl-Break, Windows), `INT_TERMINATE` (`SIGTERM` on Linux and macOS; the console window closing, logoff or shutdown on Windows) or `INT_HANGUP` (`SIGHUP`). **A second Ctrl-C ends the program at once**, with code 130, so a program that is stuck can still be stopped; `os.CatchInterrupt( false )` turns that off. On Windows a closing console window gives the program about five seconds before the system ends it regardless.

Talking over the network

`std/net` holds the three things a program needs to talk to another machine: **addresses** to say where, **sockets** to carry the bytes — TCP for a connection, UDP for single messages — and **URLs**, the text form in which addresses usually arrive.

Addresses

examples/ch25/ipaddr.tiko

```
#import "std/net"

'' Text in, the canonical form out: IPv6 in its shortest lower-case
spelling.
dim texts( 0 to 4 ) as ustring = { "192.168.1.20",
"2001:0DB8:0:0:0:0:0:1", _
    "::ffff:10.0.0.7", "fe80::1%eth0", "010.0.0.1" }
for i as long = 0 to 4
    dim r as net.NetResultIp = net.ParseIp( texts( i ) )
    if r.IsOk( ) then
        dim a as net.IpAddress = r.Value( )
        print texts( i ); " -> "; a.ToString( ); iif( a.IsPrivate( ),
" (private)", "" ); _
                    iif( a.IsLinkLocal( ), " (link-local)", "" );
iif( a.IsV4Mapped( ), " (IPv4 as IPv6)", "" )
    else
        print texts( i ); " -> "; r.Failure( ).Message( )
    end if
next

'' An address and a port.
dim sa as net.SocketAddress =
```

```
net.ParseSocketAddress( "[::1]:8080" ).Value( )
print sa.Ip.IsLoopback( ), sa.Port, sa.ToString( )
```

examples/ch25/ipaddr.out

```
192.168.1.20 -> 192.168.1.20 (private)
2001:0DB8:0:0:0:0:0:1 -> 2001:db8::1
::ffff:10.0.0.7 -> ::ffff:10.0.0.7 (private) (IPv4 as IPv6)
fe80::1%eth0 -> fe80::1%eth0 (link-local)
010.0.0.1 -> ParseIp '010.0.0.1': invalid address
true          8080          [::1]:8080
```

An `IpAddress` is a value, IPv4 or IPv6, that prints in one canonical form whatever way it was written: IPv6 in lower case with the longest run of zero groups as `::`. **Parsing is strict.** `010.0.0.1` is refused rather than read — some systems read a leading zero as octal and reach `8.0.0.1`, others as decimal, and a filter that checks one reading while a connection uses the other is a security hole. The `Is…` functions classify an address — `IsLoopback`, `IsPrivate`, `IsLinkLocal`, `IsMulticast`, `IsGlobalUnicast` — and look through an IPv4 address written as IPv6. A `SocketAddress` adds a port; written as text an IPv6 one needs brackets, `[::1]:8080`, because the address itself is full of colons.

`net.Resolve( "example.com", 443 )` asks the system's resolver for a name's addresses and returns them all, each with the port. An address written as text comes back as itself without a lookup.

A TCP connection

examples/ch25/tcpecho.tiko

```
#import "std/net"
#import "std/fs"

'' A server: port 0 lets the system pick a free one.
dim server as net.TcpListener
if server.Bind( net.SocketAddress( net.Loopback( ), 0 ) ).IsOk( ) = false
then end 1
dim where as net.SocketAddress = server.LocalAddress( ).Value( )

'' A client connects; the system completes it before Accept is called,
'' so one thread can play both parts.
dim client as net.TcpStream
dim r as net.NetResultBool = client.Connect( where, 2000 )
if r.IsOk( ) = false then
    print r.Failure( ).Message( )
```

```
    end 1
end if
dim conn as net.TcpStream
server.Accept( conn, 2000 )

'' Both ends are streams: WriteUtf8 out, ReadExact in.
client.WriteUtf8( "hello" )
dim as string buf = space( 5 )
conn.ReadExact( cptr( ubyte ptr, strptr( buf ) ), 5 )
print "server got: "; buf
conn.WriteUtf8( ucase( buf ) )
client.ReadExact( cptr( ubyte ptr, strptr( buf ) ), 5 )
print "client got: "; buf

'' Shutdown( write ) ends the data: the server's next read returns 0
bytes.
client.Shutdown( net.SHUTDOWN_WRITE )
dim as ubyte one( 0 to 0 )
print "server read after shutdown: "; conn.Read( @one( 0 ), 1 ).Value( );
" bytes"

'' Nothing more is coming, so a timed read fails instead of waiting for
ever.
client.SetReadTimeout( 100 )
dim t as fs.FsResultI64 = client.Read( @one( 0 ), 1 )
print "client read: "; t.Failure( ).Message( )

'' A port nothing listens on any more.
server.Close( )
dim late as net.TcpStream
print "refused? "; cbool( late.Connect( where, 5000 ).Failure( ).Code( )
= net.NET_CONNECTION_REFUSED )
```

examples/ch25/tcpecho.out

```
server got: hello
client got: HELLO
server read after shutdown:  0 bytes
client read: timed out
refused? true
```

A `TcpListener` is a server: `Bind` to an address and port — port 0 asks the system for a free one, which `LocalAddress( )` then reports — and `Accept` each connection into a `TcpStream`. A client `TcpStream` calls `Connect` with a `SocketAddress`, or `ConnectHost( "example.com", 80 )`, which resolves the name and tries its addresses

in turn. Each of them takes a timeout in milliseconds; without one a call waits as long as the system lets it.

A `TcpStream` is a stream, the same `fs.Stream` as a `File`: `WriteUtf8`, `WriteAll`, `ReadExact`, `ReadToEnd`, a `BufferedReader` or a `TextReader` all work on a connection. A read returns what has arrived, which may be less than asked for — TCP carries a stream of bytes, not messages, so `ReadExact` or a line reader is what puts the messages back together. A read of 0 bytes means the other side has finished sending, as `Shutdown( SHUTDOWN_WRITE )` shows. Reads and writes report `FsError`s like a file's: `FS_TIMED_OUT` after `SetReadTimeout`, `FS_CONNECTION_RESET` when the peer drops the connection. Everything else — connecting, binding, accepting — reports a `NetError`, whose code says what happened: `NET_CONNECTION_REFUSED`, `NET_TIMED_OUT`, `NET_ADDRESS_IN_USE`, `NET_HOST_NOT_FOUND`.

A `UdpSocket` sends single datagrams instead: `Bind`, then `SendTo( p, n, address )` and `ReceiveFrom( p, n, sender )`. A datagram arrives whole or not at all, and one longer than the buffer fails with `NET_MESSAGE_TOO_LONG` rather than arriving cut short without a word.

Every socket call blocks until it is done or its timeout ends. A goroutine (Chapter 23) in a socket call holds its worker thread while it waits, so a server that gives each connection a goroutine should give every connection a timeout, or serve no more connections at once than it has workers.

URLs

`examples/ch25/urls.tiko`

```
#import "std/net"

dim u as net.Url = net.ParseUrl( "https://ann@example.com:8443/docs/a%20
b.html?q=tiko+lang&page=2#intro" ).Value( )
print "scheme   "; u.Scheme
print "user     "; u.User
print "host     "; u.Host; "  port "; u.Port
print "path     "; u.Path                          '' decoded
print "query    "; u.RawQuery                      '' as written
print "q        "; u.Query( ).Get( "q" ).ValueOr( "" )
print "fragment "; u.Fragment

'' A link on that page, made absolute.
print u.Resolve( "../img/logo.png" ).Value( ).ToString( )
print u.Resolve( "?page=3" ).Value( ).ToString( )

'' Built from fields: each part is escaped as it is printed.
dim q as net.QueryValues
```

```
q.Add( "name", "Zoë Wu" )
q.Add( "tags", "a&b" )
dim v as net.Url
v.Scheme = "http"
v.Host = "localhost"
v.Port = 8080
v.Path = "/search results"
v.RawQuery = q.Encode( )
print v.ToString( )

'' A bad escape is an error, not a guess.
print net.ParseUrl( "http://example.com/100%" ).Failure( ).Message( )
```

examples/ch25/urls.out

```
scheme   https
user     ann
host     example.com  port  8443
path     /docs/a b.html
query    q=tiko+lang&page=2
q        tiko lang
fragment intro
https://ann@example.com:8443/img/logo.png
https://ann@example.com:8443/docs/a%20b.html?page=3
http://localhost:8080/search%20results?name=Zo%C3%AB+Wu&tags=a%26b
ParseUrl 'http://example.com/100%': invalid URL
```

`ParseUrl` follows RFC 3986. The fields of a `Url` hold the parts **decoded** — `Path` is `/docs/a b.html`, not `a%20b` — except `RawQuery`, kept as written because only its own parser knows where its pairs begin and end; `Query( )` parses it into `QueryValues`, an ordered list of pairs in which a name may repeat. `ToString( )` escapes each part again as it prints it, so a field can be set to any text. `Resolve` turns a relative link into an absolute one the way a browser does, removing `.` and `...` An invalid escape such as a lone `%` is `NET_INVALID_URL` rather than a guess at what was meant.

Fetching a web page

examples/ch25/httpget.tiko

```
#cmdline "-mt"
#import "std/http"
#import "std/net"
#import "std/fs"

'' A server on this machine that answers each request on one connection
```

```
with
'' the request line it received, until the client goes away.
function ReadHead( byref c as net.TcpStream, byref head as string ) as
boolean
    dim as ubyte ch
    head = ""
    while right( head, 4 ) <> !"\r\n\r\n"
        dim got as fs.FsResultI64 = c.Read( @ch, 1 )
        if got.IsOk( ) = false orelse got.Value( ) = 0 then return false
        head &= chr( ch )
    wend
    return true
end function

sub Serve( byval p as any ptr )
    dim as net.TcpListener ptr lis = p
    dim c as net.TcpStream
    if lis->Accept( c, 5000 ).IsOk( ) = false then exit sub
    dim as string head
    while ReadHead( c, head )
        dim as long at_ = instr( lcase( head ), "content-length:" )
        if at_ > 0 then
            dim as long n = valint( mid( head, at_ + 15 ) )
            dim as string body = space( n )
            c.ReadExact( cptr( ubyte ptr, strptr( body ) ), n )
        end if
              dim as string reply = "you sent: " & left( head,
instr( head, !"\r\n" ) - 1 )
       c.WriteUtf8( !"HTTP/1.1 200 OK\r\nContent-Length: " & len( reply )
& !"\r\n\r\n" & reply )
    wend
end sub

sub Main( )
    dim lis as net.TcpListener
    lis.Bind( net.SocketAddress( net.Loopback( ), 0 ) )
             dim as ustring site = "http://127.0.0.1:" &
lis.LocalAddress( ).Value( ).Port
    dim th as any ptr = threadcreate( @Serve, @lis, 1024 * 1024 )

    dim c as http.Client
    c.TimeoutMs = 5000
   dim r as http.HttpResultResponse = c.Fetch( site & "/notes?day=mon" )
    print r.Value( ).StatusCode; "  "; r.Value( ).Text( )
```

```
    r = c.Post( site & "/notes", "text/plain", "buy milk" )
    print r.Value( ).StatusCode; "  "; r.Value( ).Text( )
    print "idle connections kept:"; c.IdleCount( )

    c.CloseIdle( )                      '' the server sees the client go
    threadwait( th )
end sub

Main( )
```

examples/ch25/httpget.out

```
 200  you sent: GET /notes?day=mon HTTP/1.1
 200  you sent: POST /notes HTTP/1.1
idle connections kept: 1
```

`std/http` is an HTTP/1.1 client over those sockets, and `std/tls` encrypts the connection when the URL says `https`. The listing serves itself, on a second thread, so it runs without a network — the `Serve` routine is all a server needs to say to an HTTP client: a status line, headers, a blank line and a body of the length the headers announced.

`http.Fetch( url )` makes one request with the defaults. An `http.Client` holds settings — `TimeoutMs`, `MaxRedirects`, `UserAgent`, `DefaultHeader`, `TlsOptions` — and **keeps connections open**: both requests above travelled over one connection, which is still kept for the next request to the same site. `Send` takes a `Request` built by hand, with any method, headers and body; `Fetch`, `Head` and `Post` are shorthands for it. A `Response` holds `StatusCode`, `Header`, the `Body` bytes and `Text( )`. For a body too large to hold, `Open` returns a `ResponseStream`, an `fs.Stream` like a `File` or a `TcpStream`.

A status is not an error. A 404 is a request that worked and an answer the program has to look at, so it comes back as a `Response`. An `HttpError` means there was no answer: `HTTP_CONNECT_FAILED`, `HTTP_TIMED_OUT`, `HTTP_BAD_RESPONSE`, or `HTTP_TLS_FAILED` when the server's certificate is not trusted, out of date, or for another name — the error's `TlsCode( )` says which. Redirects are followed for you, up to ten.

What it costs

- **A `File` holds an operating-system handle until it is closed or goes out of scope.** Keep it in the narrowest `scope` that needs it. Call `Close( )` yourself when you need to know whether the close itself succeeded: it returns a `Result`, and a destructor has nowhere to report one.

- **A `File` is unbuffered**: every `Read` and `Write` is a system call. Many small writes belong in a `BufferedWriter`, a `TextWriter` over one, or a `MemoryStream` written once.
- **`Stat` is one system call**; `Exists` followed by a size check is two, and the file can change between them.
- **`WalkDir` holds one directory open at a time** and cannot exhaust handles, however deep the tree. `ReadAllText` and `ReadAllLines` hold the whole file in memory.
- **`WriteAtomic` writes the file twice over** — a temporary and a rename — which is the price of never leaving it half-written.
- **A `TomlValue` is a value**: `Get( key )` and copying a table copy the whole subtree under it. `Find( key )` and `[ i ]` reach into the tree without a copy. Looking a key up scans its table, which is what a configuration file's tables can afford.
- **An `http.Client` keeps up to eight idle connections open** for reuse, each an operating-system handle (and, for `https`, a TLS session). `CloseIdle( )` lets them go; so does the client going out of scope.
- **A socket holds an operating-system handle and, while a call waits, a thread.** Close connections in the narrowest scope that needs them, and give every network read a timeout: without one a peer that stops answering stops the program with it.

Summary

- `#import "std/fs"`. A `File` opens with `OpenRead`, `Create`, `CreateNew`, `OpenAppend` or `Open( p, OpenOptions )`, closes itself, and reports every failure as an `FsError`.
- `TextWriter.WriteLine` writes lines; `for each ln in ReadLines( p )` reads them back. `strings.SplitCsvRow` splits a line of comma-separated fields.
- Every text call takes an encoding: `Utf8( )`, `Utf16LE( )`, `Utf16BE( )`, `CodePage( n )`, `Ansi( )`, `BinaryMode( )`. Reads believe it; `TextDetect` guesses; writes refuse what the encoding cannot hold unless given `TEXT_REPLACE`.
- `RecordFile( of T )` reads and writes fixed-size records by number, from 1, with `GetRecord`, `PutRecord`, `Append` and `for each`; every field must be fixed-size. `ReadAt` and `WriteAt` are the byte-level calls underneath.
- `std/codec` lays out binary formats in a byte `Array`: `Get`/`Put`/`Append` + width + `LE`/`BE`, and LEB128 varints with `AppendUVarint`/`ReadUVarint`.
- A `File` is a `Stream`, like `MemoryStream`; buffers, text readers and writers, and `Copy` work on any stream, including one you write by extending `fs.Stream`.
- The `Path*` functions; `Stat` for every fact at once; `ReadDir`, `WalkDir`, `Glob`; `ReadAllText`/`ReadAllLines`/`WriteAllText`; `WriteAtomic`.

- `ConfigDir`, `DataDir`, `CacheDir`, `StateDir`, `HomeDir` and `ExecutablePath` say where a program keeps its files.
- **`PathJoin` with an absolute second part discards the first** — check containment with `PathStartsWith` before using a path built from input.
- The file statements (`open`, `print #`, `kill`, `mkdir`, `dir` …) still compile for older programs; each has a `std/fs` counterpart that reports failure as a value.
- `IsRedirected( )` decides whether to draw on the console. `inkey` never waits; `getkey` does.
- `command( n )`, `environ`, `setenviron`; `shell`, `exec`, `chain`, `run`, `open pipe`; `end n` sets the exit code.
- `std/os`: `Args( )` keeps empty arguments, `GetEnv` tells unset from empty, `SetEnv` / `UnsetEnv` / `ExpandEnv`; host facts such as `CpuCount( )`; `CatchInterrupt( )` turns Ctrl-C into a flag the program checks with `Interrupted( )`.
- `std/hash`: `DigestHex( alg, data )`, the same in pieces, `DigestFile`, `HmacHex`. CRC and FNV catch accidents; only SHA-2 stands up to tampering.
- `std/config`: `ParseToml` / `LoadToml` give a `TomlValue` tree read with `Path` and `As…`, each an `Optional`; `ToTomlText` writes it back, without comments. `ParseIni` / `LoadIni` give an `IniFile` with `Get`, `SetValue` and `ToText`. Errors are `ConfigError`s with a line and column.
- `strings.JoinCsvRow` and `strings.CsvWriter` write CSV that `SplitCsv` reads back.
- `std/net`: `ParseIp` (strict) and `IpAddress`; `SocketAddress`; `Resolve`. `TcpListener.Bind` and `Accept`, `TcpStream.Connect` / `ConnectHost` — a `TcpStream` is an `fs.Stream`. `UdpSocket` for datagrams. Every call blocks; give it a timeout. `ParseUrl` gives a `Url` of decoded fields, `Resolve` makes links absolute, `QueryValues` reads and writes query strings.
- `std/http`: `http.Fetch`, or an `http.Client` that reuses connections and follows redirects; `Send` a `Request`, read a `Response` — a 404 is a response, not an error. `std/tls` encrypts `https`, and checks the server's certificate against the system's trusted roots.

Exercises

1. Walk a directory tree and count its files by extension, printing the counts in a stable order. {solution="ex25-01″}
2. Write `Sell( stock, record, n )` for the stock file of `records.tiko`: it reduces the quantity of record `record` by `n` and refuses, returning `false`, if there are not enough. {solution="ex25-02″}
3. Make `pathescape.tiko` refuse any name that would leave `uploads`, and print which names were refused, on both platforms.
4. Rewrite `textfile.tiko` with `fs.WriteAllText` and `fs.ReadAllLines`. Which version would you want for a file of a few lines, and which for a log of a few gigabytes?

5. Write a program that runs `exitcode.tiko`'s executable with `exec` and prints a different message for exit status 0 and for anything else.
6. Print the SHA-256 digest of every file under a directory, sorted by path, in the format `sha256sum` uses: the digest, two spaces, the path.
7. Give `interrupt.tiko` a temporary directory that it creates at the start and removes at the end, and check that it is removed when you press Ctrl-C too.
8. Write `binformat.tiko`'s buffer to a file with `fs.WriteAllBytes`, read it back with `fs.ReadAllBytes`, and check the magic number before reading anything else. What should a reader do with a file whose version it does not know?
9. Give `tomlconfig.tiko` a `[server]` key `port = "8080"`, written as a string. What does it print, and how would you report the mistake to the user who wrote the file?
10. Read a TOML file of `[[items]]` tables, each with a `name` and a `price`, and write them to a CSV file with `CsvWriter`, a header row first.
11. Change `tcpecho.tiko` so that the client sends three lines and the server answers each with its length, reading with a `TextReader` on the server's side. What happens if the client sends all three before the server reads any?
12. Write `LinkReport( pageUrl, links )`: resolve each link of a page against its URL and print the ones that leave the page's host.
13. Make `httpget.tiko`'s server answer `/old` with `302 Found` and a `Location: /notes` header, and fetch `/old`. What does `r.Value( ).Url` hold afterwards? Set `c.MaxRedirects = 0` and fetch it again.

Dates and Times

A date looks like the simplest data a program handles, and is not. Months have four different lengths. The same moment has a different wall-clock time in every city, and the difference changes twice a year in many of them. “Tomorrow at this time” and “24 hours from now” are the same thing on most days and not on two. A date written `03/04/2025` means March in one country and April in the next.

tiko’s answer is the `std/chrono` module. It gives each of those ideas its own type, so a program says which one it means, and it refuses ambiguous input instead of guessing. This chapter covers the types, what makes a value valid, reading and writing dates as text, calendar arithmetic, time zones, the clocks a program can read, and the older `datetime.bi` functions that chrono sits beside.

The types and the tick

examples/ch26/ticks.tiko

```
#import "std/chrono"

'' Five value types, one clock tick: 100 nanoseconds.
var day  = chrono.LocalDate( 2025, 3, 4 )
var tod  = chrono.LocalTime( 14, 30, 5, 250 )
var when = chrono.DateTime( 2025, 3, 4, 14, 30, 5, 250 )
var at   = when.WithOffset( 60 ).ToInstant( )
var span = chrono.TimeSpan( 1, 2, 3, 4, 500 )

print "LocalDate  "; day.ToIsoString( )
print "LocalTime  "; tod.ToIsoString( )
print "DateTime   "; when.ToIsoString( )
print "Instant    "; at.ToIsoString( )
print "TimeSpan   "; span.ToIsoString( )

'' The parts are read with methods; 1 = Monday.
print "year"; day.Year; ", month"; day.Month; ", day"; _
      day.Day; ", weekday"; day.DayOfWeek
print "one second ="; chrono.TimeSpan.FromSeconds( 1 ).Ticks; _
      " ticks"
print "first: "; chrono.DateTime.MinValue( ).ToIsoString( )
print "last:  "; chrono.DateTime.MaxValue( ).ToIsoString( )
```

examples/ch26/ticks.out

```
LocalDate  2025-03-04
LocalTime  14:30:05.2500000
```

```
DateTime   2025-03-04T14:30:05.2500000
Instant    2025-03-04T13:30:05.2500000Z
TimeSpan   P1DT2H3M4.5S
year 2025, month 3, day 4, weekday 2
one second = 10000000 ticks
first: 0001-01-01T00:00:00
last:  9999-12-31T23:59:59.9999999
```

`#import "std/chrono"` brings in five value types, reached as `chrono.LocalDate` and so on:

Type	Holds	Example
`LocalDate`	a calendar date	`2025-03-04`
`LocalTime`	a time of day	`14:30:05.2500000`
`DateTime`	a date and a time, with or without a UTC offset	`2025-03-04T14:30:05+01:00`
`Instant`	a point on the UTC timeline	`2025-03-04T13:30:05Z`
`TimeSpan`	a fixed length of time	`P1DT2H3M4.5S`

Every one of them counts in **ticks of 100 nanoseconds**, so a second is ten million ticks, and the range runs from the first moment of year 1 to the last tick of year 9999. The resolution is the same at both ends of that range; a value does not lose precision by being far from some epoch.

The values are **immutable**. No method changes a date in place: `day.AddDays( 1 )` returns a new date and leaves `day` alone, so a value can be passed around and stored without anyone changing it underneath. The parts are read with properties — `Year`, `Month`, `Day`, `Hour` and so on. `DayOfWeek` numbers the days the ISO 8601 way, **1 for Monday** through 7 for Sunday.

The constructors take their fields from largest to smallest: `LocalDate( y, mo, d )`, `LocalTime( h, mi, s, ms )`, `DateTime( y, mo, d )` or `DateTime( y, mo, d, h, mi, s, ms )`, and for a span, `TimeSpan( h, mi, s )` or `TimeSpan( d, h, mi, s, ms )`. `TimeSpan` also has `FromDays`, `FromHours`, `FromMinutes`, `FromSeconds` and `FromMilliseconds`.

A call whose argument count no constructor takes is refused: `chrono.TimeSpan( 0, 1, 30, 0 )` — four arguments, where the constructors take three or five — is `TKO0632: no overload of 'timespan.constructor' matches these arguments`.

Validity

`examples/ch26/validity.tiko`

```
#import "std/chrono"

'' Out-of-range parts are refused, never rolled into March.
var feb30 = chrono.LocalDate( 2025, 2, 30 )
print "2025-02-30 valid: "; feb30.IsValid
print "as text: ["; feb30.ToIsoString( ); "]"

'' A default date is Invalid; a default span is a real zero.
dim d as chrono.LocalDate
dim s as chrono.TimeSpan
print "default date valid: "; d.IsValid
print "default span zero:  "; s.IsZero

'' Invalid absorbs arithmetic, and equals nothing, not even itself.
print "Invalid + 10 days valid: "; feb30.AddDays( 10 ).IsValid
print "Invalid = Invalid: "; ( feb30 = feb30 )
print "CompareTo:"; feb30.CompareTo( chrono.LocalDate( 2025, 3, 1 ) )

'' The ends of the range do not wrap around.
var last = chrono.DateTime.MaxValue( )
print "MaxValue + 1 tick valid: "; last.AddTicks( 1 ).IsValid
```

examples/ch26/validity.out

```
2025-02-30 valid: false
as text: []
default date valid: false
default span zero:  true
Invalid + 10 days valid: false
Invalid = Invalid: false
CompareTo:-2
MaxValue + 1 tick valid: false
```

There is no 30 February, and chrono does not pretend there is. A constructor given an out-of-range part does not roll the extra days into March; it returns the **Invalid** value of its type, and `IsValid` says so. Rolling over would make a typo and a real date look the same one line later.

Invalid behaves like a NaN in floating point:

- **It absorbs arithmetic.** Adding days, months or a span to an Invalid value gives Invalid, so a chain of calculations can be checked once at the end.
- **It equals nothing, not even itself.** `feb30 = feb30` is false. Test with `IsValid`, never by comparing with `Invalid( )`.

- **`CompareTo` returns −2** when either side is Invalid, which is not an ordering: −1, 0 and 1 are.
- **It prints as an empty string.** `ToIsoString( )` never fails; for Invalid it has nothing to say.

Arithmetic **never wraps**. Stepping one tick past `MaxValue( )` gives Invalid, not year 1.

A `LocalDate`, `Instant` or `DateTime` declared with no initialiser is Invalid, because no date is a sensible default. A `TimeSpan` and a `LocalTime` start at zero and at midnight, which are real values.

Offsets and instants

examples/ch26/offsets.tiko

```
#import "std/chrono"

'' No offset: a reading on some clock, nobody has said which.
var naive = chrono.DateTime( 2025, 3, 4, 9, 0, 0, 0 )
print "naive      "; naive.ToIsoString( ); _
      "  has offset: "; naive.HasOffset
print "ToInstant valid: "; naive.ToInstant( ).IsValid
print "AssumeUtc  "; naive.AssumeUtc( ).ToIsoString( )

'' WithOffset says which clock; ToOffset moves to another one.
var toronto = naive.WithOffset( -300 )
print "Toronto    "; toronto.ToIsoString( )
print "in UTC     "; toronto.ToOffset( 0 ).ToIsoString( )
print "in Kolkata "; toronto.ToOffset( 330 ).ToIsoString( )
print "relabelled "; toronto.WithOffset( 330 ).ToIsoString( )

'' The same moment, read on two clocks, is one Instant.
var a = toronto.ToInstant( )
var b = toronto.ToOffset( 330 ).ToInstant( )
print "same instant: "; ( a = b )
```

examples/ch26/offsets.out

```
naive      2025-03-04T09:00:00  has offset: false
ToInstant valid: false
AssumeUtc  2025-03-04T09:00:00Z
Toronto    2025-03-04T09:00:00-05:00
in UTC     2025-03-04T14:00:00Z
in Kolkata 2025-03-04T19:30:00+05:30
relabelled 2025-03-04T09:00:00+05:30
same instant: true
```

A `DateTime` is a reading of a wall clock, and it is in one of three states:

- **UTC**, offset 0, printed with a `Z`.
- **An offset** from UTC, in minutes, up to 18 hours either way: `-300` is five hours behind UTC, printed `-05:00`.
- **Unspecified**: the time on *some* clock, with nothing to say which. `HasOffset` is false and the text has no suffix.

An `Instant` is a single point in time, the same everywhere. An unspecified `DateTime` cannot become one without a guess, so **`ToInstant( ) on it is Invalid`**. `AssumeUtc( )` makes the guess, and makes it visible in the code.

Two methods put an offset on a `DateTime`, and they do different things:

Call	Keeps	Changes
`WithOffset( m )`	the wall-clock reading	the moment it refers to
`ToOffset( m )`	the moment	the wall-clock reading

`WithOffset` says "this 09:00 was read in Toronto". `ToOffset` says "what did the clocks in Kolkata read at that moment?" The last line of the listing shows the difference: relabelling Toronto's 09:00 as Kolkata's names a moment ten and a half hours earlier.

To compare readings taken in different places, convert them to `Instant` first. Two `DateTime` values for the same moment in different offsets are different readings; their instants are equal.

> **PITFALL**
> **`WithOffset` where `ToOffset` was meant** compiles, runs, and moves every timestamp by the difference between the two offsets. If the value already has an offset and the goal is to show it somewhere else, the call is `ToOffset`.

Dates as text

ISO 8601

examples/ch26/isotext.tiko

```
#import "std/chrono"

dim inputs( ... ) as ustring = { _
    "2025-03-04T14:30:05Z", "2025-03-04 14:30:05+05:30", _
    "20250304T143005Z", "2025-W10-2", "2025-063", _
    "2025-03-04T14:30:05.5-05:00", _
    "03/04/2025", "25-03-04", "2025-03-04T24:00:00" }

'' One canonical form out, several ISO 8601 forms in.
```

```
for i as int32 = 0 to ubound( inputs )
    dim d as chrono.DateTime
    dim shown as ustring = "refused"
    if chrono.DateTime.TryParseIso( inputs( i ), d ) then
        shown = d.ToIsoString( )
    end if
    print left( inputs( i ) & space( 28 ), 28 ); shown
next

'' Durations: weeks are accepted, months are not.
dim spans( ... ) as ustring = { "PT90M", "P1W", "P1DT0.25S", "P1M" }
for i as int32 = 0 to ubound( spans )
    dim span as chrono.TimeSpan
    dim shown as ustring = "refused"
    if chrono.TimeSpan.TryParseIso( spans( i ), span ) then
        shown = span.ToIsoString( )
    end if
    print left( spans( i ) & space( 12 ), 12 ); shown
next
```

examples/ch26/isotext.out

```
2025-03-04T14:30:05Z        2025-03-04T14:30:05Z
2025-03-04 14:30:05+05:30   2025-03-04T14:30:05+05:30
20250304T143005Z            2025-03-04T14:30:05Z
2025-W10-2                  2025-03-04T00:00:00
2025-063                    2025-03-04T00:00:00
2025-03-04T14:30:05.5-05:00 2025-03-04T14:30:05.5000000-05:00
03/04/2025                  refused
25-03-04                    refused
2025-03-04T24:00:00         refused
PT90M       PT1H30M
P1W         P7D
P1DT0.25S   P1DT0.25S
P1M         refused
```

`ToIsoString( )` writes the ISO 8601 form, and **one value has exactly one string**: a capital `T`, a four-digit year, `Z` for UTC, `±HH:MM` for an offset, nothing for unspecified. A fraction of a second is written with all seven digits or not at all, so the strings are all the same width and sort in time order as plain text — the property that matters in a log file or a database key.

Parsing accepts more than it writes. `TryParseIso` takes the basic format without separators (`20250304T143005Z`), ISO week dates (`2025-W10-2`, the Tuesday of week 10), ordinal dates (`2025-063`, the 63rd day), a space in place of the `T` as RFC 3339

and SQL databases write it, and a comma as the decimal point. It refuses anything else: there is no guessing at `03/04/2025`, no two-digit years, and no `24:00`, which ISO allows but which means the same as the next day's `00:00`.

Every type has the same pair:

- **`TryParseIso( text, result )`** returns `true` and sets `result`, or returns `false` and sets `result` to Invalid. The result is always written, so a stale value is never left behind. It never raises a runtime error.
- **`ParseIso( text )`** returns the value, or Invalid. Use it for text known to be good, such as a literal in the program.

A `TimeSpan` reads and writes ISO 8601 durations: `P1DT2H3M4.5S` is one day, two hours, three minutes and four and a half seconds. `P1W` is accepted as seven days. `P1M` is refused, because a month is not a fixed length of time — the same reason there is no `TimeSpan.FromMonths`.

`DateTime.TryParse` is `TryParseIso` under another name; it is not a lenient parser.

Custom patterns

examples/ch26/patterns.tiko

```
#import "std/chrono"

var d = chrono.DateTime( 2025, 3, 4, 14, 30, 5, 0 ).WithOffset( 60 )

'' Custom patterns: invariant English, never the machine's locale.
print d.ToString( "dddd, d MMMM yyyy" )
print d.ToString( "dd/MM/yy h:mm tt" )
print d.ToString( "yyyy-MM-dd'T'HH:mm:ssK" )
print d.ToString( "R" )

'' An unknown letter is an error, not a literal: quote the T.
dim text as ustring
print "TryFormat with a bare T: "; _
      d.TryFormat( "yyyy-MM-ddTHH:mm", text )

'' Parsing with a pattern says which number is the day.
dim p as chrono.DateTime
if chrono.DateTime.TryParseExact( "04.03.2025 17:45", _
                                  "dd.MM.yyyy HH:mm", p ) then
    print "parsed: "; p.ToIsoString( )
end if

'' Redundant fields are cross-checked: 4 March 2025 is a Tuesday.
```

```
print "Monday 04.03.2025: "; chrono.DateTime.TryParseExact( _
      "Monday 04.03.2025", "dddd dd.MM.yyyy", p )
```

examples/ch26/patterns.out

```
Tuesday, 4 March 2025
04/03/25 2:30 PM
2025-03-04T14:30:05+01:00
Tue, 04 Mar 2025 13:30:05 GMT
TryFormat with a bare T: false
parsed: 2025-03-04T17:45:00
Monday 04.03.2025: false
```

`ToString( pattern )` builds the text from pattern letters:

Letters	Field	For 2025-03-04 14:30:05 +01:00
`yyyy`, `yy`	year	`2025`, `25`
`M`, `MM`, `MMM`, `MMMM`	month	`3`, `03`, `Mar`, `March`
`d`, `dd`, `ddd`, `dddd`	day, weekday name	`4`, `04`, `Tue`, `Tuesday`
`H`, `HH`, `h`, `hh`	hour, 24- and 12-hour	`14`, `14`, `2`, `02`
`mm`, `ss`	minute, second	`30`, `05`
`fff`, `FFF`	fraction; `F` drops trailing zeros	
`tt`	AM or PM	`PM`
`zzz`, `K`	offset; `K` writes `Z` for UTC and nothing when unspecified	`+01:00`

Text in single quotes is copied as it is, and so is any character that is not a letter. The names are **always English**, whatever the machine's language, so the output depends only on the value and the pattern — what a file or a protocol needs. A pattern of one character is a named format: `"o"` round-trip ISO, `"s"` sortable, `"u"` universal, `"R"` RFC 1123 for HTTP headers (converted to GMT first, as the listing shows), `"d"` the date, `"T"` the time.

A letter that is not a pattern letter is an error, not a literal. The `T` of an ISO timestamp has to be quoted, `'T'`; written bare, `ToString` returns an empty string and `TryFormat` returns `false`. A typo in a pattern shows up at once instead of in the output a week later.

`TryParseExact( text, pattern, result )` reads with a pattern, so a program says whether `04.03.2025` is the fourth of March or the third of April. Fields that appear

twice are checked against each other: “Monday” does not parse beside a date that fell on a Tuesday.

For text a person will read on their own machine, `ToLocaleDateString`, `ToLocaleTimeString` and `ToLocaleString` ask the operating system, in the user’s language and order. Their output differs from machine to machine, so never write it to a file or compare it.

Every chrono member that produces text without taking any — `ToIsoString`, `ToString( )`, the locale family, `MonthName`, `WeekdayName` and the zone names — returns a `ustring`. A Spanish “miércoles” therefore prints intact whatever the console’s code page, and assigning the result to a `string` still works, as UTF-8.

Calendar arithmetic

examples/ch26/calendar.tiko

```
#import "std/chrono"

'' Months have no fixed length, so month arithmetic clamps.
var jan31 = chrono.LocalDate( 2025, 1, 31 )
print "Jan 31 + 1 month:      "; jan31.AddMonths( 1 ).ToIsoString( )
print "  and back 1 month:    "; _
      jan31.AddMonths( 1 ).AddMonths( -1 ).ToIsoString( )
print "Jan 31 + 2 months:     "; jan31.AddMonths( 2 ).ToIsoString( )
print "  as two steps of 1:   "; _
      jan31.AddMonths( 1 ).AddMonths( 1 ).ToIsoString( )
print "2024-02-29 + 1 year:   "; _
      chrono.LocalDate( 2024, 2, 29 ).AddYears( 1 ).ToIsoString( )

'' Periods: start and end. End is the LAST tick, not the next day.
var t = chrono.DateTime( 2025, 3, 4, 14, 30, 0, 0 )
print "StartOfWeek  "; t.StartOfWeek( ).ToIsoString( )
print "EndOfDay     "; t.EndOfDay( ).ToIsoString( )
print "StartOfMonth "; t.StartOfMonth( ).ToIsoString( )
print "EndOfYear    "; t.EndOfYear( ).ToIsoString( )

'' NextWeekday is strictly later: a Tuesday's next Tuesday.
print "next Tuesday "; t.NextWeekday( 2 ).ToIsoString( )

'' The ISO week belongs to the ISO week-year, not to Year.
var dec29 = chrono.LocalDate( 2025, 12, 29 )
print "2025-12-29: Year"; dec29.Year; ", week"; _
      dec29.IsoWeek; " of"; dec29.IsoWeekYear
```

examples/ch26/calendar.out

```
Jan 31 + 1 month:      2025-02-28
  and back 1 month:    2025-01-28
Jan 31 + 2 months:     2025-03-31
  as two steps of 1:   2025-03-28
2024-02-29 + 1 year:   2025-02-28
StartOfWeek  2025-03-03T00:00:00
EndOfDay     2025-03-04T23:59:59.9999999
StartOfMonth 2025-03-01T00:00:00
EndOfYear    2025-12-31T23:59:59.9999999
next Tuesday 2025-03-11T14:30:00
2025-12-29: Year 2025, week 1 of 2026
```

Adding time comes in two kinds, and the names say which:

- **Exact**: `Add( span )`, `AddDays`, `AddHours`, `AddMinutes`, `AddSeconds`. `AddDays( 1 )` is exactly 86 400 seconds later.
- **Calendar**: `AddMonths` and `AddYears`, and on `DateTime` also `AddCalendarDays`, `AddCalendarMonths` and `AddCalendarYears`. These work on the calendar's fields and then build a date from them.

A month has no fixed length, so **month arithmetic clamps at the end of the month**: 31 January plus one month is 28 February. That is the only sensible answer, but it loses information. Going back a month from there gives 28 January, not 31; and two steps of one month end on 28 March, where one step of two months ends on 31 March. Add the whole amount in one call rather than building it up a step at a time. `AddYears` is `AddMonths` of twelve, so 29 February 2024 plus a year is 28 February 2025.

The difference between `AddDays` and `AddCalendarDays` shows only across a daylight-saving change. `AddDays( 1 )` is 24 hours later, and the wall-clock time moves by an hour. `AddCalendarDays( 1 )` keeps the wall-clock time and works out the new offset, so it is 23 or 25 hours later. "24 hours from now" is `AddDays`; "the same time tomorrow" is `AddCalendarDays`.

The period helpers keep the offset and set the time:

- `StartOfDay`, `StartOfWeek`, `StartOfMonth`, `StartOfYear` are the first tick of the period. `StartOfWeek` takes the first day of the week, and defaults to Monday.
- `EndOfDay`, `EndOfMonth`, `EndOfYear` are the **last tick**, `23:59:59.9999999`, not the start of the next period. Compare against them with `<=`.
- `NextWeekday( n )` and `PrevWeekday( n )` are strictly later and earlier: the next Tuesday after a Tuesday is a week away.

PITFALL

The ISO week belongs to the ISO week-year, not to `Year`. 29 December 2025 is in week 1 of 2026. A program that prints `Year` beside `IsoWeek` writes "week 1 of 2025" for it, a date a year off. Print `IsoWeekYear` with `IsoWeek`, always.

Spans and differences

examples/ch26/spans.tiko

```
#import "std/chrono"

'' Totals are the whole span; components are the pieces.
var span = chrono.TimeSpan( 1, 2, 3, 4, 500 )
print span.ToIsoString( )
print "TotalHours"; span.TotalHours; ", Hours"; span.Hours

'' A negative span has negative pieces, which add back up.
var back = chrono.TimeSpan( 1, 30, 0 ).Negate( )
print back.ToIsoString( ); ": hours "; back.Hours; _
      ", minutes "; back.Minutes

'' Subtracting two moments gives a span ...
var dep = chrono.Instant.ParseIso( "2025-03-04T09:00:00-05:00" )
var arr = chrono.Instant.ParseIso( "2025-03-04T21:40:00+00:00" )
var flight = arr - dep
print "flight "; flight.ToIsoString( ); " ="; _
      flight.TotalMinutes; " minutes"

'' ... and subtracting two dates gives a count of days.
var today = chrono.LocalDate( 2025, 3, 4 )
var xmas = chrono.LocalDate( 2025, 12, 25 )
print "days to Christmas:"; xmas - today
```

examples/ch26/spans.out

```
P1DT2H3M4.5S
TotalHours 26.05125, Hours 2
-PT1H30M: hours -1, minutes -30
flight PT7H40M = 460 minutes
days to Christmas: 296
```

A `TimeSpan` has two families of readers. **`TotalHours`** and its siblings give the whole span in one unit, with a fraction: 26.05 hours. **`Hours`** and its siblings give the pieces: the 2 in "1 day, 2 hours". Mixing them up gives numbers that look plausible and are wrong. The pieces of a negative span are all negative, so they always add back up to the whole.

Subtracting gives a difference of the right kind:

Expression	Gives
`instant - instant`, `datetime - datetime`	`TimeSpan`
`localtime - localtime`	`TimeSpan`
`localdate - localdate`	a count of days, as a number
`instant + span`, `datetime - span`	the same type as the left side

The flight in the listing leaves at 09:00, five hours behind UTC, and lands at 21:40 UTC. Both are converted to instants when parsed, so the subtraction is correct without any offset arithmetic in the program.

A worked example: ages and birthdays

`examples/ch26/age-calc.tiko`

```
#import "std/chrono"

'' Whole years from 'born' to 'today': the birthday may not have
'' come round yet this year.
function AgeOn( byval born as chrono.LocalDate, _
                byval today as chrono.LocalDate ) as int32
    dim as int32 years = today.Year - born.Year
    if born.AddYears( years ) > today then years -= 1
    return years
end function

'' The next birthday on or after 'today'.
function NextBirthday( byval born as chrono.LocalDate, _
                       byval today as chrono.LocalDate ) _
                       as chrono.LocalDate
    dim as int32 years = today.Year - born.Year
    if born.AddYears( years ) < today then years += 1
    return born.AddYears( years )
end function

'' A fixed 'today' keeps the output reproducible; a real program
'' would use chrono.WallClock.Today( ).
var today = chrono.LocalDate( 2025, 3, 4 )

dim births( ... ) as ustring = { _
    "1990-03-04", "1990-03-05", "2004-02-29", "1969-07-20" }
for i as int32 = 0 to ubound( births )
    var born = chrono.LocalDate.ParseIso( births( i ) )
    var nb = NextBirthday( born, today )
```

```
    print born.ToIsoString( ); "  age"; AgeOn( born, today ); _
          "  next birthday "; nb.ToIsoString( ); _
          ", days away:"; nb - today
next
```

examples/ch26/age-calc.out

```
1990-03-04  age 35  next birthday 2025-03-04, days away: 0
1990-03-05  age 34  next birthday 2025-03-05, days away: 1
2004-02-29  age 21  next birthday 2026-02-28, days away: 361
1969-07-20  age 55  next birthday 2025-07-20, days away: 138
```

An age is the difference in years, less one if this year's birthday has not come yet. `AddYears` does the hard part: it lands on the birthday in the target year, and for someone born on 29 February it clamps to the 28th in a year with no 29th. Whether a leap-day birthday falls on 28 February or 1 March in other years is a rule that varies by country; this code chooses the 28th because that is what `AddYears` gives, and a program that must follow a particular law should write that rule out.

The fixed `today` keeps the output the same on every run. A real program would use `chrono.WallClock.Today( )`.

Time zones

examples/ch26/zone.tiko

```
#import "std/chrono"

'' The offset depends on the moment: ask for it AT that instant.
var jan = chrono.Instant.ParseIso( "2025-01-15T12:00:00Z" )
var jul = chrono.Instant.ParseIso( "2025-07-15T12:00:00Z" )
dim as int32 offJan = chrono.TimeZoneInfo.LocalOffsetAt( jan )
dim as int32 offJul = chrono.TimeZoneInfo.LocalOffsetAt( jul )

'' What this machine says varies; these facts do not.
print "January offset in range: "; cbool( abs( offJan ) <= 18 * 60 )
print "July offset in range:    "; cbool( abs( offJul ) <= 18 * 60 )

'' Local wall time for a moment, and back to the same moment.
var localJul = chrono.DateTime.FromInstant( jul, offJul )
print "round trip: "; ( localJul.ToInstant( ) = jul )

'' Now, as a DateTime, carries the offset it was taken under.
var now = chrono.WallClock.Now( )
print "Now has an offset: "; now.HasOffset
```

```
dim as int32 nowOff = _
    chrono.TimeZoneInfo.LocalOffsetAt( now.ToInstant( ) )
print "Now matches the zone: "; cbool( now.OffsetMinutes = nowOff )
```

examples/ch26/zone.out

```
January offset in range: true
July offset in range:    true
round trip: true
Now has an offset: true
Now matches the zone: true
```

Time zones come from the **operating system**. `TimeZoneInfo.LocalOffsetAt( instant )` returns the machine's offset from UTC, in minutes, **at that instant** — January and July can differ. `TimeZoneInfo.LocalOffset( )` is the offset right now, which is the wrong one for converting a timestamp from another season and the most common time-zone bug there is. `IsDaylightSavingTime( instant )`, `SupportsDaylightSavingTime( )`, `StandardName( )` and `DaylightName( )` complete the set. `WallClock.Now( )` records the offset it was read under, so it converts back to the right instant even across a change.

There is no database of named zones. chrono cannot answer "what time is it in Tokyo?" from the name alone, and has no rule for a wall time that occurs twice when the clocks go back. A program that must know an exact point in time should carry an `Instant`, store and send it in UTC, and convert to local time only for display.

The output of this listing is the same everywhere because it prints only facts that hold in every zone; the offsets themselves depend on the machine.

Clocks

examples/ch26/benchmark.tiko

```
#import "std/chrono"

'' Two ways to look values up in a sorted array.
function FindLinear( a( ) as int32, byval v as int32 ) as boolean
    for i as int32 = 0 to ubound( a )
        if a( i ) = v then return true
    next
    return false
end function

function FindBinary( a( ) as int32, byval v as int32 ) as boolean
```

```
    dim as int32 lo = 0, hi = ubound( a )
    do while lo <= hi
        dim as int32 mid_ = ( lo + hi ) \ 2
        if a( mid_ ) = v then return true
        if a( mid_ ) < v then lo = mid_ + 1 else hi = mid_ - 1
    loop
    return false
end function

dim as int32 sorted( 0 to 199999 )
for i as int32 = 0 to ubound( sorted )
    sorted( i ) = i * 3
next

'' A Stopwatch reads the monotonic clock: it never jumps.
dim as int32 hitsA, hitsB
var sw = chrono.Stopwatch.StartNew( )
for k as int32 = 0 to 999
    if FindLinear( sorted( ), k * 600 ) then hitsA += 1
next
var linearTime = sw.Elapsed
sw.Restart( )
for k as int32 = 0 to 999
    if FindBinary( sorted( ), k * 600 ) then hitsB += 1
next
var binaryTime = sw.Elapsed
print "same hits: "; cbool( hitsA = hitsB ); ","; hitsA
print "binary faster: "; ( binaryTime < linearTime )

'' Stop_ pauses it; time spent stopped is not counted.
sw.Restart( )
sw.Stop_( )
sleep 300, 1
sw.Start( )
sw.Stop_( )
print "paused 300 ms, counted under 100 ms: "; _
      cbool( sw.ElapsedMilliseconds < 100 )

'' CpuClock counts processor time, which sleeping does not use.
var wall = chrono.Stopwatch.StartNew( )
var cpu0 = chrono.CpuClock.ProcessTime( )
sleep 300, 1
var cpuUsed = chrono.CpuClock.ProcessTime( ) - cpu0
print "slept: wall >= 250 ms: "; _
```

```
      cbool( wall.Elapsed.TotalMilliseconds >= 250 )
print "slept: CPU < 100 ms: "; _
      cbool( cpuUsed.TotalMilliseconds < 100 )
```

examples/ch26/benchmark.out

```
same hits: true, 1000
binary faster: true
paused 300 ms, counted under 100 ms: true
slept: wall >= 250 ms: true
slept: CPU < 100 ms: true
```

Three clocks answer three questions:

- **`WallClock`** — what time is it? `UtcNow( )` gives an `Instant`, `Now( )` a `DateTime` with the local offset, `Today( )` a `LocalDate`. The wall clock can **jump backwards**: the operating system corrects it, and users change it. `timer`, the classic function, reads the same clock, and neither should be used to measure how long something took.
- **`Stopwatch`** — how long did it take? It reads a monotonic clock that only ever moves forward. `StartNew( )` creates one running; `Elapsed` is a `TimeSpan` and `ElapsedMilliseconds` a number; `Restart( )` zeroes it and runs on. **`Stop_( )`** has a trailing underscore because `stop` is a statement. A stopwatch adds up the time it runs, so stopping and starting it again leaves the paused time out.
- **`CpuClock`** — how much processor time did it use? `ProcessTime( )` is the time the process has spent running, which a `sleep` does not add to. It moves in steps of about 15 milliseconds on Windows, so a short measurement reads zero.

The listing's output is a set of comparisons, not times, because times differ from run to run and machine to machine. For a real measurement, run the work several times and keep the fastest, and measure a build with optimisation on.

The legacy bridge

examples/ch26/legacy.tiko

```
#import "std/chrono"
#include "datetime.bi"

'' datetime.bi keeps a date as a double: days since 1899-12-30.
dim as float64 serial = dateserial( 2025, 3, 4 ) _
                      + timeserial( 14, 30, 0 )
var d = chrono.DateTime.FromSerial( serial )
print "from serial: "; d.ToIsoString( )
print "and back:    "; cbool( d.Serial = serial )
```

```
'' The two count the days of the week differently.
print "weekday( ):  "; weekday( serial ); "  (1 = Sunday)"
print "DayOfWeek:   "; d.DayOfWeek; "  (1 = Monday)"

'' dateserial rolls over; the chrono constructors refuse.
dim as float64 rolled = dateserial( 2025, 13, 40 )
print "dateserial( 2025, 13, 40 ): "; year( rolled ) & "-" & _
      month( rolled ) & "-" & day( rolled )
print "DateTime( 2025, 13, 40 ) valid: "; _
      chrono.DateTime( 2025, 13, 40 ).IsValid

'' Unix time is one more epoch, both ways.
var epoch = chrono.Instant.FromUnixSeconds( 1741098600 )
print "unix 1741098600: "; epoch.ToIsoString( )
```

examples/ch26/legacy.out

```
from serial: 2025-03-04T14:30:00
and back:    true
weekday( ):   3  (1 = Sunday)
DayOfWeek:    2  (1 = Monday)
dateserial( 2025, 13, 40 ): 2026-2-9
DateTime( 2025, 13, 40 ) valid: false
unix 1741098600: 2025-03-04T14:30:00Z
```

Older code keeps a date as a `float64`: the whole part counts days from 30 December 1899 and the fraction is the time of day. `#include "datetime.bi"` provides the functions for it — `dateserial`, `timeserial`, `now`, `year`, `month`, `day`, `weekday` and the rest — and `vbcompat.bi` adds `format`. They still work, and both styles can be used in one program.

`DateTime.FromSerial( serial )` and `.Serial` convert between the two. The conversion is exact to the millisecond; below that, a `float64` holding a present-day date has no digits left to keep.

Three differences catch programs moving across:

- **`weekday( )` counts from Sunday = 1; `DayOfWeek` from Monday = 1.**
- **`dateserial` rolls over**: month 13, day 40 of 2025 is 9 February 2026. The chrono constructors refuse it.
- **A serial has no offset.** `now` returns local time with nothing to say so; `WallClock.Now( )` records the offset.

The other common epochs are one call away: `Instant.FromUnixSeconds` and `UnixSeconds` for Unix time, `FromFileTime` and `FileTime` for Windows `FILETIME`, `FromOleDate` and `OleDate` for COM, and `FromJulianDay` and `JulianDay`.

A worked example: merging logs

examples/ch26/iso-log-times.tiko

```
#import "std/chrono"

'' Log lines from three servers, each stamped in its own offset.
dim lines( ... ) as ustring = { _
    "2025-03-04T09:15:02-05:00 build started", _
    "2025-03-04T15:14:40+01:00 tests started", _
    "2025-03-04T14:20:11Z build finished", _
    "2025-03-04T19:59:58+05:30 tests finished", _
    "not-a-time something odd" }

type Entry
    as chrono.Instant at
    as ustring text
end type

dim entries( ) as Entry
for i as int32 = 0 to ubound( lines )
    dim as int32 sp = instr( lines( i ), " " )
    dim t as chrono.Instant
    if chrono.Instant.TryParseIso( left( lines( i ), sp - 1 ), t ) then
        redim preserve entries( ubound( entries ) + 1 )
        entries( ubound( entries ) ).at = t
        entries( ubound( entries ) ).text = mid( lines( i ), sp + 1 )
    else
        print "skipped: "; lines( i )
    end if
next

'' Instants sort by the moment itself, whatever offset they came in.
for i as int32 = 1 to ubound( entries )
    var e = entries( i )
    dim as int32 j = i - 1
    do while j >= 0 andalso entries( j ).at > e.at
        entries( j + 1 ) = entries( j )
        j -= 1
    loop
    entries( j + 1 ) = e
next

'' Log in UTC, with the gap since the previous line.
for i as int32 = 0 to ubound( entries )
    dim as ustring gap
```

```
    if i > 0 then
        var since = entries( i ).at - entries( i - 1 ).at
        gap = "+" & since.ToIsoString( )
    end if
    print entries( i ).at.ToIsoString( ); "  "; _
          left( gap & space( 10 ), 10 ); entries( i ).text
next
```

examples/ch26/iso-log-times.out

```
skipped: not-a-time something odd
2025-03-04T14:14:40Z            tests started
2025-03-04T14:15:02Z  +PT22S    build started
2025-03-04T14:20:11Z  +PT5M9S   build finished
2025-03-04T14:29:58Z  +PT9M47S  tests finished
```

Three machines wrote these lines, each in its own offset. Read as text they are out of order: the line stamped `09:15` was written after the one stamped `15:14`. Parsed into `Instant` values they sort by the moment itself, and the gaps between them are plain subtractions. The line whose stamp does not parse is reported, not given a made-up time.

`Instant.TryParseIso` insists on an offset or a `Z` in its input, because a time with neither cannot be placed on the UTC timeline. That is the check that keeps an unlabelled timestamp out of a merged log.

What it costs

- **The values are small and copied like numbers**: an `Instant`, `LocalTime` or `TimeSpan` is 8 bytes of ticks, a `LocalDate` is a 4-byte day number, and a `DateTime` is 16 — its ticks and its offset.
- **Conversions are constant time.** Turning ticks into year, month and day is arithmetic with no loops, so the last date in the range costs the same as the first.
- **Formatting allocates the result string once; parsing allocates nothing.**
- **`Stopwatch` reads a high-resolution counter**; `CpuClock` asks the operating system, which costs more and is far coarser.

Summary

- `#import "std/chrono"`: `LocalDate`, `LocalTime`, `DateTime`, `Instant`, `TimeSpan`. Immutable, 100 ns ticks, years 1 to 9999, `DayOfWeek` 1 = Monday.
- Out-of-range parts give **Invalid**, which absorbs arithmetic, equals nothing and prints as `""`. Test with `IsValid`. Arithmetic never wraps.

- A `DateTime` is UTC, offset or unspecified. `WithOffset` relabels; `ToOffset` converts; `AssumeUtc` turns an unspecified reading into an `Instant`.
- `ToIsoString` writes one canonical form. `TryParseIso` accepts the ISO 8601 variants and refuses everything else; `ParseIso` returns Invalid on failure.
- `ToString( pattern )` and `TryParseExact` use invariant English. An unknown pattern letter is an error: quote literal letters.
- `AddDays` is exact; `AddMonths` and `AddYears` clamp at month end and do not reverse. `EndOf*` is the last tick. Print `IsoWeekYear` with `IsoWeek`.
- Time zones are the operating system's: `TimeZoneInfo.LocalOffsetAt( instant )`.
- `Stopwatch` for elapsed time, `CpuClock` for processor time, `WallClock` for the time of day.
- `FromSerial`/`Serial` bridge to `datetime.bi`.

Exercises

1. Write `WorkingDays( first, last )`, the number of Monday-to-Friday days from `first` up to but not including `last`, and check it on a week, a month and a year. {solution="ex26-01″}
2. Read a list of lap times written as ISO 8601 durations, report any that do not parse, and print the total and the mean. {solution="ex26-02″}
3. Extend `age-calc.tiko` to give an age in years, months and days.
4. Change `AgeOn` so that someone born on 29 February has their birthday on 1 March in years with no 29 February. Which line has to change, and why is `AddYears` no longer enough?
5. `iso-log-times.tiko` prints its times in UTC. Print them instead in the offset of the first line of the log, using `DateTime.FromInstant`.

Numbers

Chapter 4 introduced the numeric types and their operators. This chapter is about what a program does with them: the built-in functions, the library that fills the gaps in them, rounding, overflow, the special floating-point values, random numbers, statistics, and — because binary floating point cannot hold a cent exactly — the types for money.

Two modules carry the library. **`#import "std/numeric"`** brings in the mathematics: rounding and selection, exact integer work, the floating-point environment, random numbers, statistics, and strict conversion between numbers and text. **`#import "std/money"`** brings in `Decimal`, `Currency` and the financial functions. Both add functions under their own prefix (`numeric.Round`, `money.Decimal`) and **change nothing that already exists**: `^`, `mod`, `val`, `str` and `rnd` behave exactly as they always have. Where one of them has a defect, the library adds a correct neighbour under another name.

The built-in functions

examples/ch27/intrinsics.tiko

```
'' The built-in numeric functions need no import.
print "abs( -3.5 ) ="; abs( -3.5 ); ", sgn( -2 ) ="; sgn( -2 )

'' int rounds down, fix rounds toward zero, frac is what fix drops.
print "int( -2.5 ) ="; int( -2.5 ); ", fix( -2.5 ) ="; fix( -2.5 ); _
      ", frac( -2.5 ) ="; frac( -2.5 )

print "sqr( 2 )    ="; sqr( 2 )
print "atn( 1 ) * 4 ="; atn( 1 ) * 4
print "exp( 1 )    ="; exp( 1 )
print "log( 10 )   ="; log( 10 ); "  (natural log)"
print "sin( 0.5 )^2 + cos( 0.5 )^2 ="; sin( 0.5 ) ^ 2 + cos( 0.5 ) ^ 2

'' mod truncates: the remainder takes the sign of the left side.
print "-7 mod 3 ="; -7 mod 3; ", -7 \ 3 ="; -7 \ 3
```

examples/ch27/intrinsics.out

```
abs( -3.5 ) = 3.5, sgn( -2 ) =-1
int( -2.5 ) =-3, fix( -2.5 ) =-2, frac( -2.5 ) =-0.5
sqr( 2 )    = 1.414213562373095
atn( 1 ) * 4 = 3.141592653589793
exp( 1 )    = 2.718281828459045
log( 10 )   = 2.302585092994046  (natural log)
```

```
sin( 0.5 )^2 + cos( 0.5 )^2 = 1
-7 mod 3 =-1, -7 \ 3 =-2
```

These need no import:

Function	Gives
`abs( x )`, `sgn( x )`	the magnitude; −1, 0 or 1
`int( x )`	the largest whole number not above `x` — it rounds **down**, so `int( -2.5 )` is −3
`fix( x )`, `frac( x )`	the whole part toward zero, and what is left
`sqr( x )`	the square root
`sin`, `cos`, `tan`, `asin`, `acos`, `atn`, `atan2`	trigonometry, in radians
`exp( x )`, `log( x )`	e to the power `x`, and the **natural** logarithm

`int` is a floor and `fix` truncates, but the names do not say so, and code that uses `int` believing it truncates is wrong for every negative number. The library's `Floor`, `Ceil` and `Truncate` say which they mean.

`mod` truncates too: the remainder takes the sign of the left-hand side, so `-7 mod 3` is −1, not the 2 that a wrapping index or a day-of-week calculation wants. `\` is integer division with the same rule.

Exact integer arithmetic

examples/ch27/intmath.tiko

```
#import "std/numeric"

'' ^ always works in float64; above 2^53 the answer is wrong.
dim as int64 b = 3, e = 38
dim as int64 viaPow = b ^ e
print "3 ^ 38       ="; viaPow
print "IPow( 3, 38 ) ="; numeric.IPow( b, e )

'' Three kinds of remainder, each named for what it does.
print "-7 mod 3        ="; -7 mod 3
print "FloorMod( -7, 3 ) ="; numeric.FloorMod( -7, 3 )
print "EuclidMod( -7, -3 ) ="; numeric.EuclidMod( -7, -3 )

'' A quotient and a remainder from one division, as a tuple.
var (pages, left_) = numeric.DivMod( 47, 10 )
print "47 items, 10 a page:"; pages; " full pages and"; left_; " over"
```

```
'' Exact integer square root, greatest common divisor, least common
'' multiple.
print "ISqrt( 99 ) ="; numeric.ISqrt( 99 ); _
      ", GCD( 12, 18 ) ="; numeric.GCD( 12, 18 ); _
      ", LCM( 4, 6 ) ="; numeric.LCM( 4, 6 )
```

examples/ch27/intmath.out

```
3 ^ 38       = 1350851717672992000
IPow( 3, 38 ) = 1350851717672992089
-7 mod 3        =-1
FloorMod( -7, 3 ) = 2
EuclidMod( -7, -3 ) = 2
47 items, 10 a page: 4 full pages and 7 over
ISqrt( 99 ) = 9, GCD( 12, 18 ) = 6, LCM( 4, 6 ) = 12
```

`^` always works in `float64`: it converts both sides, calls the floating-point power function and converts back. A `float64` has 53 bits of precision, so above 2^53 the answer is rounded, even when the true result fits in an `int64`. `IPow` multiplies integers and is exact.

> **PITFALL**
> **`^` on integers is not exact above 2^53.** `3 ^ 38` is off by 89 in the listing, and nothing says so. A literal such as `2 ^ 62` happens to come out right because the compiler works it out before the program runs, so the bug appears only once the operands are variables. Use `numeric.IPow` for integer powers.

The three remainders differ only for negative operands, and each is named for its rule: `FloorMod` takes the sign of the divisor, `EuclidMod` is never negative, and `mod` takes the sign of the dividend. `FloorDiv` and `EuclidDiv` are the matching quotients. `DivMod( a, b )` returns the quotient and remainder together as a tuple (Chapter 20). `ISqrt` is the exact integer square root, which `int( sqr( n ) )` is not for large `n`; `GCD` and `LCM` complete the set.

Overflow

examples/ch27/overflow.tiko

```
#import "std/numeric"

dim as int32 top = 2147483647

'' Stored back in an int32, a sum past the top wraps, silently.
```

```
dim as int32 wrapped = top + 1
print "top + 1, as int32 ="; wrapped

'' Checked: an empty Optional instead of a wrong number.
var sum = numeric.AddChecked( top, 1 )
print "AddChecked has a value: "; sum.HasValue( )
var ok = numeric.AddChecked( top, -1 )
print "AddChecked( top, -1 ) ="; ok.Value( )

'' Saturating: stop at the end of the range.
print "AddSaturating     ="; numeric.AddSaturating( top, 1 )

'' Wrapping: what + does, spelled so the reader knows it was meant.
print "AddWrapping       ="; numeric.AddWrapping( top, 1 )

'' abs of the most negative value has no positive partner.
dim as int64 lo = -9223372036854775807 - 1
print "abs( lo )         ="; abs( lo )
print "AbsChecked has a value: "; numeric.AbsChecked( lo ).HasValue( )

'' The bit functions.
dim as uint32 flags = &b10110000
print "PopCount ="; numeric.PopCount( flags ); _
      ", TrailingZeros ="; numeric.TrailingZeros( flags ); _
      ", NextPow2( 100 ) ="; numeric.NextPow2( 100ul )
```

examples/ch27/overflow.out

```
top + 1, as int32 =-2147483648
AddChecked has a value: false
AddChecked( top, -1 ) = 2147483646
AddSaturating     = 2147483647
AddWrapping       =-2147483648
abs( lo )         =-9223372036854775808
AbsChecked has a value: false
PopCount = 3, TrailingZeros = 4, NextPow2( 100 ) =128
```

Integer arithmetic **wraps** when a result does not fit: one past the largest `int32` is the smallest. Nothing reports it. The library offers three ways to say what should happen instead, for every integer type:

- **Checked** — `AddChecked`, `SubChecked`, `MulChecked`, `DivChecked`, `NegChecked`, `AbsChecked`, `IPowChecked` — return an `Optional` (Chapter 18) that is empty on overflow.
- **Saturating** — `AddSaturating` and its siblings — stop at the end of the range.

- **Wrapping** — `AddWrapping` and its siblings — do exactly what the operators do. They exist so that deliberate wrapping, in a hash or a checksum, says it is deliberate.

`abs` of the most negative value is the classic case: it has no positive partner, so `abs` returns the same negative number. `AbsChecked` reports it.

These protect the calls a programmer thought to protect. Overflow elsewhere is still silent.

The bit functions work on the unsigned types: `PopCount`, `LeadingZeros`, `TrailingZeros`, `BitWidth`, `Parity`, `RotateLeft`, `RotateRight`, `ByteSwap`, `BitReverse`, `IsPow2` and `NextPow2`.

Rounding and selection

examples/ch27/rounding.tiko

```
#import "std/numeric"

'' Half away from zero, and half to even (banker's rounding).
print "Round:         "; numeric.Round( 2.5 ); numeric.Round( 3.5 ); _
      numeric.Round( -2.5 )
print "RoundHalfEven: "; numeric.RoundHalfEven( 2.5 ); _
      numeric.RoundHalfEven( 3.5 ); numeric.RoundHalfEven( -2.5 )

'' Floor, Ceil and Truncate say which way they go.
print "Floor( -1.5 ) ="; numeric.Floor( -1.5 ); _
      ", Ceil( -1.5 ) ="; numeric.Ceil( -1.5 ); _
      ", Truncate( -1.5 ) ="; numeric.Truncate( -1.5 )

'' Rounding to places scales by a power of ten first.
print "Round( 3.14159, 2 ) ="; numeric.Round( 3.14159, 2 )
print "Round( 2.675, 2 )   ="; numeric.Round( 2.675, 2 )

'' Selection.
print "Min ="; numeric.Min( 3, 8 ); ", Max ="; numeric.Max( 3, 8 ); _
      ", Clamp( 15, 0, 10 ) ="; numeric.Clamp( 15, 0, 10 )
print "Lerp( 0, 200, 0.25 ) ="; numeric.Lerp( 0.0, 200.0, 0.25 )
print "Sign( -0.5 ) ="; numeric.Sign( -0.5 )
```

examples/ch27/rounding.out

```
Round:          3 4-3
RoundHalfEven:  2 4-2
Floor( -1.5 ) =-2, Ceil( -1.5 ) =-1, Truncate( -1.5 ) =-1
Round( 3.14159, 2 ) = 3.14
```

```
Round( 2.675, 2 )   = 2.68
Min = 3, Max = 8, Clamp( 15, 0, 10 ) = 10
Lerp( 0, 200, 0.25 ) = 50
Sign( -0.5 ) =-1
```

`Round` takes a half away from zero. `RoundHalfEven` takes a half to the even neighbour, which is banker's rounding: across many values it rounds up as often as down, where `Round` always pushes the halves outward. Both take an optional number of decimal places.

PITFALL

`Round( 2.675, 2 )` is 2.68, though the nearest `float64` to 2.675 is slightly below it. Rounding to places multiplies by a power of ten first, and 2.675×100 comes out at exactly 267.5, which is a tie. No scale-and-round method avoids this. For amounts of money, use `Decimal`, below.

`Min`, `Max` and `Clamp( v, lo, hi )` work on every numeric type. `Lerp( a, b, t )` moves from `a` toward `b` by the fraction `t`. `Sign` is like `sgn` but always returns an `int32`, so its result can index an array or drive a `select case`. `numeric.PI`, `TAU`, `E`, `SQRT2`, `LN2` and `LN10` are the usual constants.

The special values

examples/ch27/special.tiko

```
#import "std/numeric"

'' The special values are functions, not constants.
var nan_ = numeric.NaN( )
var inf = numeric.Infinity( )

'' A NaN equals nothing, not even itself: ask with IsNaN.
print "NaN = NaN:      "; cbool( nan_ = nan_ )
print "IsNaN:          "; numeric.IsNaN( nan_ )
print "IsInfinite:     "; numeric.IsInfinite( inf )
print "IsFinite( 1e308 * 10 ): "; numeric.IsFinite( 1e308 * 10 )

'' Classify names the kind of value.
print "Classify( 0 ) is FP_ZERO: "; _
      cbool( numeric.Classify( 0.0 ) = numeric.FP_ZERO )
print "Classify( inf ) is FP_INFINITE: "; _
      cbool( numeric.Classify( inf ) = numeric.FP_INFINITE )
```

```
'' NaN survives Min instead of quietly disappearing.
print "Min( NaN, 1 ) is NaN: "; _
      numeric.IsNaN( numeric.Min( nan_, 1.0 ) )

'' Two zeros: equal as numbers, different in the sign bit.
dim as float64 negZero = -0.0
print "-0 = 0: "; cbool( negZero = 0.0 ); _
      ", IsNegative( -0 ): "; numeric.IsNegative( negZero )
```

examples/ch27/special.out

```
NaN = NaN:      false
IsNaN:          true
IsInfinite:     true
IsFinite( 1e308 * 10 ): false
Classify( 0 ) is FP_ZERO: true
Classify( inf ) is FP_INFINITE: true
Min( NaN, 1 ) is NaN: true
-0 = 0: true, IsNegative( -0 ): true
```

A `float64` can hold more than numbers: positive and negative infinity, a NaN ("not a number", the result of 0/0 and similar), and a zero with a sign. `numeric.NaN( )` and `numeric.Infinity( )` make them; they are functions because a constant cannot be initialised by dividing by zero.

A NaN is not equal to anything, itself included, so `x = x` is false for a NaN. Ask `IsNaN( x )`. `IsInfinite`, `IsFinite`, `IsNormal` and `IsSubnormal` complete the questions, and `Classify( x )` names the kind: `FP_NAN`, `FP_INFINITE`, `FP_ZERO`, `FP_SUBNORMAL` or `FP_NORMAL`.

The library **passes a NaN on** rather than dropping it: `Min( NaN, 1 )` is NaN. Losing the NaN there would turn an error that could be found into a wrong answer that cannot.

`-0.0` and `0.0` are equal as numbers and differ in the sign bit; `IsNegative` reads the bit.

How `print` shows a NaN or an infinity depends on the platform's runtime, so print a test of the value, as the listing does, rather than the value itself.

The floating-point environment

examples/ch27/fenv.tiko

```
#import "std/numeric"

'' One divided by three, under a chosen rounding mode. The mode is
'' put back when 'r' goes out of scope, however the function ends.
```

```
function Third( byval mode as int32 ) as float64
    dim r as numeric.ScopedRounding = numeric.ScopedRounding( mode )
    dim as float64 one = 1.0, three = 3.0
    return one / three
end function

print "toward +inf: "; _
      numeric.ToString( Third( numeric.ROUND_TOWARD_POSITIVE ) )
print "toward -inf: "; _
      numeric.ToString( Third( numeric.ROUND_TOWARD_NEGATIVE ) )
print "nearest:     "; _
      numeric.ToString( Third( numeric.ROUND_TO_NEAREST ) )
print "restored:    "; _
      cbool( numeric.GetRoundingMode( ) = numeric.ROUND_TO_NEAREST )

'' Exception flags record what went wrong, without stopping anything.
numeric.ClearFpExceptions( )
dim as float64 x = 1.0, zero = 0.0
var r = x / zero
print "divide by zero flagged: "; _
      numeric.TestFpException( numeric.FP_DIVBYZERO )
print "result is infinite:     "; numeric.IsInfinite( r )
```

examples/ch27/fenv.out

```
toward +inf: 0.33333333333333337
toward -inf: 0.3333333333333333
nearest:     0.3333333333333333
restored:    true
divide by zero flagged: true
result is infinite:     true
```

The processor rounds every floating-point result to the nearest `float64` unless told otherwise. `ScopedRounding( mode )` changes that for as long as the variable lives and puts the old mode back when it goes out of scope, however the scope is left. The modes are `ROUND_TO_NEAREST`, `ROUND_TOWARD_POSITIVE`, `ROUND_TOWARD_NEGATIVE` and `ROUND_TOWARD_ZERO`. Rounding the same calculation both ways gives a lower and an upper bound for the true value, which is the basis of interval arithmetic.

The processor also keeps **exception flags**: invalid operation, division by zero, overflow, underflow and inexact result. They do not stop the program; they record that something happened. `ClearFpExceptions( )` resets them, `TestFpException( flag )` reads one, with `FP_INVALID`, `FP_DIVBYZERO`, `FP_OVERFLOW`, `FP_UNDERFLOW` and `FP_INEXACT`.

The rounding mode belongs to the thread that set it. And because the optimiser may move calculations, a change of mode is only reliable around work the compiler cannot move past it — which is why the listing puts the division in its own function.

Random numbers

examples/ch27/random.tiko

```
#import "std/numeric"

'' A seeded generator: the same seed gives the same numbers.
dim rng as numeric.Rng = numeric.RngSeeded( 12345 )

print "ten dice:";
for i as int32 = 1 to 10
    print rng.NextInt( 1, 6 );
next
print

print "a double in [0, 1): "; numeric.ToString( rng.NextDouble( ) )

'' Shuffle every element into a random order, each order equally likely.
dim deck( 1 to 8 ) as int32 = { 1, 2, 3, 4, 5, 6, 7, 8 }
numeric.Shuffle( rng, deck( ) )
print "shuffled:";
for i as int32 = 1 to 8
    print deck( i );
next
print

'' An Rng is a value: a copy carries on from the same point.
var copy_ = rng
print "original and copy agree: "; _
      cbool( rng.NextInt( 1, 1000 ) = copy_.NextInt( 1, 1000 ) )
```

examples/ch27/random.out

```
ten dice: 5 1 6 1 4 1 1 2 3 6
a double in [0, 1): 0.7818442024444772
shuffled: 8 3 2 4 1 7 6 5
original and copy agree: true
```

The classic pair still works: `randomize` seeds the generator and `rnd` returns a value from 0 up to but not including 1. `#include "fbprng.bi"` adds a second argument to `randomize` that picks one of several algorithms. Two things are wrong with that pair for serious use. There is **one** generator for the whole program, so a test cannot have

a reproducible stream of its own without disturbing everyone else's. And the usual way to get a whole number, `int( rnd * n )`, is very slightly **biased**: some results come up more often than others.

`numeric.Rng` fixes both. **An `Rng` is a value**: each one is its own stream, and copying one makes a second stream that continues from the same point. `RngSeeded( seed )` gives the same numbers every time, on every platform, which is what tests and simulations need; `RngFromEntropy( )` seeds from the operating system for a different stream each run.

Member	Gives
`NextInt( lo, hi )`	a whole number from `lo` to `hi`, **both included**, without bias
`NextDouble( )`	a `float64` from 0 up to but not including 1
`NextBoolean( )`, `NextBooleanP( p )`	true half the time, or with probability `p`
`NextNormal( )`, `NextNormalP( mean, sd )`	the bell curve
`NextExponential( λ )`, `NextPoisson( λ )`	waiting times and counts

`Shuffle( rng, a( ) )` puts an array in random order with every order equally likely, and `Choice( rng, a( ) )` picks one element. For keys and tokens, where unpredictability matters more than reproducibility, use `CryptoBytes` and `CryptoULongInt` instead: they have no seed and are not an `Rng`.

Identifiers

A record that is made in more than one place — two servers, an offline phone, a program that merges files — cannot be numbered by a counter, because no one counter sees every record. A **UUID** is a 128-bit number chosen so that the chance of two being equal is negligible, made anywhere with no coordination:

examples/ch27/ids.tiko

```
#import "std/uuids"
#import "std/containers"

'' Random identifiers: no two alike, nothing to guess.
dim a as uuids.Uuid = uuids.NewV4( )
dim b as uuids.Uuid = uuids.NewV4( )
print "version"; a.Version( ); ","; len( a.ToString( ) ); " characters"
print "a = b? "; a = b
```

```
'' Time-ordered identifiers sort in the order they were made.
dim older as uuids.Uuid = uuids.NewV7( )
dim newer as uuids.Uuid = uuids.NewV7( )
print "version"; older.Version( ); ", in order? "; older < newer
print "made this century? "; cbool( older.UnixMillis( ) > 946684800000 )

'' Parsing accepts the common spellings, and prints one.
var r = uuids.Parse( "{F81D4FAE-7DEC-11D0-A765-00A0C91E6BF6}" )
print r.Value( ).ToString( )

'' A Uuid is a key like any other.
dim owners as containers.Map( of uuids.Uuid, ustring )
owners[ a ] = "ada"
print "owner of a: "; owners[ a ]
```

examples/ch27/ids.out

```
version 4, 36 characters
a = b? false
version 7, in order? true
made this century? true
f81d4fae-7dec-11d0-a765-00a0c91e6bf6
owner of a: ada
```

`uuids.NewV4( )` fills 122 bits from the operating system's secure generator, the one behind `CryptoBytes`: nothing about the value can be predicted from another. `uuids.NewV7( )` puts the time in milliseconds first and random bits after, so identifiers made later **sort later**. As database keys, v7 values arrive in order and keep an index compact where v4 values scatter across it; within one program, each `NewV7` is greater than the one before even when many are made in the same millisecond. The price is that a v7 value tells anyone who sees it when it was made — `UnixMillis( )` reads it back.

`Parse` accepts the common spellings — with or without hyphens and braces, any case, or a `urn:uuid:` prefix — and `ToString( )` always writes the lower-case 8-4-4-4-12 form. A `Uuid` has `=` and `<` and a hash, so it is a `Map` or `Set` key like any built-in type.

Statistics

examples/ch27/dice-stats.tiko

```
#import "std/numeric"
#import "std/containers"

'' Roll two dice 36000 times with a fixed seed, so every run agrees.
```

```
dim rng as numeric.Rng = numeric.RngSeeded( 2026 )
dim totals as containers.Array( of float64 )
dim counts( 2 to 12 ) as int32
for i as int32 = 1 to 36000
    var t = rng.NextInt( 1, 6 ) + rng.NextInt( 1, 6 )
    totals.Push( cdbl( t ) )
    counts( t ) += 1
next

'' A sideways bar chart, one # for every 100 rolls.
for t as int32 = 2 to 12
    print right( " " & t, 2 ); right( space( 6 ) & counts( t ), 6 ); _
          "  "; string( counts( t ) \ 100, "#" )
next

'' Summary statistics over the whole array.
print "mean   "; numeric.Round( numeric.Mean( totals ), 3 )
print "median "; numeric.Median( totals )
print "stddev "; numeric.Round( numeric.StdDev( totals ), 3 )
```

examples/ch27/dice-stats.out

```
 2  1004  ##########
 3  1990  ####################
 4  3023  ##############################
 5  3894  ######################################
 6  5070  ##################################################
 7  5955  ###########################################################
 8  5094  ##################################################
 9  3942  ######################################
10  3041  ##############################
11  1972  ####################
12  1015  ##########
mean    7.004
median  7
stddev  2.415
```

The statistics functions take a `containers.Array` (Chapter 24) of numbers: `Mean`, `Median`, `Percentile`, `Variance` and `StdDev` (sample, dividing by n - 1), `VariancePopulation` and `StdDevPopulation` (dividing by n), `MinOf`, `MaxOf`, `Range`, `Covariance` and `Correlation`. `Median` and `Percentile` copy the data before sorting it, so asking a question does not reorder the array. The mean of an empty array is NaN, not zero, because zero is a real mean.

examples/ch27/running.tiko

```
#import "std/numeric"
#import "std/containers"

'' One pass, constant space: RunningStats sees each value once.
dim temps( ... ) as float64 = { 21.5, 23.0, 19.5, 25.0, 22.0, 24.5 }
dim acc as numeric.RunningStats
for i as int32 = 0 to ubound( temps )
    acc.Add( temps( i ) )
next
print "count "; acc.Count( ); ", mean"; acc.Mean( ); _
      ", min"; acc.MinValue( ); ", max"; acc.MaxValue( )
print "stddev"; numeric.Round( acc.StdDev( ), 4 )

'' Adding many small values to a large one: the plain loop loses them.
dim values as containers.Array( of float64 )
values.Push( 1e16 )
for i as int32 = 1 to 100000
    values.Push( 1.0 )
next
dim as float64 loopTotal = 0
for each v in values
    loopTotal += v
next
print "loop:      "; numeric.ToString( loopTotal )
print "NaiveSum:  "; numeric.ToString( numeric.NaiveSum( values ) )
print "Sum:       "; numeric.ToString( numeric.Sum( values ) )
```

examples/ch27/running.out

```
count 6, mean 22.58333333333333, min 19.5, max 25
stddev 2.0351
loop:      1e+16
NaiveSum:  1e+16
Sum:       10000000000100000
```

`RunningStats` sees each value once and keeps only a few numbers, so it can summarise a stream too large to hold. `Add`, then `Count`, `Mean`, `Variance`, `StdDev`, `MinValue` and `MaxValue`; `Merge` combines two accumulators exactly, so separate threads can each summarise a share and then combine the results.

`Sum` is not a loop. Adding a small number to a large `float64` can lose the small one entirely: 1e16 + 1 is 1e16. A plain loop, and `NaiveSum`, lose all hundred thousand ones in the listing. `Sum` keeps a running correction and gets the exact answer, at a few extra operations per element.

Numbers as text

examples/ch27/numtext.tiko

```
#import "std/numeric"

'' val cannot fail: garbage and a real zero look the same.
print "val( ""42abc"" ) ="; val( "42abc" ); _
      ", val( ""abc"" ) ="; val( "abc" )

'' TryParseDouble takes the whole string or nothing.
dim inputs( ... ) as ustring = { "2.5", "42abc", " 7", "1e3", "" }
for i as int32 = 0 to ubound( inputs )
    var v = numeric.TryParseDouble( inputs( i ) )
    print left( """ & inputs( i ) & """ & space( 8 ), 8 ); _
          iif( v.HasValue( ), str( v.Value( ) ), "refused" )
next

'' str rounds for show; ToString gives digits that read back exactly.
dim as float64 x = 0.1, y = 0.2
print "str:      "; str( x + y )
print "ToString: "; numeric.ToString( x + y )
var back = numeric.TryParseDouble( numeric.ToString( x + y ) ).Value( )
print "str reads back: "; cbool( val( str( x + y ) ) = x + y ); _
      ", ToString reads back: "; cbool( back = x + y )

'' A NumFormat says how a number should look.
dim f as numeric.NumFormat
f.style = numeric.NUM_FIXED
f.digits = 2
f.grouping = true
print numeric.ToString( 1234567.891, f )
```

examples/ch27/numtext.out

```
val( "42abc" ) = 42, val( "abc" ) = 0
"2.5"   2.5
"42abc" refused
" 7"    refused
"1e3"   1000
""      refused
str:      0.3
ToString: 0.30000000000000004
str reads back: false, ToString reads back: true
1,234,567.89
```

`val` cannot fail. It reads as much of the text as looks like a number and ignores the rest, so `"42abc"` is 42 and `"abc"` is 0 — the same as a real zero. **`TryParseDouble`** and its siblings `TryParseSingle`, `TryParseLong`, `TryParseLongInt` and the unsigned forms return an `Optional` that is empty unless the **whole** text is a number. They do not trim spaces unless asked: a second argument of `PARSE_TRIM` or `PARSE_ALLOW_TRAILING` relaxes them.

`str` rounds a `float64` to fewer digits than it holds, which is easy to read and does not always read back as the same value: `0.1 + 0.2` prints as `0.3`, which is a different number. **`numeric.ToString( x )`** writes the shortest text that reads back exactly. For display, `ToString( x, format )` takes a `NumFormat` that sets the style (`NUM_FIXED`, `NUM_SCIENTIFIC`, `NUM_PERCENT` and others), the number of digits, thousands grouping and the separators. The separators are whatever the program says — never the machine's locale, so a file written on one machine reads correctly on another.

Money

examples/ch27/money.tiko

```
#import "std/money"

'' Binary floating point cannot hold 0.1 exactly.
dim as float64 a = 0.1, b = 0.2
print "float64: 0.1 + 0.2 = 0.3 is "; cbool( a + b = 0.3 )

'' Decimal holds decimal fractions exactly.
var d1 = money.DecimalParse( "0.1" ).Value( )
var d2 = money.DecimalParse( "0.2" ).Value( )
print "Decimal: 0.1 + 0.2 = "; ( d1 + d2 ).ToString( ); " is "; _
      ( d1 + d2 = money.DecimalParse( "0.3" ).Value( ) )

'' Currency: four places, fixed, in eight bytes.
var price = money.CurrencyParse( "19.99" ).Value( )
print "3 at 19.99 = "; ( price * 3 ).ToString( )
print "2 / 3      = "; ( money.CurrencyFromInt( 2 ) / 3 ).ToString( )

'' Text is parsed strictly: a fifth place is refused, not dropped.
print "parse 1.23456 as Currency: "; _
      money.CurrencyParse( "1.23456" ).HasValue( )

'' A Decimal keeps the scale it was written with.
var rate = money.DecimalParse( "0.0825" ).Value( )
var amount = money.DecimalParse( "59.97" ).Value( )
print "tax at 8.25% = "; ( amount * rate ).ToString( ); _
      ", to cents "; ( amount * rate ).Round( 2 ).ToString( )
```

```
'' Split 100.00 three ways so the parts add back to the total.
dim weights( 0 to 2 ) as int64 = { 1, 1, 1 }
dim parts( ) as money.Decimal
money.DecimalApportion( money.DecimalParse( "100.00" ).Value( ), 2, _
                        weights( ), parts( ) )
print "100.00 in three:";
for i as int32 = 0 to ubound( parts )
    print " "; parts( i ).ToString( );
next
print
```

examples/ch27/money.out

```
float64: 0.1 + 0.2 = 0.3 is false
Decimal: 0.1 + 0.2 = 0.3 is true
3 at 19.99 = 59.9700
2 / 3      = 0.6667
parse 1.23456 as Currency: false
tax at 8.25% = 4.947525, to cents 4.95
100.00 in three: 33.34 33.33 33.33
```

A `float64` stores binary fractions, and most decimal fractions are not one: 0.1 is stored as a nearby value, and 0.1 + 0.2 is not quite 0.3. For measurements that error is far too small to matter. For money it is a cent that does not reconcile.

PITFALL
Never keep money in a `float64`. Each amount is very slightly wrong, and the errors add up across a ledger until a total is off by a cent — or, in the loan below, by 97 of them.

`#import "std/money"` provides two exact types:

	`Currency`	**`Decimal`**
holds	an `int64` counting ten-thousandths	a 96-bit whole number, a scale and a sign
decimal places	4, always	0 to 28, per value
size	8 bytes	16 bytes
speed	as fast as an integer	slower: done in software

Use **`Currency`** when four places are enough; use **`Decimal`** for exchange rates, tax rates and anything else that needs more. `DecimalFromCurrency` and `ToCurrency( )` convert between them.

Both are made by **named functions**, not constructors: `DecimalParse( text )` and `CurrencyParse( text )` return an `Optional`, and `DecimalFromInt`, `CurrencyFromInt`, `DecimalFromDouble` and `CurrencyFromDouble` say what they convert from. A constructor taking an `int64` would also accept a `float64` without a word and drop its fraction. Parsing is strict: `CurrencyParse( "1.23456" )` is refused rather than cut to four places.

The arithmetic follows the same rules in both:

- **Addition, subtraction and multiplication by a whole number are exact.**
- **Division rounds half to even**, never truncates: 2 / 3 is 0.6667.
- **A `Decimal` keeps its scale.** 59.97 × 0.0825 is 4.947525, exactly; `Round( 2 )` makes it cents when cents are wanted. `1.10` and `1.1` are equal and print differently.
- **Overflow gives an Invalid value**, which absorbs arithmetic and equals nothing, as chrono's does (Chapter 26). Test with `IsValid( )`. A value declared with no initialiser is zero.

`Currency × Currency` does not exist: money times money is not money. `Decimal` is a number that happens to be decimal, and it multiplies and divides by itself freely.

Splitting an amount is where hand-written code goes wrong: 100.00 in three is 33.33 three times, which is 99.99. `DecimalApportion( total, scale, weights( ), parts( ) )` and `Apportion` for `Currency` hand the leftover cents out so the parts always add up to the total. The weights are an array of `int64`.

A `Decimal` array passed for the weights is refused: `TKO0687: a bydesc parameter takes an array of exactly its element type`.

Loans

examples/ch27/loan-payment.tiko

```
#import "std/money"

'' The usual float64 amortization: round the payment, round each
'' month's interest, subtract. The balance does not end at zero.
dim as float64 bal = 400000, i = 0.0595 / 12
dim as float64 pmt = bal * i / ( 1 - ( 1 + i ) ^ -360 )
pmt = int( pmt * 100 + 0.5 ) / 100
for k as int32 = 1 to 360
    dim as float64 interest = int( bal * i * 100 + 0.5 ) / 100
    bal -= pmt - interest
next
print "float64 payment"; pmt; ", final balance"; bal

'' The same loan in Decimal, with the last payment absorbing the rest.
var p = money.DecimalParse( "400000" ).Value( )
```

```
var r = money.DecimalParse( "0.0595" ).Value( )
var t = money.LoanTermsUsMortgage( p, r, 30 )
dim as int32 n = 0
dim as money.Decimal last
for each row in money.LoanSchedule( t ).Value( )
    n += 1
    last = row.ClosingBalance_
next
var payment = money.LoanPayment( t ).Value( )
print "Decimal payment "; payment.ToStringFixed( 2 ); ","; n; _
      " payments, final balance "; last.ToStringFixed( 2 )

'' A Canadian mortgage compounds half-yearly but is paid monthly.
var ca = money.LoanTermsCanadianMortgage( p, r, 30 )
print "Canadian payment "; _
      money.LoanPayment( ca ).Value( ).ToStringFixed( 2 )
```

examples/ch27/loan-payment.out

```
float64 payment 2385.36, final balance-0.9699999996669249
Decimal payment 2385.36, 360 payments, final balance 0.00
Canadian payment 2366.81
```

The first half of the listing is how a loan is usually computed: work out the payment, round it to cents, round each month's interest to cents, subtract. The payment is right. After 360 payments the balance is not zero but a negative 97 cents, plus a trace of binary error — the borrower has overpaid, because the rounding of each payment was never accounted for.

`money.LoanSchedule` does the same calculation in `Decimal` and lets the final payment absorb whatever is left, so the closing balance is exactly zero. Each row's principal is the payment minus the interest, so every row adds up. `LoanPayment( terms )` gives the payment alone; `LoanTermsUsMortgage` and `LoanTermsCanadianMortgage` set up the two common cases, which differ in how often interest compounds — monthly in the United States, half-yearly in Canada — though both are paid monthly.

`std/money` goes much further: present and future value, internal rate of return, depreciation schedules, day-count conventions and portfolio returns. The calls that can fail return a `Result` (Chapter 18) whose error says why.

What it costs

- **The `numeric` functions are ordinary function calls.** None of them is worked out when the program is compiled, even on constant arguments.
- **`IPow` multiplies**: about as many multiplications as the exponent has bits.

- **Checked arithmetic adds a comparison** to each operation it guards.
- **`Sum` costs a few more operations per element than a loop**; `Median` copies and sorts.
- **`Currency` adds like an `int64`. `Decimal` arithmetic is done in software**, and is many times slower; it is still fast compared with anything a program does with the result.
- **Integer, `Currency`, `Decimal` and `Rng` results are the same on every platform.** Square roots, logarithms and trigonometry come from each platform's maths library and may differ in the last bit.

Summary

- The intrinsics need no import. `int` is a floor, `fix` truncates, and `mod` takes the sign of the dividend.
- `#import "std/numeric"`: `Round`, `RoundHalfEven`, `Floor`, `Ceil`, `Truncate`, `Min`, `Max`, `Clamp`, `Lerp`, `Sign`, and the constants.
- `^` works in `float64`; `IPow` is exact. `FloorMod`, `EuclidMod`, `DivMod`, `ISqrt`, `GCD`.
- Overflow wraps silently. `*Checked` returns an empty `Optional`; `*Saturating` stops at the limit; `*Wrapping` says the wrap is meant.
- A NaN equals nothing: use `IsNaN` and `Classify`. NaN survives `Min` and `Max`.
- `ScopedRounding` sets the rounding mode for a scope; the exception flags record what happened.
- `Rng` is a value: `RngSeeded` for reproducible streams, `NextInt` with both ends included.
- `#import "std/uuids"`: `NewV4` is random, `NewV7` sorts by creation time; `Parse`, `ToString`, and `Uuid` works as a map key.
- Statistics take a `containers.Array`; `RunningStats` works in one pass; `Sum` is exact where a loop is not.
- `TryParseDouble` takes the whole text or nothing; `ToString` reads back exactly.
- `#import "std/money"`: never keep money in a `float64`. `Currency` for four places, `Decimal` for up to 28; division rounds half to even; `DecimalApportion` splits exactly; `LoanSchedule` ends at exactly zero.

Exercises

1. Put 1000.00 in a savings account at 4% a year, credit and round the interest to cents each year, and print the balance for ten years in `Decimal`. {solution="ex27-01″}
2. Estimate π by throwing a million random points at a unit square and counting how many land inside the quarter circle. Use a seeded `Rng`, so the answer is the same every run. {solution="ex27-02″}

3. Rewrite `dice-stats.tiko` with `RunningStats` instead of an array. Which of its three statistics can no longer be computed, and why?
4. Write `Average( a, b )` for two `int32` values that cannot overflow, and test it with both values near the top of the range.
5. Change the `float64` loop in `loan-payment.tiko` so its last payment absorbs the remainder. Does the balance now end at exactly zero?
6. Make 1000 `NewV7` identifiers in an ordinary array, put them in random order with `numeric.Shuffle` and a seeded `Rng`, copy them into a `containers.Array` and `Sort` it, and check they come back in the order they were made. What would the same test show for `NewV4`?

Logging, Terminal UIs and SQLite

Three libraries cover three things most programs end up doing. A program that runs for any length of time needs to say what it did, and **`std/logs`** writes that record properly: with levels, typed fields, more than one destination and no lines torn apart by threads. A tool that a person drives from a terminal can do better than `print` and `input`, and **`std/tui`** builds full-screen interfaces from a grid of cells, a layout solver and some thirty widgets. A program that keeps data between runs usually wants a database rather than files of its own invention, and **`vendor/sqlite3`** puts SQLite, the most widely deployed database there is, an `#import` away.

The chapter ends with a map of the finance library, the rest of the `std/money` module from Chapter 27.

Logging

examples/ch28/log-demo.tiko

```
#import "std/logs"

'' A renderer with no timestamp, so the output is the same every run.
dim tr as logs.TextRenderer
tr.stamp = logs.TS_NONE

'' Logs normally go to stderr; this listing sends them to stdout.
dim con as logs.ConsoleSink = logs.ConsoleSink( logs.CT_STDOUT, @tr )

dim lg as logs.Logger
lg.MinLevel = logs.LOG_DEBUG
lg.AddSink( @con )

'' One call per record. Below MinLevel, nothing is built at all.
lg.Info( "server started" )
lg.Debug( "cache warm" )
lg.Trace( "not shown: trace is below debug" )

'' Fields are typed, and chained; Msg( ) sends the record.
lg.Warn( ).AddText( "user", "quinn" ).AddInt( "attempts", 3 ) _
          .Msg( "login failed" )

'' A child logger carries fields of its own into every record.
var db = lg.WithField( "component", logs.LogValue( "db" ) )
db.Info( ).AddInt( "rows", 42 ).AddFloat( "ms", 1.5 ) _
          .Msg( "query complete" )
```

```
'' Levels as text, both ways.
dim lv as logs.LogLevel
if logs.LevelFromName( "warn", lv ) then
    print "warn is level"; lv; ", "; logs.LevelName( logs.LOG_ERROR ); _
          " is"; logs.LOG_ERROR
end if
```

examples/ch28/log-demo.out

```
INFO   server started
DEBUG  cache warm
WARN   login failed  user=quinn attempts=3
INFO   query complete  component=db rows=42 ms=1.5
warn is level 40, ERROR is 50
```

Three objects make a log. A **`Logger`** is what the program calls. A **sink** is where records go — `ConsoleSink` to the terminal, `FileSink` to a file — and a logger may have several. A **renderer** turns a record into text: `TextRenderer` for people, `JsonRenderer` for programs. `lg.AddSink( @con )` connects them. **The logger does not own its sinks**: it keeps a pointer, so a sink must live at least as long as every logger that uses it.

Levels

Every record has a **level**, from least to most severe: `LOG_TRACE`, `LOG_DEBUG`, `LOG_INFO`, `LOG_WARN`, `LOG_ERROR` and `LOG_FATAL` (10 to 60, ten apart so a program can define its own in between), plus `LOG_OFF`. **`MinLevel`** discards everything below it, and a discarded record is never built — its fields are not converted, and nothing is allocated. Its arguments are still evaluated, though: `lg.Trace( ).AddText( "dump", Expensive( ) )` calls `Expensive` whatever the level.

`LOG_FATAL` is a severity and nothing more. It does not end the program; a library that called `end` on its host's behalf would be a bad library. Write the `end` yourself.

`LevelName`, `LevelNameShort` and `LevelFromName` convert to and from text, so a level can come from a configuration file or the command line. `LevelFromName` leaves its output unchanged when the name is not a level — a typo in a setting does not silently switch logging off.

Records with fields

`lg.Info( "message" )` logs a line. For anything with data in it, **`lg.Warn( )`** without a message starts a record, each **field** method adds a name and a value, and **`Msg( "text" )`** finishes and sends it: `AddText`, `AddInt`, `AddUInt`, `AddFloat`, `AddBool`, `AddPtr`, and `AddTime` and `AddSpan` for Chapter 26's `Instant` and `TimeSpan`. The values keep their types all the way to the renderer, so a JSON renderer writes `3` and not

`"3"`. A record that is never finished is never sent: forgetting `Msg( )` costs a missing line, which is noticed, rather than a stray one.

`WithField( name, value )` returns a **child logger** that adds that field to every record it writes, with the parent's level and sinks — `db` above marks everything from the database layer. The child is a copy: changing the parent later does not change it.

Files, JSON and the default logger

examples/ch28/log-json.tiko

```
#import "std/logs"

'' JSON lines in a file: one object per record, for other programs.
dim js as logs.JsonRenderer
dim file_ as logs.FileSink = logs.FileSink( "orders.log", @js )

dim lg as logs.Logger
lg.AddSink( @file_ )
lg.Info( ).AddText( "user", "quinn" ).AddInt( "items", 3 ) _
          .Msg( "order placed" )
lg.Error( ).AddText( "card", "visa" ).Msg( "payment declined" )
file_.Close( )

'' Each line starts with the time, which differs on every run, so this
'' prints each record from its level onwards.
dim as int32 f = freefile( )
open "orders.log" for input as #f
dim as ustring record
do until eof( f )
    line input #f, record
    print "{"; mid( record, instr( record, """level""" ) )
loop
close #f
kill "orders.log"
```

examples/ch28/log-json.out

```
{"level":"INFO","msg":"order placed","user":"quinn","items":3}
{"level":"ERROR","msg":"payment declined","card":"visa"}
```

`FileSink( path, @renderer )` appends to a file. It can also **rotate** the file — set `policy` to `RP_SIZE`, `RP_DAILY` or `RP_BOTH`, with `maxbytes` and `maxbackups` — so a long-running program's log does not grow without bound. `JsonRenderer` writes one JSON object per line, which log-collecting tools read directly; each object begins with the time, which the listing skips because it is different on every run.

examples/ch28/log-default.tiko

```
#import "std/logs"

'' std/logs holds one process-wide default logger, so any procedure,
'' in any module, can log without being handed a logger.
sub SaveSettings( byref path as ustring )
    logs.DefaultLogger( ).Info( ).AddText( "path", path ).Msg( "saved" )
end sub

dim tr as logs.TextRenderer
tr.stamp = logs.TS_NONE
dim con as logs.ConsoleSink = logs.ConsoleSink( logs.CT_STDOUT, @tr )

logs.LogInit( @con, logs.LOG_INFO )
logs.DefaultLogger( ).Info( "starting" )
SaveSettings( "settings.ini" )
logs.LogShutdown( )
```

examples/ch28/log-default.out

```
INFO   starting
INFO   saved  path=settings.ini
```

Passing a logger to every procedure that might log is tedious. The **default logger** is one logger for the whole program, and `std/logs` holds it, so there is nothing to define: **`LogInit`** attaches a sink and sets the level, **`DefaultLogger( )`** reaches it from anywhere, and **`LogShutdown`** flushes and closes its sinks. It is not called for you at exit, so a program calls it last, or risks losing its final lines.

A sink that cannot write — a full disk, a deleted folder — does not stop the program or return an error to every log call. It counts what it dropped (`DroppedCount( )`) and reports the failure through a handler that `SetLogErrorHandler` replaces. **`AsyncSink`** wraps another sink and writes from a background thread, so a slow destination does not slow the program down; its `policy` decides what happens when its queue is full — wait (`OP_BLOCK`), or drop the newest or oldest record and count it.

Every sink is safe to share between threads. Each one holds a lock while it renders and writes a record, so two threads' records never interleave inside a line — which `print #` to a shared file does not promise. And a `ConsoleSink` writes to **stderr** by default: logs are diagnostics, and a program whose output is piped into another must not mix them into the pipe. It colours the level word on a terminal and never when its output is redirected.

Terminal user interfaces

`std/tui` draws full-screen terminal programs: editors, dashboards, wizards, the menus of an installer. It works on a grid of cells, and three ideas carry the whole library.

The application is three methods. A `TuiApp` has `Init`, which sets up state; `Update`, which receives an event (a key, the mouse, a resize, a timer tick) and changes state; and `View`, which draws the screen from state. Update never draws and View never changes anything.

The screen is redrawn from scratch every frame, and only the difference is written. There is no tree of widget objects to keep in step with the data. `View` describes the whole screen each time; the library compares that with the previous frame and sends the terminal only the cells that changed. An unchanged frame costs nothing.

Widgets are values; their state is yours. A widget is built in `View` and thrown away. What must survive between frames — a list's selection, a text box's contents, which widget has the focus — lives in a state record that your app owns and the widget points at.

examples/ch28/tui-hello.tiko

```
#import "std/tui"

'' An application is three methods: Init, Update and View.
type HelloApp extends tui.TuiApp
    as int32 presses
    declare function Init( ) as tui.TuiCmd override
    declare function Update( byref e as tui.TuiEvent ) _
                            as tui.TuiCmd override
    declare sub View( byref f as tui.TuiFrame ) override
end type

function HelloApp.Init( ) as tui.TuiCmd
    presses = 0
    return tui.TuiNone( )
end function

'' Update changes state and says what should happen next.
function HelloApp.Update( byref e as tui.TuiEvent ) as tui.TuiCmd
    if e.kind = tui.TUIEVENT_KEY then
        if e.key.code = tui.TUIKEY_ESCAPE then return tui.TuiQuit( 0 )
        presses += 1
        return tui.TuiRedraw( )
    end if
    return tui.TuiNone( )
```

```
end function

'' View draws the whole screen from state, every frame.
sub HelloApp.View( byref f as tui.TuiFrame )
    dim as tui.TuiBlock b
    b.borderKind = tui.TUIBORDER_ASCII
    b.sides = tui.TUISIDE_ALL
    b.AddTitle( " hello ", tui.TuiStyleDefault( ), _
                tui.TUISIDE_TOP, tui.TUIALIGN_CENTER )
    b.Render( f, f.Area( ) )
    dim as tui.TuiParagraph p
    p.align = tui.TUIALIGN_CENTER
    p.text = "keys pressed: " & str( presses )
    p.Render( f, b.Inner( f.Area( ) ) )
end sub

'' Print what a screen buffer holds, one row per line.
sub Dump( byref scr as tui.TuiScreen )
    dim as tui.TuiBuffer ptr bf = scr.Front( )
    for y as int32 = 0 to bf->rows - 1
        dim as ustring row
        for x as int32 = 0 to bf->cols - 1
            dim as tui.TuiCell ptr c = bf->At( x, y )
            if c->width > 0 then row &= c->text
        next
        print rtrim( row )
    next
end sub

'' With no terminal: render once into a 30 x 4 screen.
dim as HelloApp app
dim as tui.TuiScreen scr
scr.Resize( 30, 4 )
app.Init( )
var first = tui.TuiRenderOnce( app, scr )
Dump( scr )

'' Feed Update a key press, as the loop would, and render again.
dim as tui.TuiEvent k
k.kind = tui.TUIEVENT_KEY
k.key.code = tui.TUIKEY_CHAR
k.key.ch = asc( "a" )
var cmd = app.Update( k )
print "Update asked to redraw: "; cbool( cmd.kind = tui.TUICMD_REDRAW )
```

```
var second = tui.TuiRenderOnce( app, scr )
Dump( scr )

'' Unchanged state renders nothing.
var third = tui.TuiRenderOnce( app, scr )
print "bytes written:"; len( first ); len( second ); len( third )

'' A real program hands the app to the loop instead:
''     tui.TuiRun( app, tui.TuiTermOptionsDefault( ) )
```

examples/ch28/tui-hello.out

```
+---------- hello -----------+
|      keys pressed: 0       |
|                            |
+----------------------------+
Update asked to redraw: true
+---------- hello -----------+
|      keys pressed: 1       |
|                            |
+----------------------------+
bytes written: 158 14 0
```

A real program ends with **`tui.TuiRun( app, tui.TuiTermOptionsDefault( ) )`**, which takes over the terminal, runs the loop — wait for events, apply them all, draw once — and restores the terminal on the way out, even after a crash. It returns a `Result`: run with its output redirected, it fails with a message instead of drawing a dashboard into a file.

The listing does what a test does instead. **`TuiRenderOnce( app, screen )`** draws one frame into a screen with no terminal at all and returns the bytes it would have sent, and `screen.Front( )` holds the cells, which `Dump` prints. `Update` is a plain method, so feeding it an event is just a call. The byte counts show the difference engine at work: the first frame draws everything, the second only the changed digit, the third nothing.

`Update` returns a **command**: `TuiNone( )`, `TuiRedraw( )`, `TuiQuit( code )`, or `TuiTick( ms )` to receive a timer event — one tick per call, so an animation re-arms it on each tick.

Layout and widgets

examples/ch28/tui-widgets.tiko

```
#import "std/tui"
```

```
'' Widget STATE outlives the frame, so it lives in the app.
type FilesApp extends tui.TuiApp
    as tui.TuiListState listState
    as tui.TuiTableState tableState
    declare function Init( ) as tui.TuiCmd override
    declare function Update( byref e as tui.TuiEvent ) _
                             as tui.TuiCmd override
    declare sub View( byref f as tui.TuiFrame ) override
end type

function FilesApp.Init( ) as tui.TuiCmd
    listState.selected = 1
    return tui.TuiNone( )
end function

function FilesApp.Update( byref e as tui.TuiEvent ) as tui.TuiCmd
    return tui.TuiNone( )
end function

function Boxed( byref title as ustring ) as tui.TuiBlock
    dim as tui.TuiBlock b
    b.borderKind = tui.TUIBORDER_ASCII
    b.sides = tui.TUISIDE_ALL
    b.AddTitle( title, tui.TuiStyleDefault( ), _
                tui.TUISIDE_TOP, tui.TUIALIGN_LEFT )
    return b
end function

sub FilesApp.View( byref f as tui.TuiFrame )
    '' Layout: a body that fills, over a status bar one row high ...
    dim as tui.TuiConstraintArray rowsC
    rowsC.Push( tui.TuiFill( ) )
    rowsC.Push( tui.TuiLength( 1 ) )
    dim as tui.TuiRectArray band = tui.TuiSplitV( f.Area( ), rowsC )

    '' ... and the body split into a 14-column list and the rest.
    dim as tui.TuiConstraintArray colsC
    colsC.Push( tui.TuiLength( 14 ) )
    colsC.Push( tui.TuiFill( ) )
    dim as tui.TuiRectArray col = tui.TuiSplitH( band[0], colsC )

    var lb = Boxed( " files " )
    lb.Render( f, col[0] )
    dim as tui.TuiList lst
```

```
    lst.items.Push( "notes.txt" )
    lst.items.Push( "todo.md" )
    lst.items.Push( "data.csv" )
    lst.state = @listState
    lst.selectPrefix = "> "
    lst.Render( f, lb.Inner( col[0] ) )

    var tb = Boxed( " sizes " )
    tb.Render( f, col[1] )
    dim as tui.TuiTable t
    t.headers.Push( "Name" ) : t.widths.Push( tui.TuiFill( ) )
    t.headers.Push( "Bytes" ) : t.widths.Push( tui.TuiLength( 6 ) )
    dim as tui.TuiRow r
    r.cells.Push( "notes.txt" ) : r.cells.Push( "1204" )
    t.rows.Push( r )
    r.cells.Clear( )
    r.cells.Push( "todo.md" ) : r.cells.Push( "88" )
    t.rows.Push( r )
    t.state = @tableState
    t.showHeader = true
    t.Render( f, tb.Inner( col[1] ) )

    dim as tui.TuiStatusBar sb
    sb.Add( " q quit ", tui.TuiStyleDefault( ), tui.TUIALIGN_LEFT )
    sb.Add( "2/3 ", tui.TuiStyleDefault( ), tui.TUIALIGN_RIGHT )
    sb.Render( f, band[1] )
end sub

dim as FilesApp app
dim as tui.TuiScreen scr
scr.Resize( 40, 7 )
app.Init( )
tui.TuiRenderOnce( app, scr )

dim as tui.TuiBuffer ptr bf = scr.Front( )
for y as int32 = 0 to bf->rows - 1
    dim as ustring row
    for x as int32 = 0 to bf->cols - 1
        if bf->At( x, y )->width > 0 then row &= bf->At( x, y )->text
    next
    print rtrim( row )
next
```

examples/ch28/tui-widgets.out

```
+ files -----++ sizes -----------------+
|  notes.txt ||Name              Bytes |
|> todo.md   ||notes.txt         1204  |
|  data.csv  ||todo.md           88    |
|            ||                       |
+------------++-----------------------+
 q quit                          2/3
```

Layout divides a rectangle by constraints: `TuiLength( n )` for exactly `n` cells, `TuiPercent`, `TuiRatio`, `TuiMin_` and `TuiMax_`, and `TuiFill( )` for a share of what is left. `TuiSplitV` stacks the pieces and `TuiSplitH` puts them side by side, and splitting a piece again builds any screen. The result is the same for the same sizes on every platform.

The listing uses four widgets: **`TuiBlock`** (a border with titles, whose `Inner` is the space inside), **`TuiList`**, **`TuiTable`** with columns sized by the same constraints, and **`TuiStatusBar`**. The rest of the set includes `TuiParagraph`, `TuiTabs`, `TuiGauge` and `TuiSpinner`; the form widgets `TuiTextInput`, `TuiTextArea`, `TuiCheckbox`, `TuiRadioGroup` and `TuiSelect`, with a `TuiFocusRing` that moves the focus on Tab; the overlays `TuiPopup`, `TuiMessageBox` and `TuiMenuBar`; `TuiTree`; and the charts `TuiGraph`, `TuiSparkline`, `TuiBarChart` and `TuiMeter`.

The listings draw `TUIBORDER_ASCII` borders so that the output prints anywhere. A real program usually wants `TUIBORDER_ROUNDED` or `TUIBORDER_PLAIN`, drawn with the box-drawing characters, and colour: a `TuiStyle` sets foreground, background and attributes, in 24-bit colour that the library steps down to what the terminal supports.

> **PITFALL**
> **A focusable widget must outlive the frame.** A focus ring holds pointers to widgets, so a `TuiTextInput` declared inside `View` is gone by the time Tab moves to it, and the form cannot be navigated. Make every widget that takes the focus a field of the app, as the state records are.

SQLite

SQLite is a database in a library: no server, one file per database (or none, in memory), and the SQL language. tiko ships it as **`vendor/sqlite3`**, one of the toolchain's **vendor packages**: a C library with its header already translated into tiko and its compiled code beside it, reached with one `#import`. What you call is SQLite's own C interface, so every function is written the way SQLite's documentation writes it: `sqlite3_open`, not `sqlite3.sqlite3_open`. Chapter 30 shows how a package like this is made, for any C library.

examples/ch28/contacts-db.tiko

```
'' SQLite's own C interface, from the toolchain's vendor package.
#import "vendor/sqlite3"

'' SQLite reports failure as a result code; this turns one into a stop.
sub Check( byval rc as int32, byval db as sqlite3 ptr )
    if rc <> SQLITE_OK andalso rc <> SQLITE_DONE then
        print "SQLite error"; rc; ": "; *sqlite3_errmsg( db )
        end 1
    end if
end sub

dim db as sqlite3 ptr
Check( sqlite3_open( ":memory:", @db ), db )
Check( sqlite3_exec( db, "create table contact( " & _
    "id integer primary key, name text not null, email text )", _
    nullptr, nullptr, nullptr ), db )

'' Values go in as parameters, never pasted into the SQL text.
dim as string names( ... ) = { "Grace", "Ada", "Edsger" }
dim as string emails( ... ) = { "grace@example.org", _
                                "ada@example.org", "" }
dim ins as sqlite3_stmt ptr
Check( sqlite3_prepare_v2( db, _
    "insert into contact( name, email ) values( ?, ? )", _
    -1, @ins, nullptr ), db )
for i as int32 = 0 to ubound( names )
    sqlite3_bind_text( ins, 1, names( i ), -1, SQLITE_TRANSIENT )
    sqlite3_bind_text( ins, 2, emails( i ), -1, SQLITE_TRANSIENT )
    Check( sqlite3_step( ins ), db )
    sqlite3_reset( ins )
next
sqlite3_finalize( ins )
print "last id:"; sqlite3_last_insert_rowid( db )

'' A query is stepped through one row at a time.
dim sel as sqlite3_stmt ptr
Check( sqlite3_prepare_v2( db, "select id, name, email from " & _
    "contact where email <> '' order by name", -1, @sel, nullptr ), db )
do while sqlite3_step( sel ) = SQLITE_ROW
    print sqlite3_column_int64( sel, 0 ); "  "; _
          *cast( zstring ptr, sqlite3_column_text( sel, 1 ) ); "  "; _
          *cast( zstring ptr, sqlite3_column_text( sel, 2 ) )
loop
```

```
sqlite3_finalize( sel )

'' A transaction: everything in it happens, or nothing does.
Check( sqlite3_exec( db, "begin", nullptr, nullptr, nullptr ), db )
Check( sqlite3_exec( db, "delete from contact", _
    nullptr, nullptr, nullptr ), db )
Check( sqlite3_exec( db, "rollback", nullptr, nullptr, nullptr ), db )
Check( sqlite3_prepare_v2( db, "select count(*) from contact", _
    -1, @sel, nullptr ), db )
sqlite3_step( sel )
print "after the rollback:"; sqlite3_column_int( sel, 0 ); " contacts"
sqlite3_finalize( sel )

'' A failure is a result code, and the connection keeps the message.
dim errMsg as zstring ptr
if sqlite3_exec( db, "insert into contact( email ) values( 'x' )", _
        nullptr, nullptr, @errMsg ) <> SQLITE_OK then
    print "refused: "; *errMsg
    sqlite3_free( errMsg )
end if
sqlite3_close( db )
```

examples/ch28/contacts-db.out

```
last id: 3
 2  Ada  ada@example.org
 1  Grace  grace@example.org
after the rollback: 3 contacts
refused: NOT NULL constraint failed: contact.name
```

A connection is a **`sqlite3 ptr`**. `sqlite3` is a type SQLite declares but never defines, so a program can only hold a pointer to one, and `sqlite3_open` fills it in through its address, `@db`. The name `:memory:` opens a private database that lives as long as the connection; any other name is a file, created when it does not exist. `sqlite3_close` closes it.

Almost every function returns a **result code**: `SQLITE_OK` for success, anything else for a failure, and then `sqlite3_errmsg( db )` says what went wrong. Nothing raises a runtime error, so a code that is ignored is a failure that is ignored. The listing's `Check` turns every failure into a message and a stop, which is enough for a short program; a larger one decides call by call.

`sqlite3_exec` runs SQL that returns no rows. Its last three arguments are a callback that would receive rows, an argument for that callback, and the address of an error message. Given that address, as the last statement gives it, SQLite stores its message

there, and the program frees it with `sqlite3_free`. That is how the last line of output arrives: a `not null` constraint refusing a row.

Prepared statements

Everything else goes through a **statement**, in four steps:

1. **`sqlite3_prepare_v2`** compiles the SQL once, into a `sqlite3_stmt ptr`.
2. **`sqlite3_bind_text`**, `sqlite3_bind_int64`, `sqlite3_bind_null` and their siblings give a value to each `?`, counting from 1.
3. **`sqlite3_step`** runs it: `SQLITE_ROW` for each row of a result, `SQLITE_DONE` at the end. On a row, `sqlite3_column_text`, `sqlite3_column_int64` and the others read its columns, counting from
   0. **`sqlite3_reset`** makes the statement ready to run again

   with new values.
4. **`sqlite3_finalize`** frees it.

> PITFALL
> **Never build SQL by pasting values into the text.** `"... where name = '" & name & "'"` breaks on the first name with an apostrophe in it, and lets anyone who controls the value rewrite the query. Write `?` where each value goes and bind the values: they travel separately from the SQL and cannot change what it means.

The last argument of `sqlite3_bind_text` says how long SQLite may keep the text. `SQLITE_TRANSIENT` asks it to take a copy, which is always safe; `SQLITE_STATIC` promises the text will not change while the statement uses it, and saves the copy. The `-1` before it means "the text ends at its zero byte".

Text crosses as UTF-8. SQLite's `_text` functions take and return C's `char *`, holding UTF-8. A `ustring` passed where the C function wants a `const char *` is converted to UTF-8 for the call; a `string`, tiko's byte string, passes its bytes as they are. These listings keep their values in `string` because they are bytes that go to SQLite and come back unchanged. The other way, `sqlite3_column_text` returns a pointer to bytes SQLite owns, valid until the next step, and `*cast( zstring ptr, … )` copies them into a string before then; a `string` holding UTF-8, assigned to a `ustring`, is read as UTF-8. Chapter 35's ledger keeps its text in `ustring` and works this way.

A transaction groups statements: `begin`, then `commit` to keep everything or `rollback` to undo everything, as the listing does with its `delete`. They are SQL like any other, run with `sqlite3_exec`. A transaction is also how many inserts become fast: each statement outside one is its own transaction, written to disk separately.

Importing CSV

examples/ch28/people.csv

```
name,city,phone
"Lovelace, Ada",London,555-0100
Grace Hopper,"New York",
"Linus ""the"" Torvalds",Portland,555-0199
```

examples/ch28/csv-import.tiko

```
'' Loading a CSV file: std/strings parses it, SQLite stores it.
#import "vendor/sqlite3"
#import "std/strings"

'' The file's bytes, as they are: UTF-8, which is what SQLite takes.
dim as string text
dim f as int32 = freefile
open "people.csv" for binary access read as #f
text = space( lof( f ) )
get #f, , text
close #f
var rows = strings.SplitCsv( text )

dim db as sqlite3 ptr
sqlite3_open( ":memory:", @db )
sqlite3_exec( db, "create table people( name text, city text, " & _
    "phone text )", nullptr, nullptr, nullptr )

'' Row 0 is the header. Every other row goes in, in one transaction.
dim ins as sqlite3_stmt ptr
sqlite3_prepare_v2( db, "insert into people values( ?, ?, ? )", _
    -1, @ins, nullptr )
sqlite3_exec( db, "begin", nullptr, nullptr, nullptr )
for r as int32 = 1 to rows.Count( ) - 1
    for c as int32 = 0 to 2
        dim as string value = rows[ r ].fields[ c ]
        if len( value ) = 0 then
            sqlite3_bind_null( ins, c + 1 )   '' an empty field is NULL
        else
            sqlite3_bind_text( ins, c + 1, value, -1, SQLITE_TRANSIENT )
        end if
    next
    sqlite3_step( ins )
    sqlite3_reset( ins )
next
```

```
sqlite3_exec( db, "commit", nullptr, nullptr, nullptr )
sqlite3_finalize( ins )
print rows.Count( ) - 1; " rows imported"

dim sel as sqlite3_stmt ptr
sqlite3_prepare_v2( db, "select name, city, phone from people " & _
    "order by name", -1, @sel, nullptr )
do while sqlite3_step( sel ) = SQLITE_ROW
    dim as string phone = "(none)"
    if sqlite3_column_type( sel, 2 ) <> SQLITE_NULL then
        phone = *cast( zstring ptr, sqlite3_column_text( sel, 2 ) )
    end if
    print *cast( zstring ptr, sqlite3_column_text( sel, 0 ) ); " | "; _
          *cast( zstring ptr, sqlite3_column_text( sel, 1 ) ); " | "; _
          phone
loop
sqlite3_finalize( sel )
sqlite3_close( db )
```

examples/ch28/csv-import.out

```
 3 rows imported
Grace Hopper | New York | (none)
Linus "the" Torvalds | Portland | 555-0199
Lovelace, Ada | London | 555-0100
```

Reading CSV properly is Chapter 9's **`strings.SplitCsv`**: quoted fields, commas and doubled quotes inside them, line breaks inside quotes. Given a `string`, it returns rows whose fields are `string`s as well, the same UTF-8 bytes the file holds, so they are bound with no conversion. The listing reads the whole file as bytes, takes row 0 as the header, binds an empty field as SQL's `NULL` with `sqlite3_bind_null`, and loads every row inside one transaction. `sqlite3_column_type` tells a `NULL` from text when it is read back.

Static or dynamic

`#import "vendor/sqlite3"` links SQLite **dynamically**: the program calls into SQLite's shared library, `sqlite3.dll` on Windows, and **tiko copies that library beside the program** when it builds it, so the program starts wherever the two travel together. To have SQLite's code linked into the program instead, which is then one file, say so on the import:

```
#import "vendor/sqlite3" link static
```

Nothing else in the program changes. A program links a library one way only: two imports asking for different modes are an error, `TK00157`.

The finance library

Chapter 27 used `Decimal`, `Currency` and loan schedules from **`#import "std/money"`**. The same module carries the rest of a finance library, one source file per subject, all built on `Decimal`:

File	Covers
`fincore.tiko`	interest, rates and the conventions they are quoted in; compounding and payment frequency; day counts
`fintvm.tiko`	the time value of money: present and future value, annuities
`finloan.tiko`	loans, mortgages and amortization schedules
`fincash.tiko`	cash flows: net present value, internal rate of return, payback
`findeprec.tiko`	depreciation, by six methods
`finport.tiko`	portfolio returns and risk: volatility, drawdown, Sharpe and Sortino ratios
`finsolve.tiko`	the one root finder behind every implied rate
`finerror.tiko`	`FinanceError`, which every fallible call returns in its `Result`

Each calculation is exact where it can be — growth over a whole number of periods, every sum and product of money — and uses `float64` only where a fractional power or a solver makes that impossible, converting back to `Decimal` at a fixed scale. The comments in `pkg/std/money` document every function, and all of them are reached as `money.`.

What it costs

- **A disabled log level costs one comparison.** An enabled record costs its fields, the rendering, and a lock per sink.
- **A frame costs its drawing plus a cell-by-cell comparison** of the whole screen; an unchanged frame writes nothing, and an idle interface uses no processor time.

- **Each SQL statement outside a transaction is a separate write to disk.** Thousands of inserts belong in one transaction; a prepared statement saves compiling the SQL each time.
- **`vendor/sqlite3` costs a second file or a larger program.** Linked dynamically, SQLite's shared library, 3.5 MB on Windows, travels beside the program; linked statically, the program grows by about 1.1 MB. Either way nothing needs installing on the machine that runs it.

Summary

- `#import "std/logs"`: a `Logger` with `MinLevel` and sinks; `Info( "text" )`, or `Warn( ).AddText( … ).AddInt( … ).Msg( "text" )` for fields; `WithField` for a child logger.
- Sinks: `ConsoleSink` (stderr by default), `FileSink` with rotation, `AsyncSink`; renderers: `TextRenderer`, `JsonRenderer`. The logger does not own its sinks.
- `LogInit`, `DefaultLogger( )` and `LogShutdown` for the one program-wide logger, which `std/logs` holds.
- `#import "std/tui"`: a `TuiApp` with `Init`, `Update` and `View`; `TuiRun` drives it, and `TuiRenderOnce` draws a frame with no terminal. Layout with constraints, widgets as values, state in the app.
- `#import "vendor/sqlite3"`: SQLite's own C interface, unqualified: `sqlite3_open`, `sqlite3_exec`; statements with `sqlite3_prepare_v2`, `?` and `sqlite3_bind_*`, then `sqlite3_step`, `sqlite3_column_*` and `sqlite3_finalize`. Every call returns a result code, and `sqlite3_errmsg` explains a failure. Text is UTF-8, so values travel as `string`.
- `link static` or `link dynamic` on the import chooses how SQLite is linked; dynamic, the default, copies the shared library beside the program. `strings.SplitCsv` reads CSV to load.
- `std/money` also carries rates, the time value of money, loans, cash flows, depreciation and portfolio measures.

Exercises

1. Store a list of sales in a table, then print each region's total and number of sales, largest first, with one SQL query. {solution="ex28-01″}
2. Set a logger's minimum level from a level name given on the command line, falling back to `INFO` for a name that is not a level. {solution="ex28-02″}
3. Add a `FileSink` to `log-demo.tiko` beside the console, so every record goes to both. What happens to the file if the program is run twice?
4. Make `tui-widgets.tiko` respond to the Down key by moving the list's selection, and check it the way `tui-hello.tiko` checks its key press.

5. Import a CSV file of your own, then, inside a transaction, delete the rows that fail a check, and roll back if more than half of them would go. `sqlite3_changes( db )` says how many rows the last statement changed.

Interfacing with C

Most of the code a program could use is written in C: the operating system's own interfaces, the C runtime, compression, databases, graphics, cryptography. tiko calls C directly, with no wrapper layer between: a C function is declared, and called like any other. This chapter covers declaring C functions by hand and having the compiler read a C header instead, matching C's types and layouts, passing strings and callbacks across, calling conventions, linking a library — including one shipped inside the project's own tree — and loading one while the program runs.

Every listing here runs on both targets. Several print different answers on each, and those differences are the point.

Declaring a C function

examples/ch29/declare.tiko

```
'' Two ways to declare a C function. The C runtime is always linked,
'' so neither needs a library named.

'' One at a time: 'cdecl' for C's convention, 'alias' for its exact
'' name, so the tiko name can differ from it.
declare function CLength cdecl alias "strlen" _
    ( byval s as const zstring ptr ) as uinteger

'' A block: everything inside is cdecl, and keeps its name as written.
extern "C"
    declare function toupper( byval c as int32 ) as int32
    declare function abs_ alias "abs" ( byval n as int32 ) as int32
end extern

dim as zstring * 16 word = "interface"
print "strlen: "; CLength( @word )

for i as int32 = 0 to len( word ) - 1
    word[ i ] = toupper( word[ i ] )
next
print "toupper: "; word

print "abs:"; abs_( -42 )
```

examples/ch29/declare.out

```
strlen: 9
toupper: INTERFACE
abs: 42
```

A C function is declared with `declare`, like a tiko one, plus three things C needs:

- **`cdecl`**: C's calling convention. On x86-64 every convention is the same, but saying `cdecl` keeps the declaration honest and matches what the C header says.
- **`alias "name"`**: the exact name the linker looks for. tiko's names are case-insensitive and C's are not, so `alias` fixes the spelling, and it also lets the tiko name differ: `CLength` above is C's `strlen`.
- **`lib "name"`** (optional): the library the function lives in, which is then linked. The C runtime is always linked, so the listing names none.

`extern "C" … end extern` declares a block at once. Everything inside is `cdecl`, and every name keeps the exact spelling it was written with, so `toupper` links to C's `toupper` with no `alias`. Inside the block, `alias` still renames: `abs_` is C's `abs`, renamed because `abs` is a tiko function already. `extern "C" lib "name"` names a library for the whole block.

A declaration is a promise the compiler cannot check: it believes every parameter type and the result type, and a mistake shows up when the program runs, not when it compiles. Copy the types from the C header carefully, which is the first thing the next section is about — or let the compiler read the header itself, which is the section after it.

Matching C's types

examples/ch29/ctypes.tiko

```
#include once "crt/long.bi"
#include once "crt/stddef.bi"

'' C's types do not have the same sizes everywhere.
print "int       "; sizeof( int32 )
print "long      "; sizeof( clong )
print "long long "; sizeof( int64 )
print "size_t    "; sizeof( size_t )
print "void *    "; sizeof( any ptr )
print "wchar_t   "; sizeof( wchar_t )
print "wstring character "; sizeof( wstring * 1 )

'' The compiler says which C runtime it is building against.
#ifdef __FB_MSVC__
    print "C runtime: Microsoft's"
```

```
#else
    print "C runtime: the system's"
#endif
```

On Windows:

`examples/ch29/ctypes.out`

```
int        4
long       4
long long  8
size_t     8
void *     8
wchar_t    2
wstring character  2
C runtime: Microsoft's
```

On Linux:

`examples/ch29/ctypes.linux-x86_64.out`

```
int        4
long       8
long long  8
size_t     8
void *     8
wchar_t    4
wstring character  4
C runtime: the system's
```

On macOS:

`examples/ch29/ctypes.macos-arm64.out`

```
int        4
long       8
long long  8
size_t     8
void *     8
wchar_t    4
wstring character  4
C runtime: the system's
```

Most C types have one tiko equivalent on both platforms:

C	tiko
`char`, `signed char`, `unsigned char`	`byte` / `ubyte`
`short`, `unsigned short`	`int16` / `uint16`
`int`, `unsigned int`	`int32` / `uint32`
`long long`, `unsigned long long`	`int64` / `uint64`
`float`, `double`	`float32` / `float64`
`size_t`, `intptr_t`	`uinteger` / `integer`, or `size_t` from `crt/stddef.bi`
`T *`	`T ptr`; `void *` is `any ptr`
`const char *`	`const zstring ptr`
a function pointer	a `type` alias for a `function cdecl( … )` or `sub cdecl( … )`

Two do not:

- **C's `long` is 4 bytes on Windows and 8 on Linux and macOS.** Use `clong` and `culong` from `crt/long.bi`, which follow the platform. Writing `int32` for a `long` works on Windows and breaks everywhere else.
- **`wchar_t` is 2 bytes on Windows (UTF-16) and 4 on Linux and macOS (UTF-32).** tiko's `wstring` follows the same rule, so a `wstring ptr` is a `wchar_t *` everywhere.

> **PITFALL**
> **`long` is not `int32`.** A C function returning `long` declared `as int32` loses the top half of every value on Linux and macOS, and nothing complains — the Windows build, where the two agree, works. The solution to exercise 2 uses `strtol`, which returns `long`, for that reason.

`__FB_MSVC__` is defined when the program is built against Microsoft's C runtime — the default on Windows, the `windows-x86_64` target. `__FB_WIN32__`, `__FB_LINUX__` and `__FB_DARWIN__` name the platform. The `crt/` headers (`crt/string.bi`, `crt/stdlib.bi`, `crt/math.bi` and the rest, or all of them at once through `crt.bi`) declare the C runtime by hand, ready to use.

Structs and their layout

`examples/ch29/layout.tiko`

```
'' A struct C will read must match C's layout byte for byte.

'' Natural alignment: 'value' starts on a 4-byte boundary, as in C.
type Natural
```

```
    as ubyte tag
    as int32 value
end type

'' 'field = 1' packs it, as C's #pragma pack(1) does.
type Packed field = 1
    as ubyte tag
    as int32 value
end type

print "Natural: size"; sizeof( Natural ); _
      ", value at"; offsetof( Natural, value )
print "Packed:  size"; sizeof( Packed ); _
      ", value at"; offsetof( Packed, value )

'' A fixed-size zstring is an inline char array, like C's char[8].
type Record
    as zstring * 8 name_
    as int32 id
end type
print "Record:  size"; sizeof( Record ); _
      ", id at"; offsetof( Record, id )
```

examples/ch29/layout.out

```
Natural: size 8, value at 4
Packed:  size 5, value at 1
Record:  size 12, id at 8
```

A `type` lays its fields out the way C lays out a `struct` on the same platform: each field on a boundary of its own size, with padding between where needed, and the whole rounded up to its largest alignment. So a `type` with the same fields in the same order as a C struct has the same size and the same offsets, and a pointer to one can be handed to C.

`field = 1` packs a type with no padding, as C's `#pragma pack(1)` does; `field = 2` and `field = 4` limit the alignment the same way. Use them only to match a C struct that is packed — a packed type is slower to use, and an unpacked C struct read through a packed type is garbage. A fixed-size `zstring * n` is an inline array of `n` characters, like C's `char name[n]`; a `ustring` field is not, since it holds a pointer to the text, and never belongs in a struct C will read. `offsetof( T, field )` checks an offset against the C header.

Strings across the boundary

examples/ch29/cstrings.tiko

```
extern "C"
    declare function strlen( byval s as const zstring ptr ) as uinteger
    declare function strchr( byval s as const zstring ptr, _
                             byval c as int32 ) as zstring ptr
    declare function malloc( byval n as uinteger ) as any ptr
    declare sub free( byval p as any ptr )
    declare function strcpy( byval dst as zstring ptr, _
                             byval src as const zstring ptr ) _
                             as zstring ptr
end extern

'' Into C: a string's own bytes, ending in a zero, through strptr.
dim as string s = "key=value"
print "length from C: "; strlen( strptr( s ) )

'' Out of C: a pointer into memory the C side owns. '*p' copies the
'' characters into a tiko string at once.
dim as zstring ptr eq = strchr( strptr( s ), asc( "=" ) )
if eq <> nullptr then
    dim as string rest = *( eq + 1 )
    print "after '=': "; rest
end if

'' Memory C allocates, C must free.
dim as zstring ptr copy_ = malloc( len( s ) + 1 )
strcpy( copy_, strptr( s ) )
dim as ustring u = *copy_
free( copy_ )
print "copied into a ustring: "; u
```

examples/ch29/cstrings.out

```
length from C: 9
after '=': value
copied into a ustring: key=value
```

C's strings are bytes ending in a zero byte, and that is how a tiko `string` and a `zstring` are stored. **Into C**: `strptr( s )` is a pointer to a `string`'s own bytes, valid until the string next changes; `@z` does the same for a `zstring`. **Out of C**: a `zstring ptr` points at C's memory, and `*p` reads the characters into a tiko string, which copies them. Copy at once; C may reuse or free that memory.

Memory C allocates, C must free. The listing copies into memory from `malloc` and then reads it into a `ustring` — a copy — before `free`. A pointer from a C library

comes with rules about who frees it, and the library's documentation says which; the C runtime's `free` is right only for memory from the C runtime's `malloc`.

A `ustring` holds UTF-8, so `*p` from a C function that returns UTF-8 gives the right text. A C interface that takes `wchar_t *` takes a `wstring ptr` on both platforms; on Windows that is UTF-16, which is what the `W` functions of the Windows API expect.

Callbacks

examples/ch29/callbacks.tiko

```
'' C's qsort calls back into the program to compare two elements.
type Compare as function cdecl( byval a as const any ptr, _
                                byval b as const any ptr ) as int32

extern "C"
    declare sub qsort( byval base_ as any ptr, byval n as uinteger, _
                       byval size as uinteger, byval cmp as Compare )
end extern

'' A named procedure, cdecl, with exactly the parameters C passes.
function Ascending cdecl( byval a as const any ptr, _
                          byval b as const any ptr ) as int32
    return *cptr( const int32 ptr, a ) - *cptr( const int32 ptr, b )
end function

dim nums( 0 to 5 ) as int32 = { 42, 7, 19, 3, 88, 21 }
qsort( @nums( 0 ), 6, sizeof( int32 ), @Ascending )
for i as int32 = 0 to 5
    print nums( i );
next
print

'' A lambda that captures nothing is a plain function too.
qsort( @nums( 0 ), 6, sizeof( int32 ), _
       function cdecl( byval a as const any ptr, _
                       byval b as const any ptr ) as int32
           return *cptr( const int32 ptr, b ) - _
                  *cptr( const int32 ptr, a )
       end function )
for i as int32 = 0 to 5
    print nums( i );
next
print
```

examples/ch29/callbacks.out

```
 3 7 19 21 42 88
 88 42 21 19 7 3
```

A C function that takes a function pointer calls back into the program: `qsort` calls its comparison function once per comparison. A tiko procedure can be the callback if it is **`cdecl`** and takes exactly the parameters C passes, with the right types. `@Ascending` is its address.

A **lambda that captures nothing** (Chapter 21) is an ordinary function, so it converts to a function pointer and can be passed straight to C, as the second call does. A lambda that captures variables cannot: it carries the captured values with it, and C has nowhere to put them.

C's way to pass state into a callback is the extra `void *` parameter that most callback APIs have — `user`, `context` or `arg`. The program passes a pointer to its own data, and the callback casts it back. The geom listings below use one to count visits.

Variadic functions

`examples/ch29/variadic.tiko`

```
#include once "crt/stddef.bi"

'' Calling a variadic C function: declare the fixed part, then '...'.
extern "C"
    declare function snprintf( byval buf as zstring ptr, _
                               byval n as size_t, _
                               byval fmt as const zstring ptr, _
                               ... ) as int32
end extern

dim as zstring * 64 buf
var n = snprintf( @buf, 64, "%d items at %.2f each: %s", _
                  3, 4.5, "ok" )
print buf; " ("; n; " characters)"

'' Writing one: cva_list walks the arguments after the fixed ones.
'' The caller has to say how many there are -- here, the first one.
function Total cdecl( byval count as int32, ... ) as int32
    dim args as cva_list
    cva_start( args, count )
    dim as int32 sum = 0
    for i as int32 = 1 to count
        sum += cva_arg( args, int32 )
    next
```

```
    cva_end( args )
    return sum
end function

print "Total:"; Total( 4, 10, 20, 30, 40 )

'' A cva_list is a pointer on Windows and a 24-byte record on Linux.
dim probe as cva_list
print "cva_list is"; sizeof( probe ); " bytes"
```

On Windows:

examples/ch29/variadic.out

```
3 items at 4.50 each: ok ( 24 characters)
Total: 100
cva_list is 8 bytes
```

On Linux the last line is:

```
cva_list is 24 bytes
```

A **variadic** C function, like `printf` and `snprintf`, is declared with its fixed parameters followed by `...`. The arguments after the fixed ones are passed by C's rules: integers as at least an `int`, `float32` as a `float64`. Nothing checks them against the format string, so a `%d` given a `float64` prints nonsense.

A tiko procedure can be variadic too, for C to call or for tiko's own use: declare it `cdecl` with `...`, then walk the arguments with a **`cva_list`**: `cva_start( list, last )` starts after the last fixed parameter, `cva_arg( list, T )` takes the next one as type `T`, and `cva_end( list )` finishes. Nothing says how many arguments there are or what types they have; the fixed parameters must, as `count` does here and the format string does for `printf`.

A `cva_list` is 8 bytes on Windows (a pointer) and 24 on Linux (a small record), so it is not something to store or pass by value across platforms. Pass it to C's `v` functions — `vsnprintf` and the rest — which take a `va_list`.

`sizeof( cva_list )` names the type directly and gives the same 8 and 24.

Calling conventions and names

On 32-bit x86, C had several calling conventions — `cdecl`, `stdcall`, `pascal`, `fastcall` and `thiscall` — that differed in which side cleaned up the stack and in what order the arguments went. x86-64 has **one** per platform: Microsoft's on Windows and the

System V convention on Linux. tiko compiles only for x86-64, so `cdecl` and `stdcall` mean the same thing and can be used interchangeably; a Windows header's `WINAPI` (which is `stdcall`) and a plain `cdecl` describe the same call.

The exception is **`pascal`**, which passes its arguments in reverse order on every platform. Assigning a `pascal` procedure to a `cdecl` function pointer is warning `TKO0830`, because called through the pointer, its arguments arrive backwards.

Each convention also decides the **name** the linker sees:

examples/ch29/mangle.tiko

```
'' Three linkages for one kind of function. --dump-mangle shows the
'' symbol each becomes.
function Plain( byval x as int32 ) as int32
    return x + 1
end function

extern "C"
    function c_style( byval x as int32 ) as int32
        return x + 2
    end function
end extern

extern "C++"
    namespace geo
        function cpp_style( byval x as int32 ) as int32
            return x + 3
        end function
    end namespace
end extern

print Plain( 1 ); c_style( 1 ); geo.cpp_style( 1 )
```

examples/ch29/mangle.out

```
 2 3 4
```

`tiko --dump-mangle mangle.tiko` prints the symbol each procedure becomes, the same on both targets:

```
PLAIN
c_style
_ZN3geo9cpp_styleEi
```

A plain procedure's symbol is its name in capitals, because tiko's names are case-insensitive. Inside `extern "C"` the name is kept exactly as written, which is what

a C library expects to find. Inside **`extern "C++"`** it is *mangled* the way a C++ compiler would, with the namespace and the parameter types folded into the symbol, so a tiko declaration can link to a C++ function that is not `extern "C"`: `geo::cpp_style( int )` is `_ZN3geo9cpp_styleEi`. Only functions link this way; C++ classes, templates and exceptions do not cross. `--dump-mangle` is the way to find out why a link fails with an unresolved symbol: compare its spelling with what the library exports.

Linking

A declaration says a function exists; the linker needs the library that contains it. There are three ways to name one:

- **`#inclib "name"`** in the source links the library `name`: `libname.a` or `libname.so` on Linux, `name.lib` on Windows. `#libpath "dir"` adds a directory to search.
- **`-l name`** and **`-p dir`** on the command line do the same from outside.
- **`lib "name"`** on a declaration, or on an `extern` block, links the library too, as if `#inclib` had been written.

The linker searches the directories from `-p` and `#libpath` first, in the order given, then the toolchain's own library directory, and **the current directory last**, so a stray library there cannot replace the runtime that was shipped with the compiler. `#libpath` directories are relative to where the compiler runs, not to the source file — which is what the next section fixes.

A library inside the project: #foreignlib

examples/ch29/geom-hand.tiko

```
'' A hand-written binding: the library, and a declaration per function.
#foreignlib GEOM static "geom/geom"

type geom_point
    as float64 x, y
end type

type geom_visit as sub cdecl( byval p as const geom_point ptr, _
                              byval user as any ptr )

extern "C" lib GEOM
    declare function geom_version( ) as const zstring ptr
    declare function geom_distance( byval a as geom_point, _
                                    byval b as geom_point ) as float64
    declare function geom_centroid( byval pts as const geom_point ptr, _
                                    byval n as int32, _
                                    byval out_ as geom_point ptr ) _
```

```
                                    as int32
    declare sub geom_each( byval pts as const geom_point ptr, _
                           byval n as int32, byval fn as geom_visit, _
                           byval user as any ptr )
end extern

'' A callback C will call: cdecl, and only the parameters C passes.
sub Visit cdecl( byval p as const geom_point ptr, _
                 byval user as any ptr )
    dim as int32 ptr count = user
    *count += 1
    print "  visit"; *count; ": ("; p->x; ","; p->y; " )"
end sub

print *geom_version( )

dim pts( 0 to 2 ) as geom_point = _
    { ( 0, 0 ), ( 4, 0 ), ( 4, 3 ) }
print "distance:"; geom_distance( pts( 0 ), pts( 2 ) )

dim c as geom_point
if geom_centroid( @pts( 0 ), 3, @c ) then
    print "centroid: ("; c.x; ","; c.y; " )"
end if

dim as int32 seen = 0
geom_each( @pts( 0 ), 3, @Visit, @seen )
print "visited"; seen
```

examples/ch29/geom-hand.out

```
geom 1.0
distance: 5
centroid: ( 2.666666666666667, 1 )
  visit 1: ( 0, 0 )
  visit 2: ( 4, 0 )
  visit 3: ( 4, 3 )
visited 3
```

`geom` is a small C library that ships with this chapter: `geom.h` and `geom.c`, a `build.sh` that builds it, and one prebuilt static library per platform:

```
geom/geom.h
geom/geom.c
geom/build.sh
```

```
geom/geom_windows_x86_64.lib
geom/geom_linux_x86_64.a
```

`#foreignlib GEOM static "geom/geom"` binds the name `GEOM` to that library. Three things set it apart from `#inclib`:

- **The path is relative to the file the directive is in**, not to wherever the compiler is run. A binding can sit in a subdirectory of a project, with its libraries beside it, and work from any working directory.
- **The platform picks the file.** The compiler adds the target to the name, most specific first: `geom_windows_x86_64`, `geom_windows`, `geom_x86_64`, then plain `geom`, each with the platform's extensions. One directive serves every platform the project ships libraries for, with no `#ifdef`.
- **`static` or `dynamic` must be said**, and limits which files can match: `.a` and `.lib` for static; `.so`, `.dll`, `.dll.a` and `.lib` for dynamic. On Windows a `.lib` can be either kind — a static library or the import library of a DLL — so it matches both.

The bound name then goes where a library name would: `extern "C" lib GEOM`. A `#foreignlib` library is linked whether or not anything names it, as `#inclib`'s is.

A spec beginning **`system:`** names an installed library instead of a file: `"system:m"` links Linux's maths library, exactly as `#inclib "m"` would, and its `static` or `dynamic` is not checked.

When nothing matches, the error lists every file that was tried. On Windows:

examples/ch29/nolib.tiko

```
'' A library spec that matches no file: the error lists every name
'' that was tried, most specific first.
#foreignlib SHAPES static "geom/shapes"

extern "C" lib SHAPES
    declare function shapes_version( ) as const zstring ptr
end extern

print *shapes_version( )
```

```
nolib.tiko(3,1): error TKO0329: #foreignlib found no matching library
file: SHAPES static geom/shapes; tried:
    geom/shapes_windows_x86_64.a
    geom/shapes_windows_x86_64.lib
    geom/shapes_windows.a
    geom/shapes_windows.lib
    geom/shapes_x86_64.a
```

```
    geom/shapes_x86_64.lib
    geom/shapes.a
    geom/shapes.lib
```

(The first line is one line; it is broken here to fit the page.)

The other `#foreignlib` errors are about the directive itself: `TKO0327` and `TKO0328` when the name, `static`/`dynamic` or the quoted spec is missing, and `TKO0330` when one name is bound twice to different libraries.

The geom listing passes a `geom_point` **by value** to `geom_distance`. That works because the `type` has C's layout, and tiko passes it by the platform's rules for a 16-byte struct — by hidden reference on Windows, in two floating-point registers on Linux — exactly as a C compiler would.

Reading the C header: #cinclude

examples/ch29/geom-cinclude.tiko

```
'' The same library, with the declarations taken from its own header.
#cinclude once "geom/geom.h"
#foreignlib GEOM static "geom/geom"

sub Visit cdecl( byval p as const geom_point ptr, _
                 byval user as any ptr )
    print "  ("; p->x; ","; p->y; " )"
end sub

print *geom_version( ); ", major version"; GEOM_VERSION_MAJOR

dim pts( 0 to 2 ) as geom_point = _
    { ( 0, 0 ), ( 4, 0 ), ( 4, 3 ) }
print "distance:"; geom_distance( pts( 0 ), pts( 2 ) )
geom_each( @pts( 0 ), 3, @Visit, nullptr )
```

examples/ch29/geom-cinclude.out

```
geom 1.0, major version 1
distance: 5
  ( 0, 0 )
  ( 4, 0 )
  ( 4, 3 )
```

Writing a binding by hand means copying every type and declaration out of the header, and getting each one right. **`#cinclude`** reads the header itself: it hands `geom.h` to **clang**, the C compiler, and turns what clang reports into tiko declarations, for the

platform being built. The listing has no `type`, no `declare` and no `extern`: `geom_point`, `geom_visit`, all four functions and the constant `GEOM_VERSION_MAJOR` come from the header. `#foreignlib` still does the linking; the two do separate jobs.

What comes across: structs and unions, enums, `typedef`s, functions (variadic ones included), function pointer types, global variables, and `#define`s that are constants. What does not: macros with parameters, which are C code rather than declarations; C++ headers; and `NULL`, which is `0` — write `nullptr`. Every name keeps its C spelling through `alias`, and a name that would collide with a tiko keyword gets an underscore — `out` becomes `out_`.

The layout is checked, not trusted. The importer writes tiko declarations and lets tiko lay out each struct, then compares every size, alignment and field offset against clang's. A struct whose layout disagrees is left out rather than imported wrong.

A quoted header is looked for in two places: first under the directory the compiler runs in, then beside the file that says `#cinclude`. So this listing compiles from any directory, as `#foreignlib`'s path does. A header that includes others with angle brackets, as in `#include <lib/other.h>`, is a different matter: clang looks for those only in the directories given with `-cinclude`, and in the system's own.

> **PITFALL**
> **A header under the working directory wins over the one beside the file.** If the directory you compile from happens to hold a `geom/geom.h` of its own, an older copy or another library's, that is the header `#cinclude` imports, and nothing says so. `#include` searches the other way round, beside the file first. Keep a vendored header's path specific to its library.

`#cinclude` needs **libclang**, which comes with the LLVM installation tiko already uses; a program without `#cinclude` never loads it. The result is cached, so a large header such as `windows.h` is slow the first time and quick after.

Steering the importer

The importer takes options of its own, each passed to clang:

Option	Does
`-cinclude dir`	adds a directory clang searches for headers (C's `-I`)
`-cdefine name[=value]`	defines a C macro before the header is read (C's `-D`)
`-csysroot dir`	the root of the C headers and libraries for the target
`-copt arg`	passes any other argument to clang unchanged
`-cnomacros`	leaves `#define` constants out
`-cnoverify`	skips the layout check — there is no good reason to
`-cnocache`, `-ccachedir dir`	skip the cache, or keep it somewhere else

To see what `#cinclude` would produce, **`-cimport header -cimport-out file`** writes the binding as a file instead (`-` for the screen), and `-cimport-manifest file` adds clang's layout of every struct. For `geom.h` it writes an `extern "C"` block holding the struct —

```
type geom_point
    x as double
    y as double
end type
```

— a `type geom_visit as sub cdecl( … )` for the callback, one `declare … cdecl alias "…"` per function, and `const GEOM_VERSION_MAJOR = 1`. They are the declarations `geom-hand.tiko` wrote by hand. A generated binding can be kept in the project and compared with the next version of the header, which shows exactly what the library changed.

On Windows, `#define UNICODE` before `#cinclude once "windows.h"` makes the unsuffixed Windows names such as `MessageBox` mean the `W` (UTF-16) functions; without it they mean the `A` ones. Both suffixed forms are always there.

Loading a library at run time

`examples/ch29/dylib.tiko`

```
'' Load a library while the program runs, and find a function in it.
#if defined( __FB_WIN32__ )
    const LIBNAME = "ucrtbase.dll"
#elseif defined( __FB_DARWIN__ )
    const LIBNAME = "libm.dylib"
```

```
#else
    const LIBNAME = "libm.so.6"
#endif

type MathFn as function cdecl( byval x as float64 ) as float64

dim as any ptr lib_ = dylibload( LIBNAME )
if lib_ = nullptr then
    print "could not load the library"
    end 1
end if

'' dylibsymbol returns an address, or nullptr for a name it lacks.
dim as MathFn cosine = dylibsymbol( lib_, "cos" )
dim as MathFn missing = dylibsymbol( lib_, "no_such_function" )
print "found cos: "; cbool( cosine <> nullptr ); _
      ", found no_such_function: "; cbool( missing <> nullptr )

if cosine <> nullptr then
    print "cos( 0 ) ="; cosine( 0.0 )
    print "cos( pi ) ="; cosine( atn( 1 ) * 4 )
end if

dylibfree( lib_ )
```

examples/ch29/dylib.out

```
found cos: true, found no_such_function: false
cos( 0 ) = 1
cos( pi ) =-1
```

A library linked at build time must be present for the program to start. **`dylibload( name )`** opens a shared library while the program runs and returns a handle, or `nullptr` if it cannot. A program can then decide what to do when a plugin or an optional library is missing. **`dylibsymbol( handle, "name" )`** finds a function or variable by name and returns its address, or `nullptr`; assigning it to a function pointer of the right type makes it callable. **`dylibfree`** closes the library, after which every address taken from it is invalid.

The names differ by platform: `name.dll` on Windows, `libname.so` (often with a version, as in `libm.so.6`) on Linux, `libname.dylib` on macOS. The listing loads the C runtime's maths library on each — `ucrtbase.dll`, `libm.so.6` and `libm.dylib` — and calls its `cos`.

`dylibsymbol` returns an untyped address, so the function pointer's type is a promise exactly as a `declare` is. Nothing checks it. A plugin system typically agrees on one entry point with a fixed signature, looks that up, and receives everything else from it.

A tiko program can be the library: `tiko -dll plugin.tiko` builds a DLL or shared object, and `tiko -lib` a static library, whose procedures C or another tiko program can call.

What it costs

- **A call to C costs what a call to tiko costs.** There is no wrapper and no conversion: the arguments go in registers and on the stack exactly as C would put them.
- **Strings are copied when they come back.** `*p` copies C's characters into a tiko string. `strptr` going in copies nothing.
- **`#cinclude` costs compile time on the first build**, seconds for a large header, and almost nothing after, from the cache.
- **`dylibload` and `dylibsymbol` cost a search at run time.** Look a symbol up once and keep the pointer.

Summary

- `declare … cdecl alias "name"`, or an `extern "C" … end extern` block, declares a C function. `lib "name"` links its library.
- `int` is `int32`, `long long` is `int64`, `double` is `float64`. **C's `long` is `clong`**: 4 bytes on Windows, 8 on Linux. `wchar_t` is 2 bytes on Windows, 4 on Linux, like `wstring`.
- A `type` has C's struct layout; `field = n` packs it. A `ustring` never belongs in a struct C reads.
- `strptr` passes a string's bytes to C; `*p` copies C's back. Memory C allocates, C frees.
- A callback is a `cdecl` procedure, or a lambda that captures nothing.
- `...` declares a variadic C function; `cva_list`, `cva_start`, `cva_arg`, `cva_end` write one.
- x86-64 has one calling convention per platform; `pascal` alone reverses its arguments. `extern "C++"` mangles names as C++ does; `--dump-mangle` shows them.
- `#inclib`, `#libpath`, `-l`, `-p` link installed libraries. `#foreignlib name static|dynamic "path"` links one inside the project, relative to the declaring file, picked per platform.
- `#cinclude "header.h"` turns a C header into declarations through clang, layout-checked; the header is found under the working directory, then beside the file, then in `-cinclude` directories.
- `dylibload`, `dylibsymbol` and `dylibfree` load a library while the program runs.

Exercises

1. Bind C's `bsearch` and use it to look values up in a sorted array, with a `cdecl` comparison function. {solution="ex29-01″}
2. Bind `strtol`, using its end pointer to read every number at the start of a string and report where the numbers stopped. Use the right type for C's `long`. {solution="ex29-02″}
3. Add a function `geom_area` to `geom.c` that returns the area of a polygon, rebuild the library with `build.sh`, and call it through `#cinclude`. What did you have to change on the tiko side?
4. Rewrite `callbacks.tiko` so the comparison sorts by distance from a point chosen at run time. Why can the comparison not be a capturing lambda, and how does the `user` pointer of `geom_each` solve the same problem?
5. Build a tiko procedure into a DLL or shared object with `-dll`, and load it from another program with `dylibload`.

Using C libraries

Chapter 28 used SQLite through `vendor/sqlite3`, one of the toolchain's vendor packages: a C library that arrives ready to `#import`. The toolchain ships a handful of those. Every other C library — and there are thousands, for databases, compression, graphics, sound, networking — you connect yourself.

Real libraries arrive as an archive from a project's website, with the files arranged the way their authors chose, built for compilers you may not have, in several variants whose differences the archive does not explain. Before the first line of your program compiles you have to choose a package, unpack it, give tiko the library's declarations, decide which file to link, and make sure the program can find the library again when it runs, on your machine and on someone else's.

tiko does it in two steps, and they are the steps every vendor package was made with:

1. **`tiko c2t`** translates the library's C header, once, into a tiko header: a `.ti` file that reads like the C it came from, which you look over, finish where it says so, and keep with your code.
2. **`#foreignlib`** links the library's compiled code, statically or dynamically, choosing the right file for each platform.

This chapter takes four of the most widely used C libraries through both steps:

Library	Version	What it is for	Licence
SQLite	3.54.0	an SQL database in a single file, with no server	public domain
raylib	6.0	games and graphics: windows, drawing, input, sound	zlib
GLFW	3.5.1	a window and an OpenGL context, and the input that goes with them	zlib
SDL3	3.4.18	windows, rendering, audio, input and more, for games and media	zlib

Each one is taken from its download page to a running program, linked both **statically** and **dynamically**, kept either **inside the project** or **installed once** for every program on the machine, on Windows and on Linux. The programs are the

libraries' own first examples, translated into tiko, so you can put them beside the library's documentation and read the two together.

> **ON MACOS**
> The SQLite programs run on macOS too: `fetch-libs.sh` builds SQLite's static archive and its `.dylib` there. raylib, GLFW and SDL link Apple's frameworks — Cocoa, IOKit, OpenGL — and tiko cannot name a framework on its link line yet, so their programs in this chapter are for Windows and Linux. On a Mac, the toolchain's `vendor/SDL3` links Homebrew's SDL instead.

Every listing in this chapter was compiled and run on both platforms, against exactly the versions in the table.

Before you start

Building a program that uses a translated header needs nothing more than tiko and LLVM, as Chapter 1 set up. **Translating** a header needs more, because `tiko c2t` hands the header to Clang and Clang has to read every header it includes:

- **libclang**, Clang as a library, which the LLVM installation provides.
- **The Windows headers, on Windows.** c2t reads a header once for each of tiko's targets. For the Windows pass Clang reads the C runtime's and the Windows SDK's headers from the **Visual Studio Build Tools**: install them, free, with *Desktop development with C++*.
- **The Linux headers, for the Linux pass.** On Linux they are already there, with the `-dev` package of any library a header includes. On Windows, the headers of a Linux installation in WSL serve, named with `--sysroot`.

All of that is needed **only by whoever translates**. The `.ti` it writes is an ordinary tiko file: anyone who builds the program afterwards, on either platform, needs no Clang, no Visual Studio and no C headers. This chapter translates on Windows, with Ubuntu in WSL for the Linux pass. On Linux, c2t translates a header that includes nothing beyond C's freestanding headers (`stddef.h`, `stdint.h`, `stdbool.h`, `stdarg.h`), as `sqlite3.h` and `raylib.h` do; one that includes `<GL/gl.h>` or `<stdio.h>` fails its Windows pass there, for want of the Windows headers.

Two more tools are used to build libraries from their source: **CMake**, from `cmake.org` or your package manager, and on Linux **a C compiler and the development packages** a library lists, named in each library's section.

The shell commands in this chapter are written for **bash**, and the same commands work on both platforms: on Windows, run them in Git Bash, which comes with Git for Windows. The LLVM tools they use (`clang`, `llvm-lib`, `llvm-dlltool`) are in LLVM's

`bin` directory, `C:\Program Files\LLVM\bin` by default; add it to `PATH`, or write the full path.

What a library package contains

Open any C library's download and you find the same few kinds of file, under different names.

File	Windows	Linux	What it is for
header	`.h`	`.h`	the declarations: what `tiko c2t` translates
static library	`.lib`	`.a`	the library's compiled code, copied into your program when it links
shared library	`.dll`	`.so`, `.so.6`, `.so.6.0.0`	the compiled code as a separate file, loaded when your program starts
import library	`.lib`	none	Windows only: a small file that tells the linker which DLL holds each function

On Windows, `.lib` means two different things. A **static** `.lib` holds the library's code; an **import** `.lib` holds none, only the names of the functions and the name of the DLL they live in. You cannot tell them apart by the name, only by the size and by the package's documentation: raylib's `raylib.lib` is 5 MB of code, its `raylibdll.lib` 400 KB of names.

The linker never reads a `.dll`. To link with a DLL, you link with its import library, and the DLL is used only when the program runs. If a package ships a DLL without one, `#foreignlib` makes one for you when the program links; the SQLite section shows it.

Choosing the right package

Libraries publish several builds of the same version. Two choices matter for tiko:

- **64-bit only.** tiko builds x86-64 programs. A package marked `win32`, `x86`, `i386` or `i686` cannot be linked into them.

- **On Windows, the Visual C++ build.** tiko's Windows target, `windows-x86_64`, uses Microsoft's conventions and Microsoft's C runtime, the same as Visual C++. Packages are labelled `msvc`, `VC`, `vc2022` or similar. The `mingw` packages are built for GCC's conventions on Windows, and are not the ones this chapter uses.

On Linux the choice is simpler: a library built by your distribution's compiler, for x86-64.

Translating a header

Chapter 29's `#cinclude` reads a C header at every build and keeps what it learns in a cache for the compiler alone. **`tiko c2t`** reads the header once and writes a tiko header that a person can read:

```
tiko c2t vendor/sqlite/sqlite3.h --verify
```

It writes `vendor/sqlite/sqlite3.ti`, beside the header, in about three seconds. Here is a piece of `sqlite3.h`, the type every connection is and two of its result codes:

```
typedef struct sqlite3 sqlite3;
SQLITE_API int sqlite3_close(sqlite3*);
#define SQLITE_ROW         100  /* sqlite3_step() has another row ready
*/
#define SQLITE_DONE        101  /* sqlite3_step() has finished executing */
```

and the same lines in `sqlite3.ti`:

```
type sqlite3__fwd as sqlite3
declare function sqlite3_close( byval as sqlite3 ptr ) as int32
#define SQLITE_ROW 100          /' sqlite3_step() has another row ready
'/
#define SQLITE_DONE 101       /' sqlite3_step() has finished executing '/
```

Every declaration is where the C put it, with its comments, its blank lines and SQLite's documentation around it, so the `.ti` reads alongside the library's manual. C's types become tiko's fixed-width ones — `int` is `int32`, `const char *` is `const zstring ptr` — the same on both targets. `SQLITE_API`, the macro SQLite puts before every function, is recognised as decoration and dropped. A struct the header declares but never defines, like `sqlite3`, is an **opaque** type: a program can hold a pointer to one and nothing else.

Two platforms in one file

c2t parses the header twice, once as Clang would compile it for **windows-x86_64** and once for **linux-x86_64**, and merges the two. What is the same on both is written once. What differs is written under a guard the translation makes, `#if`

`defined(TIKO_OS_WINDOWS)` … `#endif`, and the header's own `#if` lines are kept, with C's platform tests spelled as tiko's: `_WIN32` becomes `TIKO_OS_WINDOWS`, `__linux__` becomes `TIKO_OS_LINUX`. One `.ti` serves both platforms, which is why it can be committed and shared.

A branch neither platform takes — an `#ifdef SQLITE_ENABLE_SESSION` for an optional feature, an `#elif` for macOS — is translated as well, inside its `#if`. Define the macro before including the `.ti`, and the program sees what a C program that defined it would see.

The banner

Every translation opens with a banner, which is the first thing to read:

```
'' sqlite3.ti -- tiko header translated by c2t (tiko 0.1.0-dev)
''   from the C header "sqlite3.h"
''   source sha256 : cd20a2877bc82b94...c581aac6e240925535cbd90166
''   clang         : clang version 21.1.0
''   passes        : x86_64-pc-windows-msvc, x86_64-pc-linux-gnu
''   options       : --verify
''
'' Generated from C. Search for C2T-TODO and translate those parts by
'' hand.
''
'' Renamed:
''   parameter  sqlite3_bind_text64.encoding
''              -> sqlite3_bind_text64.encoding_   (tiko reserved word)
''   ...
''   constant   double -> double_   (tiko reserved word)
''
'' 1 item needs manual translation:
''   line 11414  sqlite3.h:11450  C2T-TODO(macro)  double
```

(Wrapped here to fit the page, with the hash shortened and five renames left out; and this is the banner of a plain `--verify` run, before the `--shared-macro` pass that *Variables a DLL exports* adds.) The SHA-256 names the exact header that was translated, so the `.ti` says which version of the library it belongs to. There is no date in it: translating the same header again gives the same bytes, and after an upgrade, a diff of the `.ti` shows exactly what the library changed.

Renamed lists every C name that is a tiko word. `encoding` and `type` are keywords in tiko, so a parameter called that gets an underscore. Parameter names do not affect a call, so these renames cost nothing; a renamed function or type would be called by its new name.

Items that need manual translation are the parts c2t could not carry. Each is marked where it stands, the C quoted underneath:

```
#ifdef SQLITE_OMIT_FLOATING_POINT
'' C2T-TODO(macro) sqlite3.h:11450 -- 'double' is a tiko keyword:
'' #undef would remove it
'' # undef double
#endif
```

(The TODO's first line is one line in the file.)

The exit status says how it went: **0**, clean; **2**, translated, with TODOs to read; **1**, nothing written. A TODO is API that is missing until someone writes it by hand, and the reason in brackets says why. Most fall into three kinds:

- **A C feature tiko cannot express**: a variadic macro (`__VA_ARGS__`), a macro that pastes tokens with `##`, a `static inline` function, whose body c2t quotes for porting because there is no symbol in the library to link against.
- **A system header with no tiko counterpart.** c2t knows C's standard headers (`<stdio.h>` becomes `crt/stdio.bi`), Windows' (`<windows.h>` becomes `windows.bi`) and OpenGL's; anything else is a TODO.
- **A branch Clang cannot parse** even when it is switched on: C++ under `#ifdef __cplusplus`, or a Vulkan type in a header that expects Vulkan's own to have been included first.

Read the list and decide, item by item, whether your program needs it. SQLite's one TODO sits in a branch that only a build without floating point would take, and nothing a program calls is missing. GLFW, below, has twenty-two, and none of them is on its way.

A **NOTE** is information rather than a gap: the translation is complete, and something deserves a look. SQLite's `SQLITE_TRANSIENT` is a C macro that casts `-1` to a function pointer; c2t translates it, `cast(sqlite3_destructor_type, (-1))`, and notes that Clang gives no value for a pointer to check it against.

`--verify` compiles the translation for both targets inside c2t and compares every struct's size, alignment and member offsets, and every integer macro's value, with Clang's. A struct that differs is refused and marked, never written wrong. It costs almost nothing — `sqlite3.h` takes three seconds with it and without — so use it every time.

> **PITFALL**
> **A translation describes one version of the library.** Link `sqlite3.ti`, made from 3.54.0's header, with an older SQLite, and a function the old library lacks

is a link error, which is noticed. A struct that changed shape between versions is not: the program compiles, links and runs, and reads the wrong bytes. Translate the header that came in the same package as the library you link, and translate again whenever you upgrade either. The banner's SHA-256 says which header a `.ti` was made from.

Finishing a translation

When a TODO is something your program needs, write it by hand — but **not in the generated file**, where the next translation would overwrite it. Put it in a file of its own beside it, `sqlite3-residue.ti` say, included after the translation, with a line saying why each item is there. The vendor packages do exactly this: `vendor/sqlite3` is the translated `sqlite3.ti`, a `sqlite3-residue.ti`, and the line that links the library.

Variables a DLL exports

Most C libraries export only functions, and a translated header calls them the same way however the library is linked. A **variable** the library exports is different on Windows: a program linked to the DLL must reach it through the import table, which tiko writes `extern import`, and a program linked statically must not. SQLite has three, among them `sqlite3_version`. Without help, a dynamic program that reads one fails to link:

```
lld-link: error: undefined symbol: sqlite3_version
```

c2t solves it with one more pass, given the library's **shared-build switch**: the macro that makes the header declare things for a DLL. raylib's is `USE_LIBTYPE_SHARED`. SQLite has none of its own, so the switch is its export macro defined the way a DLL build would define it:

```
tiko c2t vendor/sqlite/sqlite3.h --verify \
    --shared-macro "SQLITE_API=__attribute__((dllimport))"
```

Each exported variable is then written both ways, the import form under `defined(TIKO_LINK_DYNAMIC)`. A program linked to the DLL defines that before it includes the translation:

```
#define TIKO_LINK_DYNAMIC 1
#include once "vendor/sqlite/sqlite3.ti"
#foreignlib SQLITE dynamic "vendor/sqlite/sqlite3"
```

and `print sqlite3_version` prints `3.54.0`. A vendor package defines it for you, from the `link` on its `#import`.

Headers that include each other

c2t translates **one header per run**. An `#include` of another of the library's headers becomes `#include once` of that header's `.ti`, at the same relative path, so translating each header into the same tree gives a tree of `.ti` files that include one another as the `.h` files did. `--verify` compiles each translation, so the headers a header includes must be translated first. For a library with many headers, a script walks them in that order; SDL's section shows one.

Because the includes are relative, a translated library needs no include directory on the command line, wherever it is kept. Its `.ti` files find each other as they found each other in C.

When to use c2t

`#cinclude` and `tiko c2t` share tiko's C importer, so a type comes across the same way in both. Use `#cinclude` for a quick experiment, or a header you never want to look at. Use `tiko c2t` for a library your program depends on: the translation is a file you can read, search, review in a pull request and diff on an upgrade; it builds on machines that have no Clang; and the parts that did not translate are written down where they are, rather than discovered at a call.

Static or dynamic

Every library in this chapter can be linked either way. The difference is where its code ends up.

Linked statically, the library's code is copied into your program. The program is one file; nothing can go missing when it is copied to another machine; and only the parts of the library the program uses are copied in. It is larger, and updating the library means relinking it.

Linked dynamically, your program holds only references to the library, and the library's own file, the DLL or `.so`, must be present when the program starts. The program is small, several programs can share one copy of the library, and the library can be updated without relinking. But the program now comes in two parts, and the second part has to be found.

Measured on the programs in this chapter, each linked both ways, in kilobytes of 1,000 bytes:

Program	Windows, static	Windows, dynamic	Linux, static	Linux, dynamic
`sqlite-quick`	1,168	38 + 3,519	1,114	44 + 1,197
`raylib-window`	608	27 + 1,905	996	35 + 2,100
`glfw-window`	227	29 + 234	331	35 + 353
`sdl-clear`	2,745	29 + 2,848	3,604	40 + 4,096

The second figure in each dynamic column is the shared library that has to travel with the program. A static program is smaller than the program plus its library, because the linker copies in only what is called.

All four licences allow either way. That is not true of every library: under the LGPL, for instance, linking statically brings obligations that linking dynamically does not. Read a library's licence before you decide.

`#foreignlib` says which, after the name it gives the library:

```
#foreignlib SQLITE dynamic "vendor/sqlite/sqlite3"
#foreignlib SQLITE static "vendor/sqlite/sqlite3-static"
```

The path is relative to **the file the directive is in**, and names the library without its platform's prefix or extension. The rest of this section is about what happens after the link.

Where a program finds its shared libraries

On Windows, a program looks for a DLL first in **its own directory**, then in the system directories, then in the current directory and the directories on `PATH`. Put the DLL beside the `.exe` and it is found. When it is not, the program does not start; from a console it exits at once with code `-1073741515`, which is `0xC0000135`, *DLL not found*, and prints nothing.

On Linux, a program looks for a `.so` in the directories listed in `LD_LIBRARY_PATH`, then in the system's library directories (`/usr/lib`, `/usr/local/lib` and the others the system's `ldconfig` knows), and **not** in its own directory. When it fails, it says so:

```
./raylib-dll: error while loading shared libraries: libraylib.so.600:
cannot open shared object file: No such file or directory
```

and exits with code 127. The name it looks for is the library's **soname**, a name recorded inside the `.so` when it was built (`libraylib.so.600`), not the name of the file you linked. That is why a Linux package carries the same library under three

names, such as `libraylib.so`, `libraylib.so.600` and `libraylib.so.6.0.0`: the first is for the linker, the second is the soname the program will ask for, and the third is the real file.

For a library installed on the system, nothing more is needed. For one inside your project, there are two ways to say where it is. The quick one is to run the program with the project's directory on `LD_LIBRARY_PATH`:

```
LD_LIBRARY_PATH=. ./sqlite-quick
```

The lasting one is an **rpath**: a directory recorded inside the program, which the loader searches before the system's own directories. The usual value is `$ORIGIN`, which the loader reads as "the directory this program is in", so a program and its `.so` files can be copied anywhere together. It is given to the linker, through tiko's **`-Wl`** option, which passes its value to the linker unchanged, split at commas:

examples/ch30/raylib-rpath.tiko

```
#cmdline "-x raylib-rpath -Wl -rpath,$ORIGIN"
#include once "vendor/raylib/raylib.ti"
#foreignlib RAYLIB dynamic "vendor/raylib/raylibdll"
```

This listing is `raylib-dll` with one more line. It runs with no `LD_LIBRARY_PATH` at all:

examples/ch30/raylib-rpath.out

```
drew 60 frames
```

Typed in a shell, `$ORIGIN` must be in single quotes, or the shell replaces it with an empty variable before tiko sees it: `tiko game.tiko -Wl '-rpath,$ORIGIN'`. In a `#cmdline`, as here, and in a `tiko.toml`'s `options`, no shell is involved and it needs no quoting. On Windows, where a program looks beside itself anyway, `-Wl` passes options to `lld-link` instead: `-Wl /STACK:8388608` reserves an 8 MB stack.

PITFALL

On Linux, a missing library is not always an error. Ubuntu has `libsqlite3.so.0` installed for its own use. A program linked with the SQLite 3.54.0 in this chapter's project asks for that same soname, so run without `LD_LIBRARY_PATH` it starts without complaint, using the system's copy, an older version, and nothing says so. The same program prints `SQLite 3.54.0` with `LD_LIBRARY_PATH=.` and `SQLite 3.45.1` without it. A program that depends on a newer feature fails much later, and somewhere that looks unrelated. An rpath of `$ORIGIN` puts your copy first; and when it matters which copy is loaded, check it the way the listings here do: ask the library for its version.

A static library brings its dependencies with it

A static library is the library's code and nothing else. Whatever *that* code calls becomes your program's to link. raylib calls Windows' multimedia timer, so a program that links raylib statically must also link `winmm`; SDL asks Windows about the devices plugged in, so it needs `setupapi`. A shared library carries those links inside itself, which is why the dynamic listings in this chapter name no system libraries.

`#foreignlib` takes them after the library itself, each as `"system:name"`:

```
#foreignlib RAYLIB static "vendor/raylib/raylib", _
    "system:winmm", "system:gdi32", "system:user32", "system:shell32"
```

To find the list for a library, look first in its documentation: most say what a static build needs. If they do not, link without any and read what the linker cannot find:

```
lld-link: error: undefined symbol: __declspec(dllimport)
SetupDiGetClassDevsA
lld-link: error: undefined symbol: __declspec(dllimport) CoInitializeEx
lld-link: error: undefined symbol: __declspec(dllimport) SysFreeString
```

Each name is a function of the operating system. On Windows, Microsoft's documentation for the function names the library in its *Requirements* table: `SetupDiGetClassDevsA` is in `setupapi.lib`, `CoInitializeEx` in `ole32.lib`, `SysFreeString` in `oleaut32.lib`. On Linux, the manual page or the package that provides the function says which library holds it. Add one `system:` entry for each library, and link again until nothing is missing.

Vendored or installed

Separately from static or dynamic, you choose where the library lives.

Vendored means the project carries its own copy, in a directory of its own, usually `vendor/`. Anyone who checks out the project can build it with nothing to install; the version is the one you tested with, and stays that way until you change it; and two projects can use two different versions without meeting. The cost is that each project carries its own copy.

Installed means one copy, in a shared place, used by every program on the machine. On Linux that place exists already, and the package manager fills it. On Windows there is no such place for C libraries, so you make one.

A vendored library

A translated header and `#foreignlib` were both made for vendoring. `#include` finds a quoted name beside the file that includes it first, and `#foreignlib` resolves its

path relative to the file that contains the directive, so a program finds the library's declarations and its files wherever it is built from.

`#foreignlib` also chooses the file **for the target**. Given `"vendor/raylib/raylib"`, it tries `raylib_windows_x86_64`, `raylib_windows`, `raylib_x86_64` and plain `raylib`, each with the target's extensions. So both platforms' files can sit side by side, and one directive serves both — as one `.ti` does:

```
vendor/raylib/raylib.h                          the C header
vendor/raylib/raylib.ti                        its translation, committed
vendor/raylib/raylib_windows_x86_64.lib         static, for Windows
vendor/raylib/raylib_linux_x86_64.a             static, for Linux
vendor/raylib/raylibdll_windows_x86_64.lib      import lib for raylib.dll
vendor/raylib/raylibdll_linux_x86_64.so         libraylib.so.600, renamed
raylib.dll                                      beside the program
libraylib.so.600                                beside the program
```

The target tags are not optional when both platforms share a folder. On Windows a static search tries `.a` before `.lib`, so an untagged `raylib.a` built for Linux is found first, and the link fails with an error that does not say why: `lld-link: error: unknown file type: rcore.o`.

Notice which files are renamed and which are not. You rename the files the **linker** reads, to give them their target tags. You never rename the files the program loads when it **runs**: the DLL keeps the name its import library records (`raylib.dll`) and the `.so` keeps its soname (`libraylib.so.600`), because those are the names the program will ask for.

Of all those files, only the `.ti` is source. The libraries are built artefacts, recreated from the download, and the C header is only an input to c2t. A project commits its translations and fetches the rest; this chapter's examples do exactly that. The script `examples/ch30/fetch-libs.sh` builds the `vendor/` directory: it downloads each package, checks it against a SHA-256 checksum, and copies and renames the files, building the ones that have no download. Run it once on each platform:

```
bash fetch-libs.sh
```

The script `examples/ch30/translate.sh` then holds the c2t commands for all four libraries, the ones this chapter shows. Between them, the two scripts are the exact record of every step in the chapter, in order. When the prose says "copy this file there", the scripts are where to look to see it done.

An installed library

On **Linux**, the package manager installs a library's header into `/usr/include` and the library into `/usr/lib/x86_64-linux-gnu`. A library you build yourself and install

with `sudo cmake --install build` goes to `/usr/local/include` and `/usr/local/lib`. The linker searches all of them by default, so a program links an installed library by name alone:

```
#include once "sqlite3.ti"
#foreignlib SQLITE dynamic "system:sqlite3"
```

The header still needs translating, once: `tiko c2t /usr/include/sqlite3.h -o sqlite3.ti`, and the `.ti` lives with the project. It describes the version the machine had when it was made, so retranslate when the distribution upgrades the library.

`"system:sqlite3"` hands `sqlite3` to the linker the way `#inclib "sqlite3"` would, and the linker finds `libsqlite3.so`. Two things to know about it: the `static` or `dynamic` before a `system:` spec is **not checked**, and when a directory has both `libsqlite3.so` and `libsqlite3.a`, the linker takes the `.so`. After installing a library into `/usr/local/lib` yourself, run `sudo ldconfig` once, so that programs can find it when they start.

On **Windows** you choose a folder, such as `C:\tiko-libs`, and put each library in it the same way as `vendor/`, its translation beside it. A program then needs to be told where that folder is, which is the job of a **binding header**, at the end of this chapter.

SQLite

SQLite is an SQL database engine in a library: no server to install or start, and a whole database is one ordinary file. Its authors describe it as the most widely deployed database engine in the world, and it is inside many of the programs you already use.

> **NOTE**
> tiko already ships SQLite, as `vendor/sqlite3`, and Chapter 28 uses it. This section makes the same library from its download, which is what you do for a newer SQLite than the toolchain's, or SQLite built with options of your own — and it is the pattern for every library that has no vendor package.

Downloading

Everything is on the download page, `https://sqlite.org/download.html`:

File	What is in it
`sqlite-amalgamation-3540000.zip`	`sqlite3.c` and `sqlite3.h`: the **whole library as one C file**, with its header
`sqlite-dll-win-x64-3540000.zip`	`sqlite3.dll` and `sqlite3.def`, the library already built for 64-bit Windows
`sqlite-autoconf-3540000.tar.gz`	the same `sqlite3.c` and `sqlite3.h`, with a build script, for Unix

The number in each name is the version: 3.54.0 is `3540000`.

SQLite is unusual in shipping its source as one file. It is meant to be compiled into your program directly, which makes a static library easy to build yourself.

The header

`sqlite3.h` goes to `vendor/sqlite/`, and is translated there, with the shared-build pass from *Variables a DLL exports*:

```
tiko c2t vendor/sqlite/sqlite3.h --verify \
    --shared-macro "SQLITE_API=__attribute__((dllimport))"
```

`sqlite3.h` includes nothing but `<stdarg.h>`, which tiko has built in, so it needs no `--sysroot`, and c2t translates it on Linux as well. Exit status 2, and the one TODO from *The banner*.

Windows: the dynamic library

The DLL zip has `sqlite3.dll` and `sqlite3.def`, a text file that lists every function the DLL exports, but **no import library**. You do not have to make one. Put the DLL into the project under its target-tagged name, and the original beside the program:

```
vendor/sqlite/sqlite3_windows_x86_64.dll     the DLL, for the link
sqlite3.dll                                  the same file, for the
program to load
```

When `#foreignlib SQLITE dynamic "vendor/sqlite/sqlite3"` finds only a `.dll`, the link reads the DLL's own table of exports and makes the import library from it, with `llvm-dlltool` from the LLVM installation tiko already uses, then deletes it with the other intermediate files. Run with `-v`, the `llvm-dlltool` command is printed before the link.

The import library names the DLL **as the DLL names itself**, the name it was built with, `sqlite3.dll`, and not by the name of the file it was read from. That is the Windows counterpart of a soname: the copy in `vendor/` can carry its target tag,

and the program still loads `sqlite3.dll`. A DLL that is not 64-bit x86-64 is `TKO0172`, with the reason:

```
error TKO0172: cannot make an import library for 'lib/
w32_windows_x86_64.dll':
it is not a 64-bit x86-64 DLL (machine &h014C)
```

(One line on the screen, its path shortened here.) That is the 32-bit download, taken by mistake.

Windows: an import library of your own

To keep an import library in the project instead, so the link does not make one each time, make it from the `.def` with `llvm-dlltool`:

```
llvm-dlltool -m i386:x86-64 -d sqlite3.def -D sqlite3.dll \
    -l vendor/sqlite/sqlite3_windows_x86_64.lib
```

`-m i386:x86-64` asks for a 64-bit import library, `-d` names the export list, `-l` the library to write, and `-D` the DLL that the program will load. The import library records that name, and the program asks for exactly that file when it starts.

> **PITFALL**
> **An import library with no DLL name links, and the program then cannot start.** `sqlite3.def` has no `LIBRARY` line naming its DLL, so the name must come from the command line. Made with `llvm-lib /def:sqlite3.def` and no name, the import library is accepted, the program links cleanly, and running it fails at once with `0xC0000135`: Windows was asked to load a DLL called nothing. `llvm-readobj --coff-imports` on the `.exe` shows it, an import block with an empty `Name:`. Give the name with `-D sqlite3.dll`, or add a first line `LIBRARY sqlite3.dll` to the `.def`.

A `.lib` beside the DLL under one of the names `#foreignlib` tries is found before the `.dll`, so when both are there, the import library you made is the one that is linked.

Windows: the static library

Compile the amalgamation with clang and put the one object in a library with `llvm-lib`:

```
clang -c -O2 -o sqlite3.obj sqlite3.c
llvm-lib /out:vendor/sqlite/sqlite3-static_windows_x86_64.lib
sqlite3.obj
```

It takes about twenty seconds; `sqlite3.c` is over nine megabytes of C. On Windows clang compiles for Microsoft's conventions by default, the same as tiko, and finds the C runtime's headers in the Visual Studio Build Tools. Since a static `.lib` and an import `.lib` cannot share a name, the static one is called `sqlite3-static`.

Linux

The same two libraries, from the autoconf archive's `sqlite3.c`:

```
gcc -O2 -fPIC -c -o sqlite3.o sqlite3.c
ar rcs vendor/sqlite/sqlite3-static_linux_x86_64.a sqlite3.o
gcc -shared -Wl,-soname,libsqlite3.so.0 \
    -o vendor/sqlite/sqlite3_linux_x86_64.so sqlite3.o -lm
cp vendor/sqlite/sqlite3_linux_x86_64.so libsqlite3.so.0
```

`-fPIC` makes code that works at any address, which a shared library needs. `-Wl,-soname,…` records the soname, the name a program linked with the library will look for. Without one, the program would record the path it was linked with, `vendor/sqlite/sqlite3_linux_x86_64.so`, and work only from the directory it was built in. The last line puts the library beside the program under its soname.

To install SQLite for every program instead, use the distribution's package, which installs both the header and the library:

```
sudo apt install libsqlite3-dev
```

A first program

SQLite's documentation opens with a short C program, *SQLite In 5 Minutes Or Less*: it opens the database named by its first argument, runs the SQL in its second, and prints every row of the result. Here it is in tiko, linked dynamically:

examples/ch30/sqlite-quick.tiko

```
#include once "vendor/sqlite/sqlite3.ti"
#foreignlib SQLITE dynamic "vendor/sqlite/sqlite3"

'' Called once for each row of the result.
function Callback cdecl( byval notUsed as any ptr, _
        byval argc as int32, byval argv as zstring ptr ptr, _
        byval colName as zstring ptr ptr ) as int32
    for i as int32 = 0 to argc - 1
        dim value as ustring = "NULL"
        if argv[i] <> nullptr then value = *argv[i]
        print *colName[i]; " = "; value
    next
    print
```

```
    return 0
end function

if command( 2 ) = "" then
    print "Usage: sqlite-quick DATABASE SQL-STATEMENT"
    end 1
end if

dim db as sqlite3 ptr
if sqlite3_open( command( 1 ), @db ) <> SQLITE_OK then
    print "Can't open database: "; *sqlite3_errmsg( db )
    sqlite3_close( db )
    end 1
end if

dim errMsg as zstring ptr
if sqlite3_exec( db, command( 2 ), @Callback, nullptr, @errMsg ) _
        <> SQLITE_OK then
    print "SQL error: "; *errMsg
    sqlite3_free( errMsg )
end if
sqlite3_close( db )
```

Run with an in-memory database and four statements, one string that the shell lets run over several lines:

```
./sqlite-quick :memory: "create table tbl1(one text, two int);
    insert into tbl1 values('hello!', 10);
    insert into tbl1 values('goodbye', 20);
    select * from tbl1"
```

examples/ch30/sqlite-quick.out

```
one = hello!
two = 10

one = goodbye
two = 20
```

The two directives at the top are the whole connection to SQLite: the translation for the declarations, `#foreignlib` for the code. Everything else is the C program, line for line, and it shows the parts of Chapter 29 that a real library uses most:

- **Opaque handles.** `sqlite3` is a struct the header declares but never defines, so a program can hold only a pointer to one. `sqlite3_open` fills `db` through its address, `@db`, the way the C passes `&db`.
- **Strings into C.** `command( 2 )` is a tiko string; passed where the header says `const char *`, the compiler passes a pointer to its bytes.
- **Strings out of C.** `sqlite3_errmsg` returns `const char *`, and `*sqlite3_errmsg( db )` copies the text into a tiko string.
- **A callback.** `sqlite3_exec` calls `Callback` once per row, so `Callback` is `cdecl` and takes exactly the parameters the translated declaration says: `any ptr`, `int32`, and two `zstring ptr ptr`. `argv[i]` is the `i`th column's text, or a null pointer for SQL's `NULL`.
- **Memory C allocated.** On failure, `sqlite3_exec` points `errMsg` at a message SQLite allocated; SQLite's documentation says to free it with `sqlite3_free`, so the program does.

Prepared statements, linked statically

Gluing values into SQL text is how SQL injection happens, and the text has to be parsed again for every row. Real programs **prepare** a statement once, with `?` where the values go, and then **bind** a value to each `?`. This program does that, linked statically:

examples/ch30/sqlite-bind.tiko

```
#include once "vendor/sqlite/sqlite3.ti"
#foreignlib SQLITE static "vendor/sqlite/sqlite3-static"

sub Check( byval rc as int32, byval db as sqlite3 ptr )
    if rc <> SQLITE_OK andalso rc <> SQLITE_DONE then
        print "SQLite error "; rc; ": "; *sqlite3_errmsg( db )
        end 1
    end if
end sub

print "SQLite "; *sqlite3_libversion( )

dim db as sqlite3 ptr
Check( sqlite3_open( ":memory:", @db ), db )
Check( sqlite3_exec( db, "create table city( name text, pop int )", _
    nullptr, nullptr, nullptr ), db )

dim names( 0 to 2 ) as string = _
    { "St. John's", "Gander", "Corner Brook" }
dim pops( 0 to 2 ) as int32 = { 110525, 11880, 19316 }
```

```
dim ins as sqlite3_stmt ptr
Check( sqlite3_prepare_v2( db, "insert into city values( ?, ? )", _
    -1, @ins, nullptr ), db )
for i as int32 = 0 to 2
    '' -1: the text ends at its zero byte. SQLITE_STATIC: the string
    '' outlives the statement's use of it, so SQLite need not copy it.
    Check( sqlite3_bind_text( ins, 1, names( i ), -1, SQLITE_STATIC ), _
        db )
    Check( sqlite3_bind_int( ins, 2, pops( i ) ), db )
    Check( sqlite3_step( ins ), db )
    sqlite3_reset( ins )
next
sqlite3_finalize( ins )

dim sel as sqlite3_stmt ptr
Check( sqlite3_prepare_v2( db, _
    "select name, pop from city where pop > ? order by pop desc", _
    -1, @sel, nullptr ), db )
sqlite3_bind_int( sel, 1, 15000 )
do while sqlite3_step( sel ) = SQLITE_ROW
    print *cast( zstring ptr, sqlite3_column_text( sel, 0 ) ); ": "; _
        str( sqlite3_column_int( sel, 1 ) )
loop
sqlite3_finalize( sel )
sqlite3_close( db )
```

examples/ch30/sqlite-bind.out

```
SQLite 3.54.0
St. John's: 110525
Corner Brook: 19316
```

Only the `#foreignlib` line differs from the dynamic program:

examples/ch30/sqlite-bind.tiko

```
#include once "vendor/sqlite/sqlite3.ti"
#foreignlib SQLITE static "vendor/sqlite/sqlite3-static"
```

`static` instead of `dynamic`, and the static library's base name. The program now needs no DLL: copied to another machine, it runs. The translation is the same file: it does not care how the library is linked.

Chapter 28 covered SQLite's interface itself — preparing, binding, stepping, finalizing. Two points belong to the translation. `SQLITE_STATIC`, which promises SQLite the string outlives the statement's use of it, is one of the pointer macros c2t trans-

lated with a NOTE, `cast(sqlite3_destructor_type, 0)`. And `sqlite3_column_text` returns `const unsigned char *`, so the translation declares it `const uint8 ptr`; cast it to a `zstring ptr` to read it as text.

raylib

raylib is a library for making games, designed to be easy to learn: one header, 600 functions with plain names, and no setup beyond opening a window. It draws with OpenGL and handles the keyboard, mouse, gamepads, sound, textures, fonts and 3D models.

Downloading

raylib publishes prebuilt packages on GitHub, `https://github.com/raysan5/raylib/releases`. For 6.0, two are for tiko:

File	What is in it
`raylib-6.0_win64_msvc16.zip`	`include/raylib.h` (and `raymath.h`, `rlgl.h`); `lib/raylib.lib`, static; `lib/raylibdll.lib`, the import library; `lib/raylib.dll`
`raylib-6.0_linux_amd64.tar.gz`	the same headers; `lib/libraylib.a`, static; `lib/libraylib.so.6.0.0` with its links `libraylib.so.600` and `libraylib.so`

The other packages on the page are for 32-bit Windows (`win32`), the MinGW compiler, ARM, macOS and the web.

Installing it in a project

From the Windows package:

From the package	Into the project
`include/raylib.h`	`vendor/raylib/raylib.h`
`lib/raylib.lib`	`vendor/raylib/raylib_windows_x86_64.lib`
`lib/raylibdll.lib`	`vendor/raylib/raylibdll_windows_x86_64.lib`
`lib/raylib.dll`	`raylib.dll`, beside the program

From the Linux package:

From the package	Into the project
`include/raylib.h`	`vendor/raylib/raylib.h` (the same file)
`lib/libraylib.a`	`vendor/raylib/raylib_linux_x86_64.a`
`lib/libraylib.so.6.0.0`	`vendor/raylib/raylibdll_linux_x86_64.so`
`lib/libraylib.so.6.0.0`	`libraylib.so.600`, beside the program

The dynamic base name is `raylibdll` on both platforms, after the Windows import library, so that one `#foreignlib` line finds the right file on each. Then translate the header:

```
tiko c2t vendor/raylib/raylib.h --verify
```

Exit status 0: all of raylib's 600 functions, its structs, enums and colours, with nothing left to finish. raylib exports no variables, so it needs no `--shared-macro`.

Installing it for every program

On **Linux**, recent distributions package raylib as `libraylib-dev`; Ubuntu 24.04 does not. Without a package, copy the prebuilt files into `/usr/local` and refresh the loader's cache:

```
sudo cp include/*.h /usr/local/include/
sudo cp -P lib/libraylib.* /usr/local/lib/
sudo ldconfig
```

`cp -P` copies the two links as links. A program then includes its translation of `raylib.h` and writes `#foreignlib RAYLIB dynamic "system:raylib"`.

On **Windows**, see *A binding header* below.

A first window

raylib's examples start with `core_basic_window`, which opens a window and writes one line of text in it:

examples/ch30/raylib-window.tiko

```
#include once "vendor/raylib/raylib.ti"
#ifdef __FB_WIN32__
    #foreignlib RAYLIB static "vendor/raylib/raylib", _
        "system:winmm", "system:gdi32", "system:user32", _
        "system:shell32"
#else
    #foreignlib RAYLIB static "vendor/raylib/raylib", _
        "system:GL", "system:m", "system:pthread", "system:dl", _
        "system:rt", "system:X11"
#endif
```

```
const ScreenWidth = 800
const ScreenHeight = 450
dim maxFrames as int32 = valint( command( 1 ) )  '' 0: until closed

SetTraceLogLevel( LOG_WARNING )   '' raylib logs to stdout otherwise
InitWindow( ScreenWidth, ScreenHeight, _
    "raylib [core] example - basic window" )
SetTargetFPS( 60 )

dim frames as int32 = 0
do until WindowShouldClose( )
    BeginDrawing( )
    ClearBackground( RAYWHITE )
    DrawText( "Congrats! You created your first window!", _
        190, 200, 20, LIGHTGRAY )
    EndDrawing( )
    frames += 1
    if frames = maxFrames then exit do
loop
CloseWindow( )
print "drew "; str( frames ); " frames"
```

examples/ch30/raylib-window.out

```
drew 60 frames
```

Figure 1: The window that `raylib-window` opens.

Everything after the directives is raylib's example as it is written in C. A few lines are new:

- **Linked statically**, raylib needs a different set of system libraries on each platform, so the `#foreignlib` line is chosen with `#ifdef __FB_WIN32__`. On Linux it is the list raylib's documentation gives: OpenGL, the maths library, threads, `dl`, `rt` and X11.
- **`SetTraceLogLevel( LOG_WARNING )`** comes first because raylib reports everything it does, about forty lines when a window opens, on standard output. The listings keep it quiet so that their output is the program's own.
- **`maxFrames`** lets the program stop by itself after a number of frames given on the command line, so this book's checks can run it. Started with no argument, it runs until you close the window, as raylib's example does.

Two things in the translation are more than declarations. `RAYWHITE` and `LIGHTGRAY` are C macros that build a struct with a C compound literal:

```
#define RAYWHITE   CLITERAL(Color){ 245, 245, 245, 255 }
```

c2t asks Clang for each field's value and writes the tiko equivalent:

```
#define RAYWHITE type<Color>(245, 245, 245, 255)
```

And `DrawText` takes that `Color` **by value**, a four-byte struct passed the way C passes it, as Chapter 29 described for `geom_point`.

Linked dynamically

The same program, with its first lines replaced:

`examples/ch30/raylib-dll.tiko`

```
#include once "vendor/raylib/raylib.ti"
#foreignlib RAYLIB dynamic "vendor/raylib/raylibdll"
```

No system libraries this time: `raylib.dll` and `libraylib.so.600` already link everything they need. The program is 27 KB instead of 608 KB on Windows, and the DLL must travel with it.

GLFW

GLFW does less than raylib, on purpose. It creates windows, gives each one an **OpenGL context**, the surface OpenGL draws on, and reports keyboard, mouse, gamepad and window events. All drawing is left to OpenGL, or to Vulkan. It is the library to reach for when you want to write the graphics code yourself.

Downloading

On **Windows**, take *64-bit Windows binaries* from `https://www.glfw.org/download.html`, `glfw-3.5.1.bin.WIN64.zip`. It holds `include/GLFW/glfw3.h` and one folder of libraries per compiler:

Folder	What is in it
`lib-vc2022`	**the one to use:** `glfw3.lib`, static; `glfw3dll.lib`, the import library; `glfw3.dll`
`lib-vc2017`, `lib-vc2019`, `lib-vc2026`	the same, from other Visual C++ versions
`lib-static-ucrt`	a DLL built with the C runtime linked into it
`lib-mingw-w64`	for the MinGW compiler

`lib-vc2022` also has `glfw3_mt.lib`. GLFW's own `README.md` explains the difference: `glfw3.lib` is for programs that use the C runtime as a DLL, and `glfw3_mt.lib` for programs that link it statically. tiko's programs use the C runtime DLL, `ucrtbase.dll`, so take `glfw3.lib`.

On **Linux**, GLFW publishes source only: `glfw-3.5.1.zip` from the same page. Its documentation lists what building it needs, for both of Linux's window systems, X11 and Wayland:

```
sudo apt install libwayland-dev libxkbcommon-dev xorg-dev
```

Then build it twice with CMake, once as a static library and once as a shared one (a `\` at the end of a line continues the command on the next):

```
cmake -S glfw-3.5.1 -B build-static -DCMAKE_BUILD_TYPE=Release
cmake --build build-static
cmake -S glfw-3.5.1 -B build-shared -DCMAKE_BUILD_TYPE=Release \
    -DBUILD_SHARED_LIBS=ON
cmake --build build-shared
```

`build-static/src/libglfw3.a` becomes `vendor/glfw/glfw3_linux_x86_64.a`, and `build-shared/src/libglfw.so.3` becomes both `vendor/glfw/glfw3dll_linux_x86_64.so` and, beside the program, `libglfw.so.3`. To build for Wayland alone, without the X11 packages, add `-DGLFW_BUILD_X11=OFF`; this book's checks were built that way.

To install GLFW for every program on Linux, `sudo apt install libglfw3-dev` gives the distribution's version, which on Ubuntu 24.04 is 3.3.10; or run `sudo cmake --install build-shared` after the build above.

The header

The package's `include/` goes to `vendor/glfw/include/`. `glfw3.h` includes OpenGL's header, `<GL/gl.h>`, and the Linux pass has to read it, so on Windows it needs the Linux headers:

```
tiko c2t vendor/glfw/include/GLFW/glfw3.h \
    --sysroot //wsl.localhost/Ubuntu --verify
```

`//wsl.localhost/Ubuntu` is the root of the Ubuntu installation in WSL, as Windows sees it, and `GL/gl.h` comes from its Mesa development package (`sudo apt install libgl-dev`). Without `--sysroot`, c2t stops and says what it needs:

```
tiko-lang: error TKO0120: unable to import C header
vendor/glfw/include/GLFW/glfw3.h (x86_64-pc-linux-gnu)
  note: clang: vendor/glfw/include/GLFW/glfw3.h:241:12: fatal error:
  'GL/gl.h' file not found
  note: the Linux pass needs Linux's system headers: give --sysroot
  (on Windows, the WSL root, e.g. //wsl.localhost/Ubuntu)
```

(Three lines on the screen.)

Reading the Linux headers through WSL is slow, so the translation takes about twenty seconds. OpenGL itself is not translated: c2t maps `<GL/gl.h>` to tiko's own `GL/gl.bi`, which the toolchain ships.

Exit status 2, with twenty-two TODOs, and none of them is in this program's way. Sixteen are headers c2t has no counterpart for — Vulkan's, OpenGL ES's, macOS's

and GLU's — each under the `#if` that asks for it; four are functions whose parameters are Vulkan types, which Clang cannot parse without Vulkan's header; and two are the lines around a `GLAPIENTRY` definition that other OpenGL headers use. A program that uses GLFW with Vulkan would translate Vulkan's header first. This one draws with OpenGL, and loses nothing.

A window and an OpenGL context

GLFW's documentation shows this program first: a window, an OpenGL context, and a loop that clears the window until it is closed.

examples/ch30/glfw-window.tiko

```
#include once "vendor/glfw/include/GLFW/glfw3.ti"
#ifdef __FB_WIN32__
    #foreignlib GLFW static "vendor/glfw/glfw3", "system:opengl32", _
        "system:gdi32", "system:user32", "system:shell32"
#else
    #foreignlib GLFW static "vendor/glfw/glfw3", "system:GL", _
        "system:m"
#endif

'' GLFW reports every error here, as it happens.
sub OnError cdecl( byval code as int32, _
        byval description as const zstring ptr )
    print "GLFW error "; hex( code ); ": "; *description
end sub

dim maxFrames as int32 = valint( command( 1 ) )  '' 0: until closed

glfwSetErrorCallback( @OnError )
if glfwInit( ) = GLFW_FALSE then end 1

dim win as GLFWwindow ptr = _
    glfwCreateWindow( 640, 480, "Hello World", nullptr, nullptr )
if win = nullptr then
    glfwTerminate( )
    end 1
end if
glfwMakeContextCurrent( win )

dim frames as int32 = 0
do until glfwWindowShouldClose( win )
    glClear( GL_COLOR_BUFFER_BIT )
    glfwSwapBuffers( win )
    glfwPollEvents( )
    frames += 1
```

```
    if frames = maxFrames then exit do
loop
glfwTerminate( )

dim as int32 major, minor, revision
glfwGetVersion( @major, @minor, @revision )
print "GLFW "; str( major ); "."; str( minor ); "."; str( revision ); _
    ": drew "; str( frames ); " frames"
```

examples/ch30/glfw-window.out

```
GLFW 3.5.1: drew 60 frames
```

`glClear` and `GL_COLOR_BUFFER_BIT` are not GLFW's. They are OpenGL's, declared in `GL/gl.bi`, which `glfw3.ti` includes as `glfw3.h` includes `GL/gl.h`. But the code behind them is OpenGL's, in `opengl32.dll` on Windows and `libGL.so` on Linux, so the program links OpenGL itself: `"system:opengl32"` and `"system:GL"`. A program that called only GLFW would not need them.

`OnError` is a **callback** GLFW calls whenever something fails, with an error code and a description. `glfwSetErrorCallback( @OnError )` installs it before `glfwInit`, so that even a failure to start is explained, which is why GLFW's documentation sets it first.

The dynamic version keeps the OpenGL library, for the same reason:

examples/ch30/glfw-dll.tiko

```
#include once "vendor/glfw/include/GLFW/glfw3.ti"
#ifdef __FB_WIN32__
    #foreignlib GLFW dynamic "vendor/glfw/glfw3dll", "system:opengl32"
#else
    #foreignlib GLFW dynamic "vendor/glfw/glfw3dll", "system:GL"
#endif
```

> **NOTE**
> Through `opengl32.dll`, Windows offers only OpenGL 1.1, the version from 1997, which is where `glClear` comes from. The functions of every later version are looked up while the program runs, with `glfwGetProcAddress`, usually by a loader library such as glad. That is OpenGL's design on every platform, not something tiko adds.

SDL3

SDL, the *Simple DirectMedia Layer*, is the oldest and broadest of the four: windows, 2D rendering, a GPU interface, audio, gamepads, cameras, file dialogs, threads and

timers, on desktop and mobile platforms alike. Many commercial games and emulators are built on it. SDL3 is its current version.

Downloading

From `https://github.com/libsdl-org/SDL/releases`, release 3.4.18:

File	What is in it
`SDL3-devel-3.4.18-VC.zip`	for Visual C++: `include/SDL3/*.h`, and in `lib/x64/` the import library `SDL3.lib` and `SDL3.dll`
`SDL3-3.4.18.tar.gz`	the source, for Linux and for a static library on Windows
`SDL3-3.4.18-win32-x64.zip`	`SDL3.dll` alone, for people who run a program, not for building one

There is no static library in the Windows package, and no Linux package at all: SDL is built from its source for both. Into the project go `include/` as `vendor/sdl3/include/`, `SDL3.lib` as `vendor/sdl3/SDL3_windows_x86_64.lib`, and `SDL3.dll` beside the program.

On Linux, SDL's `docs/README-linux.md` lists the packages its build uses; it uses whatever it finds, so more packages give more features. Then:

```
cmake -S SDL3-3.4.18 -B build -DCMAKE_BUILD_TYPE=Release \
    -DSDL_STATIC=ON -DSDL_SHARED=ON
cmake --build build
```

`build/libSDL3.a` becomes `vendor/sdl3/SDL3-static_linux_x86_64.a`; `build/libSDL3.so.0` becomes `vendor/sdl3/SDL3_linux_x86_64.so` and, beside the program, `libSDL3.so.0`. To install it for every program, `sudo cmake --install build`; distributions newer than Ubuntu 24.04 package it as `libsdl3-dev`.

A library of headers

SDL's interface is not one header but dozens. `SDL.h` brings in fifty-seven others, and they include one another the way C programs include them:

```
#include <SDL3/SDL_stdinc.h>
```

c2t translates one header per run, so SDL is translated by a loop: start at `SDL.h`, visit each header it includes before the header itself, and translate them in that order, so that every header's includes are already translated when `--verify` compiles it. `-I` names SDL's `include` directory, where Clang finds `<SDL3/...>`:

```
```

```
tiko c2t vendor/sdl3/include/SDL3/SDL_stdinc.h -I vendor/sdl3/include \
    --sysroot //wsl.localhost/Ubuntu --verify
```

is one step of it; `translate.sh` has the whole loop. Two of SDL's headers are left out: `SDL_begin_code.h` and `SDL_close_code.h` only wrap the others in compiler decorations, which c2t recognises and strips where they are used, and neither can be parsed on its own.

`SDL_main.h` is not among `SDL.h`'s includes, and a program that uses SDL's main callbacks needs it. That makes fifty-nine translations in all, in under five minutes. In C it supplies the program's `main` function; a tiko program has its own beginning, so it is translated with SDL's macro for that case, `SDL_MAIN_HANDLED`, defined:

```
tiko c2t vendor/sdl3/include/SDL3/SDL_main.h -I vendor/sdl3/include \
    --sysroot //wsl.localhost/Ubuntu -D SDL_MAIN_HANDLED --verify
```

A `-D` is written into the translation as a `#define`, so `SDL_main.ti` reads the macro exactly as Clang did.

Twenty-six of the fifty-nine end with exit status 2, and their banners list 1,704 TODOs between them. The number looks worse than it is. 1,175 are in `SDL_oldnames.h`, a table of SDL2's names kept to help port SDL2 programs: in C it turns each old name into a compile error that names its replacement, and without it an old name is still an error in tiko, an undeclared one. Most of the rest are in `SDL_stdinc.h` and its neighbours, SDL's portable C library — `#if` tests on what the C compiler predefines, `printf` format macros, `alloca` — the plumbing SDL is built from rather than the interface a program calls. The TODOs that are API are a short list: twenty-eight `static inline` functions, among them `SDL_rect.h`'s rectangle tests such as `SDL_PointInRect`, quoted for porting. A program that needs one writes it in a residue file of its own.

Each `SDL3/X.h` gets an `SDL3/X.ti` beside it, and `#include <SDL3/SDL_stdinc.h>` became `#include once "SDL_stdinc.ti"`, relative to the header that includes it. So a program includes SDL by its path in the project and needs no include directory at all.

The main callbacks

SDL3 can run a program for you: instead of writing the main loop, you give SDL four functions, and it calls the first once at the start, the second for every event, the third once per frame, and the fourth at the end. It is SDL's recommended way to write a program, because it works the same on platforms where a program cannot own its main loop, such as phones and the web. SDL's example `renderer/01-clear` is written that way; here it is in tiko:

examples/ch30/sdl-clear.tiko

```
#include once "vendor/sdl3/include/SDL3/SDL.ti"
#include once "vendor/sdl3/include/SDL3/SDL_main.ti"
#foreignlib SDL3 dynamic "vendor/sdl3/SDL3"

'' Everything the callbacks share, reached through SDL's appstate.
type AppState
    window as SDL_Window ptr
    renderer as SDL_Renderer ptr
    frames as int32
    maxFrames as int32
end type

function AppInit cdecl( byval appstate as any ptr ptr, _
        byval argc as int32, byval argv as zstring ptr ptr ) _
        as SDL_AppResult
    SDL_SetAppMetadata( "Example Renderer Clear", "1.0", _
        "com.example.renderer-clear" )
    if SDL_Init( SDL_INIT_VIDEO ) = false then
        print "Couldn't initialize SDL: "; *SDL_GetError( )
        return SDL_APP_FAILURE
    end if
    var st = new AppState
    st->maxFrames = valint( command( 1 ) )   '' 0: until closed
    *appstate = st
    if SDL_CreateWindowAndRenderer( "examples/renderer/clear", _
            640, 480, SDL_WINDOW_RESIZABLE, _
            @st->window, @st->renderer ) = false then
        print "Couldn't create window/renderer: "; *SDL_GetError( )
        return SDL_APP_FAILURE
    end if
    SDL_SetRenderLogicalPresentation( st->renderer, 640, 480, _
        SDL_LOGICAL_PRESENTATION_LETTERBOX )
    return SDL_APP_CONTINUE
end function

function AppEvent cdecl( byval appstate as any ptr, _
        byval event as SDL_Event ptr ) as SDL_AppResult
    if event->type = SDL_EVENT_QUIT then return SDL_APP_SUCCESS
    return SDL_APP_CONTINUE
end function

function AppIterate cdecl( byval appstate as any ptr ) as SDL_AppResult
    var st = cast( AppState ptr, appstate )
```

```
    var now = SDL_GetTicks( ) / 1000.0
    var red = 0.5 + 0.5 * SDL_sin( now )
    var green = 0.5 + 0.5 * SDL_sin( now + SDL_PI_D * 2 / 3 )
    var blue = 0.5 + 0.5 * SDL_sin( now + SDL_PI_D * 4 / 3 )
    SDL_SetRenderDrawColorFloat( st->renderer, red, green, blue, _
        SDL_ALPHA_OPAQUE_FLOAT )
    SDL_RenderClear( st->renderer )
    SDL_RenderPresent( st->renderer )
    st->frames += 1
    if st->frames = st->maxFrames then return SDL_APP_SUCCESS
    return SDL_APP_CONTINUE
end function

sub AppQuit cdecl( byval appstate as any ptr, _
        byval result as SDL_AppResult )
    var st = cast( AppState ptr, appstate )
    if st <> nullptr then
        print "drew "; str( st->frames ); " frames"
        delete st
    end if
end sub

'' SDL calls AppInit once, then AppEvent and AppIterate until one of
'' them returns something other than SDL_APP_CONTINUE, then AppQuit.
var rc = SDL_EnterAppMainCallbacks( 0, nullptr, @AppInit, _
    @AppIterate, @AppEvent, @AppQuit )
end rc
```

examples/ch30/sdl-clear.out

```
drew 60 frames
```

The connection to SDL comes first:

examples/ch30/sdl-clear.tiko

```
#include once "vendor/sdl3/include/SDL3/SDL.ti"
#include once "vendor/sdl3/include/SDL3/SDL_main.ti"
#foreignlib SDL3 dynamic "vendor/sdl3/SDL3"
```

- `SDL.ti` brings the whole of SDL's interface, through the `.ti` files it includes.
- `SDL_main.ti` declares `SDL_EnterAppMainCallbacks`, the function that runs the callbacks. It is what `SDL_main.h`'s `main` would have called, and the program calls it at the end, passing the four procedures by address.

The four callbacks are `cdecl` and take exactly the parameters SDL's header gives their types, `SDL_AppInit_func` and the rest. Get one wrong, say `int32 ptr` where SDL

passes `void *`, and the compiler says so with warning `TKO0833: passing different pointer types`, but builds the program anyway. Treat that warning as an error: called with the wrong parameter types, a callback reads garbage.

State without globals. The callbacks share a window and a renderer, and a C program would keep them in global variables, as SDL's example does. SDL offers a better place: `AppInit` gets the address of a pointer, `appstate`, and whatever it stores there is passed to every later callback. The listing creates one `AppState` with `new`, stores it, and `AppQuit` deletes it.

`bool` results. SDL3's functions return C's `bool`, which the translation declares `boolean`, and the listing compares them with `false`: `if SDL_Init( SDL_INIT_VIDEO ) = false then`.

Constants. `SDL_PI_D` and `SDL_ALPHA_OPAQUE_FLOAT` are `#define`s in SDL's headers with number values, and they are `#define`s in the translation, with the same values.

> **COMING FROM FREEBASIC**
> SDL reads the environment variable `SDL_VIDEO_DRIVER` to choose its video system, and the value `dummy` gives it one that draws nowhere. A program that opens windows can then run where there is no screen, such as on a build server, and still exercise all of its code: `SDL_VIDEO_DRIVER=dummy ./sdl-clear 60`.

SDL linked statically

SDL's Windows package has no static library, so build one from the source archive, with CMake and the Visual Studio Build Tools:

```
cmake -S SDL3-3.4.18 -B build -DSDL_STATIC=ON -DSDL_SHARED=OFF
cmake --build build --config Release
```

`build/Release/SDL3-static.lib`, 13 MB, becomes `vendor/sdl3/SDL3-static_windows_x86_64.lib`. SDL uses more of Windows than the other three, and linked statically it needs five system libraries, found by the method from *A static library brings its dependencies with it*:

examples/ch30/sdl-static.tiko

```
#include once "vendor/sdl3/include/SDL3/SDL.ti"
#include once "vendor/sdl3/include/SDL3/SDL_main.ti"
#ifdef __FB_WIN32__
    #foreignlib SDL3 static "vendor/sdl3/SDL3-static", _
        "system:cfgmgr32", "system:ole32", "system:oleaut32", _
        "system:setupapi", "system:shell32"
#else
```

```
    #foreignlib SDL3 static "vendor/sdl3/SDL3-static"
#endif
```

On Linux the static library needs nothing more: SDL finds the window system and the sound system while the program runs, rather than linking them. On both platforms the program now runs with no `SDL3.dll` or `libSDL3.so.0` beside it.

A binding header

Each program so far has named its library's files itself. That is fine for one program, but it repeats in each, with the platform's system libraries every time, and it ties every program to where the files are. Move those lines into a header of their own, a **binding header**, and a program needs one line:

examples/ch30/vendor/raylib-binding.ti

```
'' raylib-binding.ti -- raylib 6.0 for tiko: its translated header,
'' and the library to link, in one file a program includes. Both paths
'' are relative to this file, so the folder it sits in can live
'' anywhere: inside one project, or in a single place that every
'' program shares.
#pragma once

#include once "raylib/raylib.ti"

#ifdef __FB_WIN32__
    #foreignlib RAYLIB static "raylib/raylib", _
        "system:winmm", "system:gdi32", "system:user32", _
        "system:shell32"
#else
    #foreignlib RAYLIB static "raylib/raylib", _
        "system:GL", "system:m", "system:pthread", "system:dl", _
        "system:rt", "system:X11"
#endif
```

Both paths in `raylib-binding.ti` are relative to the binding header itself: `#include` looks beside the file that includes, and `#foreignlib` always resolves that way. So the folder holding `raylib-binding.ti` and `raylib/` can be anywhere: inside one project, or in one place that every program on the machine shares.

A program includes it by name, and the project says where the folder is with `-i`, which adds a directory to the ones `#include` searches:

examples/ch30/game/tiko.toml

```
```

```
# A program that uses raylib without saying where raylib is: the
# -i below names the folder of bindings, relative to this file.
[package]
name = "game"

[target.game]
main    = "main.tiko"
options = ["-i", "../vendor"]
```

examples/ch30/game/main.tiko

```
#include once "raylib-binding.ti"

const ScreenWidth = 800
const ScreenHeight = 450
dim maxFrames as int32 = valint( command( 1 ) )  '' 0: until closed

SetTraceLogLevel( LOG_WARNING )
InitWindow( ScreenWidth, ScreenHeight, _
    "raylib [core] example - keyboard input" )
dim ball as Vector2 = type( ScreenWidth / 2, ScreenHeight / 2 )
SetTargetFPS( 60 )

dim frames as int32 = 0
do until WindowShouldClose( )
    if IsKeyDown( KEY_RIGHT ) then ball.x += 2
    if IsKeyDown( KEY_LEFT ) then ball.x -= 2
    if IsKeyDown( KEY_UP ) then ball.y -= 2
    if IsKeyDown( KEY_DOWN ) then ball.y += 2

    BeginDrawing( )
    ClearBackground( RAYWHITE )
    DrawText( "move the ball with arrow keys", 10, 10, 20, DARKGRAY )
    DrawCircleV( ball, 50, MAROON )
    EndDrawing( )
    frames += 1
    if frames = maxFrames then exit do
loop
CloseWindow( )
print "ball at "; str( ball.x ); ", "; str( ball.y ); _
    " after "; str( frames ); " frames"
```

`tiko build game`, then `out/debug/game 60`:

examples/ch30/game/main.out

```
```

```
ball at 400, 225 after 60 frames
```

This is raylib's example `core_input_keys`, which moves a ball with the arrow keys. Its `ball` is a `Vector2`, a struct of two `float`s, filled with `type( … )` and passed by value to `DrawCircleV`; `KEY_RIGHT` and the other keys are the values of a C `enum`.

That is the Windows answer to "install once for every program". Put each library's folder and its binding header in one place, say `C:\tiko-libs`:

```
C:\tiko-libs\raylib-binding.ti
C:\tiko-libs\raylib\raylib.ti
C:\tiko-libs\raylib\raylib_windows_x86_64.lib
```

Then every project that uses raylib writes `#include once "raylib-binding.ti"` and adds `options = ["-i", "C:/tiko-libs"]` to its `tiko.toml`, or `-i C:\tiko-libs` on the command line. Notice what is not there: the C header. Once translated, it is no longer needed to build. For a library linked dynamically, its DLL still has to be found when the program runs: copy it beside each program, or add its folder to `PATH`.

A binding module

A binding can also be a **module**, which Chapter 15 introduced: a directory a program `#import`s. A module's names are its own unless it marks them `public`, and that is true of an `#include` inside one too. Written as **`public #include`**, the header travels with the module, whole, and every program that imports it sees its names exactly as if it had included the header itself:

examples/ch30/modules/raylib/raylib.tiko

```
'' raylib as a module. 'public #include' makes the names of raylib.ti,
'' the translation of raylib.h, the names of every program that imports
'' the module, and the module's own #foreignlib links the library into
'' each of them. Both paths are relative to this file.
#module "raylib"

public #include once "../../vendor/raylib/raylib.ti"

#ifdef __FB_WIN32__
    #foreignlib RAYLIB static "../../vendor/raylib/raylib", _
        "system:winmm", "system:gdi32", "system:user32", _
        "system:shell32"
#else
    #foreignlib RAYLIB static "../../vendor/raylib/raylib", _
        "system:GL", "system:m", "system:pthread", "system:dl", _
        "system:rt", "system:X11"
#endif
```

examples/ch30/modgame/tiko.toml

```
# A program that imports raylib as a module: the -i below names the
# folder that holds module folders, relative to this file.
[package]
name = "modgame"

[target.modgame]
main    = "main.tiko"
options = ["-i", "../modules"]
```

examples/ch30/modgame/main.tiko

```
#import "raylib"

dim maxFrames as int32 = valint( command( 1 ) )  '' 0: until closed

SetTraceLogLevel( LOG_WARNING )
InitWindow( 800, 450, "raylib [core] example - basic window" )
SetTargetFPS( 60 )

dim frames as int32 = 0
do until WindowShouldClose( )
    BeginDrawing( )
    ClearBackground( RAYWHITE )
    DrawText( "Congrats! You created your first window!", _
        190, 200, 20, LIGHTGRAY )
    EndDrawing( )
    frames += 1
    if frames = maxFrames then exit do
loop
CloseWindow( )
print "drew "; str( frames ); " frames"
```

`tiko build modgame`, then `out/debug/modgame 60`:

examples/ch30/modgame/main.out

```
drew 60 frames
```

The program writes `#import "raylib"` and nothing else: `InitWindow`, the `LOG_WARNING` constant and the `RAYWHITE` macro come from the module, and the module's `#foreignlib` links the library into the program. The names are bare, as an `#include` would make them, not `raylib.InitWindow`, because a C header's `#define`s and struct macros are preprocessor names, which no namespace can hold.

This is exactly the shape of the toolchain's vendor packages. `vendor/sqlite3` is a module whose file holds a `public #include` of its translated `sqlite3.ti`, another of

its hand-written `sqlite3-residue.ti`, and a `#foreignlib` for each link mode, chosen by the `link` its importer wrote. Chapter 28's one line, `#import "vendor/sqlite3"`, is this section's binding module, made by the people who ship tiko.

A plain `#include` in a module stays private: the module can use a C library to do its work without handing the whole C interface to everyone who imports it. `public #include` outside a module file is `TKO0369`.

To install a binding module for every program, put its folder in one directory and name that directory in **`TIKO_PATH`**, the list of directories `#import` searches, separated by `;` on Windows and `:` on Linux. Because a translated library's headers include one another by relative path, any library can be a module this way, SDL's fifty-nine headers included.

When something goes wrong

Every failure in this table happened while this chapter was being written.

What you see	What it means	What to do
c2t: `TKO0120: unable to import C header`, `fatal error: 'X.h' file not found`	Clang cannot find a header the header includes	for one of the library's own, add `-I`; for a system header in the Linux pass on Windows, `--sysroot`
c2t exits with status 2	the translation has TODOs	read the banner's list; finish what you need in a file of its own
c2t: `TKO0120` on a header that `included without matching` another	the header cannot stand alone	leave it out: it is meant only to be included
`TKO0329: #foreignlib found no matching library file`, with a list of names tried	no file has any of the names in the list	rename the library to one of them, usually with its target tag
`TKO0172: cannot make an import library for '…': it is not a 64-bit x86-64 DLL`	the DLL is the 32-bit build	download the 64-bit package
`undefined symbol: …` when linking statically	a system library the static library needs is not linked	add it as `"system:name"`; see *A static library brings its dependencies with it*
`undefined symbol:` a variable, when linking to a DLL	the program reads a variable the DLL exports	translate with `--shared-macro`; define `TIKO_LINK_DYNAMIC` before the include
the program exits at once, code `-1073741515` (`0xC0000135`)	Windows cannot find a DLL, or the import library names none	put the DLL beside the program; check the import library has the DLL's name
`error while loading shared libraries: libX.so.N` and exit 127	Linux cannot find the `.so`	`LD_LIBRARY_PATH=.`, an rpath, or install the library
the program runs but reports an older version of the library	Linux found the system's copy, not yours	`LD_LIBRARY_PATH=.` or an rpath; ask the library its version
a C library prints pages of messages	many libraries log to standard output	look for a log level in the library: raylib's is `SetTraceLogLevel`
`warning TKO0833: passing different pointer types` where a callback is passed	the callback's parameters differ from the C type	make them match the translated declaration exactly

What it costs

- **Calling a C library costs what calling C costs**, as in Chapter 29: no wrapper, no conversion beyond what the C function itself would need.
- **Translating costs seconds, once.** `sqlite3.h` took three seconds, `glfw3.h` twenty, through WSL; SDL's fifty-nine headers, run by the loop, under five minutes. A build after that reads a `.ti` like any tiko header and runs no Clang.
- **A translation costs upkeep**: it belongs to one version of the library, and an upgrade means translating again and reading the diff.
- **Static linking costs size**: the table in *Static or dynamic* shows from 200 KB to 2.7 MB added to each program. **Dynamic linking costs a second file** that must be shipped and found.
- **A vendored library costs space in every project** that uses it, and keeping it current is your job. **An installed library costs reproducibility**: the program is built and run with whatever version the machine has.

Summary

- A C library comes as a header, a static library, a shared library and, on Windows, an import library. On Windows take the 64-bit Visual C++ package.
- **`tiko c2t x.h --verify`** translates a header once into `x.ti`, for both targets in one file, laid out like the C, with layouts checked against Clang's. Only the person translating needs Clang, the Visual Studio Build Tools and, on Windows, a Linux sysroot through `--sysroot`.
- The **banner** names the header by SHA-256 and lists every rename and every **TODO**. Exit 2 means TODOs: read them, and write what you need in a file of its own, never in the `.ti`.
- One header per run; includes become relative `#include once "x.ti"`, so a library's translations find each other with no include directory. `-I` and `-D` pass C's options; `--shared-macro` writes a DLL's variables for `TIKO_LINK_DYNAMIC`.
- **`#foreignlib NAME static|dynamic "path"`** links the code, relative to its own file, choosing `name_windows_x86_64.lib` or `name_linux_x86_64.a` by target. A static link also needs the system libraries the library calls, each as `"system:name"`.
- Linked **dynamically**, the library must be found when the program starts: beside the `.exe` on Windows; on Linux, in the system's directories, on `LD_LIBRARY_PATH`, or in an rpath: `-Wl -rpath,$ORIGIN` records "beside the program". A Linux program asks for its library by **soname**, and a missing one may silently be replaced by the system's own copy.
- A **vendored** library lives in the project: commit the `.ti`, fetch the binaries. An **installed** library on Linux is linked with `"system:x"`; its header still gets translated.

- **SQLite**: compile the amalgamation for a static library; its DLL needs no import library. **raylib**: prebuilt, translates clean. **GLFW**: needs the Linux headers to translate; its TODOs are Vulkan's and GLES's. **SDL3**: fifty-nine headers, translated by a loop in include order.
- A **binding header** holds a library's `#include` and `#foreignlib`, relative to itself. A **binding module** does the same with `public #include`, and is installed through `TIKO_PATH` — which is exactly what a vendor package is.

Exercises

1. Run `sqlite-quick` with a file name instead of `:memory:`, once to create a table and insert a row, and again to select it. Then open the file with SQLite's own `sqlite3` command-line tool, from the *precompiled binaries* on the download page.
2. Link `sqlite-quick` statically and compare its size and its start-up with the dynamic version. On Linux, run both without `LD_LIBRARY_PATH` and explain the difference in what they print with a `select sqlite_version()`.
3. Translate `raymath.h`, which is in `vendor/raylib/` beside `raylib.h`, and read its TODOs. Which kind of C is most of them, and which of raymath's functions can a program still call?
4. In `raylib-window.tiko`, draw the frame number in the window with raylib's `TextFormat`, which is variadic like C's `printf`. What type must the argument have for `%i`?
5. Add a key callback to `glfw-window.tiko` with `glfwSetKeyCallback`, so that Escape closes the window by calling `glfwSetWindowShouldClose`.
6. Make `sdl-clear.tiko`'s `AppEvent` return `SDL_APP_SUCCESS` when Escape is pressed: an event whose `type` is `SDL_EVENT_KEY_DOWN` carries the key in `event->key.key`, to compare with `SDLK_ESCAPE`.
7. Make SDL a binding module, the way `modules/raylib` makes raylib one, and a `tiko build` project that uses it to run `sdl-clear`. Which `-i` does the project need, and why does it need nothing for SDL's fifty-nine headers?

Building and Shipping

Every program so far was built the same way: one command, one source file, one executable. Real projects outgrow that. They are split into parts compiled separately, they produce libraries for other programs, they need a Windows program with no console window or an icon, a debug build and a fast one, and in the end a set of files that runs on someone else's machine. This chapter is about the command line that does all of that, and about what a shipped program depends on.

Targets

tiko builds for three targets, chosen with `-target`:

Target	Builds	Default on
`windows-x86_64`	a Windows x86-64 executable, linked against Microsoft's Universal C Runtime	Windows
`linux-x86_64`	a Linux x86-64 executable, linked against the system's C library	Linux
`macos-arm64`	a macOS executable for Apple Silicon, linked by Apple's linker against the macOS SDK	macOS

The compiler turns tiko into LLVM's intermediate form and hands it to **clang**, which produces machine code and drives the linker. tiko looks for clang in the place LLVM installs it, then on the `PATH`; the environment variable **`TIKO_CLANG`** names a particular one instead. Its runtime library, the code behind `print`, strings and files, comes with the compiler, under its `lib` directory; **`TIKO_LIBPATH`** points it at another copy, and **`TIKO_INCLUDE`** does the same for the headers.

examples/ch31/target.tiko

```
'' What the compiler says about the target, through predefined symbols.
#ifdef __FB_WIN32__
    print "platform: Windows"
#endif
#ifdef __FB_LINUX__
    print "platform: Linux"
#endif
#ifdef __FB_DARWIN__
    print "platform: macOS"
```

```
#endif
#ifdef __FB_MSVC__
    print "C runtime: Microsoft's Universal CRT"
#else
    print "C runtime: the system's C library"
#endif
print "pointer size:"; sizeof( any ptr ); " bytes"
print "backend: "; __FB_BACKEND__
```

On Windows:

examples/ch31/target.out

```
platform: Windows
C runtime: Microsoft's Universal CRT
pointer size: 8 bytes
backend: llvm
```

On Linux:

examples/ch31/target.linux-x86_64.out

```
platform: Linux
C runtime: the system's C library
pointer size: 8 bytes
backend: llvm
```

On macOS:

examples/ch31/target.macos-arm64.out

```
platform: macOS
C runtime: the system's C library
pointer size: 8 bytes
backend: llvm
```

The predefined symbols are how a program adapts to its target: `__FB_WIN32__`, `__FB_LINUX__` and `__FB_DARWIN__` for the platform, `__FB_MSVC__` when the C runtime is Microsoft's, `__FB_BACKEND__` for the code generator. Code inside an `#ifdef` that does not apply is never compiled, so it may call functions that do not exist on the other platform.

Separate compilation

examples/ch31/separate/greet.tiko

```
'' Compiled on its own with 'tiko -c greet.tiko'; main.tiko calls it.
sub Greet( byref who as ustring )
```

```
    print "hello, "; who
end sub

'' The top level of a module that is not main runs first, as a module
'' constructor.
print "greet.tiko is set up"
```

examples/ch31/separate/main.tiko

```
'' Linked with the object file greet.o: 'tiko main.tiko greet.o'.
declare sub Greet( byref who as ustring )

print "main starts"
Greet( "world" )
```

examples/ch31/separate/main.out

```
greet.tiko is set up
main starts
hello, world
```

`tiko -c greet.tiko` compiles without linking and writes `greet.o`, an **object file**. **`tiko main.tiko greet.o`** compiles `main.tiko` and links the two. An object file is recompiled only when its source changes, so a large program is rebuilt a part at a time — the same job the module system (Chapter 15) does by itself for code split into modules.

One module owns the program's **main** code, the top level that runs as the program. By default it is the **first file named on the command line**; `-m name` chooses another. Every other module's top-level code becomes a **module constructor** and runs before main, which is why `greet.tiko is set up` comes first. `tiko greet.tiko main.tiko`, with the files the other way round, makes `greet.tiko` the main module: `main.tiko`'s lines run first and `greet.tiko is set up` last.

> **PITFALL**
> **With `-c`, no module is main unless `-m` says so.** Compiling every file with `-c` and linking only object files ends in a linker error about a missing entry point — on Windows, `lld-link: error: subsystem must be defined`. Compile the main module with `-c -m name`, or leave it for the final command, as `main.tiko` is here.

Static libraries

examples/ch31/static/mathlib.tiko

```
'' A small library, built with 'tiko -lib mathlib.tiko'.
function Gcd( byval a as int64, byval b as int64 ) as int64
    do while b <> 0
        dim as int64 t = a mod b
        a = b
        b = t
    loop
    return abs( a )
end function

function Factorial( byval n as int32 ) as int64
    dim as int64 r = 1
    for i as int32 = 2 to n
        r *= i
    next
    return r
end function
```

examples/ch31/static/use-static.tiko

```
'' Links libmathlib.a, found in the current directory.
#inclib "mathlib"

declare function Gcd( byval a as int64, byval b as int64 ) as int64
declare function Factorial( byval n as int32 ) as int64

print "Gcd( 84, 36 ) ="; Gcd( 84, 36 )
print "Factorial( 20 ) ="; Factorial( 20 )
```

examples/ch31/static/use-static.out

```
Gcd( 84, 36 ) = 12
Factorial( 20 ) = 2432902008176640000
```

`tiko -lib mathlib.tiko` compiles the file and packs the result into a **static library**, `libmathlib.a`, on both targets. A program names it with `#inclib "mathlib"` (or `-l mathlib`), without the `lib` prefix or the extension, and the linker copies in the parts the program uses. The program then carries that code inside it, and runs with nothing more beside it.

The library's user needs a declaration of each procedure, written out as `use-static.tiko` does or, better, kept in a header (`mathlib.ti`) that the library ships beside itself and every user includes. The linker searched the current directory for `libmathlib.a`; `-p dir` or `#libpath "dir"` adds others, searched first (Chapter 29).

Shared libraries

examples/ch31/shared/mathlib.tiko

```
'' The same library as a DLL or shared object: 'tiko -dll mathlib.tiko'.
'' 'export' puts a procedure in the library's table of exports.
function Gcd( byval a as int64, byval b as int64 ) as int64 export
    do while b <> 0
        dim as int64 t = a mod b
        a = b
        b = t
    loop
    return abs( a )
end function

function Factorial( byval n as int32 ) as int64 export
    dim as int64 r = 1
    for i as int32 = 2 to n
        r *= i
    next
    return r
end function
```

examples/ch31/shared/use-shared.tiko

```
'' Links mathlib.dll through libmathlib.dll.a on Windows, libmathlib.so
on
'' Linux, or libmathlib.dylib on macOS. The library must be found again
at
'' run time.
#inclib "mathlib"

declare function Gcd( byval a as int64, byval b as int64 ) as int64
declare function Factorial( byval n as int32 ) as int64

print "Gcd( 84, 36 ) ="; Gcd( 84, 36 )
print "Factorial( 20 ) ="; Factorial( 20 )
```

examples/ch31/shared/use-shared.out

```
Gcd( 84, 36 ) = 12
Factorial( 20 ) = 2432902008176640000
```

`tiko -dll mathlib.tiko` (or `-dylib`) builds a **shared library**: one file that every program using it loads when it starts, so it is stored once and can be replaced without rebuilding the programs. On Windows that file is `mathlib.dll`, and beside it tiko writes `libmathlib.dll.a`, an **import library** that tells the linker what the DLL

contains. On Linux it is `libmathlib.so`, and on macOS `libmathlib.dylib`. The program links it exactly as it linked the static one, with `#inclib "mathlib"`.

`export` after a procedure's declaration puts it in the library's table of exported names. On Windows only exported procedures can be called from outside the DLL: built without `export`, the client fails to link with `undefined symbol: GCD`. Mark every procedure that is part of the library's interface, and nothing else.

> **PITFALL**
> **A shared library must be found again when the program runs.** Linking only records its name. Windows looks in the program's own directory, the system directories, the current directory and the `PATH`. Linux looks in the system's library directories and in `LD_LIBRARY_PATH`, and a tiko executable records no directory of its own — run from beside the library, the client still fails with `error while loading shared libraries: libmathlib.so: cannot open shared object file`. This chapter's listing runs it with `LD_LIBRARY_PATH=.`; a shipped program installs the library where the system looks, or ships a script that sets the variable. macOS loads the library by the name it gives itself, `@rpath/libmathlib.dylib`, and stops the program with `dyld: Library not loaded` until it is found: `DYLD_LIBRARY_PATH=.` does that, as the listing does, and so does a run path in the program, `#cmdline "-Wl -rpath,@loader_path"`, which keeps the library beside it.

A variable that a C DLL exports is reached with **`extern import`**, which tells the compiler to go through the pointer Windows fills in when the DLL loads: `extern import MyDll_Data alias "MyDll_Data" as int32`.

Against a C DLL exporting `int MyDll_Data = 0x1234`, that line reads `&h1234`, and a write or an `@` reaches the DLL's own copy. On Linux, `extern import` compiles as a plain `extern`.

Windows programs

`examples/ch31/gui.tiko`

```
'' A Windows program with no console window: the GUI subsystem.
#cmdline "-s gui"
#include once "windows.bi"

'' A real program would create a window here. This one shows that it
'' was started without a console, and writes to its standard output,
'' which still works when something has redirected it.
print "has a console window: "; cbool( GetConsoleWindow( ) <> nullptr )
```

examples/ch31/gui.out

```
has a console window: false
```

A Windows executable belongs to a **subsystem**. The **console** subsystem, the default, gets a console window when it is started from the desktop, and shares the one it was started from in a terminal. **`-s gui`** builds for the **GUI** subsystem instead: no console appears, which is what a program with windows of its own wants. Its `print` output goes nowhere unless something has redirected it, as the book's test harness does here. On Linux there is no such distinction, and `-s gui` is accepted and changes nothing.

Resources

examples/ch31/resources/strings.rc

```
// strings.rc -- a resource script, compiled into the executable.
#define IDS_GREETING 101

STRINGTABLE
BEGIN
    IDS_GREETING "Hello from a resource"
END
```

examples/ch31/resources/strings.tiko

```
'' Built with 'tiko strings.tiko strings.rc': the .rc is compiled by
'' llvm-rc and linked into the executable.
#include once "windows.bi"

const IDS_GREETING = 101

dim as wstring * 64 buf
var n = LoadStringW( GetModuleHandle( nullptr ), IDS_GREETING, _
                     @buf, 64 )
dim as ustring text = buf
print "string"; IDS_GREETING; ","; n; " characters: "; text
```

examples/ch31/resources/strings.out

```
string 101, 21 characters: Hello from a resource
```

A Windows executable can carry **resources** inside it: icons, version information, dialog layouts, string tables, the manifest that asks for modern controls and high-DPI awareness. They are written in a **resource script**, a `.rc` file, and named on the command line with the sources — `tiko strings.tiko strings.rc`. tiko compiles it with `llvm-rc`, which comes with LLVM, and links the result in. The program reads

a string table entry back with the Windows function `LoadStringW`; an `ICON` line gives the executable the icon Explorer shows.

Build profiles

The same source can be built for finding bugs or for speed.

examples/ch31/checked.tiko

```
'' A checked build: -exx adds bounds and null-pointer checks.
#cmdline "-exx"

dim scores( 1 to 3 ) as int32 = { 70, 85, 90 }
dim as int32 i = 4
dim as int32 total = scores( i )
```

examples/ch31/checked.out

```

Aborting due to runtime error 6 (out of bounds array access) at line 6
of checked.tiko::(),
'SCORES' accessed with invalid index = 4, must be between 1 and 3
```

(The program ends with exit code 6.)

Option	Does
`-e`	checks for runtime errors: a failed `open` stops the program with a message
`-exx`	`-e`, plus array bounds and null-pointer checks, and where each error happened
`-earray`, `-enullptr`, `-elocation`	one part of `-exx` at a time
`-g`	debug information for a debugger, and turns `assert` on
`-O0` … `-O3`	how hard the optimiser works; `-O0`, none, is the default
`-mt`	the thread-safe runtime, needed with threads (Chapter 23)

A **checked build** (`-exx -g`) stops at the first bad index with its line and the bounds, as above. Without `-exx` nothing checks the index: the program reads whatever memory lies past the end of the array, and carries on with it. A **release build** (`-O2`, or `-O3`) leaves the checks out and lets the optimiser rearrange the code; it runs faster, and a bug that the checks would have caught becomes a wrong answer or a crash somewhere else. Test with the checks on, ship with them off — or, for a program where a wrong answer costs more than speed buys, ship checked.

Build settings in the source

examples/ch31/profile.tiko

```
'' Build settings can live in the source, so the file alone says how
'' it must be built.
#cmdline "-mt -O2"

print "thread-safe runtime: "; cbool( __FB_MT__ )
#if __FB_DEBUG__
    print "debug build"
#else
    print "release build"
#endif
```

examples/ch31/profile.out

```
thread-safe runtime: true
release build
```

`#cmdline "options"` puts compiler options in the source file, so the file alone says how it must be built and nobody has to remember. When the compiler meets it, it starts the compilation again with those options added. Two rules: it must come **after** any `#import` — an `#import` below it is `TKO0362: '#import' must come before any code` — and it counts only in the **first** source file named on the command line; anywhere else it is warning `TKO0334` and ignored. Every listing in this book that needs an option carries it this way. `__FB_ERR__`, `__FB_MT__` and `__FB_DEBUG__` let the code see what it was given.

`-x name` names the program, as `-o` does: `tiko hello.tiko -x greeter.exe` writes `greeter.exe`. The name is used exactly as written, so give the extension yourself. Where the two meet, `-x` wins over `-o`, and of several `-x` the last one counts. With `-c` it is ignored, and each object file keeps its own name. In `#cmdline` it lets the source name its own program:

examples/ch31/named.tiko

```
'' The source names its own program. -x in #cmdline wins over any -o
'' or -x on the command line, so 'tiko named.tiko' writes renamed.exe.
#cmdline "-x renamed.exe"

print "built as renamed.exe"
```

examples/ch31/named.out

```
built as renamed.exe
```

`tiko named.tiko` writes `renamed.exe`. So does `tiko named.tiko -x other.exe`: like every option, the one in the source wins over the command line's.

Projects

The command lines in this chapter grow with the program. Every file, every option and the name of the output are typed again for each build, and the compiler starts from nothing each time. A **project** writes them down once, in the `tiko.toml` that Chapter 15 used for packages, and **`tiko build`** reads it.

examples/ch31/project/tiko.toml

```
# One program, built from main.tiko and the module it imports.
[package]
name = "greeter"

[target.greeter]
main = "main.tiko"

[target.greeter.defines]
TIMES = "2"
```

examples/ch31/project/main.tiko

```
'' The main file of a project. tiko.toml beside it says how to build
'' it, so 'tiko build' needs no file names and 'tiko run' no -o.
#import "./text"

dim as ustring who = "world"
if len( command( 1 ) ) > 0 then who = command( 1 )
for i as int32 = 1 to TIMES
    print text.Greeting( who )
next
#if __FB_DEBUG__
    print "debug build"
#else
    print "release build"
#endif
```

examples/ch31/project/text/text.tiko

```
#module "text"

public function Greeting( byref who as ustring ) as ustring
    return "Hello, " & who & "!"
end function
```

`tiko run -- tiko`, in the project's directory:

`examples/ch31/project/main.out`

```
Hello, tiko!
Hello, tiko!
debug build
```

`[target.greeter]` declares a **target**: something the project builds, here an executable called `greeter` whose main file is `main.tiko`. **`tiko build`**, run in the directory that holds `tiko.toml`, builds every target the file declares. It takes no file names, because the target names them, and no `-o`: the program is written to `out/debug/greeter.exe` on Windows and `out/debug/greeter` on Linux, beside `tiko.toml`. The module that `main.tiko` imports is found and compiled as in Chapter 15. **`[target.greeter.defines]`** passes `TIMES` as though `-d TIMES=2` had been typed, which is why the greeting appears twice. `tiko build greeter` builds the one target by name.

`tiko run` builds the project's one executable target and runs it. Everything after `--` goes to the program, so `tiko run -- tiko` greets `tiko`, and the exit code is the program's. A project with more than one executable says which: `tiko run greeter -- tiko`.

A target takes these keys:

Key	Does
`main`	the main file, which owns the program's top-level code; required
`sources`	more files compiled with it, as though named after it on the command line
`kind`	`"exe"`, the default, `"dll"` or `"lib"` — `-dll` and `-lib`
`options`	any further compiler options, `["-mt"]`
`libs`, `libpaths`	libraries to link and where to look — `-l` and `-p`
`resources`	`.rc` files, on Windows
`subsystem`	`"gui"` or `"console"` — `-s`
`output`	where to write the result, instead of `out/<profile>/<name>`

Keys that hold for one platform only go in a **`[target.greeter.windows]`**, **`[target.greeter.linux]`** or **`[target.greeter.macos]`** table. It takes `sources`, `resources`, `libs`, `libpaths` and `options`, which add to the target's own in a build for that platform and are skipped in any other. A program's icon is the usual case:

its `.rc` file goes under `windows`, because a Linux build refuses a resource script, and the project then builds on both.

Every path in `tiko.toml` is relative to `tiko.toml`, an `-i` or `-p` inside `options` included, so the project builds the same whatever directory the command is typed in.

Profiles

The `debug` in `out/debug` is the **profile**: a named set of compiler options. `tiko build` uses `debug`, which is `-g -exx -O0`, a checked build, and so does `tiko run`. `--profile release` builds with `-O2` instead, into `out/release`, and the program above then prints `release build`. Both are built in; a `[profile.name]` table changes one or declares another:

```
[profile.fast]
inherits = "release"
options  = ["-mt"]
```

`inherits` starts from another profile; `opt` sets the optimiser level (`0` to `3`), `errors` the checks (`"e"`, `"ex"`, `"exx"` or `""`), `debug` whether `-g` is given, and `options` adds the rest. Each profile has its own directory under `out`, so a debug build and a release build of the same project sit side by side.

What it does not do twice

`tiko build` remembers what it compiled. Each main file, with its options, is stored in the same cache that holds compiled modules (Chapter 15), and a second `tiko build` with nothing changed compiles nothing; a third does not even link. A change to one file recompiles that file. A change inside a module's procedures recompiles that module, but not the files that import it, because what they see of it — its public declarations — is unchanged. Files that miss are compiled in parallel; `-j 4` sets how many at once.

`tiko check` reports every error and warning a build would, and writes nothing — no program, no objects. A file the last build compiled cleanly is not looked at again, so after a build a check is quick. It is what an editor runs on save. **`tiko clean`** removes the program a build of the same profile wrote.

> **FOR EDITORS AND TOOLS**
> `tiko build --message-format json` reports a build as one JSON object per line — each file compiled or found in the cache, each program written, the totals at the end — and `tiko metadata` prints the whole project, targets, profiles and modules, without compiling. An editor's build task, a language server and a CI script read these instead of `tiko.toml`.

Shipping

A program built for `windows-x86_64` needs only what every supported Windows already has:

```
DLL Name: ucrtbase.dll
DLL Name: KERNEL32.dll
```

— the Universal C Runtime and the Windows kernel interface. tiko's own runtime is linked into the executable, so there is no tiko DLL to ship: the executable, plus any DLLs and data files of its own, is the whole program. On Linux the same program needs `libc.so.6`, `libm.so.6` and `libtinfo.so.6`, the C library, the maths library and the terminal database, which every desktop and server distribution installs. A program built on one Linux distribution runs on others with the same or a newer C library — build on the oldest one to be supported. On macOS a tiko program needs `libSystem` and `libncurses`, which are part of the system, and runs on macOS 13 or newer.

> **ON MACOS**
> **A program that reaches another Mac through a browser arrives quarantined.** tiko's programs are signed ad hoc by Apple's linker, not notarised, and a quarantined one is refused: it does not start. Clearing the mark lets it run: `xattr -d com.apple.quarantine program`. A program copied with `scp`, a USB stick or `curl` is not marked.

The runtime's licence travels with the executable, because its code is inside. tiko's runtime is licensed under the LGPL 2.1 with a **static-linking exception**, the same terms as FreeBASIC's runtime: a program compiled by tiko and linked with it may be distributed under any licence its author chooses, with no obligation to publish its source. The exception covers the runtime only. Every other library a program links — a vendored C library, a DLL — comes with its own licence, and that licence's conditions apply.

What it costs

- **An executable carries the runtime parts it uses.** `target.tiko` is 24 KB on Windows and 118 KB on Linux, built at `-O0`.
- **A static library is copied into every program that uses it**; a shared one is stored once, and costs a lookup when each program starts.
- **`-exx` makes array-heavy code slower**: every index is compared with both bounds, and every pointer with null.

- **`-O2` and `-O3` make the compiler slower** and the program faster. For a quick edit-and-run loop, the default `-O0` builds fastest.

Summary

- `-target windows-x86_64` (Windows' default), `linux-x86_64` or `macos-arm64`. `TIKO_CLANG`, `TIKO_LIBPATH` and `TIKO_INCLUDE` point the compiler at other tools, runtimes and headers.
- `-c` compiles to an object file; the first file named is main unless `-m` says otherwise; the other modules' top levels run first, as constructors.
- `-lib` builds `libname.a`; `-dll` builds a DLL and `libname.dll.a`, or `libname.so`, or `libname.dylib`, with the procedures marked `export`. Link either with `#inclib` or `-l`.
- A shared library must be found at run time: beside the program on Windows, in the system's directories or `LD_LIBRARY_PATH` on Linux, through `DYLD_LIBRARY_PATH` or an `@loader_path` run path on macOS.
- `-s gui` builds a Windows program with no console; a `.rc` on the command line adds resources.
- `-exx` and `-g` for checked builds, `-O2`/`-O3` for release; `#cmdline` keeps the options in the source.
- `-x name` names the program exactly, over `-o`; the last `-x` wins, and one in `#cmdline` wins over the command line's.
- A project's `tiko.toml` declares its targets; `tiko build` builds them into `out/<profile>`, `tiko run` builds and runs, `tiko check` reports without writing. Paths in it are relative to it. Profiles `debug` (the default) and `release`, and your own, set the options. A second build compiles only what changed.
- A `windows-x86_64` program needs only `ucrtbase.dll` and `KERNEL32.dll`. The runtime's licence lets a compiled program be distributed on any terms.

Exercises

1. Build a static library with `Gcd` and an `Lcm` built on it, and use it to find the smallest number that every number from 1 to 20 divides. {solution="ex31-01″}
2. Write a program that reports, from the predefined symbols, whether it was built with error checking, with the thread-safe runtime, and against which C runtime. {solution="ex31-02″}
3. Write `mathlib.ti`, a header with the two declarations, and change both clients to include it instead of declaring the functions themselves.
4. Build `use-shared.tiko`, then delete `mathlib.dll` (or `libmathlib.so`) and run it. What does each platform say?
5. Add an `ICON` line to `strings.rc` with an `.ico` file of your own, and check that Explorer shows it for the executable.

6. Turn the static-library example into a project: a `[target.mathlib]` with `kind = "lib"`, and a `[target.use-static]` that links it with `libs` and `libpaths`. Which target must `tiko build` build first, and how do you make sure it does?

Classic BASIC and Compatibility

BASIC is older than most of the languages around it, and a great deal of code was written in its earlier forms: QuickBASIC programs from the 1980s, FreeBASIC programs written in its `qb` compatibility mode, and every generation of code since that kept the old habits. This chapter is for reading and bringing over that code. It covers the classic features tiko keeps — `data` and `read`, computed jumps, the classic date and time functions, `common` and `chain` — and, just as important, the ones it refuses, with the diagnostic each produces and what to write instead.

The listings here end in `.bas`, as the code they stand for usually does. tiko compiles a `.bas` file exactly as it compiles a `.tiko` one.

One dialect

FreeBASIC has four **dialects**, chosen with `#lang` at the top of a file (fbc also has a `-lang` option; tiko does not): `fb`, the modern language; `fblite` and `qb`, which keep QuickBASIC's rules; and `deprecated`, an older version of `fb`. tiko implements **`fb` only**. Asking for another is refused:

examples/ch32/dialect.bas

```
'' Asking for another dialect is refused outright.
#lang "qb"
print "hello"
```

```
dialect.bas(2,1): error TKO0317: unsupported language mode 'qb';
tiko-lang implements "fb" only
```

(One line in the compiler's output, broken here to fit the page.)

Everything this book teaches is the `fb` dialect, and everything below that compiles is legal in it. Code written for `qb` or `fblite` has to be brought over, and most of this chapter is how.

Inline data: data, read and restore

examples/ch32/quiz-data.bas

```
'' A quiz whose questions live in DATA statements.
dim as string question, answer
dim as long count

'' The first DATA item is how many questions follow.
read count
for i as long = 1 to count
```

```
    read question, answer
    print "Q"; i; ": "; question; " -- "; answer
next

'' RESTORE to a label starts reading from that label's DATA.
restore bonus
read question, answer
print "Bonus: "; question; " -- "; answer

'' RESTORE with no label goes back to the first DATA in the file.
restore
read count
print "questions in the main set:"; count

data 3
data "Capital of France?", "Paris"
data "Largest planet?", "Jupiter"
data "Square root of 81?", "9"

bonus:
data "Year the first BASIC ran?", "1964"
```

examples/ch32/quiz-data.out

```
Q 1: Capital of France? -- Paris
Q 2: Largest planet? -- Jupiter
Q 3: Square root of 81? -- 9
Bonus: Year the first BASIC ran? -- 1964
questions in the main set: 3
```

`data` statements hold a list of constant values — numbers and strings — anywhere in the module. **`read`** takes the next values from that list, in the order the `data` statements appear in the file, into the variables it names, converting each to the variable's type. Reading past the last item is **not an error**, even with `-exx`: the variable gets 0 or an empty string, as fbc's does.

`restore` moves the reading position back: with no argument to the first `data` in the module, and with a **label** to the first `data` after that label. Labels in tiko are names followed by a colon on a line of their own, as `bonus:` is; the old numbered lines are not legal in the `fb` dialect.

`data` is a compact way to put a table in a program, and it is easy to get wrong: nothing checks that a `read` takes values of the types the `data` holds, or that each record has the number of fields the code expects, and one missing field shifts every later

read by one. In new code an array with an initialiser (Chapter 8) or a file (Chapter 25) holds the same table, with types the compiler checks.

Computed jumps

examples/ch32/menu.bas

```
'' ON n GOTO jumps to the n-th label in its list.
for choice as long = 0 to 4
    print "choice"; choice; ": ";
    on choice goto add, list, quit
    '' Out of range -- 0, or past the end of the list -- falls through.
    print "no such option"
    goto next_choice
add:
    print "adding a record"
    goto next_choice
list:
    print "listing the records"
    goto next_choice
quit:
    print "quitting"
next_choice:
next
```

examples/ch32/menu.out

```
choice 0: no such option
choice 1: adding a record
choice 2: listing the records
choice 3: quitting
choice 4: no such option
```

`goto label` jumps to a label in the same procedure, and **`on n goto a, b, c`** jumps to the nth label in its list — the first for 1, the second for 2. A value of 0, a negative value or one past the end of the list jumps nowhere: the statement does nothing, and execution carries on with the next line, which is how `menu.bas` catches choices 0 and 4.

`on … goto` is how menus and state machines were written before `select case`. In new code, `select case choice` with a `case` per option says the same thing without the labels and without a `goto` in every branch, and the compiler can see where each one ends.

Classic dates and times

examples/ch32/classic-dates.bas

```
'' The classic date functions keep a date as a double: whole days since
'' 30 December 1899, with the time of day as the fraction.
#include once "vbcompat.bi"

dim as double d = dateserial( 2025, 3, 4 ) + timeserial( 14, 30, 0 )
print "serial:"; d
print "parts:"; year( d ); month( d ); day( d ); hour( d ); minute( d )
print "weekday:"; weekday( d ); "  (1 = Sunday)"
print "formatted: "; format( d, "yyyy-mm-dd hh:nn:ss" )

'' Adding months clamps at the end of the month; differences in days.
dim as double jan31 = dateserial( 2025, 1, 31 )
dim as double feb = dateadd( "m", 1, jan31 )
print "Jan 31 + 1 month: "; format( feb, "yyyy-mm-dd" )
dim as double today = dateserial( 2025, 3, 4 )
dim as double xmas = dateserial( 2025, 12, 25 )
print "days to Christmas:"; datediff( "d", today, xmas )

'' DATE and TIME are the clock, as text: "mm-dd-yyyy" and "hh:mm:ss".
print "date is"; len( date ); " characters, time is"; len( time )
```

examples/ch32/classic-dates.out

```
serial: 45720.60416666666
parts: 2025 3 4 14 30
weekday: 3  (1 = Sunday)
formatted: 2025-03-04 14:30:00
Jan 31 + 1 month: 2025-02-28
days to Christmas: 296
date is 10 characters, time is 8
```

The classic date functions keep a date as a **`double`**: the whole part counts days from 30 December 1899, and the fraction is the time of day — 0.5 is noon. `#include "datetime.bi"` declares them, and `#include "vbcompat.bi"` includes that and adds `format`, `dateadd` and `datediff`, the Visual Basic names.

Function	Gives
`dateserial( y, m, d )`, `timeserial( h, m, s )`	a serial date, or a time to add to one
`year`, `month`, `day`, `hour`, `minute`, `second`	the parts of a serial
`weekday( d )`	the day of the week, **1 for Sunday**
`now`	the current date and time, as a serial
`dateadd( unit, n, d )`, `datediff( unit, a, b )`	arithmetic in `"d"`, `"m"`, `"yyyy"` and other units
`format( d, pattern )`	text; `"yyyy-mm-dd hh:nn:ss"`, with `nn` for minutes
`date`, `time`	the clock as text, `"mm-dd-yyyy"` and `"hh:mm:ss"`
`setdate`, `settime`	set the computer's clock

`setdate` and `settime` change the clock of the whole computer, not just the program's view of it, and need administrator rights on most systems; the listing does not call them.

> **PITFALL**
> **The named formats follow the machine's language.** `format( d, "dddd d mmmm" )` and `monthname( 3 )` ask the operating system for the names, so the same program prints *Tuesday* on one computer and *mardi* on another. Use the numeric patterns for anything written to a file.

Chapter 26's `std/chrono` is the modern replacement, and the differences are worth knowing when moving code across: a serial date has no time zone, `dateserial( 2025, 13, 40 )` rolls over into the next year instead of being refused, `weekday` counts from Sunday where chrono's `DayOfWeek` counts from Monday, and `DateTime.FromSerial` and `.Serial` convert between the two.

common and chain

`examples/ch32/common.bas`

```
'' COMMON SHARED: one variable, declared in every module that uses it.
'' Built with its second module: 'tiko common.bas common-hit.bas'.
common shared hits as long

declare sub Hit( )

hits = 10
```

```
Hit( )
Hit( )
print "hits ="; hits
```

examples/ch32/common-hit.bas

```
'' The second module of common.bas. Its COMMON names the same variable.
common shared hits as long

sub Hit( )
    hits += 1
end sub
```

examples/ch32/common.out

```
hits = 12
```

`common shared` declares a variable that every module of a program can share, by writing the same `common` line in each. Built together with `tiko common.bas common-hit.bas`, the two modules use one `hits`. In new code, declare the variable once with `dim shared` in one module and `extern` in a header the others include, so the compiler checks that every module agrees on its type.

examples/ch32/chain/parent.bas

```
'' CHAIN runs another program, waits for it, and returns its exit code.
#ifdef __FB_WIN32__
    const CHILD = "child.exe"
#else
    const CHILD = "./child"
#endif

print "parent: chaining"
dim as long rc = chain( CHILD )
print "parent: back; the child's exit code was"; rc
print "a missing program gives"; chain( "no-such-program" )
```

examples/ch32/chain/parent.out

```
parent: chaining
  child: running
parent: back; the child's exit code was 3
a missing program gives-1
```

`chain` runs another program, waits for it to finish and returns its exit code, or −1 when the program cannot be started. In QuickBASIC, `chain` with `common` passed variables to the next program. Here the two programs are separate processes that

share nothing: anything one tells the other goes through the command line, a file or the exit code. **`run`** starts a program and does not come back. Chapter 25 covers `exec` and `shell`, which do the same with more control.

What tiko refuses

Each of these compiles in fbc's older dialects and not in `fb`. The table gives the diagnostic tiko reports and the replacement.

Classic code	tiko says	Write instead
`gosub label … return`	`TKO0409: 'GOSUB' is not available in the fb dialect`	a `sub`, with what it changed passed `byref`
`resume`, `resume next`	`TKO0409: 'RESUME' is not available …`	handle the error where it is caught, or return a `Result` (Chapter 18)
`dim count%`	`TKO0320: 'count' has no type …`	`dim as int32 count` — every variable is declared with its type
`defint a-z` and the other `def` statements	`TKO0409: 'DEFINT' is not available …`	the same: declare each variable with its type
`#lang "qb"`	`TKO0317: unsupported language mode 'qb'`	port the code to `fb`
`asm … end asm`	`TKO0407: inline assembly is not supported`	a C function (Chapter 29), or compile that module with fbc
`10 print "x"`	`TKO0944: a line number is not available in the fb dialect`	labels: `start:`

examples/ch32/gosub.bas

```
'' GOSUB belongs to the qb and fblite dialects. tiko implements fb only.
dim as long total = 0
gosub add_one
print total
end

add_one:
    total += 1
    return
```

```
gosub.bas(3,7): error TKO0409: 'GOSUB' is not available in the fb dialect
```

`gosub` jumps to a label and `return` comes back to the line after the `gosub`. It is a subroutine without parameters or local variables, so everything it touches is shared with its caller, and it ends wherever a `return` happens to be. Exercise 1 rewrites the listing with a `sub`: the variable the subroutine changed becomes a `byref` parameter.

examples/ch32/suffix.bas

```
'' A type suffix gives a variable its type in the qb dialect only.
dim count%
count% = 3
print count%
```

```
suffix.bas(2,5): warning TKO0206: suffix ignored in 'count%'
suffix.bas(3,1): warning TKO0206: suffix ignored in 'count%'
suffix.bas(4,7): warning TKO0206: suffix ignored in 'count%'
suffix.bas(2,5): error TKO0320: 'count' has no type; tiko-lang implements
#lang "fb" only, where default types and suffixes are not valid
```

(Again one line.) The warnings are each `%` being thrown away; the error is what stops the build, because without its suffix `count` has no type at all.

In the old dialects a **suffix** gave a variable its type — `%` an integer, `&` a long, `!` a single, `#` a double, `$` a string — and `defint`, `deflng`, `defsng`, `defdbl` and `defstr` set a type for every name starting with given letters. In `fb` every variable is declared, with its type, before it is used; a typo then becomes an error instead of a new, empty variable.

examples/ch32/asm.bas

```
'' Inline assembly: fbc compiles it; tiko does not.
function Twice( byval n as long ) as long
    asm
        mov eax, [n]
        add eax, eax
        mov [function], eax
    end asm
end function
print Twice( 21 )
```

```
asm.bas(7,5): error TKO0407: inline assembly is not supported; compile
that module with fbc
```

tiko produces LLVM's intermediate form, not assembly text, so there is nowhere for written assembly to go. A routine that needs particular instructions can be written in C, where clang's intrinsics reach them, and called through Chapter 29's interface — or that one module can be compiled with fbc and linked in.

Several more classic statements are refused in the `fb` dialect — by fbc with its error 146, "only valid in -lang deprecated or fblite or qb", and by tiko with `TKO0409` naming the keyword: **`let`** in front of an assignment, **`call`** in front of a procedure call, **`lprint`** to the printer, the **`option`** statements (`option explicit`, `option base`), `on n gosub`, and the **`'$` metacommands** (`'$include`, `'$dynamic`, `'$static`). Drop `let` and `call` — `k = 5` and `Hello( )` say the same — open the printer as a file to print to it, `#include` a file, and give arrays explicit bounds.

examples/ch32/let.bas

```
'' LET in front of an assignment: only in the older dialects.
dim as long k
let k = 5
print k
```

```
let.bas(3,1): error TKO0409: 'LET' is not available in the fb dialect
```

`let` is still a keyword of the `fb` dialect in one place: `let ( a, b ) = t` assigns a type's fields to several variables at once, and that form compiles.

Bringing old code across

A QuickBASIC or `fblite` program comes across in the same few steps, most of them mechanical:

1. **Declare every variable**, with its type. The suffixes and `defint` lines tell you which type each one had.
2. **Replace numbered lines with labels**, and delete the numbers that nothing jumps to.
3. **Turn each `gosub` subroutine into a `sub` or `function`**, passing what it reads and changes.
4. **Drop `let` and `call`**; replace `'$include` with `#include` and `option base 1` with explicit bounds.
5. **Replace `on error … resume` with structured handling**, where the error is caught by the code that can do something about it.
6. **Compile with `-exx`** (Chapter 31) and run the program's tests, so an index out of bounds stops the program at the line that caused it.

Then, when the program works, the modern forms in this chapter's notes — arrays for `data`, `select case` for `on goto`, `std/chrono` for serial dates — can come in one at a time.

What it costs

- **Each `read` converts one item** to the variable's type.

- **Serial dates are `doubles`**: cheap to store, and precise to about a millisecond for present-day dates.
- **`chain` starts a whole new process**, which costs milliseconds; calling a procedure costs nanoseconds.

Summary

- tiko implements the `fb` dialect only: `#lang "qb"` and the rest are `TKO0317`.
- `data`, `read` and `restore [label]` keep a table in the program; nothing checks its types.
- `goto label` and `on n goto a, b, c`; an index out of range falls through.
- `datetime.bi` and `vbcompat.bi`: serial dates as `doubles`, `weekday` from Sunday, `format` with numeric patterns for anything stored.
- `common shared` shares a variable between modules; `chain` runs a program and returns its exit code, sharing nothing.
- Refused: `gosub`, `resume`, `let`, `call`, `lprint`, `option`, `defint` and the `'$` metacommands (`TKO0409`), suffixes (`TKO0320`), `asm` (`TKO0407`), other dialects (`TKO0317`) and numbered lines (`TKO0944`).

Exercises

1. Rewrite `gosub.bas` without `gosub`, so that it compiles. {solution="ex32-01″}
2. Read a price list from `data` statements until a sentinel item `"END"`, printing each line and the total. {solution="ex32-02″}
3. Rewrite `menu.bas` with `select case`. Which version makes it easier to see that every branch ends?
4. Rewrite `quiz-data.bas` so the questions live in an array of a `type` with two fields, initialised with `= { … }`. What can the compiler now check that it could not before?
5. Write `classic-dates.bas` again with `std/chrono` (Chapter 26), and compare the weekday each prints for 4 March 2025.

Capstone: a CLI Search Tool

This chapter builds a complete program: **`tsearch`**, a small cousin of `grep`. It walks a directory tree, reads every file in it, and prints the lines that contain a pattern. There is nothing in it the earlier chapters have not shown; what is new is putting the pieces together and making the decisions a real tool needs — what goes in a module and what stays in the program, how a failure reaches the user, what the exit code means, and how the parts are tested.

The specification

```
tsearch [-i] [-n] [-l] [-g glob] pattern [path ...]
```

- Print every line, in every file under each **path**, that contains **pattern**, as `path:line text`. With no path, search the current directory.
- **`-i`** ignores case, for letters outside ASCII as well as inside it. **`-n`** adds the line number: `path:number:line text`. **`-l`** prints only the names of files that match. **`-g glob`** searches only files whose names match a glob pattern such as `*.txt`; case matters in the glob on every platform.
- Paths are printed with `/` between components on every platform, and files are searched in the sorted order of the paths as printed, so the output is the same everywhere.
- When the output goes to a terminal, the file name is coloured; when it is redirected to a file or a pipe, it is plain text.
- Problems — a bad option, an empty argument, a path that does not exist, a file or directory that cannot be read — go to **stderr**, never into the results.
- The **exit code** says what happened, as `grep`'s does: **0** if anything matched, **1** if nothing did, **2** if something went wrong. A script can test it.

The layout

```
ch33/
    tiko.toml           the package
    search/search.tiko  the module: everything that can be tested on its
own
    tsearch.tiko         the program: the command line, the output, the
exit code
    searchtests.tiko    the module's tests
    sample/             files to search
        notes/readme.md
```

```
        notes/todo.txt
        poems/fox.txt
```

The work is split along one line: **everything that does not touch the command line or the screen goes in the module.** Parsing the arguments takes an array and returns options; matching takes two strings; finding files takes a root and returns paths. None of them prints anything, so the tests can call each one directly and check what it returns. The program is what is left: reading `command( )`, printing, and choosing the exit code.

`tiko.toml` makes the directory a **package** (Chapter 15), which lets the exercise solutions in `solutions/` reach the module as `"../search"` — a relative import may not leave its package:

examples/ch33/tiko.toml

```
[package]
name = "tsearch"
```

The module

examples/ch33/search/search.tiko

```
#module "search"
#import "std/fs"
#import "std/containers"
```

The module uses `std/fs` for the file system and `std/containers` for its arrays, and declares two **public** types:

examples/ch33/search/search.tiko

```
'' What one run of tsearch was asked to do.
public type Options
    as ustring pattern
    as containers.Array( of ustring ) roots
    as boolean ignoreCase
    as boolean lineNumbers
    as boolean filesOnly
    as ustring nameGlob             '' "" matches every file name
end type

'' One matching line.
public type Hit
    as int32 lineNo
```

```
    as ustring text
end type
```

`Options` is everything the command line said, already interpreted: the program builds one, and every other procedure reads it. `Hit` is one result. Neither has methods: they are plain records passed between the procedures, which keeps each procedure testable with values a test can write out.

Parsing the arguments

examples/ch33/search/search.tiko

```
'' Read the arguments into opts. Returns "" when they make sense, and
'' otherwise what is wrong with them, for the usage message. An empty
'' argument is refused: it is almost always a variable left unset.
public function ParseArgs _
    ( byref args as containers.Array( of ustring ), _
      byref opts as Options ) as ustring
    dim as int32 i = 0
    do while i < args.Count( )
        var a = args[ i ]
        if len( a ) = 0 then return "an argument is empty"
        select case a
        case "-i" : opts.ignoreCase = true
        case "-n" : opts.lineNumbers = true
        case "-l" : opts.filesOnly = true
        case "-g"
            i += 1
            if i >= args.Count( ) then return "-g needs a pattern"
            if len( args[ i ] ) = 0 then return "-g needs a pattern"
            opts.nameGlob = args[ i ]
        case else
            if left( a, 1 ) = "-" then return "unknown option " & a
            if len( opts.pattern ) = 0 then
                opts.pattern = a
            else
                opts.roots.Push( a )
            end if
        end select
        i += 1
    loop
    if len( opts.pattern ) = 0 then return "no pattern given"
    if opts.roots.Count( ) = 0 then opts.roots.Push( "." )
    return ""
end function
```

`ParseArgs` reads an array rather than calling `command( n )` itself, so a test can hand it any argument list. It returns **an error message, or an empty string** — the one thing the caller needs to know — and fills `opts` in place. A flag that takes a value, `-g`, moves the index past it, and says so if the value is missing. Anything else starting with `-` is refused rather than searched for: a misspelt option should be an error, not a silent search for `-x`. The first other argument is the pattern and the rest are roots, with `.` as the default.

An empty argument is refused as well, with `an argument is empty`, and an empty `-g` value as a missing one. An empty argument is almost always a shell variable that was never set — `tsearch "$WORD" notes` — and taken at face value it would be a pattern that says nothing or a path that names nothing. Refusing it says which.

Matching

examples/ch33/search/search.tiko

```
'' Does line contain pattern? lcase lowers every letter that has a
'' single lower-case form, not only A to Z.
public function LineMatches( byref line_ as ustring, _
                             byref pattern as ustring, _
                             byval ignoreCase as boolean ) as boolean
    if ignoreCase then
        return instr( lcase( line_ ), lcase( pattern ) ) > 0
    end if
    return instr( line_, pattern ) > 0
end function
```

A case-insensitive match compares both strings in lower case. For a `ustring`, **`lcase` lowers more than `A` to `Z`**: `ÄRGER` matches `ärger`, and Greek and Cyrillic capitals lower too. It maps one character to one character, though, which is lower-casing and not full case folding, so a few pairs a reader would call equal stay apart: `STRASSE` does not match `straße`, a Greek capital sigma lowers to `σ` and never to the final `ς`, and the Turkish capital `İ` is left as it is. The tests pin down both sides. Lower-casing a whole line for every comparison is not the fastest possible way — the pattern could be lowered once — but it is easy to check, which is the right first version.

Finding the files

examples/ch33/search/search.tiko

```
'' Does a file name match the -g glob? The POSIX grammar, on every
'' platform, so the answer is the same on Windows and Linux.
function NameMatches( byref nm as ustring, _
                      byref nameGlob as ustring ) as boolean
    if len( nameGlob ) = 0 then return true
```

```
    return fs.MatchPattern( nm, nameGlob, fs.FB_PATH_POSIX )
end function

'' Appends to files every file under root whose name matches nameGlob,
'' as display paths in sorted order, so the output depends neither on
'' the order the file system returns them nor on the platform. Returns
'' how many it appended. What could not be read goes into problems,
'' one message each, naming the path that failed.
public function FindFiles _
    ( byref root as ustring, byref nameGlob as ustring, _
      byref files as containers.Array( of ustring ), _
      byref problems as containers.Array( of ustring ) ) as int32
    dim found as containers.Array( of ustring )
    if fs.IsFile( root ) then
        found.Push( DisplayPath( root ) )
    else
        for each r in fs.WalkDir( root )
            if r.IsOk( ) = false then
                var e = r.Failure( )
                problems.Push( DisplayPath( e.Path( ) ) & ": " & _
                               e.Message( ) )
            elseif r.Value( ).IsFile( ) then
                if NameMatches( r.Value( ).Name( ), nameGlob ) then
                    found.Push( DisplayPath( r.Value( ).Path( ) ) )
                end if
            end if
        next
    end if
    containers.Sort( found )
    for each f in found
        files.Push( f )
    next
    return found.Count( )
end function
```

A root that is a file is searched as it is. A directory is walked with **`fs.WalkDir`** (Chapter 25), which yields one `Result` per entry: a successful entry that is a file is kept when its name matches the glob, using **`fs.MatchPattern`** — the same matcher `fs.Glob` uses, applied to one name.

`NameMatches` passes `fs.MatchPattern` a grammar, **`fs.FB_PATH_POSIX`**, rather than leaving it the default. The default is the platform's own grammar, which ignores case on Windows and respects it on Linux, so `-g "*.TXT"` would find `todo.txt` on one

and not on the other. With the POSIX grammar named, case matters on both, and the same glob gives the same answer.

A failed entry — a directory that cannot be opened, or a root that does not exist — becomes a message in `problems`, so the program can report it and still search everything else. The message names **`FsError.Path( )`**, the path that failed: for a missing root that is the root, `nowhere: no such file or directory`, and for an unreadable directory deep in the tree it is that directory, `sample/notes: access denied`, not the root the user typed.

The paths are **sorted** before they are returned, and sorted **as they will be printed**. A file system lists a directory in whatever order it likes, and that order differs between Windows and Linux; a tool whose output changes between runs on the same files is a tool whose output cannot be compared. Sorting the native paths would not be enough, because `\` sorts after the capital letters and `/` before the digits: `d\f.txt` comes after `dA\f.txt`, but `d/f.txt` before it. So each path goes through `DisplayPath` first, and the sort sees the same strings on both platforms. Windows accepts `/` as a separator, so the display path is also the one the program opens.

`FindFiles` appends what it finds to `files`, in that order, and returns how many it appended, whether the root is a file or a directory.

Searching a file

`examples/ch33/search/search.tiko`

```
'' The matching lines of one file. A file that cannot be read is an
'' FsError in the Result, for the caller to report.
public function SearchFile( byref path as ustring, _
                            byref opts as Options, _
                           byref hits as containers.Array( of Hit ) ) _
                            as fs.FsResultLong
    dim lines as containers.Array( of ustring )
    var got = fs.ReadAllLines( path, lines )
    if got.IsOk( ) = false then return got
    for i as int32 = 0 to lines.Count( ) - 1
        if LineMatches( lines[ i ], opts.pattern, opts.ignoreCase ) then
            dim h as Hit
            h.lineNo = i + 1
            h.text = lines[ i ]
            hits.Push( h )
        end if
    next
    return got
end function
```

`fs.ReadAllLines` returns a **`Result`** (Chapter 18): the number of lines, or an **`FsError`** that says why the file could not be read. `SearchFile` passes that `Result` straight back, so the caller learns about a failure in the same value that carries success, and cannot forget to look: there is no error number to check separately and no runtime error to trap.

examples/ch33/search/search.tiko

```
'' A path as tsearch prints it: '/' between components everywhere.
'' Only Windows separates them with '\'; on Linux a '\' is part of
'' a file name, and stays as it is.
function DisplayPath( byref path as ustring ) as ustring
    dim as ustring p = path
    #ifdef __FB_WIN32__
    for i as int32 = 0 to len( p ) - 1
        if p[ i ] = asc( "\" ) then p[ i ] = asc( "/" )
    next
    #endif
    return p
end function
```

`DisplayPath` is the last piece of the "same output everywhere" promise: Windows writes `\` between path components, and the tool prints `/` on both. It converts only on Windows. On Linux `\` is an ordinary character a file name may contain — `a\b.txt` is one file — and printing it as `a/b.txt` would name a different one. It is not `public`: `FindFiles` applies it, and nothing outside the module needs to.

The program

examples/ch33/tsearch.tiko

```
'' tsearch -- print the lines of files that contain a pattern.
''
''     tsearch [-i] [-n] [-l] [-g glob] pattern [path ...]
''
'' Exit code: 0 if anything matched, 1 if nothing did, 2 on an error.
#import "./search"
#import "std/containers"
#include once "fbio.bi"
#ifdef __FB_WIN32__
#include once "windows.bi"
#endif

const EXIT_FOUND = 0, EXIT_NONE = 1, EXIT_ERROR = 2

sub Complain( byref message as ustring )
```

```
    dim as int32 f = freefile( )
    open err for output as #f
    print #f, "tsearch: "; message
    close #f
end sub

'' Colour only when a person is watching; never into a pipe or a file.
'' A Windows console draws escape sequences only once it is asked to,
'' and one that cannot be asked gets plain text.
function ColourWanted( ) as boolean
    if IsRedirected( ) then return false
    #ifdef __FB_WIN32__
    '' ENABLE_VIRTUAL_TERMINAL_PROCESSING, which windows.bi lacks
    const VT_PROCESSING = &h0004
    var h = GetStdHandle( STD_OUTPUT_HANDLE )
    dim as ulong mode
    if GetConsoleMode( h, @mode ) = 0 then return false
    return SetConsoleMode( h, mode or VT_PROCESSING ) <> 0
    #else
    return true
    #endif
end function

'' The arguments, without the program's own name. An empty one is
'' still an argument, and ParseArgs decides what it means.
dim args as containers.Array( of ustring )
for n as int32 = 1 to __FB_ARGC__ - 1
    args.Push( command( n ) )
next

dim opts as search.Options
var problem = search.ParseArgs( args, opts )
if len( problem ) > 0 then
    Complain( problem )
    Complain( "usage: tsearch [-i] [-n] [-l] [-g glob] " & _
              "pattern [path ...]" )
    end EXIT_ERROR
end if

dim as boolean colour = ColourWanted( )

dim as int32 matched = 0, failures = 0
for each root in opts.roots
    dim files as containers.Array( of ustring )
```

```
    dim problems as containers.Array( of ustring )
    search.FindFiles( root, opts.nameGlob, files, problems )
    for each p in problems
        Complain( p )
        failures += 1
    next
    for each path in files
        dim hits as containers.Array( of search.Hit )
        var got = search.SearchFile( path, opts, hits )
        if got.IsOk( ) = false then
            Complain( path & ": " & got.Failure( ).Message( ) )
            failures += 1
            continue for
        end if
        if hits.Count( ) = 0 then continue for
        matched += hits.Count( )
        '' The file name, in magenta when a person is watching.
        dim as ustring shown = path
        if colour then shown = !"\27[35m" & path & !"\27[0m"
        if opts.filesOnly then
            print shown
            continue for
        end if
        for each h in hits
            dim as ustring where_ = shown & ":"
            if opts.lineNumbers then where_ &= h.lineNo & ":"
            print where_; h.text
        next
    next
next

if failures > 0 then end EXIT_ERROR
end iif( matched > 0, EXIT_FOUND, EXIT_NONE )
```

The program reads like the specification. It collects the arguments — `command( 1 )` to `command( __FB_ARGC__ - 1 )` (Chapter 25) — and hands them to `ParseArgs`; a bad command line prints the problem and the usage line to **stderr** through `Complain`, and ends with exit code 2. The loop counts rather than stopping at the first empty `command( n )`: that would read `tsearch fox "" sample` as `tsearch fox` and quietly search `.`.

`IsRedirected( )` (from `fbio.bi`, Chapter 25) is true when standard output is not a terminal. The program colours only when it is false, so `tsearch fox > hits.txt` writes plain text and a person at a terminal sees the file names in magenta, in `-l`

output as well. The colour itself is an ANSI escape sequence, `ESC [35m` to start and `ESC [0m` to stop.

A Linux terminal draws those sequences as they arrive. A Windows console draws them only once it has been asked to, and the runtime does not ask on the program's behalf, so `ColourWanted` does: it switches on the console's virtual-terminal mode through **`windows.bi`**, the Windows API header, and gives up colour if the console refuses. `windows.bi` does not name the `ENABLE_VIRTUAL_TERMINAL_PROCESSING` flag, so the listing defines it. Both `#include` lines sit below the `#import` lines, because an import must come before any code and a header brings some: above them, an import fails with `TKO0362: '#import' must come before any code, in the compiled file`.

Every failure is counted, and reported where it happens, without stopping the search: one unreadable file should not hide the matches in all the others. Only at the end does the program choose the exit code: **2** if anything failed, otherwise **0** or **1** by whether anything matched.

Running it

Run from the chapter's directory as `tsearch -n -i fox sample`:

examples/ch33/tsearch.out

```
sample/notes/readme.md:2:The fox example lives in poems/.
sample/notes/todo.txt:2:Read the chapter on the quick brown FOX
sample/poems/fox.txt:1:The quick brown fox
sample/poems/fox.txt:3:A Fox is not a dog,
sample/poems/fox.txt:4:and a dog is not a fox.
```

A few more runs, on either platform, with the exit code after each:

```
$ tsearch -l -g "*.txt" fox sample
sample/poems/fox.txt
exit 0
$ tsearch zebra sample
exit 1
$ tsearch fox nowhere
tsearch: nowhere: no such file or directory
exit 2
$ tsearch -x fox
tsearch: unknown option -x
tsearch: usage: tsearch [-i] [-n] [-l] [-g glob] pattern [path ...]
exit 2
```

The second run prints nothing at all and still says something, through its exit code: a script can write `if tsearch -l secret . ; then …` and never look at the output.

The glob is quoted because a Unix shell expands an unquoted `*.txt` against the current directory before tsearch sees it. On Windows, quoting is not enough.

PITFALL
On Windows the runtime expands * and ? in every argument before the program starts — quoted or not, from cmd, PowerShell or any other shell — against the files in the current directory, or in the directory the argument names. Run from `sample/notes`, which holds `todo.txt`, `tsearch -l -g "*.txt" fox ..` receives `-g todo.txt`, searches only files named `todo.txt`, prints nothing and exits 1. Nothing reports the change. With two `.txt` files there, the second would become the pattern and the real pattern a path.

An argument that matches no file is passed on unchanged, and the expansion knows only * and ?. So write one character of the glob as a class: `-g "*.tx[t]"` matches no file name literally, reaches tsearch as typed, and still means `*.txt` to `fs.MatchPattern`. It needs quoting in a Unix shell too, where it is otherwise a glob for the shell. The runtime asks for the expansion in its startup code, and the pinned version has no switch to turn it off.

Testing

examples/ch33/searchtests.tiko

```
'' tsearch's tests: the search module, without the command line.
#import "./search"
#import "std/containers"
#include once "fbcunit.bi"

function Args3( byref a as ustring, byref b as ustring, _
                byref c as ustring ) as containers.Array( of ustring )
    dim r as containers.Array( of ustring )
    if len( a ) then r.Push( a )
    if len( b ) then r.Push( b )
    if len( c ) then r.Push( c )
    return r
end function

SUITE( arguments )
    TEST( pattern_and_default_root )
        dim o as search.Options
        var problem = search.ParseArgs( Args3( "-i", "fox", "" ), o )
        CU_ASSERT_EQUAL( problem, "" )
        CU_ASSERT( o.ignoreCase )
        CU_ASSERT_EQUAL( o.pattern, "fox" )
```

```
        CU_ASSERT_EQUAL( o.roots[ 0 ], "." )
    END_TEST

    TEST( refuses_what_it_cannot_use )
        dim o1 as search.Options, o2 as search.Options
        dim o3 as search.Options
        CU_ASSERT_EQUAL( search.ParseArgs( Args3( "-x", "", "" ), _
                                           o1 ), "unknown option -x" )
        CU_ASSERT_EQUAL( search.ParseArgs( Args3( "-n", "", "" ), _
                                           o2 ), "no pattern given" )
        CU_ASSERT_EQUAL( search.ParseArgs( Args3( "fox", "-g", "" ), _
                                           o3 ), "-g needs a pattern" )
    END_TEST

    TEST( refuses_an_empty_argument )
        dim o as search.Options
        dim a as containers.Array( of ustring )
        a.Push( "fox" )
        a.Push( "" )
        CU_ASSERT_EQUAL( search.ParseArgs( a, o ), _
                         "an argument is empty" )
    END_TEST
END_SUITE

SUITE( matching )
    TEST( case_matters_unless_asked )
        CU_ASSERT( search.LineMatches( "a quick fox", "fox", false ) )
        CU_ASSERT( search.LineMatches( "A FOX", "fox", false ) = false )
        CU_ASSERT( search.LineMatches( "A FOX", "fox", true ) )
    END_TEST

    TEST( folding_is_not_only_ascii )
        CU_ASSERT( search.LineMatches( !"STRA\u00DFE UND \u00C4RGER", _
                                       !"\u00E4rger", true ) )
    END_TEST

    TEST( folding_is_one_letter_at_a_time )
        '' Lower case is not case folding: SS does not become sharp s.
        CU_ASSERT( search.LineMatches( "STRASSE", !"stra\u00DFe", _
                                       true ) = false )
    END_TEST
END_SUITE

SUITE( files )
```

```
    TEST( finds_and_filters_in_order )
        dim all_ as containers.Array( of ustring )
        dim txt as containers.Array( of ustring )
        dim problems as containers.Array( of ustring )
        CU_ASSERT_EQUAL( search.FindFiles( "sample", "", all_, _
                                           problems ), 3 )
        CU_ASSERT_EQUAL( search.FindFiles( "sample", "*.txt", txt, _
                                           problems ), 2 )
        CU_ASSERT_EQUAL( txt[ 0 ], "sample/notes/todo.txt" )
        CU_ASSERT_EQUAL( problems.Count( ), 0 )
    END_TEST

    TEST( globs_match_case_on_every_platform )
        dim upper as containers.Array( of ustring )
        dim problems as containers.Array( of ustring )
        CU_ASSERT_EQUAL( search.FindFiles( "sample", "*.TXT", upper, _
                                           problems ), 0 )
    END_TEST

    TEST( reports_line_numbers )
        dim o as search.Options
        o.pattern = "dog"
        dim hits as containers.Array( of search.Hit )
        var got = search.SearchFile( "sample/poems/fox.txt", o, hits )
        CU_ASSERT( got.IsOk( ) )
        CU_ASSERT_EQUAL( hits.Count( ), 3 )
        CU_ASSERT_EQUAL( hits[ 2 ].lineNo, 4 )
    END_TEST

    TEST( a_missing_file_is_an_error_value )
        dim o as search.Options
        o.pattern = "x"
        dim hits as containers.Array( of search.Hit )
        var got = search.SearchFile( "sample/absent.txt", o, hits )
        CU_ASSERT( got.IsOk( ) = false )
    END_TEST
END_SUITE

end iif( fbcu.run_tests( ), 0, 1 )
```

On Windows:

```
examples/ch33/searchtests.windows-x86_64.out
```

```
SUMMARY

 Asserts   Passed   Failed  Suite                                  Tests
--------   --------   --------   ------------------------------------
--------
      9        9        0  files                                     4
      5        5        0  matching                                  3
      8        8        0  arguments                                 3
--------   --------   --------   ------------------------------------
--------
     22       22        0  Total                                    10
```

On Linux the same suites run in another order (Chapter 19):

examples/ch33/searchtests.out

```
SUMMARY

 Asserts   Passed   Failed  Suite                                  Tests
--------   --------   --------   ------------------------------------
--------
      8        8        0  arguments                                 3
      5        5        0  matching                                  3
      9        9        0  files                                     4
--------   --------   --------   ------------------------------------
--------
     22       22        0  Total                                    10
```

The tests are a program of their own, built with fbcunit (Chapter 19), that imports the module just as `tsearch.tiko` does. Each test calls one procedure with values it chose and checks what came back: the argument parser with good and bad command lines, an empty argument among them; matching with and without case, lower-casing beyond ASCII, and where lower-casing stops; finding files in `sample/`, with and without a glob, in sorted order, and a glob whose case does not match on either platform; line numbers; and a missing file, which must come back as a failed `Result` rather than a crash. Nothing here needs a terminal or anything outside `sample/`, so it can run on every build.

The program itself is tested the way the runs above were: by running it on known files and comparing its output and exit code with what they should be.

What it costs

- **Every file is read whole** with `ReadAllLines`, so a search holds one file in memory at a time. A tool for very large files would read a line at a time with `line input #`.
- **Case-insensitive matching lower-cases every line**, one copy per line. Lowering the pattern once and comparing without copying is the obvious optimisation, and the tests are what make it safe to try.
- **Sorting the file list** costs a little time for a large tree, and buys output that does not depend on the file system.

Summary

- Put everything that does not touch the command line or the screen in a module, and test it directly; keep the program a thin layer over it.
- Parse arguments from an array into an options record, counting to `__FB_ARGC__`, and report problems as messages; refuse unknown options and empty arguments.
- `fs.WalkDir` and `fs.MatchPattern` find the files; name the POSIX grammar for the same glob answer on both platforms, and sort the paths as printed for the same order. `fs.ReadAllLines` returns a `Result` whose `FsError` says why a file could not be read, and `Path( )` which file.
- `lcase` lowers letters beyond ASCII one character at a time; it is not full case folding.
- On Windows the runtime expands `*` and `?` in arguments even inside quotes; write one character of a glob as a class, `*.tx[t]`, to get it through.
- Diagnostics go to stderr; `IsRedirected( )` decides about colour, and a Windows console needs its virtual-terminal mode switched on first; the exit code — 0, 1 or 2 — says what happened.
- fbcunit tests the module's procedures one by one, on both platforms.

Exercises

1. Write a program that uses the `search` module to print how many lines match in each file, as `path: count`. {solution="ex33-01"}
2. Add whole-word matching, so that `dog` matches `the lazy dog.` and `hot-dog` but not `dogma` or `underdog`. Count a letter outside ASCII as part of a word, so that `Schädog` does not match either. {solution="ex33-02"}
3. Add a `-c` option that prints only the number of matching lines in each file, with tests for its parsing.
4. Add `-v`, which prints the lines that do *not* match. Which other options does it interact with, and how should `-l -v` behave?
5. Make the match cheaper: lower-case the pattern once, outside the loop, and check that every test still passes.

Capstone: a Concurrent Pipeline

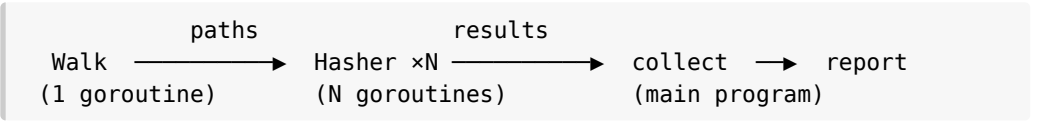
The second capstone is **`thash`**, which finds duplicate files. It walks a directory tree, reads every file, hashes the contents, and reports the files whose contents are the same. Chapter 33's tool did one thing after another; this one does several things at once. Walking the tree, reading files and hashing them become **stages** that run as goroutines, joined by channels (Chapter 23). The work is what the earlier chapters showed: `fs.WalkDir` and `fs.ReadAllBytes` (Chapter 25), a `Map` (Chapter 24), a `Stopwatch` (Chapter 26), `RunningStats` (Chapter 27) and a `Logger` (Chapter 28). What is new is fitting them into a pipeline. That means deciding who closes which channel, how full a channel may get, how the whole thing stops early, and how output that is produced in no fixed order can still be the same on every run.

The specification

```
thash [-j workers] [-t ms] [-v] directory
```

- Hash every file under **directory** and print the groups of files with identical contents. Each group gives its size and hash, then its paths in sorted order. The groups come in order of their first path, and a total says how many bytes the extra copies take.
- Print a line of statistics first: how many files, how many bytes, and the smallest, largest and mean sizes. When no file was read, the line stops at the two counts.
- **`-j`** sets how many files are hashed at once (default 4). **`-t ms`** stops after that many milliseconds and reports what was done so far. **`-v`** logs every file as it is hashed, and the time the run took.
- Paths are printed with `/` between components. The output does not depend on how many workers there are or on the order in which they finish.
- A file that cannot be read is reported on **stderr** and does not stop the run.
- **Exit code:** 0 if every file was hashed, 1 if some could not be read or the run was stopped early, and 2 for a bad command line.

The shape



There are three stages, and each one owns the end of a channel. **Walk** is the only sender on `paths`, so it is the one that closes it. **N hashers** share `paths` as receivers and `results` as senders. No single hasher can close `results`, because the others may

still be sending, so a small goroutine waits on a **`WaitGroup`** and closes it once the last hasher is done. The **main program** receives from `results` with `for each`, which ends when the channel closes. Every channel is closed exactly once, by whoever knows that nothing more will be sent. That one rule is what lets each loop end by itself.

The program is one file, `thash.tiko`, shown here a section at a time. `sample/` holds nine small files. Seven of them form three groups of copies: three of one image, two of a readme and two of some notes. The other two are unique:

```
ch34/
    thash.tiko
    sample/archive/2025/IMG_0042.img     copy of photos/lighthouse.img
    sample/archive/2025/readme.txt       copy of docs/readme.txt
    sample/archive/notes-old.txt         copy of docs/notes.txt
    sample/docs/notes.txt
    sample/docs/plans.txt
    sample/docs/readme.txt
    sample/photos/harbour.img
    sample/photos/lighthouse-copy.img    copy of photos/lighthouse.img
    sample/photos/lighthouse.img
```

The records

examples/ch34/thash.tiko

```
#import "std/tasks"
#cmdline "-mt"
#import "std/fs"
#import "std/containers"
#import "std/numeric"
#import "std/chrono"
#import "std/logs"
```

Every concurrent listing starts like this. `#import "std/tasks"` comes first, then `#cmdline "-mt"` so that the program links the thread-safe runtime. The import also brings `select channel`.

examples/ch34/thash.tiko

```
'' What one run of thash was asked to do.
type Options
    as ustring root
    as int32 workers = 4
    as int32 timeoutMs = 0          '' 0: no time limit
    as boolean verbose
end type
```

```
'' What a hasher found out about one file.
type FileHash
    as ustring path
    as int64 size
    as ulongint hash
    as ustring problem              '' "" when the file was read
end type
```

`Options` is the command line, already interpreted. `FileHash` is the message that travels from the hashers to the collector: a whole record, with a `ustring` in it, sent through a channel by value. A file that could not be read still produces a record, with `problem` filled in. The collector must hear about every file, or it could not count the failures.

Stage 1: walking the tree

`examples/ch34/thash.tiko`

```
'' Stage 1: every file under root, into paths. Stops early when the
'' context is done; closes paths either way, so the hashers finish.
'' A select case needs the channel itself, not a SendChan view of it.
sub Walk( byval root as ustring, byval ctx as tasks.Context, _
          byval paths as tasks.Channel( of ustring ) )
    for each r in fs.WalkDir( root )
        if r.IsOk( ) andalso r.Value( ).IsFile( ) then
            dim as ustring p = r.Value( ).Path( )
            select channel
                case send p to paths            : '' on to the next
                case receive from ctx.Done( )   : exit for
            end select
        end if
    next
    paths.Close( )
end sub
```

`Walk` sends each file's path into `paths`. It does not use a plain `paths.Send( p )`, because a send can **wait**: if the channel is full, the walker sits there until a hasher takes something out. The `select channel` waits on two things at once. Either the send goes through, or the context's `Done( )` channel closes, which means that someone has said stop. Whichever happens first wins. Stopping leaves the loop, and the `Close` after it still runs, so the hashers see the end of `paths` either way.

`paths` is declared as a whole `tasks.Channel`, not a `SendChan`. **A `select case` needs the channel itself.** The helpers behind `select channel` are written for `Channel( of T )`, so a `SendChan` or `RecvChan` view of it does not fit. Declare `paths` as a `SendChan`

here and the compiler stops at the `select channel` line with `TKO0809: cannot infer the type arguments of 'selectcommitsend'`. The narrower types are still the right choice wherever a stage only calls `Send` or `Receive`, as the hashers do.

Stage 2: hashing, N at a time

examples/ch34/thash.tiko

```
'' Stage 2, run by several goroutines at once: read each file whole and
'' hash its bytes. A file that cannot be read is reported, not dropped.
sub Hasher( byval ctx as tasks.Context, _
            byval paths as tasks.RecvChan( of ustring ), _
            byval results as tasks.SendChan( of FileHash ), _
            byval lg as logs.Logger ptr, _
            byval wg as tasks.WaitGroup ptr )
    dim as ustring p
    do while paths.Receive( p )
        if len( ctx.Err( ) ) > 0 then exit do
        dim fh as FileHash
        fh.path = p
        dim bytes as containers.Array( of ubyte )
        var got = fs.ReadAllBytes( p, bytes )
        if got.IsOk( ) = false then
            fh.problem = got.Failure( ).Message( )
            lg->Warn( ).AddText( "path", DisplayPath( p ) ) _
                       .Msg( fh.problem )
        else
            fh.size = got.Value( )
            if fh.size = 0 then
                fh.hash = containers.HashBytes( nullptr, 0 )
            else
                fh.hash = containers.HashBytes( @bytes[ 0 ], fh.size )
            end if
            lg->Debug( ).AddText( "path", DisplayPath( p ) ) _
                        .AddInt( "bytes", fh.size ).Msg( "hashed" )
        end if
        results.Send( fh )
    loop
    wg->Done( )
end sub

'' Closes results once every hasher has finished: the collector's
'' 'for each' ends there.
sub CloseWhenDone( byval wg as tasks.WaitGroup ptr, _
                   byval results as tasks.Channel( of FileHash ) )
```

```
    wg->Wait( )
    results.Close( )
end sub
```

Several `Hasher` goroutines run the same loop over the same `paths` channel, and each path goes to exactly one of them: whichever is free first. That is the whole of the **fan-out**. Each hasher reads a file whole with `fs.ReadAllBytes` and hashes the bytes with `containers.HashBytes`, a 64-bit FNV-1a hash, the one `HashOf` uses when a `Map` has string keys. The result goes into `results`, which all of the hashers send to. That is the **fan-in**.

When `paths` is closed and empty, `Receive` returns false, the loop ends, and the hasher calls `wg->Done( )`. The main program calls `wg.Add( )` for each hasher just before spawning it, and spawns `CloseWhenDone` only after the last one, so the count is complete before anyone waits on it. `CloseWhenDone` waits for the count to reach zero and only then closes `results`. A hasher also checks the context before each file, so a stopped run does not read every file left in the channel's buffer.

Logging from goroutines

The hashers log through one `Logger`, passed to each of them as a pointer. **A logger can be shared between threads.** Each record is built by the thread that calls `Warn( )` or `Debug( )`. A sink takes one lock for the whole of a write, so two goroutines logging at the same moment produce two whole lines, never one line with the other's text inside it (Chapter 28). With `-v`, the logger's `MinLevel` is `LOG_DEBUG` and every file produces a line. Without it, the level is `LOG_WARN`, and only failures are written. A `Debug( )` record is then switched off by one comparison, and nothing is formatted or locked. Its arguments are still evaluated, though: `DisplayPath( p )` still builds a string, and `AddText`, `AddInt` and `Msg` are still called, each returning at once. A call cannot avoid working out what it is passed.

Back-pressure

A channel's capacity is how far its sender may get ahead of its receiver. When the buffer is full, `Send` waits. That is **back-pressure**: a fast stage cannot run away from a slow one and fill memory with work that nobody has started yet.

examples/ch34/backpressure.tiko

```
#import "std/tasks"
#cmdline "-mt"
#include once "fbatomic.bi"

'' Sends as fast as it can, counting each value the channel accepted.
sub Produce( byval out_ as tasks.SendChan( of int32 ), _
```

```
            byval sent as long ptr )
    for i as int32 = 1 to 6
        out_.Send( i )
        FB.AtomicFetchAddLong( sent, 1 )
    next
    out_.Close( )
end sub

dim as long sent
var queue = tasks.Channel( of int32 )( 2 )    '' room for two values
spawn Produce( queue, @sent )

'' Nobody receives yet: the producer fills the buffer, then waits.
sleep 200, 1
print "sent before anyone received:"; FB.AtomicLoadLong( @sent )

dim as int32 total = 0
for each v in queue
    total += v
next
print "received them all, total"; total; _
      ", sent"; FB.AtomicLoadLong( @sent )
```

examples/ch34/backpressure.out

```
sent before anyone received: 2
received them all, total 21, sent 6
```

The producer wants to send six values, and the channel has room for two. For the first 200 milliseconds nobody receives, so the producer stops after two sends and waits inside the third. It finishes only as the main program drains the channel.

In `thash`, `paths` holds `workers * 2` paths, so the walker can stay a little ahead of the hashers and never leave them idle. It cannot list a whole disk into memory while they read the first file. `results` holds `workers` records. If the collector were slow, the hashers would wait for it rather than pile up finished work.

Stopping early

A **`Context`** carries the decision to stop. `tasks.WithCancel` makes one that stops when its `Cancel( )` is called. `tasks.WithTimeout` makes one that also stops by itself after a number of milliseconds. A stage that could be left waiting for ever once the others stop puts a `case receive from ctx.Done( )` beside its real work.

examples/ch34/stopearly.tiko

```
#import "std/tasks"
#cmdline "-mt"

'' Would send a million values; stops as soon as its context is done.
sub Produce( byval ctx as tasks.Context, _
             byval out_ as tasks.Channel( of int32 ), _
             byval done_ as tasks.SendChan( of ustring ) )
    dim as int32 sent = 0
    for i as int32 = 1 to 1000000
        select channel
            case send i to out_            : sent += 1
            case receive from ctx.Done( )  : exit for
        end select
    next
    out_.Close( )
    done_.Send( "producer sent " & sent & ", then " & ctx.Err( ) )
end sub

var ctx = tasks.WithCancel( tasks.Context( ) )
var values = tasks.Channel( of int32 )( 0 )   '' no buffer: a handover
var finished = tasks.Channel( of ustring )( 1 )
spawn Produce( ctx, values, finished )

dim as int32 v
for k as int32 = 1 to 3
    values.Receive( v )
    print "got"; v
next
ctx.Cancel( )                                  '' enough

dim as ustring why
finished.Receive( why )
print why
```

examples/ch34/stopearly.out

```
got 1
got 2
got 3
producer sent 3, then cancelled
```

The producer would send a million values. The channel has no buffer, so each send is a handover that completes only when the main program receives. After three, the main program cancels. The producer is waiting in its `select` for a fourth handover that will never come, `Done( )` closes, and it stops, having sent exactly three. `ctx.Err( )`

says why a context stopped: `cancelled` here, and `deadline exceeded` when a timeout ran out.

In `thash`, only the walker needs that `select`. When the run stops, the hashers stop taking paths, so a walker waiting on a full `paths` would wait for ever. The other stages wait without one: a hasher in a plain `paths.Receive` and `results.Send`, and `CloseWhenDone` in `wg->Wait( )`. Each of those waits is sure to end, because of who closes what. The walker closes `paths` whether or not it was stopped, which ends every `Receive`. The collector keeps receiving until `results` closes, so no `Send` waits long. Every hasher calls `Done( )` once its loop ends, which releases `Wait( )`. A hasher checks the context only before each file, so that a stopped run reads no more of them.

Stage 3: collecting

examples/ch34/thash.tiko

```
dim opts as Options
var problem = ParseArgs( __FB_ARGC__, opts )
if len( problem ) > 0 then
    dim as int32 e = freefile( )
    open err for output as #e
    print #e, "thash: "; problem
    print #e, "usage: thash [-j workers] [-t ms] [-v] directory"
    close #e
    end 2
end if

'' Diagnostics go to stderr; -v adds a line for every file hashed.
dim tr as logs.TextRenderer
tr.stamp = logs.TS_NONE
dim con as logs.ConsoleSink = logs.ConsoleSink( logs.CT_STDERR, @tr )
dim lg as logs.Logger
lg.MinLevel = iif( opts.verbose, logs.LOG_DEBUG, logs.LOG_WARN )
lg.AddSink( @con )

dim ctx as tasks.CancelContext
if opts.timeoutMs > 0 then
    ctx = tasks.WithTimeout( tasks.Context( ), opts.timeoutMs )
else
    ctx = tasks.WithCancel( tasks.Context( ) )
end if

'' Buffered channels: a full one makes its sender wait.
var paths = tasks.Channel( of ustring )( opts.workers * 2 )
var results = tasks.Channel( of FileHash )( opts.workers )
```

```
dim wg as tasks.WaitGroup

var clock = chrono.Stopwatch.StartNew( )
spawn Walk( opts.root, ctx, paths )
for w as int32 = 1 to opts.workers
    wg.Add( )
    spawn Hasher( ctx, paths, results, @lg, @wg )
next
spawn CloseWhenDone( @wg, results )

'' Stage 3, here in the main program: collect.
dim byHash as containers.Map _
    ( of ulongint, containers.Array( of ustring ) )
dim bytesOf as containers.Map( of ulongint, int64 )
dim sizes as numeric.RunningStats
dim as int64 total = 0
dim as int32 unreadable = 0
for each fh in results
    if len( fh.problem ) > 0 then
        unreadable += 1
    else
        byHash[ fh.hash ].Push( DisplayPath( fh.path ) )
        bytesOf[ fh.hash ] = fh.size
        sizes.Add( fh.size )
        total += fh.size
    end if
next
var stopped = ctx.Err( )
ctx.Cancel( )

'' With no sizes there is no smallest, largest or mean to print.
print sizes.Count( ); " files,"; total; " bytes";
if sizes.Count( ) > 0 then
    print "; smallest"; sizes.MinValue( ); ", largest"; _
          sizes.MaxValue( ); ", mean"; numeric.Round( sizes.Mean( ), 1 )
else
    print
end if
Report( byHash, bytesOf )
if unreadable > 0 then
    print unreadable; " files could not be read"
end if
if len( stopped ) > 0 then print "stopped early ("; stopped; ")"
if opts.verbose then
```

```
    lg.Info( ).AddInt( "ms", clock.ElapsedMilliseconds ) _
              .Msg( "done" )
end if
end iif( unreadable > 0 orelse len( stopped ) > 0, 1, 0 )
```

The main program is the last stage. It parses the command line and sets up the logger with `CT_STDERR`, so diagnostics never mix with the report. It makes a context, with a deadline if `-t` asked for one. It creates the channels and the `WaitGroup`, starts the `Stopwatch`, and spawns everything. Then it collects:

- **`byHash`**, a `Map` from hash to the paths with that hash, already passed through `DisplayPath`. `byHash[ h ]` creates an empty array the first time a hash is seen, so `.Push( )` works for the first path and every later one.
- **`bytesOf`**, the size that goes with each hash.
- **`sizes`**, a `numeric.RunningStats` (Chapter 27), which keeps the count, minimum, maximum and mean in constant space, one value at a time.

After the loop, `ctx.Err( )` says whether the deadline cut the run short. It is read **before** `ctx.Cancel( )`, which every context gets when it is finished with, and which would otherwise change the answer to `cancelled`.

The statistics line prints the smallest, largest and mean sizes only when there are some. With no sizes at all, from an empty directory or one where nothing could be read, `RunningStats` has none to give: it answers NaN, which would print as a different string on each target. The line stops at `0 files, 0 bytes` instead, and the exit code is still 0 for an empty directory and 1 when files could not be read.

Turning arrival order into stable output

The records arrive in whatever order the hashers finish, and that order changes from run to run. Nothing in the report depends on it:

examples/ch34/thash.tiko

```
'' Groups of identical files, each sorted, the groups in order of their
'' first path; then how much space the extra copies take.
sub Report( byref byHash as containers.Map _
                ( of ulongint, containers.Array( of ustring ) ), _
            byref bytesOf as containers.Map( of ulongint, int64 ) )
    dim heads as containers.Array( of ustring )
    dim headHash as containers.Map( of ustring, ulongint )
    for each kv in byHash
        if kv.value.Count( ) > 1 then
            var group = kv.value
            containers.Sort( group )
            heads.Push( group[ 0 ] )
```

```
            headHash[ group[ 0 ] ] = kv.key
        end if
    next
    containers.Sort( heads )

    dim as int64 wasted = 0
    for each h in heads
        var group = byHash[ headHash[ h ] ]
        containers.Sort( group )
        dim as int64 size = bytesOf[ headHash[ h ] ]
        print group.Count( ); " copies of"; size; " bytes, ";
        print "hash "; lcase( hex( headHash[ h ], 16 ) )
        for each p in group
            print "    "; p
        next
        wasted += size * ( group.Count( ) - 1 )
    next
    print heads.Count( ); " groups;"; wasted; " bytes in extra copies"
end sub
```

Each group's paths are sorted. Each group is named by its first path, and the groups are printed in sorted order of that name. The totals are sums, which come out the same in any order. The `Stopwatch` time differs on every run, so it is logged only with `-v`, on stderr, and never appears in the report.

The paths being sorted are the ones that are printed. The collector stores each one as `DisplayPath` gives it, so the order cannot depend on the target. Sorting the raw paths would let it: `/` sorts before the digits and `\` after them, so on Windows `s2\a0.txt` would come before `s2\a\z.txt`, and on Linux `s2/a/z.txt` before `s2/a0.txt`. `DisplayPath` is the same as Chapter 32's. It turns `\` into `/` only on Windows, because on Linux `\` is an ordinary character in a file name:

`examples/ch34/thash.tiko`

```
'' A path as thash prints it: '/' between components everywhere. Only
'' Windows writes '\' there; on Linux it is an ordinary file-name
'' character and is left alone.
function DisplayPath( byref path as ustring ) as ustring
    dim as ustring p = path
    #ifdef __FB_WIN32__
    for i as int32 = 0 to len( p ) - 1
        if p[ i ] = asc( "\" ) then p[ i ] = asc( "/" )
    next
    #endif
```

```
    return p
end function
```

`ParseArgs` reads every argument. The main program passes it `__FB_ARGC__`, which exists only at the main module's top level (Chapter 25). It refuses an unknown option, a second directory, an empty argument, and a `-j` or `-t` without a whole number above 0 after it. `numeric.TryParseLong` (Chapter 27) takes the whole text or nothing, where `valint( "2x" )` would quietly give 2:

examples/ch34/thash.tiko

```
'' Read the argc - 1 arguments into opts; "" if they make sense, else
'' why not.
function ParseArgs( byval argc as int32, byref opts as Options ) _
        as ustring
    dim as int32 i = 1
    do while i < argc
        dim as ustring a = command( i )
        select case a
        case "-j", "-t"
            i += 1
            var n = numeric.TryParseLong( command( i ) )
            if n.HasValue( ) = false orelse n.Value( ) < 1 then
                return a & " needs a whole number above 0"
            end if
            if a = "-j" then
                opts.workers = n.Value( )
            else
                opts.timeoutMs = n.Value( )
            end if
        case "-v" : opts.verbose = true
        case "" : return "an empty argument"
        case else
            if left( a, 1 ) = "-" then return "unknown option " & a
            if len( opts.root ) > 0 then return "only one directory"
            opts.root = a
        end select
        i += 1
    loop
    if len( opts.root ) = 0 then return "no directory given"
    if fs.IsDir( opts.root ) = false then
        return opts.root & " is not a directory"
    end if
    return ""
end function
```

Running it

Run from the chapter's directory as `thash -j 4 sample`:

`examples/ch34/thash.out`

```
9 files, 395 bytes; smallest 22, largest 78, mean 43.9
 3 copies of 27 bytes, hash d957e1846cbc67d7
    sample/archive/2025/IMG_0042.img
    sample/photos/lighthouse-copy.img
    sample/photos/lighthouse.img
 2 copies of 78 bytes, hash a6a3f3d0cd5a4640
    sample/archive/2025/readme.txt
    sample/docs/readme.txt
 2 copies of 56 bytes, hash a2991430a071803d
    sample/archive/notes-old.txt
    sample/docs/notes.txt
 3 groups; 188 bytes in extra copies
```

The output is the same with `-j 1` and with `-j 8`, on Windows and on Linux: the harness runs it on both, and it was also run 30 times with 1 to 8 workers. With `-v` on Windows, the log lines go to stderr in the order the files were hashed, which changes between runs:

```
$ thash -v -j 2 sample
DEBUG  hashed  path=sample/archive/2025/readme.txt bytes=78
DEBUG  hashed  path=sample/archive/2025/IMG_0042.img bytes=27
…
INFO   done  ms=1
```

On Linux, with one file made unreadable (`chmod 000 sample/docs/plans.txt`), the warning goes to stderr, the report covers the other eight files, and the exit code is 1:

```
$ thash sample
WARN   access denied  path=sample/docs/plans.txt
8 files, 373 bytes; smallest 24, largest 78, mean 46.6
…
 3 groups; 188 bytes in extra copies
 1 files could not be read
exit 1
```

On a larger tree of several thousand files, `-t 20` stops part of the way through:

```
$ thash -t 20 tests
589 files, 766691 bytes; smallest 13, largest 51770, mean 1301.7
```

```
  0 groups; 0 bytes in extra copies
stopped early (deadline exceeded)
```

How far it gets depends on the machine and on how busy it is. That is the nature of a time limit, and the reason the report says plainly that it is incomplete.

What it costs

- **Every file is read whole.** `fs.ReadAllBytes` reads the file into one buffer and then copies it into the `Array`, so for a moment a hasher holds the file twice, and N hashers up to 2N files. An `Array`'s count is a 32-bit `long`, so a file of 2 GiB or more cannot be read this way at all. For large files, read and hash in fixed-size blocks instead.
- **Equal hashes are evidence, not proof.** Two different files with the same 64-bit FNV-1a hash are very unlikely, but possible, and FNV is not designed to resist someone making a collision on purpose. A tool that deletes copies must compare the bytes first (Exercise 1).
- **Every file is hashed**, even when no other file has the same size. Grouping by size first and hashing only the sizes that occur more than once would skip most files on a real disk.
- **More workers is not always faster.** Hashing a small file costs little; reading it is the slow part, and a disk serves only so many reads at once. Measure with `-v` before choosing `-j`.

Summary

- A pipeline is stages joined by channels. Each channel is closed once, by the stage that knows nothing more will be sent; when several stages send, a `WaitGroup` and a closer goroutine decide when.
- Several goroutines receiving from one channel is a fan-out; several sending to one channel is a fan-in.
- A channel's capacity limits how far a sender can get ahead: back-pressure.
- A `Context` stops a pipeline. A stage that could wait for ever once the others stop puts `case receive from ctx.Done( )` beside that wait; a wait that the close discipline is sure to end does not need one. Read `ctx.Err( )` to learn why it stopped. A `select` case needs a whole `Channel`, not a `SendChan` or `RecvChan` (`TKO0809`).
- A `Logger` can be shared between goroutines; each record is written whole.
- Output that arrives in no fixed order becomes stable by sorting it in the form it is printed in, and by keeping times and other run-dependent values out of it.

Exercises

1. Write `SameBytes( a, b )`, which compares two files byte by byte, and check it on the groups that `thash` reports for `sample/`. {solution="ex34-01″}
2. Start two walkers, one for `sample/docs` and one for `sample/photos`, both sending to a single channel, and print every file name they find in sorted order. Who closes the channel? {solution="ex34-02″}
3. Group the files by size first, and hash only the files whose size is shared by another file. Print how many files that skipped.
4. Add `-min bytes`, which ignores files smaller than the given size, and decide whether the walker or the hashers should apply it.
5. Add a fourth stage between the hashers and the collector that confirms each group with `SameBytes` before it is reported.

Capstone: a TUI Ledger

The last capstone is **`tledger`**, a personal ledger: the money that came in and went out, kept in a database, loaded from a bank's CSV export, shown and edited on terminal screens, and summed into a monthly report. It joins the parts of the book that deal with real data:

- `Decimal` for money that must be exact (Chapter 27);
- `LocalDate` for days (Chapter 26);
- SQLite through `vendor/sqlite3` (Chapter 28), and CSV through `strings.SplitCsv` (Chapter 9);
- an `OrderedMap` for the report (Chapter 24);
- the TUI library for the screens (Chapter 28);
- modules and a package to hold it together (Chapter 15).

The specification

```
tledger [--import file.csv]... [--list | --report] ledger.db
```

- **The ledger** is one SQLite file. Each transaction has a date, a payee, a category and an amount: positive for money in, negative for money out. `:memory:` gives a ledger that disappears when the program ends.
- **`--import file.csv`** loads a bank export with a `date,payee,category,amount` header, and may be given more than once. **All or nothing:** if any record is wrong, none of the file is kept, and every wrong record is reported with its number in the file. The file can then be fixed and loaded again without making duplicates. Blank lines are skipped, and a file with no records imports nothing, without complaint.
- **`--list`** prints every transaction, oldest first. **`--report`** prints income, spending and net for each month, and the totals. Asking for both is a mistake.
- With neither, **the screens** open: a table of transactions, a form for a new one, and the monthly report. F1, F2 and F3 switch between them, and Esc leaves.
- Amounts print as `$1,234.56`, the same on every machine, whatever its locale.
- **Exit code:** 0; 1 if an import was refused; 2 for a bad command line, a ledger that cannot be opened, or no terminal to open the screens on.

The layout

```
ch35/
    tiko.toml           the package
    ledger/ledger.tiko  the data: model, storage, import, report -- and
all of the SQL
```

```
    ui/ui.tiko          the screens: a TuiApp
    tledger.tiko        the program: the command line, the text output,
the exit code
    screens.tiko        the screens driven by a script, with no terminal
    schema.tiko         migrations, run twice
    bank.csv, bad.csv   a bank export, and one with mistakes in it
```

The split follows the rule from Chapter 33, with one more reason behind it. **`ledger` is the only file that imports `vendor/sqlite3`, and nothing from SQLite is public**: the rest of the program sees transactions and totals, never a `sqlite3 ptr` or a statement. Swapping SQLite for something else would change one file.

The model

examples/ch35/ledger/ledger.tiko

```
#module "ledger"
#import "std/chrono"
#import "std/money"
#import "std/containers"
#import "std/strings"
'' This module is the only place that imports SQLite, and nothing
'' from it is public. Linked statically, it is inside the program.
#import "vendor/sqlite3" link static
```

examples/ch35/ledger/ledger.tiko

```
'' One line of the ledger. Money in is positive, money out negative.
public type Txn
    as longint id
    as chrono.LocalDate day
    as ustring payee
    as ustring category
    as money.Decimal amount
end type

'' One month of the report.
public type MonthTotals
    as money.Decimal income
    as money.Decimal spending       '' zero or negative
end type

'' Months by "YYYY-MM". A generic is named through an alias.
public type MonthMap as containers.OrderedMap( of ustring, MonthTotals )
```

A `Txn` holds its money as a **`Decimal`**, never as a `float64`. `0.1 + 0.2` is not `0.3` in binary floating point (Chapter 27), and a ledger that is a cent out after a year of additions is wrong. The date is a **`LocalDate`**: a calendar day, with no time and no zone, which is all a bank statement says. `MonthTotals` is one row of the report. `MonthMap` names the generic the report fills, so the screens and the program can write it without repeating its type arguments.

One check for every way in

`examples/ch35/ledger/ledger.tiko`

```
'' Check four fields of text and build a Txn from them: "" if they make
'' a transaction, else what is wrong. The import and the entry form
'' both come through here, so they refuse exactly the same things.
public function ParseTxn _
    ( byref day as ustring, byref payee as ustring, _
      byref category as ustring, byref amount as ustring, _
      byref t as Txn ) as ustring
    '' TryParseIso also reads the compact 20260105. A ledger keeps one
    '' spelling, so the dashes are required.
    dim as boolean dashed = len( day ) = 10 andalso _
                            mid( day, 5, 1 ) = "-" andalso _
                            mid( day, 8, 1 ) = "-"
    if dashed = false orelse _
       chrono.LocalDate.TryParseIso( day, t.day ) = false then
        return "'" & day & "' is not a date (YYYY-MM-DD)"
    end if
    t.payee = trim( payee )
    if len( t.payee ) = 0 then return "the payee is empty"
    t.category = lcase( trim( category ) )
    if len( t.category ) = 0 then t.category = "other"
    var d = money.DecimalParse( trim( amount ) )
    if d.HasValue( ) = false then
        return "'" & amount & "' is not an amount"
    end if
    if d.Value( ).Scale( ) > 2 then
        return "'" & amount & "' has more than two decimal places"
    end if
    t.amount = d.Value( )
    return ""
end function
```

Text becomes a transaction in **one place**. The import calls `ParseTxn` for each line of the CSV, and the entry form calls it for what was typed. The two can never disagree about what a valid transaction is. It refuses a date that does not exist (`LocalDate.TryParseIso` knows that April has thirty days) or is not written `YYYY-MM-`

`DD`, an empty payee, and an amount that is not a number or has more than two decimal places. `TryParseIso` also reads the compact ISO form, `20260105`, so the dashes are checked first: a ledger keeps one spelling of a date, and the message promises that one. A Decimal keeps the scale it was written with, so `Scale( )` says how many places the text had. Categories are folded to lower case, so `Food` and `food` are one category.

Storage

examples/ch35/ledger/ledger.tiko

```
public sub CloseLedger( )
    sqlite3_close( db )             '' closing a null connection is fine
    db = nullptr
end sub

'' Open (or create) a ledger file, or ":memory:" for one that is gone
'' when the program ends. Returns "" or why it could not.
public function OpenLedger( byref path as ustring ) as ustring
    if sqlite3_open( path, @db ) <> SQLITE_OK then
        var why = LastError( )
        CloseLedger( )
        return why
    end if
    return Migrate( )
end function
```

The module keeps **one database for the whole program**, a connection in a module variable, and four small procedures that every other part of the module goes through:

examples/ch35/ledger/ledger.tiko

```
'' The program's one database. OpenLedger and CloseLedger bracket it.
private dim shared as sqlite3 ptr db

'' What SQLite says went wrong with the last call on the connection.
function LastError( ) as ustring
    dim as string msg = *sqlite3_errmsg( db )
    return msg
end function

'' Run SQL that returns no rows: "" or SQLite's reason.
function RunSql( byref sql as ustring ) as ustring
    if sqlite3_exec( db, sql, nullptr, nullptr, nullptr ) <> SQLITE_OK _
            then return LastError( )
```

```
    return ""
end function

'' Compile SQL into st: "" or SQLite's reason.
function Prepare( byref sql as ustring, _
                  byref st as sqlite3_stmt ptr ) as ustring
    if sqlite3_prepare_v2( db, sql, -1, @st, nullptr ) <> SQLITE_OK _
            then return LastError( )
    return ""
end function

'' A column of the current row as text. SQLite's text is UTF-8, and a
'' string holding UTF-8 converts to a ustring correctly.
function ColText( byval st as sqlite3_stmt ptr, _
                  byval col as int32 ) as ustring
    dim as string bytes
    var p = sqlite3_column_text( st, col )
    if p <> nullptr then bytes = *cast( zstring ptr, p )
    return bytes
end function
```

The variable is `private` for a reason that goes beyond tidiness. A module's interface describes its module-level variables to the files that import it, so that a generic's body can use them. A variable that is not `private` would put `sqlite3 ptr` into that description, and SQLite would reach every importer. That is exactly what the module exists to prevent. `private` keeps the variable, and its type, inside the module.

The four procedures turn SQLite's C conventions into the module's own. SQLite reports a failure as a result code and keeps the reason on the connection; every procedure here returns `""` or that reason, as text, so the callers read like the rest of the program. **`RunSql`** runs SQL that returns no rows, **`Prepare`** compiles a statement, **`LastError`** fetches the reason, and **`ColText`** reads a column back as a `ustring`. (`Exec` and `Field` would have been shorter names, and both are taken: tiko predeclares `field` and `exec`, and declaring either as these procedures are is an error, `TKO0600` and `TKO0793`.)

Text crosses in UTF-8 both ways. A `ustring` passed where SQLite's C wants `const char *` is converted to UTF-8 for the call, so `RunSql` hands its `ustring` straight to `sqlite3_exec`. Coming back, `sqlite3_column_text` points at UTF-8 bytes SQLite owns; `ColText` copies them into a `string`, and a `string` assigned to a `ustring` is read as UTF-8.

The schema, and migrations

`examples/ch35/ledger/ledger.tiko`

```
'' The schema, one step per version. A step that has shipped is never
'' edited -- some database has already run it -- so a change to the
'' schema is always a new step at the end.
const LATEST_SCHEMA = 2

function SchemaStep( byval v as int32 ) as ustring
    select case v
    case 1
        return "create table txn( " & _
               "id integer primary key, " & _
               "day text not null, " & _
               "payee text not null, " & _
               "category text not null, " & _
               "amount text not null )"
    case 2
        return "create index txn_day on txn( day )"
    end select
    return ""
end function

'' The version the file is at, in v: "" or why it cannot be read --
'' "file is not a database", for a file that is something else.
function ReadVersion( byref v as int32 ) as ustring
    dim st as sqlite3_stmt ptr
    var why = Prepare( "pragma user_version", st )
    if len( why ) > 0 then return why
    defer sqlite3_finalize( st )
    if sqlite3_step( st ) <> SQLITE_ROW then return LastError( )
    v = sqlite3_column_int( st, 0 )
    return ""
end function

public function SchemaVersion( ) as int32
    dim as int32 v = -1
    ReadVersion( v )
    return v
end function

'' Bring the schema up to date, one transaction per step, so a failure
'' leaves the database at the last version that completed.
function Migrate( ) as ustring
    dim as int32 current
    var why = ReadVersion( current )
    if len( why ) > 0 then return why
```

```
    if current > LATEST_SCHEMA then
        return "the database is from a newer tledger (schema " & _
               str( current ) & ")"
    end if
    for v as int32 = current + 1 to LATEST_SCHEMA
        why = RunSql( "begin" )
        if len( why ) = 0 then why = RunSql( SchemaStep( v ) )
        if len( why ) = 0 then
            why = RunSql( "pragma user_version = " & v )
        end if
        if len( why ) = 0 then why = RunSql( "commit" )
        if len( why ) > 0 then
            RunSql( "rollback" )        '' undo whatever the step did
            return "schema step " & str( v ) & ": " & why
        end if
    next
    return ""
end function
```

A program that keeps data outlives the first version of its schema. **Migrations** are how it changes without losing what is already stored. Each version is one step, applied in order, and the database remembers how far it has got.

SQLite keeps a number for exactly that purpose: **`pragma user_version`**, an integer in the database file's header that SQLite itself never uses. A new file has 0. `Migrate` reads it, runs every step above it, and sets it after each one, inside the same transaction. A step that fails rolls back with its version number, so the file is always at a version that completed. A file whose version is above the program's is refused: it was written by a newer `tledger`, and this one cannot know what the newer steps did.

Reading the version is also the first time SQLite looks inside the file. `sqlite3_open` succeeds on any file, because SQLite reads nothing until it must, so a ledger path that names a text file fails here, and `Migrate` returns SQLite's own reason: `tledger notes.txt` prints `tledger: notes.txt: file is not a database` and exits with 2. `SchemaVersion`, the public question, says −1 for the same file.

Every statement in a step is checked, `begin` and `commit` included: they are SQL like the step itself, run by `RunSql`. On any failure the step runs `rollback`, which undoes whatever it had done. `commit` can fail too — a file locked by another program refuses it — and then the transaction is still open, which is why the `rollback` follows a failed `commit` as well.

The rule that makes this work is in the comment: **a step that has shipped is never edited.** Version 2 here added an index. Adding a column next year is a version

3 that says `alter table txn add column …`. Editing step 1 instead would change nothing in every database that has already run it.

examples/ch35/schema.tiko

```
'' Migrations: a new file is brought to the latest schema; opening it
'' again finds nothing to do.
#import "./ledger"

kill "schema-demo.db"                  '' start from nothing
print "open: ["; ledger.OpenLedger( "schema-demo.db" ); "]"
print "schema version"; ledger.SchemaVersion( )
ledger.CloseLedger( )

print "reopen: ["; ledger.OpenLedger( "schema-demo.db" ); "]"
print "schema version"; ledger.SchemaVersion( )
ledger.CloseLedger( )
kill "schema-demo.db"
```

examples/ch35/schema.out

```
open: []
schema version 2
reopen: []
schema version 2
```

Rows in and out

examples/ch35/ledger/ledger.tiko

```
'' Dates are stored as ISO text, which sorts in date order; amounts as
'' the Decimal's own text, which is exact. SQL never does arithmetic on
'' them: sum( ) would turn them into binary floating point.
function Insert( byref t as Txn ) as ustring
    dim st as sqlite3_stmt ptr
    var why = Prepare( "insert into txn( day, payee, category, " & _
                       "amount ) values( ?, ?, ?, ? )", st )
    if len( why ) > 0 then return why
    defer sqlite3_finalize( st )
    sqlite3_bind_text( st, 1, t.day.ToIsoString( ), -1, _
                       SQLITE_TRANSIENT )
    sqlite3_bind_text( st, 2, t.payee, -1, SQLITE_TRANSIENT )
    sqlite3_bind_text( st, 3, t.category, -1, SQLITE_TRANSIENT )
    sqlite3_bind_text( st, 4, t.amount.ToString( ), -1, _
                       SQLITE_TRANSIENT )
    if sqlite3_step( st ) <> SQLITE_DONE then return LastError( )
    t.id = sqlite3_last_insert_rowid( db )
```

```
    return ""
end function

public function AddTxn( byref t as Txn ) as ustring
    return Insert( t )
end function

'' Every transaction, oldest first.
public function ListTxns( byref txns as containers.Array( of Txn ) ) _
                          as ustring
    dim st as sqlite3_stmt ptr
    var why = Prepare( "select id, day, payee, category, amount " & _
                       "from txn order by day, id", st )
    if len( why ) > 0 then return why
    defer sqlite3_finalize( st )
    var rc = sqlite3_step( st )
    do while rc = SQLITE_ROW
        dim t as Txn
        t.id = sqlite3_column_int64( st, 0 )
        t.day = chrono.LocalDate.ParseIso( ColText( st, 1 ) )
        t.payee = ColText( st, 2 )
        t.category = ColText( st, 3 )
        var amount = money.DecimalParse( ColText( st, 4 ) )
        t.amount = amount.Value( )
        txns.Push( t )
        rc = sqlite3_step( st )
    loop
    if rc <> SQLITE_DONE then return LastError( )
    return ""
end function
```

Two storage decisions are deliberate:

- **Dates are ISO text**, `2026-01-05`. They sort as text in date order, so `order by day` needs nothing else. A month is the first seven characters.
- **Amounts are the Decimal's own text**, `-1450.00`, which is exact. SQLite could store a number instead, but its `sum( )` would add in binary floating point, and the exactness the model bought would be lost at the last step. So SQL stores and sorts, and arithmetic happens in the program, in Decimal.

Every value travels as a `?` parameter (Chapter 28), so a payee called `O'Brien` is just text. Nothing is pasted into SQL. Each statement is finalized by a `defer` written straight after it is prepared (Chapter 22), so every `return` below it, the early ones included, frees it. `ListTxns` keeps `sqlite3_step`'s last result: a loop that stops at

anything but `SQLITE_ROW` would otherwise treat a failure halfway through as the end of the rows.

Importing a bank's CSV

examples/ch35/ledger/ledger.tiko

```
'' Field i of a CSV record, or "" when the record is shorter.
function FieldAt( byref rec as strings.CsvRow, byval i as int32 ) _
                as ustring
    if i < 0 orelse i >= rec.fields.Count( ) then return ""
    return rec.fields[ i ]
end function

'' Check every record and insert the good ones, in one transaction
'' that is committed only if every record was good.
function AddRecords _
    ( byref path as ustring, _
      byref rows as containers.Array( of strings.CsvRow ), _
      byref problems as containers.Array( of ustring ) ) as int32
    '' Row 0 is the header: it says which column holds what.
    dim as int32 cDate = -1, cPayee = -1, cCategory = -1, cAmount = -1
    for c as int32 = 0 to rows[ 0 ].fields.Count( ) - 1
        select case lcase( trim( FieldAt( rows[ 0 ], c ) ) )
        case "date": cDate = c
        case "payee": cPayee = c
        case "category": cCategory = c
        case "amount": cAmount = c
        end select
    next
    if cDate < 0 orelse cPayee < 0 orelse cAmount < 0 then
        problems.Push( path & ": the header needs date, payee " & _
                       "and amount" )
        return 0
    end if

    var why = RunSql( "begin" )
    if len( why ) > 0 then
        problems.Push( path & ": " & why )
        return 0
    end if
    dim as int32 added = 0
    for r as int32 = 1 to rows.Count( ) - 1
        var day = FieldAt( rows[ r ], cDate )
        var payee = FieldAt( rows[ r ], cPayee )
        var category = FieldAt( rows[ r ], cCategory )
```

```
        var amount = FieldAt( rows[ r ], cAmount )
        '' A blank line -- most often the last one -- is a record with
        '' every field empty. It is not a mistake, and adds nothing.
        if len( day & payee & category & amount ) = 0 then continue for
        dim t as Txn
        why = ParseTxn( day, payee, category, amount, t )
        if len( why ) = 0 then why = Insert( t )
        if len( why ) > 0 then
            '' Row r is record r + 1 of the file: the header was 1.
            problems.Push( path & ":" & str( r + 1 ) & ": " & why )
        else
            added += 1
        end if
    next
    if problems.Count( ) = 0 then
        why = RunSql( "commit" )
        if len( why ) > 0 then problems.Push( path & ": " & why )
    end if
    if problems.Count( ) > 0 then
        RunSql( "rollback" )            '' keep none of the file
        return 0
    end if
    return added
end function

'' Load a bank's CSV export (date,payee,category,amount). All or
'' nothing: one bad record and no record is kept, so the file can be
'' fixed and loaded again without making duplicates. Returns the number
'' of rows added; problems get one message per bad record.
public function ImportCsv _
    ( byref path as ustring, _
      byref problems as containers.Array( of ustring ) ) as int32
    '' The file's bytes, as they are: UTF-8, which SplitCsv reads.
    dim as string text
    dim f as int32 = freefile
    if open( path for binary access read as #f ) <> 0 then
        problems.Push( path & ": cannot be read" )
        return 0
    end if
    text = space( lof( f ) )
    get #f, , text
    close #f

    var rows = strings.SplitCsv( text )
```

```
    '' A file with no records -- a header alone, or nothing at all --
    '' adds nothing, and is not a mistake.
    if rows.Count( ) < 2 then return 0
    return AddRecords( path, rows, problems )
end function
```

`ImportCsv` reads the file's bytes whole, and **`strings.SplitCsv`** (Chapter 9) does the parsing: quoted fields such as `"Smith, Jones & Co"`, doubled quotes, either kind of line ending. The header row says which column is which, so a bank that puts `amount` first still loads. `AddRecords` then sends every record through `ParseTxn` and inserts the good ones, all inside one **transaction**. If any record was refused, it runs `rollback`, and the ledger is exactly as it was.

Three cases are easy to miss:

- **A blank line**, most often the last one, is a record whose fields are all empty. It is not a mistake, so it is skipped rather than refused as `'' is not a date`.
- **A file with no records** — a header alone, or an empty file — gives `SplitCsv` fewer than two rows, and `ImportCsv` returns before starting a transaction, so the program reports `imported 0 transactions` either way.
- **The number in a message** is the record's: row 0 is the header, record 1 of the file, so row `r` is record `r + 1`. That is the line number for every file a bank exports. A quoted field may hold a line break, though, and then every record after it sits further down the file than its number says.

`examples/ch35/bank.csv`

```
date,payee,category,amount
2026-01-02,Harbour Property Management,rent,-1450.00
2026-01-05,Northwind Ltd,salary,3200.00
2026-01-09,Corner Grocer,food,-86.42
2026-01-17,"Smith, Jones & Co",utilities,-112.30
2026-01-24,Corner Grocer,food,-64.15
2026-02-02,Harbour Property Management,rent,-1450.00
2026-02-05,Northwind Ltd,salary,3200.00
2026-02-11,City Transit,transport,-98.00
2026-02-14,Corner Grocer,food,-71.80
2026-02-20,"Smith, Jones & Co",utilities,-104.95
2026-03-02,Harbour Property Management,rent,-1450.00
2026-03-05,Northwind Ltd,salary,3200.00
2026-03-06,Tax refund,income,412.60
2026-03-12,Corner Grocer,food,-93.27
```

The report

examples/ch35/ledger/ledger.tiko

```
'' Income and spending per month, keyed "YYYY-MM". An OrderedMap keeps
'' its keys sorted, so the months come out in calendar order.
public sub SumByMonth( byref txns as containers.Array( of Txn ), _
                       byref months as MonthMap )
    for each t in txns
        var key = left( t.day.ToIsoString( ), 7 )
        if t.amount.IsNegative( ) then
            months[ key ].spending = months[ key ].spending + t.amount
        else
            months[ key ].income = months[ key ].income + t.amount
        end if
    next
end sub

'' The report over the whole ledger: "" or why it could not be read.
public function MonthlyReport( byref months as MonthMap ) as ustring
    dim txns as containers.Array( of Txn )
    var why = ListTxns( txns )
    if len( why ) = 0 then SumByMonth( txns, months )
    return why
end function

'' Money as the reports print it. Every field is set, so the text is
'' the same whatever the locale of the machine running the program.
public function FormatMoney( byref d as money.Decimal ) as ustring
    dim as money.MoneyFormat mf = d
    mf.Symbol = "$"
    mf.DecimalPoint = "."
    mf.ThousandsSep = ","
    mf.Grouping = "3;0"             '' threes, repeated: 1,234,567
    mf.FracDigits = 2
    mf.SymbolPrecedes = true
    mf.SpaceBySymbol = false
    mf.Negative = money.MONEY_NEG_MINUS
    return mf.ToString( )
end function
```

`MonthlyReport` reads every transaction, and `SumByMonth` adds each to its month, as income or as spending. The two are separate so that the screens, which already hold the list, can sum it without reading the ledger again. **`months[ key ]`** creates a zero `MonthTotals` the first time a month is seen, as a `Map` does. Because the map is an **`OrderedMap`**, iterating it visits the keys in sorted order. `YYYY-MM` sorts as the calendar does, so the months come out in order without a separate sort.

`MoneyFormat` turns a Decimal into money text. By default it follows the user's locale: symbol, separators, grouping, and where the minus sign goes. That is right for a program showing a person their own money, and wrong for a book whose output must be the same on every machine. So `FormatMoney` sets every field itself. To follow the locale instead, leave the fields unset.

> **PITFALL**
> `Grouping` is a list of group sizes, read from the right, and it does not repeat by itself. `"3"` makes one group of three and stops, so `-1234567.89` prints as `-$1234,567.89`: every amount under a million looks right, and the first large one does not. A trailing 0 means "repeat the last size", so `"3;0"` gives `-$1,234,567.89`, and the Indian `"3;2;0"` gives `-$12,34,567.89`.

The screens

examples/ch35/ui/ui.tiko

```
#module "ui"
#import "std/tui"
#import "std/containers"
#import "std/money"
#import "../ledger"
```

examples/ch35/ui/ui.tiko

```
public enum Page
    PAGE_LIST
    PAGE_ENTRY
    PAGE_REPORT
end enum

'' A generic is named through an alias before it can be a field.
public type TxnArray as containers.Array( of ledger.Txn )

'' Everything the screens show is drawn from this state, every frame.
public type LedgerApp extends tui.TuiApp
    as Page page
    as TxnArray txns
    as ledger.MonthMap months                    '' the report
    as tui.TuiTableState listState
    '' The form's four fields -- date, payee, category, amount -- and
    '' which of them has the keys.
    as tui.TuiTextInputState inputs( 0 to 3 )
    as int32 focus
```

```
    as ustring message                          '' the status line
    declare function Init( ) as tui.TuiCmd override
    declare function Update( byref e as tui.TuiEvent ) _
                             as tui.TuiCmd override
    declare sub View( byref f as tui.TuiFrame ) override
    declare sub Reload( )
    declare sub StartEntry( )
    declare sub Edit( byref e as tui.TuiEvent )
    declare sub Save( )
end type
```

`LedgerApp` is a `TuiApp` (Chapter 28): `Init`, `Update` and `View`. Its fields are **everything the screens show**: which page is up, the transactions, the report's months, the table's selection, the four fields of the form, and the status line. `Update` changes them and draws nothing. `View` draws from them and reads nothing else, not even the database: `Reload` reads the ledger once, and sums the report from the list it read, so a frame costs no query. That separation is what lets the screens be tested without a terminal, below.

`View` does write one thing back. `TuiTable.Render` keeps the selection inside the rows that exist and moves `listState.offset`, the first row shown, so that the selected row stays in view. That has to happen while drawing, because only then is the table's height known. It is state the table owns, and the app only stores it.

Update

examples/ch35/ui/ui.tiko

```
function LedgerApp.Init( ) as tui.TuiCmd
    page = PAGE_LIST
    Reload( )
    return tui.TuiNone( )
end function

'' Read the ledger into the state: the list, and the report summed
'' from it. View draws from these and never reads the database.
sub LedgerApp.Reload( )
    txns.Clear( )
    months.Clear( )
    var why = ledger.ListTxns( txns )
    if len( why ) > 0 then message = why
    ledger.SumByMonth( txns, months )
end sub

sub LedgerApp.StartEntry( )
```

```
    for i as int32 = 0 to 3
        tui.TuiTextInputStateInit( inputs( i ) )
    next
    focus = 0
    message = "Tab: next field   Enter: save   Esc: cancel"
    page = PAGE_ENTRY
end sub

'' A key or a paste for the field that has the focus.
sub LedgerApp.Edit( byref e as tui.TuiEvent )
    '' A widget is a value: make one over the state, hand it the
    '' event, and let it edit the text.
    dim as tui.TuiTextInput box
    box.state = @inputs( focus )
    box.isFocused = true                    '' or it ignores the event
    box.HandleEvent( e )
    '' This box was never drawn, so it does not know how wide the field
    '' is and scrolls as if one column were. The next frame's box knows:
    '' let it do the scrolling.
    inputs( focus ).scrollX = 0
end sub

'' The form's text goes through the same check as an imported line.
sub LedgerApp.Save( )
    dim t as ledger.Txn
    var why = ledger.ParseTxn( inputs( 0 ).text, inputs( 1 ).text, _
                               inputs( 2 ).text, inputs( 3 ).text, t )
    if len( why ) = 0 then why = ledger.AddTxn( t )
    if len( why ) > 0 then
        message = why                   '' stay on the form, fix, retry
        return
    end if
    Reload( )
    for i as int32 = 0 to txns.Count( ) - 1
        if txns[ i ].id = t.id then listState.selected = i
    next
    message = "added: " & t.payee
    page = PAGE_LIST
end sub

function LedgerApp.Update( byref e as tui.TuiEvent ) as tui.TuiCmd
    '' Pasted text arrives as one event, not as keys.
    if e.kind = tui.TUIEVENT_PASTE then
        if page <> PAGE_ENTRY then return tui.TuiNone( )
```

```
        Edit( e )
        return tui.TuiRedraw( )
    end if
    if e.kind <> tui.TUIEVENT_KEY then return tui.TuiNone( )
    select case e.key.code
    case tui.TUIKEY_F1 : page = PAGE_LIST : message = ""
    case tui.TUIKEY_F2 : StartEntry( )
    case tui.TUIKEY_F3 : page = PAGE_REPORT : message = ""
    case tui.TUIKEY_ESCAPE
        if page = PAGE_LIST then return tui.TuiQuit( 0 )
        page = PAGE_LIST
        message = ""
    case else
        if page = PAGE_LIST then
            if e.key.code = tui.TUIKEY_UP then listState.selected -= 1
            if e.key.code = tui.TUIKEY_DOWN then listState.selected += 1
            if listState.selected >= txns.Count( ) then
                listState.selected = txns.Count( ) - 1
            end if
            if listState.selected < 0 then listState.selected = 0
        elseif page = PAGE_ENTRY then
            if e.key.code = tui.TUIKEY_TAB then
                focus = ( focus + 1 ) mod 4
            elseif e.key.code = tui.TUIKEY_ENTER then
                Save( )
            else
                Edit( e )
            end if
        end if
    end select
    return tui.TuiRedraw( )
end function
```

F1, F2 and F3 work everywhere. Esc goes back to the list, or quits from the list. On the list, Up and Down move the selection, clamped to the rows that exist. On the form, Tab moves between the fields and Enter saves. Every other key goes to the field that has the focus.

So does a **paste**. `TuiTermOptionsDefault` asks the terminal for bracketed paste, and on a terminal that has it, pasted text arrives as one `TUIEVENT_PASTE` event carrying all of it, not as a run of keys. An `Update` that looks only at `TUIEVENT_KEY` drops every paste without a sound, so this one sends a paste on the form to the focused field, and ignores it on the other pages.

A widget is a value. `Edit` makes a `TuiTextInput` over the state of the focused field, hands it the event, and lets it go. The text, the cursor and the scroll live in the `TuiTextInputState`, which the app owns. Two details are worth knowing:

- **`isFocused` must be set**, or the input ignores the event.
- **A box that has never been drawn does not know how wide its field is**, so it scrolls the text as if the field were one column wide. Setting `scrollX` back to 0 leaves the scrolling to the next frame's box, which is drawn into a real area and knows.

`Save` sends the four fields through `ledger.ParseTxn`. If they are refused, the message goes into the status line and the form stays, so the mistake can be fixed. If they are accepted, the list is reloaded and the new row is selected.

View

`examples/ch35/ui/ui.tiko`

```
sub LedgerApp.View( byref f as tui.TuiFrame )
    dim as tui.TuiConstraintArray rowsC
    rowsC.Push( tui.TuiLength( 1 ) )        '' the page tabs
    rowsC.Push( tui.TuiFill( ) )            '' the page
    rowsC.Push( tui.TuiLength( 1 ) )        '' the status line
    dim as tui.TuiRectArray band = tui.TuiSplitV( f.Area( ), rowsC )

    dim as tui.TuiStatusBar tabs
    tabs.Add( " F1 list  F2 add  F3 report  Esc quit", _
              tui.TuiStyleDefault( ), tui.TUIALIGN_LEFT )
    tabs.Add( "tledger ", tui.TuiStyleDefault( ), tui.TUIALIGN_RIGHT )
    tabs.Render( f, band[0] )

    select case page
    case PAGE_LIST   : ViewList( this, f, band[1] )
    case PAGE_ENTRY  : ViewEntry( this, f, band[1] )
    case PAGE_REPORT : ViewReport( this, f, band[1] )
    end select

    dim as tui.TuiStatusBar sb
    sb.Add( " " & message, tui.TuiStyleDefault( ), tui.TUIALIGN_LEFT )
    if page = PAGE_LIST andalso txns.Count( ) > 0 then
        sb.Add( str( listState.selected + 1 ) & "/" & _
                str( txns.Count( ) ) & " ", _
                tui.TuiStyleDefault( ), tui.TUIALIGN_RIGHT )
    end if
    sb.Render( f, band[2] )
end sub
```

Three bands: the page tabs, the page, and a status line. Each page is a procedure:

examples/ch35/ui/ui.tiko

```
sub ViewList( byref app as LedgerApp, byref f as tui.TuiFrame, _
              byval area as tui.TuiRect )
    var b = Boxed( " transactions " )
    b.Render( f, area )
    dim as tui.TuiTable t
    '' A table puts no gap between columns: the widths leave one.
    t.headers.Push( "Date" )      : t.widths.Push( tui.TuiLength( 11 ) )
    t.headers.Push( "Payee" )     : t.widths.Push( tui.TuiFill( ) )
    t.headers.Push( " Category" ) : t.widths.Push( tui.TuiLength( 11 ) )
    t.headers.Push( RightIn( "Amount", 13 ) )
    t.widths.Push( tui.TuiLength( 13 ) )
    for each x in app.txns
        dim as tui.TuiRow r
        r.cells.Push( x.day.ToIsoString( ) )
        r.cells.Push( x.payee )
        r.cells.Push( " " & x.category )
        r.cells.Push( MoneyCell( x.amount, 13 ) )
        t.rows.Push( r )
    next
    t.state = @app.listState
    t.showHeader = true
    t.Render( f, b.Inner( area ) )
end sub
```

A table leaves no gap between its columns, so the widths leave one: the date is 10 characters in a column of 11, the category starts with a space, and `RightIn` always puts at least one space in front of what it aligns. Amounts are right-aligned with `MoneyCell`, so their decimal points line up for any amount of up to one character less than the column: 12 characters in the list's 13, which is `-$999,999.99`. A longer one is cut off, and ends in `...`. The report has more room, and gives its amounts 15 columns, as does the text that `--list` and `--report` print.

examples/ch35/ui/ui.tiko

```
sub ViewEntry( byref app as LedgerApp, byref f as tui.TuiFrame, _
               byval area as tui.TuiRect )
    var b = Boxed( " new transaction " )
    b.Render( f, area )
    var inner = b.Inner( area )
    dim as ustring labels( 0 to 3 ) = _
        { "Date", "Payee", "Category", "Amount" }
    dim as ustring hints( 0 to 3 ) = _
```

```
        { "YYYY-MM-DD", "who was paid, or who paid", "food, rent ...", _
          "-12.50 out, 12.50 in" }
    for i as int32 = 0 to 3
        if i * 2 >= inner.h then exit for
        dim as tui.TuiParagraph lab
        lab.text = iif( i = app.focus, "> ", "  " ) & labels( i )
        dim as int32 y = inner.y + i * 2
        lab.Render( f, tui.TuiRectOf( inner.x, y, 12, 1 ) )
        dim as tui.TuiTextInput box
        box.state = @app.inputs( i )
        box.placeholder = hints( i )
        box.isFocused = ( i = app.focus )
        box.Render( f, tui.TuiRectOf( inner.x + 12, y, _
                                      inner.w - 12, 1 ) )
    next
end sub
```

examples/ch35/ui/ui.tiko

```
sub ViewReport( byref app as LedgerApp, byref f as tui.TuiFrame, _
                byval area as tui.TuiRect )
    var b = Boxed( " monthly report " )
    b.Render( f, area )
    dim as tui.TuiTable t
    t.headers.Push( "Month" ) : t.widths.Push( tui.TuiLength( 7 ) )
    t.headers.Push( RightIn( "Income", 15 ) )
    t.widths.Push( tui.TuiLength( 15 ) )
    t.headers.Push( RightIn( "Spending", 15 ) )
    t.widths.Push( tui.TuiLength( 15 ) )
    t.headers.Push( RightIn( "Net", 15 ) )
    t.widths.Push( tui.TuiLength( 15 ) )
    for each m in app.months
        dim as tui.TuiRow r
        r.cells.Push( m.key )
        r.cells.Push( MoneyCell( m.value.income, 15 ) )
        r.cells.Push( MoneyCell( m.value.spending, 15 ) )
        var net = m.value.income + m.value.spending
        r.cells.Push( MoneyCell( net, 15 ) )
        t.rows.Push( r )
    next
    t.showHeader = true
    t.Render( f, b.Inner( area ) )
end sub
```

Testing the screens without a terminal

`TuiRun` needs a real terminal, and a test has none. But a frame depends only on the app's state, and the state changes only through `Update`, apart from the table's scroll offset, which rendering itself moves. So a test can feed `Update` the events a person would cause, render with `TuiRenderOnce`, and print the screen:

examples/ch35/screens.tiko

```
'' The three screens, drawn with no terminal: keys are fed to Update
'' as TuiRun would feed them, and each frame is printed as text.
#import "./ledger"
#import "./ui"
#import "std/tui"
#import "std/containers"

sub Dump( byref scr as tui.TuiScreen )
    dim as tui.TuiBuffer ptr bf = scr.Front( )
    for y as int32 = 0 to bf->rows - 1
        dim as ustring row
        for x as int32 = 0 to bf->cols - 1
            dim as tui.TuiCell ptr c = bf->At( x, y )
            if c->width > 0 then row &= c->text
        next
        print rtrim( row )
    next
    print
end sub

sub Press( byref app as ui.LedgerApp, byval code as tui.TuiKeyCode )
    app.Update( tui.TuiKeyEvent( code ) )
end sub

sub TypeText( byref app as ui.LedgerApp, byref s as ustring )
    for i as int32 = 0 to len( s ) - 1
        app.Update( tui.TuiCharEvent( s[ i ] ) )
    next
end sub

'' A paste comes as one event carrying all of the text.
sub Paste( byref app as ui.LedgerApp, byref s as ustring )
    dim as tui.TuiEvent e
    e.kind = tui.TUIEVENT_PASTE
    e.text = s
    app.Update( e )
end sub
```

```
ledger.OpenLedger( ":memory:" )
dim problems as containers.Array( of ustring )
ledger.ImportCsv( "bank.csv", problems )

dim app as ui.LedgerApp
dim as tui.TuiScreen scr
scr.Resize( 64, 13 )
app.Init( )
for k as int32 = 1 to 3
    Press( app, tui.TUIKEY_DOWN )
next
tui.TuiRenderOnce( app, scr )
Dump( scr )

'' F2, then a line typed into the form -- with a date that is wrong --
'' and the payee pasted.
Press( app, tui.TUIKEY_F2 )
TypeText( app, "2026-03-32" ) : Press( app, tui.TUIKEY_TAB )
Paste( app, "Corner Cafe" ) : Press( app, tui.TUIKEY_TAB )
TypeText( app, "Food" ) : Press( app, tui.TUIKEY_TAB )
TypeText( app, "-4.75" )
Press( app, tui.TUIKEY_ENTER )
tui.TuiRenderOnce( app, scr )
Dump( scr )

'' Back to the date: Tab round to it, rub out the 32, save again.
Press( app, tui.TUIKEY_TAB )
Press( app, tui.TUIKEY_END )
Press( app, tui.TUIKEY_BACKSPACE ) : Press( app, tui.TUIKEY_BACKSPACE )
TypeText( app, "13" )
Press( app, tui.TUIKEY_ENTER )
tui.TuiRenderOnce( app, scr )
Dump( scr )

Press( app, tui.TUIKEY_F3 )
tui.TuiRenderOnce( app, scr )
Dump( scr )
ledger.CloseLedger( )
```

examples/ch35/screens.out

```
 F1 list  F2 add  F3 report  Esc quit                    tledger
+ transactions ------------------------------------------------+
|Date       Payee                         Category         Amount|
```

```
|2026-01-02 Harbour Property Management rent         -$1,450.00|
|2026-01-05 Northwind Ltd               salary        $3,200.00|
|2026-01-09 Corner Grocer               food            -$86.42|
|2026-01-17 Smith, Jones & Co           utilities      -$112.30|
|2026-01-24 Corner Grocer               food            -$64.15|
|2026-02-02 Harbour Property Management rent         -$1,450.00|
|2026-02-05 Northwind Ltd               salary        $3,200.00|
|2026-02-11 City Transit                transport       -$98.00|
+------------------------------------------------------------+
                                                          4/14

 F1 list  F2 add  F3 report  Esc quit                   tledger
+ new transaction ---------------------------------------------+
|  Date      2026-03-32                                        |
|                                                              |
|  Payee     Corner Cafe                                       |
|                                                              |
|  Category  Food                                              |
|                                                              |
|> Amount    -4.75                                             |
|                                                              |
|                                                              |
+------------------------------------------------------------+
 '2026-03-32' is not a date (YYYY-MM-DD)

 F1 list  F2 add  F3 report  Esc quit                   tledger
+ transactions ------------------------------------------------+
|Date       Payee                       Category         Amount|
|2026-02-11 City Transit                transport       -$98.00|
|2026-02-14 Corner Grocer               food            -$71.80|
|2026-02-20 Smith, Jones & Co           utilities      -$104.95|
|2026-03-02 Harbour Property Management rent         -$1,450.00|
|2026-03-05 Northwind Ltd               salary        $3,200.00|
|2026-03-06 Tax refund                  income          $412.60|
|2026-03-12 Corner Grocer               food            -$93.27|
|2026-03-13 Corner Cafe                 food             -$4.75|
+------------------------------------------------------------+
 added: Corner Cafe                                       15/15

 F1 list  F2 add  F3 report  Esc quit                   tledger
+ monthly report ----------------------------------------------+
|Month           Income       Spending            Net          |
|2026-01      $3,200.00     -$1,712.87      $1,487.13          |
|2026-02      $3,200.00     -$1,724.75      $1,475.25          |
```

```
|2026-03      $3,612.60     -$1,548.02      $2,064.58          |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
+--------------------------------------------------------------+
```

The first frame is the list, three rows down. The second is the form after a first attempt with the 32nd of March: the status line says why, and the text is still there, the pasted payee with the rest. The third is the list after the date was corrected, with the new row at the bottom and selected (15 of 15). The last is the report, which now includes the coffee.

The program

examples/ch35/tledger.tiko

```
'' tledger -- a personal ledger in a terminal.
''
''     tledger [--import file.csv]... [--list | --report] ledger.db
''
'' With --list or --report it prints and exits; otherwise it opens the
'' interactive screens. Exit code: 0, 1 if an import was refused, 2 for
'' a bad command line, a ledger that cannot be opened, or no terminal
'' for the screens.
#import "./ledger"
#import "./ui"
#import "std/tui"
#import "std/containers"

sub Complain( byref msg as ustring )
    dim as int32 e = freefile( )
    open err for output as #e
    print #e, "tledger: "; msg
    close #e
end sub

sub PrintReport( )
    dim months as ledger.MonthMap
    var why = ledger.MonthlyReport( months )
    if len( why ) > 0 then Complain( why ) : return
    print "Month  "; ui.RightIn( "Income", 15 ); _
          ui.RightIn( "Spending", 15 ); ui.RightIn( "Net", 15 )
    dim total as ledger.MonthTotals
```

```
    for each m in months
        var mt = m.value
        print m.key; ui.MoneyCell( mt.income, 15 ); _
              ui.MoneyCell( mt.spending, 15 ); _
              ui.MoneyCell( mt.income + mt.spending, 15 )
        total.income = total.income + mt.income
        total.spending = total.spending + mt.spending
    next
    print "total  "; ui.MoneyCell( total.income, 15 ); _
          ui.MoneyCell( total.spending, 15 ); _
          ui.MoneyCell( total.income + total.spending, 15 )
end sub

dim imports as containers.Array( of ustring )
dim as ustring mode, path
dim as int32 i = 1
do while len( command( i ) ) > 0
    dim as ustring a = command( i )
    select case a
    case "--import"
        i += 1
        if len( command( i ) ) = 0 then
            Complain( "--import needs a file" )
            end 2
        end if
        imports.Push( command( i ) )
    case "--list", "--report"
        if len( mode ) > 0 andalso mode <> a then
            Complain( "--list and --report cannot be combined" )
            end 2
        end if
        mode = a
    case else
        if left( a, 1 ) = "-" orelse len( path ) > 0 then
            Complain( "usage: tledger [--import file.csv]... " & _
                      "[--list | --report] ledger.db" )
            end 2
        end if
        path = a
    end select
    i += 1
loop
if len( path ) = 0 then Complain( "no ledger file given" ) : end 2
```

```
var why = ledger.OpenLedger( path )
if len( why ) > 0 then Complain( path & ": " & why ) : end 2

dim as int32 status = 0
for each f in imports
    dim problems as containers.Array( of ustring )
    var added = ledger.ImportCsv( f, problems )
    for each p in problems
        Complain( p )
    next
    if problems.Count( ) > 0 then
        Complain( f & ": nothing imported" )
        status = 1
    else
        print "imported"; added; _
              iif( added = 1, " transaction", " transactions" ); _
              " from "; f
    end if
next

select case mode
case "--list"
    dim txns as ui.TxnArray
    ledger.ListTxns( txns )
    for each t in txns
        print t.day.ToIsoString( ); "  "; ui.LeftIn( t.payee, 28 ); _
              ui.LeftIn( t.category, 12 ); ui.MoneyCell( t.amount, 15 )
    next
case "--report"
    PrintReport( )
case else
    dim app as ui.LedgerApp
    var r = tui.TuiRun( app, tui.TuiTermOptionsDefault( ) )
    if r.IsOk( ) = false then Complain( "no terminal" ) : status = 2
end select
ledger.CloseLedger( )
end status
```

The program reads like the specification: arguments, opening the ledger, each import in order with its problems on stderr, then `--list`, `--report` or the screens. Asking for both `--list` and `--report` is refused rather than letting the last one win, because a person who typed both expected both. `ledger.OpenLedger` runs the migrations, so a new file needs no separate set-up step.

Running it

Run from the chapter's directory as `tledger --import bank.csv --report :memory:`:

examples/ch35/tledger.out

```
imported 14 transactions from bank.csv
Month           Income       Spending            Net
2026-01      $3,200.00     -$1,712.87      $1,487.13
2026-02      $3,200.00     -$1,724.75      $1,475.25
2026-03      $3,612.60     -$1,543.27      $2,069.33
total       $10,012.60     -$4,980.89      $5,031.71
```

A file with mistakes is refused whole, with a reason for each line, and the exit code is 1:

examples/ch35/bad.csv

```
date,payee,category,amount
2026-04-01,Corner Grocer,food,-41.10
2026-04-31,Harbour Property Management,rent,-1450.00
2026-04-05,,salary,3200.00
2026-04-09,City Transit,transport,-12.505
```

```
$ tledger --import bad.csv --list :memory:
tledger: bad.csv:3: '2026-04-31' is not a date (YYYY-MM-DD)
tledger: bad.csv:4: the payee is empty
tledger: bad.csv:5: '-12.505' has more than two decimal places
tledger: bad.csv: nothing imported
exit 1
```

Line 2 was valid, and it was not kept either: the list prints nothing. With `--list` and the good file:

```
$ tledger --import bank.csv --list :memory:
imported 14 transactions from bank.csv
2026-01-02  Harbour Property Management rent             -$1,450.00
2026-01-05  Northwind Ltd                salary            $3,200.00
2026-01-09  Corner Grocer                food                -$86.42
…
2026-03-12  Corner Grocer                food                -$93.27
```

Given a file name instead of `:memory:`, the ledger persists, and `tledger ledger.db` opens the screens on it.

Shipping

`ledger` imports SQLite with **`link static`**, so SQLite's code is linked into the program: on Windows, `tledger.exe` imports nothing but `KERNEL32.dll` and the C runtime, `ucrtbase.dll`. `tledger` is one executable with no database server and no shared library to install. Copy it, and it runs. Without `link static`, `vendor/sqlite3` links SQLite's shared library and copies it beside the program, and the two would have to travel together. The ledger is one ordinary file, which can be backed up by copying it while the program is not running, or at any time with SQL's `vacuum into 'copy.db'`.

`#foreignlib` (Chapter 30) is for the other case: a library the **project** ships, beside its sources, one file per platform. SQLite does not need it, because the toolchain's vendor package already carries it.

What it costs

- **The report reads every transaction** and adds in the program. For a personal ledger — some thousands of rows a year — that is nothing. For millions, store amounts as integer cents, so that SQL's `sum( )` stays exact, and let the database group by month.
- **The list loads every transaction** into memory, and the screens draw only what fits. A very large ledger would query one screenful at a time with `limit` and `offset`.
- **One database per program**, in a private module variable. That keeps SQLite out of every other file, at the price of not opening two ledgers at once.
- **All-or-nothing imports** make the user fix a file before any of it counts. That is the safe choice for money: a half-imported statement is worse than none.

Summary

- Keep a library you may want to replace, such as SQLite, in **one module**. Make nothing from it public, and make module variables of its types `private`, so that it stays out of the interface. A few procedures that turn its C conventions — result codes, UTF-8 text — into the module's own keep the rest of the module readable.
- Hold money in `Decimal` and days in `LocalDate`. Store them as text that sorts and round-trips exactly, and do the arithmetic in the program.
- Migrate with `pragma user_version`: one step per version, one transaction per step, and never edit a step that has shipped.
- Import through a staging table and one transaction, validate every row with the same function the form uses, and roll back on any error. Skip blank records, treat a file with none as empty, and check the result of `begin` and `commit` as well as of each statement.

- An `OrderedMap` keyed `YYYY-MM` gives a monthly report in calendar order. `MoneyFormat` with every field set prints money the same way everywhere, and its `Grouping` must end in `;0` to repeat.
- Give every column at least one space and enough width for the amounts it will hold, and say what that limit is.
- A `TuiApp` whose `View` only draws, from state that `Update` and `Reload` fill, can be tested by feeding `Update` events and rendering with `TuiRenderOnce`. Route `TUIEVENT_PASTE` as well as keys to a text input.

Exercises

1. Print the spending in each category over the whole ledger, the largest first. {solution="ex35-01″}
2. Accept a quick-entry line, `date;payee;category;amount`, and check it with `ParseTxn`. {solution="ex35-02″}
3. Add a migration, version 3, that adds a `note` column, and a fifth field to the form for it. Open a version 2 file and check that it is upgraded and keeps its rows.
4. Refuse an import line that is already in the ledger (same date, payee and amount), so that loading the same statement twice adds nothing. Compare only with the rows that were there before this import began: two identical lines in one statement, such as two coffees on the same day, are both real.
5. Add a fourth screen that shows the transactions of the month selected in the report.

Capstone: a Notepad for Windows

The other capstones run in a terminal. This one has windows. **`tpad`** is a plain-text editor in the shape of the Notepad that comes with Windows: a menu bar, a page of text, a status bar, the usual keys, and a question before it throws away unsaved work.

It is built from **PsControls**, the vendor package of owner-drawn Windows controls that comes with the toolchain, and from **PsApp**, the small framework in the same package that runs the message loop, the theme and the DPI scaling. With those doing the work Windows usually asks of a program, `tpad` is about 800 lines, a sixth of them its self-test, and much of the rest is about text files rather than windows. It joins:

- modules and a package (Chapter 15), with the part that needs no window kept apart from the part that does;
- procedure pointers and lambdas (Chapters 7 and 21), which is how every control reports back;
- `new` and `delete` for state that lives as long as a window (Chapter 11);
- files, text encodings and atomic writes through `std/fs` (Chapters 9 and 25);
- a C API — Windows itself — called directly where no wrapper is needed (Chapter 29);
- the GUI subsystem, resources and build profiles (Chapter 31).

This is not a Windows programming primer. Where a Win32 call appears, the text says in a sentence what it does and moves on.

> **ON LINUX AND MACOS**
> The window is Windows-only: PsControls draws with GDI and GDI+. The text-file module and its test, `filetest`, compile and run on Linux and macOS like every other listing, and so do the exercise solutions.

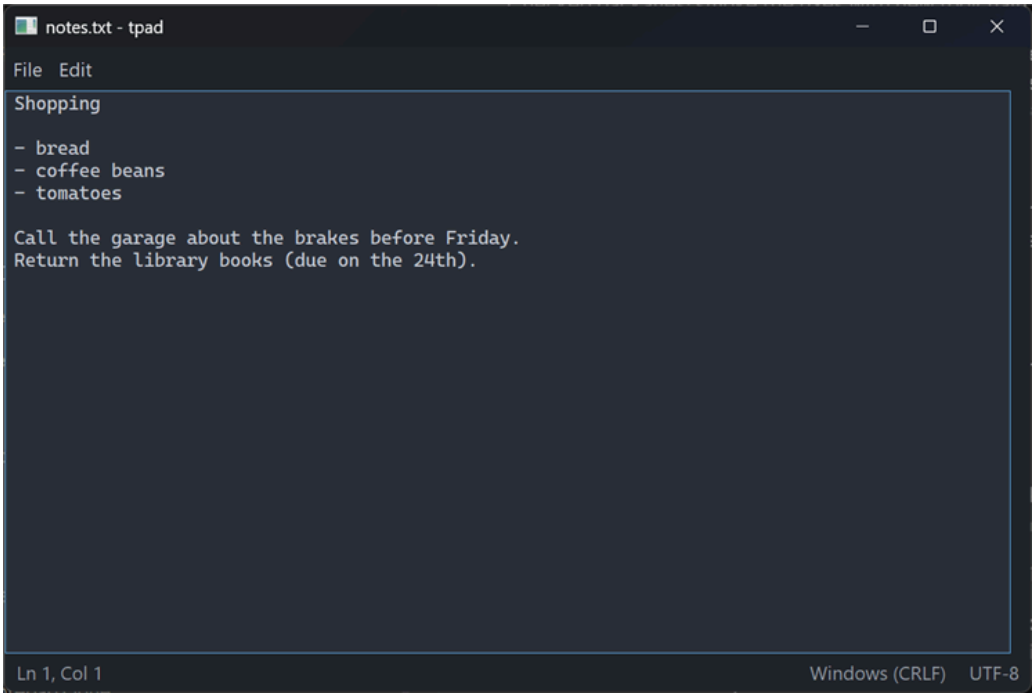


Figure 2: tpad with a short file open, in the dark theme.

The specification

```
tpad [file]
tpad --selftest
```

- **One window**: a menu bar, the text, a vertical scroll bar beside it, and a status bar.
- **File**: New (Ctrl+N), Open (Ctrl+O), Save (Ctrl+S), Save As (Ctrl+Shift+S), Exit. **Edit**: Undo, Cut, Copy, Paste and Select All, on their usual keys; Find (Ctrl+F) and Find Next (F3).
- **Text is UTF-8.** A file keeps the line ending it was written with, CR LF or LF, and its byte-order mark if it had one, when it is saved. A file with no line break is saved with CR LF. A file that is not UTF-8 opens with U+FFFD in place of what could not be read, and `tpad` says so.
- **The status bar** shows the caret's line and column, the line ending, and the encoding.
- **The title bar** shows the file's name, with a * in front while there are unsaved changes.

- **Before anything throws away unsaved text** — New, Open, closing the window — `tpad` asks: Save, Don't Save, or Cancel.
- **Find** ignores case unless asked not to, and wraps around from the end to the top.
- The window follows Windows' light or dark setting, and the DPI of the monitor it is on.
- **`--selftest`** drives the window with nobody at it, prints what happened, and exits.

The layout

```
ch36/
    tiko.toml               the package, and the target tpad with its
resources
    tpad.rc, tpad.manifest  the manifest, compiled into the program
    textfile/textfile.tiko  the document: load, save, find, line and
column
    pad/pad.tiko            the window: controls, menus, commands
    tpad.tiko               the program, and its self-test
    filetest.tiko           textfile on its own, on both targets
```

The rule is the one from Chapter 33: **the part that does not need a window does not import one.** Everything `tpad` knows about text files is in `textfile`, which imports `std/fs` and `std/strings` and nothing from Windows. It can be tested on both targets, by a program that prints, and it could serve a Linux editor unchanged. `pad` is the window, and `tpad.tiko` starts it.

The document

examples/ch36/textfile/textfile.tiko

```
#module "textfile"
#import "std/fs"
#import "std/strings"
'' Nothing here knows about windows: this module is the part of tpad
'' that compiles, and is tested, on every target.
```

examples/ch36/textfile/textfile.tiko

```
public enum LineEnding
    CrLf
    Lf
end enum

'' A text file as the editor sees it. Inside, every line break is one
'' LF, so an offset into Text is an offset into the editor control,
'' which also counts a line break as one character.
```

```
public type Document
    as ustring path             '' "" until the first Save As
    as ustring text
    as LineEnding ending = LineEnding.CrLf
    as boolean hasBom           '' the file began with a byte-order mark
    as boolean lossy            '' some bytes were not UTF-8
end type
```

A `Document` is what was read from the file, and what is needed to write it back the same way: its path, its text, its line ending, and whether it began with a byte-order mark. `lossy` records that some of its bytes could not be read. (The module is `textfile` and not `document` because a module may not export a name it shares, in any case, with its own: `TKO0870: public 'document' has the same name as its module 'document'`.)

The comment on `text` is the most important line in the module. **Inside a `Document`, a line break is one character, LF.** The editor control counts a line break as one character too, so an offset into `text` and an offset into the control are the same number, and a match found in one can be selected in the other. `ToLf` makes that true of any text, whatever its line breaks:

examples/ch36/textfile/textfile.tiko

```
'' CR LF and a lone CR both become LF.
public function ToLf( byref s as const ustring ) as ustring
    return strings.Replace( strings.Replace( s, !"\r\n", !"\n" ), _
                            !"\r", !"\n" )
end function
```

PITFALL

The text control does not hand its text back in the form it counts it. Measured with a multiline `PsTextBox` holding `ab`, CR LF, `cd`, LF, `ef`:

```
PsTextBox_GetText         10 characters: every line break comes back
as CR LF
PsTextBox_SetSel( 0, -1 )  selects 0 to 8: every line break counts
as one
```

Search the text `GetText` returns and select what you found, and the selection lands one character early for every line above the match — right on the first line, wrong everywhere else. Convert with `ToLf` first, as `pad` does in its one function that reads the control's text.

Loading

examples/ch36/textfile/textfile.tiko

```
'' Did the file start with a byte-order mark? Only its first three
'' bytes are read.
private function StartsWithBom( byref p as const ustring ) as boolean
    dim as fs.File f
    if f.OpenRead( p ).IsOk( ) = false then return false
    dim as ubyte b( 0 to 2 )
    var got = f.Read( @b( 0 ), 3 )
    if got.IsOk( ) = false then return false
    dim as longint n = got.Value( )
    if n >= 3 andalso b( 0 ) = &hEF andalso b( 1 ) = &hBB andalso _
       b( 2 ) = &hBF then return true
    return n >= 2 andalso b( 0 ) = &hFF andalso b( 1 ) = &hFE
end function

'' Reads a file into doc. Returns "" or what went wrong; on failure doc
'' is left as it was.
public function Load( byref p as const ustring, _
                      byref doc as Document ) as ustring
    var r = fs.ReadAllText( p )
    if r.IsOk( ) = false then return r.Failure( ).Message( )

    dim as Document d
    d.path = p
    dim as ustring raw = r.Value( )
    '' The first line break says how the file writes them.
    dim as int32 lf = instr( raw, !"\n" ), cr = instr( raw, !"\r" )
    if lf > 0 andalso cr <> lf - 1 then d.ending = LineEnding.Lf
    d.text = ToLf( raw )
    d.hasBom = StartsWithBom( p )
    d.lossy = instr( d.text, wchr( &hFFFD ) ) > 0
    doc = d
    return ""
end function
```

`fs.ReadAllText` does most of the work, and two things it does decide the rest of the function:

- **It removes a byte-order mark**, UTF-8's or UTF-16's, and decodes what follows. That is what a program reading text wants, but an editor must write the mark back, so `StartsWithBom` opens the file and reads its first three bytes itself.
- **It replaces bytes that are not UTF-8 with U+FFFD**, the replacement character, on both targets. A file in an old 8-bit encoding opens, but its accented letters are gone from the text, and saving writes the replacement characters. `Load` records that in `lossy`, and the window warns before anyone edits the file.

The line ending is the file's first one: an LF with a CR before it means CR LF. A file that mixes the two is saved with the first kind throughout. `Load` fills a local `Document` and copies it to `doc` only when everything has worked, so a failed open leaves the document that was there.

PITFALL

`fs.ReadAllBytes` has a three-argument form, and it is not "read the first *n* bytes". The third argument is a **limit**: a file larger than it is refused with an error, and nothing is read.

```
fs.ReadAllBytes( p, head, 3 )    '' fails on any file over 3 bytes
```

The first draft of `StartsWithBom` was that line. It compiled, every file was "too large", the function returned `false`, and the byte-order mark was quietly dropped on every save. `filetest` caught it: its first file has a mark. Read a prefix with a `File` and `Read`, as the listing does.

Saving

examples/ch36/textfile/textfile.tiko

```
'' Writes doc.text to doc.path in the file's own line ending. The old
'' file is replaced only once the new one is complete on disk.
public function Save( byref doc as Document ) as ustring
    dim as ustring s = doc.text
    if doc.ending = LineEnding.CrLf then
        s = strings.Replace( s, !"\n", !"\r\n" )
    end if
    if doc.hasBom then s = wchr( &hFEFF ) & s
    var r = fs.WriteAtomic( doc.path, s )
    if r.IsOk( ) = false then return r.Failure( ).Message( )
    return ""
end function
```

`Save` undoes what `Load` did: LF back to CR LF if that is how the file was written, and the mark back on the front. The mark is the character U+FEFF, which UTF-8 writes as the three bytes `EF BB BF`. **`fs.WriteAtomic`** writes a temporary file beside the real one and renames it into place only when the write is complete (Chapter 25), so a full disk or a crash in the middle of a save leaves the old file whole, never half of the new one.

Names, finding, line and column

examples/ch36/textfile/textfile.tiko

```
'' The file's name without its folder, or "Untitled".
public function DisplayName( byref doc as Document ) as ustring
    if len( doc.path ) = 0 then return "Untitled"
    dim as int32 i = len( doc.path )
    do until i = 0
        dim as ustring c = mid( doc.path, i, 1 )
        if c = "\" orelse c = "/" then exit do
        i -= 1
    loop
    return mid( doc.path, i + 1 )
end function

'' The title bar: "*notes.txt - tpad" while there are unsaved changes.
public function Caption( byref doc as Document, _
                         byval modified as boolean ) as ustring
    return iif( modified, "*", "" ) & DisplayName( doc ) & " - tpad"
end function
```

examples/ch36/textfile/textfile.tiko

```
'' The offset of the next match at or after start, wrapping around to
'' the top; -1 when there is none. Offsets count from 0, as the editor
'' control's do.
public function FindNext( byref text as const ustring, _
                          byref needle as const ustring, _
                          byval start as int32, _
                          byval matchCase as boolean ) as int32
    if len( needle ) = 0 then return -1
    dim as ustring hay = text, pin = needle
    if matchCase = false then
        hay = lcase( hay )
        pin = lcase( pin )
    end if
    dim as int32 at = instr( start + 1, hay, pin )
    if at = 0 then at = instr( hay, pin )
    return at - 1
end function
```

`FindNext` takes and returns offsets counted from 0, as the control's selection is, and converts at its edges: `instr` counts from 1. It searches from `start` to the end, then from the top, so a search always wraps around once. Without match case, it lowers the case of both strings and searches the copies; that keeps the offsets right, because `lcase` changes no string's length.

examples/ch36/textfile/textfile.tiko

```
'' The 1-based line and column of an offset, for the status bar.
public sub LineColumn( byref text as const ustring, _
                       byval offset as int32, _
                       byref ln as int32, byref col as int32 )
    ln = 1
    dim as int32 lineStart = 1, at = instr( text, !"\n" )
    do until at = 0 orelse at > offset
        ln += 1
        lineStart = at + 1
        at = instr( at + 1, text, !"\n" )
    loop
    col = offset + 2 - lineStart
end sub
```

Testing it on both targets

examples/ch36/filetest.tiko

```
'' The textfile module on its own, with no window: load, find, the
'' status-bar numbers, and a save that keeps the file's own format.
#import "./textfile"
#import "std/fs"
#import "std/containers"

'' Writes exact bytes, so the test does not depend on how this source
'' file's own line endings were checked out.
sub WriteBytes( byref p as ustring, bytes( ) as ubyte )
    dim as containers.Array( of ubyte ) buf
    for i as int32 = lbound( bytes ) to ubound( bytes )
        buf.Push( bytes( i ) )
    next
    fs.WriteAllBytes( p, buf )
end sub

function HexOf( byref p as ustring ) as ustring
    dim as containers.Array( of ubyte ) buf
    fs.ReadAllBytes( p, buf )
    dim as ustring s
    for i as int32 = 0 to buf.Count( ) - 1
        s &= hex( buf[i], 2 ) & " "
    next
    return rtrim( s )
end function

print "== a Windows file with a byte-order mark"
```

```
'' BOM, "Hi", CR LF, "hi there"
dim as ubyte win( ... ) = { &hEF, &hBB, &hBF, 72, 105, 13, 10, _
                            104, 105, 32, 116, 104, 101, 114, 101 }
WriteBytes( "win.tmp", win( ) )
dim as textfile.Document doc
print "load: ["; textfile.Load( "win.tmp", doc ); "]"
print "crlf: "; cbool( doc.ending = textfile.LineEnding.CrLf ); _
      ", bom: "; doc.hasBom; ", lossy: "; doc.lossy
print "length: "; len( doc.text ); ", title: "; _
      textfile.Caption( doc, false )

print "== finding"
dim as int32 at = textfile.FindNext( doc.text, "hi", 0, false )
print "'hi' from 0, any case:  "; at
at = textfile.FindNext( doc.text, "hi", at + 1, false )
print "next one:               "; at
at = textfile.FindNext( doc.text, "hi", at + 1, false )
print "next wraps to the top:  "; at
print "'hi' from 0, exact case:"; _
      textfile.FindNext( doc.text, "hi", 0, true )
print "'bye':                  "; _
      textfile.FindNext( doc.text, "bye", 0, false )

print "== line and column"
dim as int32 ln, col
dim as int32 offsets( ... ) = { 0, 2, 3, 11 }
for each offset as int32 in offsets
    textfile.LineColumn( doc.text, offset, ln, col )
    print "offset"; offset; " -> Ln"; ln; ", Col"; col
next

print "== save keeps the BOM and CR LF"
doc.text &= !"\nthird"
doc.path = "copy.tmp"
print "save: ["; textfile.Save( doc ); "] "; _
      textfile.Caption( doc, true )
print HexOf( "copy.tmp" )

print "== a Linux file that is not UTF-8"
'' "caf", the Latin-1 byte for e-acute, LF
dim as ubyte latin( ... ) = { 99, 97, 102, &hE9, 10 }
WriteBytes( "latin.tmp", latin( ) )
print "load: ["; textfile.Load( "latin.tmp", doc ); "]"
print "lf: "; cbool( doc.ending = textfile.LineEnding.Lf ); _
```

```
      ", bom: "; doc.hasBom; ", lossy: "; doc.lossy
print "last character: U+"; hex( asc( doc.text, 4 ), 4 )

print "== a file that is not there"
dim as ustring e = textfile.Load( "missing.tmp", doc )
print "refused: "; cbool( len( e ) > 0 ); ", doc kept: "; _
      textfile.DisplayName( doc )
```

examples/ch36/filetest.out

```
== a Windows file with a byte-order mark
load: []
crlf: true, bom: true, lossy: false
length:  11, title: win.tmp - tpad
== finding
'hi' from 0, any case:   0
next one:                3
next wraps to the top:   0
'hi' from 0, exact case: 3
'bye':                  -1
== line and column
offset 0 -> Ln 1, Col 1
offset 2 -> Ln 1, Col 3
offset 3 -> Ln 2, Col 1
offset 11 -> Ln 2, Col 9
== save keeps the BOM and CR LF
save: [] *copy.tmp - tpad
EF BB BF 48 69 0D 0A 68 69 20 74 68 65 72 65 0D 0A 74 68 69 72 64
== a Linux file that is not UTF-8
load: []
lf: true, bom: false, lossy: true
last character: U+FFFD
== a file that is not there
refused: true, doc kept: latin.tmp
```

The test writes its files byte by byte, so that what it reads does not depend on how this source file's own line endings were checked out. The Windows file keeps its mark and its CR LF through a load, an edit and a save. The Latin-1 file opens with LF, as `lossy`, with U+FFFD where the e-acute was. A file that is not there is refused, and `doc` still holds the file before it.

The window

An application and a form

examples/ch36/pad/pad.tiko

```
#module "pad"
#import "vendor/PsControls/PsApp"
#import "vendor/PsControls/PsMenuBar"
#import "vendor/PsControls/PsPopupMenu"
#import "vendor/PsControls/PsTextBox"
#import "vendor/PsControls/PsVScrollBar"
#import "vendor/PsControls/PsStatusBar"
#import "vendor/PsControls/PsMessageBox"
#import "vendor/PsControls/PsButton"
#import "vendor/PsControls/PsCheckBox"
#import "../textfile"

using PsApp
using PsTheme
using PsBufferPaint
using PsMenuBar
using PsPopupMenu
using PsTextBox
using PsVScrollBar
using PsStatusBar
using PsMessageBox
using PsButton
using PsCheckBox
using gui
```

Each control is a module of its own, and a program imports the ones it uses and links only those. `using` makes their names bare, so a call is `PsTextBox_SetText`, as the package's documentation writes it, rather than `PsTextBox.PsTextBox_SetText`. `gui` is `sys/win/gui`, which PsApp imports for the program, and which holds the file dialogs.

PsApp supplies two types. A **`PsApplication`** is the program: its `Init` sets up COM, GDI+, the DPI awareness and the theme, and its `Run` is the message loop, which runs until the main window closes. A **`PsForm`** is a window. It reports what happens to it through procedure pointers set by `OnSize`, `OnClose`, `OnCommand` and the rest, each of which receives the form first; a message with no handler gets Windows' default behaviour.

`examples/ch36/pad/pad.tiko`

```
'' Every command the menus and the keyboard can give.
public enum Cmd
    CmdNew = 1000
    CmdOpen
    CmdSave
```

```
    CmdSaveAs
    CmdExit
    CmdUndo
    CmdCut
    CmdCopy
    CmdPaste
    CmdSelectAll
    CmdFind
    CmdFindNext
end enum

'' The two things a person answers: a question in a message box, and
'' a file name in a file dialog. Each is reached through a pointer, so
'' a test can answer in the person's place.
public type AskProc as function( byval f as PsForm ptr, _
    byref text as ustring, byval buttons as long ) as long
public type PickProc as function( byval f as PsForm ptr, _
    byval saving as boolean, byref suggest as ustring ) as ustring

'' One open file and its window. The form's Tag points here.
public type Editor
    as PsForm ptr form
    as HWND text, scroll, status
    as textfile.Document doc
    as ustring findWhat
    as boolean matchCase
    as AskProc ask
    as PickProc pick
end type

enum
    IDC_BAR = 100
    IDC_TEXT
    IDC_SCROLL
    IDC_STATUS
    IDC_WHAT
    IDC_CASE
end enum

'' Posted to the form after a key or a click in the text, so the
'' status bar is read once the control has moved the caret.
const WM_CARETMOVED = WM_APP + 1

function EditorOf( byval h as HWND ) as Editor ptr
```

```
    return PsForm.FromHwnd( h )->Tag
end function
```

`Editor` is everything `tpad` keeps about one window, and **the form holds a pointer to it in its `Tag`**. That is the answer to a question every callback raises. A control's callback receives the control's window handle and nothing else; `PsForm.FromHwnd` finds the form that handle is on, and the form's `Tag` finds the `Editor`. `EditorOf` does both. So no part of the program needs a `dim shared` variable, and a second `tpad` window would be a second `Editor` with nothing shared.

`Cmd` names every command once. A menu item, a key and the self-test all end up passing one of these numbers to the same procedure.

The last two fields are the subject of a later section. **`ask` and `pick` are the two places a person answers `tpad`** — a question in a message box, and a file name in a file dialog. Both are procedure pointers, and `Create` points them at the real thing:

`examples/ch36/pad/pad.tiko`

```
'' The real answers come from a person.
function AskPerson( byval f as PsForm ptr, byref text as ustring, _
                    byval buttons as long ) as long
    return PsMessageBox_Show( f->hWindow, text, "tpad", _
        iif( buttons = MBX_BTN_OK, MBX_ICON_INFO, MBX_ICON_QUESTION ), _
        buttons )
end function

function PickFile( byval f as PsForm ptr, byval saving as boolean, _
                   byref suggest as ustring ) as ustring
    if saving then
        dim as CISaveFileDialog d
        d.AddFileType( "Text files", "*.txt" )
        d.AddFileType( "All files", "*.*" )
        d.SetFileTypes( )
        d.SetDefaultExtension( "txt" )
        d.SetFileName( suggest )
        if d.ShowSave( f->hWindow ) <> S_OK then return ""
        return d.GetResultString( SIGDN_FILESYSPATH )
    end if
    dim as CIOpenFileDialog d
    d.AddFileType( "Text files", "*.txt" )
    d.AddFileType( "All files", "*.*" )
    d.SetFileTypes( )
    if d.ShowOpen( f->hWindow ) <> S_OK then return ""
```

```
    return d.GetResultString( SIGDN_FILESYSPATH )
end function
```

`PsMessageBox_Show` puts up a themed message box and waits for an answer. Its button sets are named presets; `MBX_BTN_SAVE_DONTSAVE_CANCEL` returns `IDYES`, `IDNO` or `IDCANCEL`, because Windows has no "Save" result and a save question is a yes-no-cancel question with better captions. `CISaveFileDialog` and `CIOpenFileDialog` are the Windows file dialogs, from `sys/win/gui`, and `SIGDN_FILESYSPATH` asks for the result as an ordinary path.

The text and its scroll bar

`examples/ch36/pad/pad.tiko`

```
sub BuildText( byval f as PsForm ptr, byval ed as Editor ptr )
    ed->text = PsTextBox_Create( f->hWindow, IDC_TEXT, true )
    PsTextBox_SetFont( ed->text, f->Font( PSFONT_MONO ) )
    PsTextBox_SetLimitText( ed->text, &h7FFFFFFE )
    PsTextBox_SetMargins( ed->text, f->ScaleX( 6 ), f->ScaleX( 6 ) )

    '' The person typed, cut, pasted or undid something.
    PsTextBox_SetChangeCallback( ed->text, _
        sub( byval h as HWND )
            ShowTitle( EditorOf( h ) )
            ShowStatus( EditorOf( h ) )
        end sub )

    '' The caret may have moved: look once the control has acted.
    PsTextBox_SetMessageCallback( ed->text, _
        function( byval m as PSTEXTBOX_MESSAGEINFO ptr ) as boolean
            select case m->uMsg
            case WM_KEYDOWN, WM_KEYUP, WM_LBUTTONDOWN, WM_LBUTTONUP
                PostMessageW( GetParent( m->hTextBox ), _
                              WM_CARETMOVED, 0, 0 )
            end select
            return false
        end function )
    f->OnMessage( WM_CARETMOVED, _
        function( byval f as PsForm ptr, byval uMsg as UINT, _
                  byval wParam as WPARAM, byval lParam as LPARAM, _
                  byref result as LRESULT ) as boolean
            ShowStatus( f->Tag )
            return true
        end function )
```

```
    '' The control draws no scroll bar of its own; it tells one.
    ed->scroll = PsVScrollBar_Create( f->hWindow, IDC_SCROLL )
    PsTextBox_SetScrollChangedCallback( ed->text, _
        sub( byval h as HWND )
            dim as integer total, page, top
            PsTextBox_GetVScrollInfo( h, total, page, top )
            PsVScrollBar_SetRange( EditorOf( h )->scroll, total, _
                                   page, top )
        end sub )
    PsVScrollBar_SetScrollCallback( ed->scroll, _
        sub( byval h as HWND, byval newPos as integer )
            PsTextBox_ScrollToLine( EditorOf( h )->text, newPos )
        end sub )
end sub
```

The third argument to `PsTextBox_Create` makes a **multiline** box: Enter starts a new line, Tab inserts a tab, and long lines wrap. `f->Font( PSFONT_MONO )` is the form's monospaced font, Cascadia Mono where it is installed and Consolas where it is not. The text limit is raised from the control's default to nearly 2^31 characters.

Four callbacks, each a lambda, connect the box to the rest of the window:

- **The change callback** fires when *the person* changes the text: typing, cutting, pasting, undoing. It never fires for `PsTextBox_SetText`. Every PsControls setter is silent in the same way, so a program can set a control's value inside its own handler without being called back in the middle of it. `tpad` relies on that when it loads a file.
- **The message callback** sees the box's keyboard and mouse messages *before* the box acts on them. A key that moves the caret has not moved it yet, so reading the caret position here gives the old one. Instead the callback posts `WM_CARETMOVED` to the form: a posted message waits in the queue until the current one is finished, by which time the caret has moved, and the form's handler for it updates the status bar. `false` lets the box handle the message as usual.
- **The scroll callbacks** connect the box to a `PsVScrollBar`. The box draws no scroll bar of its own. It reports its line count, page size and top line whenever they may have changed, and the scroll bar reports a new position when it is dragged.

The status bar

`examples/ch36/pad/pad.tiko`

```
sub BuildStatus( byval f as PsForm ptr, byval ed as Editor ptr )
    ed->status = PsStatusBar_Create( f->hWindow, IDC_STATUS )
    PsStatusBar_AddPanel( ed->status, "" )      '' Ln, Col: springs
    PsStatusBar_AddPanel( ed->status, "" )      '' line ending
```

```
    PsStatusBar_AddPanel( ed->status, "" )      '' encoding
    PsStatusBar_SetExpandPanel( ed->status, 0 )
    '' The bar lays the panels out; what is inside one, it leaves to
    '' the program. Without this, the panels are blank.
    PsStatusBar_SetPaintCallback( ed->status, _
        sub( byval p as PSSTATUSBAR_PAINTINFO ptr )
            dim as PSTHEME_COLORS t = PsApp_Instance( )->Theme( )
            p->b->SetColors( t.TextDim, t.Window )
            p->b->PaintRect( @p->rc )
            dim as RECT r = p->rc
            r.left += PsStatusBar_GetPadding( p->hStatusBar )
            p->b->SetFont( PsStatusBar_GetFont( p->hStatusBar ) )
            p->b->PaintText( p->wszCaption, @r, DT_LEFT )
        end sub )
end sub
```

`PsStatusBar` lays its panels out: each is as wide as its text, and `SetExpandPanel` makes the first one take the space left over. **It does not draw what is in a panel.** That is the paint callback's job, and without one the bar is a blank strip. This one paints in the theme's colours, which it asks the application for each time, so a switch between light and dark needs nothing more.

`examples/ch36/pad/pad.tiko`

```
'' The editor's text, with one LF per line break, as textfile wants.
function TextOf( byval ed as Editor ptr ) as ustring
    return textfile.ToLf( PsTextBox_GetText( ed->text ) )
end function

sub ShowTitle( byval ed as Editor ptr )
    SetWindowTextW( ed->form->hWindow, textfile.Caption( ed->doc, _
                    PsTextBox_GetModify( ed->text ) ) )
end sub

sub ShowStatus( byval ed as Editor ptr )
    dim as integer first, last
    PsTextBox_GetSel( ed->text, first, last )
    dim as int32 ln, col
    textfile.LineColumn( TextOf( ed ), last, ln, col )
    PsStatusBar_SetText( ed->status, 0, _
                         "Ln " & ln & ", Col " & col )
    PsStatusBar_SetText( ed->status, 1, _
        iif( ed->doc.ending = textfile.LineEnding.CrLf, _
             "Windows (CRLF)", "Unix (LF)" ) )
    PsStatusBar_SetText( ed->status, 2, _
```

```
        iif( ed->doc.hasBom, "UTF-8 with BOM", "UTF-8" ) )
end sub
```

`TextOf` is the one place `pad` reads the control's text, and it converts to the one-LF form at once. `ShowTitle` reads the box's **modified flag**, which the control sets on every edit the person makes and `pad` clears after a load or a save. `ShowStatus` takes the end of the selection as the caret, and lets `textfile` count lines.

Layout

`examples/ch36/pad/pad.tiko`

```
sub Main_OnSize( byval f as PsForm ptr, byval cx as long, _
                 byval cy as long )
    dim as long barH = f->ScaleY( 30 ), statusH = f->ScaleY( 26 )
    dim as long sbW = f->ScaleX( 14 )
    dim as long textH = cy - barH - statusH
    SetWindowPos( f->Item( IDC_BAR ), null, 0, 0, cx, barH, _
                  SWP_NOZORDER or SWP_SHOWWINDOW )
    SetWindowPos( f->Item( IDC_TEXT ), null, 0, barH, cx - sbW, _
                  textH, SWP_NOZORDER or SWP_SHOWWINDOW )
    SetWindowPos( f->Item( IDC_SCROLL ), null, cx - sbW, barH, sbW, _
                  textH, SWP_NOZORDER or SWP_SHOWWINDOW )
    SetWindowPos( f->Item( IDC_STATUS ), null, 0, cy - statusH, cx, _
                  statusH, SWP_NOZORDER or SWP_SHOWWINDOW )
end sub
```

PsControls are created **hidden and zero-sized**. Nothing is on screen until the form's size handler places each one and shows it; `SWP_SHOWWINDOW` in `SetWindowPos` does both at once. The handler runs whenever the window's size changes, including the first time it is shown. Every literal is passed through `ScaleX` or `ScaleY`, which turn device-independent units into pixels at the form's DPI. Thirty units is thirty pixels on an ordinary monitor and sixty on one at 200%.

Menus and keys

`examples/ch36/pad/pad.tiko`

```
function AddMenu( byval f as PsForm ptr, byval bar as HWND, _
                  byref title as ustring ) as HWND
    dim as long i = PsMenuBar_AddItem( bar, title, 0 )
    dim as HWND m = PsPopupMenu_Create( f->hWindow )
    PsMenuBar_SetDropDown( bar, i, m )
    return m
end function
```

```
sub BuildMenus( byval f as PsForm ptr )
    dim as HWND bar = PsMenuBar_Create( f->hWindow, IDC_BAR )
    PsMenuBar_SetCommandCallback( bar, _
        sub( byval h as HWND, byval id as long )
            Execute( EditorOf( h ), id )
        end sub )
    dim as HWND m = AddMenu( f, bar, "File" )
    PsPopupMenu_AddItem( m, CmdNew, "New", "Ctrl+N" )
    PsPopupMenu_AddItem( m, CmdOpen, "Open...", "Ctrl+O" )
    PsPopupMenu_AddItem( m, CmdSave, "Save", "Ctrl+S" )
    PsPopupMenu_AddItem( m, CmdSaveAs, "Save As...", "Ctrl+Shift+S" )
    PsPopupMenu_AddSeparator( m )
    PsPopupMenu_AddItem( m, CmdExit, "Exit" )
    m = AddMenu( f, bar, "Edit" )
    PsPopupMenu_AddItem( m, CmdUndo, "Undo", "Ctrl+Z" )
    PsPopupMenu_AddSeparator( m )
    PsPopupMenu_AddItem( m, CmdCut, "Cut", "Ctrl+X" )
    PsPopupMenu_AddItem( m, CmdCopy, "Copy", "Ctrl+C" )
    PsPopupMenu_AddItem( m, CmdPaste, "Paste", "Ctrl+V" )
    PsPopupMenu_AddItem( m, CmdSelectAll, "Select All", "Ctrl+A" )
    PsPopupMenu_AddSeparator( m )
    PsPopupMenu_AddItem( m, CmdFind, "Find...", "Ctrl+F" )
    PsPopupMenu_AddItem( m, CmdFindNext, "Find Next", "F3" )

    '' The text control handles Ctrl+Z, X, C, V and A itself. These are
    '' the keys only tpad knows.
    f->AddAccelerator( FVIRTKEY or FCONTROL, asc( "N" ), CmdNew )
    f->AddAccelerator( FVIRTKEY or FCONTROL, asc( "O" ), CmdOpen )
    f->AddAccelerator( FVIRTKEY or FCONTROL, asc( "S" ), CmdSave )
    f->AddAccelerator( FVIRTKEY or FCONTROL or FSHIFT, asc( "S" ), _
                       CmdSaveAs )
    f->AddAccelerator( FVIRTKEY or FCONTROL, asc( "F" ), CmdFind )
    f->AddAccelerator( FVIRTKEY, VK_F3, CmdFindNext )
    f->CreateAcceleratorTable( )
    f->OnCommand( _
        function( byval f as PsForm ptr, byval id as long, _
                  byval code as long, byval hCtl as HWND ) as boolean
            if id < CmdNew orelse id > CmdFindNext then return false
            Execute( f->Tag, id )
            return true
        end function )
end sub
```

A `PsMenuBar` holds the titles; each title drops down a `PsPopupMenu`. The text after an item — `"Ctrl+N"` — is only shown. What makes the key work is the **accelerator**

table: `AddAccelerator` maps a key and its modifiers to a command number, and the message loop turns the key into a `WM_COMMAND` for the form before the text box sees it. The form's command callback hands any `Cmd` to `Execute`, and returns `false` for anything else so that it gets the default handling.

The edit keys are left out of the table on purpose. The text control already handles Ctrl+Z, X, C, V and A, wherever the keyboard focus is in it; an accelerator would take the key away from it.

Commands

`examples/ch36/pad/pad.tiko`

```
'' The one place a command is carried out, wherever it came from.
public sub Execute( byval ed as Editor ptr, byval id as long )
    select case id
    case CmdNew
        if MayDiscard( ed ) then
            dim as textfile.Document blank
            ShowDocument( ed, blank )
        end if
    case CmdOpen
        if MayDiscard( ed ) then
            dim as ustring p = ed->pick( ed->form, false, "" )
            if len( p ) then OpenPath( ed, p )
        end if
    case CmdSave
        SaveDocument( ed, false )
    case CmdSaveAs
        SaveDocument( ed, true )
    case CmdExit
        ed->form->Close( )
    case CmdUndo
        PsTextBox_Undo( ed->text )
    case CmdCut
        PsTextBox_Cut( ed->text )
    case CmdCopy
        PsTextBox_Copy( ed->text )
    case CmdPaste
        PsTextBox_Paste( ed->text )
    case CmdSelectAll
        PsTextBox_SelectAll( ed->text )
    case CmdFind
        AskWhatToFind( ed )
    case CmdFindNext
        if len( ed->findWhat ) then
```

```
            FindAgain( ed )
        else
            AskWhatToFind( ed )
        end if
    end select
end sub
```

One procedure carries out every command, whether it came from a menu, a key or the self-test. The menus and the keys say *which* command; `Execute` says *what* it does. Adding a command means adding a `Cmd`, an item, perhaps a key, and a `case`.

Files, and the question before discarding

`examples/ch36/pad/pad.tiko`

```
'' Puts doc in the window, replacing what was there.
sub ShowDocument( byval ed as Editor ptr, _
                  byref doc as textfile.Document )
    ed->doc = doc
    PsTextBox_SetText( ed->text, doc.text )
    PsTextBox_SetModify( ed->text, false )
    PsTextBox_SetSel( ed->text, 0, 0 )
    ShowTitle( ed )
    ShowStatus( ed )
end sub

'' Opens path; says why not if it cannot.
public function OpenPath( byval ed as Editor ptr, _
                          byref path as ustring ) as boolean
    dim as textfile.Document doc
    dim as ustring e = textfile.Load( path, doc )
    if len( e ) then
        ed->ask( ed->form, e, MBX_BTN_OK )
        return false
    end if
    ShowDocument( ed, doc )
    if doc.lossy then
        ed->ask( ed->form, textfile.DisplayName( doc ) & _
            " is not UTF-8. Characters that could not be read show" & _
            " as U+FFFD, and saving writes them that way.", MBX_BTN_OK )
    end if
    return true
end function

'' Saves, asking for a name when there is none (or when asked to).
function SaveDocument( byval ed as Editor ptr, _
```

```
                        byval newName as boolean ) as boolean
    if newName orelse len( ed->doc.path ) = 0 then
        dim as ustring p = ed->pick( ed->form, true, _
                                     textfile.DisplayName( ed->doc ) )
        if len( p ) = 0 then return false
        ed->doc.path = p
    end if
    ed->doc.text = TextOf( ed )
    dim as ustring e = textfile.Save( ed->doc )
    if len( e ) then
        ed->ask( ed->form, e, MBX_BTN_OK )
        return false
    end if
    PsTextBox_SetModify( ed->text, false )
    ShowTitle( ed )
    return true
end function

'' True when it is all right to throw the text away: it is saved,
'' the person saved it now, or said not to.
function MayDiscard( byval ed as Editor ptr ) as boolean
    if PsTextBox_GetModify( ed->text ) = false then return true
    select case ed->ask( ed->form, "Do you want to save changes to " & _
                         textfile.DisplayName( ed->doc ) & "?", _
                         MBX_BTN_SAVE_DONTSAVE_CANCEL )
    case IDYES
        return SaveDocument( ed, false )
    case IDNO
        return true
    end select
    return false
end function
```

`MayDiscard` is the specification's question, in one place. New and Open call it before they replace the text, and the form's close handler, below, calls it before the window goes. **Save** in the answer saves, and continues only if the save worked: a save that is cancelled at the file dialog, or fails, keeps the window and its text.

`SaveDocument` reads the text out of the control into the document, saves, and clears the modified flag only after `textfile.Save` reports success. An error from either module comes back as text, and goes to the person as a message box with an OK button.

Find

`examples/ch36/pad/pad.tiko`

```
sub FindAgain( byval ed as Editor ptr )
    if len( ed->findWhat ) = 0 then exit sub
    dim as integer first, last
    PsTextBox_GetSel( ed->text, first, last )
    dim as int32 at = textfile.FindNext( TextOf( ed ), ed->findWhat, _
                                         last, ed->matchCase )
    if at < 0 then
        ed->ask( ed->form, "Cannot find """ & ed->findWhat & """", _
                 MBX_BTN_OK )
        exit sub
    end if
    PsTextBox_SetSel( ed->text, at, at + len( ed->findWhat ) )
    SendMessageW( ed->text, EM_SCROLLCARET, 0, 0 )
    ShowStatus( ed )
end sub

'' The Find box: a modal form that only collects the search.
sub FindBox_OnSize( byval f as PsForm ptr, byval cx as long, _
                    byval cy as long )
    '' Two rows, each as tall as a button: the search and its button,
    '' then the option and Cancel.
    dim as long pad = f->ScaleX( 12 ), bw, bh
    PsButton_GetIdealSize( f->Item( IDOK ), bw, bh )
    dim as long w = cx - 3 * pad - bw
    SetWindowPos( f->Item( IDC_WHAT ), null, pad, pad, w, bh, _
                  SWP_NOZORDER or SWP_SHOWWINDOW )
    SetWindowPos( f->Item( IDC_CASE ), null, pad, 2 * pad + bh, _
                  w, bh, SWP_NOZORDER or SWP_SHOWWINDOW )
    SetWindowPos( f->Item( IDOK ), null, cx - pad - bw, pad, bw, bh, _
                  SWP_NOZORDER or SWP_SHOWWINDOW )
    SetWindowPos( f->Item( IDCANCEL ), null, cx - pad - bw, _
                  2 * pad + bh, bw, bh, SWP_NOZORDER or SWP_SHOWWINDOW )
end sub

sub FindBox_Ok( byval h as HWND, byval id as long )
    dim as PsForm ptr f = PsForm.FromHwnd( h )
    dim as Editor ptr ed = f->Tag
    ed->findWhat = PsTextBox_GetText( f->Item( IDC_WHAT ) )
    ed->matchCase = PsCheckBox_GetChecked( f->Item( IDC_CASE ) )
    f->EndModal( IDOK )
end sub

sub AskWhatToFind( byval ed as Editor ptr )
    dim as PsForm ptr f = PsApp_Instance( )->NewForm( "Find", 420, _
```

```
        150, WS_POPUP or WS_CAPTION or WS_SYSMENU or WS_CLIPCHILDREN, _
        WS_EX_CONTROLPARENT or WS_EX_DLGMODALFRAME, ed->form->hWindow )
    f->Tag = ed
    dim as HWND hWhat = PsTextBox_Create( f->hWindow, IDC_WHAT )
    PsTextBox_SetText( hWhat, ed->findWhat )
    PsTextBox_SetSelectOnFocus( hWhat, true )
    dim as HWND hCase = PsCheckBox_Create( f->hWindow, IDC_CASE )
    PsCheckBox_SetText( hCase, "Match case" )
    PsCheckBox_SetChecked( hCase, ed->matchCase )
    dim as HWND hOk = PsButton_Create( f->hWindow, IDOK )
    PsButton_SetText( hOk, "Find Next" )
    PsButton_SetClickCallback( hOk, @FindBox_Ok )
    dim as HWND hCancel = PsButton_Create( f->hWindow, IDCANCEL )
    PsButton_SetText( hCancel, "Cancel" )
    PsButton_SetClickCallback( hCancel, _
        sub( byval h as HWND, byval id as long )
            PsForm.FromHwnd( h )->EndModal( IDCANCEL )
        end sub )
    f->OnSize( @FindBox_OnSize )
    f->SetClientSize( 420, 112 )
    if f->ShowModal( ed->form->hWindow ) = IDOK then FindAgain( ed )
end sub
```

The Find box is a **modal** form. `ShowModal` disables the main window, runs the message loop until the box ends it, enables the main window again, destroys the box and returns the code passed to `EndModal`. Esc and the close box end it with `IDCANCEL`. The box only collects what to look for, into the `Editor` its `Tag` points at; `FindAgain` does the searching, so F3 can repeat a search without the box. A match is selected and `EM_SCROLLCARET` scrolls it into view. That is a classic edit-control message, and a `PsTextBox` passes such messages to its edit control unchanged.

Creating it all

`examples/ch36/pad/pad.tiko`

```
'' Builds the main window, with an empty, untitled document.
public function Create( byref app as PsApplication ) as Editor ptr
    dim as Editor ptr ed = new Editor
    ed->ask = @AskPerson
    ed->pick = @PickFile
    dim as PsForm ptr f = app.NewForm( "tpad", 720, 480 )
    ed->form = f
    f->Tag = ed

    BuildMenus( f )
```

```
    BuildText( f, ed )
    BuildStatus( f, ed )

    f->OnSize( @Main_OnSize )
    '' Asked before the window closes, however it is closed.
    f->OnClose( _
        function( byval f as PsForm ptr ) as boolean
            return MayDiscard( f->Tag )
        end function )
    '' A new DPI gives every control the form's UI font: put the
    '' monospaced one back.
    f->OnDpiChanged( _
        sub( byval f as PsForm ptr, byval oldDpi as long, _
             byval newDpi as long )
            dim as Editor ptr ed = f->Tag
            PsTextBox_SetFont( ed->text, f->Font( PSFONT_MONO ) )
        end sub )
    f->OnDestroy( _
        sub( byval f as PsForm ptr )
            delete cast( Editor ptr, f->Tag )
            f->Tag = 0
        end sub )

    ShowTitle( ed )
    ShowStatus( ed )
    return ed
end function
```

`Create` allocates the `Editor` with `new`, because it must outlive the call: it lives exactly as long as the window, and the form's destroy handler deletes it. The close handler is the specification's question again; returning `false` from it cancels the close, so Cancel keeps the window open. The DPI handler is the subject of this chapter's last pitfall:

PITFALL

When a form moves to a monitor with a different DPI, PsApp re-creates its fonts at the new size and **gives every control the form's UI font** — including a control that was given another font. Without the DPI handler, `tpad`'s text turns from Cascadia Mono into Segoe UI the first time the window is dragged to a monitor at another scale, and nothing else changes. The self-test below checks it, by sending the form the message Windows sends:

```
== moved to a monitor at twice the DPI
  the text is still monospaced: true
```

With the handler's one line commented out, that line reads `false`.

The program

examples/ch36/tpad.tiko

```
#cmdline "-s gui"
#import "./pad"
#import "std/fs"
#import "std/strings"
#import "vendor/PsControls/PsApp"
#import "vendor/PsControls/PsTextBox"
#import "vendor/PsControls/PsStatusBar"
#import "vendor/PsControls/PsMessageBox"

using PsApp
using PsTextBox
using PsStatusBar
using PsMessageBox
```

examples/ch36/tpad.tiko

```
dim as PsApplication app
app.Init( )
dim as pad.Editor ptr ed = pad.Create( app )
if command( 1 ) = "--selftest" then end SelfTest( ed )
if len( command( 1 ) ) then pad.OpenPath( ed, command( 1 ) )
end app.Run( ed->form )
```

`#cmdline "-s gui"` builds for the GUI subsystem (Chapter 31), so starting `tpad` from Explorer opens no console window. `app.Run` shows the main window and runs the message loop until the window is destroyed, and its result is the program's exit code. A file named on the command line is opened before the loop starts; if it cannot be, the person is told and gets an empty window.

The package's `tiko.toml` names the program and its resources:

examples/ch36/tiko.toml

```
# tpad: `tiko build tpad` writes out/debug/tpad.exe
[package]
name = "tpad"
```

```
[target.tpad]
main      = "tpad.tiko"
resources = ["tpad.rc"]

[target.tpad.defines]
# Every Win32 name means its W (UTF-16) form.
UNICODE = "1"
```

`UNICODE` is defined for the whole build, as the PsControls package requires: it makes every bare Win32 name, such as `SetWindowText` or `MessageBox`, mean the function that takes UTF-16 text. The manifest that `tpad.rc` compiles in declares per-monitor DPI awareness and asks for version 6 of the Windows common controls.

Testing a window with nobody at it

The harness that checks this book's listings runs every program and compares its output with what the book prints. A window waits for a person. So `tpad` takes `--selftest`, which **drives the real window — the real controls, the real callbacks, the real `Execute` — and prints what it saw**.

The only things it cannot drive are the two that wait for a person: the message box and the file dialog. That is why they are behind `ask` and `pick`. The test points them at a script:

examples/ch36/tpad.tiko

```
'' The person, scripted: what the test answers to each question.
function ScriptedAsk( byval f as PsForm ptr, byref text as ustring, _
                      byval buttons as long ) as long
    static as long replies( ... ) = { IDCANCEL, IDNO, IDNO }
    static as int32 n
    print "  asked: "; text
    if buttons = MBX_BTN_OK then return IDOK
    n += 1
    print "  answered: "; iif( replies( n - 1 ) = IDNO, _
                               "Don't Save", "Cancel" )
    return replies( n - 1 )
end function

function ScriptedPick( byval f as PsForm ptr, byval saving as boolean, _
                       byref suggest as ustring ) as ustring
    print "  picked: lf.tmp"
    return "lf.tmp"
end function
```

examples/ch36/tpad.tiko

```
'' Lets posted messages arrive, as the message loop would.
sub Pump( )
    dim as MSG m
    do until PeekMessageW( @m, null, 0, 0, PM_REMOVE ) = 0
        TranslateMessage( @m )
        DispatchMessageW( @m )
    loop
end sub

'' A key, delivered where the keyboard would deliver it.
sub Press( byval ed as pad.Editor ptr, byval vk as long )
    SendMessageW( PsTextBox_GetRichEditHandle( ed->text ), _
                  WM_KEYDOWN, vk, 0 )
    Pump( )
end sub

sub Type_( byval ed as pad.Editor ptr, byref s as ustring )
    for i as int32 = 1 to len( s )
        SendMessageW( PsTextBox_GetRichEditHandle( ed->text ), _
                      WM_CHAR, asc( s, i ), 0 )
    next
    Pump( )
end sub

sub Report( byval ed as pad.Editor ptr )
    '' Win32 fills a wstring buffer. Printed to a pipe or a file, a
    '' wstring is written as raw UTF-16, so it is copied to a ustring,
    '' which is written as UTF-8.
    dim as wstring * 260 buffer
    GetWindowTextW( ed->form->hWindow, buffer, 260 )
    dim as ustring title = buffer
    print "  title:  "; title
    print "  status: "; PsStatusBar_GetText( ed->status, 0 ); " | "; _
          PsStatusBar_GetText( ed->status, 1 ); " | "; _
          PsStatusBar_GetText( ed->status, 2 )
end sub
```

`Press` and `Type_` send the text control's own edit window the messages the keyboard would: `WM_KEYDOWN` for a key that moves, `WM_CHAR` for a character. They take the same path a key takes, through the message callback and the change callback. `Pump` then delivers whatever was posted, which is how `WM_CARETMOVED` reaches the form; the self-test runs before `app.Run`, so there is no message loop doing it.

`examples/ch36/tpad.tiko`

```
function SelfTest( byval ed as pad.Editor ptr ) as long
    ed->ask = @ScriptedAsk
    ed->pick = @ScriptedPick
    fs.WriteAllText( "lf.tmp", !"one\ntwo\nthree\n" )

    print "== a new window"
    Report( ed )
    print "== the layout, at 600 x 400"
    ed->form->SetClientSize( 600, 400 )
    dim as RECT client, text, bar, status
    GetClientRect( ed->form->hWindow, @client )
    GetWindowRect( ed->text, @text )
    GetWindowRect( ed->scroll, @bar )
    GetWindowRect( ed->status, @status )
    print "  text meets the scroll bar: "; _
          cbool( text.right = bar.left )
    print "  text meets the status bar: "; _
          cbool( text.bottom = status.top )
    print "  status bar is as wide as the window: "; _
          cbool( status.right - status.left = client.right )

    print "== File > Open"
    pad.Execute( ed, pad.CmdOpen )
    Report( ed )
    print "== Down, Down, End, and type ""!"""
    Press( ed, VK_DOWN )
    Press( ed, VK_DOWN )
    Press( ed, VK_END )
    Type_( ed, "!" )
    Report( ed )

    print "== Find Next, for ""T"", twice"
    ed->findWhat = "T"
    pad.Execute( ed, pad.CmdFindNext )
    Report( ed )
    pad.Execute( ed, pad.CmdFindNext )
    Report( ed )
    print "== Find Next, for ""four"""
    ed->findWhat = "four"
    pad.Execute( ed, pad.CmdFindNext )

    print "== File > Save"
    pad.Execute( ed, pad.CmdSave )
    Report( ed )
```

```
    dim as ustring saved = fs.ReadAllText( "lf.tmp" ).Value( )
    print "  on disk: "; strings.Escape( saved )

    print "== type, then File > New, twice"
    Type_( ed, "?" )
    pad.Execute( ed, pad.CmdNew )
    Report( ed )
    pad.Execute( ed, pad.CmdNew )
    Report( ed )

    print "== moved to a monitor at twice the DPI"
    dim as HWND w = ed->form->hWindow
    dim as long dpi = GetDpiForWindow( w )
    dim as RECT r
    GetWindowRect( w, @r )
    SendMessageW( w, WM_DPICHANGED, MAKEWPARAM( 2 * dpi, 2 * dpi ), _
                  cast( LPARAM, @r ) )
    print "  the text is still monospaced: "; _
          cbool( PsTextBox_GetFont( ed->text ) = _
                 ed->form->Font( PSFONT_MONO ) )

    print "== type, then close the window"
    Type_( ed, "x" )
    ed->form->Close( )
    print "  window closed: "; cbool( IsWindow( w ) = 0 )
    return 0
end function
```

examples/ch36/tpad.out

```
== a new window
  title:  Untitled - tpad
  status: Ln 1, Col 1 | Windows (CRLF) | UTF-8
== the layout, at 600 x 400
  text meets the scroll bar: true
  text meets the status bar: true
  status bar is as wide as the window: true
== File > Open
  picked: lf.tmp
  title:  lf.tmp - tpad
  status: Ln 1, Col 1 | Unix (LF) | UTF-8
== Down, Down, End, and type "!"
  title:  *lf.tmp - tpad
  status: Ln 3, Col 7 | Unix (LF) | UTF-8
== Find Next, for "T", twice
```

```
  title:  *lf.tmp - tpad
  status: Ln 2, Col 2 | Unix (LF) | UTF-8
  title:  *lf.tmp - tpad
  status: Ln 3, Col 2 | Unix (LF) | UTF-8
== Find Next, for "four"
  asked: Cannot find "four"
== File > Save
  title:  lf.tmp - tpad
  status: Ln 3, Col 2 | Unix (LF) | UTF-8
  on disk: one\ntwo\nthree!\n
== type, then File > New, twice
  asked: Do you want to save changes to lf.tmp?
  answered: Cancel
  title:  *lf.tmp - tpad
  status: Ln 3, Col 2 | Unix (LF) | UTF-8
  asked: Do you want to save changes to lf.tmp?
  answered: Don't Save
  title:  Untitled - tpad
  status: Ln 1, Col 1 | Windows (CRLF) | UTF-8
== moved to a monitor at twice the DPI
  the text is still monospaced: true
== type, then close the window
  asked: Do you want to save changes to Untitled?
  answered: Don't Save
  window closed: true
```

The window is never shown. The test sizes it first, and that matters twice over. A key sent to a zero-sized text box is accepted and does nothing: in the first draft, which sent its keys before sizing the window, Down and Down left the caret on line 1. And the layout itself is checked by comparing edges, which is the same at any DPI, rather than pixel counts, which are not.

> **NOTE**
> `Report` copies the title from a `wstring` buffer into a `ustring` before printing it. That is not tidiness. Printed to a pipe or a file — and the harness reads a pipe — a `wstring` is written as its raw UTF-16 code units, a zero byte after every ASCII letter, while a `ustring` is written as UTF-8. On a console the two print the same, which is why the difference shows up only when the output is redirected.

What the self-test cannot see

The first version of `tpad` had no status-bar paint callback. Its self-test passed: `Report` reads each panel's text back with `PsStatusBar_GetText`, and the text was right. On screen the panels were empty, because nothing painted them. **A test that reads**

the model does not see pixels. The status bar was caught by looking at the window, and the screenshot at the top of this chapter is the check that it now paints. What the self-test proves is that the commands, the callbacks and the files behave; that the window *looks* right is still a matter of looking at it.

Running it

From the chapter's directory:

```
tiko build tpad
out/debug/tpad.exe notes.txt
```

`tiko build tpad` builds the target `tpad` with the `debug` profile into `out/debug`; `tiko run -- notes.txt` builds and starts it in one step. The first build compiles every PsControls module the program imports, and takes a few minutes. After that, a change to `tpad`'s own files is rebuilt in under half a minute.

Shipping

`tiko build tpad --profile release` builds an optimised `out/release/tpad.exe` of 316,416 bytes. It imports only libraries that come with Windows: `KERNEL32`, `USER32`, `GDI32`, `gdiplus`, `COMCTL32`, `ole32`, `ADVAPI32`, `VERSION`, `dwmapi` and the C runtime `ucrtbase`. There is no installer and no runtime to install. Copy the file, and it runs.

The debug build is ten times the size, 3,101,696 bytes, because it carries `-exx`'s checks and the debugger's information (Chapter 31). Ship the release build.

What it costs

- **Every caret move reads the whole text.** `ShowStatus` copies the text out of the control, converts its line breaks and counts lines up to the caret. Measured on a 1 MB file, in a release build: about 16 ms per update, of which 7 ms is `PsTextBox_GetText` alone. A notes file costs nothing; a 10 MB log would make the caret lag. The cheaper way asks the control for the line with `EM_EXLINEFROMCHAR` and `EM_LINEINDEX`, but those count lines as they are *displayed*, wrapped, and the status bar wants lines as they are in the file.
- **The text always wraps.** A multiline `PsTextBox` wraps long lines and has no horizontal scroll bar, so Notepad's Word Wrap switch has nothing to switch.
- **The whole file is in memory**, twice while it is saved: once in the control and once in the document being written.
- **Text that is not UTF-8 is not preserved.** `tpad` warns, and saving writes U+FFFD where the unreadable bytes were. Keeping them would mean reading the file in its own encoding, which needs the encoding to be known or guessed.

- **A self-test is code in the shipped program.** It is a few dozen lines that do nothing unless asked, and the price buys a window that is checked on every build.

Summary

- Keep the part of a program that needs no window in a module that imports none. `textfile` is tested on Linux and Windows by a program that prints; only `pad` needs Windows.
- Give text one line-break convention inside the program — one LF, counted as one character — and convert at the edges: `ToLf` on the way in from the file and from the control, the file's own ending on the way out.
- `fs.ReadAllText` removes a byte-order mark and replaces bytes that are not UTF-8 with U+FFFD. An editor that must write a file back the same way reads the mark itself. `fs.ReadAllBytes`'s third argument is a size limit, not a prefix length.
- Save with `fs.WriteAtomic`, so a failed save leaves the old file whole.
- `PsApplication` is the program and its message loop; a `PsForm` is a window that reports through procedure pointers. Keep per-window state in a `new`ed type that the form's `Tag` points at, and reach it from any control's callback with `PsForm.FromHwnd`.
- PsControls setters are silent: only a person's actions call back. Read the caret after a key by posting a message to yourself, and give a `PsStatusBar` a paint callback or its panels stay blank.
- Controls are created hidden and zero-sized. Place them in `OnSize`, scale every literal with `ScaleX`/`ScaleY`, and set your own fonts again in `OnDpiChanged`.
- Route menus, keys and tests to one `Execute`. Put the accelerator keys the text control does not handle itself in the form's table.
- Ask the person in one place, before anything is thrown away, and wire the close handler to it.
- Put the person behind procedure pointers, and a test can drive the real window: send the control the messages the keyboard would, pump the queue, and print what changed. Then look at the window as well.

Exercises

1. Add Find Previous (Shift+F3) to `textfile` and to the Edit menu: the last match that ends at or before the start of the selection, wrapping around to the bottom. {solution="ex36-01″}
2. Show a word count in a fourth status-bar panel. Write the counting function in `textfile`, so that it can be tested on both targets. {solution="ex36-02″}
3. Add Replace (Ctrl+H): a modal box with *Find what*, *Replace with*, Replace and Replace All. Do the replacing in `textfile`, so that it can be tested on both targets.

4. Add Time/Date (F5), which inserts the local date and time at the caret, with `std/chrono`.
5. Open a file dropped on the window from Explorer. `DragAcceptFiles` turns dropping on, the form receives `WM_DROPFILES` (`OnMessage`), and `DragQueryFileW` gives the path. Ask `MayDiscard` first.
6. Extend the self-test to Save As: point `pick` at a new name, and check the title and the file on disk afterwards.

Capstone: a Cross-Platform Notepad

The `tpad` of Chapter 36 runs on one operating system, because every control in it draws with GDI and every message it handles is a Windows message. What would it take to run the same editor on Linux and on a Mac, and is the answer a port or a rewrite?

This chapter finds out. It builds **`tpad`** again, on **tkControls**, the vendor package of controls that draw into SDL3 windows, and **one source builds and runs on Windows, Linux and macOS**. The menus, the Find box, the status bar and the question before unsaved work is thrown away are the same. So is the document module, which does not change by a byte.

The window module is new, and most of what is interesting here is the short list of things that only show up once a program leaves Windows: a file dialog that does not stop your code, a text control that counts in bytes, a command key that is not Ctrl, and a Quit that arrives from the system and never touches your window. It joins:

- the module split of Chapter 36, tested by reusing its result;
- procedure pointers and lambdas (Chapters 7 and 21), which is still how every control reports back;
- a C library's callback, and the thread it may arrive on (Chapters 23 and 30);
- text as units and as bytes (Chapter 9);
- conditional compilation by platform (Chapter 16);
- `tiko build`, and what a program built on shared libraries takes with it (Chapter 31).

Read Chapter 36 first. This chapter says what is different and why, and does not repeat what is the same.

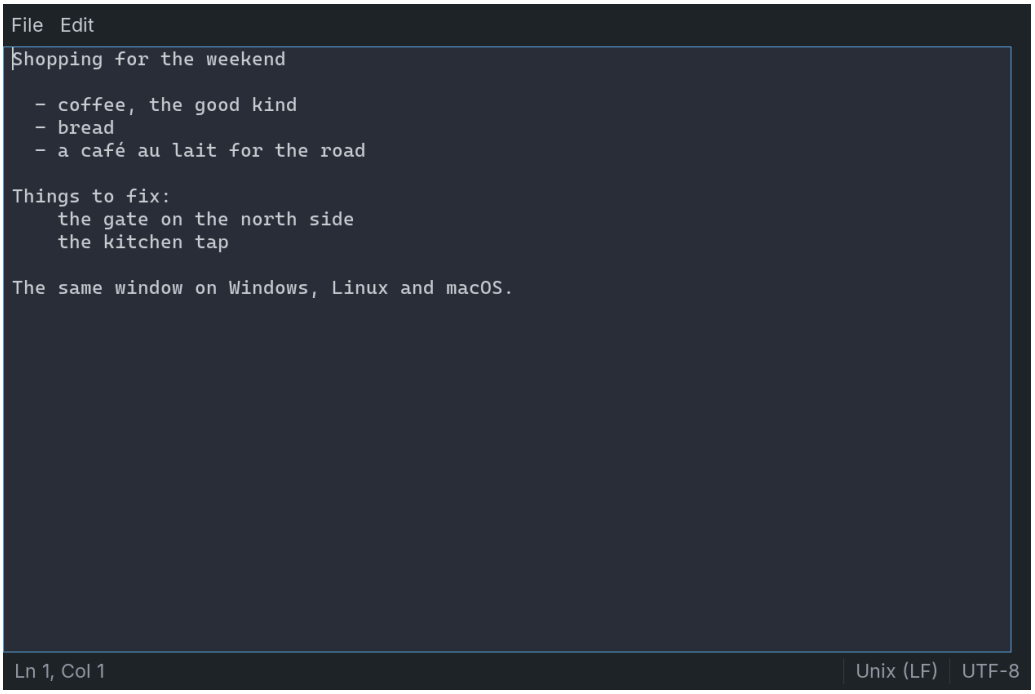


Figure 3: tpad on tkControls, a short file open, in the dark theme.

The specification

```
tpad [file]
tpad --selftest
tpad --shot out.bmp [file]
```

The specification of Chapter 36 stands, with these changes:

- **It runs on Windows, Linux and macOS**, from one source with no per-platform files.
- **The command key is Cmd on macOS** and Ctrl elsewhere, for `tpad`'s own keys and the text box's, and the menus say which.
- **Text is any UTF-8**: selection, Find and the line and column are right after accented letters and other characters that are not ASCII.
- **The question before discarding also covers the system's Quit**: Cmd+Q on a Mac, or a logout.
- **`--shot`** renders the window once and saves it as a picture, so that two systems can be compared.

The layout

```
ch37/
    tiko.toml               the package, and the target tpad
    textfile/textfile.tiko  the document: identical to Chapter 36's
    offsets/offsets.tiko    UTF-8 bytes and ustring units
    offsettest.tiko         offsets on its own, on every target
    pad/pad.tiko            the window: controls, menus, commands
    tpad.tiko               the program, and its self-test
```

examples/ch37/tiko.toml

```
# tpad on tkControls: `tiko build tpad` writes out/debug/tpad
[package]
name = "tpad"

[target.tpad]
main = "tpad.tiko"
```

That is the whole manifest. Chapter 36's needed a `UNICODE` define, a resource file and a manifest for DPI awareness; SDL3's strings are UTF-8 on every system, and SDL3 makes a window DPI-aware by itself.

The document

`textfile/textfile.tiko` is a copy of Chapter 36's, and `cmp` finds no difference between the two. That was the point of keeping the part that needs no window in a module that imports none: the module never knew which controls would show it, so it had nothing to change. Everything Chapter 36 said about it holds, including the rule that matters most here, that inside a `Document` a line break is one character, LF.

What a window is made of

examples/ch37/pad/pad.tiko

```
#module "pad"
#import "../textfile"
#import "../offsets"
#import "vendor/tkControls/tkApp"
#import "vendor/tkControls/tkMenuBar"
#import "vendor/tkControls/tkPopupMenu"
#import "vendor/tkControls/tkTextBox"
#import "vendor/tkControls/tkVScrollBar"
#import "vendor/tkControls/tkStatusBar"
#import "vendor/tkControls/tkMessageBox"
#import "vendor/tkControls/tkButton"
#import "vendor/tkControls/tkCheckBox"
```

```
using tkApp
using tkHost
using tkPaint
using tkTheme
using tkMenuBar
using tkPopupMenu
using tkTextBox
using tkVScrollBar
using tkStatusBar
using tkMessageBox
using tkButton
using tkCheckBox
```

One program-level idea replaces several. In Chapter 36 every control was a Windows window with a handle, and a form found its controls by their ids. A **tkControls window is one SDL window**, and everything in it is a `TK_WIDGET`: a record in a tree that the library draws and sends events to. There is no handle to look up, so the program keeps a pointer to each control it talks to:

`examples/ch37/pad/pad.tiko`

```
'' Every command the menus and the keyboard can give.
public enum Cmd
    CmdNew = 1000
    CmdOpen
    CmdSave
    CmdSaveAs
    CmdExit
    CmdUndo
    CmdCut
    CmdCopy
    CmdPaste
    CmdSelectAll
    CmdFind
    CmdFindNext
end enum

'' The two things a person answers: a question in a message box, and
'' a file name in a file dialog. Each is reached through a pointer, so
'' a test can answer in the person's place.
public type AskProc as function( byval f as TK_FORM ptr, _
    byref text as ustring, byval buttons as long ) as long
public type PickProc as function( byval f as TK_FORM ptr, _
    byval saving as boolean, byref suggest as ustring ) as ustring
```

```
'' One open file and its window. The form's Tag points here.
public type Editor
    as TK_FORM ptr form
    as TK_WIDGET ptr bar, text, scroll, status
    as textfile.Document doc
    as ustring findWhat
    as boolean matchCase
    as boolean picking          '' a file dialog is waiting

    as AskProc ask
    as PickProc pick
end type

'' The Find box's controls. Its form's Tag points here.
type FindBox
    as Editor ptr ed
    as TK_WIDGET ptr what, matchCase, ok, cancel
end type

public declare sub Execute( byval ed as Editor ptr, byval id as long )
public declare sub ShowTitle( byval ed as Editor ptr )
public declare sub ShowStatus( byval ed as Editor ptr )

'' A control's callback receives only the control: its form's Tag is
'' the Editor.
function EditorOf( byval w as TK_WIDGET ptr ) as Editor ptr
    return tkForm_FromWidget( w )->Tag
end function
```

The shape is Chapter 36's. An `Editor` holds everything about the window, the form's `Tag` points at it, and a control's callback gets from the control to the `Editor` through `EditorOf`: `tkForm_FromWidget` finds the form, and the form's `Tag` finds the `Editor`. `Cmd` is unchanged. The new field is `picking`, which a later section needs.

The message box is the same call with a different prefix:

`examples/ch37/pad/pad.tiko`

```
'' The real answers come from a person.
function AskPerson( byval f as TK_FORM ptr, byref text as ustring, _
                    byval buttons as long ) as long
    return tkMessageBox_Show( tkForm_Widget( f ), text, "tpad", _
        iif( buttons = MBX_BTN_OK, MBX_ICON_INFO, MBX_ICON_QUESTION ), _
        buttons )
end function
```

A dialog that does not stop your code

The file dialog is where the two systems' habits part. On Windows, `CIOpenFileDialog` does not return until the person has chosen. SDL3's dialogs **return at once and answer later**, through a callback, because on some systems a dialog cannot be run any other way. Worse for a program that wants to read an answer, the callback **may run on another thread**.

`tpad`'s commands want a function that returns a path, so `PickFile` builds one:

`examples/ch37/pad/pad.tiko`

```
'' What the dialog's callback leaves for the code waiting on it.
type PickWait
    as SDL_AtomicInt answered
    as zstring ptr path         '' SDL's copy of the answer, or null
end type

'' May run on another thread, so it touches nothing but C memory.
sub PickAnswered( byval userdata as any ptr, _
                  byval filelist as const zstring const ptr ptr, _
                  byval filter as long )
    dim as PickWait ptr w = userdata
    if filelist <> 0 then
        if filelist[0] <> 0 then w->path = SDL_strdup( filelist[0] )
    end if
    SDL_SetAtomicInt( @w->answered, 1 )
end sub

'' Shows the system's file dialog and returns the path, or "".
function PickFile( byval f as TK_FORM ptr, byval saving as boolean, _
                   byref suggest as ustring ) as ustring
    dim as SDL_DialogFileFilter filters( 0 to 1 )
    filters( 0 ).name = @"Text files" : filters( 0 ).pattern = @"txt"
    filters( 1 ).name = @"All files" : filters( 1 ).pattern = @"*"
    dim as string where = suggest
    dim as PickWait w
    dim as SDL_Window ptr win = tkForm_Host( f )->win
    if saving then
        SDL_ShowSaveFileDialog( @PickAnswered, @w, win, @filters( 0 ), _
            2, iif( len( where ), strptr( where ), 0 ) )
    else
        SDL_ShowOpenFileDialog( @PickAnswered, @w, win, @filters( 0 ), _
            2, 0, false )
    end if
    do until SDL_GetAtomicInt( @w.answered ) = 1
```

```
        tkApp_Pump( )
        SDL_Delay( 10 )
    loop
    dim as ustring picked
    if w.path <> 0 then
        picked = *w.path
        SDL_free( w.path )
    end if
    return picked
end function

'' Every question for a path comes through here. The event loop runs
'' while the dialog is up, so nothing else may start until it answers.
function PickPath( byval ed as Editor ptr, byval saving as boolean, _
                   byref suggest as ustring ) as ustring
    ed->picking = true
    dim as ustring p = ed->pick( ed->form, saving, suggest )
    ed->picking = false
    return p
end function
```

It starts the dialog, then turns the event loop by hand until the answer arrives, so the window keeps painting and SDL keeps delivering events.

Everything the callback does is chosen for the thread it may be on. It is handed `@w`, so it needs no global. It copies the path with `SDL_strdup`, which is C memory and owes nothing to tiko's string runtime, and it signals with an atomic integer (Chapter 23). Only back on the waiting thread does the copy become a `ustring`. The order matters: `path` is written first, and the atomic's write is what tells the waiting thread that it is safe to read.

If the system has no dialog to show, SDL calls back with no list at all, and `PickFile` returns `""`, as if the person had cancelled. On Linux the dialog comes from the desktop's portal or from `zenity`, so a stripped-down machine can reach that path.

> **PITFALL**
> A dialog that returns at once leaves the event loop running, and a running event loop will deliver anything. If the window is not blocked for you, the person can choose File > Exit while the dialog is open. `MayDiscard` finds nothing unsaved, the form is destroyed, its `OnDestroy` deletes the `Editor`, and when the dialog does answer, the code waiting for it carries on with a pointer to memory that has been freed.

Whether a system blocks the parent window is the system's decision, so `PickPath` does not depend on it. It sets `picking`; `Execute` starts nothing while it is set, and the form's close handler refuses. The self-test tries it, with a scripted person who closes the window from inside the dialog:

```
== File > Exit while the file dialog is open
  window closed: false
```

With the close handler's one line removed, that line reads `true`.

The text, and the callback that runs afterwards

examples/ch37/pad/pad.tiko

```
'' Queued from the message callback: by the time it runs, the text box
'' has acted on the key or the click and the caret has moved.
sub CaretMoved( byval payload as any ptr )
    ShowStatus( payload )
end sub

sub BuildText( byval f as TK_FORM ptr, byval ed as Editor ptr )
    ed->text = tkTextBox_Create( tkForm_Widget( f ), 101, true )
    tkTextBox_SetFont( ed->text, tkForm_Font( f, TKFONT_MONO ) )
    tkTextBox_SetLimitText( ed->text, &h7FFFFFFE )
    tkTextBox_SetMargins( ed->text, tkForm_Scale( f, 6 ), _
                          tkForm_Scale( f, 6 ) )

    '' The person typed, cut, pasted or undid something.
    tkTextBox_SetChangeCallback( ed->text, _
        sub( byval h as TK_WIDGET ptr )
            ShowTitle( EditorOf( h ) )
            ShowStatus( EditorOf( h ) )
        end sub )

    '' The caret may be about to move: look once the control has acted.
    tkTextBox_SetMessageCallback( ed->text, _
        function( byval m as TKTEXTBOX_MESSAGEINFO ptr ) as boolean
            select case m->msg
            case TKM_KEYDOWN, TKM_KEYUP, TKM_LBUTTONDOWN, TKM_LBUTTONUP
                tkApp_RunOnUI( @CaretMoved, EditorOf( m->hTextBox ) )
            end select
            return false
        end function )
```

```
    '' The box and its scroll bar report to each other.
    ed->scroll = tkVScrollBar_Create( tkForm_Widget( f ), 102 )
    tkTextBox_SetScrollChangedCallback( ed->text, _
        sub( byval h as TK_WIDGET ptr )
            dim as integer total, page, top
            tkTextBox_GetVScrollInfo( h, total, page, top )
            tkVScrollBar_SetRange( EditorOf( h )->scroll, total, _
                                   page, top )
        end sub )
    tkVScrollBar_SetScrollCallback( ed->scroll, _
        sub( byval h as TK_WIDGET ptr, byval newPos as integer )
            tkTextBox_ScrollToLine( EditorOf( h )->text, newPos )
        end sub )
end sub
```

The third argument makes a multiline box, as it did in Chapter 36, and `tkForm_Font( f, TKFONT_MONO )` is the form's monospaced font, which the package ships.

The four callbacks are the four from Chapter 36, and the change callback still fires only for a person's edits. The one with a different mechanism is the message callback. It runs *before* the box acts on a key, so the caret has not moved yet. Chapter 36 posted itself a Windows message; **`tkApp_RunOnUI` is the same idea without Windows**. It queues `CaretMoved` to run when the current event has been handled, by which time the caret is where the key put it.

The status bar

examples/ch37/pad/pad.tiko

```
sub BuildStatus( byval f as TK_FORM ptr, byval ed as Editor ptr )
    ed->status = tkStatusBar_Create( tkForm_Widget( f ), 103 )
    tkStatusBar_AddPanel( ed->status, "" )      '' Ln, Col: springs
    tkStatusBar_AddPanel( ed->status, "" )      '' line ending
    tkStatusBar_AddPanel( ed->status, "" )      '' encoding
    tkStatusBar_SetExpandPanel( ed->status, 0 )
end sub
```

There is less here than in Chapter 36, because a `tkStatusBar` draws its own panels. Setting a panel's text is all there is to do, and its Pitfall about blank panels has no counterpart.

Two ways of counting

examples/ch37/pad/pad.tiko

```
'' The editor's text, with one LF per line break, as textfile wants.
function TextOf( byval ed as Editor ptr ) as ustring
    return textfile.ToLf( tkTextBox_GetText( ed->text ) )
end function

public sub ShowTitle( byval ed as Editor ptr )
    dim as string title = textfile.Caption( ed->doc, _
        tkTextBox_GetModify( ed->text ) )
    SDL_SetWindowTitle( tkForm_Host( ed->form )->win, strptr( title ) )
end sub

public sub ShowStatus( byval ed as Editor ptr )
    dim as integer first, last
    tkTextBox_GetSel( ed->text, first, last )
    dim as ustring body = TextOf( ed )
    dim as int32 ln, col
    textfile.LineColumn( body, offsets.UnitsIn( body, last ), ln, col )
    tkStatusBar_SetText( ed->status, 0, "Ln " & ln & ", Col " & col )
    tkStatusBar_SetText( ed->status, 1, _
        iif( ed->doc.ending = textfile.LineEnding.CrLf, _
             "Windows (CRLF)", "Unix (LF)" ) )
    tkStatusBar_SetText( ed->status, 2, _
        iif( ed->doc.hasBom, "UTF-8 with BOM", "UTF-8" ) )
end sub
```

`ShowTitle` hands SDL the caption as UTF-8: `title` is a `string`, and assigning a `ustring` to it stores the text's UTF-8 bytes. `ShowStatus` is Chapter 36's with one call added, and that call is the subject of this section.

A `tkTextBox` holds its text in UTF-8 and **its offsets count bytes**. A `ustring` counts **UTF-16 units** (Chapter 9), and so do `len`, `instr` and everything in `textfile`. While every character before the offset is ASCII, the two numbers are equal. The first character that is not, and they are not:

`examples/ch37/offsets/offsets.tiko`

```
#module "offsets"
'' Two ways of counting the same position in a text. A ustring counts
'' UTF-16 units; a tkTextBox counts UTF-8 bytes. They agree only while
'' everything before the position is ASCII.
''
'' A string assigned from a ustring holds its UTF-8 bytes, and a ustring
'' assigned from a string decodes them, so each conversion is a prefix
'' turned into the other form and measured.
```

```
'' How many ustring units the first `count` UTF-8 bytes of text make.
public function UnitsIn( byref text as const ustring, _
                         byval count as int32 ) as int32
    dim as string utf8 = text
    dim as ustring head = left( utf8, count )
    return len( head )
end function

'' How many UTF-8 bytes the first `count` ustring units of text make.
public function BytesIn( byref text as const ustring, _
                         byval count as int32 ) as int32
    dim as string head = left( text, count )
    return len( head )
end function
```

The two functions do the conversion by letting tiko do it: take the prefix in one form, turn it into the other, measure. They need no window, so they get a module and a test of their own:

examples/ch37/offsettest.tiko

```
'' The two counts of a position, with no window. U+00E9 is two bytes
'' and one unit; U+1F600 is four bytes and two units.
#import "./offsets"

dim as ustring ascii = "plain text"
dim as ustring accent = "caf" & wchr( &hE9 ) & " au lait"
dim as ustring emoji = wchr( &h1F600 ) & " smile"

print "== ASCII: the two counts agree"
print "byte 5 is unit"; offsets.UnitsIn( ascii, 5 )
print "unit 5 is byte"; offsets.BytesIn( ascii, 5 )

print "== 'au' in cafe-acute au lait"
dim as int32 at = instr( accent, "au" ) - 1
print "unit offset:"; at
print "byte offset:"; offsets.BytesIn( accent, at )
print "and back:   "; offsets.UnitsIn( accent, _
                          offsets.BytesIn( accent, at ) )

print "== 'smile' after U+1F600"
at = instr( emoji, "smile" ) - 1
print "unit offset:"; at
print "byte offset:"; offsets.BytesIn( emoji, at )
print "and back:   "; offsets.UnitsIn( emoji, _
```

```
                          offsets.BytesIn( emoji, at ) )

print "== the edges"
print "offset 0:   "; offsets.BytesIn( accent, 0 ); _
      offsets.UnitsIn( accent, 0 )
dim as int32 units = len( accent )
dim as int32 count = offsets.BytesIn( accent, units )
print "the end:    "; count; offsets.UnitsIn( accent, count )
```

examples/ch37/offsettest.out

```
== ASCII: the two counts agree
byte 5 is unit 5
unit 5 is byte 5
== 'au' in cafe-acute au lait
unit offset: 5
byte offset: 6
and back:    5
== 'smile' after U+1F600
unit offset: 3
byte offset: 5
and back:    3
== the edges
offset 0:    0 0
the end:     13 12
```

An e-acute is one unit and two bytes. U+1F600 is two units and four bytes. Both directions land on a boundary between characters, and the box snaps an offset to one in any case.

PITFALL

Nothing about this mistake is visible with ASCII text, and every test written in English passes. Take out the three conversions in `pad`, and open a file whose first line is `café au lait` and whose second is `naïve lait`. Find Next for `lait` selects:

```
== text that is not ASCII, Find Next for "lait", twice
  selected:  lai
  status: Ln 1, Col 13 | Unix (LF) | UTF-8
  selected: e la
  status: Ln 2, Col 11 | Unix (LF) | UTF-8
```

The search worked and the selection is wrong, by one byte for each character before it that took two. The status bar is right, for the wrong reason: the position

it measured was the one the selection had been given. A file that is only slightly non-English looks fine until someone uses Find.

This is the same mistake as Chapter 36's Pitfall about line breaks, in another unit. Each control counts in its own way, so give the program one way, and convert where the two meet.

Layout, menus, keys

examples/ch37/pad/pad.tiko

```
sub Main_OnSize( byval f as TK_FORM ptr, byval cx as long, _
                 byval cy as long )
    dim as Editor ptr ed = f->Tag
    dim as long barH = tkForm_Scale( f, 30 )
    dim as long statusH = tkForm_Scale( f, 26 )
    dim as long sbW = tkForm_Scale( f, 14 )
    dim as long textH = cy - barH - statusH
    tkWidget_SetPos( ed->bar, 0, 0, cx, barH )
    tkWidget_SetPos( ed->text, 0, barH, cx - sbW, textH )
    tkWidget_SetPos( ed->scroll, cx - sbW, barH, sbW, textH )
    tkWidget_SetPos( ed->status, 0, cy - statusH, cx, statusH )
end sub
```

Controls start with no size and `OnSize` places them. `tkForm_Scale` takes the place of `ScaleX` and `ScaleY`: an SDL window has one scale.

examples/ch37/pad/pad.tiko

```
function AddMenu( byval f as TK_FORM ptr, byval bar as TK_WIDGET ptr, _
                  byref title as ustring ) as TK_MENU ptr
    dim as long i = tkMenuBar_AddItem( bar, title, 0 )
    dim as TK_MENU ptr m = tkPopupMenu_Create( tkForm_Host( f ) )
    tkMenuBar_SetDropDown( bar, i, m )
    return m
end function

'' The command key, as the menus spell it: Cmd on macOS, Ctrl
'' elsewhere. TK_KMOD_PRIMARY is the same key as a modifier mask.
#if defined( TIKO_OS_MACOS )
const PrimaryKey = "Cmd+"
#else
const PrimaryKey = "Ctrl+"
#endif

function FormKey( byval f as TK_FORM ptr, byval key as long, _
```

```
                    byval mods as long ) as boolean
    dim as boolean primary = (mods and TK_KMOD_PRIMARY) <> 0
    dim as boolean shift = (mods and SDL_KMOD_SHIFT) <> 0
    dim as long id = 0
    if primary then
        select case key
        case SDLK_N : id = CmdNew
        case SDLK_O : id = CmdOpen
        case SDLK_S : id = iif( shift, CmdSaveAs, CmdSave )
        case SDLK_F : id = CmdFind
        end select
    elseif key = SDLK_F3 then
        id = CmdFindNext
    end if
    if id = 0 then return false
    Execute( f->Tag, id )
    return true
end function

sub BuildMenus( byval f as TK_FORM ptr, byval ed as Editor ptr )
    dim as TK_WIDGET ptr bar = _
        tkMenuBar_Create( tkForm_Widget( f ), 100 )
    ed->bar = bar
    tkMenuBar_SetCommandCallback( bar, _
        sub( byval h as TK_WIDGET ptr, byval id as long )
            Execute( EditorOf( h ), id )
        end sub )
    dim as TK_MENU ptr m = AddMenu( f, bar, "File" )
    dim as ustring k = PrimaryKey
    tkPopupMenu_AddItem( m, CmdNew, "New", k & "N" )
    tkPopupMenu_AddItem( m, CmdOpen, "Open...", k & "O" )
    tkPopupMenu_AddItem( m, CmdSave, "Save", k & "S" )
    tkPopupMenu_AddItem( m, CmdSaveAs, "Save As...", k & "Shift+S" )
    tkPopupMenu_AddSeparator( m )
    tkPopupMenu_AddItem( m, CmdExit, "Exit" )
    m = AddMenu( f, bar, "Edit" )
    tkPopupMenu_AddItem( m, CmdUndo, "Undo", k & "Z" )
    tkPopupMenu_AddSeparator( m )
    tkPopupMenu_AddItem( m, CmdCut, "Cut", k & "X" )
    tkPopupMenu_AddItem( m, CmdCopy, "Copy", k & "C" )
    tkPopupMenu_AddItem( m, CmdPaste, "Paste", k & "V" )
    tkPopupMenu_AddItem( m, CmdSelectAll, "Select All", k & "A" )
    tkPopupMenu_AddSeparator( m )
    tkPopupMenu_AddItem( m, CmdFind, "Find...", k & "F" )
```

```
    tkPopupMenu_AddItem( m, CmdFindNext, "Find Next", "F3" )
    tkForm_OnKeyDown( f, @FormKey )
end sub
```

A `tkMenuBar` holds the titles and each drops down a `tkPopupMenu`, as in Chapter 36. There is no accelerator table. The text box handles the primary key with Z, X, C, V and A itself and **leaves every other key unclaimed**, and an unclaimed key reaches the form's `OnKeyDown`. Returning `true` says that it was used.

The primary key is the first thing in this chapter that is written differently for one system. `TK_KMOD_PRIMARY`, from `tkHost`, is the Cmd key on macOS and Ctrl elsewhere, and the text box tests it too, so `tpad` writes that name once. Only the menu's *label* needs the platform spelled out, so that is the only `#if` in the program (Chapter 16).

Commands

examples/ch37/pad/pad.tiko

```
'' The one place a command is carried out, wherever it came from.
public sub Execute( byval ed as Editor ptr, byval id as long )
    if ed->picking then exit sub
    select case id
    case CmdNew
        if MayDiscard( ed ) then
            dim as textfile.Document blank
            ShowDocument( ed, blank )
        end if
    case CmdOpen
        if MayDiscard( ed ) then
            dim as ustring p = PickPath( ed, false, "" )
            if len( p ) then OpenPath( ed, p )
        end if
    case CmdSave
        SaveDocument( ed, false )
    case CmdSaveAs
        SaveDocument( ed, true )
    case CmdExit
        tkForm_Close( ed->form )
    case CmdUndo
        tkTextBox_Undo( ed->text )
    case CmdCut
        tkTextBox_Cut( ed->text )
    case CmdCopy
        tkTextBox_Copy( ed->text )
    case CmdPaste
        tkTextBox_Paste( ed->text )
```

```
    case CmdSelectAll
        tkTextBox_SelectAll( ed->text )
    case CmdFind
        AskWhatToFind( ed )
    case CmdFindNext
        if len( ed->findWhat ) then
            FindAgain( ed )
        else
            AskWhatToFind( ed )
        end if
    end select
end sub
```

`Execute` is Chapter 36's, with its first line new: while a file dialog is waiting, no command starts.

Files, and Find

`examples/ch37/pad/pad.tiko`

```
'' Puts doc in the window, replacing what was there.
sub ShowDocument( byval ed as Editor ptr, _
                  byref doc as textfile.Document )
    ed->doc = doc
    tkTextBox_SetText( ed->text, doc.text )
    tkTextBox_SetModify( ed->text, false )
    tkTextBox_SetSel( ed->text, 0, 0 )
    ShowTitle( ed )
    ShowStatus( ed )
end sub

'' Opens path; says why not if it cannot.
public function OpenPath( byval ed as Editor ptr, _
                          byref path as ustring ) as boolean
    dim as textfile.Document doc
    dim as ustring e = textfile.Load( path, doc )
    if len( e ) then
        ed->ask( ed->form, e, MBX_BTN_OK )
        return false
    end if
    ShowDocument( ed, doc )
    if doc.lossy then
        ed->ask( ed->form, textfile.DisplayName( doc ) & _
            " is not UTF-8. Characters that could not be read show" & _
            " as U+FFFD, and saving writes them that way.", MBX_BTN_OK )
```

```
    end if
    return true
end function

'' Saves, asking for a name when there is none (or when asked to).
function SaveDocument( byval ed as Editor ptr, _
                       byval newName as boolean ) as boolean
    if newName orelse len( ed->doc.path ) = 0 then
        dim as ustring p = PickPath( ed, true, _
                                     textfile.DisplayName( ed->doc ) )
        if len( p ) = 0 then return false
        ed->doc.path = p
    end if
    ed->doc.text = TextOf( ed )
    dim as ustring e = textfile.Save( ed->doc )
    if len( e ) then
        ed->ask( ed->form, e, MBX_BTN_OK )
        return false
    end if
    tkTextBox_SetModify( ed->text, false )
    ShowTitle( ed )
    return true
end function

'' True when it is all right to throw the text away: it is saved,
'' the person saved it now, or said not to.
function MayDiscard( byval ed as Editor ptr ) as boolean
    if tkTextBox_GetModify( ed->text ) = false then return true
    select case ed->ask( ed->form, "Do you want to save changes to " & _
                         textfile.DisplayName( ed->doc ) & "?", _
                         MBX_BTN_SAVE_DONTSAVE_CANCEL )
    case MBX_IDYES
        return SaveDocument( ed, false )
    case MBX_IDNO
        return true
    end select
    return false
end function
```

Apart from where the path comes from, this is Chapter 36's code with the prefixes changed. `tkTextBox_SetText` is silent, as a setter should be, and the modified flag is cleared after a load and a successful save.

`examples/ch37/pad/pad.tiko`

```
sub FindAgain( byval ed as Editor ptr )
    if len( ed->findWhat ) = 0 then exit sub
    dim as integer first, last
    tkTextBox_GetSel( ed->text, first, last )
    dim as ustring body = TextOf( ed )
    dim as int32 at = textfile.FindNext( body, ed->findWhat, _
        offsets.UnitsIn( body, last ), ed->matchCase )
    if at < 0 then
        ed->ask( ed->form, "Cannot find """ & ed->findWhat & """", _
                 MBX_BTN_OK )
        exit sub
    end if
    '' SetSel scrolls the caret into view itself.
    tkTextBox_SetSel( ed->text, offsets.BytesIn( body, at ), _
        offsets.BytesIn( body, at + len( ed->findWhat ) ) )
    ShowStatus( ed )
end sub

'' The Find box: a modal form that only collects the search.
sub FindBox_OnSize( byval f as TK_FORM ptr, byval cx as long, _
                    byval cy as long )
    dim as FindBox ptr box = f->Tag
    dim as long pad = tkForm_Scale( f, 12 ), bw, bh
    tkButton_GetIdealSize( box->ok, bw, bh )
    dim as long w = cx - 3 * pad - bw
    tkWidget_SetPos( box->what, pad, pad, w, bh )
    tkWidget_SetPos( box->matchCase, pad, 2 * pad + bh, w, bh )
    tkWidget_SetPos( box->ok, cx - pad - bw, pad, bw, bh )
    tkWidget_SetPos( box->cancel, cx - pad - bw, 2 * pad + bh, bw, bh )
end sub

sub FindBox_Accept( byval box as FindBox ptr )
    box->ed->findWhat = tkTextBox_GetText( box->what )
    box->ed->matchCase = tkCheckBox_GetChecked( box->matchCase )
    tkForm_EndModal( tkForm_FromWidget( box->what ), TKAPP_IDOK )
end sub

sub AskWhatToFind( byval ed as Editor ptr )
    dim as TK_FORM ptr f = tkApp_NewForm( "Find", 420, 112, ed->form )
    dim as FindBox box
    box.ed = ed
    f->Tag = @box
    box.what = tkTextBox_Create( tkForm_Widget( f ), 104 )
    tkTextBox_SetText( box.what, ed->findWhat )
```

```
    tkTextBox_SetSelectOnFocus( box.what, true )
    tkTextBox_SetEnterPressedCallback( box.what, _
        sub( byval h as TK_WIDGET ptr )
            FindBox_Accept( tkForm_FromWidget( h )->Tag )
        end sub )
    box.matchCase = tkCheckBox_Create( tkForm_Widget( f ), 105 )
    tkCheckBox_SetText( box.matchCase, "Match case" )
    tkCheckBox_SetChecked( box.matchCase, ed->matchCase )
    box.ok = tkButton_Create( tkForm_Widget( f ), TKAPP_IDOK )
    tkButton_SetText( box.ok, "Find Next" )
    tkButton_SetClickCallback( box.ok, _
        sub( byval h as TK_WIDGET ptr, byval id as long )
            FindBox_Accept( tkForm_FromWidget( h )->Tag )
        end sub )
    box.cancel = tkButton_Create( tkForm_Widget( f ), TKAPP_IDCANCEL )
    tkButton_SetText( box.cancel, "Cancel" )
    tkButton_SetClickCallback( box.cancel, _
        sub( byval h as TK_WIDGET ptr, byval id as long )
            tkForm_EndModal( tkForm_FromWidget( h ), TKAPP_IDCANCEL )
        end sub )
    tkForm_OnSize( f, @FindBox_OnSize )
    '' Esc ends a modal form with IDCANCEL by itself.
    if tkForm_ShowModal( f ) = TKAPP_IDOK then FindAgain( ed )
end sub
```

`FindAgain` converts twice: the selection's end into units for `textfile`, and the match back into bytes for the box. `tkTextBox_SetSel` scrolls the caret into view itself, so there is nothing after it. Enter in the search box is Find Next, through the box's Enter callback, and Esc ends a modal form with `TKAPP_IDCANCEL` without being asked. Because `tkForm_ShowModal` returns only once the box is closed, its `FindBox` can live on the stack of `AskWhatToFind`.

A Quit that is not a close

The close button goes through `OnClose`. The system's own Quit does not. SDL gives every Mac application a menu with **Quit** on it, and Cmd+Q reaches the program as `SDL_EVENT_QUIT`, an event that belongs to no window. A logout can arrive the same way. tkControls responds by marking every window closed, and the program ends with the question never asked.

`examples/ch37/pad/pad.tiko`

```
'' The event hook is handed only the event, so the one window it
'' protects is kept where the hook can reach it.
```

```
function MainEditor( byval ed as Editor ptr = 0 ) as Editor ptr
    static as Editor ptr kept
    if ed <> 0 then kept = ed
    return kept
end function

'' The system asking the whole program to quit (Cmd+Q, logging out)
'' becomes an ordinary close, which asks first.
function QuitHook( byval e as SDL_Event ptr ) as boolean
    if e->type <> SDL_EVENT_QUIT then return false
    tkForm_Close( MainEditor( )->form )
    return true
end function
```

`tkHost_AddEventHook` is called for every event before the library looks at it, and a hook that returns `true` has dealt with it. `QuitHook` turns the system's Quit into an ordinary close of the main window, which runs `OnClose`, which asks. An event hook is handed the event and nothing else, so `MainEditor` keeps a pointer in a `static` for it. It is the one piece of state in the program that is not in the `Editor`, and it is there because the C-shaped callback left no place for it.

PITFALL
The window's close button is an event for the window, and it is the one you remember to handle. With the hook removed, the self-test's Quit closes the window with unsaved text in it and no question: the two lines `asked:` and `answered:` vanish, and `window closed:` reads `true`. On Windows nothing a person ordinarily does produces this event, which is why a program developed there does not find out.

Creating it all

`examples/ch37/pad/pad.tiko`

```
'' Builds the main window, with an empty, untitled document.
public function Create( byval forcedScale as single = 0 ) as Editor ptr
    dim as Editor ptr ed = new Editor
    ed->ask = @AskPerson
    ed->pick = @PickFile
    dim as TK_FORM ptr f = tkApp_NewForm( "tpad", 720, 480, 0, _
                                          forcedScale )
    ed->form = f
    f->Tag = ed
```

```
    BuildMenus( f, ed )
    BuildText( f, ed )
    BuildStatus( f, ed )

    tkForm_OnSize( f, @Main_OnSize )
    '' Asked before the window closes, however it is closed.
    tkForm_OnClose( f, _
        function( byval f as TK_FORM ptr ) as boolean
            dim as Editor ptr ed = f->Tag
            if ed->picking then return false
            return MayDiscard( ed )
        end function )
    tkForm_OnDestroy( f, _
        sub( byval f as TK_FORM ptr )
            tkHost_RemoveEventHook( @QuitHook )
            delete cast( Editor ptr, f->Tag )
            f->Tag = 0
        end sub )
    MainEditor( ed )
    tkHost_AddEventHook( @QuitHook )

    ShowTitle( ed )
    ShowStatus( ed )
    return ed
end function
```

Chapter 36's `Create` needed a DPI handler, because PsApp gives every control the form's UI font when the scale changes. Here a scale change re-sizes the fonts each control was given and hands out no others, so there is nothing to put back. The self-test still checks it, because the fact is the library's and could change. `forcedScale` pins a scale instead of the display's.

The program

`examples/ch37/tpad.tiko`

```
#cmdline "-s gui"
#import "std/fs"
#import "std/strings"
#import "./pad"
#import "vendor/tkControls/tkApp"
#import "vendor/tkControls/tkTextBox"
#import "vendor/tkControls/tkStatusBar"
#import "vendor/tkControls/tkMessageBox"
```

```
using tkApp
using tkHost
using tkTextBox
using tkStatusBar
using tkMessageBox

'' tpad [file]                 run it
'' tpad --selftest             drive the real window, print what it saw
'' tpad --shot out.bmp [file]  render once, dark theme, and save it
```

examples/ch37/tpad.tiko

```
dim as string first = command( 1 )
dim as boolean testing = (first = "--selftest")
dim as boolean shooting = (first = "--shot")

dim as TKAPP_OPTIONS opts
if testing orelse shooting then opts.ThemeMode = TKAPP_THEME_DARK
if tkApp_Init( opts ) = false then
    print "tkApp_Init: " & *SDL_GetError( )
    end 1
end if

'' 0 means the display's own scale. A test pins it, so that its layout
'' is the same on every machine.
dim as single uiScale = val( environ( "TPAD_SCALE" ) )
if testing then uiScale = 1.0
if shooting then uiScale = 1.75
dim as pad.Editor ptr ed = pad.Create( uiScale )

if testing then
    dim as long result = SelfTest( ed )
    tkApp_Shutdown( )
    end result
end if
if shooting then
    if len( command( 3 ) ) then pad.OpenPath( ed, command( 3 ) )
    TakeShot( ed, command( 2 ) )
    tkApp_Shutdown( )
    end 0
end if

if len( first ) then pad.OpenPath( ed, first )
end tkApp_Run( ed->form )
```

`#cmdline "-s gui"` matters on Windows, where it means no console window; the other two systems make no such distinction. `tkApp_Run` shows the form and runs the event loop until it closes, and its result is the exit code. Under `--selftest` and `--shot` the loop never runs.

Testing on three systems

The test is Chapter 36's idea, and it is more valuable here. It drives the real window, with the person behind `ask` and `pick`:

examples/ch37/tpad.tiko

```
'' The person, scripted: what the test answers to each question.
function ScriptedAsk( byval f as TK_FORM ptr, byref text as ustring, _
                      byval buttons as long ) as long
    static as long replies( ... ) = { MBX_IDCANCEL, MBX_IDNO, _
                                      MBX_IDCANCEL, MBX_IDNO }
    static as int32 n
    print "  asked: "; text
    if buttons = MBX_BTN_OK then return MBX_IDOK
    n += 1
    print "  answered: "; iif( replies( n - 1 ) = MBX_IDNO, _
                               "Don't Save", "Cancel" )
    return replies( n - 1 )
end function

function ScriptedPick( byval f as TK_FORM ptr, _
                       byval saving as boolean, _
                       byref suggest as ustring ) as ustring
    print "  picked: lf.tmp"
    return "lf.tmp"
end function

'' A person who tries to close the window while the dialog is up.
function ImpatientPick( byval f as TK_FORM ptr, _
                        byval saving as boolean, _
                        byref suggest as ustring ) as ustring
    pad.Execute( f->Tag, pad.CmdExit )
    tkForm_Close( f )
    print "  window closed: "; cbool( tkForm_Host( f )->bQuit )
    return ""
end function
```

examples/ch37/tpad.tiko

```
'' Lets queued events arrive, as the loop would.
sub Pump( )
    tkApp_Pump( )
end sub

'' Keeps the loop turning for a while.
sub PumpFor( byval ms as uint64 )
    dim as uint64 start = SDL_GetTicks( )
    do until SDL_GetTicks( ) - start >= ms
        Pump( )
        SDL_Delay( 10 )
    loop
end sub

'' A key, delivered where the keyboard would deliver it.
sub Press( byval ed as pad.Editor ptr, byval key as long )
    dim as SDL_Event e
    e.type = SDL_EVENT_KEY_DOWN
    e.key.windowID = tkForm_Host( ed->form )->winId
    e.key.key = key
    e.key.down = true
    SDL_PushEvent( @e )
    Pump( )
end sub

'' Typed text, as the system hands it over after the keys: UTF-8. The
'' event points at this string, so it has to outlive the event.
sub TypeText( byval ed as pad.Editor ptr, byref s as ustring )
    static as string typed
    typed = s
    dim as SDL_Event e
    e.type = SDL_EVENT_TEXT_INPUT
    e.text.windowID = tkForm_Host( ed->form )->winId
    e.text.text = strptr( typed )
    SDL_PushEvent( @e )
    Pump( )
end sub

sub Report( byval ed as pad.Editor ptr )
    dim as ustring title = _
        *SDL_GetWindowTitle( tkForm_Host( ed->form )->win )
    print "  title:  "; title
    print "  status: "; tkStatusBar_GetText( ed->status, 0 ); " | "; _
          tkStatusBar_GetText( ed->status, 1 ); " | "; _
```

```
          tkStatusBar_GetText( ed->status, 2 )
end sub
```

Where Chapter 36 sent `WM_KEYDOWN` and `WM_CHAR` to a window, this one pushes SDL events onto the queue, where the system would put them. Three things follow from that:

- **Pump after every event.** `SDL_PushEvent` only queues it.
- **Size the window, then wait.** A system applies a new window size when it gets to it, and a key sent to a text box with no size moves nothing. `PumpFor` gives it time, and the layout is checked by comparing edges rather than pixels.
- **An event's text has to outlive the event.** `SDL_EVENT_TEXT_INPUT` points at its string, so `TypeText` keeps it in a `static`.

examples/ch37/tpad.tiko

```
function SelfTest( byval ed as pad.Editor ptr ) as long
    ed->ask = @ScriptedAsk
    ed->pick = @ScriptedPick
    fs.WriteAllText( "lf.tmp", !"one\ntwo\nthree\n" )
    dim as TK_HOST ptr h = tkForm_Host( ed->form )
    tkForm_Show( ed->form )
    Pump( )

    print "== a new window"
    Report( ed )
    print "== the layout, at 600 x 400"
    SDL_SetWindowSize( h->win, 600, 400 )
    PumpFor( 300 )
    print "  text meets the scroll bar: "; _
          cbool( ed->text->x + ed->text->cw = ed->scroll->x )
    print "  text meets the status bar: "; _
          cbool( ed->text->y + ed->text->ch = ed->status->y )
    print "  status bar is as wide as the window: "; _
          cbool( ed->status->cw = h->root->cw )

    print "== File > Open"
    pad.Execute( ed, pad.CmdOpen )
    Report( ed )
    print "== Down, Down, End, and type ""!"""
    tkHost_SetFocus( h, ed->text )
    Press( ed, SDLK_DOWN )
    Press( ed, SDLK_DOWN )
    Press( ed, SDLK_END )
    TypeText( ed, "!" )
```

```
    Report( ed )

    print "== Find Next, for ""T"", twice"
    ed->findWhat = "T"
    pad.Execute( ed, pad.CmdFindNext )
    Report( ed )
    pad.Execute( ed, pad.CmdFindNext )
    Report( ed )
    print "== Find Next, for ""four"""
    ed->findWhat = "four"
    pad.Execute( ed, pad.CmdFindNext )

    print "== File > Save"
    pad.Execute( ed, pad.CmdSave )
    Report( ed )
    dim as ustring saved = fs.ReadAllText( "lf.tmp" ).Value( )
    print "  on disk: "; strings.Escape( saved )

    print "== type, then File > New, twice"
    TypeText( ed, "?" )
    pad.Execute( ed, pad.CmdNew )
    Report( ed )
    pad.Execute( ed, pad.CmdNew )
    Report( ed )

    print "== moved to a monitor at twice the DPI"
    h->fScaleForced = 2 * h->fScale
    dim as SDL_Event e
    e.type = SDL_EVENT_WINDOW_DISPLAY_SCALE_CHANGED
    e.window.windowID = h->winId
    SDL_PushEvent( @e )
    Pump( )
    print "  the text is still monospaced: "; _
          cbool( tkTextBox_GetFont( ed->text ) = _
                 tkForm_Font( ed->form, TKFONT_MONO ) )

    print "== text that is not ASCII, Find Next for ""lait"", twice"
    fs.WriteAllText( "lf.tmp", "caf" & wchr( &hE9 ) & !" au lait\n" & _
                     "na" & wchr( &hEF ) & "ve lait" )
    pad.Execute( ed, pad.CmdOpen )
    ed->findWhat = "lait"
    pad.Execute( ed, pad.CmdFindNext )
    print "  selected: "; tkTextBox_GetSelText( ed->text )
    Report( ed )
```

```
    pad.Execute( ed, pad.CmdFindNext )
    print "  selected: "; tkTextBox_GetSelText( ed->text )
    Report( ed )

    print "== File > Exit while the file dialog is open"
    ed->pick = @ImpatientPick
    pad.Execute( ed, pad.CmdOpen )
    ed->pick = @ScriptedPick

    print "== type, then the system asks the program to quit"
    TypeText( ed, "x" )
    e.type = SDL_EVENT_QUIT
    SDL_PushEvent( @e )
    Pump( )
    print "  window closed: "; cbool( h->bQuit )

    print "== type, then close the window"
    TypeText( ed, "x" )
    tkForm_Close( ed->form )
    print "  window closed: "; cbool( h->bQuit )
    fs.RemoveFile( "lf.tmp" )
    return 0
end function
```

examples/ch37/tpad.out

```
== a new window
  title:  Untitled - tpad
  status: Ln 1, Col 1 | Windows (CRLF) | UTF-8
== the layout, at 600 x 400
  text meets the scroll bar: true
  text meets the status bar: true
  status bar is as wide as the window: true
== File > Open
  picked: lf.tmp
  title:  lf.tmp - tpad
  status: Ln 1, Col 1 | Unix (LF) | UTF-8
== Down, Down, End, and type "!"
  title:  *lf.tmp - tpad
  status: Ln 3, Col 7 | Unix (LF) | UTF-8
== Find Next, for "T", twice
  title:  *lf.tmp - tpad
  status: Ln 2, Col 2 | Unix (LF) | UTF-8
  title:  *lf.tmp - tpad
  status: Ln 3, Col 2 | Unix (LF) | UTF-8
```

```
== Find Next, for "four"
  asked: Cannot find "four"
== File > Save
  title:  lf.tmp - tpad
  status: Ln 3, Col 2 | Unix (LF) | UTF-8
  on disk: one\ntwo\nthree!\n
== type, then File > New, twice
  asked: Do you want to save changes to lf.tmp?
  answered: Cancel
  title:  *lf.tmp - tpad
  status: Ln 3, Col 2 | Unix (LF) | UTF-8
  asked: Do you want to save changes to lf.tmp?
  answered: Don't Save
  title:  Untitled - tpad
  status: Ln 1, Col 1 | Windows (CRLF) | UTF-8
== moved to a monitor at twice the DPI
  the text is still monospaced: true
== text that is not ASCII, Find Next for "lait", twice
  picked: lf.tmp
  selected: lait
  title:  lf.tmp - tpad
  status: Ln 1, Col 13 | Unix (LF) | UTF-8
  selected: lait
  title:  lf.tmp - tpad
  status: Ln 2, Col 11 | Unix (LF) | UTF-8
== File > Exit while the file dialog is open
  window closed: false
== type, then the system asks the program to quit
  asked: Do you want to save changes to lf.tmp?
  answered: Cancel
  window closed: false
== type, then close the window
  asked: Do you want to save changes to lf.tmp?
  answered: Don't Save
  window closed: true
```

The book's harness compares that output with what the program prints on all three systems, and there is one `.out` file for all of them. On Linux it also holds under both SDL video drivers, X11 and Wayland.

The first 36 lines are Chapter 36's, byte for byte. The two self-tests are the same script run against two window systems, so that agreement says more about both programs than either test says alone. The sections after it are the ones this chapter added: text that is not ASCII, the dialog, the system's Quit.

What the self-test cannot see

The self-test reads the model, and Chapter 36's lesson stands: it cannot see pixels. `--shot` is for that:

examples/ch37/tpad.tiko

```
'' Renders the window once and saves what it drew.
sub TakeShot( byval ed as pad.Editor ptr, byref file as ustring )
    tkForm_Show( ed->form )
    PumpFor( 300 )
    dim as TK_HOST ptr h = tkForm_Host( ed->form )
    tkHost_Render( h )
    tkHost_RenderTo( h, false )
    dim as SDL_Surface ptr s = SDL_RenderReadPixels( h->renderer, 0 )
    dim as string path = file
    if s then
        SDL_SaveBMP( s, strptr( path ) )
        SDL_DestroySurface( s )
    end if
end sub
```

It runs at a fixed scale, in the dark theme, and saves a bitmap; the picture at the top of the chapter is one. Reading the pixels back needs the second argument of `tkHost_RenderTo` to be `false`, because on Metal the back buffer is black once it has been presented. With the same scale on two systems you can compare the files, and what remains is small: mostly a level or two of anti-aliasing, from one platform's build of SDL3_ttf to another's.

The real file dialogs and the real message box are the other thing the test does not reach. They are behind `pick` and `ask` because they wait for a person, and that person has to try them once on each system.

Running it

From the chapter's directory:

```
tiko build tpad
out/debug/tpad notes.txt
```

On Windows the program is `out/debug/tpad.exe`. The first build compiles the tkControls modules the program imports; later builds compile only what changed. `tiko run -- notes.txt` builds and starts it in one step.

Shipping

`tiko build tpad --profile release` writes an optimised program, and the build copies what it needs beside it. What that is differs by system:

	The program	**Beside it**	**The folder**
Windows	337,920 bytes	the fonts, `SDL3.dll`, `SDL3_ttf.dll`, `SDL3_image.dll`	10.3 MB
Linux	348,888 bytes	the fonts	5.0 MB
macOS	361,480 bytes	the fonts	5.0 MB

On **Windows** the folder is the installation: the toolchain ships the three SDL libraries and copies them. The fonts are Inter, Cascadia Mono and Fluent UI System Icons, with their licences, and they are why the program looks the same everywhere.

On **Linux and macOS** the program links the system's SDL3 and needs it on the machine that runs it. On a Mac that is `brew install sdl3 sdl3_ttf sdl3_image`. On Linux it is the distribution's SDL3 packages, and where a distribution has none yet, `examples/tkControls/tools/build-linux-libs.sh`, which comes with the toolchain, builds them. Tell the link where they are with `tiko build tpad --options "-p /path/to/libs"`, and the program with `LD_LIBRARY_PATH` or an installed copy.

What changed from Chapter 36

Chapter 36, PsControls	Here, tkControls
`PsApplication`, `PsForm`, `f->Item( id )`	`tkApp_Init`, `TK_FORM`; the `Editor` keeps its widget pointers
`PsForm.FromHwnd( h )->Tag`	`tkForm_FromWidget( w )->Tag`
Posting `WM_CARETMOVED` to the form	`tkApp_RunOnUI( @CaretMoved, ed )`
An accelerator table	`tkForm_OnKeyDown`: keys the text box leaves unclaimed
A modal `CIOpenFileDialog`	`SDL_ShowOpenFileDialog`, waited for by `PickFile`
`SetWindowTextW`, in UTF-16	`SDL_SetWindowTitle`, in UTF-8
Offsets that match `ustring` units	UTF-8 offsets, converted by `offsets`
`EM_SCROLLCARET` after `SetSel`	Nothing: `SetSel` scrolls into view
A status-bar paint callback	Nothing: `tkStatusBar` draws its panels
`OnDpiChanged` restores the monospaced font	Nothing: a scale change keeps each control's font
Ctrl	`TK_KMOD_PRIMARY`: Cmd on macOS
The window's close	The close, and the system's `SDL_EVENT_QUIT` through a hook
`tpad.rc`, a manifest, `UNICODE`	Nothing
`WM_KEYDOWN`, `WM_CHAR` in the self-test	`SDL_EVENT_KEY_DOWN`, `SDL_EVENT_TEXT_INPUT`

What it costs

- **A larger installation.** Chapter 36's `tpad` is one 316 KB file that imports only what comes with Windows. This one is about the same size, and it brings 10 MB of libraries and fonts on Windows and needs SDL3 installed elsewhere. Portability is paid for in what you ship.
- **Every caret move reads the whole text, and now converts it.** Measured on a 1 MB file in a release build on Windows, `ShowStatus` takes about 10 ms, of which under 2 ms is `tkTextBox_GetText` and the rest is copying and counting. A notes file costs nothing; a large log would make the caret lag.
- **The controls are drawn, not provided.** A `tkTextBox` looks and behaves the same on every system, which is the reason for using it, and for the same reason it is nothing the system recognises. There is no accessibility interface, so a screen reader sees an empty window. The menu bar is in the window, on a Mac too, not in the Mac's menu bar at the top of the screen.
- **The column counts UTF-16 units.** A character outside the Basic Multilingual Plane, such as U+1F600, moves the column by two. Exercise 1 fixes it.

- **A new document is saved with CR LF on every system**, because `textfile` was written for Windows, and the status bar says so honestly. A file that was read keeps its own ending. Exercise 2 fixes it.
- **The text always wraps**, as in Chapter 36: a multiline box wraps and has no horizontal scroll bar.
- **The same limits as Chapter 36** on the file in memory, and on text that is not UTF-8.

Summary

- A module that imports no window moves to a new window system unchanged. `textfile` is the same file in both chapters.
- tkControls replaces handles and messages with a `TK_WIDGET` tree in one SDL window. Keep a pointer to each control you talk to, and reach the program's state from a callback through `tkForm_FromWidget` and the form's `Tag`.
- SDL3's file dialogs return at once and answer through a callback that may run on another thread. Wait for the answer by turning the event loop, and have the callback touch nothing but C memory and an atomic.
- While a loop turns under a dialog, anything can be delivered. Refuse new commands and the close until the dialog answers, or an `Editor` can be freed underneath the code waiting on it.
- `tkTextBox` offsets are UTF-8 bytes and a `ustring` counts UTF-16 units. They agree for ASCII and nowhere else. Convert at the edge, in one place, and test with text that is not English.
- `tkApp_RunOnUI` runs code after the current event, as a posted message did.
- `TK_KMOD_PRIMARY` is Cmd on macOS and Ctrl elsewhere. Only the menu's label needs `#if`.
- A system's Quit is `SDL_EVENT_QUIT`, an event that belongs to no window. Turn it into a close with an event hook, or it skips the question before discarding.
- One self-test and one expected output, run on every system, is what shows that the program is one program. Push SDL events, pump after each, and wait after resizing. `--shot` covers the pixels.
- A program built on SDL3 ships its libraries and fonts on Windows, and asks for SDL3 on Linux and macOS.

Exercises

1. Make the status bar's column count characters, so that U+1F600 counts as one. A ustring holds such a character as two units, and the second is a low surrogate. {solution="ex37-01″}

2. Make a new document use the line ending of the system it was started on, through a `NewDocument` function and `#if`, while a file that was read keeps its own. {solution="ex37-02″}
3. Port Find Previous (Shift+F3) from Chapter 36's exercises. Mind which count each side uses.
4. On macOS, make File > Exit read Quit, with Cmd+Q as its key, while Windows and Linux keep Exit.
5. Open a file dropped on the window. Add an event hook for `SDL_EVENT_DROP_FILE`, whose `data` field is the path in UTF-8, and call `MayDiscard` first.
6. Remember the window's size between runs, in a file under the folder `SDL_GetPrefPath` returns, and check that the same code finds the right place on all three systems.

Appendix A: Keywords and Operator Precedence

A name in tiko is one of five things, and they follow different rules. A **reserved word** is never a name. A **runtime name** may not be reused at module level. A **contextual word** is a keyword only where its statement expects it. A **fixed-width type name**, like the constant `nullptr`, is predeclared and may be replaced. Everything else is free. The rules below were measured by compiling each word in each position with tiko and with fbc-modern 1.20, and the two give the same answer for every word in every position.

Reserved words (247)

A reserved word cannot name a variable, a constant or a procedure, at module level, inside a procedure or inside a namespace. Case does not matter. The diagnostic is usually `TKO0600: '…' is already declared in this scope`, because the word is declared by the language itself; where the word starts a statement of its own, the parser reports that statement's error instead.

__fastcall	cos	draw	let	print	tan
__thiscall	cptr	dynamic	lib	private	then
abs	cshort	else	line	procptr	threadcall
abstract	csign	elseif	local	property	to
access	csng	encoding	lock	protected	trim
acos	cubyte	end	log	pset	type
alias	cuint	endif	long	ptr	typeof
and	culng	enum	longint	public	ubound
andalso	culngint	eqv	loop	put	ubyte
any	cunsg	erase	lprint	random	ucase
append	cushort	err	lset	read	uinteger
as	cva_arg	error	ltrim	redim	ulong
asc	cva_copy	exit	mid	rem	ulongint
asin	cva_end	exp	mkd	restore	union
asm	cva_start	explicit	mki	resume	unlock
atan2	cvd	export	mkl	return	unsigned
atn	cvi	extends	mklongint	rset	until
base	cvl	extern	mks	rtrim	ushort
binary	cvlongint	field	mkshort	sadd	using
boolean	cvs	fix	mod	scope	ustring
byref	cvshort	for	name	screen	va_first
byte	data	frac	namespace	seek	var
byval	declare	function	new	select	varptr
call	defbyte	get	next	sgn	view
case	defdbl	gosub	not	shared	virtual
cast	defined	goto	on	shl	wchr
cbool	defint	if	open	short	wend
cbyte	deflng	iif	operator	shr	while
cdbl	deflongint	imagecreate	option	sin	width
cdecl	defshort	imp	or	single	window
chr	defsng	implements	orelse	sizeof	winput
cint	defstr	import	output	spc	with
circle	defubyte	include	overload	sqr	write
class	defuint	input	paint	static	wstr
clng	defulng	instr	palette	stdcall	wstring
clngint	defulongint	integer	pcopy	step	xor
cobj	destructor	is	peek	stop	zstring
common	dim	isredir	pmap	string	
const	do	key	point	sub	
constructor	double	lbound	poke	swap	

The list includes every **statement and function word** of the classic language — `print`, `open`, `len`, `read`, `name`, `width`, `lock` — and every built-in type name, `double` and `string` included. `function Double( )` is refused, and so is a variable called `read`.

Keywords as field names

A **field** may be named by most reserved words, because a field's name is only ever read after `.` or `->`, or in its type's body. A type can have a field called `name`, `type`, `len` or `constructor`, and `t.name` reads it. The exceptions:

- The operator words `and`, `andalso`, `eqv`, `imp`, `mod`, `not`, `or`, `orelse`, `shl`, `shr` and `xor`, and `delete` and `new`, are never field names.
- In the type-first spelling, `as T name`, `const`, `ptr` and `pointer` are part of the type, not a name; `ptr as long` is a field.
- In the name-first spelling, `name as T`, the declaration words `const`, `declare`, `dim`, `enum`, `redim`, `static`, `private`, `protected` and `public` start a declaration instead, and `type` and `union` cannot take array bounds (`type( 0 to 2 ) as long`).
- In the type-first spelling, `alias` is not a name either. `alias as long` is fine.
- `rem` starts a comment where a statement can end, so `rem as long` is a comment. `as long rem` declares a field named `rem`, and `t.rem` reads it.

Runtime names (104)

These are the runtime library's procedures. They are not reserved words. The runtime declares them in the global namespace, so a declaration of the same name at module level collides with it: `TKO0813: '…' is already declared by the runtime library`. That covers a variable, a constant, a procedure, a `for` counter, a local inside a module-level procedure, and a parameter of a procedure's definition. The same name is fine inside a namespace or a method, as a field, a member procedure or an enum member, and as a parameter of a prototype.

allocate	csrlin	fre	multikey	rnd	settime
beep	curdir	freefile	mutexcreate	run	shell
bload	date	getjoystick	mutexdestroy	screencopy	space
callocate	deallocate	getkey	mutexlock	screenevent	stop
chain	dylibfree	imageconvertrow	mutexunlock	screenlist	system
chdir	dylibload	imagedestroy	out	screenlock	threadcreate
clear	environ	inkey	pcopy	screenptr	threadwait
cls	eof	inp	pmap	screenres	time
command	erfn	kill	pointcoord	screenset	timer
condbroadcast	erl	loc	pos	screensync	wait
condcreate	ermn	locate	randomize	screenunlock	windowtitle
conddestroy	exec	lof	reallocate	setdate	wspace
condsignal	exepath	lpos	reset	setenviron	
condwait	flip	mkdir	rmdir	setmouse	

These also take a **new overload**: `sub Hex( )` compiles beside the runtime's `hex`, but `dim as long hex` does not.

bin	getmouse	oct	screeninfo	vallng	whex
bsave	hex	right	sleep	valuint	woct
dir	imageinfo	screencontrol	val	valulng	
dylibsymbol	left	screenglproc	valint	wbin	

`#define` is the exception: tiko accepts `#define run 5`, where fbc refuses it as a duplicated definition (Appendix D).

Contextual words

FBC-Modern's extensions added these words without reserving them, so existing code that uses them as names keeps compiling. Each one is a keyword only in the position shown, and an ordinary name everywhere else: each one compiles as a variable, a procedure, a field, a local and a namespace member.

Word	A keyword in
`channel`	`select channel` (Chapter 23)
`defer`	`defer` statement, when a name, a keyword other than `as`, or the end of the statement follows (Chapter 6)
`each`	`for each` (Chapter 20)
`from`	`case receive v from ch` in `select channel`
`in`	`for each x in …`
`of`	type arguments: `Array( of int32 )` (Chapter 17)
`override`	after a method's declaration: `declare sub M( ) override` (Chapter 14)
`receive`	`case receive` in `select channel`
`send`	`case send v to ch` in `select channel`
`spawn`	`spawn Proc( args )` (Chapter 23)
`timeout`	`case timeout ms` in `select channel`

Fixed-width type names

tiko predeclares ten type names that fbc does not have. They are typedefs, not reserved words (Chapter 4): a program's own declaration of one of them replaces tiko's without a diagnostic, so code that already declares `INT32` — as `win/basetsd.bi` does — compiles unchanged.

Name	Same type as	Name	Same type as
`int8`	`byte`	`uint8`	`ubyte`
`int16`	`short`	`uint16`	`ushort`
`int32`	`long`	`uint32`	`ulong`
`int64`	`longint`	`uint64`	`ulongint`
`float32`	`single`	`float64`	`double`

The predeclared constant

tiko predeclares one constant that fbc does not have, `const nullptr as any ptr = 0`: the null pointer, as a pointer (Chapter 11). It is replaced on the same terms as the type names above, so a program's own `nullptr` — fbc's own test suite has a field by that name — compiles unchanged.

Operator precedence

Tightest first. Operators on one row bind equally, and **every binary operator is left-associative**, `^` included. The member operators `.` and `->`, calls `( )` and indexing `[ ]` bind tighter than everything here.

Level	Operators
1	`^`
2	`-` (sign) `+` (sign)
3	`* /`
4	`\`
5	`mod`
6	`shl shr`
7	`+ -`
8	`is &`
9	`= <> < <= > >=`
10	`not`
11	`and`
12	`or`
13	`imp eqv xor`
14	`orelse andalso`

The table is FreeBASIC's, measured operator pair by operator pair against fbc, and several rows differ from C and from intuition (Chapter 5). Each line below prints the same with tiko and with fbc-modern:

```
2 ^ 3 ^ 2            64     ^ is left-associative: (2 ^ 3) ^ 2
-2 ^ 2               -4     ^ binds tighter than the sign
1 shl 2 + 1           5     shl binds tighter than +, unlike C's <<
8 \ 4 * 2             1     * binds tighter than \: 8 \ (4 * 2)
2 * 3 mod 4           2     mod is looser than * and \
12 or 10 xor 6        8     or binds tighter than xor
1 or 2 and 0          1     and binds tighter than or
0 andalso 1 orelse 1 -1     andalso and orelse share one level
not 1 = 0            -1    not is looser than the comparisons: not (1 =
0)
not 0 and 3           3     ...and tighter than and: (not 0) and 3
```

`&` binds looser than `+` and `-`, so `"a" & 1 + 2` means `"a" & (1 + 2)`, and both compilers print `a3`.

Appendix B: Diagnostics

Every diagnostic the compiler can emit, as of the pinned version: 602 in all 567 errors, 14 notes, 21 warnings.

A diagnostic's number is stable forever. Retiring one leaves its row in place rather than reusing the number, so an id in an old build log still means what it meant then. Suppress a warning with `-Wno-TKOnnnn`, and turn warnings into errors with `-Werror`.

Text like ‹arg1› marks where the compiler substitutes a name, type or path.

Source and file handling (TKO0001–TKO0099)

Id	Severity	Message
`TKO0001`	error	cannot open source file '‹arg1›': ‹arg2›
`TKO0002`	error	source file '‹arg1›' is empty
`TKO0003`	warning	source file '‹arg1›' is not valid UTF-8; bytes preserved verbatim

Driver and command line (TKO0100–TKO0199)

Id	Severity	Message
TKO0100	error	unknown option ‘‹arg1›’
TKO0101	error	option ‘‹arg1›’ requires an argument
TKO0102	error	no input files
TKO0103	error	unknown target ‘‹arg1›’
TKO0104	error	32-bit targets are not supported; x86-64 only
TKO0105	warning	unknown diagnostic id ‘‹arg1›’
TKO0106	error	‘‹arg1›’ is not a recognised value for ‘‹arg2›’
TKO0108	error	‘‹arg1›’ is a Windows resource script; the target is not Windows
TKO0109	error	no resource compiler: ‹arg1›
TKO0110	error	compiling resource ‘‹arg1›’ failed (exit ‹arg2›)
TKO0111	error	no usable toolchain: ‹arg1›
TKO0112	error	cannot write intermediate file ‘‹arg1›’
TKO0113	error	clang failed on ‘‹arg1›’ (exit ‹arg2›)
TKO0114	error	‘‹arg1›’ is a ‹arg2› object, which target triple ‘‹arg3›’ does not make
TKO0115	error	archiving failed (exit ‹arg1›)
TKO0116	error	link failed (exit ‹arg1›)
TKO0117	error	cannot write the diagnostics file ‘‹arg1›’
TKO0118	error	-cimport needs -cimport-out : the binding has to go somewhere
TKO0119	error	importing a C header needs libclang: ‹arg1›
TKO0120	error	unable to import C header ‹arg1›
TKO0121	warning	[illegible] newer, or set

Lexer (TKO0200–TKO0299)

Id	Severity	Message
`TKO0201`	warning	missing closing quote in string literal
`TKO0202`	error	unterminated block comment
`TKO0203`	error	‘‹arg1›’ is not followed by any digits
`TKO0205`	warning	numeric literal is too large for 64 bits, truncated
`TKO0206`	warning	suffix ignored in ‘‹arg1›’
`TKO0207`	error	block string has no closing line; one must start with three quotes
`TKO0208`	error	block string line does not start with the closing line’s indentation

Preprocessor (TKO0300–TKO0399)

Id	Severity	Message
TKO0300	error	unknown directive ‘#‹arg1›’
TKO0301	error	cannot open include file ‘‹arg1›’
TKO0302	error	‘#‹arg1›’ without a matching ‘#if’
TKO0303	error	‘#if’ has no matching ‘#endif’
TKO0304	error	preprocessor nesting exceeded ‹arg1› levels
TKO0305	error	macro ‘‹arg1›’ is already defined
TKO0306	error	macro ‘‹arg1›’ takes ‹arg2› argument(s)
TKO0307	error	unterminated argument list for macro ‘‹arg1›’
TKO0308	error	invalid expression in ‘#‹arg1›’
TKO0309	error	#error: ‹arg1›
TKO0310	error	‘#‹arg1›’ is not implemented in this build
TKO0311	error	‘#macro’ without ‘#endmacro’
TKO0312	warning	unknown ‘#pragma ‹arg1›’
TKO0314	error	builtin macro ‘‹arg1›’ is not implemented in this build
TKO0315	note	#print ‹arg1›
TKO0316	error	expected an identifier after ‘#‹arg1›’
TKO0317	error	unsupported language mode ‘‹arg1›’; tiko-lang implements “fb” only
TKO0318	error	cannot pop ‘#pragma ‹arg1›’: the stack is empty
TKO0319	error	**FB__UNIQUEID__POP**(‹arg1›) has no matching push
TKO0320	error	‘‹arg1›’ has no type; tiko-lang implements #lang
TKO0321	warning	[illegible]

Parser (TKO0400–TKO0599)

Id	Severity	Message
TKO0400	error	expected an expression
TKO0401	error	expected ‘‹arg1›’
TKO0402	error	expected a member name after ‘.’ or ‘->’
TKO0403	error	expression nesting exceeded ‹arg1› levels
TKO0404	error	unexpected token after expression
TKO0405	error	expected ‘end ‹arg1›’
TKO0406	error	expected an identifier
TKO0407	error	inline assembly is not supported; compile that module with fbc
TKO0408	error	expected a statement
TKO0409	error	‘‹arg1›’ is not available in the fb dialect
TKO0410	error	‘for each’ with a name list takes no ‘as’ clause; each name is typed by its own element
TKO0411	error	‘for each byref’ cannot take a name list; bind the whole element byref instead
TKO0412	error	‘for each’ with a name list needs at least two names
TKO0413	error	every lambda capture needs an explicit ‘byval’ or ‘byref’
TKO0414	error	‘select channel’ needs at least one case
TKO0415	error	expected receive, send, timeout or else after ‘case’
TKO0416	error	expected ‘‹arg1›’ in this ‘select channel’ case
TKO0417	error	only one ‘case timeout’ and one ‘case else’ are allowed
TKO0418	error	‘select case’ needs at least one ‘case’
TKO0419	error	‘is’ is not allowed in a ‘select case as const’
TKO0520	error	[illegible] ‘return’

Semantic analysis (TKO0600–TKO0899)

Id	Severity	Message
TKO0860	error	‘‹arg1›’ is not public in module ‘‹arg2›’
TKO0862	error	alias ‘‹arg1›’ is already bound in this file (by ‘#import “‹arg2›”’)
TKO0863	error	alias ‘‹arg1›’ conflicts with a declaration of the same name
TKO0864	error	‘‹arg1›’ belongs to imported module ‘‹arg2›’; only that module can define it
TKO0865	note	‘‹arg1›’ here is the procedure ‘‹arg2›’, which hides ‹arg3›
TKO0866	note	‘‹arg1›’ is also declared here, in a header an ‘#import’ reads
TKO0867	error	constant definitions form a cycle: ‹arg1›
TKO0869	error	‘‹arg1›’ is initialised from ‘‹arg2›’, which is declared later
TKO0868	error	‘‹arg1›’ is module-private but appears in the public declaration of ‘‹arg2›’
TKO0870	error	public ‘‹arg1›’ has the same name as its module ‘‹arg2›’
TKO0600	error	‘‹arg1›’ is already declared in this scope
TKO0602	error	‘‹arg1›’ is a ‹arg2›, not a procedure
TKO0603	error	calling convention does not match the declaration of ‘‹arg1›’
TKO0604	error	#assert condition failed
TKO0605	error	expected a constant expression
TKO0606	error	‘‹arg1›’ cannot be walked
[illegible]	[illegible]	[illegible]

Notes (TKO0900–TKO0989)

Id	Severity	Message
TKO0962	error	clang reported success but wrote no object ‘‹arg1›’
TKO0908	error	a constructor( ) chain call must be the first statement of the constructor
TKO0909	error	operator -> must return a UDT, by value or byref
TKO0910	error	a pointer does not fit in a ‘‹arg1›’; only a pointer-sized integer takes one
TKO0911	error	type<‹arg1›>( ) needs values: the type has no constructor to call
TKO0912	error	no constructor of ‘‹arg1›’ takes these values: a type that extends OBJECT is not filled field by field
TKO0913	error	‘‹arg1›’ is an array of ‘‹arg2›’, whose destructor is not cdecl; an array can only destroy through a cdecl one
TKO0914	error	‘‹arg1›’ is an array: GET # and PUT # take an element, or the whole array written ‘‹arg1›( )’
TKO0915	error	‘‹arg1›’ takes no name: the next thing is FOR or AS
TKO0916	error	this type has more than one conversion to a pointer, so the graphics statement cannot choose one
TKO0917	error	‘‹arg1›’ has no constructor, so ‘‹arg1›(…)’ is no call
TKO0918	error	a store through the address of a temporary: the value is gone at the end of the statement
TKO0919	error	‘‹arg1›’ is ambiguous: more
TKO0988	warning	than one USING in effect [illegible]

Compiler limits and internal errors (TKO0990–TKO0999)

Id	Severity	Message
TKO0990	error	too many errors; stopping
TKO0991	error	input exceeds a compiler limit: ‹arg1›
TKO0992	error	not implemented in this build: ‹arg1›
TKO0993	error	internal compiler error in emission: ‹arg1›
TKO0994	error	internal compiler error: this ‹arg2› was not compiled (‹arg1› in this module); the build stops rather than leave it out
TKO0995	warning	internal compiler error: this ‹arg2› was not compiled (‹arg1› in this module) and is left out of the build
TKO0996	error	‘‹arg1›’ is a bare ‹arg2›, which has no length; write ‘‹arg2› * N’ or ‘‹arg2› ptr’
TKO0997	error	‘‹arg1›’ is a procedure, not a type
TKO0998	error	‘‹arg1›’ cannot be a member of a union: ‹arg2›
TKO0999	error	‘‹arg1›’ is not a static member; call it through an object of its type

Interfaces (TKO1000–TKO1049)

Id	Severity	Message
TKO1000	error	‘‹arg1›’ does not implement ‘‹arg2›.‹arg3›’; add ‘implements ‹arg2›.‹arg3›’ to one of its methods
TKO1001	error	‘‹arg1›’ cannot implement ‘‹arg2›’: ‹arg3› differ
TKO1002	error	‘‹arg1›’ is not an interface
TKO1003	error	interface ‘‹arg1›’ has no member named ‘‹arg2›’
TKO1004	error	‘‹arg1›’ implements ‘‹arg2›.‹arg3›’, but does not list ‘‹arg2›’ after ‘implements’
TKO1005	error	‘‹arg2›.‹arg3›’ is implemented twice in ‘‹arg1›’
TKO1006	error	interface ‘‹arg1›’ is listed more than once
TKO1008	error	interface ‘‹arg1›’ inherits itself
TKO1009	error	‘‹arg1›’ does not convert to ‘‹arg2›’‹arg3›
TKO1007	error	interface ‘‹arg1›’ cannot be created; create a type that implements it
TKO1012	error	an interface reference cannot refer to a temporary; store the object in a variable first
TKO1013	error	‘delete’ cannot destroy through interface ‘‹arg1›’; cast to the object’s type first
TKO1010	error	‘‹arg1›’ is ambiguous: both ‘‹arg2›’ and ‘‹arg3›’ declare it; cast to one of them
TKO1015	error	interface type ‘‹arg1›’ cannot appear in a foreign declaration
TKO1016	error	‘‹arg1›’ does not implement ‘‹arg2›’: ‹arg3› differ in ‘‹arg4›’ parameters; call through ‘‹arg2›’ use the interface’s

String slices (TKO1050–TKO1099)

Id	Severity	Message
`TKO1050`	error	‘‹arg1›’ cannot be sliced: it is not a string, a string slice or a zstring/wstring pointer
`TKO1051`	error	slice ‹arg1› is outside ‘‹arg2›’ (0 to ‹arg3›)
`TKO1052`	error	a const ‹arg1› cannot convert to a mutable ‹arg1›
`TKO1053`	error	a mutable ‹arg1› parameter needs a ‹arg2› variable, not ‘‹arg3›’
`TKO1054`	error	a ‹arg1› cannot be ‹arg2›
`TKO1055`	error	type ‘‹arg1›’ holds slice field ‘‹arg2›’ and must be declared ‘as view’
`TKO1056`	error	cannot take the address of slice ‘‹arg1›’ except to pass it
`TKO1057`	error	‘‹arg1›’ outlives ‘‹arg2›’, which this slice borrows from
`TKO1058`	error	this slice borrows from a temporary that ends with the statement
`TKO1059`	error	cannot return a slice of ‹arg1›
`TKO1060`	error	‹arg1› cannot capture slice ‘‹arg2›’
`TKO1061`	error	a slice stored through byref parameter ‘‹arg1›’ must outlive the call
`TKO1062`	error	‘‹arg1›’ cannot be changed while slice ‘‹arg2›’ borrows from it (last used at line ‹arg3›)

Appendix C: Command-line reference

All 95 options of the pinned compiler, from the table that also drives `tiko --help`.

Environment variables

Variable	What it does
`TIKO_CLANG`	Full path to `clang`. Without it tiko looks at `C:\Program Files\LLVM\bin\clang.exe` on Windows or `/usr/bin/clang` on Linux, and searches `PATH` only when that file is absent. `llvm-ar` and `llvm-rc` are taken from the same directory.
`TIKO_INCLUDE`	Replaces the default include directory (`<exe>/inc`). `-i` paths are searched first either way.
`TIKO_LIBPATH`	Replaces the bundled library directory (`<exe>/lib/<target>`). `-p` paths and `#libpath` are searched first either way.

Exit codes

Code	Meaning
0	success
1	errors in the program
2	command-line error
3	no usable toolchain (clang not found)
4	internal compiler error
5	clang or the linker failed

How arguments are parsed

- An option matches exactly (`-g`), as `--option=value`, or with the value attached (`-O2`, `-Wno-TKO0201`).
- An optional value must be attached with `=`: `--dump-tokens=pp`.
- `--` ends option parsing; everything after it is an input file.
- Inputs are classified by extension: `.o` is an object, `.rc` is a resource script, anything else is source.
- A `#cmdline "..."` line in the first source file restarts the compile with those options, and the source wins over the command line.

Subcommands

A subcommand is the first word on the command line. It builds a project from its `tiko.toml`, as Chapter 31 shows; every other command line is the compiler driver described by the rest of this appendix.

Command	What it does
`tiko build`	build the targets of tiko.toml, or the roots given – cached
`tiko clean`	remove what 'tiko build' would write
`tiko metadata`	print the project as JSON: targets, profiles, modules
`tiko run`	build one exe target, then run it with the arguments after –
`tiko check`	every diagnostic a build would give, through lowering; nothing written
`tiko c2t`	translate a C header into a tiko header: tiko c2t <in.h> [-o <out.ti>|-]

General

Option	What it does
`--help[=‹category›]`	show help; optionally for one category
`--version`	show version and exit
`-v`	echo every tool invocation
`-restart`	accepted and ignored; a '#cmdline' restarts by itself
`--print-graph`	print the modules the inputs import, in build order, and stop
`--emit-tki`	print the interface of the module directory given, and stop
`--frontend-only`	report every diagnostic a build would, through lowering; write no object, link nothing ('tiko check')
`--no-module-cache`	build imported modules without reading or writing the module cache
`--clean-module-cache`	empty the module cache and stop
`-z ‹option›`	fbc extended option; accepted and ignored
`-maxinstdepth ‹n›`	how deep one generic instantiation may drive another (default 64)

Input

Option	What it does
`-i ‹dir›`	add an include search path
`--pkg-dir ‹dir›`	the package tree for #import std/… is found in (else TIKO_PKG, else /pkg)
`-d ‹name=val›`	define a preprocessor symbol
`-p ‹dir›`	add a library search path
`-l ‹name›`	link against a library

Output

Option	What it does
`-Wl ‹a,b,c›`	pass options to the linker, split at commas, as fbc's -Wl
`-norpath`	no $ORIGIN run path when a Linux link takes a shared library
`-nocopylibs`	do not copy vendor libraries or modules' runtime files beside the output
`-o ‹file›`	output path
`-x ‹file›`	the linked output's path, as fbc's -x: over -o, the last one winning; ignored with -c
`-c`	compile only; do not link
`-dll`	build a DLL rather than an executable
`-dylib`	same as -dll
`-lib`	build a static library
`-e`	enable runtime error checking
`-ex`	as -e: RESUME, which fbc adds here, does not exist in the fb dialect
`-exx`	-ex plus array bounds and null-pointer checking
`-g`	emit debug information
`-earray`	check array bounds (one part of -exx)
`-enullptr`	check pointer dereferences for null (one part of -exx)
`-elocation`	record the procedure and module an error happens in (one part of -exx)
`-earraydims`	check that an array is dimensioned, and with the rank used (one part of -exx)
`-eassert`	enable assert and assertwarn (one part of -g)
`-edebug`	define **FB__DEBUG** as −1 (one part of -g)
`-edebuginfo`	emit debug information alone (one part of -g)
`-eunwind`	unwind information (one part of -e; always emitted)
`-O‹level›`	optimisation level 0-3
`-mt`	link the thread-safe runtime
`-pic`	position-independent code, for a Unix shared library (Linux only)
`-m ‹name›`	which module owns main; the others become module constructors

Target

Option	What it does
`-target ‹name›`	windows-x86_64 (the Windows default), linux-x86_64 or macos-arm64
`-m64`	64-bit target (the only supported width)
`-m32`	rejected: 32-bit is not supported (x86-64 only)
`-s ‹console|gui›`	Windows subsystem of an executable; console is the default
`-t ‹kbytes›`	stack reserve and commit of a Windows executable or DLL, in KB (default 2048, minimum 64)

Diagnostics

Option	What it does
`-Werror`	treat warnings as errors
`-Wno-‹id›`	suppress one diagnostic id
`-w ‹level›`	fbc's -w: all, pedantic, none, error, or n – hide fbc's warnings below level n
`-fmax-errors ‹n›`	stop after n errors
`--color ‹when›`	always, never or auto
`--diagnostics-format ‹fmt›`	fbc or gnu
`--diagnostics-file ‹path›`	also write every diagnostic to as tab-separated records
`--allow-unlowered`	warn (TKO0995) instead of failing when a statement could not be compiled

C header import (`#cinclude` and `-cimport`)

Option	What it does
`-cinclude ‹dir›`	add a C include directory (clang's -I) for #cinclude
`-cimport ‹header›`	import a C header as a FreeBASIC binding; needs -cimport-out
`-cimport-out ‹file›`	where -cimport writes the binding; - for standard output
`-cimport-manifest ‹file›`	write clang's layout of every imported record
`-cdefine ‹name[=value]›`	define a C macro (clang's -D)
`-csysroot ‹dir›`	the C sysroot (clang's –sysroot)
`-copt ‹arg›`	pass an argument to clang verbatim
`-cnomacros`	do not import object-like macros as constants
`-cnoverify`	do not check tiko's layout of each imported record against clang's
`-cnocache`	#cinclude neither reads nor writes the binding cache
`-ccachedir ‹dir›`	where #cinclude keeps its bindings (default: the user's cache directory)
`--print-cincludes`	import each #cinclude the inputs reach, print where its binding is as JSON, and stop

Developer and compiler introspection

Option	What it does
`--dump-source`	list loaded source files and stop
`--dump-tokens[=‹raw|pp›]`	dump the token stream, before or after the preprocessor
`--lex-only`	lex every input and report counts; no dump
`--pp-only`	preprocess every input and report counts; no dump
`--dump-ast`	dump the syntax tree
`--dump-types`	dump the type table and its layout
`--dump-symbols`	collect declarations and dump the symbol table
`--dump-scan`	run the front end for an editor and dump its symbol records
`--resolve`	collect and resolve; report reference binding
`--dump-mangle`	print the emitted symbol name of each procedure
`--check`	type expressions; report the type of each PRINT argument
`--dump-decisions`	type expressions; list the overload Check chose for each node that implies one
`--dump-udt`	print each UDT's size, alignment and field offsets
`--cimport-probe ‹header›`	load libclang, parse one C header, print what clang saw
`--dump-vtable`	print each UDT's virtual method table
`--dump-itable`	print each interface's flattened members (docs/tiko-interfaces)
`--dump-rtl`	print the runtime inventory and stop
`--parse-expr`	parse each line as one expression and dump it
`--parse-only`	parse every input and report counts; the corpus gate
`--dump-ir`	dump MIR
`--dump-ir-pre-abi`	dump MIR as the frontend made it, before the ABI pass
`--verify-selftest`	run the MIR verifier over modules built to [illegible]
[illegible]	[illegible] subset into the cache, and link nothing

Appendix D: Coming from FreeBASIC

tiko compiles FreeBASIC. Its dialect is FreeBASIC 1.20's `#lang "fb"` plus tiko's extensions, which are most of what Parts III to V of this book describe. The aim is the **same accept/reject decision** as fbc and the **same observable behaviour**: what a program prints, returns and does. Existing code moves over, and mixes freely with tiko code in one program.

This appendix is the list of what is different. Every row was measured on 2026-10-04 by compiling the same program with the toolchain this book is pinned to and with **fbc 1.20** (Windows unless a row says otherwise), and comparing what each did. Where the two agree, the appendix says so too, because "does tiko do this the same way?" is the question it exists to answer.

Dialects and the features they carry

tiko implements one dialect, `fb`. A file that asks for another one is refused, not quietly compiled as `fb`:

Program	fbc	tiko
`#lang "qb"`, `"fblite"`, `"deprecated"`	compiles in that dialect	`TKO0317`: unsupported language mode
`'$lang: "qb"`	compiles as `qb`	`TKO0317`
`-lang qb` on the command line	compiles as `qb`	`TKO0100`: unknown option
inline `asm … end asm`	compiles	`TKO0407`: inline assembly is not supported

The features of the older dialects are refused by **both** compilers in `-lang fb`. fbc calls each one error 146, "Only valid in -lang deprecated or fblite or qb", and tiko `TKO0409` (`TKO0944` for a line number):

Construct	Write instead (Chapter 32)
`gosub` / `return`, `on k gosub`	a `sub`
line numbers, `10 print 1`	a label, `here:`
`call Foo( )`	`Foo( )`
`let x = 1`	`x = 1`
`option explicit`	nothing: `fb` is always explicit
`option base 1`	bounds written out, `dim a( 1 to n )`
`defint a-z` and the other `def` types	declare each variable's type
`'$include: 'x.bi'`, `'$dynamic`	`#include "x.bi"`; arrays are dynamic when declared so
`lprint`	`open lpt` and `print #`

threadcall

`threadcall Proc( args )` is meant to start a thread for an ordinary procedure. tiko does not implement it: it warns, the procedure does not run, and the handle is 0.

```
warning TKO0797: 'threadcall' is not supported: 'hello' will not run, and
the thread handle is 0
```

The fbc build used for these measurements did not run the procedure either: it also returned a handle of 0, without a warning. Use `threadcreate` with a parameter block, or `spawn` (Chapter 23).

assert

The same in both compilers. `assert( expr )` checks nothing unless the program is compiled with `-g` or `-eassert`. When it does check and the expression is false, it prints the file, the line, the procedure and the expression, and ends the program with exit code 1:

```
assert.bas(2): assertion failed at __FB_MAINPROC__: x = 3
```

The intrinsic defines

The defines a program can test with `#ifdef`, and their values. **The same in both:** `__FB_VERSION__` (`"1.20.0"`), `__FB_LANG__` (`"fb"`), `__FB_64BIT__`, `__FB_X86__`, `__FB_WIN32__` on Windows and `__FB_LINUX__` and `__FB_UNIX__` on Linux, `__FB_ASM__`, `__FB_PCOS__` on Windows, `__FB_FPU__`, `__FB_FPMODE__`, `__FB_DEBUG__`, `__FB_ERR__`, `__FB_MT__`, `__FB_VER_MAJOR__`, `__FB_VER_MINOR__` and `__FB_VER_PATCH__` (`1`, `20`, `0`), `__FB_GUI__` (`-1` under `-s gui`), `__FB_OPTIMIZE__` (the `-O` level), `__FB_PROFILE__` and

`__FB_VECTORIZE__` (`0`), `__FB_MAIN__` in the module that becomes `main`, `__FB_ARGC__` and `__FB_ARGV__` (`main`'s own parameters, usable in module-level code only), an empty `__FB_BUILD_SHA1__`, every `__FB_OPTION_…__` and `__FB_OUT_…__`, and the macro-like ones: `__FB_ARG_COUNT__` and the other `__FB_ARG_…__`, `__FB_EVAL__`, `__FB_IIF__`, `__FB_JOIN__`, `__FB_QUOTE__`, `__FB_UNQUOTE__` and the `__FB_UNIQUEID…__` family.

Different:

Define	**fbc**	**tiko**
`__TIKO_COMPILER__`	not defined	`1`: how a source tree asks "is this tiko?"
`__TIKO_VERSION__`	not defined	`"0.1.2"`
`__FB_SIGNATURE__`	`"FreeBASIC 1.20.0"`	`"tiko-lang"`
`__FB_BACKEND__`	`"gcc"`	`"llvm"`
`__FB_GCC__`	`-1`	`0`
`__FB_MIN_VERSION__( … )`	defined	not defined: `#if __FB_MIN_VERSION__( 1, 20, 0 )` is `TKO0308`
`__FB_BUILD_FORK_ID__`	the fork id fbc was built with	`"tiko-lang"`
`__FB_BUILD_DATE__`, `__FB_BUILD_DATE_ISO__`	the date fbc was built	the date tiko was built
`__FB_QUERY_SYMBOL__( … )`	defined	not defined: it queries fbc's symbol table while preprocessing, and tiko preprocesses before any symbol exists

Command-line options

The options that mean the same thing in both compilers: `-c`, `-d`, `-dll`, `-dylib`, `-e`, `-ex`, `-exx`, `-g`, `-i`, `-l`, `-lib`, `-m`, `-mt`, `-p`, `-s`, `-v`, `-x`, and the extension options `-maxinstdepth`, `-cinclude`, `-cdefine` and `-cimport` (with `-cimport-out`). `-z` is accepted and ignored.

Those spelled differently:

fbc	**tiko**
`-o file` after a source: that source's object file	`-o file` names the program too, when there is no `-x`
`-O 2`	`-O2`, the level joined to the option
`-w all`, `-w pedantic`	`-Werror`, `-Wno-TKOnnnn`
`-maxerr n`	`-fmax-errors n`
`-r`, `-rr`, `-R`, `-RR`	`--emit-ir`, `--emit-asm`, `--keep-temps`
`-C`	`--keep-temps`
`-pp`	none: `--pp-only` preprocesses and reports counts without writing a file

Everything else is `TKO0100: unknown option`, because tiko has one backend, one dialect and its own linker driver: `-a`, `-arch`, `-asm`, `-b`, `-export`, `-forcelang`, `-gen`, `-include`, `-lang`, `-map`, `-nodeflibs`, `-noerrline`, `-nolib`, `-noobjinfo`, `-nostrip`, `-pic`, `-prefix`, `-print`, `-profile`, `-profgen`, `-showincludes`, `-static`, `-strip`, `-sysroot`, `-t`, `-title`, `-vec`, `-Wa`, `-Wc` and `-Wl`. Appendix C lists tiko's own options.

Diagnostics

tiko's messages are its own. Each has a stable id, `TKOnnnn` (Appendix B), and an id is never reused, so a build log stays meaningful. fbc's error numbers survive only where a running program can see them, through `err`: the runtime errors are fbc's, number for number.

The usual mistakes, side by side:

Mistake	fbc	tiko
a variable or procedure never declared	42 Variable not declared	`TKO0743`
a string assigned to a number	181 Invalid assignment/ conversion	`TKO0702`
the wrong number of arguments	1 Argument count mismatch	`TKO0775`
the same name declared twice	4 Duplicated definition	`TKO0600`
a name expected, something else found	14 Expected identifier	`TKO0406`
a block not closed: `if` without `end if`	32 Expected 'END IF'	`TKO0405`
a `for` without `next`	13 Expected 'NEXT'	`TKO0401`
a member the type does not have	18 Element not defined	`TKO0739`
an unknown type name	14 Expected identifier	`TKO0808`
a constant division by zero	100 Division by zero	`TKO0671`
an include file not found	23 File not found	`TKO0301`
`redim` of a fixed-size array	54 Expected var-len array	`TKO0723`
calling something that is not a procedure	10 Expected '='	`TKO0772`
a string initialising a pointer	24 Invalid data types	`TKO0653`
`end function` closing a `sub`	125 Expected 'END SUB'	`TKO0405`
a string returned from a numeric function	181 Invalid assignment/ conversion	`TKO0702`
`exit for` outside any `for`	239 Illegal outside a FOR compound statement	`TKO0947`
`exit for, for` inside only one `for`	247 No outer FOR compound statement found	`TKO0948`
`goto` to a label the procedure does not have	48 Undefined label (104 at module level)	`TKO0949`

Other differences

- **tiko accepts `#define run 5`.** `run` is a runtime procedure (Appendix A), and fbc refuses to define a macro over it, error 4. tiko defines it, and `print run` prints 5.

- **A virtual call's object is evaluated once.** fbc evaluates the object expression of a virtual method call twice: `GetP( )->VM( )` calls `GetP` twice under fbc and once under tiko. Only a program whose object expression has a side effect can tell.
- **A `"""` that ends its line starts a block string** (Chapter 9). In fbc it is a string missing its closing quote, warning 12. `"""abc"""` and `""""` mean what they always meant, in both.
- **`defer` is a contextual keyword**, and so are the other words in Appendix A's list. Where they are not in their statement's position they are ordinary names, as they are in fbc 1.20.
- **Properties do more.** fbc allows at most one index, no `static` property, no property outside a type, and refuses `swap` on a property. tiko allows any number of indices, `static` and namespace properties, auto-properties (`get, private set`), property blocks (Chapter 12), `swap`, and writes a property passed `byref` back through its setter, which fbc does not.
- **The ten fixed-width type names**, `int8` to `float64`, are predeclared typedefs. A program's own declaration of one replaces tiko's.
- **Graphics is Windows-only.** The Linux and macOS toolchains ship no graphics library, so a program that uses `screen` builds on Windows only.
- **tiko targets x86-64 on Windows and Linux, and arm64 on macOS** (`macos-arm64`, Apple Silicon). There it defines `__FB_DARWIN__`, `__FB_UNIX__` and `__FB_ARM__`, and not `__FB_X86__`. `-m32` is `TKO0104`.

What fbc 1.20 does not have

Generics, `for each`, tuples, `defer`, lambdas, `ustring`, the fixed-width type names, `spawn` and channels, `#cinclude`, `#foreignlib`, `#import` and modules, auto-properties and property blocks are tiko's. Code that uses them does not compile under FreeBASIC's own fbc, and that is not a compatibility difference: it is the language this book teaches. A source tree that must build with both tests `#ifdef __TIKO_COMPILER__` and keeps the extensions on one side of it.

Appendix F: Compiler Internals

This appendix is a map of the compiler for readers who want to know what happens between `tiko hello.tiko` and `hello.exe`, or who want to read the source. tiko is written in tiko — `.tiko` sources and `.ti` headers, built by tiko itself — and it lives in one repository, `tiko-lang`, with its runtime, its graphics library and this book. The figures here were measured at the commit the book is pinned to.

The pipeline

```
 source → lexer → preprocessor → parser → Collect → Resolve → Check
→ late checks
                                    |                                  |
                                      AST                             typed
AST + symbols
                                                                       |
     executable ← clang + lld ← clang -c ← LLVM IR ← ABI pass ← MIR
← lowering
```

Each stage is a separate pass over a separate representation, and each one hands the next a finished product: the lexer does not ask the parser anything, and the backend never sees the AST. `-v` shows the last two steps as the commands they are:

```
clang.exe  -target  x86_64-pc-windows-msvc  -c  "hello.exe.1.o.ll"  -o
"hello.exe.1.o" ...
clang.exe -target x86_64-pc-windows-msvc -fuse-ld=lld "hello.exe.1.o" -
o "hello.exe" ... libfb.a ...
```

Lexer (`src/lexer`). Bytes to tokens, one file at a time. The keyword table, `keywords.bi`, is the 247 words of Appendix A. A token is 16 bytes: its kind, flags (did a space precede it, did a macro make it, was it a `:`), a source location, its length in the source and a payload — an interned name or an index into the literal pool. A source file may be UTF-8, with or without a byte-order mark, or UTF-16 or UTF-32 with one; anything else is read as plain bytes. Each file is detected on its own and converted to UTF-8 before it is lexed.

Preprocessor (`src/pp`). Pulls tokens from the lexer and gives the parser one stream: macros expanded, `#include`d files spliced in, `#if` branches chosen. It is also where `#import` is handled: the imported module's **interface** — a `.tki` file, written in tiko source — is spliced in as a frame of its own, with the headers its public declarations need (Chapter 15).

Parser (`src/parser`). Recursive descent, statement by statement, into an **AST** kept in an arena: nodes are indices, not pointers. Operator precedence comes from one table, `prectable.bi`, the only place in the compiler that states it. Nearly every diagnostic in this book with an id in `TKO0400`–`TKO0599` comes from here; the two exceptions are `TKO0409`, which the preprocessor reports, and `TKO0423`, which comes from name resolution.

Collect, Resolve and Check (`src/symbols`, `src/types`). Collect enters every declaration into the symbol table. Resolve binds every name to its symbol and works out every type, instantiating generics as it meets them. Check judges what Resolve found: conversions, overloads, arguments, assignments. A last group of passes — visibility, abstract types, override promises, the rules about what may be initialised how — runs once everything is known. Together they are the largest part of the frontend, and the source of most `TKO06xx`–`TKO08xx` diagnostics.

Lowering to MIR (`src/mir`). MIR is tiko's one intermediate representation: functions of basic blocks, each a list of typed instructions ending in exactly one terminator. Lowering is where FreeBASIC's semantics are eliminated — strings and their descriptors, dynamic arrays, `print`, `for`'s exact iteration rules, constructors and destructors, `defer`, closures, goroutine frames — so that what comes out is plain code over plain values. A verifier checks every function before it goes on. This is the biggest directory in the compiler.

The ABI pass (`src/codegen`). Rewrites each signature and call for the target's calling convention — the Microsoft x64 convention on Windows, System V on Linux, Apple's AAPCS64 on macOS — so the IR that follows has no implicit ABI behaviour: which structures travel in registers, which by hidden pointer, how a function returns one. `--dump-ir` shows MIR after this pass, so what will really be passed is visible without reading LLVM.

LLVM IR and clang. The emitter writes LLVM IR as text, one `.ll` file per unit, and clang compiles it. clang is not bundled: tiko runs whatever `TIKO_CLANG` names, else `C:\Program Files\LLVM\bin\clang.exe` on Windows, `/usr/bin/clang` on Linux or Homebrew's `/opt/homebrew/opt/llvm/bin/clang` on macOS, else the first clang on the `PATH`, and it needs clang 18 or newer. The link goes through clang with lld, or on Linux the system linker when there is no `ld.lld`; on macOS it goes through Apple's linker with the macOS SDK. `-O1` to `-O3` are clang's optimisations; tiko's own optimiser is deliberately thin.

The driver (`src/driver`). Reads the command line, decides what is being built, builds the modules a program imports in dependency order, and runs everything above. A module is compiled once and **cached** — its object and its interface, keyed by its sources, its options and the interfaces of everything it imports — so a second

build that imports `std/containers` does not compile it again. `#cmdline` restarts the driver with the new options. `#cinclude` (Chapter 29) loads libclang the first time a program uses it, to read the C header itself.

Diagnostics (`src/diag`). Every message the compiler can emit is one row in one table, `diagids.bi`, which is what Appendix B is generated from. Nothing else may invent a message, and an id is never reused.

Where the code is

At the pin, counting source files (`.tiko`, `.ti`, and the few `.bas`, `.c` and `.h` files), tests included:

Directory	Files	Lines	Holds
`src/support`, `src/source`	18	2,437	arenas, interning, text, files and source locations
`src/diag`	9	1,945	the diagnostic table and its rendering
`src/lexer`	5	1,979	the lexer and the keyword table
`src/pp`	15	8,402	the preprocessor and `#import`
`src/parser`, `src/ast`	29	12,141	the parser and the AST
`src/symbols`, `src/types`	126	47,419	Collect, Resolve, Check and the late checks; the type table
`src/mir`	89	55,880	MIR and lowering
`src/codegen`	21	6,213	the ABI pass and the LLVM IR emitter
`src/driver`	43	9,980	the driver, modules and linking
`src/build`	8	1,380	`tiko build`
`src/cimport`	12	3,541	`#cinclude`
`src/api`	2	397	the parser library an editor loads
`lib/rt`	363	87,238	tiko-rt, the runtime, and its tests
`lib/gfx`	99	23,450	libtkogfx, the graphics library, and its A/B tools

The compiler is about 152,000 lines; with its runtime and graphics library, about 262,000. The directories are **layered**: a module may include only modules below it, the frontend never includes anything from MIR or the backend, and a lint pass enforces that, along with a limit of 1,500 lines per source file.

The runtime: tiko-rt

A compiled program calls a runtime library for everything the language does that the machine does not: strings, arrays, files, the console, errors, threads. tiko's is **tiko-rt** (`lib/rt`), and it is a reimplementation of FreeBASIC's `libfb` **written in tiko**: 229 `.tiko` and `.ti` files and nine small C files for what the language cannot express — atomics, the jump behind `on error`, the start-up and shutdown hooks, the Microsoft C runtime's start-up glue, the base `Object`'s constructor and type record, and the two port I/O instructions behind `inp` and `out` on Linux.

It is **ABI-compatible with `libfb`**: the same `FBSTRING` and array descriptor layouts, the same `fb_*` function names and signatures. That is what makes tiko's runtime errors fbc's, number for number, and lets an object fbc compiled call into it. The toolchain ships two builds of it, `libfb.a` and the thread-safe `libfbmt.a`, and the driver picks the second by itself when a program creates a thread, whether or not `-mt` was given (Chapter 23). Linux adds position-independent variants for shared libraries. tiko-rt is built with tiko, like the compiler, and is licensed LGPL-2.1-or-later with the static-linking exception: a program tiko compiles may carry any licence.

The graphics library: libtkogfx

libtkogfx (`lib/gfx`) is FreeBASIC's graphics library, `libfbgfx`, reimplemented in tiko: the same `fb_Gfx*` entry points, so it is a drop-in replacement. Its A/B tests (`lib/gfx/tools`) run drawing primitives through both libraries and compare what they drew. It is derived from FreeBASIC's `gfxlib2`, under the same LGPL licence and exception. It is built for windows-x86_64 only, which is why graphics programs build on Windows only (Appendix D).

Self-hosting

tiko compiles itself. `build/bootstrap.sh` (and `bootstrap.bat` on Windows) runs the whole cycle:

```
stage0   the seed builds tiko                    -> stage1
stage1   compiles tiko's own sources             -> stage2
stage2   compiles them again                     -> stage3
```

The **seed** is a packaged tiko compiler built from an earlier commit, which the repository pins; no other compiler takes part. The script asserts five things: that stage1

lowers **every statement** of every module (a statement the compiler declines would otherwise vanish from the compiler it is building); that **stage2 and stage3 are byte-identical files**, with nothing masked, which is a reproducible link as well as a fixpoint; that **stage1 and stage2 are identical too**, because the seed builds stage1 with exactly the command stage1 builds stage2 with; that stage3 compiles a program that prints what the seed's build of it prints; and, with `--gate`, that the test suite fails against stage2 exactly what it fails against stage1. A successful run reports each step (the target line that opens it is left out here):

```
bootstrap: stage1  built by <seed directory> in <t> s, <size> bytes
bootstrap: stage1  lowers every statement of all <n> modules; all verify
bootstrap: stage2  built by stage1 in <t> s, <size> bytes
bootstrap: stage3  built by stage2 in <t> s, <size> bytes
bootstrap: fixpoint  stage1 == stage2 as well: the seed reproduces this
commit's compiler
bootstrap: fixpoint  stage2 == stage3, sha256 <first 16 hex digits>...
bootstrap: stage3  builds hello.tiko, and it prints what the seed's build
prints
bootstrap: ok
```

A fixpoint proves the compiler reproduces itself. It proves nothing about code the compiler's own source never uses, which is why the last check runs the suite.

How it is tested

- **The test suite**: 2,046 tests at the pin, each a small program with its expected diagnostics, IR or output. A fix lands with the test that would have caught it.
- **fbc's own test corpus**, vendored in `tests/vendor/` and never edited, is the acceptance gate: every file's diagnostics are recorded, and a change that moves any of them is examined before it lands.
- **Differential probes.** Every compatibility fact in the compiler — and in this book — was established by compiling a program with fbc and with tiko and comparing the results. The source carries the measurement beside the code that depends on it, so a later change has to argue with it.
- **This book's listings**, compiled and run on both targets against the pinned toolchain before each release of the book.

Appendix G: Solutions

75 of the book's 202 exercises have a worked solution. Each is a program in the companion examples, beside the chapter's listings, under `examples/chNN/solutions/`, with its expected output in a `.out` file next to it. The solutions are not printed here: the listing harness compiles and runs every one of them on both targets, exactly as it does the chapters' listings, so the files are the version that is known to work.

A solution is one way to do the exercise, not the only one. Several build on a chapter's own listings or modules and import them, and run from their `solutions/` directory. A solution marked *blocked* depends on a compiler fix that the toolchain this book is checked against does not have yet; its `.out` file is the result it gives once the fix is in.

Exercise	Chapter	Task	Solution
1.1	Getting Started	Write a program that reports which compiler built it and which version that compiler is.	`ch01/solutions/ex01-01.tiko`
1.2	Getting Started	Make a program greet you by name and then exit with status 3, and check the status from your shell (`echo $?`, or `echo %ERRORLEVEL%` in `cmd`).	`ch01/solutions/ex01-02.tiko`
1.3	Getting Started	Call a procedure that was never declared, and note the identifier of the diagnostic.	`ch01/solutions/ex01-03.tiko`
3.1	A Tour of tiko	Report the longest word in the file as well as the most common ones.	`ch03/solutions/ex03-01.tiko`
3.2	A Tour of tiko	List the words that appear exactly once, in alphabetical order.	`ch03/solutions/ex03-02.tiko`
4.1	Types, Variables and Constants	Print the largest value each signed integer type can hold, then add one to the `int32` maximum and explain what you see.	`ch04/solutions/ex04-01.tiko`
4.2	Types, Variables and Constants	Build a table comparing `cint`, `int` and `fix` over 2.4, 2.5, 2.6 and their negatives.	`ch04/solutions/ex04-02.tiko`
5.1	Operators and Expressions	Build a permissions value from three named bit positions, then read one back.	`ch05/solutions/ex05-01.tiko`
[illegible]	[illegible]	[illegible]	`ch[illegible]/solutions/ex[illegible]-02.tiko`
[illegible]	Preprocessor	[illegible] of `amount`,	[illegible]