Programming in Goose

Memory-safe systems programming without a heap

Written for Goose v2026.3, October 2026

Contents

Part I: Getting Started 14

1 Introduction 15
- 1.1 Why learn Goose? 16
- 1.2 What Goose is not 18
- 1.3 A first taste 19
- 1.4 How to read this book 20
- 1.5 About the examples 21

2 Hello, Goose 23
- 2.1 Getting the compiler 23
- 2.2 The first program 24
- 2.3 Two ways to run a program 25
- 2.4 Checking and formatting 26
- 2.5 Printing 27
- 2.6 Comments 28
- 2.7 When something is wrong 29
  - 2.7.1 *Release and debug* 30
- 2.8 Program arguments 31
- 2.9 Exercises 32

3 Values and Types 33
- 3.1 Declaring variables 33
  - 3.1.1 *Assigned exactly once* 35
- 3.2 Integers 36
  - 3.2.1 *Literals adapt* 36
  - 3.2.2 *Arithmetic happens at the operands' width* 37

3.2.3 *Conversions* 39
3.2.4 *`%` is Euclidean* 40
3.2.5 *Overflow* 42
3.3 Floats 43
3.4 Booleans 44
3.5 Bitwise operators 44
3.6 Named constants and type aliases 45
3.7 Exercises 46

4 Control Flow as Expressions 48
4.1 `if` 48
4.1.1 *Statements that end with a block* 49
4.2 `match` 50
4.3 Blocks are expressions too 52
4.3.1 *`block` and `break` with a value* 52
4.4 `guard` 53
4.5 Loops 54
4.5.1 *`for`* 54
4.5.2 *`while` and `loop`* 56
4.6 `defer` 57
4.7 Exercises 59

5 Functions 60
5.1 Declaring a function 60
5.1.1 *Parameters are `lets`* 61
5.2 Reference parameters 61
5.3 Default arguments 63
5.4 Several results 63
5.5 Overloading 65
5.6 Method-call syntax 66

5.7 Nested functions 67
5.8 Recursion is declared 68
5.9 Functions that never return 69
5.10 Exercises 70

Part II: Memory Without a Heap 71

6 Where Values Live 72
6.1 Three kinds of size 72
6.2 Data stacks 73
6.3 Leaving a scope frees everything 75
6.4 Nothing moves 76
6.5 Copies are visible 78
6.6 Built in place 79
6.7 Globals 81
6.8 Design note: why no heap? 82
6.9 Exercises 83

7 The Array Family 84
7.1 The family 84
7.2 Fixed arrays: `T[k]` 86
7.3 Frozen arrays: `T[]` 87
7.4 Limited arrays: `T[..k]` and `T[..]` 88
7.5 Grow-only arrays: `T[>..]` 89
7.5.1 *Grow-only does not mean "never shrinks"* 90
7.6 Grow-shrink arrays: `T[>..<]` 91
7.7 What can go inside what 92
7.7.1 *Sequential arrays* 93
7.8 Array literals 94

7.9 Members 94
7.10 Choosing an array 96
7.11 Exercises 96

8 References and Slices 98
8.1 References 98
8.1.1 Where references come from 100
8.2 Slices 101
8.3 The one rule 103
8.3.1 Roots 104
8.3.2 A function that returns one of its arguments 106
8.4 Liveness, not scope 107
8.5 Writability 109
8.6 Rebinding 110
8.7 Exercises 111

9 Strings Are Bytes 112
9.1 Four kinds of string 112
9.2 Building text 114
9.2.1 Literals and escapes 114
9.3 Storing text 115
9.4 Printing values inside text 116
9.5 Cutting text up 118
9.6 Converting to and from numbers 119
9.7 Comparing and changing text 120
9.8 Bytes and characters 121
9.9 Exercises 121

10 Structs and Inline Data 123

10.1 Declaring and building structs 123
10.1.1 *Structs are nominal* 125
10.2 Packed layout 125
10.3 Field qualifiers 127
10.4 Variable-size fields 127
10.5 A growable tail 130
10.6 Generic structs 131
10.7 Arithmetic and equality 131
10.8 Exercises 132

11 Optionals 133
11.1 A reference that may be null 133
11.2 Narrowing 135
11.2.1 *Narrowing ends when you rebind* 137
11.3 Optionals as fields 138
11.4 Optionals or a trailing `bool?` 139
11.5 Exercises 140

Part III: Modelling Data 141

12 Enums in Two Sizes 142
12.1 Declaring enums 142
12.2 Matching 143
12.3 Fixed mode and variable mode 145
12.4 Case functions 149
12.5 Design note: why two modes? 151
12.6 Exercises 152

13 Links Without Pointers 153
13.1 Self-relative links 153
13.1.1 *Same root, exactly* 156

13.1.2 *Values with self-relative links are not copied* 156
13.2 Links into a pool 157
13.3 `self`: links that point back 159
13.4 An index for a sequential array 160
13.5 Choosing a link 161
13.6 Exercises 161

14 Pools for Lifetimes That Don't Nest 163
14.1 A pool of slots 163
14.2 Freeing frees nothing 164
14.3 A waiting list 166
14.4 Local pools and pool parameters 168
14.5 Slice pools 169
14.6 When to use a pool 170
14.7 Exercises 170

15 Errors 172
15.1 Two outcomes: a trailing `bool` 172
15.2 Found or not: an optional 173
15.3 Failing from deep inside: `return … from` 173
15.3.1 *How it works, and what it costs* 175
15.3.2 *It is checked at compile time* 177
15.4 The `exception` module 177
15.5 Stopping the program 178
15.6 Choosing 180
15.7 Exercises 180

Part IV: Abstraction and the Library 181

16 Generics and Blocks **182**
16.1 Generic functions 182
16.2 Function values 184
16.3 Trailing blocks 185
16.3.1 `return` inside a block 186
16.4 The library's higher-order functions 187
16.5 Nested functions as values 189
16.6 Exercises 190

17 Recursion **191**
17.1 The rule 191
17.1.1 Why 192
17.2 Pattern 1: finish with the scratch before recursing 193
17.3 Pattern 2: pass the output in 194
17.4 Pattern 3: a context of references 195
17.4.1 Sharing a scratch stack 196
17.5 Mutual recursion 197
17.6 Design note: why recursion is marked 198
17.7 Exercises 198

18 A Tour of the Standard Library **200**
18.1 Importing modules 200
18.2 `std`: arrays 201
18.3 `std`: numbers, hashing and randomness 202
18.4 `dictionary` 203
18.5 `math` and `vec` 206

18.6 `os`: files, arguments and time 206
18.7 `csv` 207
18.8 `json` 208
18.9 `regex` 211
18.10 The rest 212
18.11 Exercises 213

19 Programs, Namespaces and Packages 214
19.1 Whole-program compilation 214
19.2 Splitting a program into files 215
19.3 Namespaces 215
19.4 `main` and globals 217
19.5 Packages 218
19.5.1 Blessed packages 219
19.6 Embedding files 219
19.7 A program with C of its own 220
19.8 Exercises 220

Part V: Systems Work 222

20 Threads That Share Nothing 223
20.1 A worker is a separate program 223
20.2 Typed queues 225
20.3 What can cross: flat values 226
20.4 Workers that fail 227
20.5 Design note: no shared memory 228
20.6 Exercises 229

21 Saving and Loading Bytes 230
21.1 Three built-ins 230

21.2 Verification 232
21.3 Variable-size records 234
21.4 What cannot be saved 235
21.5 Writing without copying 235
21.6 Exercises 236

22 Talking to C 237
22.1 `extern fn` 237
22.2 What can cross 238
22.3 A program's own C 239
22.4 Shared libraries, bound at startup 241
22.5 Calling Goose from C 243
22.6 Exercises 244

23 Performance 246
23.1 Measure a real build 246
23.2 Bounds checks 247
23.2.1 What the analysis knows 248
23.2.2 How much a check costs 250
23.3 Index types 250
23.4 `checkfree`: no checks allowed 252
23.5 Asking for inlining and vector code 254
23.6 Data layout 255
23.7 Design note: safe by default, fast on request 259
23.8 Exercises 259

Part VI: Two Programs 261

24 A Word Counter 262

24.1 The program 262
24.2 One buffer, no copies 265
24.3 Counting with `update` 266
24.4 Ranking 266
24.5 The output loop 267
24.6 Errors and exit codes 268
24.7 How fast is it? 268
24.8 Exercises 269

25 A Calculator 270
25.1 The program 270
25.2 One function per rule 275
25.3 Nested functions instead of a parser object 276
25.3.1 Marking the recursion 277
25.4 Errors from anywhere 278
25.5 Where the data lives 279
25.6 Limits 280
25.7 Exercises 281

Appendix A: Goose for C, Odin and Rust Programmers 282
A.1 From C 283
A.2 From Odin 285
A.3 From Rust 288

Appendix B: Limitations 291
B.1 Limits of the design 291
B.2 Limits at the boundaries 293
B.3 Omissions of version 1 294

Appendix C: Solutions to the Exercises **295**
C.1 Chapter 2: Hello, Goose 295
C.2 Chapter 3: Values and Types 296
C.3 Chapter 4: Control Flow as Expressions 297
C.4 Chapter 5: Functions 298
C.5 Chapter 6: Where Values Live 300
C.6 Chapter 7: The Array Family 301
C.7 Chapter 8: References and Slices 303
C.8 Chapter 9: Strings Are Bytes 304
C.9 Chapter 10: Structs and Inline Data 306
C.10 Chapter 11: Optionals 307
C.11 Chapter 12: Enums in Two Sizes 309
C.12 Chapter 13: Links Without Pointers 310
C.13 Chapter 14: Pools for Lifetimes That Don't Nest 313
C.14 Chapter 15: Errors 316
C.15 Chapter 16: Generics and Blocks 319
C.16 Chapter 17: Recursion 320
C.17 Chapter 18: A Tour of the Standard Library 322
C.18 Chapter 19: Programs, Namespaces and Packages 324
C.19 Chapter 20: Threads That Share Nothing 326
C.20 Chapter 21: Saving and Loading Bytes 328
C.21 Chapter 22: Talking to C 330
C.22 Chapter 23: Performance 331

C.23 Chapter 24: A Word Counter 335
C.24 Chapter 25: A Calculator 338

PART I

Getting Started

The first five chapters cover the parts of Goose that will feel familiar if you have written C, Odin or Rust: running programs, the scalar types, control flow and functions. Goose has its own opinions even here (about integer widths, about `%`, about what a block evaluates to), so even the familiar chapters are worth reading closely.

Chapter 1

Introduction

Every systems language has to answer one question before any other: when a program needs memory whose size it does not know in advance, where does that memory come from, and who gives it back?

C answers with `malloc` and `free`, and trusts you. Rust answers with ownership: every heap block has exactly one owner, borrows are checked against lifetimes, and the block is freed when its owner is dropped. Odin answers with allocators. Every allocation goes through an explicit allocator, the `context` carries a default one, and experienced Odin programmers reach for arenas so that a whole phase of a program can be freed at once.

Goose gives an answer that none of these languages give: **there is no heap**. There is no `malloc`, no allocator interface, no garbage collector, no reference counting and no destructors. Every value whose size is not known at compile time lives on one of a small number of *data stacks* that the compiler manages for you. Memory is taken from a data stack by bumping a pointer, and it is given back, all at once, when the scope that owns it ends. The compiler proves that no reference ever outlives what it

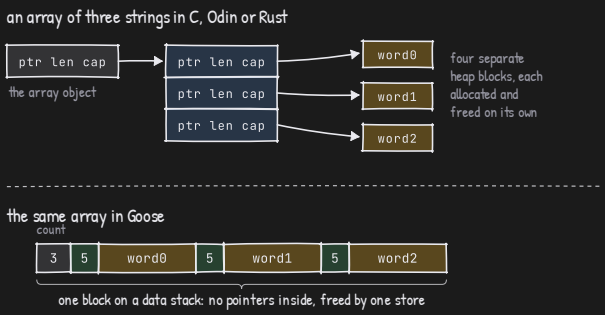
points at, and it does so without a single lifetime annotation in the source.

That sounds like a restriction, and in some ways it is. This book is about what you get in exchange, and how to write programs that work with the memory model rather than against it.

> **From Odin**
>
> If you have ever written `free_all(context.temp_allocator)` at the end of a frame, you already know the idea of throwing a whole region of memory away in one operation. Goose turns that into the only way memory is freed, and has the compiler work out where the regions are.

Why learn Goose? 1.1

The most visible reason is speed, which comes from data layout more than from anything clever in the compiler. Look at what an array of three short strings is in most languages:



In most languages an array of strings is a tree of heap blocks. In Goose it is one contiguous block.

In C, Odin or Rust the array is a small header pointing at a buffer of string headers, and each of those points at its own characters somewhere else on the heap. Building it costs four allocations, walking it touches four unrelated places in memory, and destroying it costs four frees.

In Goose the same array is a single run of bytes: a count, and then each string's length followed by its characters. It was built by bumping a pointer three times, it is walked from front to back, and it disappears when its owner's scope ends, with no per-string work at all. There is also nothing inside it that points elsewhere, which turns out to matter a great deal: such a block can be copied with `memcpy`, written to a file as it is, or handed to another thread as a single message.

The second reason is safety. Goose is memory-safe in the same sense as safe Rust: no use-after-free, no dangling references, no out-of-bounds access, no data races. But it reaches that point by a different route. Because nothing is ever freed except by a scope ending, the lifetime question the compiler has to answer is always the same simple one: does this reference outlive the variable that owns its target? There are no lifetime parameters to write, no borrow-exclusivity rules to satisfy, and two references to the same thing are perfectly fine.

The third reason is that the language is small and pleasant to write. It has C-family syntax, type inference, generics without trait bounds, `match` over algebraic data types, several return values, `defer`, and blocks that are expressions. Much of it will feel like a cross between Odin and Rust, and the parts that do not are the parts this book spends most of its time on.

What Goose is not 1.2

It is worth being clear about the trade-offs before you invest time in them.

Goose compiles a **whole program at once**, to a single C file, and then builds that with a C compiler. There is no separate compilation and there are no Goose shared libraries. Functions can be passed to other functions, but only at compile time: there are no function pointers, no closures that escape, and no dynamic dispatch except through the tag of an enum. And you have to think about *where data lives*. Most of the time the answer is "in a local of the function that builds it" and there is nothing more to think about, but when data has lifetimes that do not nest neatly, you will need one of the tools from Part III.

The language is also young. This book was written against one specific build of the compiler, and some details will have changed by the time you read it. The ideas will not.

A first taste 1.3

Here is a complete Goose program. Do not worry about the details yet; every line of it will be explained in the coming chapters.

```
struct Goose { name: u8[..12], honks: i32 }

fn main() {
    var flock: Goose[>..] = [];
    let leader .= flock.push(Goose { name:
        "Ada", honks: 3 });
    for i in 1..1000 {
        flock.push(Goose { name: "", honks: 0
            });
    }
    leader.honks += 1;      // still refers to
        flock[0]

    var names: u8[][>..] = [];
    for i in 3 { names.push(str("goose #", i +
        1)); }

    print(flock.len, " geese, led by ",
        flock[0].name, " with ", flock[0].honks,
        " honks");
    print(names, " in ", bytes_of(names).len, "
        bytes");
}
```

```
1000 geese, led by Ada with 4 honks
["goose #1", "goose #2", "goose #3"] in 36 bytes
```

A few things stand out even at a glance:

- `Goose[>..]` is a *grow-only array*. It is the closest thing Goose has to a `std::vector`, an Odin `[dynamic]T` or a Rust `Vec<T>`, but it never moves its elements. That is why `leader`, a reference to the first element, is still valid after 999 more pushes.

- `.=` binds a reference. A plain `=` would have copied the element.
- `u8[..12]` is a string that lives *inside* the struct, with room for 12 bytes. There is no pointer to a separately allocated string.
- `names` holds three strings in 36 bytes: for each one, a 4-byte length and its 8 characters, packed one after the other.
- There is no `free`, no `defer delete(...)` and no `drop`. The two arrays disappear when `main` returns.

> **From Rust**
>
> The reference `leader` points into `flock`, and the program then pushes into `flock` a thousand times. In Rust that is the classic error: you cannot call `push` while a borrow of the vector is alive, because `push` might reallocate. In Goose it is fine, because a grow-only array never reallocates. The compiler still checks that `leader` does not outlive `flock`.

How to read this book 1.4

The book assumes that you are comfortable with C and know at least one of Odin or Rust. It does not explain what a pointer, a struct or a stack frame is; it explains how Goose's versions of them differ from the ones you know. When a difference is mostly interesting to readers coming from one particular language, it is in a margin note labelled **From C**, **From Odin** or **From Rust**. Unlabelled notes are asides that anyone can skip on a first reading.

The book is in six parts:

1. *Getting Started* covers running programs, scalars, control flow and functions.
2. *Memory Without a Heap* is the heart of the book: data stacks, the array family, references and slices, strings, structs and optionals.
3. *Modelling Data* covers enums, links between values, pools for data with irregular lifetimes, and error handling.
4. *Abstraction and the Library* covers generics, recursion, the standard library and packages.
5. *Systems Programming* covers threads, serialization, calling C, and performance.
6. *Projects* puts it all together in two larger programs.

Most chapters end with a few exercises. Solutions are in Appendix C.

About the examples 1.5

Every example in this book was compiled and run with the prebuilt Goose compiler of release `v2026.3`, from October 2026, apart from a few that need a feature added since, which are marked where they appear. Where an example prints something, the output shown below it is exactly what that compiler printed. Where an example is supposed to *fail* to compile, the error shown is the one the compiler gave. Short examples usually show only the interesting lines;

the surrounding `fn main() { … }` is left out when it adds nothing.

If an example does not work for you, the most likely reason is that the language has moved on. The official documentation, in the `docs/` folder of the repository, describes the current state of things and should be trusted over this book.

Chapter 2

Hello, Goose

This chapter gets a compiler onto your machine, runs a first program, and looks at the handful of commands you will use every day. By the end of it you will know how Goose programs are compiled, which is a little different from what you are used to.

Getting the compiler 2.1

The Goose compiler is a single executable called `goose`. The project publishes prebuilt compilers for Windows, Linux and macOS on its releases page: unpack one somewhere and add that directory to your `PATH`.

You can also build it from source. You need CMake, a C++20 compiler and Python 3:

```
git clone https://github.com/aardappel/goose.git
cd goose
cmake -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build
```

That leaves the compiler at `build/goose`. The standard library is not a separate download: the compiler finds it in the `stdlib/` folder of the source tree it came from.

To check that everything works, ask for the version:

```
$ goose version
goose runtime 4d3da949e563
```

> The examples in this book were checked against release `v2026.3`, whose `goose version` prints the line above. A compiler built from source names the commit it was built from as well.

The first program 2.2

Create a file called `hello.goose`:

```
fn main() {
    print("Hellope from Goose!");
}
```

and run it:

```
$ goose hello.goose
```

```
Hellope from Goose!
```

> Each time it compiles a program, the `v2026.3` compiler also prints a three-line summary of what it did on stderr, starting `parsed ok:`. The transcripts in this book leave it out. Chapter 23 shows what it says.

That is the whole ceremony. There is no project file, no `package` declaration and no build directory to set up. Let's go through the program line by line anyway, because even three lines have something to say.

`fn main() { … }` declares the program's entry point. Functions are declared with `fn`, parameters go in the parentheses, and a result

type would follow a `->`. Like Odin and Rust, and unlike C, there is no header and no forward declaration: top-level declarations can appear in any order.

`print("Hellope from Goose!");` calls the built-in `print`, which writes its arguments followed by a newline. The string literal is a run of bytes in the program's read-only data. Goose has no string type; you will see in Chapter 9 what a string literal really is.

The semicolon ends the statement. Goose needs semicolons after expression statements, but not after a statement that ends with a block, such as an `if` or a loop.

> **From Odin**
>
> Yes, "Hellope" is a nod to Odin's traditional first program. Goose has its own word for it, which you will meet shortly.

Two ways to run a program 2.3

The command you just ran compiled the program and ran it *inside the compiler's own process*. Goose compiles every program to C, and the compiler carries a small C compiler of its own, TinyCC, which turns that C into machine code in memory and calls `main`. That makes `goose file.goose` (or `goose run file.goose`, which is the same thing) about as quick as running a script, and you do not need any other tools installed.

When you want a real executable, use `goose build`:

```
$ goose build hello.goose
built hello
$ ./hello
Hellope from Goose!
```

`build` writes the same C, but hands it to the optimizing C compiler it finds on your machine (Clang, GCC or MSVC), and puts the executable next to the source file. This is what you use for anything you want to ship or measure, and it is required for programs that use threads, which the in-process backend cannot run.

You can also stop one step earlier and look at the C:

```
$ goose -o hello.c hello.goose
```

The result is a single C file holding the whole program. It is readable, but it is not meant to be edited; think of it as the compiler's assembly output.

> **From C**
>
> Because the output is C, a Goose program can call C functions directly, and C can call Goose. Chapter 22 covers both directions.

Checking and formatting 2.4

Two more commands are worth knowing from the start.

`goose check file.goose` parses and type-checks a program and everything it imports, without generating or running anything. Editors use the same checks through `goose lsp`, the compiler's language server.

`goose format file.goose` rewrites a file in the standard layout: indentation, the spacing between tokens, blank lines and trailing commas. It keeps your line breaks where they are. `goose format --check` writes nothing and only lists the files that are not formatted, which is handy in continuous integration.

Printing 2.5

You will see `print` in almost every example, so it is worth knowing exactly what it does. It takes any number of arguments, of any types, renders each one as text, and writes them all with a newline at the end. It inserts nothing between the arguments, so you add your own spaces:

```
fn main() {
    let geese = 12;
    let average_weight = 4.25;
    print("geese: ", geese, ", average weight:
        ", average_weight, " kg");
    print("more than ten? ", geese > 10);
}
```

```
geese: 12, average weight: 4.25 kg
more than ten? true
```

There is no format string, so there is nothing to get wrong about `%d` versus `%ld`. Every value can

be printed, not just numbers and strings: arrays print in brackets, structs print as their type name and fields. That makes `print` the quickest debugging tool you have.

```
struct Egg { weight: f32, day: i32 }

fn main() {
    let nest = [Egg { 61.5, 3 }, Egg { 72.0, 4
        }];
    print(nest);
}
```

```
[Egg { 61.5, 3 }, Egg { 72.0, 4 }]
```

Each call to `print` is a single write to standard output, so lines printed by different threads never get mixed together. `print` has two relatives that you will meet in Chapter 9: `str`, which builds the same text as a new string, and `format`, which appends it to an existing one.

Comments 2.6

Line comments start with `//`. Block comments are `/* … */`, and unlike in C they nest, so you can comment out a region that already contains a block comment:

```
fn main() {
    /* print("not today");
       /* a comment inside a comment */
       print("nor this"); */
    print("honk");   // only this line runs
}
```

```
honk
```

When something is wrong 2.7

Misspell a function name and the compiler tells you where:

```
fn main() {
    prnt("Hellope from Goose!");
}
```

```
hello.goose:2: error: unknown function: prnt
    prnt("Hellope from Goose!");
  in fn main() instantiated from hello.goose:1
```

The last line is the first sign of something you will see a lot. Goose type-checks generic code only where it is used, with concrete types, so every error comes with the chain of calls that led to it. For a mistake in `main` the chain is one line long. For a mistake deep inside a generic library function, the chain tells you which of your calls caused it.

Some mistakes can only be caught while the program runs. An index out of range is one of them, and Goose always checks:

```
fn main() {
    let eggs: i64[3] = [61, 72, 58];
    var i = 0;
    for day in 5 { i += 1; }
    print(eggs[i]);
}
```

```
goose runtime error: index 5 out of bounds
    (length 3) (hello.goose:5)
```

The program stops with a message naming the line. It never reads past the end of the array. As

you will see in Chapter 23, the compiler removes most of these checks from the compiled program when it can prove they cannot fail.

Release and debug 2.7.1

By default a program is compiled for release, which has one effect you should know about early: **signed integer overflow wraps around**. Here an 8-bit counter is pushed past its maximum of 127:

```
fn main() {
    var honks: i8 = 120;
    for i in 10 { honks += 1; }
    print(honks);
}
```

```
-126
```

Compile with `--debug` and the same program stops at the overflow instead, naming the operation and the line:

```
fn main() {
    var honks: i8 = 120;
    for i in 10 { honks += 1; }
    print(honks);
}
```

```
goose runtime error: integer overflow (debug):
    127 + 1 at i8 (hello.goose:3)
```

`--debug` also checks conversions that lose information. Use it while you develop and test. Unsigned arithmetic is different: it wraps in

every build, because that is what unsigned arithmetic is for. Chapter 3 has the details.

> **From Rust**
>
> This is the same split as Rust's debug and release profiles: overflow panics in debug and wraps in release. The difference is that in Goose, unsigned types always wrap, by definition, so hash functions and random number generators need no `wrapping_mul`.

Program arguments 2.8

A program's command-line arguments come from the `os` module in the standard library. `import os;` makes its functions available, and `args()` returns all the arguments as an array of strings, with the program itself first. When you run a program in-process, everything after `--` is passed to it:

```
import os;

fn main() {
    let a = args();
    print(a.len, " arguments");
    for x in a[1..] { print("  ", x); }
}
```

```
$ goose args.goose -- one "two three"
```

```
3 arguments
  one
  two three
```

`a[1..]` is a *slice* of the array that leaves out the first element. Slices get a whole chapter later on.

To end a program early with an exit code, call `exit(code)`; it is a built-in and needs no import.

Exercises 2.9

1. Write a program that prints the numbers from 1 to 5, each on its own line, followed by a line that says `liftoff`. You will need `for i in n`, which counts from 0 up to `n - 1`.
2. Run the overflow example above with a `u8` counter that starts at 250 instead of an `i8` that starts at 120, with and without `--debug`. What happens, and why is it the same in both builds?

Chapter 3

Values and Types

This chapter covers the building blocks: variables, the scalar types, and the rules of arithmetic. Most of it will look familiar, but Goose is stricter than C about integer widths and looser than Rust about literals, and it has one rule about `%` that is different from all three of the languages you know. Those are the parts to read carefully.

Declaring variables 3.1

Goose has three ways to declare a variable:

```
fn main() {
    let flock_size = 24;        // cannot be reassigned
    var hatched: i32 = 0;       // can be reassigned
    const pens = [4, 8, 12];    // cannot be reassigned or written into
    hatched += 3;
    hatched++;
    print(flock_size, " ", hatched, " ", pens);
}
```

```
24 4 [4, 8, 12]
```

The type comes after the name, separated by a colon, and it can almost always be left out: an un-annotated declaration takes the type of its initializer. `hatched++` is a statement, not an expression, so `x = y++` is not something you can

write. That removes a whole class of puzzles about evaluation order.

The difference between `let` and `const` deserves a moment. `let` stops you reassigning the *name*, but the contents of a `let` array or struct are as writable as their type allows. `const` makes the contents read-only too:

```
fn main() {
    let shelf = [1, 2, 3];
    shelf[0] = 10;              // fine: the
        elements of a let are writable
    print(shelf);
}
```

```
[10, 2, 3]
```

```
fn main() {
    const shelf = [1, 2, 3];
    shelf[0] = 10;
}
```

```
error: cannot assign through this path (const,
    or a read-only instantiation, §9.5)
```

From Rust

Goose's `let` is closer to a Rust `let` holding a `&mut`: the binding is fixed, the thing it names is not. There is no `mut` on bindings. Read-only contents come from `const`, which is part of the *type*, and you will see it again on references and slices.

Reassigning a `let` is an error that names the rule it breaks:

```
fn main() {
    let count = 3;
    count = 4;
}
```

```
error: cannot assign to let count (§4.4)
```

The `§4.4` refers to the section of the language specification that has the rule. You will see these references in many error messages.

Assigned exactly once 3.1.1

A `let` can be declared without an initializer, as long as it is assigned exactly once on every path before it is read. The compiler checks this:

```
fn main() {
    let honks = 3;
    let verdict: i64;
    if honks > 2 { verdict = 1; } else { verdict
        = 0; }
    print(verdict);
}
```

```
1
```

Reading a variable that might not have been assigned is an error, for `let` and `var` alike. Goose never gives you an uninitialized value:

```
fn main() {
    let x: i64;
    print(x);
}
```

```
error: variable x may be used before it is
    assigned
```

Integers 3.2

The integer types are `i8`, `i16`, `i32` and `i64`, and their unsigned counterparts `u8` to `u64`. There is no `int`, `usize` or `size_t`. By convention `i64` is the type for counts, sizes and indices, and it is the type that `.len` returns and that an unconstrained integer literal becomes.

The narrower types are for data: fields and array elements that should be compact, and loops that need a particular width.

Literals adapt 3.2.1

An integer literal has no type of its own. It takes whatever integer type its context needs, provided its value fits:

```
fn main() {
    let weight: u8 = 200;       // 200 fits a u8
    let total = 42;             // nothing says
        otherwise: an i64
    let sum = weight + 50;      // 50 adapts to
        u8, so this is a u8 add
    print(weight, " ", total, " ", sum);
}
```

```
200 42 250
```

A literal that does not fit is a compile error, not a silent truncation:

```
fn main() {
    let x: u8 = 256;
}
```

```
error: constant 256 does not fit u8
```

Integer literals can be written in decimal or hexadecimal (`0xFF`). There are no binary literals; use hexadecimal or a shift. A character literal such as `'A'` is just an integer, the value of that byte, so `'A' + 1` is 66.

Arithmetic happens at the operands' width 3.2.2

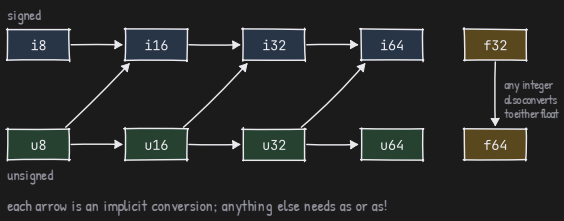
This is the rule that matters most: **an operation is performed at the width of its operands**. Adding two `u8` values is an 8-bit addition. Multiplying two `i32` values is a 32-bit multiplication. Nothing is promoted to a wider type behind your back.

> **From C**
>
> In C, `u8 + u8` is an `int` addition because of the integer promotions, and the result only wraps when you store it back into a `uint8_t`. In Goose the addition itself wraps. If you want the wide result, widen an operand first.

When the two operands have different types, Goose looks for one type that both can be converted to *without losing information*:

- If one operand is a literal, it adapts to the other operand's type.
- If one type widens into the other, the wider one is used: `u8 + i64` is an `i64` addition.
- An integer meeting a float becomes that float type.
- Anything else is an error.

These are the implicit widenings:



Implicit conversions between scalar types. Narrowing, changing sign, and converting a float to an integer all need a cast.

An unsigned type widens into a *strictly wider* signed type, because every `u8` value fits in an `i16`. But a `u32` and an `i32` have no common type that holds every value of both without going to 64 bits, and Goose will not silently pick one for you:

```
fn main() {
    let eggs: u32 = 7;
    let change: i32 = -3;
    print(eggs + change);
}
```

```
error: operands of + have no common type: u32
    and i32 (convert one with `as`)
```

The fix is to say what you mean, for example `eggs as i64 + change`, which converts one operand to `i64` and lets the other widen to meet it.

Because literals adapt to the other operand, you will occasionally be surprised in the other direction:

```
fn main() {
    let small: u8 = 200;
    let total = small + 1000;
}
```

```
error: constant 1000 does not fit u8
```

The literal tried to become a `u8` to match `small`. Write `small as i64 + 1000`, or store `small` in an `i64` first.

Conversions 3.2.3

A conversion that might lose information is never implicit. Narrowing, changing signedness, and converting a float to an integer all need a cast:

```
fn main() {
    let big: i64 = 1000;
    let small: i32 = big;
}
```

```
error: expected a value of type i32, got i64
    (narrowing and sign changes require an
    explicit `as`)
```

There are two cast operators:

- `x as T` converts, and in a debug build checks that the value survives. If it does not, the program stops. In a release build it truncates, wraps or rounds toward zero without checking.
- `x as! T` converts without checking, in every build. Use it when losing bits is the point, such as taking the low byte of a hash.

```
fn main() {
    let weight = 4.75;
    let whole = weight as i64;          //
        rounds toward zero: 4
    let big = 300;
    let low_byte = big as! u8;          // 300 -
        256 = 44, deliberately
    let neg = -1;
    print(whole, " ", low_byte, " ", neg as!
        u32, " ", -2.7 as i64);
}
```

```
4 44 4294967295 -2
```

The compiler also tells you when a cast is unnecessary, which happens more often than you would think once you are used to C:

```
fn main() {
    let b: u8 = 7;
    let w: i64 = b as i64;
    print(w);
}
```

```
warning: redundant `as i64`: a u8 converts to
    i64 implicitly here
```

```
7
```

A cast that *chooses* something is not redundant and gets no warning. For example, `let id = 5 as u32;` picks a type that nothing else gives, and `(a as i64) * b` with `i32` operands picks the width of the multiply.

`%` is Euclidean 3.2.4

Here is the rule that differs from C, Odin and Rust alike. In Goose, **the result of `%` is never**

negative. For any `a` and any non-zero `b`, `a % b` is in the range `0` up to `|b| - 1`:

```
fn main() {
    print(-7 % 3, " ", 7 % -3, " ", -7 / 3);
}
```

```
2 1 -2
```

In C, `-7 % 3` is `-1`. The Goose rule is the one you actually want when you use `%` to wrap an index, because `x % n` is then a valid index into an array of length `n` for *every* `x`, negative or not. It also means the compiler can drop the bounds check on `arr[x % arr.len]`, as Chapter 23 shows.

Division still truncates toward zero, as in C, so `/` and `%` no longer pair up for negative numbers. The `std` module provides the two missing functions: `rem(a, b)` is the C-style remainder that pairs with `/`, and `div_euclid(a, b)` is the quotient that pairs with `%`.

```
import std;

fn main() {
    print(rem(-7, 3), " ", div_euclid(-7, 3));
    // which 10-wide pen is position -13 in, and
        where in it?
    let pos = -13;
    print("pen ", div_euclid(pos, 10), ", offset
        ", pos % 10);
}
```

```
-1 -3
pen -2, offset 7
```

On floats, `%` is C's `fmod`, so `-7.5 % 2.0` is `-1.5`. The Euclidean rule is for integers only.

> **From Odin**
>
> Odin has both: `%` truncates like C, and `%%` is the floored modulo. Goose has only one operator, and it is the Euclidean one. For a positive divisor, which is the usual case, Euclidean and floored modulo agree.

Overflow 3.2.5

What happens on overflow depends on signedness:

- **Unsigned arithmetic wraps, in every build.** This is the definition of unsigned arithmetic in Goose, not an accident of the hardware, so hash functions and random number generators can rely on it.
- **Signed overflow is a bug.** A debug build stops with a message, and a release build wraps.

Here is the 32-bit FNV-1a hash of a string, which depends on unsigned wrap-around:

```
fn main() {
    var h: u32 = 2166136261;
    for c in "goose" { h = (h ^ c) * 16777619; }
    print(h);
}
```

```
3265791416
```

Division by zero aborts in every build.

Floats 3.3

`f32` and `f64` are IEEE floats. Arithmetic on `f32` stays 32-bit. An `f32` widens to `f64` implicitly, never the reverse, and every integer type converts to either float type implicitly.

A float literal adapts, like an integer literal, to the float type around it. Where nothing gives it a type it is an `f64`. The same goes for an expression made only of literals and integers, which is computed at whatever width its destination needs:

```
fn main() {
    let weight: f32 = 4.5;
    let geese: i32 = 3;
    let total = weight * geese;          // the
        i32 converts: an f32 multiply
    let third: f32 = 1.0 / 3.0;          //
        computed in f32
    let third64 = 1.0 / 3.0;             // an
        f64
    print(total, " ", third, " ", third64);
}
```

```
13.5 0.33333334 0.3333333333333333
```

Floats print in the shortest form that reads back as the same value *of their own type*, which is why the `f32` third has fewer digits. A whole number prints with `.0`, so you can always tell a float from an integer in output:

```
fn main() {
    print(2.0, " ", 0.1 + 0.2, " ", 1e20, " ",
        1e-7, " ", 1.0 / 0.0);
}
```

```
2.0 0.30000000000000004 1e+20 1e-07 inf
```

Booleans 3.4

`bool` is `true` or `false`, stored in one byte. Comparisons produce one, `&&` and `||` short-circuit, and `!` negates. `if` and `while` need a `bool`, and there is no conversion from integers:

```
fn main() {
    let eggs = 5;
    if eggs { print("there are eggs"); }
}
```

```
error: condition must be bool or an optional,
    got i64
```

Write `if eggs != 0`. The "or an optional" in that message is a hint of things to come: an *optional* can be tested directly, and Chapter 11 is about that.

Bitwise operators 3.5

`&`, `|`, `^`, `~`, `<<` and `>>` work as you expect, and keep the type of their operand. `>>` fills with zeroes on unsigned types and copies the sign bit on signed ones.

The precedence is Rust's, not C's, and there is one difference that will bite you if you are used to C: **bitwise operators bind tighter than comparisons**. So this does what it looks like it does:

```
fn main() {
    let flags: u8 = 0x35;
    let mask: u8 = 0x0F;
    print(flags & mask != 0, " ", flags >> 4, "
        ", ~mask, " ", flags ^ 0xFF);
}
```

```
true 3 240 202
```

In C, `flags & mask != 0` parses as `flags & (mask != 0)`, which is a classic bug. In Goose it is `(flags & mask) != 0`.

Named constants and type aliases 3.6

A `let` at the top level of a file, with no type written and a constant initializer, is a *named constant*. A named constant behaves exactly like the literal it stands for: it adapts to any type its value fits, and it can be used as an array size.

```
let FLOCK_SIZE = 24;
let HALF = FLOCK_SIZE / 2;

fn main() {
    let small: u8 = FLOCK_SIZE;        //
        adapts, as 24 would
    let pens: i32[HALF] = [0; HALF];   // an
        array of 12 zeroes
    print(small, " ", pens.len);
}
```

```
24 12
```

Give the declaration a type, `let FLOCK_SIZE: i64 = 24;`, and it becomes an ordinary `i64` value that no longer adapts.

`type Name = T;` gives a type a second name. An alias is the same type, not a new one, so values convert freely between the two names:

```
type Grams = u32;

fn main() {
    let egg: Grams = 61;
    let plain: u32 = egg;
    print(egg + plain);
}
```

```
122
```

From Odin

This is Odin's `Grams :: u32`, not `Grams :: distinct u32`. Goose has no `distinct`. A new type with its own identity comes from a `struct`, as in Chapter 10.

Exercises 3.7

1. Without running it, work out what this prints. Then run it and check.

`goose check fn main() { let a: u8 = 250; let b: u8 = 10; let c: i64 = 250; print(a + b, " ", c + b, " ", -1 % 5, " ", -1 / 5); }`

2. Write a function-free program that converts a temperature of `-40` degrees Celsius, stored in an `i32`, into Fahrenheit as an `f32`, using `f = c * 9 / 5 + 32`. Make sure the division happens in floating point. How many casts do you need?

3. A clock shows hours `0` to `23`. Write the expression for the hour that is `k` hours

before hour `h`, for any `k`, and check it for `h = 3`, `k = 50`.

Chapter 4

Control Flow as Expressions

Goose's control flow will look familiar, and most of it works the way you expect. The main idea to absorb is that `if`, `match` and blocks are *expressions*: they produce values. Once you are used to that, a lot of small temporary variables disappear from your code.

`if` 4.1

An `if` used as a statement needs no `else`, and its condition needs no parentheses. The braces are required:

```
fn main() {
    let honks = 7;
    if honks > 5 { print("noisy goose"); }
}
```

```
noisy goose
```

Used as a value, an `if` must have an `else`, and both branches must agree on a type. This is how Goose does what C does with `?:`, so there is no ternary operator:

```
fn main() {
    let weight = 3.8;
    let size = if weight > 4.0 { "large" } else
        if weight > 3.0 { "medium" } else {
        "small" };
    print(size);
}
```

```
medium
```

Leaving out the `else` is an error, because the `if` would have no value when the condition is false:

```
fn main() {
    let honks = 5;
    let loud = if honks > 2 { true };
}
```

```
error: an if used as a value requires an else
    branch
```

A block's value is its last expression, written without a semicolon. That is the same rule as in Rust. Put a semicolon after it and the block has no value.

> **From Odin**
>
> Odin has `x if cond else y` for an expression `if` and uses statements everywhere else. In Goose there is only one `if`, and it is an expression whenever you use its value.

Statements that end with a block 4.1.1

An `if`, a `match`, a loop or any other construct that ends with a `}` is a complete statement by itself, and needs no semicolon after it. The flip side is that an operator cannot continue such a

statement. This is an error, even though it looks reasonable:

```
fn main() {
    let honks = 5;
    let v = if honks > 2 { 1 } else { 2 } + 3;
}
```

```
error: expected ';' in variable declaration,
    found '+'
```

The parser saw a complete `if`. To use one as an operand, put it in parentheses:

```
fn main() {
    let honks = 5;
    let v = (if honks > 2 { 1 } else { 2 }) + 3;
    print(v);
}
```

```
4
```

`match` 4.2

`match` picks one of several arms by comparing a value against patterns. Chapter 12 covers matching on enums, which is where `match` really earns its keep. Here we look at matching on integers.

The patterns of an integer `match` are literals, named constants, half-open ranges, and comma-separated lists of those. The first arm that matches is taken, and a final `_` arm is required:

```
let EGG = 0;
let GOSLING = 1;
let JUVENILE = 3;
let ADULT = 52;

fn stage(weeks: i64) -> u8[:] {
    match weeks {
        EGG => "egg",
        GOSLING..JUVENILE => "gosling",
        JUVENILE..ADULT => "juvenile",
        _ => "adult",
    }
}

fn main() {
    for w in [0, 1, 2, 3, 30, 52] { print(w, "
        weeks: ", stage(w)); }
}
```

```
0 weeks: egg
1 weeks: gosling
2 weeks: gosling
3 weeks: juvenile
30 weeks: juvenile
52 weeks: adult
```

Like every range in Goose, `GOSLING..JUVENILE` is **half-open**: it includes 1 and 2 but not 3. There is no inclusive range syntax, so adjacent ranges share their boundary values, as `GOSLING..JUVENILE` and `JUVENILE..ADULT` do here, without overlapping.

A `match` is an expression, like `if`, so `stage` simply returns the value of the `match`. Leaving out the `_` arm is an error even if you think you have covered every case:

```
fn describe(n: i64) -> u8[:] {
    match n { 0 => "none", 1..10 => "a few" }
}

fn main() { print(describe(3)); }
```

```
error: integer match requires a _ arm
```

> **From C**
>
> There is no fall-through between arms, so there is no `break` to forget. To share an arm between several values, list them: `0, 7, 10..12 => …`.

Blocks are expressions too 4.3

A pair of braces on its own makes a scope, and like any block it has the value of its last expression. That is useful for computing a value with a few temporaries that should not leak into the surrounding code:

```
fn main() {
    let area = {
        let width = 12;
        let depth = 8;
        width * depth
    };
    print(area);
}
```

```
96
```

`block` and `break` with a value 4.3.1

Sometimes you want to bail out of a computation early with a result, without writing a separate function. `block { … }` gives `break` a target that is not a loop, and `break value` makes that the block's value:

```
fn main() {
    let weight = 0.0;
    let eggs = 3;
    let verdict = block {
        if weight <= 0.0 { break "no goose"; }
        if eggs == 0 { break "not laying"; }
        "healthy"
    };
    print(verdict);
}
```

```
no goose
```

Every `break` of the same `block` must give a value of the same type, and the block's final expression must match it too.

`guard` 4.4

`guard` states a condition that the rest of the block depends on. It is an `if` turned inside out: rather than nesting the main path inside the condition, you rule out the bad cases first and continue at the same indentation.

`guard cond else { … }` runs the `else` block when the condition is false. To leave the function, the loop or the `block`, put a `return`, `break` or `continue` inside the `else`:

```
fn eggs_per_goose(eggs: i64, geese: i64) -> i64
    {
    guard geese > 0 else { return 0; }
    eggs / geese
}

fn main() {
    print(eggs_per_goose(30, 6), " ",
        eggs_per_goose(30, 0));
}
```

```
5 0
```

`guard cond;` without an `else` simply skips the rest of the current block when the condition is false. In a loop body, skipping the rest of the body means going on to the next iteration:

```
fn main() {
    for i in 6 {
        guard i % 2 == 0;
        print("even ", i);
    }
}
```

```
even 0
even 2
even 4
```

`guard` really shines with optionals, because after `guard x else { … }` the compiler knows that `x` is not null for the rest of the block. That has to wait for Chapter 11.

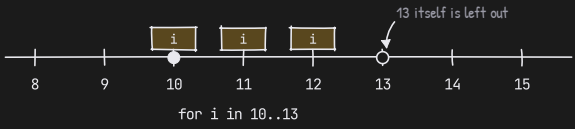
> **From Rust**
>
> `guard c else { return; }` is `let … else` without the pattern, or Swift's `guard`. Unlike both of those, the `else` block is not required to leave the scope. Without a jump it just runs instead of the rest of the block.

Loops 4.5

Goose has three loops: `for`, `while` and `loop`.

for 4.5.1

`for i in n` counts from `0` up to `n - 1`, and `for i in a..b` counts from `a` up to but not including `b`:



A range a..b starts at a and stops just before b.

```
fn main() {
    for i in 3 { print("lap ", i); }
    for i in 10..13 { print("pen ", i); }
}
```

```
lap 0
lap 1
lap 2
pen 10
pen 11
pen 12
```

The loop variable has the type of the range's bounds, `i64` for plain literals. You can ask for another type, and because the end of a range is one past the last value, you can visit every value of a small type, including its maximum:

```
fn main() {
    var count = 0;
    for b: u8 in 0..256 { count += 1; }
    print(count, " byte values");
}
```

```
256 byte values
```

`for x in array` visits the elements of an array. For elements of a fixed size, `x` is a *copy* of each element, and the compiler stops you from writing to it, since the write would change only the copy:

```
fn main() {
    let weights: i64[3] = [4, 5, 6];
    for w in weights { w += 1; }
}
```

```
error: cannot write w: a by-value binding is a
    copy of the element; bind it by reference
    (&w) to write the element (§6.5)
```

The message tells you the fix. `for &x in array` binds each element by reference, so writes go to the array. `for x, i in array` also gives you the index:

```
fn main() {
    var weights: i64[3] = [4, 5, 6];
    for &w in weights { w *= 1000; }
    for w, i in weights { print("goose ", i, ":
        ", w, " g"); }
}
```

```
goose 0: 4000 g
goose 1: 5000 g
goose 2: 6000 g
```

`while` and `loop` 4.5.2

`while cond { … }` is what you expect. `loop { … }` runs until something leaves it, and like `block` it can produce a value with `break value`:

```
fn main() {
    var n = 0;
    let first = loop {
        n += 6;
        if n % 4 == 0 { break n; }
    };
    print("first multiple of 6 and 4: ", first);

    var feed = 10;
    while feed > 0 { feed -= 3; }
    print(feed);
}
```

```
first multiple of 6 and 4: 12
-2
```

`break` leaves the innermost loop or `block`, and `continue` goes to the next iteration of the innermost loop. **There are no labels**, so you cannot break out of two loops at once. When you need to, put the loops in a function and `return`, or use a `block` around them.

`defer` 4.6

`defer` runs a statement, or a block, when the enclosing block ends, however it ends: by falling off the end, by `break`, `continue` or `return`. It is the same idea as Odin's `defer`, and it is used for the same thing: keeping cleanup next to the code that needs it.

When several defers are pending, they run **latest first**. A defer inside a loop body runs at the end of each iteration, including iterations that end with `continue`:

> **Newer than v2026.3**

> This example uses a feature added to Goose after the v2026.3 release, so it is not checked against the compiler the rest of the book is.

```
fn main() {
    {
        defer print("first deferred, runs
            last");
        defer print("second deferred, runs
            first");
        print("in the block");
    }
    for i in 3 {
        defer print("  end of lap ", i);
        if i == 1 { continue; }
        print("  lap ", i);
    }
}
```

```
in the block
second deferred, runs first
first deferred, runs last
  lap 0
  end of lap 0
  end of lap 1
  lap 2
  end of lap 2
```

A deferred body is ordinary code that runs later. It sees variables as they are when the block exits, not as they were when the `defer` was written. It cannot itself `return`, `break` or `continue` out of the block.

You will use `defer` much less in Goose than in Odin, because there is no memory to free. It is for the other kinds of cleanup: closing a window, restoring a counter, printing a closing bracket.

From Odin

> Odin's `defer` is also scoped to the block, not the function, and runs in reverse order. The difference is in what you defer. In Odin it is often `delete(x)` or `free(x)`. Goose has no equivalent of either.

Exercises 4.7

1. Write a function `grade(score: i64) -> u8[:]` that returns `"A"` for scores of 90 and above, `"B"` for 80 to 89, `"C"` for 70 to 79 and `"F"` otherwise, using a single `match`. What should happen for scores above 100?
2. Use `loop` with `break value` to find the smallest `n` for which `n * n` is larger than 2026.
3. Print a 4 by 4 checkerboard of `#` and `.` characters, starting with `#` in the top-left corner. Each call to `print` writes one row, so give it four arguments, each an `if` expression that picks the character for one square.

Chapter 5

Functions

Goose functions are free functions: there are no methods, no classes and no `impl` blocks. That does not stop you from writing `goose.honk()`, as you will see. This chapter covers declaring and calling functions, the several ways of getting values in and out of them, overloading, and functions declared inside other functions.

Declaring a function 5.1

A function is `fn`, a name, a parameter list and, after `->`, the result type. The body's last expression is the result, so short functions need no `return`:

```
fn feed(grams: i64) -> i64 { grams * 2 }

fn ration(geese: i64, grams_each: i64) -> i64 {
    if geese <= 0 { return 0; }
    return geese * feed(grams_each);
}

fn main() {
    print(ration(12, 150));
}
```

```
3600
```

Both styles work: an explicit `return`, or a final expression without a semicolon. Most Goose code uses the final expression for short functions and `return` for early exits.

The result type can be left out when the compiler can infer it from the body. It is good style to write it on anything other than a trivial helper, because it documents the function and keeps error messages close to the mistake.

Top-level declarations can appear in any order. There are no prototypes or headers, and `main` can call functions declared below it.

Parameters are `let`s 5.1.1

A parameter behaves like a `let`: you cannot reassign it. If you want a local copy you can change, mark the parameter `var`:

```
fn countdown(var n: i64) -> i64 {
    var steps = 0;
    while n > 0 { n -= 3; steps += 1; }
    steps
}

fn main() {
    print(countdown(10));
}
```

```
4
```

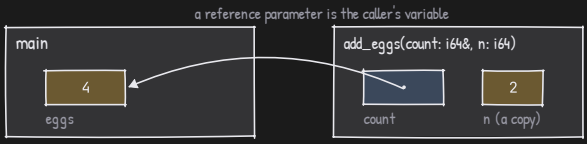
Without the `var`, the `n -= 3` would be the familiar error, `cannot assign to let n`.

Reference parameters 5.2

A parameter of type `T&` is a *reference* to a `T` that belongs to the caller. Writing to it writes the caller's variable. At the call site you pass the variable itself, with no `&`:

```
fn add_eggs(count: i64&, n: i64) {
    count += n;
}

fn main() {
    var eggs = 0;
    add_eggs(eggs, 4);
    add_eggs(eggs, 2);
    print(eggs);
}
```

```
6
```



A reference parameter refers to a variable in the caller's frame. An ordinary parameter is a copy.

Inside the function, `count` behaves exactly like an `i64`. There is no dereference operator, no `*count` and no `->`. A reference is *transparent*: wherever it is used as a value, it stands for the thing it refers to. Chapter 8 is about references in general, and explains how you talk about the reference itself rather than its target.

From C

The call `add_eggs(eggs, 4)` does not tell you that `eggs` might change, as `add_eggs(&eggs, 4)` would in C. The trade is the same one C++ makes with `int&`. In practice you will mostly pass arrays and structs this way, and those cannot be copied implicitly anyway (Chapter 6).

Default arguments 5.3

Trailing parameters can have default values, which a call may leave out:

```
fn portion(grams: f64, geese: f64 = 1.0, waste:
    f64 = 0.1) -> f64 {
    grams / geese * (1.0 - waste)
}

fn main() {
    print(portion(1000.0), " ", portion(1000.0,
        4.0), " ", portion(1000.0, 4.0, 0.0));
}
```

```
900.0 225.0 250.0
```

A default is evaluated at each call that leaves it out. It can use globals, constants and other functions, but never the caller's local variables, even if one happens to have the same name as something the default refers to.

Several results 5.4

A function can return more than one value. List the types after `->`, and return the values separated by commas:

```
fn split_flock(geese: i64, pens: i64) -> i64,
    i64 {
    return geese / pens, geese % pens;
}

fn main() {
    let per_pen, left_over = split_flock(23, 5);
    print(per_pen, " per pen, ", left_over, "
        left over");
}
```

```
4 per pen, 3 left over
```

Several results are a calling convention, not a tuple type. Goose has no tuples. You cannot store "the pair" in a variable or pass it around. You bind the results to names with `let a, b = f();`, or assign them to variables that already exist with `a, b = f();`. Used where a single value is expected, a call gives its first result.

The usual way to report success or failure is a trailing `bool`, which says whether the other results mean anything:

```
fn eggs_on_day(day: i64) -> i64, bool {
    if day % 7 == 0 { return 0, false; }    //
        the geese rest on Sundays
    return day % 3 + 1, true;
}

fn main() {
    let n, ok = eggs_on_day(14);
    print(n, " ", ok);
    var count = 0;
    var laid = false;
    count, laid = eggs_on_day(5);
    print(count, " ", laid, " ",
        eggs_on_day(5));
}
```

```
0 false
3 true 3
```

The last call is in a value context, so only its first result, `3`, is printed.

Only `return` and a function call can carry several values. An `if` cannot produce two values, so

`return if c { 1, 2 } else { 3, 4 };` is not allowed; each branch has to `return` on its own.

> **From Odin**
>
> This works just like Odin's multiple return values, including the convention of a trailing `ok`. Goose has no named results and no `or_return`. Chapter 15 describes Goose's ways of passing errors along.

> **From Rust**
>
> Where Rust would return `Option<i64>` or `(i64, bool)`, idiomatic Goose returns `i64, bool`. Chapter 11 introduces optionals, which are the natural choice when the result is "maybe a reference to something".

Overloading 5.5

Several functions can share a name, as long as their parameter types differ. The call picks the one that fits its arguments, at compile time:

```
fn describe(x: i64) -> u8[:] { "a count" }
fn describe(x: f64) -> u8[:] { "a weight" }
fn describe(x: bool) -> u8[:] { "a yes or no" }

fn main() {
    print(describe(12), ", ", describe(4.5), ",
        ", describe(true));
}
```

```
a count, a weight, a yes or no
```

An exact match beats a match that needs a conversion. Converting an integer to a float is the last resort, so an integer argument picks an integer overload over a float one when both are possible:

```
fn weigh(x: i32) -> u8[:] { "i32" }
fn weigh(x: f64) -> u8[:] { "f64" }

fn main() {
    let b: u8 = 3;
    print(weigh(2), " ", weigh(b), " ",
        weigh(2.5));
}
```

```
i32 i32 f64
```

When no overload fits, the error lists each candidate and why it was rejected:

```
fn hatch(day: i64) { }

fn main() {
    hatch(1.5);
}
```

```
error: no matching overload for call to hatch
candidate hatch at main.goose:1: argument 1:
    cannot pass f64 as i64
```

The result type never selects an overload. You can declare two functions that differ only in what they return, but every call to them is then ambiguous, and the compiler says so at the call.

Method-call syntax 5.6

Any function can be called as if it were a method of its first argument: `x.f(a, b)` means `f(x, a, b)`. This is called *uniform function call syntax*, or UFCS, and Goose uses it everywhere. Most of the standard library is written as free functions that read naturally in this form:

```
fn twice(x: i64) -> i64 { x * 2 }
fn plus(x: i64, y: i64) -> i64 { x + y }

fn main() {
    print(21.twice(), " ", 4.twice().twice(), "
        ", 5.plus(3).twice());
}
```

```
42 16 16
```

When you write `x.f()`, fields and built-in members such as `.len` are looked up first, then functions. You will see `xs.push(v)`, `text.trim()` and `list.sort()` throughout this book. They are all ordinary functions called with UFCS.

> **From Rust**
>
> There is no `self` parameter and nothing to `impl`. Any function whose first parameter fits can be called with a dot, including functions you write for types you did not define.

Nested functions 5.7

A function can be declared inside another function. The nested function can read and write the enclosing function's local variables:

```
fn main() {
    var total = 0;
    var visits = 0;
    fn feed(grams: i64) {
        total += grams;
        visits++;
    }
    feed(150);
    feed(200);
    feed(125);
    print(total, " g over ", visits, "
        feedings");
}
```

```
475 g over 3 feedings
```

A nested function is not a closure object. It exists only at compile time, so it cannot be stored in a variable, returned or put in a data structure. It can be passed to other functions as a compile-time argument, which is the subject of Chapter 16. That restriction is what lets every call be a direct call that the compiler can inline.

Nested functions are more useful than they look. A parser, for example, can keep its input, its cursor and its output as locals of one function and declare its recursive-descent functions inside it, all sharing that state without passing a context struct around. Chapter 25 is built that way.

Recursion is declared 5.8

There is one thing about functions that is quite unlike other languages: **a function that calls itself must say so**. This factorial does not compile:

```
fn fact(n: i64) -> i64 {
    if n <= 1 { 1 } else { n * fact(n - 1) }
}

fn main() { print(fact(10)); }
```

```
error: recursive call cycle through fact, which
    is not declared `recursive fn` (§7.8)
```

Mark the function `recursive fn` and it does:

```
recursive fn fact(n: i64) -> i64 {
    if n <= 1 { 1 } else { n * fact(n - 1) }
}

fn main() { print(fact(10)); }
```

```
3628800
```

The keyword exists because of the memory model. A recursive function has some restrictions on the data it can hold across a recursive call, and the compiler needs to know where every recursive cycle starts in order to check them. Chapter 17 explains the rules. Until then, a recursive function that only uses scalars, like `fact`, needs nothing more than the keyword.

Functions that never return 5.9

The built-ins `abort(message)` and `exit(code)` never return, and the compiler knows it. A branch that ends in one of them does not need to produce a value:

```
fn eggs_per_goose(eggs: i64, geese: i64) -> i64
    {
    if geese > 0 { eggs / geese } else {
        abort("no geese") }
}

fn main() {
    print(eggs_per_goose(30, 6));
}
```

```
5
```

This also works through your own functions: a function whose every path ends in `abort` or `exit` is known never to return, so it can be called in the same position.

Exercises 5.10

1. Write a function `min_max(a: i64, b: i64, c: i64) -> i64, i64` that returns the smallest and the largest of its three arguments, and call it with `7, -2, 4`.
2. Write `fn clamp_weight(w: f64, lo: f64 = 2.0, hi: f64 = 8.0) -> f64`, and use it with UFCS to clamp `9.5`, `1.0` and `5.0`, the last one with a custom range of 6 to 7.
3. Write a recursive function `digits(n: i64) -> i64` that returns how many decimal digits a non-negative number has, with `digits(0) == 1`.

PART II

Memory Without a Heap

This part is the heart of the book. It explains where Goose keeps values whose size is not known at compile time, why references into them can never dangle, and how strings, arrays and structs are laid out as plain, contiguous bytes. Read it in order: each chapter builds on the one before.

Chapter 6

Where Values Live

In C, the question "where does this value live?" has three answers: in a register or on the call stack, in static data, or on the heap. Goose keeps the first two and replaces the third. This chapter explains what replaces it, and the three consequences that everything else in the language builds on: freeing is free, nothing moves, and copies are always visible.

Three kinds of size 6.1

Every Goose type falls into one of three *size classes*:

- **Fixed**: the size is known at compile time. Scalars, structs made of fixed fields, fixed-length arrays such as `i32[4]`, references and slices.
- **Variable**: the size is chosen when the value is built, and never changes afterwards. A finished string `u8[]`, for example, or a struct that contains one.
- **Resizable**: the size can change after the value is built. The two growable arrays, `T[>..]` and `T[>..<]`, which you will meet properly in the next chapter.

Fixed-size values live where they do in C: in registers, on the native call stack, and in static

data. Nothing about them is surprising. They are copied when you assign them, passed by value, and returned by value.

Variable and resizable values are where Goose is different.

Data stacks 6.2

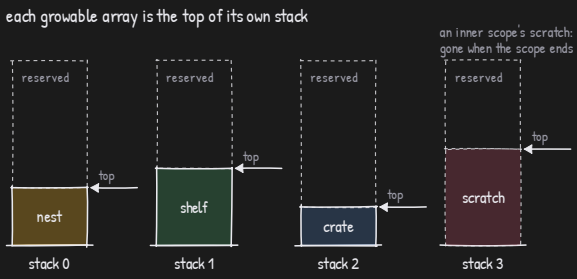
A running Goose program has the native call stack, its static data, and a small number of **data stacks**. A data stack is a large reservation of address space, gigabytes of it, with a pointer to its current top. Reserving address space costs nothing until the memory is touched, so the size of the reservation is not a real cost.

Every value that is not fixed-size lives on a data stack, and is owned by the variable that declared it. Two rules make this work:

1. Values are pushed onto a stack in the order of their lifetimes, and freed all at once when the scope that owns them ends. Freeing them means setting the stack's top pointer back to where it was: one store.
2. A growable value is always the *topmost* thing on its stack, for as long as it lives. So growing it is a pointer bump, with no capacity check and no reallocation, and it never moves.

Rule 2 sounds very restrictive until you notice the plural in "data stacks". If three growable

arrays are alive at the same time, they get three different stacks:



Every growable array is the topmost value on a data stack of its own, so it can grow in place. When an inner scope ends, its stack's top drops back.

The compiler works out how many stacks a program needs, and which value goes on which stack, from the program's structure. A stack is reused as soon as the scope that used it has ended. You never write any of this down; there is no syntax for a stack anywhere in the language. You can ask the compiler what it decided, though:

```
fn main() {
    var nest: i64[>..] = [];
    var shelf: i64[>..] = [];
    var crate: i64[>..] = [];
    for i in 3 { nest.push(i); shelf.push(i *
        10); crate.push(i * 100); }
    print(nest, shelf, crate);
}
```

```
$ goose check --stacks crates.goose
```

```
data stacks: 2 GB reserved per stack, 1 MB guard
    gap, 32 TB budget: 16376 regions,
    GS_MAX_STACKS 1024
main program: 4 data stacks + 0 global stacks =
    4 regions
thread cap: none, no workers
fn main: 4 own, 4 with callees
```

Three of the four stacks are for the three arrays. The fourth holds the text that `print` builds before writing it.

> **From Odin**
>
> A data stack is an arena with a bump pointer, the same structure as a `mem.Arena`. The difference is that you never create one, pick one, pass one in `context`, or free one. The compiler assigns values to arenas and resets them at the right moments.
>
> **From Rust**
>
> There is no `Box`, `Vec` or `String` that owns a separate heap block. Ownership is still the organizing idea, and every dynamic value has exactly one owning variable. But the owner owns a region of a stack rather than a heap block, and dropping it is a pointer reset rather than a call to the allocator.

Leaving a scope frees everything 6.3

Here is a loop that builds a scratch buffer of a thousand bytes on every iteration:

```
fn main() {
    for round in 3 {
        var scratch: u8[>..] = [];
        for i in 1000 { scratch.push((i % 256)
            as! u8); }
        print("round ", round, ": ",
            scratch.len, " bytes");
    }
}
```

```
round 0: 1000 bytes
round 1: 1000 bytes
round 2: 1000 bytes
```

At the closing brace of each iteration, `scratch`'s stack has its top reset to where it was before the iteration started, so each round starts with an empty stack and reuses the same memory. No destructor runs, no free list is updated, and nothing walks over the bytes. In a C program the equivalent is a `malloc` and a `free` per round, or a buffer you manage by hand. In Goose it is the default.

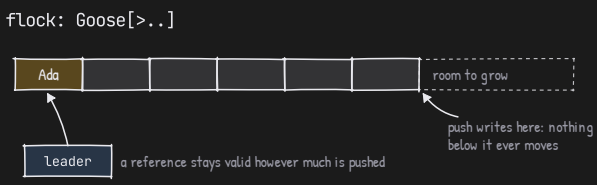
The same is true of every scope, not just loop bodies: a function's body, an `if` branch, a bare `{ … }` block. When the scope ends, everything it owned is gone.

Nothing moves 6.4

Because a growable array is always on top of its stack, it never has to be moved to make room. Its elements stay at the same addresses for as long as the array lives, so **a reference into a growing array stays valid**:

```
struct Goose { name: u8[..12], honks: i32 }

fn main() {
    var flock: Goose[>..] = [];
    let leader .= flock.push(Goose { name:
        "Ada", honks: 0 });
    for i in 100000 { flock.push(Goose { name:
        "", honks: 0 }); }
    leader.honks = 7;
    print(flock.len, " geese; ", flock[0]);
}
```

```
100001 geese; Goose { "Ada", 7 }
```



The elements of a grow-only array never move, so a reference to an element stays valid after any number of pushes.

`push` returns a reference to the element it just built, and `.=` binds `leader` to that reference. After a hundred thousand more pushes, writing through `leader` still writes `flock[0]`.

This is the single most important property of the memory model. A `std::vector`, an Odin `[dynamic]T` or a Rust `Vec<T>` reallocates when it outgrows its capacity, which invalidates every pointer into it. That is why C and Odin programmers keep indices into arrays instead of pointers, and why the Rust borrow checker refuses to let you hold a reference into a vector while you push to it. In Goose you can hold a

typed reference, and the compiler only has to check that the reference does not outlive the array.

Copies are visible 6.5

Fixed-size values are copied when you assign them, exactly as in C:

```
struct Egg { weight: f32, day: i32 }

fn main() {
    var first = Egg { 61.0, 1 };
    var second = first;          // a copy
    second.day = 2;
    print(first.day, " ", second.day);
}
```

```
1 2
```

Non-fixed values are different. Copying one costs time proportional to its size, so Goose never does it behind your back. Wherever a growable array would be implicitly copied, it is bound by reference instead, or the program is rejected:

```
fn main() {
    var eggs: i64[>..] = [61, 72, 58];
    let same = eggs;             // no copy:
        `same` names the same array
    same.push(66);
    var mine = copy(eggs);       // an explicit,
        visible copy
    mine.push(70);
    print(eggs, " ", mine);
}
```

```
[61, 72, 58, 66] [61, 72, 58, 66, 70]
```

`let same = eggs;` did not copy anything: `same` is a reference to `eggs`, and pushing through it pushes onto `eggs`. `copy(eggs)` made a real copy, which lives on its own stack.

Where a declaration's type says it needs a value of its own, a plain assignment from another variable is an error, and the error tells you the two ways out:

```
fn main() {
    var eggs: i64[>..] = [61, 72, 58];
    var mine: i64[>..] = eggs;
}
```

```
error: eggs is not fixed-size and is not copied
    implicitly (§4.1): pass copy(eggs) for a
    copy, or bind it by reference
```

The same rule applies at calls. Passing a growable array to a function passes it by reference, whatever the parameter looks like, and a function that wants its own copy calls `copy` itself.

> **From C**
>
> Think of every non-fixed variable as already being "the pointer", and `copy(x)` as the explicit `memcpy` into a fresh allocation. There is no way to write a struct assignment that silently performs a deep copy.

Built in place 6.6

If copies are never implicit, how does a function return a new array? It does not copy one. Every non-fixed value is **constructed directly in its final location**:

```
struct Egg { weight: f32, day: i32 }

fn lay_week() -> Egg[>..] {
    var eggs: Egg[>..] = [];
    for d in 7 { eggs.push(Egg { 60.0 + d, d as
        i32 }); }
    return eggs;
}

fn main() {
    var week = lay_week();
    print(week.len, " eggs, the last one ",
        week[6]);
}
```

```
7 eggs, the last one Egg { 66.0, 6 }
```

When `main` calls `lay_week`, the compiler knows where the result is going: into `week`. It compiles `lay_week` so that its local `eggs` *is* `week` from the start, and every push goes directly into `main`'s variable. `return eggs` copies nothing.

The same thing happens when the destination is an element being pushed, the field of a struct being built, or the argument of another call. `str`, which formats its arguments as a new string, writes its text directly into the new element here, with no temporary string in between:

```
fn main() {
    var log: u8[][>..] = [];
    for day in 3 { log.push(str("day ", day + 1,
        ": ", day * 2 + 1, " eggs")); }
    for line in log { print(line); }
}
```

```
day 1: 1 eggs
day 2: 3 eggs
day 3: 5 eggs
```

The practical upshot is that returning a value is as cheap as filling in an out-parameter, so Goose code rarely has out-parameters. Write the function that returns the value.

Globals 6.7

Globals are declared at the top level, like constants, but with `var`. A global is initialized before `main` runs and lives for the whole program. A global growable array owns the bottom of a data stack for the program's lifetime, which makes it a natural program-wide arena:

```
var registry: u8[][>..];          // a var with
    only a type starts out empty

fn register(name: u8[:]) { registry.push(name);
    }

fn main() {
    register("Ada");
    register("Grace");
    print(registry.len, " geese registered: ",
        registry);
}
```

```
2 geese registered: ["Ada", "Grace"]
```

`name` is a *slice*, a view of bytes that belong to the caller. The elements of `registry` are `u8[]`, finished strings with bytes of their own, so `push` builds a new element by copying the bytes that `name` views, directly into place. That is a copy you can see coming: the type of the element says

it owns its bytes. Chapter 9 covers the different kinds of string.

Design note: why no heap? 6.8

A general-purpose heap answers a question most programs never ask: "give me memory whose lifetime I will decide later, in an order I cannot predict". Answering it costs a lot. It needs an allocator with free lists and size classes, fragmentation, a header on every block, a pointer to every block, and either a programmer who frees each block correctly, a garbage collector, or a type system that proves the frees correct.

Most data does not need that flexibility. A request handler's scratch buffers live exactly as long as the request. A compiler's syntax tree lives exactly as long as the compilation of one file. A frame's draw list lives for one frame. Arena allocation has always been the fast way to handle these cases, and Odin programmers use it routinely. Goose's bet is that arena allocation, done automatically and checked by the compiler, covers nearly all of a program's data. For the rest, the data whose lifetimes really do not nest, Chapter 14 introduces pools, which reuse slots without ever freeing memory.

Getting rid of the heap also gets rid of pointers inside data. A Goose array of strings is one block, not a tree of blocks. That makes it faster to build

and to walk, and it means a value can be copied, saved to disk or sent to another thread as a single run of bytes.

Exercises 6.9

1. Write a function `squares(n: i64) -> i64[>..]` that returns the first `n` square numbers, and print `squares(6)`. Where does the array returned by the function actually live?
2. Predict the output of this program, then run it:

```
goose check fn main() { var a: i64[>..] = [1]; let b = a; var c = copy(a); b.push(2); c.push(3); a.push(4); print(a, " ", b, " ", c); }
```

3. Use `goose check --stacks` on the program from exercise 2. How many data stacks does `main` need, and why?

Chapter 7

The Array Family

Goose does not have *an* array type. It has a family of them, and the members differ in only one thing: what can happen to their size after they are built. Choosing between them is one of the main design decisions in a Goose program, so this chapter goes through each in turn and ends with a guide to picking one.

The family 7.1

Type	Size class	What it is
`T[k]`	fixed	exactly `k` elements, `k` known at compile time
`T[]`	variable	sized when built, frozen afterwards
`T[..k]`	fixed	room for `k` elements inline, length from 0 to `k`
`T[..]`	variable	capacity chosen when built, length up to it
`T[>..]`	resizable	**grow-only**, the default growable array
`T[>..<]`	resizable	**grow-shrink**, for stacks, queues and heaps

The type syntax is postfix: the brackets go after the element type, so a growable array of 16-byte

names is `u8[..16][>..]`, read from left to right as "an array of 16-byte names that can grow".

Every one of them has the same representation: some metadata, followed by the elements, inline and packed together. There is never a pointer to a separate block of elements.

The layout of each kind of array. Metadata comes first, then the elements, in one block.

Here they all are at once:

```
fn main() {
    var pen: i32[4] = [3, 1, 4, 1];          //
        fixed
    let frozen: i64[] = [7, 8, 9];           //
        sized once
    var nest: i64[..6] = [];                 //
        up to 6, inline
    nest.push(10); nest.push(20);
    var sized: i64[..] = [..10];             //
        capacity 10, chosen now
    sized.push(42);
    var flock: i64[>..] = [];                //
        grow-only
    for i in 4 { flock.push(i * i); }
    var stack: i64[>..<] = [];               //
        grow-shrink
    stack.push(5); stack.push(6); stack.push(7);
    let top = stack.pop();
    print(pen, " ", frozen, " ", nest, " ",
        nest.len, "/", nest.cap, " ",
        sized.cap);
    print(flock, " ", stack, " ", top);
}
```

```
[3, 1, 4, 1] [7, 8, 9] [10, 20] 2/6 10
[0, 1, 4, 9] [5, 6] 7
```

Fixed arrays: `T[k]` 7.2

A fixed array is exactly what it is in C: `k` elements, no metadata. The size must be a compile-time constant, that is, a literal, a named constant or an expression made of them. A local variable will not do, even a `let`:

```
fn main() {
    let n = 5;
    let pens: i64[n] = [0; n];
}
```

```
error: array size n names a local variable, not
    a constant
```

`[v; n]` is the *fill literal*: `n` copies of `v`, evaluated once. It is the easy way to initialize a large fixed array, and Goose requires every array to be fully initialized. A fixed array of a fixed element type is fixed-size, so it is copied on assignment like any other fixed value.

Frozen arrays: `T[]` 7.3

`T[]` is an array whose length is decided when it is built and never changes afterwards. It stores its length as a `u32` in front of the elements. You cannot push to it or pop from it:

```
fn main() {
    let frozen: i64[] = [7, 8, 9];
    frozen.push(10);
}
```

```
error: .push is not available on i64[]
```

This is the type for finished data: a string inside a record, the list of children of a node, anything that is built once and then only read. Because its size depends on its contents, `T[]` is a *variable-size* type, which has consequences you will meet in the next few chapters.

The length can be stored in a narrower type when you know the array is small: `T[u8]` holds up to 255 elements with a one-byte length, and `T[u16]` up to 65535. There is also `T[varint]`, whose length takes as few bytes as its value needs. Chapter 10 has more about `varint`.

From C

> `T[]` is close to C's flexible array member, `struct { uint32_t len; T items[]; }`, except that it can appear anywhere in a struct, not only at the end, and the language tracks its length for you.

Limited arrays: `T[..k]` and `T[..]` 7.4

A limited array has a capacity and a length. `T[..k]` reserves room for `k` elements inline, wherever the array sits, and its length can go up and down between 0 and `k`. Because the room is always there, `T[..k]` is a *fixed*-size type, so it can be a field of a struct, an element of another array, or a local, exactly like a `T[k]`.

`T[..]` is the same thing with a capacity chosen when the array is built, written `[..capacity]`. Since the capacity is not known at compile time, `T[..]` is a variable-size type.

Pushing past the capacity stops the program:

```
fn main() {
    var nest: i64[..2] = [];
    nest.push(1);
    nest.push(2);
    nest.push(3);
}
```

```
goose runtime error: limited array capacity
    exceeded
```

The most common limited array by far is a short string inside a struct: `name: u8[..16]` stores up to 16 bytes and a one-byte length right inside the

struct, so an array of such structs is still one flat block.

> **From Rust**
>
> `T[..k]` is an `ArrayVec<T, k>`: a fixed-capacity vector stored inline. Odin's `core:container/small_array` is the same idea.

Grow-only arrays: `T[>..]` 7.5

The grow-only array is the workhorse. It is what you use wherever you would use a `Vec`, a `[dynamic]T` or a hand-rolled growable buffer: building up results, arenas of nodes, string builders, scratch space.

It lives on top of its own data stack, so it can grow without limit and without ever moving. `push(v)` builds one element at the end, in place, and returns a reference to it. `append(xs)` copies in all the elements of another array, a slice or a literal:

```
fn main() {
    var weights: i64[>..] = [];
    weights.push(4200);
    weights.append([3900, 4600]);
    let frozen: i64[] = [5100, 4400];
    weights.append(frozen);
    let last .= weights.push(4800);
    print(weights, " ", weights.len, " ",
        weights.index_of(last));
}
```

```
[4200, 3900, 4600, 5100, 4400, 4800] 6 5
```

`index_of(r)` turns a reference to an element back into its index.

Grow-only does not mean "never shrinks" 7.5.1

The name "grow-only" describes a guarantee, not a restriction on the operations. A `T[>..]` has `pop`, `pop_n`, `resize` and `clear`, and you can use them, but **only where nothing points into the array**. While a reference or slice into the array is still going to be used, shrinking it is an error:

```
fn main() {
    var eggs: i64[>..] = [61, 72, 58];
    let first .= eggs[0];
    eggs.pop();
    print(first);
}
```

```
error: cannot pop eggs while first is still
    used: it may hold a reference or slice into
    it (§5.1)
```

The check is about *liveness*, not scope. Move the `print(first)` above the `pop` and the program compiles, because `first` is never used after the array shrinks. The compiler does this analysis for you, which is how a grow-only array can be both a structure you hold references into and a scratch buffer you clear on every iteration.

```
fn main() {
    var eggs: i64[>..] = [61, 72, 58, 66];
    let first .= eggs[0];
    print(first);                // first's last use
    eggs.pop();
    eggs.resize(6, 0);           // grow with copies of 0
    print(eggs);
    eggs.clear();
    print(eggs.len);
}
```

```
61
[61, 72, 58, 0, 0, 0]
0
```

Grow-shrink arrays: `T[>..<]` 7.6

A grow-shrink array is for data that really does go up and down: a stack, a queue, a priority heap. It shrinks from anywhere, at any time, with no analysis needed. The price is a different restriction: **references into a grow-shrink array may only be kept in local variables**, never stored in a field, an element or a global. That way the compiler only has to look at the local variables in scope when it checks a `pop`.

```
fn main() {
    var todo: i64[>..<] = [];
    for job in [3, 1, 4, 1, 5] { todo.push(job); }
    var done = 0;
    while todo.len > 2 { done += todo.pop(); }
    print(todo, " done: ", done);
}
```

```
[3, 1] done: 10
```

The elements of a grow-shrink array must be fixed-size, because a popped slot may later be reused for a different value.

> **From Odin**
>
> `[dynamic]T` covers both of these cases, at the cost of invalidating pointers whenever it grows. Goose splits it in two. Use `T[>..]` when you want stable references, and `T[>..<]` when you want a stack.

What can go inside what 7.7

The array kinds can be nested, within limits. The rules follow from the size classes:

- **Fixed** elements can go in any array.
- **Variable** elements, such as `u8[]` strings, can go in a `T[]` or a `T[>..]`. Such an array is *sequential* (see below).
- **Resizable** values can never be elements. An array of growable arrays would need one data stack per element.

```
fn main() {
    var groups: i64[>..][>..] = [];
}
```

```
error: array elements may not be resizable:
    i64[>..]
```

When you want "a list of lists", store finished inner arrays (`i64[][>..]`), or keep all the inner elements in one array and the boundaries in another. Chapter 14 shows a third way.

Sequential arrays 7.7.1

An array whose elements have different sizes, like an array of strings, cannot be indexed. Where the third element starts depends on the lengths of the first two, so `words[2]` would have to walk the array. Goose does not hide that cost: you can iterate such an array, but not index it.

```
fn main() {
    var words: u8[][>..] = [];
    words.push("honk");
    print(words[0]);
}
```

```
error: arrays of variable-size elements cannot
    be indexed, only iterated; for random
    access, keep your own array of references
    to them (§3.3)
```

```
fn main() {
    var words: u8[][>..] = [];
    for w in ["honk", "hiss", "flap"] {
        words.push(w); }
    for w, i in words { print(i, ": ", w); }
}
```

```
0: honk
1: hiss
2: flap
```

When you need both a compact layout and random access, keep an index beside the array, as the error message suggests. Chapter 13 shows how to do that with four bytes per entry. When you only need short strings, use a fixed-size element such as `u8[..16]` instead, and the array becomes indexable again.

Array literals 7.8

An array literal is written in brackets. Its element type comes from where it is going, if that is known: a typed variable, a parameter, a field. Without a destination, `[1, 2, 3]` is an `i64[3]`. There are a few special forms:

- `[]` is an empty array, and needs a destination type.
- `[v; n]` is `n` copies of `v`.
- `[..cap]` is an empty `T[..]` with capacity `cap`.
- `var out = [];` on its own creates a grow-only array whose element type is decided by the first `push` or `append`.

```
fn main() {
    var readings = [];
    readings.push(36.6);
    readings.push(37.1);
    let tally: i64[4] = [0; 4];
    print(readings, " ", tally, " ", [1, 2,
        3].len);
}
```

```
[36.6, 37.1] [0, 0, 0, 0] 3
```

Members 7.9

Every array kind has `.len`, and the limited ones have `.cap`. The rest are available where they make sense:

Member	Available on	Does
`.push(v)`	limited, grow-only, grow-shrink	builds `v` at the end; returns a reference to it
`.append(xs)`	the same	copies in the elements of an array or slice
`.pop()`	limited, grow-shrink; grow-only where nothing points in	removes and returns the last element
`.pop_n(n)`	the same	drops the last `n` elements
`.resize(n, v)`	the same	grows with copies of `v`, or shrinks to `n`
`.clear()`	the same	empties the array
`.index_of(r)`	arrays of fixed-size elements	the index of the element `r` refers to

Growing always provides a value for every new element, so there is no operation that exposes uninitialized memory.

Two arrays compare with `==` element by element, and arrays of different kinds can be compared as long as their element types match:

```
fn main() {
    let frozen: i64[] = [7, 8, 9];
    var grown: i64[>..] = [];
    for i in 7..10 { grown.push(i); }
    print(frozen == [7, 8, 9], " ", grown ==
        frozen, " ", grown == [7, 8]);
}
```

```
true true false
```

Choosing an array 7.10

- `T[>..]` is the default. Reach for it first.
- `T[>..<]` when you need to pop from a stack or a heap in a loop, and only ever hold references into it in local variables.
- `T[..k]` when an array must live inside something else, like a name inside a struct, and has a sensible upper bound.
- `T[]` for data that is complete when it is built.
- `T[k]` for things that really have a fixed shape: a 4 by 4 matrix, a 256-entry lookup table.

Exercises 7.11

1. Build a `u8[..8][>..]` holding the names `"Ada"`, `"Grace"` and `"Linus"`, then print the second name. Why can you index this array, when you could not index a `u8[][>..]`?
2. Write a program that pushes the numbers 1 to 10 onto a `i64[>..<]` and then pops numbers until the sum of what was popped exceeds 20.

Print the popped sum and what is left on the stack.

3. What goes wrong if you change the type of the stack in exercise 2 to `i64[..8]`? Try it.

Chapter 8

References and Slices

So far, references have appeared in passing: `push` returns one, `.=` binds one, and a parameter of type `i64&` is one. This chapter looks at them properly, together with their many-element cousin, the slice. It also explains the single rule the compiler uses to make sure neither can ever dangle.

References 8.1

A reference `T&` is the address of a `T`. It is eight bytes, it can never be null, and there is no arithmetic on it and no way to cast it to anything else.

References are **transparent**. Wherever a reference is used as a value, it behaves exactly like the thing it refers to: you read it, do arithmetic with it, compare it, print it and pass it to functions as if it were the target. Assigning to it writes the target. There is no `*` and no `->`.

That raises an obvious question. If `r = 5` writes the target, how do you change which thing `r` refers to? The answer is a separate operator, `.=`, which works on the reference itself:

```
fn main() {
    var pens: i64[>..] = [10, 20, 30, 40, 50];
    var r .= pens[1];          // r refers to
        pens[1]
    var c = pens[1];           // c is a copy of
        pens[1]
    r += 1;                    // writes pens[1]
    c += 100;                  // writes the
        copy
    print(pens, " ", r, " ", c);
    r .= pens[4];              // r now refers
        to pens[4]
    r = 0;
    print(pens);
}
```

```
[10, 21, 30, 40, 50] 21 120
[10, 21, 30, 40, 0]
```

The dot is the visual cue: `x .= e` and `x .== e` are about the reference `x`, while `x = e` and `x == e` are about whatever it refers to.

- `let r .= e;` or `var r .= e;` declares `r` as a reference to `e`, which can be a variable, a field, an element, or a call that returns a reference.
- `r .= e;` rebinds an existing reference variable, field or element.
- `r1 .== r2` asks whether two references refer to the *same* thing; `r1 == r2` compares the two things' values.

```
fn main() {
    var pens: i64[>..] = [40, 40];
    let a .= pens[0];
    let b .= pens[1];
    let c .= pens[0];
    print(a == b, " ", a .== b, " ", a .== c);
}
```

```
true false true
```

From C

A reference is a pointer that you cannot do arithmetic on, cannot make null, and never dereference by hand. Comparing addresses, which `p == q` does in C, is `p .== q` in Goose.

Where references come from 8.1.1

You will get most of your references from `push`, from parameters of reference type, and from `.=`. The explicit way to make one is `&x`, which takes a reference to a variable, a field or an element:

```
fn main() {
    var honks = 1;
    let a = &honks;             // a reference,
        because of the explicit &
    let b .= honks;             // also a
        reference
    var r .= honks;
    let copy_of = r;            // a copy: a
        reference to a fixed-size value decays
    a += 1;
    b += 1;
    print(honks, " ", copy_of);
}
```

```
3 1
```

The line `let copy_of = r;` is worth a second look. When a variable's type is inferred from a

reference to a *fixed-size* value, it gets a copy of the value, not a second reference. You have to say `.=` or `&` to get a reference. Non-fixed values never copy implicitly, as you saw in Chapter 6, so for those a plain `let` gives a reference.

You can only take a reference to something that has a location. `&5`, or `&` of a function's temporary result, is an error.

Slices 8.2

A slice `T[:]` is a view of a run of elements in some array: a reference to the first one and a count. It is the universal "here are some elements" type. Every kind of array converts to a slice of itself, without copying, wherever a slice is expected:

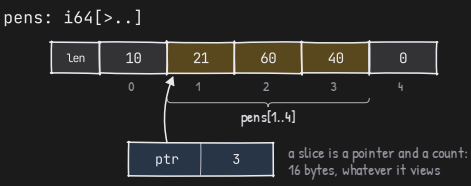
```
fn total(xs: i64[:]) -> i64 {
    var t = 0;
    for x in xs { t += x; }
    return t;
}

fn main() {
    let fixed: i64[3] = [1, 2, 3];
    var grown: i64[>..] = [10, 20, 30, 40, 50];
    print(total(fixed), " ", total(grown), " ",
        total([4, 5]));
}
```

```
6 150 9
```

You can also take a slice of part of an array with a half-open range. Either bound can be left out, and `^k` means "`k` from the end":

```
fn main() {
    var pens: i64[>..] = [10, 21, 60, 40, 0];
    let mid = pens[1..4];
    print(mid, " ", pens[..2], " ", pens[^2..],
        " ", mid.len);
}
```

```
[21, 60, 40] [10, 21] [40, 0] 3
```



A slice holds a pointer to its first element and a count. It views the elements in place; nothing is copied.

A slice is bounds-checked against its own length, and a slice of a slice can never be larger than the original. A slice taken from a grow-only array stays valid when the array grows, because nothing moves. It keeps viewing the elements it was made from, and does not include the new ones:

```
fn main() {
    var pens: i64[>..] = [10, 21, 60, 40, 0];
    let mid = pens[1..4];
    pens.push(60);
    print(mid, " of ", pens.len);
}
```

```
[21, 60, 40] of 6
```

From Odin

> `T[:]` is Odin's `[]T`, and `a[1..4]` is `a[1:4]`. The difference is that a Goose slice is checked: it can never outlive the array it views.

The one rule 8.3

Here is the whole of Goose's borrow checker:

> A reference or slice must not outlive the variable that owns what it points at.

That is all. There are no lifetime parameters, no rules about how many references may exist at once, and no distinction between shared and exclusive borrows. Two references to the same element are fine, and you can write through both:

```
fn main() {
    var nest: i64[>..] = [3];
    let a .= nest[0];
    let b .= nest[0];
    a += 1;
    b += 1;
    print(nest);
}
```

```
[5]
```

Aliasing like this is harmless in Goose, because the things that make it dangerous elsewhere are absent. Nothing can be freed while a reference exists, a grow-only array cannot move, and there is no memory shared between threads.

From Rust

> This program would need `Cell` or `unsafe` in Rust, because two `&mut` to the same location are not allowed. Rust needs exclusivity because it has no other way to stop a reference into a `Vec` from surviving a reallocation. Goose's arrays do not reallocate, so it doesn't need it.

Roots 8.3.1

To enforce the rule, the compiler gives every reference and slice a **root**: the variable whose lifetime bounds what the reference points at. `&x` is rooted at `x`. A slice of `pens` is rooted at `pens`. A reference returned by a function is rooted at whichever arguments it might have come from.

You never write roots down. The compiler works them out, and when a function is called with references that have different roots, it type-checks the function separately for each combination. That gives the precision of Rust's lifetime parameters without any of the syntax.

Most of the time you will only notice roots when you break the rule. There are three ways to do that.

Returning a reference to a local. A function's locals die when it returns, so nothing rooted in them can be returned:

```
fn first_three(s: u8[:]) -> u8[:] {
    var buf: u8[>..] = [];
    buf.append(s);
    return buf[..3];
}

fn main() { print(first_three("goose")); }
```

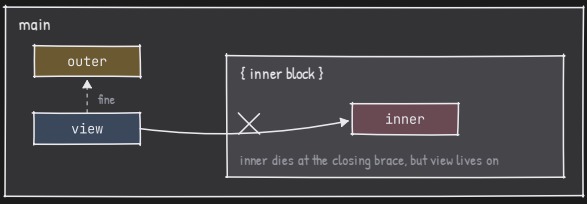
```
error: returning a reference rooted in buf,
    which dies with this function (§9.2)
```

The fix is to return the array itself, which is built in the caller's destination, or a slice of the parameter `s`, which belongs to the caller.

Storing a reference somewhere that lives longer. Here `view` outlives the block that owns `inner`:

```
fn main() {
    var outer: u8[>..] = [];
    outer.append("outside");
    var view = outer[..];
    {
        var inner: u8[>..] = [];
        inner.append("inside");
        view = inner[..];
    }
    print(view);
}
```

```
error: storing a reference rooted at inner,
    which does not outlive the destination
    (§9.2)
```



view lives in main's scope, but inner dies at the end of the block, so view may not refer to it.

Keeping a view of a temporary. A function's result, a literal or a `copy` is a temporary that lasts until the end of its statement. You can pass a view of it to a function, but you cannot keep one:

```
fn make() -> u8[>..] { str("honk honk") }

fn main() {
    let v = make()[..4];
    print(v);
}
```

```
error: binding v to a slice rooted at a
    temporary, which does not outlive it
    (§9.2): a temporary lasts until the end of
    its statement, so bind it to a variable of
    its own first
```

The message says what to do: `let t = make(); let v = t[..4];`.

A function that returns one of its arguments 8.3.2

A function may return a reference that came from one of its parameters, because the arguments belong to the caller. When it could

return any of several, the result is rooted at all of them:

```
fn heavier(a: i64[:], b: i64[:]) -> i64[:] {
    if total(a) >= total(b) { a } else { b }
}

fn total(xs: i64[:]) -> i64 { var t = 0; for x
    in xs { t += x; } t }

fn main() {
    var morning: i64[>..] = [4200, 3900];
    var evening: i64[>..] = [4600, 4100];
    let w = heavier(morning, evening);
    print(w);
}
```

```
[4600, 4100]
```

Here `w` is rooted at both `morning` and `evening`, and it must satisfy the rule for both: it may not outlive either one. In Rust you would have written `fn heavier<'a>(a: &'a [i64], b: &'a [i64]) -> &'a [i64]`. Goose works the same thing out from the body.

Liveness, not scope 8.4

The compiler cares about when a reference is *last used*, not about where its scope ends. A slice that is never used again is dead, even if its variable is still in scope, and a dead slice does not stop its array from shrinking:

```
fn main() {
    var line: u8[>..] = [];
    line.append("honk honk");
    let first = line[..4];
    print(first);               // first's last
        use
    line.clear();               // fine
    line.append("hiss");
    print(line);
}
```

```
honk
hiss
```

Swap the last two uses around, so that `first` is printed after the `clear`, and the compiler objects, naming both the array and the slice that is still in use:

```
fn main() {
    var line: u8[>..] = [];
    line.append("honk honk");
    let first = line[..4];
    line.clear();
    print(first);
}
```

```
error: cannot clear line while first is still
    used: it may hold a reference or slice into
    it (§5.1)
```

This is why a scratch buffer can be refilled in a loop and still hand out slices of itself on every iteration. Each iteration's slices are dead by the time the next `clear` comes around.

Writability 8.5

Everything is writable unless its type says otherwise. `const T&` is a reference through which the target cannot be written, and `const T[:]` is a read-only slice. Some things are read-only by nature: a string literal is static data, so it is a `const u8[:]`.

Parameters are **generic over constness**. A parameter declared `u8[:]` accepts both a writable slice and a read-only one, and the function is checked separately for each. Writing through the parameter is fine for a writable argument, and an error, reported at the call, for a read-only one:

```
fn shout(s: u8[:]) {
    for &c in s { if c >= 'a' && c <= 'z' { c -=
        32; } }
}

fn main() {
    var buf: u8[>..] = [];
    buf.append("honk");
    shout(buf);                    // fine: buf is
        writable
    shout("hiss");                 // a string
        literal is not
}
```

```
error: cannot assign through this path (const,
    or a read-only instantiation, §9.5)
in fn shout(u8[:]) instantiated from
    main.goose:9
```

The second line of the message points at the call that caused the problem. Remove that call and the program runs, turning `honk` into `HONK`.

So you rarely write `const` on a parameter. You write it when you want to *promise* that a function only reads, which also lets the compiler catch an accidental write inside the function itself. The places where `const` is required are *slots* that store references: a struct field, an array element or a global that is meant to hold read-only data, such as a string literal, must say so in its type. Chapter 9 has examples.

Rebinding 8.6

A reference variable that you rebind with `.=` stays tied to the root it started with, or to another root at the same scope depth. In practice that means you can walk around inside a structure, or move between sibling variables, which is what rebinding is for:

```
fn main() {
    var a: i64[>..] = [1, 2];
    var b: i64[>..] = [3, 4];
    var cursor .= a[0];
    cursor .= a[1];
    cursor .= b[0];              // another local
        at the same depth: fine
    print(cursor);
}
```

```
3
```

Anything more adventurous, such as pointing an outer variable at something from an inner scope, is the "storing a reference" error from before.

Exercises 8.7

1. Write a function `largest(xs: i64[:]) -> i64&` that returns a reference to the largest element, and use it to double the largest element of `[3, 9, 4]` in place.

2. Write `fn middle(s: u8[:]) -> u8[:]` that returns its argument without its first and last bytes (or an empty slice if it is shorter than 2). Call it on a string literal and on a builder. Why is it fine to return a slice here, when `first_three` above was rejected?

3. Change the `line` example above so that it prints `first` after the `clear`, but compiles. (There is more than one way.)

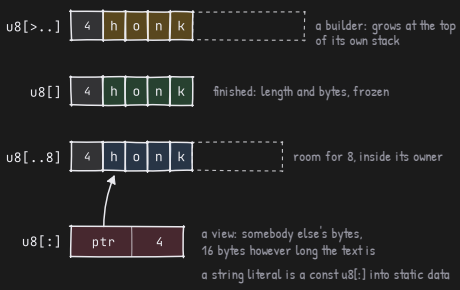
Chapter 9

Strings Are Bytes

Goose has no string type. A string is an array of `u8`, and which member of the array family you choose says what you want to do with it. This turns out to be a good fit for the memory model: building text, storing text inside records and cutting text into pieces each get the representation that suits them, and none of them needs a heap.

Four kinds of string 9.1

Type	Use it for
`u8[>..]`	building text: appending is a pointer bump
`u8[]`	finished text that owns its bytes, such as a name in a record
`u8[..k]`	short text inside a struct, with room for `k` bytes
`u8[:]`	a view of bytes that live somewhere else



The four everyday kinds of string. Three of them hold their bytes; the slice only views bytes held elsewhere.

A **string literal** is a view: a `const u8[:]` of bytes in the program's read-only data. It is read-only, which is why some slots have to be declared `const` before they can hold a literal, as you will see below.

No encoding is enforced. Strings are bytes, and the standard library treats them as UTF-8 by convention where it matters.

> **From C**
>
> There is no NUL terminator. A slice knows its length, and so does every array kind, so `strlen` has no reason to exist. When you call C you append the `\0` yourself (Chapter 22).
>
> **From Odin**
>
> `u8[:]` plays the role of Odin's `string` and `[]u8`, which in Goose are the same thing. A builder plays the role of `strings.Builder`, but it needs no allocator and no `defer strings.builder_destroy`.

Building text 9.2

A `u8[>..]` is a string builder. `append` adds the bytes of another string, and `push` adds a single byte. The built-in `format` appends the text form of any values, exactly as `print` would write them:

```
fn main() {
    var s: u8[>..] = [];
    s.append("honk");
    s.push('!');
    format(s, " x", 3, ", weight ", 4.5);
    print(s, " [", s.len, " bytes]");
}
```

```
honk! x3, weight 4.5 [20 bytes]
```

`str(…)` takes the same arguments as `print`, and returns the text as a new builder. Like every value in Goose it is built directly in its destination, so `names.push(str("goose #", i))` writes the text straight into the new element:

```
fn main() {
    let msg = str("eggs: ", 12, ", geese: ", 3);
    print(msg, " (", msg.len, ")");
}
```

```
eggs: 12, geese: 3 (18)
```

Remember that a character literal is an integer. `str('g')` is the text `"103"`, not `"g"`; to add one byte, use `push`.

Literals and escapes 9.2.1

String literals use the escapes you know: `\n`, `\t`, `\r`, `\0`, `\\`, `\"`, `\'`, and `\xNN` for any byte. A *raw*

string is written between `"""` delimiters and has no escapes at all. A multi-line raw string has its closing delimiter's indentation removed from every line, so it can be indented with the code around it:

```
fn main() {
    print("tab\there, quote \" and \x41");
    print("""C:\geese\list.txt""");
    print("""
        Usage: honk [-n count]
          -n   how many times
        """);
}
```

```
tab here, quote " and A
C:\geese\list.txt
Usage: honk [-n count]
  -n   how many times
```

Storing text 9.3

Text that is finished and needs a home goes into a `u8[]` or a `u8[..k]`. Initializing one from any string, including a literal or a slice, copies the bytes into it, so the new value owns its text:

```
fn main() {
    var draft: u8[>..] = [];
    draft.append("Ada of the pond");
    let name: u8[] = draft[..3];     // three
        bytes of its own
    draft.clear();                   // fine:
        name does not point into draft
    print(name);
}
```

```
Ada
```

A `u8[..k]` is the natural choice inside a struct, because it keeps the struct fixed-size, which in turn keeps an array of such structs indexable and sortable. A limited array is also a builder up to its capacity, so you can `format` into one:

```
struct Goose { name: u8[..12], eggs: i32 }

fn main() {
    var g = Goose { name: "Ada", eggs: 0 };
    format(g.name, " Jr.");
    print(g.name, " ", g.name.len, "/",
        g.name.cap);
}
```

```
Ada Jr. 7/12
```

When a struct holds a *view* rather than its own bytes, its field type says whether the view may be read-only. A field declared `u8[:]` holds a writable view, and a string literal cannot go there. Declare it `const u8[:]` and it can:

```
struct Label { text: const u8[:] }

fn main() {
    let l = Label { text: "fragile: contains
        eggs" };
    print(l.text);
}
```

```
fragile: contains eggs
```

Printing values inside text 9.4

Every value has a text form. Strings print as their bytes, but a string *inside* another value, such as

an array or a struct, is quoted and escaped, so you can tell where it begins and ends:

```
struct Goose { name: u8[..12], eggs: i32 }

fn main() {
    let name: u8[] = "Ada";
    print(name, " ", [name], " ", Goose { name:
        "Bea", eggs: 2 });
}
```

```
Ada ["Ada"] Goose { "Bea", 2 }
```

To give your own type a text form of its own, write an overload of `format` that takes the type. `print`, `str` and `format` all use it, anywhere the type appears, including inside arrays:

```
struct Goose { name: u8[..12], weight: f32 }

fn format(out: u8[>..]&, g: Goose) {
    format(out, g.name, " (", g.weight, " kg)");
}

fn main() {
    let flock = [Goose { "Ada", 4.5 }, Goose {
        "Bea", 3.75 }];
    print(flock);
    print("leader: ", flock[0]);
}
```

```
[Ada (4.5 kg), Bea (3.75 kg)]
leader: Ada (4.5 kg)
```

From Rust

This is Goose's `impl Display`. There is no trait: the overload is found by name, in the namespace where the type is declared. Chapter 19 covers namespaces.

Cutting text up 9.5

The text functions are in the `std` module, so they need `import std;`. Almost all of them take a `u8[:]` and return slices of it. **Cutting text up copies nothing**: the pieces are views into the original bytes.

```
import std;

fn main() {
    let line = "  name = Ada Lovelace  ";
    let body = trim(line);
    let eq = find(body, "=");
    print("[", trim(body[..eq]), "] [",
        trim(body[eq + 1..]), "]");
}
```

```
[name] [Ada Lovelace]
```

`find` returns the position of the first occurrence, or `-1`. `trim`, `trim_start` and `trim_end` remove ASCII whitespace by returning a smaller slice.

`split` cuts a string at a separator and returns a growable array of slices; `words` splits at runs of whitespace and drops the empty pieces:

```
import std;

fn main() {
    let parts = split("Ada,4.2,7", ',');
    print(parts.len, " parts: ", parts);
    print(words("  waddle   to  the pond "));
}
```

```
3 parts: ["Ada", "4.2", "7"]
["waddle", "to", "the", "pond"]
```

Each part takes sixteen bytes, a pointer and a length. None of the text was copied. If you do not need the array at all, `each_split` and `each_word` call a block for each piece instead. The `{ … }` after the call is a *trailing block*, a small function passed to `each_split`, with `it` as its argument. Chapter 16 covers them properly.

```
import std;

fn main() {
    each_split("north-east-south", '-') {
        print("go ", it); };
}
```

```
go north
go east
go south
```

Converting to and from numbers 9.6

`parse_int` and `parse_flt` read a number from the whole of a string, and return it together with a `bool` that says whether the text was valid:

```
import std;

fn main() {
    let n, ok = parse_int("7");
    let w, wok = parse_flt("4.2");
    let bad, bok = parse_int("seven");
    print(n * 2, " ", ok, " ", w, " ", wok, " ",
        bad, " ", bok);
}
```

```
14 true 4.2 true 0 false
```

Going the other way, `format` writes numbers in their natural form. When you want control over it, `format_int` takes a base, a minimum width and a padding byte, and `format_flt` a number of decimals:

```
import std;

fn main() {
    var out: u8[>..] = [];
    format_int(out, 255, 16, 6, '0');
    out.append("  ");
    format_flt(out, 3.14159, 2);
    print(out);
}
```

```
0000ff  3.14
```

Comparing and changing text 9.7

Strings compare with `==` like any arrays, and this works across kinds: a builder against a literal, a `u8[..12]` field against a slice. For ordering, `compare(a, b)` returns `-1`, `0` or `1`. `to_lower` and `to_upper` change ASCII letters in place, so they need a writable string. `join` writes several strings into a builder with a separator between them:

```
import std;

fn main() {
    var call: u8[>..] = [];
    call.append("Honk");
    to_upper(call);
    var out: u8[>..] = [];
    join(out, ["ada", "bea", "cleo"], ", ");
    print(call, " ", call == "HONK", " ",
        compare("ada", "bea"), " ", out);
}
```

```
HONK true -1 ada, bea, cleo
```

Bytes and characters 9.8

Indexing a string gives a byte, a `u8`, which prints as a number. A string literal's `.len` is its length in bytes, which is not the number of characters once UTF-8 is involved. `each_utf8` decodes the code points for you, and `push_utf8` appends one:

```
import std;

fn main() {
    let word = "gänse";
    var points = 0;
    each_utf8(word) { points += 1; };
    print(word[0], " ", points, " code points in
        ", word.len, " bytes");
}
```

```
103 5 code points in 6 bytes
```

Exercises 9.9

1. Write a function `initials(full_name: u8[:]) -> u8[>..]` that returns the first letter of every

word, so that `initials("ada byron lovelace")` is `"abl"`.

2. Parse the line `"Ada,4200;Bea,3900;Cleo,4600"` into the total weight of all three geese. Use `split` twice, or `each_split` twice.

3. Write a `format` overload for `struct Egg { day: i32, grams: f32 }` that prints an egg as `day 3: 61.5 g`, and print an array of two eggs.

Chapter 10

Structs and Inline Data

A Goose struct is a named, packed sequence of fields. That part is familiar. What is new is that a struct can hold variable-size data, such as strings and lists, *inline*, with no pointers to anywhere else. This chapter covers structs from the basics up to records that are just runs of bytes.

Declaring and building structs 10.1

```
struct Goose { name: u8[..12], weight: f32 =
    4.0, eggs: i32, tagged: bool }

fn main() {
    let a = Goose { name: "Ada", eggs: 3,
        tagged: true };   // named fields
    let b = Goose { "Bea", 3.5, 1, false };
        // positional
    let c = Goose { name: "Cleo", .. };
        // the rest default
    print(a);
    print(b);
    print(c);
}
```

```
Goose { "Ada", 4.0, 3, true }
Goose { "Bea", 3.5, 1, false }
Goose { "Cleo", 4.0, 0, false }
```

There are two ways to write a struct literal. The named form gives fields by name, in any order, and may leave out any field that has a declared default, like `weight` here. The positional form lists the fields in declaration order, and may

leave out defaulted fields only at the end. The two forms do not mix.

Ending a literal with `..` lets you leave out *any* field. A field without a declared default then takes its type's zero value: `0`, `false`, an empty string or array, or `null`. Without the `..`, leaving out a field that has no default is an error:

```
struct Goose { name: u8[..12], eggs: i32 }

fn main() {
    let g = Goose { name: "Ada" };
}
```

```
error: missing initializer for field eggs of
    Goose (it has no default)
```

A field's default is an expression evaluated each time a literal leaves the field out. It can call functions and read globals, so a default can hand out serial numbers, for example. `default<T>()` builds a `T` with every field at its default, which is useful in generic code.

A global `var` declared with only a type starts out with every field at its default, so `var settings: Config;` at the top of a file needs no initializer.

> **From Rust**
>
> `Goose { name: "Cleo", .. }` looks like Rust's struct update syntax, `Goose { name: …, ..Default::default() }`, but there is nothing to spread: the `..` simply asks for defaults. There is no `Default` trait to implement.

Structs are nominal 10.1.1

Two structs with the same fields are different types:

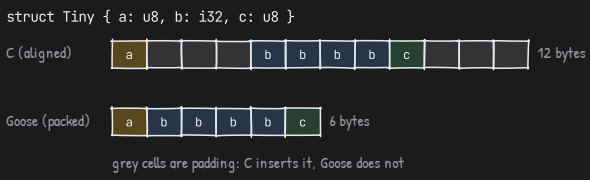
```
struct Hen { eggs: i64 }
struct Goose { eggs: i64 }

fn main() {
    let h = Hen { 3 };
    let g: Goose = h;
}
```

```
error: expected a value of type Goose, got Hen
```

This is the way to make a type that is "an `i64`, but not just any `i64`": wrap it in a one-field struct.

Packed layout 10.2

Goose lays out a struct's fields in declaration order with **no padding**. Unaligned access is assumed to be cheap, as it is on every current desktop and phone CPU. A struct holding a `u8`, an `i32` and another `u8` takes 6 bytes, where C would use 12:



C pads fields to their alignment. Goose packs them back to back.

```
struct Tiny { a: u8, b: i32, c: u8 }

fn main() {
    var two: Tiny[>..] = [Tiny { 1, 2, 3 }, Tiny
        { 4, 5, 6 }];
    print(bytes_of(two).len, " bytes for two");
}
```

```
12 bytes for two
```

`bytes_of(a)` gives a read-only view of an array's elements as bytes, which is handy for seeing exactly how much space data takes. It comes back in Chapter 21.

When a struct must match a layout defined elsewhere, such as a C struct or a file format, write the padding yourself. `pad n` inserts `n` unused bytes, and a bare `pad` aligns the next field to its own size:

```
struct Record { tag: u8, pad, size: u32 }

fn main() {
    var rs: Record[>..] = [Record { 1, 512 }];
    print(rs[0], " in ", bytes_of(rs).len, "
        bytes");
}
```

```
Record { 1, 512 } in 8 bytes
```

From C

A Goose struct is a C struct with `__attribute__((packed))` or `#pragma pack(1)`. Odin's `#packed` is the same thing.

Field qualifiers 10.3

A field declared with `let` cannot be assigned after the struct is built, and a field declared `const` cannot have its contents written either:

```
struct Tag { id: u32, let issued: u32 }

fn main() {
    var t = Tag { 7, 2026 };
    t.id = 8;             // fine
    t.issued = 2027;
}
```

```
error: cannot assign to let issued (§4.4)
```

Variable-size fields 10.4

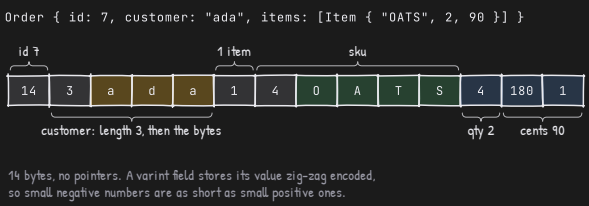
So far every field has been fixed-size. A struct can also hold variable-size values: a finished string `u8[]`, a frozen array `T[]`, or a `varint`. They sit *inline*, in declaration order, which makes the whole struct variable-size.

`varint` deserves an introduction. It is an integer type for storage only. A `varint` field takes one byte for small values and more for large ones, up to ten bytes for the largest. Reading one gives an `i64`. Combined with `u8[varint]` and `T[varint]`, whose lengths are varints too, it lets you describe records that take exactly as many bytes as their contents need:

```
struct Item  { sku: u8[varint], qty: varint,
    cents: varint }
struct Order { id: varint, customer: u8[varint],
    items: Item[varint] }

fn main() {
    var book: Order[>..] = [];
    book.push(Order { id: 1001, customer: "ada",
                      items: [Item { sku: "CORN-
                          1", qty: 2, cents: 350
                          },
                              Item { sku:
                                  "OATS", qty:
                                  1, cents: 900
                                  }] });
    book.push(Order { id: 1002, customer: "bea",
                      items: [Item { sku: "CORN-
                          1", qty: 5, cents: 350
                          }] });
    for o in book {
        var total = 0;
        for it in o.items { total += it.qty *
            it.cents; }
        print(o.customer, " owes ", total, "
            cents");
    }
    print(bytes_of(book).len, " bytes for the
        whole order book");
}
```

```
ada owes 1600 cents
bea owes 1750 cents
42 bytes for the whole order book
```

Each order is its fields laid end to end, including its list of items, and the whole order book is one block. Here is a smaller order, byte by byte:



A record with variable-size fields is a run of bytes: each length comes just before the data it describes.

In C, Odin or Rust, `Order` would be a struct holding a pointer to a customer string and a pointer to an array of items, each of which holds a pointer to its own SKU string. Building it would take several allocations, and the struct itself would be dozens of bytes before you counted any of the data.

The cost of this layout is that the fields after a variable-size one are found by walking, and an array of such records is *sequential*: you can iterate it but not index it, as you saw in Chapter 7. Variable-size fields are also **frozen once built**. A `varint` field, for example, cannot be assigned, because a larger value might need more bytes:

```
struct Order { id: varint, customer: u8[varint]
    }

fn main() {
    var book: Order[>..] = [Order { 1, "ada" }];
    for o in book { o.id = 2; }
}
```

```
error: varint fields are written only at
    construction (§3.6)
```

Fixed-size fields of a variable-size struct can still be assigned. Note that the loop above wrote `for o in book`, not `for &o in book`: elements that are not fixed-size are never copied, so `for` binds them by reference already, and the compiler warns that an `&` would be redundant. Use variable-size records for data you build once and then traverse: log entries, parsed documents, messages. Use fixed-size records, with `u8[..k]` for their strings, for data you index, sort and update.

A growable tail 10.5

A struct's *last* field may be a growable array. The struct then becomes resizable itself: it can be a local, a global or a by-value parameter, but not an array element, and it can have only one such field. The tail grows through the struct like any growable array:

```
struct Barn { name: u8[..8], log: u8[][>..] }

fn note(b: Barn&, line: u8[:]) {
    b.log.push(line); }

fn main() {
    var barn = Barn { name: "north", log: [] };
    note(barn, "fed the geese");
    note(barn, "collected 4 eggs");
    print(barn.name, ": ", barn.log);
}
```

```
north: ["fed the geese", "collected 4 eggs"]
```

Generic structs 10.6

A struct can take type parameters, which are filled in when it is used. They can usually be inferred from the field values:

```
struct Pair<T> { a: T, b: T }

fn main() {
    let weights = Pair<i64> { 4200, 3900 };
    let ratios = Pair { a: 0.5, b: 1.5 };
    print(weights, " ", ratios);
}
```

```
Pair<i64> { 4200, 3900 } Pair<f64> { 0.5, 1.5 }
```

Arithmetic and equality 10.7

`==` and `!=` compare structs field by field. The arithmetic operators work field by field too, on two values of the same struct whose fields are all numbers of one kind, and `*` and `/` also accept a single number on one side. This is how the standard library's vector types work, without any operator overloading:

```
struct V2 { x: f32, y: f32 }

fn main() {
    let a = V2 { 1.0, 2.0 };
    let b = V2 { 0.5, 0.5 };
    let c = a + b;
    print(c, " ", c * 2.0, " ", -c, " ", c == V2
        { 1.5, 2.5 });
}
```

```
V2 { 1.5, 2.5 } V2 { 3.0, 5.0 } V2 { -1.5, -2.5
    } true
```

The same memberwise arithmetic works on fixed arrays: `[1, 2] + [10, 20]` is `[11, 22]`. There are no `<` or `>` on structs.

Exercises 10.8

1. Declare `struct Nest { owner: u8[..12], eggs: i32 = 0, warm: bool = true }` and build three nests using the named, positional and `..` forms.
2. Using `bytes_of`, find out how many bytes these two structs take. Explain the difference.

```
goose check struct A { flag: bool, count: i64, small: u8 } struct B { count: i64, flag: bool, small: u8 }
```

3. Change `Order` so that each order also records a `paid: bool`. Where in the struct should the field go, so that it can be read without walking past the variable-size fields? Can it be assigned after the order is built?

Chapter 11

Optionals

References are never null. That is a good property, but sometimes "there is nothing here" is exactly what you need to say: the search found nothing, the list has ended, the field has not been set. For those cases Goose has the optional, `T?`, which is a reference that *may* be null. This chapter is short, because optionals are simple, but they appear in almost every program.

A reference that may be null 11.1

`T?` is an optional `T`: either a reference to a `T`, or `null`. It is represented as a plain address, with null as address 0, so it takes no more space than a reference. A function that searches for something returns one:

```
struct Goose { name: u8[..12], weight: f32 }

fn find_heavier(flock: Goose[:], than: f32) ->
    Goose? {
    for &g in flock {
        if g.weight > than { return g; }
    }
    return null;
}

fn main() {
    var flock: Goose[>..] = [Goose { "Ada", 4.5
        }, Goose { "Bea", 5.25 }];
    let big = find_heavier(flock, 5.0);
    if big { print("found ", big.name); } else {
        print("nobody"); }
    let huge = find_heavier(flock, 9.0);
    if huge { print("found ", huge.name); } else
        { print("nobody"); }
}
```

```
found Bea
nobody
```

Because an optional is a reference, the result *is* the element of `flock`, not a copy of it. You can write through it once you know it is not null.

`T?` composes with any type. `i64?` is an optional reference to an `i64`, and an optional struct field that has no default is null when a literal leaves it out.

From Rust

`Goose?` is `Option<&mut Goose>`, with the same "null pointer optimization": `None` is the zero address. There is no optional by value, so there is no `Option<i64>` in the sense of "an `i64` or nothing". For that, Goose code returns `i64, bool`.

From C

> An optional is the C pointer that may be `NULL`, with one difference that matters: you cannot dereference it until the compiler has seen you check it.

Narrowing 11.2

An optional is **not transparent**. You cannot use it as its target until you have checked that it is not null:

```
fn first_over(xs: i64[:], min: i64) -> i64? {
    for &x in xs { if x > min { return x; } }
    return null;
}

fn main() {
    let xs: i64[3] = [3, 8, 5];
    let big = first_over(xs, 4);
    print(big + 1);
}
```

```
error: operator + cannot be applied to i64? and
    i64
```

Checking it *narrows* it. Inside the region where the compiler knows it is not null, the optional behaves as a plain reference: transparent and writable. These all narrow:

- `if o { … }`, for the `if` branch, and `if !o { … } else { … }` for the `else` branch;
- `while o { … }`, for the loop body;
- `guard o else { … }`, for the rest of the block;
- `assert(o)`, for the rest of the block;
- a comparison with `null`, as in `if o != null`.

```
fn first_over(xs: i64[:], min: i64) -> i64? {
    for &x in xs { if x > min { return x; } }
    return null;
}

fn main() {
    var xs: i64[3] = [3, 8, 5];
    let big = first_over(xs, 4);
    if big { big = 0; }                // writes
        xs[1]
    print(xs);
    let small = first_over(xs, 2);
    assert(small);                     // stops
        the program if it is null
    print(small + 1);
}
```

```
[3, 0, 5]
4
```

`guard` reads especially well with optionals, because it lets the main path continue at the same indentation:

```
struct Goose { name: u8[..12], weight: f32 }

fn weight_or_zero(g: Goose?) -> f32 {
    guard g else { return 0.0; }
    g.weight                           // g is a
        Goose& from here on
}

fn main() {
    var ada = Goose { "Ada", 4.5 };
    print(weight_or_zero(ada), " ",
        weight_or_zero(null));
}
```

```
4.5 0.0
```

Note how `ada`, a plain variable, is passed where a `Goose?` is expected. A variable binds to an

optional parameter as it would to a reference parameter, with no `&`.

Narrowing ends when you rebind 11.2.1

Narrowing is a property of a region of code, so it ends when the optional might have changed. Rebinding the optional with `.=` ends it, even to something that is obviously not null, and setting it to `null` certainly does:

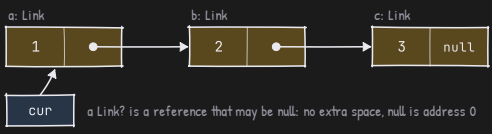
```
fn main() {
    var n = 5;
    var o: i64? = n;
    if o {
        o .= null;
        print(o + 1);
    }
}
```

```
error: operator + cannot be applied to i64? and
    i64
```

The exception is a `while` loop whose condition is the test. The condition runs again before every iteration, so the body can move the optional along and still find it narrowed on the next round. That is how you walk a chain of links:

```
struct Link { val: i64, next: Link? }

fn main() {
    var c = Link { val: 3, next: null };
    var b = Link { val: 2, next: c };
    var a = Link { val: 1, next: b };
    var cur: Link? = a;
    var sum = 0;
    while cur {
        sum += cur.val;
        cur .= cur.next;
    }
    print(sum);
}
```

```
6
```



Each next field is an optional link. The cursor starts at a and moves along until it reaches null.

`cur .= cur.next` rebinds the cursor to the next link, which may be null. The `while` re-tests it before the next iteration.

Optionals as fields 11.3

An optional field with no declared default is `null` when a literal leaves it out, and prints as `null`:

```
struct Nest { eggs: i64, mother: Nest? }

fn main() {
    var n = Nest { eggs: 3, .. };
    print(n);
    var m = Nest { eggs: 0, .. };
    n.mother .= m;
    let mom = n.mother;
    if mom { print("has a mother with ",
        mom.eggs, " eggs"); }
}
```

```
Nest { 3, null }
has a mother with 0 eggs
```

Notice the extra step: the field is copied into a local, `mom`, and the local is narrowed. The compiler narrows variables, not arbitrary expressions, because a field could be changed by something else between the check and the use. Copying an optional is cheap, since it is just an address.

The rule from Chapter 8 still applies: a reference stored in a field must not outlive what it points at. Here `n` and `m` live in the same scope, so storing `m` in `n.mother` is fine. Chapter 13 shows the usual way to build linked structures, where all the nodes live in one array and the links point within it.

Optionals or a trailing `bool`? 11.4

Goose has two common ways to say "maybe": an optional, and a trailing `bool` result. The choice follows from what you are returning.

- If the answer is *something that already exists*, such as an element of an array, a node in a tree or a value in a table, return an optional. It is a reference to the thing, so nothing is copied and the caller can modify it.
- If the answer is *a new value*, such as a parsed number or a computed result, return the value and a `bool`, like `parse_int` does. An optional cannot point at a value that was only just computed, because the value would die with the function.

Exercises 11.5

1. Write `fn find_name(flock: Goose[:], name: u8[:]) -> Goose?` and use it to rename one goose in an array, if it is there.
2. Write a function that takes an `i64[:]` and returns an optional reference to its *last* negative element. Test it on an array with negative numbers and on one without.
3. Why does this not compile, and how would you fix it?

```goose
error fn make() -> i64? { var x = 5; return x; }

fn main() { print(make() != null); }
```

```
error error: returning a reference rooted in x, which dies with this function (§9.2)
```

PART III

Modelling Data

With the memory model in place, this part turns to shaping data: values that can be one of several kinds, structures linked together without pointers, data whose lifetimes do not follow the shape of the code, and the ways a program reports that something went wrong.

Chapter 12

Enums in Two Sizes

Goose's `enum` is an algebraic data type: a value that is one of several *variants*, each of which may carry its own fields. If you know Rust's `enum` you know most of this chapter already. The new part is that every enum can be stored in two ways, and you choose between them each time you use the type.

Enums are also the *only* form of dynamic dispatch in Goose. There are no classes, traits, interfaces or vtables. When a program needs "one of several kinds of thing", it uses an enum.

Declaring enums 12.1

An enum lists its variants. A variant can carry fields, declared like a struct's, or nothing at all:

```
enum Event {
    Honk { times: i32 },
    Lay { grams: f32 },
    Moult,
    Migrate { km: f64, south: bool },
}

fn main() {
    var log: Event[>..] = [];
    log.push(Event.Honk { times: 5 });
    log.push(Event.Lay { grams: 61.5 });
    log.push(Event.Moult);
    log.push(Event.Migrate { km: 800.0, south:
        true });
    print(log);
}
```

```
[Honk { 5 }, Lay { 61.5 }, Event.Moult, Migrate
    { 800.0, true }]
```

A variant is written `Enum.Variant { … }`, or just `Enum.Variant` when it has no fields. The payload's fields have the same rules as a struct's, including defaults and the `..` form.

An enum whose variants carry nothing is a C-style enumeration, and takes one byte:

```
enum Mood { Calm, Hungry, Angry }

fn main() {
    var moods: Mood[>..] = [Mood.Calm,
        Mood.Angry];
    print(moods, " in ", bytes_of(moods).len, "
        bytes, ", moods[1] == Mood.Angry);
}
```

```
[Mood.Calm, Mood.Angry] in 2 bytes, true
```

Matching 12.2

`match` takes an enum value apart. Each arm names a variant, and may bind its payload to a name:

```
enum Event { Honk { times: i32 }, Lay { grams:
    f32 }, Moult, Migrate { km: f64, south: bool
    } }

fn describe(e: Event) -> u8[:] {
    match e {
        Honk h => if h.times > 3 { "a racket" }
            else { "a honk" },
        Lay l => "an egg",
        Moult => "feathers everywhere",
        Migrate m => if m.south { "heading
            south" } else { "heading north" },
    }
}

fn noisy(e: Event) -> bool {
    match e { Honk, Migrate => true, _ => false
        }
}

fn main() {
    let e = Event.Honk { times: 5 };
    print(describe(e), " ", noisy(e), " ",
        describe(Event.Moult), " ",
        noisy(Event.Moult));
}
```

```
a racket true feathers everywhere false
```

An arm can list several variants, as `Honk, Migrate` does. Such an arm cannot bind a payload, since each variant's payload has a different type. `_` matches whatever is left.

A `match` over an enum must be **exhaustive**: every variant needs an arm, unless there is a `_`. When you add a variant to an enum, every `match` that does not handle it stops compiling:

```
enum Event { Honk { times: i32 }, Lay { grams:
    f32 }, Moult }

fn describe(e: Event) -> u8[:] {
    match e { Honk h => "a honk", Lay l => "an
        egg" }
}

fn main() { print(describe(Event.Moult)); }
```

```
error: match does not cover variant Event.Moult
```

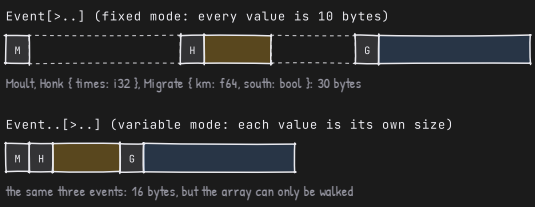
From Odin

This is Odin's `union` together with a `switch v in x` type switch, but the variants have names of their own rather than being distinguished only by type, and the compiler checks that the `match` is complete. Odin's `#partial switch` is Goose's `_ => {}`.

Fixed mode and variable mode 12.3

Here is where Goose's enums differ from every other language's. The `Event` above has variants of very different sizes. `Moult` carries nothing, `Honk` carries four bytes, and `Migrate` carries nine. Every value can be stored in two ways:

- **Fixed mode**, written plainly as `Event`, stores a tag and enough room for the *largest* payload. Every `Event` is 10 bytes, even a `Moult`. This is how Rust, C++ and Odin store their tagged unions.
- **Variable mode**, written `Event..` with two trailing dots, stores each value at exactly its own size: 1 byte for a `Moult`, 5 for a `Honk`, 10 for a `Migrate`.



The same three events in fixed mode, where each takes the room of the largest variant, and in variable mode, where each takes only its own.

```
enum Event { Honk { times: i32 }, Lay { grams:
    f32 }, Moult, Migrate { km: f64, south: bool
    } }

fn main() {
    var fixed: Event[>..] = [];
    var packed: Event..[>..] = [];
    for i in 2 {
        fixed.push(Event.Moult);
        fixed.push(Event.Honk { times: 1 });
        packed.push(Event.Moult);
        packed.push(Event.Honk { times: 1 });
    }
    print(bytes_of(fixed).len, " bytes fixed, ",
        bytes_of(packed).len, " bytes packed");
}
```

```
40 bytes fixed, 12 bytes packed
```

Each mode offers a guarantee the other cannot.

Fixed mode values all have the same size, so an array of them can be indexed, and a value can be overwritten with a *different variant* in place:

```
enum Event { Honk { times: i32 }, Lay { grams:
    f32 }, Moult }

fn main() {
    var log: Event[>..] = [Event.Honk { times: 2
        }, Event.Moult];
    log[1] = Event.Lay { grams: 70.0 };
    print(log);
}
```

```
[Honk { 2 }, Lay { 70.0 }]
```

The price is that **nothing may point into a fixed-mode payload**. If you held a reference to the `times` field of a `Honk`, and the value was then overwritten with a `Lay`, the reference would see a float where it expected an integer. So `match` binds a fixed-mode payload only by value, as a copy:

```
enum Event { Honk { times: i32 }, Moult }

fn main() {
    var log: Event[>..] = [Event.Honk { times: 1
        }];
    for e in log {
        match e { Honk &h => { h.times += 1; },
            _ => {} }
    }
}
```

```
error: cannot bind the payload of fixed-mode
    Event by reference (§3.5); bind by value:
    Honk h
```

Variable mode is the mirror image. Values are packed against each other, so a value can never change to a larger variant, and an array of them is sequential: it can be walked but not indexed.

In exchange, you may take references into a payload. `match` arms can bind with `&` and edit the payload in place:

```
enum Note { Text { body: u8[varint] }, Weight {
    kg: f32 }, Blank }

fn main() {
    var notes: Note..[>..] = [];
    notes.push(Note.Text { body: "fed at dawn"
        });
    notes.push(Note.Weight { kg: 4.5 });
    notes.push(Note.Blank);
    for n in notes {
        match n {
            Weight &w => { w.kg += 0.25; },
            _ => {},
        }
    }
    print(notes, " in ", bytes_of(notes).len, "
        bytes");
}
```

```
[Text { "fed at dawn" }, Weight { 4.75 },
    Note.Blank] in 19 bytes
```

Notice the `Text` variant. Its payload is a variable-size string, so it has no "largest size" that fixed mode could reserve. An enum with a variable-size payload can *only* be used in variable mode:

```
enum Note { Text { body: u8[varint] }, Blank }

fn main() {
    var notes: Note[>..] = [];
}
```

```
error: enum Note has non-fixed-size payloads
    and can only be used in variable mode
    (Note..)
```

The mode is chosen where the type is *used*, not where it is declared. The same `enum` can be stored in fixed mode in one array, so you can index and update it, and in variable mode in another, to save space.

> **From Rust**
>
> A Rust `enum` is always fixed mode, so a `Vec<Event>` costs the size of the largest variant per element. The usual fix is to `Box` the big variants, which adds an allocation and a pointer. Variable mode gets the same saving with no allocation at all, at the cost of indexing.

Case functions 12.4

A `match` puts all the handling of an enum in one place. Sometimes you would rather have one function per variant, the way you would have one method per subclass in an object-oriented language. Goose lets you write exactly that. **Each variant is a type of its own**, written `Event.Honk`, and you can overload a function on the variant types:

```
enum Event { Honk { times: i32 }, Lay { grams:
    f32 }, Moult, Migrate { km: f64, south: bool
    } }

fn energy(e: Event.Honk) -> f64 { e.times * 2.0
    }
fn energy(e: Event.Lay) -> f64 { 50.0 }
fn energy(e: Event.Moult) -> f64 { 10.0 }
fn energy(e: Event.Migrate) -> f64 { e.km * 0.5
    }

fn main() {
    var log: Event[>..] = [];
    log.push(Event.Honk { times: 5 });
    log.push(Event.Lay { grams: 61.5 });
    log.push(Event.Migrate { km: 800.0, south:
        true });
    var total = 0.0;
    for e in log { total += energy(e); }
    print(total);
}
```

```
460.0
```

When `energy` is called with a whole `Event`, no overload takes an `Event` directly, so the call dispatches on the tag to the overload for that variant, through a jump table. These *case functions* are checked for completeness just like a `match`: if a variant has no overload, a call with the whole enum does not compile.

```
enum Event { Honk { times: i32 }, Lay { grams:
    f32 }, Moult }

fn energy(e: Event.Honk) -> f64 { 1.0 }
fn energy(e: Event.Lay) -> f64 { 2.0 }

fn main() {
    let e: Event = Event.Moult;
    print(energy(e));
}
```

```
error: no matching overload for call to energy
```

A variant type is an ordinary type, so you can also call `energy` with a value you already know is a `Honk`, and it calls that overload directly:

```
enum Event { Honk { times: i32 }, Moult }

fn energy(e: Event.Honk) -> f64 { e.times * 2.0
    }
fn energy(e: Event.Moult) -> f64 { 10.0 }

fn main() {
    let h = Event.Honk { times: 2 };
    print(energy(h), " ", h.times);
}
```

```
4.0 2
```

This is Goose's answer to virtual functions. It is closed rather than open: you cannot add a new variant from another file without editing the enum. In return, every call is checked for completeness and dispatches with one indexed jump.

Design note: why two modes? 12.5

Allowing a value to change variant and allowing references into its payload are each useful, but together they are unsound. A reference into a payload would outlive the payload's type. Most languages pick one and live with the cost. Rust lets you replace the variant and gets references into payloads by enforcing exclusive borrows, so that nothing else can replace the variant while

the reference is alive. C allows both, and leaves it to you.

Goose makes it a choice of storage. Fixed mode forbids interior references, so replacement is always safe. Variable mode forbids replacement, so interior references are always safe. Because the choice is made at each use of the type, one program can have both: an indexable, editable array of small events, and a packed log of large ones.

Exercises 12.6

1. Declare `enum Feed { Grain { grams: i32 }, Grass, Bread { slices: i32 } }` and write a function that returns how healthy each kind of feed is, as a number from 0 to 10, first with a `match` and then with case functions.
2. Make a `Feed[>..]` and a `Feed..[>..]` holding the same 100 meals, half `Grass` and half `Grain`, and print how many bytes each takes.
3. Using a variable-mode array, write a loop that doubles the `slices` of every `Bread` in place.

Chapter 13

Links Without Pointers

Trees, graphs and linked lists are made of links between values. In C a link is a pointer, which costs eight bytes and means nothing outside the process that created it. In Rust, linked structures are hard enough that the usual advice is to put the nodes in a `Vec` and link them with integer indices.

Goose has a third option. A link can be stored as a **relative reference**: a narrow offset that the compiler turns back into an ordinary reference when you read it. This chapter shows the two kinds of relative reference and the data structures they make easy.

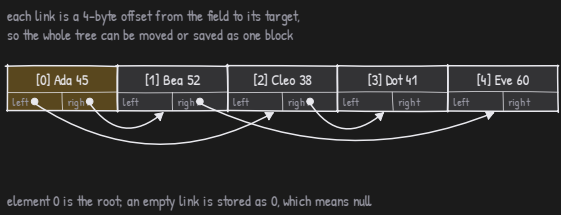
Self-relative links 13.1

`T&<u32>` is a reference to a `T`, stored as a 32-bit offset from the field that holds it to the thing it points at. The width can be `u8`, `u16`, `u32`, `u64` or `varint`. Its optional form, `T&<u32>?`, uses offset 0 as null, so a link costs four bytes including the ability to be empty.

You use a relative reference exactly like any other reference. Reading the field gives you an ordinary `T&`, and storing a reference into the field encodes the offset. Here is a binary search tree of geese, ordered by weight, that lives entirely inside one grow-only array:

```
struct Node { name: u8[..8], weight: i32, left:
    Node&<u32>?, right: Node&<u32>? }

fn insert(tree: Node[>..]&, name: u8[:], weight:
    i32) {
    if tree.len == 0 { tree.push(Node { name:
        name, weight: weight }); return; }
    var cur .= tree[0];
    loop {
        if weight < cur.weight {
            let l = cur.left;
            if l { cur .= l; continue; }
            cur.left .= tree.push(Node { name:
                name, weight: weight });
            return;
        }
        let r = cur.right;
        if r { cur .= r; continue; }
        cur.right .= tree.push(Node { name:
            name, weight: weight });
        return;
    }
}

recursive fn in_order(n: Node&, out: u8[>..]&) {
    let l = n.left;
    if l { in_order(l, out); }
    format(out, n.name, " ");
    let r = n.right;
    if r { in_order(r, out); }
}

fn main() {
    var tree: Node[>..] = [];
    insert(tree, "Ada", 45);
    insert(tree, "Bea", 52);
    insert(tree, "Cleo", 38);
    insert(tree, "Dot", 41);
    insert(tree, "Eve", 60);
    var out: u8[>..] = [];
    in_order(tree[0], out);
    print(out, "| ", tree.len, " nodes in ",
        bytes_of(tree).len, " bytes");
}
```

```
Cleo Dot Ada Bea Eve | 5 nodes in 105 bytes
```



The tree lives in one array. Each link is an offset from the field that holds it to the node it points at.

Look at the line that grows the tree, `cur.left .= tree.push(Node { name: name, weight: weight });`. `push` builds the new node at the end of the array and returns a reference to it, and `.=` stores that reference into `cur.left` as an offset. The push cannot invalidate `cur`, because a grow-only array never moves. In safe Rust this is the point where you would give up on references and switch to indices.

Each node is 21 bytes: a 9-byte name, a 4-byte weight and two 4-byte links. With pointers the links alone would be 16 bytes.

Two features from earlier chapters are doing a lot of work here. The links are optional, so they must be narrowed before use. That is why the code copies `cur.left` into the local `l` and tests it. And the walk is `recursive fn`, which Chapter 17 explains in full. Its growable output is passed in by reference rather than owned by the recursion.

Same root, exactly 13.1.1

A self-relative offset only makes sense if the link and its target are in the same block of memory, so the compiler checks that the reference you store comes from the **same array** as the field you store it in:

```
struct Node { v: i64, next: Node&<u32>? }

fn main() {
    var a: Node[>..] = [Node { v: 1 }];
    var b: Node[>..] = [Node { v: 2 }];
    a[0].next .= b[0];
}
```

```
error: a relative reference must point within
    the same root as its location (§3.9); this
    reference is rooted at b
```

The payoff is that a structure built from self-relative links means the same thing wherever it is in memory. The whole tree above is *position-independent*: you could write its bytes to a file and read them back at a different address, and every link would still be right. Chapter 21 does exactly that.

Values with self-relative links are not copied 13.1.2

The flip side of position independence is that a value *containing* a self-relative link cannot be copied anywhere else, because the copy's offsets would point at the wrong place. The compiler stops you, and suggests what to do instead:

```
struct Node { name: u8[..8], weight: i32, left:
    Node&<u32>?, right: Node&<u32>? }

fn main() {
    var tree: Node[>..] = [Node { name: "Ada",
        weight: 45 }];
    let root = tree[0];
}
```

```
error: copying a value of type Node, which
    contains self-relative references, is not
    supported; construct it in place, or bind a
    reference to it with .=: let root .=
    tree[0]
```

The same goes for `copy(tree)` of the whole array. In practice this is rarely a problem: you build nodes in place with `push`, and you refer to them with references.

Links into a pool 13.2

The second kind of relative reference is measured from the start of a named global array, the *pool*, rather than from the field: `T&<u32 in pool>`. Two things follow from that:

- Arrays *other than the pool* can hold links into it, which self-relative links cannot do.
- A value holding `in pool` links can be copied freely, because its offsets do not depend on where it is.

Here a flock records each goose's mate as a 2-byte link, and a separate index lists the geese in some other order:

```
struct Goose { name: u8[..8], mate: Goose&<u16
    in flock>? }

var flock: Goose[>..] = [];
var by_age: (Goose&<u32 in flock>)[>..] = [];

fn main() {
    let ada .= flock.push(Goose { name: "Ada",
        .. });
    let bob .= flock.push(Goose { name: "Bob",
        .. });
    flock.push(Goose { name: "Cy", .. });
    ada.mate .= bob;
    bob.mate .= ada;
    by_age.push(flock[2]);
    by_age.push(flock[0]);
    by_age.push(flock[1]);
    for g in by_age { print(g.name); }
    let m = flock[1].mate;
    if m { print("Bob's mate is ", m.name); }
    var twin = flock[0];                // a
        copy: fine for in-pool links
    twin.name = "Ada II";
    print(twin.name, " ", flock[0].name);
}
```

```
Cy
Ada
Bob
Bob's mate is Ada
Ada II Ada
```

The array type `(Goose&<u32 in flock>)[>..]` needs parentheses so that the `[>..]` applies to the link type as a whole.

Because the pool is named in the type, it has to be a global. A local array has no name that a type could mention. For links within a local array, use self-relative links. For an index into a local array, use plain references: `(Goose&)[>..]` costs eight bytes per entry.

`self`: links that point back 13.3

A non-optional link can never be null, which raises a puzzle: how do you build the first node of a structure whose links must always point somewhere? There is nothing for them to point at yet.

The answer is `self`. Written as the whole initializer of a relative reference field, it means "the value this literal is building". It is how you make a ring, or the sentinel node of a circular list:

```
struct Ring { id: i64, next: Ring&<u32 in ring>
    }

var ring: Ring[>..] = [];

fn main() {
    ring.push(Ring { id: 0, next: self });
        // a ring of one
    for i in 1..4 {
        let last .= ring[ring.len - 1];
        let n .= ring.push(Ring { id: i, next:
            ring[0] });
        last.next .= n;
    }
    var cur .= ring[0];
    var out: u8[>..] = [];
    for k in 6 { format(out, cur.id, " "); cur
        .= cur.next; }
    print(out);
}
```

```
0 1 2 3 0 1
```

Once the first node exists, every later link has a real target, so no link in the ring is ever null and walking it needs no null checks at all.

An index for a sequential array 13.4

Chapter 7 left a problem open: an array of variable-size elements is compact, but cannot be indexed. Links into a pool solve it. Keep a second array of four-byte links beside the packed one, and push a link each time you push an element:

```
enum Note { Text { body: u8[varint] }, Weight {
    kg: f32 }, Blank }

var notes: Note..[>..] = [];
var note_at: (Note..&<u32 in notes>)[>..] = [];

fn main() {
    note_at.push(notes.push(Note.Text { body:
        "fed at dawn" }));
    note_at.push(notes.push(Note.Weight { kg:
        4.5 }));
    note_at.push(notes.push(Note.Blank));
    for i in [2, 0, 1] {
        match note_at[i] {
            Text t => print(i, ": ", t.body),
            Weight w => print(i, ": ", w.kg, "
                kg"),
            Blank => print(i, ": (blank)"),
        }
    }
}
```

```
2: (blank)
0: fed at dawn
1: 4.5 kg
```

The notes stay packed at their own sizes, and `note_at[i]` reaches any of them with a load and an add.

Choosing a link 13.5

You want	Use
a link within one array, that survives saving and loading	`T&<u32>?`
links from other arrays into a global pool	`T&<u32 in pool>?`
a link that is never null, in a ring or a sentinel	`T&<u32 in pool>` with `self`
a plain reference you hold for a while in a local	`T&`

The offset width is up to you. A `u16` link is enough for a pool of up to 64 KB, and the compiler checks the offset whenever you store into a field that might be too narrow for the array.

From Rust

These links do the job that `Vec<Node>` plus `u32` indices usually does in Rust, but they are typed references rather than integers. You cannot use a link into one pool to index another, and reading one gives you the node, not a number to look up.

From C

A self-relative link is the "offset pointer" or "based pointer" trick used in memory-mapped files and shared-memory structures, with the compiler doing the arithmetic and checking that every target is in the same block.

Exercises 13.6

1. Add a function `count_lighter(tree: Node[>..]&, w: i32) -> i64` to the tree example that counts

the geese lighter than `w` without visiting the whole tree.

2. Build a singly linked list of the numbers 1 to 5 in a local `Link[>..]`, using self-relative `next: Link&<u32>?` links, where each new number is pushed and linked from the previous one. Walk it to print the numbers.

3. In the `flock` example, what would go wrong if `mate` were a self-relative link, `Goose&<u16>?`, and you tried to push `flock[0]` into `by_age`? Try changing `by_age` to hold self-relative links and read the error.

Chapter 14

Pools for Lifetimes That Don't Nest

Everything so far has relied on lifetimes that nest. A value is created in a scope and dies when that scope ends, and anything created after it dies before it. Most data really is like that, which is why data stacks work so well.

Some data is not. Patients join a waiting list and leave in any order. Entries in a cache are evicted by age, not by when they were added. Files in a directory tree are created and deleted at will. For data like this, Goose has the **reusable pool**: a grow-only array whose slots can be handed back and handed out again. This chapter shows how pools work, and why the safety story still holds when you use them.

A pool of slots 14.1

Declare a grow-only array of fixed-size elements `reusable`, and the compiler pairs it with a hidden list of free slots:

Operation	Does
`pool.alloc_index(v) -> i64`	puts `v` in a free slot if there is one, or pushes it; returns the index
`pool.alloc_ref(v) -> T&`	the same, returning a reference
`pool.free(i)`	records slot `i` as free, to be reused by a later `alloc`

```
struct Egg { day: i32, grams: f32 }

fn main() {
    reusable var tray: Egg[>..] = [];
    let a = tray.alloc_index(Egg { 1, 60.0 });
    let b = tray.alloc_index(Egg { 2, 61.0 });
    tray.free(a);
    let c = tray.alloc_index(Egg { 3, 70.0 });
        // takes slot a back
    print(a, " ", b, " ", c, ", ", tray.len, "
        slots");
}
```

```
0 1 0, 2 slots
```

That looks like an allocator. The important thing is what `free` does *not* do.

Freeing frees nothing 14.2

`free` does not release memory, end a lifetime or invalidate anything. It adds an index to the free list, and that is all. The freed slot still holds a valid value of the element type, and it still belongs to the pool, which still lives until its scope ends.

So a reference to a slot that has been freed and reused is not a dangling pointer. It reads the slot's **new** occupant: a perfectly valid value of the right type, just not the one you were thinking of.

```
struct Egg { day: i32, grams: f32 }

fn main() {
    reusable var tray: Egg[>..] = [];
    let first = tray.alloc_index(Egg { 1, 60.0
        });
    let old .= tray[first];
    tray.free(first);
    tray.alloc_index(Egg { 3, 70.0 });
    print(old);
        // the slot's new egg
}
```

```
Egg { 3, 70.0 }
```

This is a logic bug, of exactly the same kind as keeping an index into an array after you have overwritten that element. It is not a memory-safety bug. Nothing is read at the wrong type, nothing goes out of bounds, and nothing is read after it was freed, because nothing ever is. That is the whole extent of the unsafety a pool adds, and it is confined to the data you chose to put in one.

To keep this guarantee, a pool only allows what keeps every slot a valid value forever. Its elements must be fixed-size, and it can never shrink:

```
struct Egg { day: i32 }

fn main() {
    reusable var tray: Egg[>..] = [];
    tray.alloc_index(Egg { 1 });
    tray.pop();
}
```

```
error: cannot pop reusable pool tray: its slots
    stay live for the freelist (§5.4)
```

From Rust

A pool is a slab or arena with a free list, like the `slab` crate, minus the generation counters that some arena crates use to catch stale handles. You can add a generation field to your own elements if you want that check.

From C

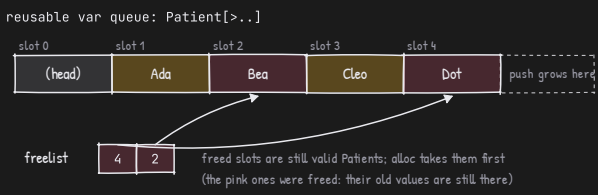
This is the free-list pool that C programmers have always written for fixed-size objects, with one difference that makes it safe. The memory of a freed slot is never returned to anything that could reuse it for another type.

A waiting list 14.3

Here is a pool used for what pools are best at: a structure that is edited in arbitrary order. Geese wait to see the vet in a circular doubly linked list. The list has a *sentinel* node, `head`, which is allocated before `main` runs. Because of the sentinel, no link is ever null, and adding or removing a patient is the same four stores whatever the position.

```
struct Patient { name: u8[..8], prev: Patient&
    <u32 in queue>, next: Patient&<u32 in queue>
    }

reusable var queue: Patient[>..] = [];
let head = queue.alloc_index(Patient { name: "",
    prev: self, next: self });

fn arrive(name: u8[:]) -> Patient& {
    let tail .= queue[head].prev;
    let p .= queue.alloc_ref(Patient { name:
        name, prev: tail, next: queue[head] });
    tail.next .= p;
    queue[head].prev .= p;
    return p;
}

fn leave(p: Patient&) {
    p.prev.next .= p.next;
    p.next.prev .= p.prev;
    queue.free(queue.index_of(p));
}

fn show() {
    var out: u8[>..] = [];
    var cur .= queue[head].next;
    while cur .!= queue[head] { format(out,
        cur.name, " "); cur .= cur.next; }
    print(out, "(", queue.len, " slots)");
}

fn main() {
    arrive("Ada");
    let bea .= arrive("Bea");
    arrive("Cleo");
    let dot .= arrive("Dot");
    show();
    leave(bea);
    leave(dot);
    show();
    arrive("Eve");
    arrive("Fern");
    show();
}
```

```
Ada Bea Cleo Dot (5 slots)
Ada Cleo (5 slots)
Ada Cleo Eve Fern (5 slots)
```



Bea and Dot have left. Their slots are on the free list, and the next two arrivals will take them.

When Bea and Dot leave, their slots go on the free list, and Eve and Fern take them over. The pool never grows past five slots, however long the program runs, as long as at most four geese are waiting at a time.

Look at the `.=` in `let bea .= arrive("Bea")`. A `Patient` is fixed-size, so a plain `let bea = arrive("Bea")` would *copy* the patient out of the pool, and `leave(bea)` would then be asked to unlink a copy. The compiler rejects that, because `index_of` needs a reference it can prove points into `queue`. With `.=`, `bea` is the patient in the pool.

Local pools and pool parameters 14.4

A pool does not have to be global. A local `reusable var` works the same way, and lives as long as its scope. A parameter of type `T[>..]&` bound to a pool carries the free list with it, so a

function can allocate from and free into a pool that it is handed:

```
struct Egg { day: i32, grams: f32 }

fn hatch(tray: Egg[>..]&, slot: i64) {
    tray.free(slot); }

fn main() {
    reusable var tray: Egg[>..] = [];
    let a = tray.alloc_index(Egg { 1, 60.0 });
    tray.alloc_index(Egg { 2, 61.0 });
    hatch(tray, a);
    print(tray.alloc_index(Egg { 3, 59.0 }), "
        ", tray.len);
}
```

```
0 2
```

Slice pools 14.5

A plain pool hands out single slots. A `reusable[]` pool hands out **runs** of consecutive elements instead, which suits things like the child lists of nodes in a tree, where each node's children should be one contiguous slice:

Operation	Does
`pool.alloc_slice(n)`	a run of `n` default values, from a free run or from the end
`pool.free_slice(s)`	records the run `s` as free, merging it with free neighbours
`pool.realloc_slice(s, n)`	`s` resized to `n`, in place if possible, otherwise moved

```
fn main() {
    reusable[] var pens: i64[>..] = [];
    let a = pens.alloc_slice(3);
    a[0] = 1; a[1] = 2; a[2] = 3;
    let b = pens.alloc_slice(2);
    b[0] = 7; b[1] = 8;
    pens.free_slice(a);
    let c = pens.alloc_slice(2);            //
        reuses the start of a's run
    let d = pens.realloc_slice(b, 4);       // b
        ends the array, so it grows in place
    print(c, " ", d, " ", pens);
}
```

```
[0, 0] [7, 8, 0, 0] [0, 0, 3, 7, 8, 0, 0]
```

The `3` left in the middle of the array belongs to the free remainder of `a`'s old run. Like a single-slot pool, a slice pool never frees memory and never shrinks, so every element is always a valid value.

When to use a pool 14.6

Reach for a pool when data is created and destroyed in an order that does not follow the structure of your code: caches, queues of things that leave in any order, graphs that are edited, game entities that are spawned and removed. If the data's lifetime matches a scope, even a long one, you do not need a pool. A plain grow-only array owned by that scope is simpler and faster.

Exercises 14.7

1. Extend the waiting list with a function `jump_queue(p: Patient&)` that moves a patient

to the front of the list. You only need to unlink it and link it back in after `head`; the slot does not change.

2. Write a program that allocates 1000 eggs from a pool, freeing each one straight after it is allocated. How long is the pool at the end? What if you free only every other egg?

3. Explain why it is safe to keep a reference to a freed slot, but not to keep a reference into a grow-shrink array across a `pop`.

Chapter 15

Errors

Goose has no exceptions, no `Result` type and no required error convention. It gives you three small tools and lets you pick whichever suits the failure in front of you. This chapter covers all three, from the one you have already been using to the one that is unique to Goose, and then the ways of stopping a program outright.

Two outcomes: a trailing `bool` 15.1

For a function that either works or does not, return the result and a `bool` that says which. You have seen this with `parse_int`, and it is the standard idiom throughout the library:

```
import std;

fn parse_weight(text: u8[:]) -> f32, bool {
    let w, ok = parse_flt(trim(text));
    if !ok || w <= 0.0 { return 0.0, false; }
    return w as f32, true;
}

fn main() {
    for t in ["4.5", " 5.25 ", "heavy", "-1"] {
        let w, ok = parse_weight(t);
        print("'", t, "' -> ", w, " ", ok);
    }
}
```

```
'4.5' -> 4.5 true
' 5.25 ' -> 5.25 true
'heavy' -> 0.0 false
'-1' -> 0.0 false
```

Found or not: an optional 15.2

For "find me the thing, if it exists", return an optional. It is a reference into the data, so nothing is copied and the caller can change the thing it found. Chapter 11 covered this in full. The two idioms cover most of the failures a program meets: a value that cannot be produced, and a thing that cannot be found.

Failing from deep inside: `return … from` 15.3

Both idioms are local: a function reports to its immediate caller. That is fine for one level, but awkward when a failure is discovered many calls deep, in code that has no business knowing what to do about it. With trailing `bool`s, every function between the failure and the place that handles it must check, and pass the failure up, and declare that it might fail.

Goose's answer is the **long-distance return**:

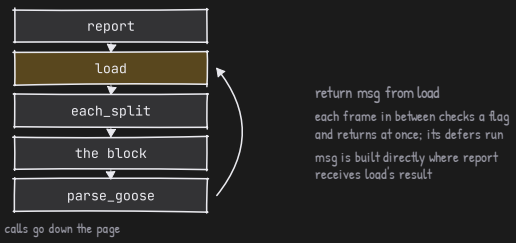
```
return value from f;
```

returns `value` as the result of the innermost call of `f` that is still running, leaving every function in between. Here a roster of geese is loaded line by line, and any malformed line ends the whole load with a message:

```
import std;

struct Goose { name: u8[..12], weight: f32 }

fn load(text: u8[:]) -> Goose[>..], u8[] {
    var flock: Goose[>..] = [];
    var line_no = 0;
    each_split(text, '\n') {
        line_no++;
        if trim(it).len > 0 {
            flock.push(parse_goose(it,
            line_no)); }
    };
    return flock, "";
}

fn parse_goose(line: u8[:], line_no: i64) ->
    Goose {
    let fields = split(line, ',');
    guard fields.len == 2 else {
        return [], str("line ", line_no, ":
            expected name,weight") from load;
    }
    let w, ok = parse_flt(trim(fields[1]));
    guard ok else {
        return [], str("line ", line_no, ": bad
            weight '", trim(fields[1]), "'")
            from load;
    }
    return Goose { name: trim(fields[0]),
        weight: w as f32 };
}

fn report(text: u8[:]) {
    let flock, err = load(text);
    if err.len > 0 { print("error: ", err);
        return; }
    print("loaded ", flock.len, ": ", flock);
}

fn main() {
    report("Ada, 4.5\nBea, 5.25\n");
    report("Ada, 4.5\nBea\n");
    report("Ada, heavy\n");
}
```

```
loaded 2: [Goose { "Ada", 4.5 }, Goose { "Bea",
    5.25 }]
error: line 2: expected name,weight
error: line 1: bad weight 'heavy'
```

Look at what `parse_goose` does *not* have. Its result type is a plain `Goose`. It has no error parameter, returns no `bool`, and has nothing like `?` or `try`. The trailing block given to `each_split` does not mention the error either. All the error handling in the program is in two places: the `guard`s that notice the problem, and the one `if` in `report` that deals with it.



return … from leaves parse_goose, the block and each_split, and delivers its value as the result of load.

How it works, and what it costs 15.3.1

This is not an exception mechanism with unwind tables. Each frame between the `return … from` and the target gains a hidden flag, which it checks after the call. On the way out, every frame returns normally, so every data stack is restored and every `defer` runs:

Newer than v2026.3

> This example uses a feature added to Goose after the v2026.3 release, so it is not checked against the compiler the rest of the book is.

```
var depth = 0;

fn inner() -> i64 { return 99 from outer; }

fn middle() -> i64 {
    depth++;
    defer depth--;              // runs even
        though inner never returns normally
    let v = inner();
    print("never printed");
    v
}

fn outer() -> i64 { middle() + 1 }

fn main() { print(outer(), " ", depth); }
```

```
99 0
```

`outer` returned 99, not 100: the `+ 1` never ran, because `middle` never returned normally. `depth` is back to 0, because `middle`'s `defer` ran on the way out. On the success path, the only cost is one flag check per frame, which is why idiomatic Goose parsers use `return … from` for every syntax error rather than saving it for emergencies.

The value is built **directly where the target's caller wants it**. The `str(…)` message above is written straight into the place where `report` receives `load`'s second result, without passing through the frames being left.

It is checked at compile time 15.3.2

A `return … from load` is only legal if every possible path to it runs inside a call of `load`. Goose compiles the whole program at once, so it can check this, and a long-distance return can never go uncaught at run time:

```
fn helper() -> i64 { return 1 from outer; }
fn outer() -> i64 { 5 }

fn main() { print(helper()); }
```

```
error: return from outer: no enclosing call of
    outer on this compile-time call path
```

From Rust

Where Rust threads `Result<T, E>` and `?` through every function between the error and its handler, Goose skips them all. The intermediate functions keep their plain signatures, and the compiler, not the type system, checks that the error has somewhere to land.

From Odin

`or_return` makes it cheap to pass an error up one level, but every function on the way still declares the error in its results. `return … from` removes the error from the intermediate functions entirely.

The `exception` module 15.4

Naming the target function in every `return … from` is precise, but it ties the failing code to one caller. The standard library's `exception` module wraps the same mechanism in a more familiar shape. `try<E>` runs a function and catches

anything thrown inside it with `throw`, as a value of type `E`:

```
import exception;

fn deep(n: i64) -> i64 {
    if n > 3 { exception::throw(str("too deep at
        ", n)); }
    return n * 10;
}

fn work() {
    print(deep(2));
    print(deep(5));
    print("not reached");
}

fn main() {
    exception::try<u8[]>(work) { err =>
        print("caught: ", err); };
    print("after");
}
```

```
20
caught: too deep at 5
after
```

The body to run is passed by name, here `work`, and the handler is the trailing block. The error type is given explicitly. `u8[]` is owned text, which is what you want for messages, but it can be any type: an integer code, or an enum of your own errors. As with `return … from`, the compiler checks that every `throw` has a `try` around it on every path.

Stopping the program 15.5

Some failures should not be handled at all. Three built-ins end the program:

- `assert(c)` stops the program if `c` is false. It is for invariants: conditions that would mean the program itself is wrong. It also narrows an optional, as you saw in Chapter 11.
- `abort(message)` stops the program with a message.
- `exit(code)` ends the program normally, with an exit code.

```
fn main() {
    let geese = 3;
    assert(geese > 5);
}
```

```
goose runtime error: assert failed
```

```
fn main() {
    abort("the goose escaped");
}
```

```
goose runtime error: the goose escaped
```

None of these can be caught, and `defer`s do not run when they happen. The compiler knows that `abort` and `exit` never return, so a branch that ends in one needs no value, as you saw in Chapter 5.

Choosing 15.6

Situation	Use
a computation that may not produce a value	a trailing `bool`
a lookup that may find nothing	an optional
a failure deep inside a parser, loader or interpreter	`return … from`, or `exception`
a bug, or a condition the program cannot continue from	`assert` or `abort`

Exercises 15.7

1. Write `fn safe_divide(a: i64, b: i64) -> i64, bool` and use it to divide 12 eggs between 0 and then 4 geese.
2. Add a check to `parse_goose` that rejects a weight above 15 kg with the message `line N: impossibly heavy`. How many other functions did you have to change?
3. Rewrite the roster loader to use the `exception` module instead of `return … from load`. What is the error type?

PART IV

Abstraction and the Library

This part covers the tools for writing code once and using it many times: generic functions, functions passed as arguments and blocks, recursion, and the standard library that is built from them. It ends with how a program is split across files and how it uses libraries written by other people.

Chapter 16

Generics and Blocks

Goose's generics are simple to write: leave out a parameter's type and the function is generic. Its function values are simple too, because they exist only at compile time. Together they let the standard library offer `map`, `filter` and `sort` that compile to plain loops, and let you write your own iteration helpers that behave exactly like built-in loops.

Generic functions 16.1

A parameter with no type is generic. The function is compiled separately for each combination of argument types it is called with:

```
fn twice(x) { x + x }

fn main() {
    print(twice(21), " ", twice(1.5));
}
```

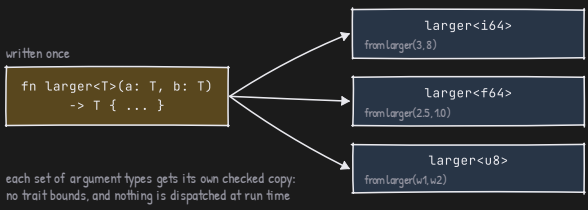
```
42 3.0
```

When two parameters must have the same type, or when a type is needed that no parameter carries, name it with angle brackets:

```
fn larger<T>(a: T, b: T) -> T { if a > b { a }
    else { b } }

fn zero<T>() -> T { default<T>() }

fn main() {
    let w1: u8 = 40;
    let w2: u8 = 52;
    print(larger(3, 8), " ", larger(2.5, 1.0), "
        ", larger(w1, w2), " ", zero<f64>() +
        0.5);
}
```

```
8 2.5 52 0.5
```

Type arguments are usually inferred from the arguments. When nothing determines one, as with `zero`, you give it explicitly: `zero<f64>()`.



A generic function is compiled once for each set of argument types it is used with.

There are **no trait bounds**. Goose never type-checks a generic body in the abstract. It checks each specialization with the real types, so a generic function can do anything its arguments allow, and an error shows up where the types do not fit, together with the chain of calls that led there:

```
fn twice(x) { x + x }

fn main() { print(twice("goose")); }
```

```
error: operator + cannot be applied to const
    u8[:] and const u8[:]
in fn twice(const u8[:]) instantiated from
    main.goose:3
```

From Rust

This is closer to C++ templates than to Rust generics: duck-typed at the point of use, with no `where T: Add<Output = T>`. You trade early checking of the generic body for not having to write bounds. The call chain in the error is what keeps the messages useful.

From Odin

`fn twice(x)` is Odin's `twice :: proc(x: $T) -> T`, and `<T>` plays the role of a `$T` that appears in several places. Like Odin, Goose specializes everything at compile time. There are no parameters for compile-time sizes, so a generic function that needs a fixed array of any length takes a slice instead.

Function values 16.2

A function can be passed to another function. The parameter that receives it is a type parameter, written in the angle brackets like any other, and it is called like a function:

```
fn square(x: i64) -> i64 { x * x }
fn apply<F>(x: i64) -> i64 { F(x) }

fn main() {
    print(apply(7, square));
}
```

```
49
```

The function argument comes after the ordinary ones. You can document its shape with `F: fn(i64) -> i64`, but that is optional.

Function values in Goose are **compile-time entities**. Passing `square` to `apply` creates a specialization of `apply` with `square` built in, and the call `F(x)` becomes a direct call that can be inlined. A function value can be passed down, but never stored, returned or put into data, so there are no function pointers and no closures that outlive their scope:

```
fn square(x: i64) -> i64 { x * x }

fn main() {
    var f = square;
}
```

```
error: function values are compile-time only
    and cannot be stored (§7.6)
```

That restriction is the reason a higher-order function in Goose costs exactly what the loop it contains costs.

Trailing blocks 16.3

Passing a named function works, but the common case is a small piece of code written right at the call. A **trailing block** is a `{ … }` after a call's closing parenthesis, and it becomes the call's last argument. Inside it, `it` is the block's

single parameter. For several parameters, name them before a `=>`:

```
fn repeat<F>(n: i64) { for i in n { F(i); } }

fn each_pair<T, F>(xs: T[:]) {
    for i in 0..xs.len - 1 { F(xs[i], xs[i +
        1]); }
}

fn main() {
    repeat(3) { print("honk ", it); };
    let weights: i64[5] = [40, 45, 52, 48, 61];
    var gains: i64[>..] = [];
    each_pair(weights) { a, b => gains.push(b -
        a); };
    print(gains);
}
```

```
honk 0
honk 1
honk 2
[5, 7, -4, 13]
```

A block can read and write the variables of the function it is written in, as `gains.push(…)` does. Because the block is inlined into the specialization, that costs nothing.

Note the `;` after the block. A call with a trailing block is an expression statement, so it ends with a semicolon like any other call.

`return` inside a block 16.3.1

A `return` inside a block returns from the **function the block is written in**, not from the function that called the block. So a hand-written iteration helper is just as good as a built-in loop. Inside a loop body you can `return` from the

enclosing function, and the same goes for the body of `repeat`:

```
fn repeat<F>(n: i64) { for i in n { F(i); } }

fn first_heavy(weights: i64[:]) -> i64 {
    repeat(weights.len) {
        if weights[it] > 50 { return
            weights[it]; }    // leaves
            first_heavy
    };
    return -1;
}

fn main() {
    let weights: i64[5] = [40, 45, 52, 48, 61];
    print(first_heavy(weights), " ",
        first_heavy(weights[..2]));
}
```

```
52 -1
```

Under the hood this uses the same mechanism as `return … from` in Chapter 15.

16.4 The library's higher-order functions

With these pieces, the standard library provides the usual toolkit. All of these take a slice and a block, and all of them compile to a single loop. Results come back as new grow-only arrays, built directly in the variable that receives them:

```
import std;

fn main() {
    let weights: i64[5] = [40, 45, 52, 48, 61];
    let heavy = weights.filter() { it > 46 };
    let kilos = weights.map() { it as f64 / 10.0
        };
    let total = fold(weights, 0) { acc, w => acc
        + w };
    print(heavy, " ", kilos, " ", total);
    print(weights.any() { it > 60 }, " ",
        weights.all() { it > 60 }, " ",
          weights.count() { it % 2 == 0 }, " ",
              sum(weights));
    let big = weights.find() { it > 50 };
    if big { print("first over 50: ", big); }
}
```

```
[52, 48, 61] [4.0, 4.5, 5.2, 4.8, 6.1] 246
true false 3 246
first over 50: 52
```

`find` returns an optional reference into the slice, like the functions you wrote in Chapter 11. `map`'s result type is whatever the block returns.

`sort` sorts in place. Without a block it uses `<`; with one, the block is the comparison, and returns true when its first argument should come first:

```
import std;

struct Goose { name: u8[..8], weight: i64 }

fn main() {
    var flock: Goose[>..] = [Goose { "Ada", 45
        }, Goose { "Bea", 52 }, Goose { "Cleo",
        38 }];
    sort(flock) { a, b => a.weight < b.weight };
    print(flock);
    var ids: i64[5] = [5, 3, 9, 1, 7];
    sort(ids);
    print(ids);
}
```

```
[Goose { "Cleo", 38 }, Goose { "Ada", 45 },
    Goose { "Bea", 52 }]
[1, 3, 5, 7, 9]
```

Chapter 18 covers the rest of `std`.

Nested functions as values 16.5

A nested function can be passed as a function value too, which gives you a way to pass a "block" that is too long to write inline, or that you want to use twice:

```
import std;

fn main() {
    var gaggle = 0;
    fn count_heavy(w: i64) -> bool {
        gaggle++;
        w > 46
    }
    let weights: i64[5] = [40, 45, 52, 48, 61];
    let heavy = weights.filter(count_heavy);
    print(heavy, " after ", gaggle, " checks");
}
```

```
[52, 48, 61] after 5 checks
```

Exercises 16.6

1. Write a generic `fn clamp_all<T>(xs: T[:], lo: T, hi: T)` that clamps every element of a slice in place, and use it on both an `i64` array and an `f32` array.

2. Write `fn each_window<T, F>(xs: T[:], n: i64)` that calls its block with every run of `n` consecutive elements, as a slice. Use it to print the sum of each run of three in `[1, 2, 3, 4, 5]`.

3. Use `filter` and `map` to turn `[61, 72, 58, 66, 70]`, a list of egg weights in grams, into the list of eggs above 60 grams, in kilograms.

Chapter 17

Recursion

Recursion in Goose is opt-in. A function that is part of a recursive cycle must be declared `recursive fn`, as Chapter 5 showed, and recursive code lives under one extra rule. This chapter explains that rule, why the memory model needs it, and the three patterns that make recursive code easy to write within it.

The rule 17.1

> No function in a recursive cycle may call back into the cycle while one of its non-fixed locals is in scope.

In other words, a recursive function can own scalars, structs, fixed arrays, references and slices across its recursive calls. It cannot keep a growable array, a string builder, or any other variable-size or resizable value alive across one.

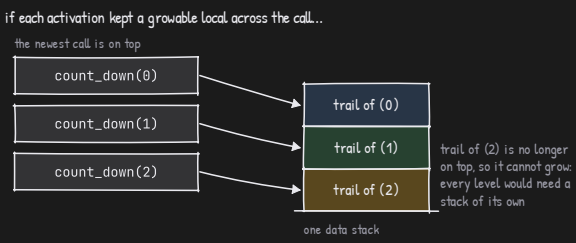
Here is the rule being broken:

```
recursive fn count_down(n: i64) {
    var trail: u8[>..] = [];
    format(trail, "at ", n);
    if n > 0 { count_down(n - 1); }
    print(trail);
}

fn main() { count_down(2); }
```

```
error: count_down calls into its recursive
    cycle while non-fixed-size local trail
    (declared at main.goose:2) is in scope
    (§7.8): end its scope before the call
```

Why 17.1.1

Remember the rule from Chapter 6: a growable value is always the topmost thing on its data stack, so that it can grow without moving. Each call of `count_down` creates a new `trail`. The compiler works out the number of data stacks at compile time, so the program has a fixed number of them. If every level of the recursion kept its own `trail` alive, each new `trail` would land on top of the previous one, and the previous one could no longer grow:



If every activation kept its own growable local across the recursive call, each would need a data stack of its own, and recursion depth is not known at compile time.

With the rule in place, each level of recursion costs only native stack space, which is the same as in C. A recursion that runs too deep stops the program with a clean error rather than corrupting anything:

```
recursive fn depth(n: i64) -> i64 { if n == 0 {
    0 } else { 1 + depth(n - 1) } }

fn main() { print(depth(100000000)); }
```

```
goose runtime error: native call stack overflow
```

The compiler's message already suggests the first way to live with the rule.

17.2 Pattern 1: finish with the scratch before recursing

A recursive function may own growable data, as long as its scope ends before the recursive call. Wrapping the scratch work in a block is often all it takes:

```
recursive fn count_down(n: i64) {
    {
        var trail: u8[>..] = [];
        format(trail, "at ", n);
        print(trail);
    }                                   // trail
        is gone before the call
    if n > 0 { count_down(n - 1); }
}

fn main() { count_down(2); }
```

```
at 2
at 1
at 0
```

The same rule applies to temporaries in the calling statement. You cannot pass `str(…)` straight into a recursive call, because the new string would be alive during the call:

```
recursive fn echo(s: u8[:], n: i64) {
    print(s);
    if n > 0 { echo(str("level ", n), n - 1); }
}

fn main() { echo("start", 2); }
```

```
error: echo calls into its recursive cycle with
    the non-fixed-size temporary str("level ",
    n) as an argument (§7.8): pass it in from
    outside the cycle
```

Pattern 2: pass the output in 17.3

Most recursive functions that build something can build it into an array that belongs to their caller. The array is owned outside the cycle, so it is not affected by the rule, and every level of the recursion pushes into the same array:

```
recursive fn hanoi(n: i64, from: u8, to: u8,
    via: u8, moves: u8[>..]&) {
    if n == 0 { return; }
    hanoi(n - 1, from, via, to, moves);
    moves.push(from);
    moves.push('>');
    moves.push(to);
    moves.push(' ');
    hanoi(n - 1, via, to, from, moves);
}

fn main() {
    var moves: u8[>..] = [];
    hanoi(3, 'A', 'C', 'B', moves);
    print(moves);
}
```

```
A>C A>B C>B A>C B>A B>C A>C
```

You already saw this shape in Chapter 13, where `in_order` walked a tree and wrote names into a

builder it was given.

Pattern 3: a context of references 17.4

A larger recursive algorithm, such as a parser or a compiler pass, often works on several tables at once: the input, a list of nodes, an output buffer. Make each table a local of a non-recursive driver function, and hand the recursion a single struct of references to them. A struct of references is fixed-size, so the recursion may hold it across its calls:

```
struct Place { name: u8[..8], inside: i64[..4] }
struct Ctx { places: Place[>..]&, out: u8[>..]&
    }

recursive fn draw(c: Ctx&, i: i64, depth: i64) {
    for d in depth { c.out.append("  "); }
    format(c.out, c.places[i].name, "\n");
    for k in c.places[i].inside { draw(c, k,
        depth + 1); }
}

fn main() {
    var places: Place[>..] = [];
    for n in ["farm", "pond", "barn", "Ada",
        "Bea"] { places.push(Place { name: n,
        inside: [] }); }
    places[0].inside.push(1);
    places[0].inside.push(2);
    places[1].inside.push(3);
    places[2].inside.push(4);
    var out: u8[>..] = [];
    var c = Ctx { places: places, out: out };
    draw(c, 0, 0);
    print(out);
}
```

```
farm
  pond
    Ada
  barn
    Bea
```

There is a variant of this pattern that needs no struct at all: declare the recursive functions as *nested* functions inside the driver, where they can reach the driver's tables as free variables. The calculator in Chapter 25 is written that way.

Sharing a scratch stack 17.4.1

Sometimes each level of a recursion needs scratch space of its own that lives across its recursive calls. Give the whole recursion one grow-shrink array, owned outside the cycle. Each activation records where the stack top was, pushes its own entries, and truncates back before it returns:

```
recursive fn depth_sum(scratch: i64[>..<]&,
    depth: i64) -> i64 {
    let mark = scratch.len;
    scratch.push(depth * 10);             //
        this level's own entries
    let mine = scratch[mark..];
    var total = mine[0];
    if depth < 3 { total += depth_sum(scratch,
        depth + 1); }
    scratch.resize(mark);                 //
        gone again before returning
    return total;
}

fn main() {
    var scratch: i64[>..<] = [];
    print(depth_sum(scratch, 0), " ",
        scratch.len);
}
```

```
60 0
```

Note that `mine`, a slice into `scratch`, is used *after* the recursive call. That is normally forbidden, because `scratch` can shrink. It is allowed here because the compiler sees that the recursive call is *balanced*: it leaves the array at least as long as it found it, so nothing below `mark` is ever freed during the call.

Mutual recursion 17.5

When several functions call each other, only the *entry* of the cycle, the function the outside world calls first, needs `recursive`. Every function in the cycle must have fully written-out parameter and result types, because the compiler cannot infer types around a loop:

```
recursive fn is_even(n: i64) -> bool { if n == 0
    { true } else { is_odd(n - 1) } }
fn is_odd(n: i64) -> bool { if n == 0 { false }
    else { is_even(n - 1) } }

fn main() { print(is_even(10), " ", is_odd(7));
    }
```

```
true true
```

```
recursive fn is_even(n: i64) -> bool { if n == 0
    { true } else { is_odd(n - 1) } }
fn is_odd(n) { if n == 0 { false } else {
    is_even(n - 1) } }

fn main() { print(is_even(10)); }
```

```
error: function is_odd is in a recursive cycle
    and needs fully explicit parameter types
```

From C

In practice, the rule rules out very little. The C idiom of a recursive function that `malloc`s a buffer, recurses, and frees it afterwards becomes "take the buffer as a parameter", which is usually the faster code anyway.

Design note: why recursion is marked 17.6

Most languages let any function call itself without ceremony, and so does Goose's type checker in principle. The keyword is there for the memory model. The compiler sizes the data stacks by walking the call graph, and a cycle in that graph would make the walk unbounded, unless the functions in the cycle are known not to hold any stack across their recursive calls. Marking the entry of each cycle makes the cycles explicit, both to the compiler and to the reader, and it is the place where the compiler checks the rule. It also tells the person reading the code that this is where the native stack can grow without bound.

Exercises 17.7

1. Write a recursive `fn binary(n: i64, out: u8[>..]&)` that appends the binary digits of a non-negative number, most significant first.

2. Write a recursive function that sums an `i64[:]` by splitting it in half, summing each half, and adding the results. Is the slice a problem for the rule? Why or why not?
3. Change `draw` so that it also prints how many places each place contains in total, including nested ones. Where does the count come from?

Chapter 18

A Tour of the Standard Library

Goose's standard library is small, written almost entirely in Goose, and built on the same principles as the language: no hidden allocations, results built directly where they are wanted, and slices instead of copies. This chapter is a tour rather than a reference. It shows the parts you will use most, with enough detail to start using them. The official `docs/stdlib.md` has every function.

Importing modules 18.1

A module is imported by name:

```
import std;
import dictionary;
```

`std`, `dictionary`, `vec`, `math` and `os` put their names in the global namespace, so after `import std;` you call `trim(s)` or `s.trim()`. The other modules have namespaces of their own, so their functions are written with a prefix: `csv::each_record`, `json::parse`, `regex::compile`.

An import only loads the file. It does not bring in anything the file does not declare, and importing the same module twice is harmless.

`std`: arrays 18.2

Most of `std` is generic functions over slices, and most of those take a block. You met several in Chapter 16. Here is a wider selection, grouped by what they do to their input.

Reading and searching leave the array alone: `find` (a block, or a sub-slice), `position`, `contains`, `any`, `all`, `count`, `min`, `max`, `binary_search`, `starts_with`, `ends_with`.

Rearranging in place keeps the length: `sort`, `stable_sort`, `reverse`, `shuffle`, `fill`.

Changing the length takes the whole array by reference, and works on any array kind that has the needed operations: `insert_at`, `remove_at`, `swap_remove`, `retain`, `dedup`, `push_n`.

```
import std;

fn main() {
    var eggs: i64[>..] = [58, 58, 61, 66, 66,
        66, 72];
    dedup(eggs);                                //
        drops adjacent duplicates
    print(eggs);
    eggs.retain() { it > 60 };                  //
        keeps what the block accepts
    eggs.insert_at(0, 59);
    let first = eggs.remove_at(1);
    print(eggs, " removed ", first);
    let at, found = binary_search(eggs, 66);
    reverse(eggs);
    print(at, " ", found, " ", eggs, " ",
        contains(eggs, 99));
}
```

```
[58, 61, 66, 72]
[59, 66, 72] removed 61
1 true [72, 66, 59] false
```

`std` also has a binary heap that works on any array you can push to and pop from. A grow-shrink array is the natural container for it:

```
import std;

fn main() {
    var queue: i64[>..<] = [];
    for urgency in [5, 1, 4, 2, 3] {
        queue.heap_push(urgency); }
    var order: i64[>..] = [];
    while queue.len > 0 {
        order.push(queue.heap_pop()); }
    print(order);
}
```

```
[1, 2, 3, 4, 5]
```

`heap_push` and `heap_pop` keep a min-heap ordered by `<`. Like `sort`, they take an optional block that decides which of two elements comes first.

`std`: numbers, hashing and randomness 18.3

`min`, `max`, `clamp`, `abs`, `sign` and `lerp` are generic over the numeric types. `rem` and `div_euclid` you met in Chapter 3. `popcount`, `clz` and `ctz` count bits.

`hash` hashes any integer, float or byte string to a `u64`, and `hash_combine` mixes two hashes. You need these when you use a struct of your own as a dictionary key.

Random numbers come from `rng`, a small splitmix64 generator whose state is its seed. The same seed always gives the same sequence, which makes tests reproducible:

```
import std;

fn main() {
    var r = rng { 42 };
    var rolls: i64[>..] = [];
    for i in 5 { rolls.push(r.rand_int(6) + 1);
        }
    var deck: i64[6] = [1, 2, 3, 4, 5, 6];
    shuffle(deck, r);
    print(rolls, " ", deck);
}
```

```
[2, 2, 1, 1, 5] [3, 4, 2, 5, 6, 1]
```

For a different sequence each run, `import os;` and seed it with `random_seed()`, which asks the operating system for entropy.

`dictionary` 18.4

`dictionary<K, V>` is a hash table with fixed-size keys and values. It uses open addressing, so it is a single grow-shrink array of slots with no per-entry allocation:

```
import dictionary;

fn main() {
    var weights = dictionary<const u8[:], f32>
        {};
    weights.insert("Ada", 4.5);
    weights.insert("Bea", 5.25);
    weights.insert("Ada", 4.75);           //
        overwrites
    let w = weights.get("Ada");
    if w { print("Ada weighs ", w); }
    print(weights.count, " ",
        weights.contains("Cleo"), " ",
        weights.get("Cleo") == null);
}
```

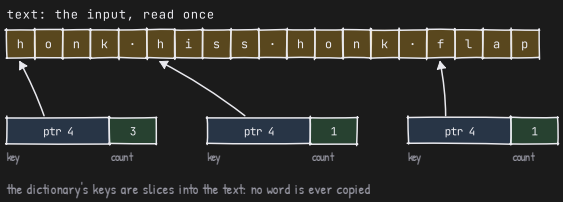
```
Ada weighs 4.75
2 false true
```

`get` returns an optional reference to the value, so you can update it through the result. `update(key, initial) { … }` does a lookup and a change in one call, inserting `initial` first if the key is missing. It is the right tool for counting:

```
import std;
import dictionary;

fn main() {
    let text = "honk hiss honk flap honk hiss";
    var counts = dictionary<const u8[:], i64>
        {};
    each_word(text) { counts.update(it, 0) { it
        += 1; }; };
    var keys: (const u8[:])[>..] = [];
    counts.each() { k, v => keys.push(k); };
    sort(keys) { a, b => compare(a, b) < 0 };
    for k in keys {
        let n = counts.get(k);
        if n { print(k, " x", n); }
    }
}
```

```
flap x1
hiss x2
honk x3
```

The keys of `counts` are *slices into* `text`. Every word is a view of the original string, so no word is copied:



A dictionary keyed by slices stores views into text the program already holds.

That is also why the key type is `const u8[:]`: the keys may be views of read-only text, such as a literal. A dictionary that needs to own its keys uses a fixed-size key such as `u8[..16]`.

The dictionary's slots live in a grow-shrink array, so the rule from Chapter 7 applies. A reference returned by `get` must not be used after the next `insert`, `update` or `remove`, and the compiler tells you if it is. `each` visits entries in an unspecified order, which is why the example sorts the keys before printing.

> **From Rust**
>
> `dictionary<K, V>` is a `HashMap<K, V>`. A `HashMap<&str, usize>` keyed by borrows of the input needs a lifetime on the map in Rust. Here the compiler works that out from where the keys came from.

`math` and `vec` 18.5

`math` is libm at both float widths: `sqrt`, `sin`, `cos`, `atan2`, `exp`, `log`, `pow`, `floor`, `ceil`, `round` and the rest, plus `PI` and `TAU`. The function is picked by the argument's type, so an integer argument has to choose: write `sqrt(n as f64)`.

`vec` provides small vector structs, `vec2<T>`, `vec3<T>` and `vec4<T>`, with the aliases `float3`, `double2`, `int3` and so on. They use the memberwise arithmetic from Chapter 10, and add `dot`, `cross`, `length`, `normalize`, `distance` and `lerp`:

```
import math;
import vec;

fn main() {
    let v = float3 { 1.0, 2.0, 2.0 };
    print(sqrt(2.0), " ", floor(2.7), " ",
        pow(2.0, 10.0));
    print(length(v), " ", dot(v, v), " ", v *
        2.0);
    print(cross(float3 { 1.0, 0.0, 0.0 }, float3
        { 0.0, 1.0, 0.0 }));
}
```

```
1.4142135623730951 2.0 1024.0
3.0 9.0 vec3<f32> { 2.0, 4.0, 4.0 }
vec3<f32> { 0.0, 0.0, 1.0 }
```

`os`: files, arguments and time 18.6

`os` is a thin layer over the operating system. File functions take a path and return a `bool` for success. Reading appends to a builder you

supply, which fits the memory model: the caller decides where the bytes live.

```
import std;
import os;

fn main() {
    print(write_file("eggs.txt",
        "61\n72\n58\n"));
    append_file("eggs.txt", "66\n");
    var text: u8[>..] = [];
    if read_file("eggs.txt", text) {
        print(text.len, " bytes: ",
            words(text));
    }
    delete_file("eggs.txt");
    print(file_exists("eggs.txt"));
}
```

```
true
12 bytes: ["61", "72", "58", "66"]
false
```

There are also `list_dir`, `make_dir`, `rename_file` and `write_file_atomic`, which writes a new file and renames it into place, so a crash never leaves a half-written file. For the console there are `read_line`, `read_stdin`, `write_stdout` and `write_stderr`. `args()` returns the program's arguments, as you saw in Chapter 2. `clock()` returns a monotonic time in seconds for measuring durations, and `sleep_ms(ms)` waits. Subprocesses and networking are not part of the library yet.

`csv` 18.7

`csv::each_record` reads comma-separated text as RFC 4180 defines it, including quoted fields, and

calls its block once per record. The fields are slices of the input:

```
import std;
import csv;

fn main() {
    let roster =
        "name,weight,laying\nAda,4.5,true\n\"Bea
        , the bold\",5.25,false\n";
    var total = 0.0;
    var row = 0;
    let ok = csv::each_record(roster) { fields
        =>
        row++;
        if row > 1 {
            let w, wok = parse_flt(fields[1]);
            if wok { total += w; }
            print(fields[0], " | ", fields[2]);
        }
    };
    print(ok, ", total weight ", total);
}
```

```
Ada | true
Bea, the bold | false
true, total weight 9.75
```

`csv::format_field` writes a field back out, quoting it if needed.

`json` 18.8

`json` offers three ways to work with JSON. The writer appends to a builder, and you write the brackets and commas yourself:

```
import json;

fn main() {
    var out: u8[>..] = [];
    out.push('{');
    json::format_key(out, "name");
    json::format_string(out, "Ada \"the
        first\"");
    out.push(',');
    json::format_key(out, "weight");
    json::format_number(out, 4.5);
    out.push('}');
    print(out);
}
```

```
{"name":"Ada \"the first\"","weight":4.5}
```

The parser builds a document tree in an array you provide, with every node linked to the next by the self-relative links of Chapter 13, and strings left as slices of the input:

```
import json;

fn main() {
    let text = """{"flock": [{"name": "Ada",
        "weight": 4.5}, {"name": "Bea",
        "weight": 5.25}], "pond": "north"}""";
    var doc: json::Node[>..] = [];
    var decoded: u8[>..] = [];
    let root, err = json::parse(text, doc,
        decoded);
    guard root else { abort(err); }
    let pond, ok =
        json::as_string(json::member(root,
        "pond"));
    print("pond: ", pond);
    let flock = json::member(root, "flock");
    if flock {
        json::each(flock) { g =>
            let name, n_ok =
                json::as_string(json::member(g,
                "name"));
            let w, w_ok =
                json::as_f64(json::member(g,
                "weight"));
            print(name, " ", w);
        };
    }
}
```

```
pond: north
Ada 4.5
Bea 5.25
```

The third way reads the text straight into your own types without building a tree, in the style of a serde derive. `json::reader` keeps a cursor, and `each_member` and `each_element` walk objects and arrays:

```
import json;

struct Goose { name: u8[..12], weight: f64 }

fn main() {
    var r = json::reader("""[{"name": "Cleo",
        "weight": 3.8, "colour": "grey"}]""");
    var geese: Goose[>..] = [];
    json::each_element(r) {
        var g = Goose { name: "", weight: 0.0 };
        var scratch: u8[>..] = [];
        json::each_member(r) { key =>
            if key == "name" { g.name =
                json::read_string(r, scratch); }
            else if key == "weight" { g.weight =
                json::read_number(r); }
            else { json::skip(r); }
        };
        geese.push(g);
    };
    print(json::finish(r), " ", geese);
}
```

```
true [Goose { "Cleo", 3.8 }]
```

Errors are sticky: after the first unexpected character every read fails quietly, and `finish` reports whether the whole text was read cleanly.

`regex` 18.9

`regex` implements regular expressions with the syntax and matching rules of Rust's `regex` crate, compiled to DFAs when you call `regex::compile`. `each_captures` calls a block for every match, with access to the groups:

```
import std;
import regex;

fn main() {
    let re = regex::compile("(\\w+) laid
        (\\d+)");
    let log = "Ada laid 3, Bea laid 1, Cleo
        slept, Dot laid 2";
    var total = 0;
    regex::each_captures(re, log) { c =>
        let n, ok = parse_int(regex::group(c,
            2));
        if ok { total += n;
            print(regex::group(c, 1), ": ", n);
            }
    };
    print(total, " eggs from ", regex::count(re,
        log), " geese");
}
```

```
Ada: 3
Bea: 1
Dot: 2
6 eggs from 3 geese
```

Anchors and word boundaries are not supported, and `compile` reports them as errors.

The rest 18.10

A few more modules round out the library. `base64` encodes and decodes Base64 text. `binary` gives checked little-endian reads of binary file formats. `ffi` copies strings and buffers that a C library owns into Goose storage. `exception` you met in Chapter 15. Graphics, audio, user interfaces and physics live in separate packages built on SDL3 and Box3D. They are imported the same way, but fetched as packages, which is the subject of the next chapter.

Exercises 18.11

1. Use a `dictionary<i64, i64>` to count how many eggs of each weight appear in `[61, 72, 61, 58, 72, 61]`, and print the most common weight.
2. Write a program that writes three lines of CSV (`name,eggs`) to a file, reads the file back with `read_file`, and prints the total number of eggs using `csv::each_record`.
3. Use `stable_sort` with a block to sort geese by weight, keeping geese of equal weight in their original order. How can you tell it kept the order?

Chapter 19

Programs, Namespaces and Packages

Every example so far has been a single file. This chapter covers what happens when a program grows: splitting it across files, keeping names apart with namespaces, and using libraries that live in other directories or other repositories.

Whole-program compilation 19.1

The first thing to understand is that Goose compiles a **whole program at once**. You name one file on the command line, the *root file*. The compiler follows its imports, reads everything, and produces one C file for the entire program. There is no separate compilation, no object file per module, and no linking of Goose code at all.

That decision underlies much of what you have seen. Function values can be compile-time entities because the compiler sees every call. Roots can be inferred because the compiler sees every caller. `return … from` can be checked because the compiler knows every path to it. The cost is that you cannot ship a precompiled Goose library, and a large program is recompiled in full. The compiler is fast enough that the second cost rarely matters in practice.

Splitting a program into files 19.2

`import a.b.c;` includes the file `a/b/c.goose`, once, however many files import it. The path is looked up relative to the root file's directory first, then among the packages the program declares, and finally in the standard library. A leading dot, `import .helpers;`, looks next to the importing file instead.

An import includes declarations and nothing else. It does not create a scope or rename anything. A file that wants its names kept apart from everyone else's declares a namespace.

Namespaces 19.3

`namespace barn;` at the top of a file puts all of the file's declarations into the namespace `barn`. From other files, they are reached as `barn::name`:

```
namespace barn;

struct Stall { number: i64, occupant: u8[..8] }

fn format(out: u8[>..]&, s: Stall) { format(out,
    "stall ", s.number, " (", s.occupant, ")");
    }

fn is_empty(s: Stall) -> bool { s.occupant.len
    == 0 }

var stalls_built = 0;

fn build(n: i64) -> Stall {
    stalls_built += 1;
    Stall { number: n, occupant: "" }
}
```

Newer than v2026.3

> This example uses a feature added to Goose after the v2026.3 release, so it is not checked against the compiler the rest of the book is.

```
import farm.barn;

fn is_empty(x: i64) -> bool { x == 0 }

fn main() {
    var s = barn::build(3);
    print(s, " ", s.is_empty(), " ",
        barn::is_empty(s), " ", is_empty(0));
    s.occupant = "Ada";
    print(s, " ", s.is_empty(), " ",
        barn::stalls_built, " built");
}
```

The first block is `farm/barn.goose`, the second the root file:

```
$ goose main.goose
```

```
stall 3 () true true true
stall 3 (Ada) false 1 built
```

Three things in that output are worth noticing.

- `barn::Stall` printed using `barn::format`, even though `main` lives in the global namespace. The `format` hook is looked up in the namespace where the *type* is declared.
- `s.is_empty()` found `barn::is_empty`, not the global `is_empty(i64)`. A method-style call looks in the namespace of its receiver's type first.
- The plain call `is_empty(0)` found the global function.

So a type's namespace travels with the type. Code that uses a `barn::Stall` can call its functions with a dot from anywhere, without importing names or writing the prefix.

Namespaces do not nest, have no privacy and cannot re-export anything. They only keep names apart. A declaration can also name its namespace directly, `fn barn::repair(…)`, and `::name` refers to the global declaration that a namespaced one would otherwise hide.

> **From Odin**
>
> A Goose namespace is lighter than an Odin package. Several files can share one, a file chooses its own, and nothing is private. The closest Odin equivalent is a package whose every declaration is public.

`main` and globals 19.4

The program's entry point is `fn main()` in the global namespace of the root file. A `main` in an imported file is ignored, so any file can contain a small `main` that runs its own demo or tests when you run it directly, and that `main` is skipped when another program imports the file. In a file with a `namespace` line, write it as `fn ::main()`, so that it is declared in the global namespace where the compiler looks for it. Inside it, the file's own declarations then need their prefix, as they do anywhere else outside the namespace.

Globals are initialized before `main` runs, imported files first, then in declaration order within each file. Their initializers can call

functions. The whole program runs inside an outermost scope that owns the globals, so for the lifetime rules a global outlives everything.

Packages 19.5

A *package* is a library in a directory of its own: a module named after the directory, plus any further modules beside it, and possibly some C. The root file declares each package it uses, saying where it comes from:

```
namespace mylib;

fn honks(n: i64) -> u8[>..] {
    var s: u8[>..] = [];
    for i in n { s.append("honk "); }
    return s;
}
```

```
import mylib from "./mylib";

fn main() { print(mylib::honks(3)); }
```

```
honk honk honk
```

A path source like `"./mylib"` is relative to the file that declares it, and is used in place, so you can edit the library and rerun the program. A git source names a repository and a ref after an `@`: a tag or commit, such as `"joe/mylib@v2"` for a GitHub repository at tag `v2`, or any git URL. The compiler fetches it once into its cache directory (`~/.goose`, or wherever `GOOSE_HOME` points) and builds any C it contains.

Only the root file declares packages. When a library file needs another package, it writes the same `import other from "…";` line, but the compiler reads it as a *requirement* that the root must meet. If the root has not declared that package, the error says exactly which line to add. So a program's dependencies are listed in one place, pinned, and flat: there is only ever one version of each package in a program.

`goose fetch program.goose` downloads everything ahead of time, and `--offline` makes sure nothing is fetched during a build.

Blessed packages 19.5.1

A few packages are known to the compiler by name: the graphics, audio, UI and physics libraries built on SDL3 and Box3D. You import them like standard modules, with `import gfx;`, and the compiler fetches and builds them the first time. A compiler built with `-DGOOSE_BUILTIN_LIBS=all` has them compiled in, and can run such programs with nothing else installed.

Embedding files 19.6

`embed_file("path")` reads a file *when the program is compiled*, and gives you its bytes as a `const u8[:]` in the program's read-only data. The path is relative to the source file that calls it. It is the easy way to ship a font, a table or a level with a

program. This program embeds its own source code:

```
import std;

let SOURCE = embed_file("main.goose");

fn main() {
    let lines = split(SOURCE, '\n');
    print(lines[0], " ... ", lines.len, "
        lines");
}
```

```
import std; ... 9 lines
```

A program with C of its own 19.7

C code that belongs to a program goes in a *sidecar file*, `foo.build` beside `foo.goose`, which lists C sources, include directories, defines and libraries. The build compiles them with the program. Chapter 22 shows how Goose calls into that C.

Exercises 19.8

1. Move the `Goose` struct and its `format` overload from Chapter 9 into a file `geese.goose` with `namespace geese;`, and use it from a root file. Which calls need the `geese::` prefix, and which do not?
2. Add a `fn main()` to your `geese.goose` that prints a test goose. Run the file directly, then run the root file. What happens to the library's `main`?

3. What would go wrong if two libraries in one program each declared a function `fn load(path: u8[:])` without a namespace? Try it.

PART V

Systems Work

The last four chapters of the language proper deal with the program's edges: running work on several threads, saving data to files and loading it back safely, calling C and being called from it, and finding and removing the costs that safety adds.

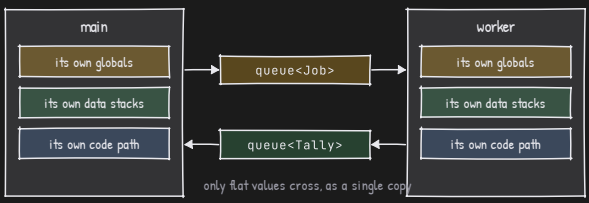
Chapter 20

Threads That Share Nothing

Goose has threads, and it has no data races. It gets there the blunt way: **threads share no mutable memory at all**. There are no locks, no atomics and no memory orderings in the language, because there is nothing for them to protect. Threads communicate by sending values through typed queues.

A worker is a separate program 20.1

A function declared `thread_fn` is the entry point of a worker. The compiler compiles it, and everything it calls, as a *separate program*, with its own data stacks and its own copies of the globals it uses:



Each worker is compiled as a separate program with its own globals and data stacks. Values travel between them through queues, one per type.

You start a worker with `thread_spawn`, which returns an id, and wait for it to finish with

`thread_wait(id)`. Here is what "its own copies of the globals" means in practice:

```
var hatched = 0;

thread_fn hatch() {
    hatched += 10;
    print("worker sees ", hatched);
}

fn main() {
    hatched = 1;
    let id = thread_spawn(hatch);
    thread_wait(id);
    print("main sees ", hatched);
}
```

```
worker sees 11
main sees 1
```

The worker's copy of `hatched` was taken from `main`'s at the moment of the spawn. After that the two are unrelated, and the worker's change never reaches `main`.

Threads need a real C compiler. The in-process backend that `goose file.goose` uses cannot run them, so build threaded programs with `goose build`:

```
thread_fn idle() { }

fn main() { thread_wait(thread_spawn(idle)); }
```

```
JIT mode does not support threads yet (TinyCC
    cannot place thread-local storage in an in-
    memory run); compile with -o and a C
    compiler instead
```

Typed queues 20.2

Values move between threads through queues, and there is one queue per type. `qput(v)` puts a value on the queue for its type, and `qget<T>()` takes the next `T`, waiting until there is one. Because the queue is chosen by the type, sending a value needs no channel object:

```
struct Job { pen: i32, geese: i32 }
struct Tally { pen: i32, eggs: i64 }

fn count_eggs(geese: i32) -> i64 {
    var total: i64 = 0;
    for g in geese { total += (g * 7 + 3) % 5; }
    return total;
}

thread_fn counter() {
    loop {
        let job = qget<Job>();
        guard job.pen >= 0 else { break; }
            // a negative pen means stop
        qput(Tally { pen: job.pen, eggs:
            count_eggs(job.geese) });
    }
}

fn main() {
    let workers = 4;
    for p in 8 { qput(Job { pen: p as i32,
        geese: (100 + p * 10) as i32 }); }
    for w in workers { qput(Job { pen: -1,
        geese: 0 }); }
    var ids: i64[>..] = [];
    for w in workers {
        ids.push(thread_spawn(counter)); }
    var by_pen: i64[8] = [0; 8];
    for p in 8 {
        let t = qget<Tally>();
        by_pen[t.pen] = t.eggs;
    }
    for id in ids { thread_wait(id); }
    print(by_pen);
}
```

```
[200, 220, 240, 260, 280, 300, 320, 340]
```

The tallies arrive in whatever order the workers finish, so `main` files each one under its pen number. Every worker reads from the same queue of `Job`s, which is how the work is shared out.

The other queue operations are `qget<T>(ms)`, which waits at most `ms` milliseconds and returns a `bool` saying whether a value came, `qpoll<T>()`, which never waits, and `qlimit<T>(n)`, which caps a queue's length so that `qput` returns `false` when it is full. `hardware_threads()` returns a sensible number of workers for the machine.

> **From Odin**
>
> This is closer to Go's channels than to Odin's `core:sync`. There is no `Mutex` or `Atomic` to reach for, because there is no memory that two threads can both write.

What can cross: flat values 20.3

A value that goes through a queue must be **flat**: it must contain no references and no slices, at any depth. A reference from one thread's memory would mean nothing in another thread's:

```
struct Note { view: u8[:] }

fn main() {
    var buf: u8[>..] = [];
    buf.append("honk");
    qput(Note { view: buf[..] });
}
```

```
error: queue elements must be flat (§11.2), not
    Note
```

Flat does not mean fixed-size. A struct with a `u8[]` string, or a `T[]` of results, is flat, because its variable-size parts are stored inline. So a message carries its data with it, and sending it is a single copy of contiguous bytes:

```
struct Report { pen: i32, names: u8[][] }

thread_fn reporter() {
    qput(Report { pen: 7, names: ["Ada", "Bea",
        "Cleo"] });
}

fn main() {
    let id = thread_spawn(reporter);
    let r = qget<Report>();
    thread_wait(id);
    print("pen ", r.pen, ": ", r.names);
}
```

```
pen 7: ["Ada", "Bea", "Cleo"]
```

From Rust

"Flat" is roughly `Send + 'static` for owned data, without any borrowed parts. Because a Goose value with strings and arrays inside is still one block, there is nothing like an `Arc<Mutex<Vec<String>>>` to build, and nothing to clone deeply when you send it.

Workers that fail 20.4

When a worker started with `thread_spawn` fails an assertion or a bounds check, the whole program stops, as it would on the main thread. A worker started with `thread_spawn_contained` is different: its failure ends only that worker. `thread_wait`

returns `false` for it, and the form that takes a builder appends the failure message:

```
thread_fn fragile(n: i64) {
    assert(n > 2);
    print("worker ", n, " finished");
}

fn main() {
    var why: u8[>..] = [];
    for n in [1, 5] {
        if
            !thread_wait(thread_spawn_contained(
            fragile, n), why) {
            print("worker ", n, " failed: ",
                why);
            why.clear();
        }
    }
    print("main carries on");
}
```

```
worker 1 failed: assert failed (main.goose:2)
worker 5 finished
main carries on
```

The example also shows arguments passed to a worker. Like queue messages, they must be flat.

Everything a worker owned is released when it ends, whether it returned or failed. Workers still running when `main` returns are stopped.

Design note: no shared memory 20.5

Shared memory is what makes concurrent programming hard. Every piece of data two threads can see needs a rule about who may touch it and when, and the compiler or the programmer must enforce the rule. Rust

enforces it with `Send`, `Sync` and borrowing, and C leaves it to you.

Goose removes the shared data. That is a real limitation: some algorithms, such as a parallel loop over a big array that every thread writes a part of, want shared memory, and in Goose they must instead send the parts out and gather the results back. The memory model softens the cost, because sending a flat value is one `memcpy` of a contiguous block, often no more work than a cache miss would have been. And what you get in return is a concurrency model with nothing left to get wrong, apart from the logic of the messages themselves.

Exercises 20.6

1. Write a program with two workers that each sum half of the numbers from 1 to 1000000 and send their sums back to `main` as a `struct Part { sum: i64 }`.
2. Send a `struct Batch { weights: f32[] }` holding five weights to a worker that sends back their average.
3. What does a worker that increments a global, and then sends it back to `main` in a message, tell you about the global's value in `main`? Write the program and check.

Chapter 21

Saving and Loading Bytes

A structure built from self-relative links means the same thing wherever its bytes are. Chapter 13 pointed that out in passing. This chapter cashes it in: saving such a structure is writing its bytes, and loading it is reading them back, with no serializer, no schema and no fixing up of pointers. The interesting part is the loading, because bytes from outside the program are the one place where the compiler cannot vouch for the references inside them.

Three built-ins 21.1

Built-in	Does
`to_bytes(a)`	a framed *image* of the array: a length, then its element bytes
`bytes_of(a)`	a read-only view of the element bytes, with nothing copied
`from_bytes<T[>..]>(b)`	checks an image, and builds the array from it if it is valid

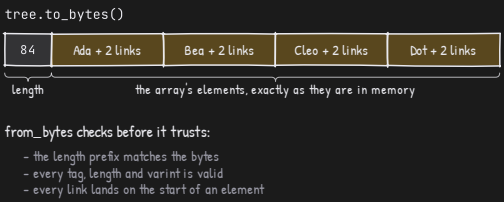
Here is the binary tree of geese from Chapter 13 saved to a file and loaded back:

```
import os;

struct Node { name: u8[..8], weight: i32, left:
    Node&<u32>?, right: Node&<u32>? }

fn insert(tree: Node[>..]&, name: u8[:], weight:
    i32) {
    if tree.len == 0 { tree.push(Node { name:
        name, weight: weight }); return; }
    var cur .= tree[0];
    loop {
        if weight < cur.weight {
            let l = cur.left;
            if l { cur .= l; continue; }
            cur.left .= tree.push(Node { name:
                name, weight: weight });
            return;
        }
        let r = cur.right;
        if r { cur .= r; continue; }
        cur.right .= tree.push(Node { name:
            name, weight: weight });
        return;
    }
}

fn lightest(tree: Node[>..]&) -> u8[:] {
    var cur .= tree[0];
    loop {
        let l = cur.left;
        guard l else { break; }
        cur .= l;
    }
    return cur.name[..];
}

fn main() {
    var tree: Node[>..] = [];
    insert(tree, "Ada", 45);
    insert(tree, "Bea", 52);
    insert(tree, "Cleo", 38);
    insert(tree, "Dot", 41);
    let image = tree.to_bytes();
    print(image.len, " bytes, ",
        bytes_of(tree).len, " of them nodes");
    write_file("flock.bin", image);

    var disk: u8[>..] = [];
    read_file("flock.bin", disk);
```

```
    var loaded, ok = from_bytes<Node[>..]>
        (disk);
    print(ok, ", ", loaded.len, " nodes, the
        lightest is ", lightest(loaded));
    insert(loaded, "Eve", 30);
    print("after one more insert: ",
        lightest(loaded));
    delete_file("flock.bin");
}
```

```
85 bytes, 84 of them nodes
true, 4 nodes, the lightest is Cleo
after one more insert: Eve
```

The image is the 84 bytes of the four nodes exactly as they sit in memory, preceded by a one-byte length. Loading copied the bytes into a new grow-only array, and that array is an ordinary one. You can walk it, index it, and keep inserting into it, as the last line does.



An image is a length followed by the array's element bytes. from_bytes checks the whole image before any of it becomes a value.

Verification 21.2

`from_bytes` is where safety could break down. Everywhere else, the compiler has proved every reference valid before the program ran. But a file can contain anything: random bytes, a file from

an older version, or bytes deliberately crafted to point somewhere they should not.

So `from_bytes` **checks the image before it becomes a value**. It checks the framing, every enum tag, every length, that every varint is in its canonical form, and that every link lands exactly on the start of an element in the same image. If anything is wrong, you get `false` and an empty array, never a wild reference:

```
struct Node { name: u8[..8], weight: i32, left:
    Node&<u32>?, right: Node&<u32>? }

fn main() {
    var tree: Node[>..] = [Node { name: "Ada",
        weight: 45 }];
    let b .= tree.push(Node { name: "Bea",
        weight: 52 });
    tree[0].right .= b;
    var image = tree.to_bytes();
    var good, ok = from_bytes<Node[>..]>(image);
    image[image.len - 4] = 99;
        // corrupt a link
    var bad, bad_ok = from_bytes<Node[>..]>
        (image);
    print(ok, " ", good.len, ", then ", bad_ok,
        " ", bad.len);
}
```

```
true 2, then false 0
```

What the check guarantees is *safety*, not *integrity*. A file whose name bytes were edited still describes a perfectly well-formed tree, and loads fine. If you need to know that a file is the one you wrote, add a checksum.

From Rust

This is what `serde` plus a format crate would do, with one difference: no code runs per field. Validating is a scan over the bytes, and the "decoded" value is the same bytes copied into place. Formats such as FlatBuffers and Cap'n Proto work on the same principle.

From C

A C program that `fwrite`s a struct full of pointers and `fread`s it back is storing addresses that are meaningless in the next run, and trusting whatever the file says. Here the links are offsets, which survive the trip, and the loader checks them.

Variable-size records 21.3

Images work for any array, including the variable-size records from Chapter 10. The image of a diary of entries with inline notes is just the diary's bytes:

```
struct Entry { day: varint, note: u8[varint] }

fn main() {
    var diary: Entry[>..] = [];
    diary.push(Entry { day: 1, note: "first egg"
        });
    diary.push(Entry { day: 3, note: "two eggs"
        });
    let image = diary.to_bytes();
    var back, ok = from_bytes<Entry[>..]>
        (image);
    print(ok, ", ", image.len, " bytes: ",
        back);
}
```

```
true, 22 bytes: [Entry { 1, "first egg" }, Entry
    { 3, "two eggs" }]
```

What cannot be saved 21.4

A link measured from a *named global pool*, `T&<u32 in pool>`, depends on the pool, which is not part of the image, so an array holding such links cannot be written out:

```
struct Goose { name: u8[..8], mate: Goose&<u16
    in flock>? }

var flock: Goose[>..] = [];

fn main() {
    flock.push(Goose { name: "Ada", .. });
    print(bytes_of(flock).len);
}
```

```
error: bytes_of cannot write Goose[>..] out:
    Goose&<u16 in flock>? is measured from
    flock, not from the image
```

Plain references and slices cannot be saved either, for the same reason. Use self-relative links for data that will be saved.

Writing without copying 21.5

`to_bytes` builds a new array holding the image. When the data is large, you can avoid that copy: write a length prefix with `format_uleb` from `std`, then append the `bytes_of` view straight to the file:

```
import std;
import os;

fn main() {
    var weights: i32[>..] = [];
    for i in 1000 { weights.push(i as i32); }
    var head: u8[>..] = [];
    format_uleb(head, bytes_of(weights).len);
    write_file("weights.bin", head);
    append_file("weights.bin",
        bytes_of(weights));
    var disk: u8[>..] = [];
    read_file("weights.bin", disk);
    var back, ok = from_bytes<i32[>..]>(disk);
    print(ok, " ", back.len, " ", back[999], "
        ", disk.len, " bytes on disk");
    delete_file("weights.bin");
}
```

```
true 1000 999 4002 bytes on disk
```

Exercises 21.6

1. Save the diary above to a file with `write_file_atomic`, load it back, and add a third entry to the loaded copy.
2. Take the image of a `u8[][>..]` holding three words and print every byte of it. Can you see where each word's length is stored?
3. Write a function that loads an image of geese and returns either the array or an error message, using a trailing `bool`. What should it return for an empty file?

Chapter 22

Talking to C

Goose compiles to C, so calling C is not a foreign-function interface in the usual sense. A Goose call to a C function compiles to a plain C call, with no marshalling and no wrapper. This chapter shows the three ways to reach C code, which Goose values can cross the boundary, and how to go the other way and call Goose from a C program.

`extern fn` 22.1

A function declared with `extern fn` and no body is a C function. The Goose signature describes it. If the C name differs from the Goose name, give it after an `=`:

> **Newer than v2026.3**
>
> This example uses a feature added to Goose after the v2026.3 release, so it is not checked against the compiler the rest of the book is.

```
extern fn cube_root(x: f64) -> f64 = "cbrt";
extern fn hypot(x: f64, y: f64) -> f64;

fn main() {
    print(cube_root(27.0), " ", hypot(3.0,
        4.0));
}
```

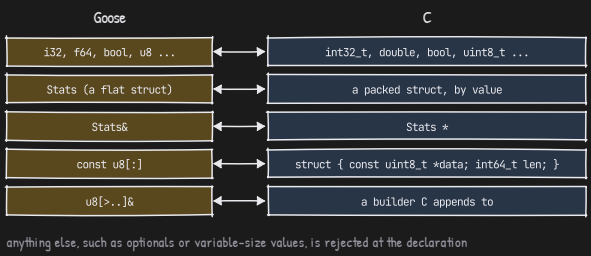
```
3.0000000000000004 5.0
```

Both functions come from the C maths library, which every Goose program links. The standard

library's `math` and `os` modules are written this way.

What can cross 22.2

Only values with a plain C shape can be passed to or returned from C:



Goose types and the C shapes they cross as. Anything without a plain C shape is rejected where the extern fn is declared.

- the scalars and `bool`;
- fixed-size *flat* values (structs, fixed arrays, fixed-mode enums) by value, as their packed C layout;
- a reference to such a value, as a pointer;
- a slice of such values, as a `{ data, len }` struct;
- a `u8[>..]&` builder, which C appends to through a runtime function.

Results are a single scalar or flat fixed-size value, or nothing. Anything else is rejected at the declaration, before any call is compiled:

```
struct Label { text: u8[] }

extern fn show(l: Label);

fn main() {}
```

```
error: extern fn show: parameter l of type
    Label cannot cross to C: only fixed-size
    values cross by value
```

Writability is part of the signature too. A parameter declared `u8[:]` promises C may write through it, so it cannot be given a string literal. Declare it `const u8[:]` if C only reads:

```
extern fn shout(s: u8[:]);

fn main() { shout("honk"); }
```

```
error: extern fn shout: parameter s of type
    u8[:] takes a writable value; a read-only
    one needs the parameter declared const
    (§9.5)
```

C must not keep a pointer it was given after the call returns. Goose cannot check what C does, so this is the one place where the safety of a program depends on code the compiler cannot see.

A program's own C 22.3

C code that belongs to your program goes in files listed in a *sidecar*, `main.build` beside `main.goose`. `goose run` compiles it into a library that the in-process run loads, and `goose build` links it into the executable. Here is a small C file

that summarizes an array of weights and hashes a string:

```
#include <stdint.h>

typedef struct { const int32_t *data; int64_t
    len; } i32_slice;
typedef struct { const uint8_t *data; int64_t
    len; } u8_slice;
typedef struct __attribute__((packed)) {
    int32_t lo, hi;
    double mean;
} Stats;

void weigh_all(i32_slice xs, Stats *out) {
    int64_t sum = 0;
    out->lo = out->hi = xs.data[0];
    for (int64_t i = 0; i < xs.len; i++) {
        if (xs.data[i] < out->lo) out->lo =
            xs.data[i];
        if (xs.data[i] > out->hi) out->hi =
            xs.data[i];
        sum += xs.data[i];
    }
    out->mean = (double)sum / (double)xs.len;
}

uint32_t checksum(u8_slice s) {
    uint32_t h = 0;
    for (int64_t i = 0; i < s.len; i++) h = h *
        31 + s.data[i];
    return h;
}
```

The sidecar names the C files. It uses Goose-like syntax:

```
c_sources = ["impl.c"];
```

And the Goose side declares the functions:

```
struct Stats { lo: i32, hi: i32, mean: f64 }

extern fn weigh_all(xs: i32[:], out: Stats&);
extern fn checksum(s: const u8[:]) -> u32;

fn main() {
    let weights: i32[5] = [45, 52, 38, 41, 60];
    var s = Stats { 0, 0, 0.0 };
    weigh_all(weights, s);
    print(s, " ", checksum("goose"));
}
```

```
Stats { 38, 60, 47.2 } 98539801
```

Two details matter. The C struct is declared `packed`, because Goose structs have no padding (Chapter 10). Here the fields happen to line up anyway, but the attribute makes the C layout match whatever the fields are. And the slice arrives as a two-field struct passed by value.

A sidecar can also list include directories, defines and system libraries, with per-platform variants, and a package can build its C with CMake instead. The official `docs/packages.md` has the details.

Shared libraries, bound at startup 22.4

An `extern "library" { … }` block declares functions to be looked up in a shared library when the program starts. No C is compiled at all, so this works even with no C compiler installed:

Newer than v2026.3

> This example uses a feature added to Goose after the v2026.3 release, so it is not checked against the compiler the rest of the book is.

```
extern "libm.so.6" {
    fn cbrt(x: f64) -> f64;
    fn root(x: f64) -> f64 = "sqrt";
}

fn main() { print(cbrt(8.0), " ", root(2.0)); }
```

```
2.0 1.4142135623730951
```

A plain name such as `"SDL2"` is turned into the platform's file name (`SDL2.dll`, `libSDL2.dylib` or `libSDL2.so`), and a dotted name like the one above is used as written. Because the right name often differs between platforms, the sidecar can map names per platform: `dylibs_linux = ["SDL2=libSDL2-2.0.so.0"];`. A missing library or symbol is reported when the program starts, naming what it tried.

When a library hands back a pointer to memory it owns, such as a `const char *`, declare it as a `u64`: a handle that Goose can pass back but not read through. The `ffi` module's `cstr` and `bytes` copy what the pointer points at into Goose storage.

> **From Odin**
>
> `extern "lib" { … }` plays the role of Odin's `foreign import` and `foreign lib { … }` blocks. The difference is that Goose resolves the symbols at startup rather than at link time, unless you use a sidecar.

Calling Goose from C 22.5

`export fn` goes the other way: it makes a Goose function callable from C. Compile the program with `--header` and the compiler writes a header for a C host program, and C code with no `main` of its own:

```
var calls: i32 = 0;

export fn honk_count(times: i32) -> i32 {
    calls += 1;
    return times * 2 + calls;
}

export fn greeting() -> const u8[:] { return
    "Hellope from Goose" }
```

```
$ goose --standalone --header main.h -o main.c
    main.goose
$ cc main.c host.c -o host -lm -lpthread
```

The host calls `goose_init` once, which sets up the runtime, initializes the globals and runs the Goose `main` if there is one. A `const u8[:]` result comes back as a pointer, with its length written through an extra argument. The bytes are not NUL-terminated.

```c
#include <stdio.h>
#include "main.h"

int main(int argc, char **argv) {
    goose_init(argc, argv);
    int64_t len;
    const uint8_t *msg = greeting(&len);
    fwrite(msg, 1, (size_t)len, stdout);
    int first = honk_count(3);
    int second = honk_count(3);
    printf("\n%d %d\n", first, second);
    return 0;
}
```

```
Hellope from Goose
7 8
```

`--standalone` puts the runtime inside the generated C file, so the C compiler needs nothing else. Exported functions must be called from the thread that called `goose_init`.

> **From C**
>
> The generated C is ordinary C99 with a few compiler-specific attributes. You can add it to an existing C or C++ build like any other source file, which makes Goose usable for one performance-sensitive part of a larger program.

Exercises 22.6

1. Declare `extern fn floor(x: f64) -> f64;` and `extern fn fmod(x: f64, y: f64) -> f64;` and compare `fmod(-7.0, 3.0)` with Goose's own `-7.0 % 3.0` and `-7 % 3`.

2. Write a C function `void count_vowels(u8_slices, int64_t *out)` in a sidecar file, and call it from Goose on a few words.

3. Why can C not be given a `Goose?` or a `u8[]` directly? What would you pass instead?

Chapter 23

Performance

Goose is meant to be fast in the way C is fast: values sit in contiguous memory, there is no garbage collector, and allocating is moving a pointer. Most of a Goose program's speed comes from that model without any effort. This chapter is about the remaining part: the checks that safety adds, how to see which of them are left, how to remove the rest, and how data layout affects speed more than any of them.

Measure a real build 23.1

`goose file.goose` compiles in process with a small, fast C compiler, which is the right tool for trying things out and the wrong one for timing them. For timing, build with the system's C compiler:

```
$ goose build flock.goose -o flock
$ ./flock
```

For the loops measured in this chapter, the in-process run was between three and eight times slower than `goose build`. A `--debug` build adds overflow and conversion checks to every operation, and was two to four times as slow as a normal build.

The `os` module's `clock()` returns seconds of processor time as an `f64`. When timing, make

sure each repetition does work that the C compiler cannot skip or move. A loop that recomputes the same sum twenty times may compile to a loop that computes it once, before the first `clock()`.

Bounds checks 23.2

Every index into an array or a slice is checked, and a failed check aborts the program. The compiler removes each check that it can prove can never fail. That analysis runs on every build, and you can ask it what it did. `--bce-lines` reports, for each line, how many checks were elided and how many were kept:

```
fn dot(a: f32[:], b: f32[:]) -> f32 {
    var t: f32 = 0.0;
    for i in a.len { t += a[i] * b[i]; }
    return t;
}

fn fill(n: i64) -> f32[>..] {
    var v: f32[>..] = [];
    for i in n { v.push(i); }
    return v;
}

fn main() {
    let a = fill(4);
    let b = fill(4);
    print(dot(a, b));
}
```

```
bce-line: main.goose:3: 1 elided, 1 kept
```

The check on `a[i]` went, because `i` counts up to `a.len`. The check on `b[i]` stayed, because

nothing says that `b` is as long as `a`. The cheapest fix is to say so, once, before the loop:

```
fn dot(a: f32[:], b: f32[:]) -> f32 {
    assert(a.len == b.len);
    var t: f32 = 0.0;
    for i in a.len { t += a[i] * b[i]; }
    return t;
}
```

```
bce-line: main.goose:4: 2 elided, 0 kept
```

The `assert` is one comparison per call instead of one per element, and it documents the function's real precondition, which the original version left for a reader to discover. This is the most useful idiom in the chapter: **state the lengths a kernel relies on at its top**.

The summary the compiler prints on stderr after every compile gives the totals for the whole program instead, along with what the optimizer did:

```
$ goose check dot.goose -O2
parsed ok: 3 top-level declarations, 1 file(s)
typechecked ok: 3 specialization(s), 0 struct/0
    enum instance(s); optimized -O2: 3 inlined,
    0 base case(s), 0 folded, 2 propagated, 0
    tail loop(s), 0 reassociated
bce: elided 2/2 index and 0/0 slice checks
```

What the analysis knows 23.2.1

The analysis knows the ordinary facts a loop and its conditions establish: a `for i in a.len` loop, an `if i < a.len`, an `assert`, a check already passed.

The language adds some facts that C cannot offer:

- A grow-only array shrinks only at a `pop`, `resize` or `clear` that the analysis can see, so a length checked before a `push` still holds after it.
- Integer `%` is Euclidean, so `x % a.len` is a valid index for every `x`.
- A fixed array's length and a limited array's capacity are constants.
- The whole program is compiled at once, so facts cross calls. When every caller of a function passes two arrays of the same length, the function can use that, even without an `assert`.

The `%` rule makes wrapping around an array free of checks:

```
fn smooth(samples: f32[:], width: i64) ->
    f32[>..] {
    var out: f32[>..] = [];
    for i in samples.len {
        var t: f32 = 0.0;
        for k in width { t += samples[(i + k) %
            samples.len]; }
        out.push(t / width);
    }
    return out;
}

fn main() {
    let s: f32[6] = [1.0, 2.0, 3.0, 4.0, 5.0,
        6.0];
    print(smooth(s, 3));
}
```

```
bce-line: main.goose:5: 1 elided, 0 kept
```

```
[2.0, 3.0, 4.0, 5.0, 4.0, 3.0]
```

> **From C**
>
> A C compiler removes checks too, when they are written out as `if`s, but it cannot assume that a pointer argument does not alias a global or that a `realloc`ed buffer has not shrunk. Goose's analysis can, because the language rules out both.

How much a check costs 23.2.2

Less than you might expect. A kept check is a comparison and a branch that is never taken, which a modern processor predicts perfectly. In the tests for this chapter, a random-access loop ran at the same speed with and without its checks, because its time went on cache misses. Where a check does cost something is in a tight loop over arrays, where the C compiler would otherwise turn the loop into vector instructions. A branch that might leave the loop part-way through can prevent that. So look at the `--bce-lines` report for your hot loops, and do not worry about the rest.

Index types 23.3

There is one kind of check that no analysis of the code can remove: an index that was read out of another array, as in `weights[ids[k]]`. Whether it is in range depends on the *contents* of `ids`, and the compiler reasons about lengths, not about every element of an array.

An **index type** moves that check to where the index is made. The type `u32 in weights` is a `u32` that is always below `weights.len`. Converting a plain integer with `as` checks it once, and from then on indexing `weights` with it needs no check, since the array it names is a grow-only global and can never get shorter:

```
var weights: i32[>..] = [];
type Id = u32 in weights;

fn gather(ids: Id[:]) -> i64 {
    var t = 0;
    for id in ids { t += weights[id]; }
    return t;
}

fn main() {
    for w in [45, 52, 38, 41, 60] {
        weights.push(w as i32); }
    var ids: Id[>..] = [];
    for i in [4, 0, 2] { ids.push(i as Id); }
    print(gather(ids));
}
```

```
bce-line: main.goose:6: 1 elided, 0 kept
bce-line: main.goose:13: 0 elided, 1 kept
```

```
143
```

The check on line 6 is gone. The one kept on line 13 is the conversion `i as Id`. That conversion is where a bad index is caught:

```
fn main() {
    for w in [45, 52, 38] { weights.push(w as
        i32); }
    var ids: Id[>..] = [];
    for i in [2, 7] { ids.push(i as Id); }
}
```

```
goose runtime error: index 7 out of bounds
    (length 3)
```

The array named by an index type must be a global grow-only array, and it can then never be shrunk anywhere in the program. The type names the array, so `u32 in weights` and `u32 in names` cannot be mixed up. Index types fit graphs, meshes and other data where one table refers to rows of another. A field of a struct can be one, and an image loaded with `from_bytes` has every such field checked against its array.

There is also a *bound type*, `u32 upto weights`, which may equal the length. It is the type for the end of a range, such as a node's `lo..hi` range of edges.

> **From Rust**
>
> An index type is a lightweight version of the "branded index" pattern, where a lifetime or a type parameter ties an index to one collection. In Goose the tie is to a named global, so it needs no generics and no closures.

`checkfree`: no checks allowed 23.4

Writing `checkfree` before `fn` turns "no checks left here" from a hope into a requirement. The function compiles only if no bounds check remains anywhere in it, and the error says which check stayed and why:

```
checkfree fn gather(xs: i32[:], ids: i64[:]) ->
    i64 {
    var t = 0;
    for k in ids.len { t += xs[ids[k]]; }
    return t;
}

fn main() {
    let xs: i32[5] = [1, 2, 3, 4, 5];
    let ids: i64[3] = [4, 0, 2];
    print(gather(xs, ids));
}
```

```
main.goose:3: error: checkfree fn gather keeps
    a bounds check: ids[k] is read out of an
    element, a field or a call result, whose
    value the analysis does not know; an index
    type (`u32 in a`) for what is stored there
    carries it (§3.11)
```

The `gather` that takes `Id`s above compiles as a `checkfree fn`. Facts from callers do not count here: even if every call passes arrays of equal length, a `checkfree` function must establish what it relies on itself, usually with an `assert`, so it stays check-free whoever calls it. A `checkfree` function can call only other `checkfree` functions, builtins and C functions, since a check in a callee would run inside it. It may still `assert`, and its asserts are how it states its preconditions.

`checkfree` does not change what the code computes or how it is compiled. It is a test that runs at compile time, and it protects a hot loop from a later edit that quietly brings a check back.

Asking for inlining and vector code 23.5

Two more qualifiers are hints to the code generator:

- `inline fn` asks for the function to be inlined at every call, however large it is. Without it, the compiler decides from size and the number of calls.
- `simd fn` asks for the function to be compiled several times, once for the baseline instruction set and once for each wider one (AVX2 and AVX-512 on x86-64), and for each call to run the widest version that the processor supports. It suits a function containing a long loop over bytes or numbers. It needs clang; other compilers and the in-process run ignore it.

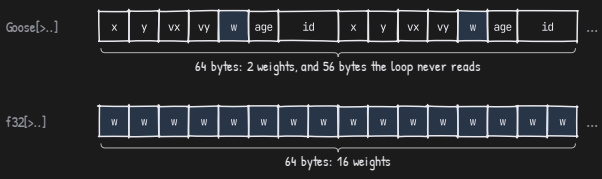
```
simd fn brighten(px: u8[:], by: u8) {
    for i in px.len {
        px[i] = if px[i] > 255 - by { 255 } else
            { px[i] + by };
    }
}

inline fn clamp(x: i64, lo: i64, hi: i64) -> i64
    {
    if x < lo { lo } else if x > hi { hi } else
        { x }
}

fn main() {
    var img: u8[8] = [0, 40, 100, 200, 250, 255,
        7, 128];
    brighten(img, 30);
    var out: u8[>..] = [];
    for p in img { format(out, p, " "); }
    print(out, clamp(300, 0, 255));
}
```

```
30 70 130 230 255 255 37 158 255
```

Neither qualifier changes a program's results. A `simd` function computes exactly what its baseline version does, including the rounding of every floating-point operation. `-O0`, `-O1` and `-O2` set how much the Goose compiler inlines on its own; the default is `-O2`.

Data layout 23.6

The largest speedups usually come from arranging data so that a loop reads only what it needs. A loop that sums one field of an array of structs reads the whole struct into the cache, field by field, to use four bytes of it:



Summing the weights of an array of structs reads two weights per cache line. A separate array of weights gives sixteen.

Here are both layouts. The code is the interesting part; the timings depend on the machine, so they are described below rather than printed:

```
import os;

struct Goose {
    x: f32, y: f32, vx: f32, vy: f32,
    weight: f32, age: i32, id: i64,
}

fn total_weight(flock: Goose[:]) -> f64 {
    var t: f64 = 0.0;
    for g in flock { t += g.weight; }
    return t;
}

fn total(weights: f32[:]) -> f64 {
    var t: f64 = 0.0;
    for w in weights { t += w; }
    return t;
}

fn main() {
    let n = 4000000;
    var flock: Goose[>..] = [];
    var weights: f32[>..] = [];
    for i in n {
        let w: f32 = 3.0 + i % 5;
        flock.push(Goose { x: 0.0, y: 0.0, vx:
            0.0, vy: 0.0,
                                weight: w, age: 1,
                                    id: i });
        weights.push(w);
    }
    var a: f64 = 0.0;
    var b: f64 = 0.0;
    let t0 = clock();
    for r in 20 {
        flock[r].weight += 1.0;             // so
            no round repeats
        a += total_weight(flock);
    }
    let t1 = clock();
    for r in 20 {
        weights[r] += 1.0;
        b += total(weights);
    }
    let t2 = clock();
    print(a == b, " structs ", t1 - t0, "s,
        array ", t2 - t1, "s");
}
```

Built with `goose build` on the machine this book was tested on, the loop over the separate array of weights was between four and five times faster than the loop over the structs. Neither loop has a bounds check left. The difference is memory traffic alone.

Splitting a struct into parallel arrays, one per field, is called a *structure of arrays*. It is worth doing for the fields that hot loops read, and not otherwise, since code that works on one whole goose at a time reads better with a `Goose`. Goose has no special syntax for it, but grow-only global arrays and index types make it straightforward: the same `u32 in xs` index can serve every parallel array, after an `assert` that the others are as long.

A few more layout points follow from earlier chapters:

- Fixed-size elements can be indexed. Arrays of variable-size elements (`u8[][>..]`, structs with inline strings, variable-mode enums) are compact but sequential, so they suit data that is read in order.
- `varint` fields and links narrower than 64 bits (`T&<u32>`, `T&<u16>`) make data smaller, and smaller data is often faster even with the decoding.
- Goose structs are packed. If a hot field is misaligned, reorder the fields from largest to smallest, or add a `pad`.

Design note: safe by default, fast on request 23.7

C makes the fast thing the default and leaves safety to the programmer. Goose does the reverse, but tries to make the cost of safety visible and removable. Every check is in the program unless the compiler proves it away, the compiler says which checks remain, and `checkfree` lets you insist that there are none. The language makes the proofs easier by design: grow-only arrays never surprise the analysis by shrinking, `%` never goes negative, and whole-program compilation means facts reach across calls. What it does not offer is a way to switch checks off for one function. Proving them unnecessary is always possible, and an `assert` is usually enough.

Exercises 23.8

1. Write `fn add(a: f32[:], b: f32[:], out: f32[:])` that adds two arrays into a third. Run it with `--bce-lines`, then add an `assert` so that no checks are kept.
2. Make `fn add` from exercise 1 a `checkfree fn`. What happens if you remove the assert?
3. A graph stores its nodes in a global `Node[>..]` and its edges as a global array of target nodes. Give the edges an index type, and

write a `checkfree` function that sums the weights of a node's neighbours.

PART VI

Two Programs

The chapters so far have shown each feature in a snippet of its own. This part shows two complete programs, each a page or two long, and looks at the decisions in them: where the data lives, how errors travel, and what the compiler checks. Neither is built up step by step. Each is shown whole, and then taken apart.

Chapter 24

A Word Counter

The first program reads a text file and prints the words that occur most often in it, with their counts. It is a classic small program because it touches files, strings, a hash table and sorting, and in a language with a garbage collector it is about ten lines long. In C it is several pages, mostly memory management. The Goose version is about sixty lines, and none of them manage memory.

The program 24.1

Given this file, `geese.txt`:

```
The goose that lays the golden eggs is not a
    goose you should cook.
A goose is a goose, said the farmer, and the
    farmer's wife agreed:
"Don't cook the goose." So the goose stayed, and
    the eggs kept coming,
and the farmer kept the eggs, and the goose kept
    the farmer honest.
```

the program, run as `goose words.goose -- geese.txt 6`, prints the six most common words:

```
import std;
import os;
import dictionary;

struct Count { word: const u8[:], n: i64 }
type Counts = dictionary<const u8[:], i64>;

fn is_word_byte(c: u8) -> bool {
    (c >= 'a' && c <= 'z') || c == '\''
}

// Counts the words of text into counts, and
    returns how many
// there were. A word is a run of letters and
    apostrophes, and
// text must be lower case.
fn count_words(text: const u8[:], counts:
    Counts&) -> i64 {
    var total = 0;
    var i = 0;
    while i < text.len {
        while i < text.len &&
            !is_word_byte(text[i]) { i += 1; }
        let start = i;
        while i < text.len &&
            is_word_byte(text[i]) { i += 1; }
        if i > start {
            counts.update(text[start..i], 0) {
                it += 1; };
            total += 1;
        }
    }
    return total;
}

// The entries of counts, most frequent first,
    and
// alphabetically on a tie.
fn ranked(counts: Counts&) -> Count[>..] {
    var out: Count[>..] = [];
    counts.each() { w, n => out.push(Count {
        word: w, n: n }); };
    sort(out) { a, b =>
        a.n > b.n || (a.n == b.n &&
            compare(a.word, b.word) < 0)
    };
    return out;
}
```

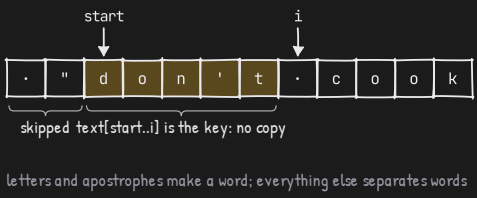
```
fn main() {
    let a = args();
    guard a.len >= 2 else {
        print("usage: words file [how-many]");
        exit(2);
    }
    var top = 10;
    if a.len >= 3 {
        let n, ok = parse_int(a[2]);
        guard ok && n > 0 else {
            print("not a count: ", a[2]);
            exit(2);
        }
        top = n;
    }
    var text: u8[>..] = [];
    guard read_file(a[1], text) else {
        print("cannot read ", a[1]);
        exit(1);
    }
    to_lower(text);

    var counts = Counts {};
    let total = count_words(text, counts);
    let list = ranked(counts);
    print(total, " words, ", counts.count, "
        different");
    for c in list[..min(top, list.len)] {
        var line: u8[>..] = [];
        format_int(line, c.n, 10, 5, ' ');
        format(line, "  ", c.word);
        print(line);
    }
}
```

```
53 words, 24 different
   11  the
    7  goose
    4  and
    3  a
    3  eggs
    3  farmer
```

The rest of the chapter goes through the decisions in it.

One buffer, no copies 24.2

The program reads the whole file into one grow-only array, `text`, and lowers its case in place. After that, **no byte of the input is copied again**. A word is a slice of `text`, a dictionary key is a slice of `text`, and so is the `word` field of every `Count`:



The scanner keeps two positions. The bytes between them are a word, and the slice is the dictionary key.

This is the shape that Goose favours: one large owner, and many small views into it. The views are cheap (a pointer and a length each), and the compiler knows what each one points into. If you tried to keep `list` after `text` was gone, for example by returning it from a function that owned `text`, the program would not compile.

`count_words` takes `text` as `const u8[:]`. It only reads the text, and the dictionary's key type is `const u8[:]` too, so the slices it stores are read-only views. Chapter 18 explains why a dictionary keyed by string slices uses the `const` form.

From C

> A C version would `strdup` each new word into the table, or else keep pointers into a buffer and hope that nobody `realloc`s it. Goose's version is the second design, with the hope replaced by a check: `text` can only grow, and nothing pushes onto it after the counting starts.

Counting with `update` 24.3

```
counts.update(text[start..i], 0) { it += 1; };
```

`update` looks the key up once, inserts `0` if the key is missing, and runs the block on a reference to the value. The obvious alternative, a `get` followed by an `insert` when the key is missing, hashes the word twice for every new word. More importantly, the block form is the one that the borrow rules make easy: the reference `it` cannot outlive the block, so it cannot be held across the next insertion, which might move the table.

Ranking 24.4

`ranked` copies the entries into a `Count[>..]` and sorts it with a block comparing two entries. Each `Count` is a slice and a count, three machine words, so sorting moves small fixed-size values around. The words themselves stay where they are in `text`.

The function returns an array whose elements refer into `text`, which it never sees: it only has `counts`. The compiler treats the result as rooted

in its argument, `counts`, and so in everything that `counts` may refer into. That is the "inferred roots" rule of Chapter 8, and it is what keeps `list` from outliving `text`. It is also conservative, as the next section shows.

`sort` is unstable, so the comparator breaks ties itself, alphabetically. Without that, the order of `a`, `eggs` and `farmer` above, which all occur three times, would depend on the hash table's layout.

The output loop 24.5

Each line of output is built in a fresh builder, with `format_int` padding the count to five columns. It is tempting to make one builder before the loop and `clear` it each time round, as you would in C. The compiler refuses:

```
    let list = ranked(counts);
    var line: u8[>..] = [];
    for c in list[..min(top, list.len)] {
        line.clear();
        format_int(line, c.n, 10, 5, ' ');
        format(line, "  ", c.word);
        print(line);
    }
```

```
error: cannot clear line: the for loop at
    words.goose:67 iterates list[..min(top,
    list.len)], whose elements may refer into
    line (§5.1)
```

The words in `list` came out of a dictionary, by way of a function. The compiler knows they are views of byte arrays that belong to `main`, but no longer which ones, so it assumes they might be

views of any of them, `line` included. Clearing `line` while the loop could still be reading such a word would leave the word pointing at discarded bytes, so the `clear` is rejected. In this program no word points into `line`, but the check errs on the side of safety. The fix costs nothing to speak of: a builder made inside the loop is a stack pointer moved forward, and the end of each iteration moves it back.

Errors and exit codes 24.6

The program handles three failures: no file name, a count that is not a positive number, and a file it cannot read. In each case it prints a message and calls `exit` with a non-zero code, which ends the program immediately. Since `exit` never returns, the compiler treats the rest of a `guard`'s `else` as unreachable, which is what lets the `guard` stand in front of code that uses `text` or `n`.

A library function would not `exit`. It would return a `bool` or use `return … from`, as the calculator in the next chapter does. In `main`, of a program this small, exiting is the right thing.

How fast is it? 24.7

Built with `goose build`, on the machine this book was tested on, the program counted a 51 MB file of 8 million words, about 150,000 of them different, in under 0.4 seconds. Nearly all of that

is the scanning and the hashing. The allocations, of which there are only a handful in the whole run, do not show up at all.

Exercises 24.8

1. Make the counter ignore words of one or two letters.
2. Print, after the table, the number of words that occur exactly once.
3. Change `count_words` so that it takes `u8[:]` instead of `const u8[:]`. What else has to change?
4. Make the program read from standard input when it is given no file name, using `read_stdin` from `os`.

Chapter 25

A Calculator

The second program is an interactive calculator. It reads expressions, one per line, and prints their values. It knows the usual operators and their precedence, parentheses, a few functions, and variables set with `let`. Its interest lies in the parser: a set of mutually recursive functions, nested inside one driver function, which report every error with `return … from`, and so contain no error-handling code at all.

The program 25.1

Given this input:

```
1 + 2 * 3
(1 + 2) * 3
-2 ^ 2
2 ^ 3 ^ 2
let r = 1.5
pi * r ^ 2
sqrt(2) * sqrt(2)

1 / (r - 1.5)
2 * (3 + 4
sqrt(-1)
wings * 2
3 4
let = 5
2 * $
```

and this program, run as `goose calc.goose < input.txt` (or with no redirection, to type expressions by hand):

```
import std;
import math;
import os;
import dictionary;

let NAME_MAX = 16;
let DEPTH_MAX = 200;
type Name = u8[..NAME_MAX];
type Vars = dictionary<Name, f64>;

fn is_digit(c: u8) -> bool { c >= '0' && c <=
    '9' }
fn is_letter(c: u8) -> bool {
    (c >= 'a' && c <= 'z') || (c >= 'A' && c <=
        'Z') || c == '_'
}
fn is_name_byte(c: u8) -> bool { is_letter(c) ||
    is_digit(c) }

// One line of input: `let name = expr`, or an
    expression. Returns
// the value and "", or 0.0 and a message saying
    what went wrong
// and where.
fn evaluate(src: const u8[:], vars: Vars&) ->
    f64, u8[] {
    var pos = 0;
    var depth = 0;                          //
        parentheses open

    fn fail(at: i64, what: u8[:]) -> f64 {
        let msg = str("column ", at + 1, ": ",
            what);
        return 0.0, msg from evaluate;
    }
    fn byte() -> u8 { if pos < src.len {
        src[pos] } else { 0 } }
    fn peek() -> u8 {
        while byte() == ' ' { pos += 1; }
        return byte();
    }
    fn here() -> i64 { peek(); pos }
    fn eat(c: u8) -> bool {
        if peek() != c { return false; }
        pos += 1;
        return true;
    }
    fn word() -> u8[:] {
        let start = pos;
```

```
        while is_name_byte(byte()) { pos += 1; }
        if pos - start > NAME_MAX { fail(start,
            "name too long"); }
        return src[start..pos];
    }
    fn number() -> f64 {
        let start = pos;
        while is_digit(byte()) || byte() == '.'
            { pos += 1; }
        let v, ok = parse_flt(src[start..pos]);
        return if ok { v } else { fail(start,
            "bad number") };
    }
    fn apply(at: i64, f: u8[:], x: f64) -> f64 {
        if f == "abs" { return abs(x); }
        if f == "sqrt" {
            if x < 0.0 {
                return fail(at, "sqrt of a
                    negative number");
            }
            return sqrt(x);
        }
        if f == "ln" {
            if x <= 0.0 {
                return fail(at, "ln of a number
                    <= 0");
            }
            return log(x);
        }
        return fail(at, str("no function called
            ", f));
    }

    // The grammar, one function per rule:
    //   sum     = product (('+' | '-')
        product)*
    //   product = power (('*' | '/') power)*
    //   power   = '-' power | atom ('^' power)?
    //   atom    = number | name | name group |
        group
    //   group   = '(' sum ')'
    recursive fn sum() -> f64 {
        var v = product();
        loop {
            if eat('+') { v += product(); }
            else if eat('-') { v -= product(); }
            else { return v; }
        }
    }
```

```
    fn product() -> f64 {
        var v = power();
        loop {
            if eat('*') { v *= power(); }
            else if eat('/') {
                let at = here();
                let d = power();
                if d == 0.0 {
                    return fail(at, "division by
                        zero");
                }
                v /= d;
            }
            else { return v; }
        }
    }
    recursive fn power() -> f64 {
        if eat('-') { return -power(); }
        let base = atom();
        return if eat('^') { pow(base, power())
            } else { base };
    }
    fn atom() -> f64 {
        let c = peek();
        let at = pos;                        //
            after the spaces
        if eat('(') { return group(at); }
        if is_digit(c) || c == '.' { return
            number(); }
        if c == 0 { return fail(at, "unexpected
            end of line"); }
        if !is_letter(c) {
            return fail(at, str("unexpected ",
                src[at..at + 1]));
        }
        let n = word();
        if eat('(') { return apply(at, n,
            group(at)); }
        let v = vars.get(n);
        if v { return v; }
        return fail(at, str("unknown name ",
            n));
    }
    // After the '(' at position at: the sum,
        then ')'.
    fn group(at: i64) -> f64 {
        depth += 1;
        if depth > DEPTH_MAX {
```

```
            return fail(at, "too deeply
                nested");
        }
        let v = sum();
        if !eat(')') { return fail(here(),
            "expected )"); }
        depth -= 1;
        return v;
    }
    fn statement() -> f64 {
        var target: Name = "";
        if is_letter(peek()) {
            let start = pos;
            if word() == "let" {
                let at = here();
                target = word();
                if target.len == 0 {
                    return fail(at, "expected a
                        name after let");
                }
                if !eat('=') { return fail(pos,
                    "expected ="); }
            } else {
                pos = start;
            }
        }
        let v = sum();
        if peek() != 0 {
            return fail(pos, "expected an
                operator");
        }
        if target.len > 0 { vars.insert(target,
            v); }
        return v;
    }

    return statement(), "";
}

fn main() {
    var vars = Vars {};
    vars.insert("pi", PI);
    var line: u8[>..] = [];
    while read_line(line) {
        if trim(line).len > 0 {
            let v, err = evaluate(line, vars);
            if err.len > 0 {
                print("error: ", err);
            } else {
```

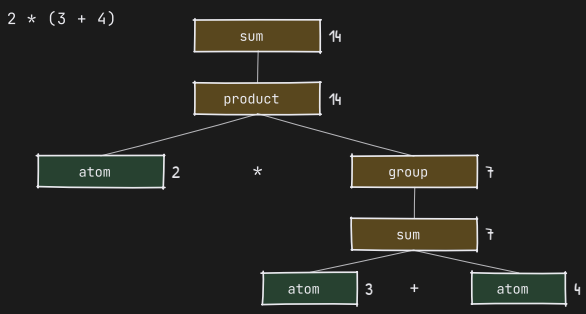
```
                print(v);
            }
        }
        line.clear();
    }
}
```

it prints:

```
7.0
9.0
-4.0
512.0
1.5
7.0685834705770345
2.0000000000000004
error: column 5: division by zero
error: column 11: expected )
error: column 1: sqrt of a negative number
error: column 1: unknown name wings
error: column 3: expected an operator
error: column 5: expected a name after let
error: column 5: unexpected $
```

One function per rule 25.2

The parser is a *recursive-descent* parser: the grammar in the comment has five rules, and each rule is one function that returns the value of the text it matched. There is no syntax tree. Each function computes its value directly from the values of the parts it calls for:



*The calls made to evaluate 2 * (3 + 4), with the value each one returns. Calls that only pass a value up, such as power and the atom around group, are left out below the top two.*

Precedence comes from the order of the calls. `sum` asks `product` for its operands, so `*` binds more tightly than `+`, and `product` asks `power`, so `^` binds more tightly than `*`. Associativity comes from the shape of each function. `sum` and `product` loop, folding each new operand into `v`, which makes `+`, `-`, `*` and `/` associate to the left: `8 - 2 - 1` is `(8 - 2) - 1`. `power` calls itself for its right operand, which makes `^` associate to the right, the convention in mathematics: `2 ^ 3 ^ 2` is `2 ^ 9`, which is 512. And because the unary minus in `power` applies to a whole `power`, `-2 ^ 2` is `-(2 ^ 2)`, which is -4.

Nested functions instead of a parser object 25.3

All the parsing functions are nested inside `evaluate`, and they share its locals `pos` and `depth` and its parameters `src` and `vars` as free variables. In C, they would be functions that all

take a `Parser *`, and in Rust, methods on a `Parser` struct. Here the state needs no type and no parameter, and the functions are declared where they are used.

Nested functions are not closures (Chapter 5). They exist only at compile time, and accessing `pos` from inside `atom` is an ordinary access to a local of `evaluate`'s frame. Nothing is allocated, and the functions cannot escape `evaluate`.

Marking the recursion 25.3.1

Only two of the functions are `recursive fn`. The rule from Chapter 17 is that the *entry* of each cycle of calls is marked. There are two cycles here: `sum` → `product` → `power` → `atom` → `group` → `sum`, entered at `sum` from `statement`, and `power` → `power`, entered at `power` from `product`. Leave the keyword off `sum`, and the compiler names the cycle:

```
fn sum() -> f64 {
    var v = product();
    loop {
        if eat('+') { v += product(); }
        else if eat('-') { v -= product(); }
        else { return v; }
    }
}
```

```
error: recursive call cycle through sum, which
    is not declared `recursive fn` (§7.8)
```

Errors from anywhere 25.4

Every error is reported by a call to `fail`, which leaves the parse at once:

```
fn fail(at: i64, what: u8[:]) -> f64 {
    let msg = str("column ", at + 1, ": ",
        what);
    return 0.0, msg from evaluate;
}
```

The `return … from evaluate` ends the innermost running call of `evaluate`, whatever is between: `fail` itself, `atom`, `group`, any number of nested `sum`s, and `statement`. `evaluate`'s caller receives `0.0` and the message, and `main` prints it. The functions in between have no error paths of their own. They do not check results, propagate failures or return `bool`s, so each one reads as if errors could not happen.

`fail` is declared to return `f64`, although it never returns at all. That lets it be used wherever a value is expected, as in `return if ok { v } else { fail(start, "bad number") }` in `number`, and keeps every function's result type its plain value.

The position of each error is the position where the problem *starts*: the operand that was zero, the name that was unknown, or the end of the line where a `)` was missing. Each function records `at` before it calls further, since by the time an error is found, `pos` has usually moved on.

- `line`, in `main`, is one builder reused for every line of input.
- `src` is a slice of `line`, and every word is a slice of `src`. Like the word counter, the parser copies nothing out of its input.
- The error message is the only thing built per error. It is a `u8[]` returned to `main`, which prints it before the next line is read.
- The variables are a `dictionary<Name, f64>`, and `Name` is `u8[..16]`.

That last point differs from the word counter, and for a good reason. The counter's keys were slices of its one input buffer, which lived for the whole program. Here, a key must outlive the line it was read from, because `line` is cleared and refilled for every line. So the dictionary owns its keys, as inline arrays of up to sixteen bytes, and `insert` copies the name into its slot. The compiler would have rejected a `dictionary<const u8[:], f64>` filled from `line`, since the next `line.clear()` would leave its keys pointing at discarded bytes.

The capacity of `Name` explains the check in `word`. Storing a longer name in a `u8[..16]` aborts the program:

```
type Name = u8[..16];

fn main() {
    let word =
        "a_name_much_longer_than_sixteen";
    var n: Name = word;
    print(n);
}
```

```
goose runtime error: limited array capacity
    exceeded
```

For a calculator, an abort on a long name is the wrong behaviour, so `word` reports an error instead.

Limits 25.6

The other guard is on depth. Each `(` costs a few native stack frames, and a recursion deeper than the native stack ends the program. It is a clean abort, not a crash, but still not what a calculator should do with a line of 100,000 parentheses. So `group` counts the open parentheses and reports `too deeply nested` past 200. Any recursive parser that reads untrusted input should have such a limit.

From Rust

A Rust parser would usually return `Result<f64, ParseError>` from each function and use `?` at every call. The `?`s are explicit about where a failure can occur, which is a real benefit. The Goose version trades that for functions with no error type at all, and keeps the knowledge of the failure in two places: the `fail` calls, and the one caller that handles the message.

Exercises 25.7

1. Add the `%` operator, with the same precedence as `*` and `/`. Should it be Euclidean, like Goose's integer `%`, or follow C's `fmod`?
2. Add functions of two arguments, `max(a, b)` and `min(a, b)`.
3. Make the calculator remember the last value it printed as `ans`.
4. Make an error message show the line with a `^` under the column where the error was found.

Appendix A: Goose for C, Odin and Rust Programmers

This appendix is a phrasebook. For each language, it lists the things you reach for out of habit and what Goose offers instead, with the chapter that explains it. The tables are about habits, not exact equivalence. Several rows translate a mechanism into a different design rather than into different syntax.

From C

A.1

In C	In Goose	Chapter
`malloc` and `free`	a value of variable or resizable size, owned by a scope; freed when the scope ends	6
a buffer that grows with `realloc`	`T[>..]`, or `T[>..<]` if it must also shrink	7
`int a[8]`	`i32[8]`	7
a pointer and a length	a slice, `T[:]`	8
`const T *`	`const T&`, or `const T[:]`	8
`T *` that may be `NULL`	`T?`, narrowed by `if` or `guard`	11
`char *` strings	`u8[:]` views, `u8[]` stored, `u8[>..]` built	9
`sprintf`, `snprintf`	`str(…)`, `format(out, …)`, and `print`	9
a struct with padding	a packed struct; `pad` where the layout must match C	10
a tagged union	`enum` with payloads, and `match`	12
a function pointer table	case functions, or an enum with `match`	12
`qsort` with a comparator	`sort(xs) { a, b => … }`	16
a linked list or tree of `malloc`ed nodes	nodes in one array, joined by `T&<u32>` links	13

In C	In Goose	Chapter
a free list	a `reusable` pool	14
`return -1` and `errno`	a trailing `bool`, an optional, or `return … from`	15
`setjmp` and `longjmp`	`return … from`, or the `exception` module	15
`assert`	`assert`, which is checked in release builds too	15
`pthread_create` and mutexes	`thread_fn`, `thread_spawn`, and typed queues; no shared memory	20
`fwrite` of a struct	`to_bytes`, and `from_bytes` to load it with checks	21
header files and separate compilation	`import`, and whole-program compilation	19
`-O2 -fno-bounds-checks`	bounds checks removed by proof; `checkfree` to insist	23

The biggest change from C is ownership. In C, any code holding a pointer may free what it points at, and nothing records who should. In Goose, storage belongs to the scope that made it, and references into it are checked against that scope.

From Odin A.2

Odin and Goose are both data-oriented languages for people who like C, so much of Odin carries over. The differences are mostly about memory.

In Odin	In Goose	Chapter
`context.allocator`, `context.temp_allocator`	data stacks, chosen by the compiler; no allocator to pass or set	6
`[dynamic]T`	`T[>..]` or `T[>..<]`	7
`[N]T`	`T[N]`	7
`[]T`	`T[:]`	8
`^T`	`T&`, which can never dangle, or `T?` if it may be null	8, 11
`string`	`u8[:]`; `u8[]` when stored; `u8[>..]` when built	9
`strings.Builder`	`u8[>..]`, written with `format`	9
`map[K]V`	`dictionary<K, V>` from the library	18
`Maybe(T)`	`T?` for a reference; a trailing `bool` for a value	11
`union` and type `switch`	`enum` with payloads, and `match`	12

In Odin	In Goose	Chapter
`#soa`	parallel arrays by hand, sharing an index type	23
explicit overload sets, `proc{a, b}`	implicit overloading by parameter types	5
`x->f()`, for structs holding procedures	`x.f(a)`, which calls `f(x, a)` for any function	5
`or_return`	`return … from`, or a trailing `bool`	15
`defer`	`defer`	4
`cast(T)x`, `transmute(T)x`	`x as T`, and `x as! T` when losing bits is intended	3
`%` (truncating) and `%%` (floored)	`%`, Euclidean on integers, and `rem` for the truncating form	3
`foreign import` and `foreign` blocks	`extern fn`, and `extern "lib" { … }`	22
packages as directories	files with `namespace`, and packages declared by the root file	19
`core:thread`, `core:sync`	workers and typed queues	20

Odin trusts you with its allocators. Goose takes that choice away, and in exchange makes every reference safe. If you wrote Odin with arenas and temporary allocators, Goose's data stacks will feel familiar: they are an arena per scope, managed by the compiler.

From Rust A.3

Rust and Goose both promise memory safety without a garbage collector, by very different routes. Rust checks borrows of heap data with lifetimes. Goose has no heap, and checks references against the scopes that own the storage, inferring the roots that Rust would ask you to write as lifetime parameters.

In Rust	In Goose	Chapter
`Box<T>`	nothing: a value is stored where it is declared	6
`Vec<T>`	`T[>..]` or `T[>..<]`	7
`&[T]`, `&mut [T]`	`const T[:]`, `T[:]`	8
`&T`, `&mut T`	`const T&`, `T&`	8
lifetime parameters, `'a`	inferred roots	8
`String`, `&str`	`u8[>..]` or `u8[]`, and `u8[:]`, all bytes rather than checked UTF-8	9
`format!`, `write!`	`str`, `format`	9
`Option<&T>`	`T?`, which is always a reference	11
`Option<T>` for a value	a trailing `bool`, or a default value	11
`enum` with data, and `match`	`enum` with payloads, and `match`	12
`Rc<RefCell<T>>`, `Weak`	a `reusable` pool, and links into it	14
`Result<T, E>` and `?`	a trailing `bool` or error value, `return … from`, `exception`	15
`panic!`	`abort`, which cannot be caught	15
traits and `impl` blocks	overloads, generics without bounds, and method-call syntax	5, 16

In Rust	In Goose	Chapter
closures	trailing blocks and nested functions, passed down but never stored	16
`Box<dyn Trait>`	an `enum` of the possible kinds	12
`std::thread`, `mpsc` channels	`thread_spawn`, and one queue per type	20
`Send`, `Sync`	flat values; nothing is shared	20
`serde`	`to_bytes` and `from_bytes`	21
`extern "C"`, `#[no_mangle]`	`extern fn`, `export fn`	22
`get_unchecked`	an index type, or an `assert` that lets the check be removed	23
Cargo and crates	packages declared in the root file, fetched by `goose`	19

The Rust habit that transfers least well is building structures of owned nodes. A tree of `Box`es becomes nodes in one array; a graph of `Rc`s becomes a pool and links; a `Vec<String>` becomes a `u8[][>..]`, one block with every string inline.

Appendix B: Limitations

Goose gets its safety and its speed by giving things up. Most of the limitations below are the price of the memory model, and are permanent. A few are omissions of the current version. This appendix collects them in one place, so you can judge whether a program suits the language before you start.

Limits of the design B.1

You decide where data lives. Every growable value belongs to a scope, and you choose which. Usually it is the function that builds the value, and there is nothing to decide. When it is not, the data goes into a `reusable` pool (Chapter 14) or is owned further up and passed down.

Recursion cannot hold growable data across a call. A recursive function may use scratch storage that ends before it recurses, but storage that must survive the call is owned outside the cycle and passed in (Chapter 17). Parsers and tree walks are written around this rule, as Chapter 25 shows.

Arrays of variable-size elements are sequential. `u8[][>..]` and arrays of variable-mode enums can be walked but not indexed. To index one,

keep a separate array of references or offsets beside it (Chapter 13).

A fixed-mode enum cannot be pointed into, and a variable-mode one cannot be overwritten. Each container chooses its mode (Chapter 12).

No escaping closures, no function pointers, no dynamic dispatch. A function value is a compile-time entity. It can be passed down to a call but never stored or returned. Dispatch on a kind of value goes through an enum's tag. A plugin system, a table of callbacks registered at run time, or a virtual-function hierarchy has to be rewritten as an enum (Chapters 12 and 16).

Generics have no bounds. A generic function is checked when it is specialized, not when it is declared, so a mistake in a generic library function shows up as an error at a use (Chapter 16).

Whole-program compilation. There are no precompiled Goose libraries and no shared libraries of Goose code. Every build compiles the whole program from source (Chapter 19).

Threads share nothing. There are no locks or atomics, and no way for two threads to work on one array. Parallel work is divided by sending values through queues (Chapter 20).

A freed pool slot can be reused while something still refers to it. The reference then sees the slot's next value, of the same type. This cannot corrupt memory, but it is a logic error that the compiler does not catch (Chapter 14).

The checker is conservative. When a reference passes through a container or a function, the compiler may lose track of exactly which array it points into, and then assumes the worst. Chapter 24 shows a program that had to make a builder inside a loop for this reason. The workaround is usually to move a declaration, but the error can take some thought to understand.

Bytes, not characters. Strings are arrays of `u8`. The library treats them as ASCII where it needs to know about letters, and UTF-8 text passes through unchanged but is not decoded or checked (Chapter 9).

Limits at the boundaries B.2

- Only fixed-size flat values, references, slices and builders cross to C. Variadic C functions cannot be declared at all (Chapter 22).
- An `export fn` cannot take a builder or have default arguments, and must be called from the thread that called `goose_init`.
- The in-process runner cannot run threads. Use `goose build` for programs that spawn workers.

- `simd fn` has an effect only with clang on x86-64.
- Data with links measured from a named global pool cannot be saved with `to_bytes` (Chapter 21).

Omissions of version 1 B.3

These are not part of the design, but are missing from the current compiler and library:

- There is no move operation for resizable values, and a struct can have at most one resizable field.
- There is no labelled `break`. A `block` with a value, or a nested function with `return`, covers most uses.
- Namespaces have no privacy, no nesting and no re-exports.
- The library has no subprocesses and no networking.

The Goose repository's documentation lists the current state of each of these, and some may have changed since this book was written.

Appendix C: Solutions to the Exercises

Every solution here was compiled and run like the rest of the book's examples. Most exercises have more than one good answer; these are just one way to do each.

Chapter 2: Hello, Goose C.1

1. `for i in 5` counts from 0 to 4, so print `i + 1`:

```
fn main() {
    for i in 5 { print(i + 1); }
    print("liftoff");
}
```

```
1
2
3
4
5
liftoff
```

2. A `u8` counter that starts at 250 and is incremented ten times wraps past 255 back to 0 and ends at 4. The result is the same with and without `--debug`, because unsigned arithmetic wraps by definition in every build; only *signed* overflow is checked in debug builds.

```
fn main() {
    var honks: u8 = 250;
    for i in 10 { honks += 1; }
    print(honks);
}
```

```
4
```

Chapter 3: Values and Types C.2

1. It prints `4 260 4 0`. `a + b` is a `u8` addition, so 260 wraps to 4. `c + b` widens `b` to `i64` and gives 260. `-1 % 5` is Euclidean and gives 4, and `-1 / 5` truncates toward zero and gives 0.

```
fn main() {
    let a: u8 = 250;
    let b: u8 = 10;
    let c: i64 = 250;
    print(a + b, " ", c + b, " ", -1 % 5, " ",
        -1 / 5);
}
```

```
4 260 4 0
```

2. No casts are needed. The `i32` converts to a float implicitly as soon as it meets a float operand, and writing the constants as float literals makes the division happen in floating point. The literals take the type of the destination, `f32`.

```
fn main() {
    let c: i32 = -40;
    let f: f32 = c * 9.0 / 5.0 + 32.0;
    print(f);
}
```

```
-40.0
```

3. Because `%` is Euclidean, `(h - k) % 24` is always a valid hour, even when `h - k` is negative:

```
fn main() {
    let h = 3;
    let k = 50;
    print((h - k) % 24);
}
```

```
1
```

Chapter 4: Control Flow as Expressions C.3

1. Ranges are half-open, so the arm for an A runs to 101. There is no open-ended range pattern, so scores outside 0 to 100 need an arm of their own. The final `_` arm, which is required anyway, is a natural place to catch them.

```
fn grade(score: i64) -> u8[:] {
    match score {
        90..101 => "A",
        80..90 => "B",
        70..80 => "C",
        0..70 => "F",
        _ => "invalid",
    }
}

fn main() {
    for s in [95, 85, 75, 12, 100, 101, -5] {
        print(s, " ", grade(s)); }
}
```

```
95 A
85 B
75 C
12 F
100 A
101 invalid
-5 invalid
```

2.

```
fn main() {
    var n = 0;
    let smallest = loop {
        if n * n > 2026 { break n; }
        n += 1;
    };
    print(smallest);
}
```

```
46
```

3. A square is `#` when its row and column add up to an even number:

```
fn main() {
    for r in 4 {
        print(if r % 2 == 0 { "#" } else { "."
            },
              if r % 2 == 1 { "#" } else { "."
                },
              if r % 2 == 0 { "#" } else { "."
                },
              if r % 2 == 1 { "#" } else { "."
                });
    }
}
```

```
#.#.
.#.#
#.#.
.#.#
```

Chapter 5: Functions C.4

1.

```
fn min_max(a: i64, b: i64, c: i64) -> i64, i64 {
    var lo = a;
    var hi = a;
    if b < lo { lo = b; }
    if c < lo { lo = c; }
    if b > hi { hi = b; }
    if c > hi { hi = c; }
    return lo, hi;
}

fn main() {
    let lo, hi = min_max(7, -2, 4);
    print(lo, " ", hi);
}
```

```
-2 7
```

2. The first argument goes before the dot, and the defaults fill in whatever is left out:

```
fn clamp_weight(w: f64, lo: f64 = 2.0, hi: f64 =
    8.0) -> f64 {
    if w < lo { lo } else if w > hi { hi } else
        { w }
}

fn main() {
    print(9.5.clamp_weight(), " ",
        1.0.clamp_weight(), " ",
        5.0.clamp_weight(6.0, 7.0));
}
```

```
8.0 2.0 6.0
```

3.

```
recursive fn digits(n: i64) -> i64 {
    if n < 10 { 1 } else { 1 + digits(n / 10) }
}

fn main() {
    print(digits(0), " ", digits(7), " ",
        digits(2026), " ", digits(1000000));
}
```

```
1 1 4 7
```

Chapter 6: Where Values Live C.5

1. The array is built directly in the caller's destination. While `squares` runs, its local `out` already is the variable or argument slot that receives the result, so `return out` copies nothing. Here the destination is the argument of `print`, a temporary that lasts until the end of the statement.

```
fn squares(n: i64) -> i64[>..] {
    var out: i64[>..] = [];
    for i in 1..n + 1 { out.push(i * i); }
    return out;
}

fn main() { print(squares(6)); }
```

```
[1, 4, 9, 16, 25, 36]
```

2. `b` is not a copy, it names the same array as `a`, so pushes through either one land in the same place. `c` is a real copy, taken while the array held only `1`.

```
fn main() {
    var a: i64[>..] = [1];
    let b = a;
    var c = copy(a);
    b.push(2);
    c.push(3);
    a.push(4);
    print(a, " ", b, " ", c);
}
```

```
[1, 2, 4] [1, 2, 4] [1, 3]
```

3. Three: one for `a`, one for the copy in `c`, and one for the text `print` builds. `b` is only a reference to `a`, so it needs no storage of its own.

```
fn main() {
    var a: i64[>..] = [1];
    let b = a;
    var c = copy(a);
    b.push(2);
    c.push(3);
    a.push(4);
    print(a, " ", b, " ", c);
}
```

```
data stacks: 2 GB reserved per stack, 1 MB guard
    gap, 32 TB budget: 16376 regions,
    GS_MAX_STACKS 1024
main program: 3 data stacks + 0 global stacks =
    3 regions
thread cap: none, no workers
fn main: 3 own, 3 with callees
```

Chapter 7: The Array Family C.6

1. A `u8[..8]` always takes the same number of bytes (a length byte and room for 8 bytes), whatever name it holds, so every element of the array has the same size and the third one starts at a known offset. A `u8[]` takes as many bytes as

its text, which makes an array of them sequential.

```
fn main() {
    var names: u8[..8][>..] = [];
    names.push("Ada");
    names.push("Grace");
    names.push("Linus");
    print(names[1]);
}
```

```
Grace
```

2. The pops take 10, 9 and then 8, at which point the sum is 27:

```
fn main() {
    var stack: i64[>..<] = [];
    for i in 1..11 { stack.push(i); }
    var sum = 0;
    while sum <= 20 { sum += stack.pop(); }
    print(sum, " ", stack);
}
```

```
27 [1, 2, 3, 4, 5, 6, 7]
```

3. A `i64[..8]` has room for only eight elements, so the ninth push stops the program:

```
fn main() {
    var stack: i64[..8] = [];
    for i in 1..11 { stack.push(i); }
}
```

```
goose runtime error: limited array capacity
    exceeded
```

Chapter 8: References and Slices

C.7

1. `best` is a reference variable that is rebound with `.=` as the loop finds larger elements. `for &x` makes each `x` a reference to the element, not a copy, so `best .= x` points at the array itself. The result is rooted at the caller's array, so it may be returned.

```
fn largest(xs: i64[:]) -> i64& {
    var best .= xs[0];
    for &x in xs { if x > best { best .= x; } }
    return best;
}

fn main() {
    var xs: i64[3] = [3, 9, 4];
    let big .= largest(xs);
    big *= 2;
    print(xs);
}
```

```
[3, 18, 4]
```

2. The returned slice views the caller's argument, so it is rooted at whatever the caller passed in, which outlives the call. `first_three` returned a view of its own local `buf`, which dies when the function returns.

```
fn middle(s: u8[:]) -> u8[:] {
    if s.len < 2 { return s[..0]; }
    return s[1..s.len - 1];
}

fn main() {
    var b: u8[>..] = [];
    b.append("gosling");
    print(middle("goose"), " ", middle(b), " [",
        middle("x"), "]");
}
```

```
oos oslin []
```

3. Either print `first` before the `clear`, or give it bytes of its own. A variable of type `u8[]` is a finished string that owns its bytes, so initializing one from a slice copies the bytes:

```
fn main() {
    var line: u8[>..] = [];
    line.append("honk honk");
    let first: u8[] = line[..4];
    line.clear();
    print(first);
}
```

```
honk
```

Chapter 9: Strings Are Bytes C.8

1. `each_word` hands each word to the block as a slice, and `push` adds a single byte to the builder:

```
import std;

fn initials(full_name: u8[:]) -> u8[>..] {
    var out: u8[>..] = [];
    each_word(full_name) { out.push(it[0]); };
    return out;
}

fn main() { print(initials("ada byron
    lovelace")); }
```

```
abl
```

2.

```
import std;

fn main() {
    var total = 0;
    each_split("Ada,4200;Bea,3900;Cleo,4600",
        ';') {
        let fields = split(it, ',');
        let grams, ok = parse_int(fields[1]);
        if ok { total += grams; }
    };
    print(total);
}
```

```
12700
```

3.

```
struct Egg { day: i32, grams: f32 }

fn format(out: u8[>..]&, e: Egg) { format(out,
    "day ", e.day, ": ", e.grams, " g"); }

fn main() { print([Egg { 3, 61.5 }, Egg { 4,
    72.0 }]); }
```

```
[day 3: 61.5 g, day 4: 72.0 g]
```

Chapter 10: Structs and Inline Data C.9

1.

```
struct Nest { owner: u8[..12], eggs: i32 = 0,
    warm: bool = true }

fn main() {
    let n1 = Nest { owner: "Ada", eggs: 4 };
    let n2 = Nest { "Bea", 2, false };
    let n3 = Nest { owner: "Cleo", .. };
    print(n1, " ", n2, " ", n3);
}
```

```
Nest { "Ada", 4, true } Nest { "Bea", 2, false }
    Nest { "Cleo", 0, true }
```

2. Both take 10 bytes: 8 for the `i64` and one each for the `bool` and the `u8`. Goose packs fields without padding, so their order makes no difference to the size. In C the first layout would take 24 bytes and the second 16.

```
struct A { flag: bool, count: i64, small: u8 }
struct B { count: i64, flag: bool, small: u8 }

fn main() {
    var a: A[>..] = [A { true, 1, 2 }];
    var b: B[>..] = [B { 1, true, 2 }];
    print(bytes_of(a).len, " ",
        bytes_of(b).len);
}
```

```
10 10
```

3. Put `paid` first, before any variable-size field, so that its offset is a constant. It is a fixed-size field, so it can be assigned after the order is built:

```
struct Item  { sku: u8[varint], qty: varint,
    cents: varint }
struct Order { paid: bool, id: varint, customer:
    u8[varint], items: Item[varint] }

fn main() {
    var book: Order[>..] = [];
    book.push(Order { paid: false, id: 1,
        customer: "ada",
                        items: [Item { sku:
                            "OATS", qty: 1, cents:
                            900 }] });
    for o in book { o.paid = true; }
    for o in book { print(o.customer, " paid: ",
        o.paid); }
}
```

```
ada paid: true
```

Chapter 11: Optionals C.10

1. `for &g` binds each element by reference, so returning `g` returns a reference into the caller's array. Comparing a `u8[..12]` field with a slice works because arrays of different kinds compare as slices.

```
struct Goose { name: u8[..12], weight: f32 }

fn find_name(flock: Goose[:], name: u8[:]) ->
    Goose? {
    for &g in flock { if g.name == name { return
        g; } }
    return null;
}

fn main() {
    var flock: Goose[>..] = [Goose { "Ada", 4.5
        }, Goose { "Bea", 5.25 }];
    let g = find_name(flock, "Bea");
    if g { g.name = "Beatrix"; }
    print(flock);
}
```

```
[Goose { "Ada", 4.5 }, Goose { "Beatrix", 5.25
    }]
```

2. Keep an optional and rebind it each time a negative element turns up:

```
fn last_negative(xs: i64[:]) -> i64? {
    var found: i64? = null;
    for &x in xs { if x < 0 { found .= x; } }
    return found;
}

fn main() {
    let xs: i64[5] = [3, -1, 4, -5, 9];
    let ys: i64[2] = [1, 2];
    let n = last_negative(xs);
    if n { print(n); }
    print(last_negative(ys) == null);
}
```

```
-5
true
```

3. An optional is a reference, and `x` is a local of `make` that dies when the function returns, so the reference would dangle. A function that computes a new value should return the value itself, usually with a `bool`:

```
fn make() -> i64, bool {
    let x = 5;
    return x, true;
}

fn main() {
    let v, ok = make();
    print(v, " ", ok);
}
```

```
5 true
```

1. Both versions behave the same. Note the type on `meals`: an array literal with no destination type takes its element type from its first element, which here would be the variant type `Feed.Grain`, and the other variants would not fit.

```
enum Feed { Grain { grams: i32 }, Grass, Bread {
    slices: i32 } }

fn health(f: Feed) -> i64 {
    match f {
        Grain g => if g.grams > 200 { 6 } else {
            8 },
        Grass => 10,
        Bread b => 2,
    }
}

fn health2(f: Feed.Grain) -> i64 { if f.grams >
    200 { 6 } else { 8 } }
fn health2(f: Feed.Grass) -> i64 { 10 }
fn health2(f: Feed.Bread) -> i64 { 2 }

fn main() {
    let meals: Feed[3] = [Feed.Grain { grams:
        150 }, Feed.Grass, Feed.Bread { slices:
        2 }];
    for m in meals { print(health(m), " ",
        health2(m)); }
}
```

```
8 8
10 10
2 2
```

2. In fixed mode every meal takes a tag and four bytes of payload room, 5 bytes in all. In variable mode a `Grass` takes only its one-byte tag.

```
enum Feed { Grain { grams: i32 }, Grass, Bread {
    slices: i32 } }

fn main() {
    var fixed: Feed[>..] = [];
    var packed: Feed..[>..] = [];
    for i in 50 {
        fixed.push(Feed.Grass);
        fixed.push(Feed.Grain { grams: 100 });
        packed.push(Feed.Grass);
        packed.push(Feed.Grain { grams: 100 });
    }
    print(bytes_of(fixed).len, " ",
        bytes_of(packed).len);
}
```

```
500 300
```

3.

```
enum Feed { Grain { grams: i32 }, Grass, Bread {
    slices: i32 } }

fn main() {
    var meals: Feed..[>..] = [];
    meals.push(Feed.Bread { slices: 1 });
    meals.push(Feed.Grass);
    meals.push(Feed.Bread { slices: 3 });
    for f in meals {
        match f { Bread &b => { b.slices *= 2;
            }, _ => {} }
    }
    print(meals);
}
```

```
[Bread { 2 }, Feed.Grass, Bread { 6 }]
```

Chapter 13: Links Without Pointers C.12

1. When a node is not lighter than `w`, nothing in its right subtree can be lighter either, so only the

left side needs visiting. Taking the node as a `Node?` lets the recursion pass the links straight through:

```
struct Node { name: u8[..8], weight: i32, left:
    Node&<u32>?, right: Node&<u32>? }

fn insert(tree: Node[>..]&, name: u8[:], weight:
    i32) {
    if tree.len == 0 { tree.push(Node { name:
        name, weight: weight }); return; }
    var cur .= tree[0];
    loop {
        if weight < cur.weight {
            let l = cur.left;
            if l { cur .= l; continue; }
            cur.left .= tree.push(Node { name:
                name, weight: weight });
            return;
        }
        let r = cur.right;
        if r { cur .= r; continue; }
        cur.right .= tree.push(Node { name:
            name, weight: weight });
        return;
    }
}

recursive fn count_lighter(n: Node?, w: i32) ->
    i64 {
    guard n else { return 0; }
    if n.weight >= w { return
        count_lighter(n.left, w); }
    return 1 + count_lighter(n.left, w) +
        count_lighter(n.right, w);
}

fn main() {
    var tree: Node[>..] = [];
    for p in [45, 52, 38, 41, 60] { insert(tree,
        "g", p as i32); }
    print(count_lighter(tree[0], 50), " ",
        count_lighter(tree[0], 39), " ",
          count_lighter(tree[0], 100));
}
```

```
3 1 5
```

2. Keep a reference to the last link and move it along as you push:

```
struct Link { v: i64, next: Link&<u32>? }

fn main() {
    var list: Link[>..] = [Link { v: 1 }];
    var last .= list[0];
    for i in 2..6 {
        let n .= list.push(Link { v: i });
        last.next .= n;
        last .= n;
    }
    var cur: Link? = list[0];
    var out: u8[>..] = [];
    while cur { format(out, cur.v, " "); cur .=
        cur.next; }
    print(out);
}
```

```
1 2 3 4 5
```

3. A self-relative link must point into the same array as the field that holds it. The elements of `by_age` live in `by_age`, but the geese live in `flock`, so the store is rejected:

```
struct Goose { name: u8[..8], mate: Goose&<u16>?
    }

var flock: Goose[>..] = [];
var by_age: (Goose&<u32>)[>..] = [];

fn main() {
    flock.push(Goose { name: "Ada", .. });
    by_age.push(flock[0]);
}
```

```
error: a relative reference must point within
    the same root as its location (§3.9); this
    reference is rooted at flock
```

Chapter 14: Pools for Lifetimes That Don't Nest C.13

1. Unlink the patient exactly as `leave` does, but instead of freeing the slot, link it back in between `head` and the first patient:

```
struct Patient { name: u8[..8], prev: Patient&
    <u32 in queue>, next: Patient&<u32 in queue>
    }

reusable var queue: Patient[>..] = [];
let head = queue.alloc_index(Patient { name: "",
    prev: self, next: self });

fn arrive(name: u8[:]) -> Patient& {
    let tail .= queue[head].prev;
    let p .= queue.alloc_ref(Patient { name:
        name, prev: tail, next: queue[head] });
    tail.next .= p;
    queue[head].prev .= p;
    return p;
}

fn jump_queue(p: Patient&) {
    p.prev.next .= p.next;
    p.next.prev .= p.prev;
    let first .= queue[head].next;
    p.prev .= queue[head];
    p.next .= first;
    first.prev .= p;
    queue[head].next .= p;
}

fn show() {
    var out: u8[>..] = [];
    var cur .= queue[head].next;
    while cur .!= queue[head] { format(out,
        cur.name, " "); cur .= cur.next; }
    print(out);
}

fn main() {
    arrive("Ada");
    arrive("Bea");
    let cleo .= arrive("Cleo");
    jump_queue(cleo);
    show();
}
```

```
Cleo Ada Bea
```

2. When every egg is freed straight away, every allocation after the first reuses slot 0, so the pool ends with one slot. When only every other egg is freed, each freed slot is taken by the next allocation and then kept, so every two allocations add one slot: 500 in all.

```
struct Egg { day: i32 }

fn main() {
    reusable var tray: Egg[>..] = [];
    for i in 1000 { let s = tray.alloc_index(Egg
        { i as i32 }); tray.free(s); }
    print(tray.len);
    reusable var tray2: Egg[>..] = [];
    for i in 1000 {
        let s = tray2.alloc_index(Egg { i as i32
            });
        if i % 2 == 0 { tray2.free(s); }
    }
    print(tray2.len);
}
```

```
1
500
```

3. A pool never gives its memory back and only holds fixed-size values of one type, so a freed slot is always a valid value of the element type, and a reference to it can only ever read such a value. A grow-shrink array does give memory back: after a `pop`, the space above the new top belongs to its data stack again and can be reused for a value of a completely different type. A reference kept across the pop could then read memory at the wrong type. That is why the compiler only allows references into a grow-

shrink array in local variables, where it can check every one of them at each shrink.

Chapter 15: Errors C.14

1.

```
fn safe_divide(a: i64, b: i64) -> i64, bool {
    if b == 0 { return 0, false; }
    return a / b, true;
}

fn main() {
    for geese in [0, 4] {
        let each, ok = safe_divide(12, geese);
        print(each, " ", ok);
    }
}
```

```
0 false
3 true
```

2. Only `parse_goose` changes: one more `guard`. Nothing between it and `load` needs to know that a new kind of error exists.

```
import std;

struct Goose { name: u8[..12], weight: f32 }

fn load(text: u8[:]) -> Goose[>..], u8[] {
    var flock: Goose[>..] = [];
    var line_no = 0;
    each_split(text, '\n') {
        line_no++;
        if trim(it).len > 0 {
            flock.push(parse_goose(it,
            line_no)); }
    };
    return flock, "";
}

fn parse_goose(line: u8[:], line_no: i64) ->
    Goose {
    let fields = split(line, ',');
    guard fields.len == 2 else {
        return [], str("line ", line_no, ":
            expected name,weight") from load;
    }
    let w, ok = parse_flt(trim(fields[1]));
    guard ok else { return [], str("line ",
        line_no, ": bad weight") from load; }
    guard w <= 15.0 else { return [], str("line
        ", line_no, ": impossibly heavy") from
        load; }
    return Goose { name: trim(fields[0]),
        weight: w as f32 };
}

fn main() {
    let flock, err = load("Ada, 4.5\nBig,
        40\n");
    print(flock.len, " ", err);
}
```

```
0 line 2: impossibly heavy
```

3. The error type is `u8[]`, owned text. The loading code becomes a nested function, so that it can fill in the caller's `flock` and be passed to `try` by name:

```
import std;
import exception;

struct Goose { name: u8[..12], weight: f32 }

fn parse_goose(line: u8[:], line_no: i64) ->
    Goose {
    let fields = split(line, ',');
    guard fields.len == 2 else {
        exception::throw(str("line ", line_no,
        ": expected name,weight")); }
    let w, ok = parse_flt(trim(fields[1]));
    guard ok else { exception::throw(str("line
        ", line_no, ": bad weight")); }
    return Goose { name: trim(fields[0]),
        weight: w as f32 };
}

fn report(text: u8[:]) {
    var flock: Goose[>..] = [];
    fn body() {
        var line_no = 0;
        each_split(text, '\n') {
            line_no++;
            if trim(it).len > 0 {
                flock.push(parse_goose(it,
                line_no)); }
        };
        print("loaded ", flock.len, ": ",
            flock);
    }
    exception::try<u8[]>(body) { err =>
        print("error: ", err); };
}

fn main() {
    report("Ada, 4.5\nBea, 5.25\n");
    report("Ada\n");
}
```

```
loaded 2: [Goose { "Ada", 4.5 }, Goose { "Bea",
    5.25 }]
error: line 1: expected name,weight
```

1. The literal bounds adapt to the element type of each instantiation:

```
fn clamp_all<T>(xs: T[:], lo: T, hi: T) {
    for &x in xs { if x < lo { x = lo; } else if
        x > hi { x = hi; } }
}

fn main() {
    var a: i64[4] = [-5, 3, 12, 7];
    var b: f32[3] = [0.5, -1.0, 2.5];
    clamp_all(a, 0, 10);
    clamp_all(b, 0.0, 1.0);
    print(a, " ", b);
}
```

```
[0, 3, 10, 7] [0.5, 0.0, 1.0]
```

2.

```
import std;

fn each_window<T, F>(xs: T[:], n: i64) {
    for i in 0..xs.len - n + 1 { F(xs[i..i +
        n]); }
}

fn main() {
    let xs: i64[5] = [1, 2, 3, 4, 5];
    each_window(xs, 3) { print(sum(it)); };
}
```

```
6
9
12
```

3. The calls chain with method-call syntax. `filter` returns a new array, and `map` is called on it:

```
import std;

fn main() {
    let eggs: i64[5] = [61, 72, 58, 66, 70];
    let kg = eggs.filter() { it > 60 }.map() {
        it as f64 / 1000.0 };
    print(kg);
}
```

```
[0.061, 0.072, 0.066, 0.07]
```

Chapter 17: Recursion C.16

1. Recurse first, then push, so the most significant digit comes out first. The builder belongs to the caller, so it may be passed down:

```
recursive fn binary(n: i64, out: u8[>..]&) {
    if n >= 2 { binary(n / 2, out); }
    out.push(('0' + n % 2) as u8);
}

fn main() {
    for n in [0, 5, 10, 255] {
        var out: u8[>..] = [];
        binary(n, out);
        print(out);
    }
}
```

```
0
101
1010
11111111
```

2. A slice is fixed-size: it is a pointer and a count, sixteen bytes whatever it views. Holding slices across the recursive calls is therefore fine.

```
recursive fn halves(xs: i64[:]) -> i64 {
    if xs.len == 0 { return 0; }
    if xs.len == 1 { return xs[0]; }
    let mid = xs.len / 2;
    return halves(xs[..mid]) +
        halves(xs[mid..]);
}

fn main() {
    let xs: i64[7] = [1, 2, 3, 4, 5, 6, 7];
    print(halves(xs));
}
```

```
28
```

3. The count comes from a second recursive walk of the subtree, which returns an integer, so it needs nothing beyond the native stack. (Having `draw` return its own count would also work, but the count must be printed before the children are drawn.)

```
struct Place { name: u8[..8], inside: i64[..4] }
struct Ctx { places: Place[>..]&, out: u8[>..]&
    }

recursive fn count(c: Ctx&, i: i64) -> i64 {
    var t = 0;
    for k in c.places[i].inside { t += 1 +
        count(c, k); }
    return t;
}

recursive fn draw(c: Ctx&, i: i64, depth: i64) {
    for d in depth { c.out.append("  "); }
    format(c.out, c.places[i].name, " (",
        count(c, i), ")\n");
    for k in c.places[i].inside { draw(c, k,
        depth + 1); }
}

fn main() {
    var places: Place[>..] = [];
    for n in ["farm", "pond", "barn", "Ada",
        "Bea"] { places.push(Place { name: n,
        inside: [] }); }
    places[0].inside.push(1);
    places[0].inside.push(2);
    places[1].inside.push(3);
    places[2].inside.push(4);
    var out: u8[>..] = [];
    var c = Ctx { places: places, out: out };
    draw(c, 0, 0);
    print(out);
}
```

```
farm (4)
  pond (1)
    Ada (0)
  barn (1)
    Bea (0)
```

Chapter 18: A Tour of the Standard Library C.17

1. `update` counts each weight. The block given to `each` can write the enclosing function's locals, so

it can track the best entry as it goes:

```
import dictionary;

fn main() {
    var counts = dictionary<i64, i64> {};
    for w in [61, 72, 61, 58, 72, 61] {
        counts.update(w, 0) { it += 1; }; }
    var best = 0;
    var best_n = 0;
    counts.each() { w, n => if n > best_n { best
        = w; best_n = n; } };
    print(best, " appears ", best_n, " times");
}
```

```
61 appears 3 times
```

2.

```
import std;
import os;
import csv;

fn main() {
    write_file("nests.csv",
        "Ada,3\nBea,1\nCleo,4\n");
    var text: u8[>..] = [];
    if !read_file("nests.csv", text) {
        abort("cannot read nests.csv"); }
    var total = 0;
    csv::each_record(text) { f =>
        let n, ok = parse_int(f[1]);
        if ok { total += n; }
    };
    delete_file("nests.csv");
    print(total, " eggs");
}
```

```
8 eggs
```

3. Give two geese the same weight, in a known order. After the sort, Ada still comes before Cleo, as she did in the input:

```
import std;

struct Goose { name: u8[..8], weight: i64 }

fn main() {
    var flock: Goose[>..] = [Goose { "Ada", 45
        }, Goose { "Bea", 52 },
                                Goose { "Cleo", 45
                                    }, Goose {
                                    "Dot", 38 }];
    stable_sort(flock) { a, b => a.weight <
        b.weight };
    print(flock);
}
```

```
[Goose { "Dot", 38 }, Goose { "Ada", 45 }, Goose
    { "Cleo", 45 }, Goose { "Bea", 52 }]
```

Chapter 19: Programs, Namespaces and Packages C.18

1. Naming the type in a literal needs the prefix, `geese::Goose { … }`. Printing a goose and calling its functions with a dot need none, because the `format` hook and method-style calls are looked up in the namespace of the value's type.

```
namespace geese;

struct Goose { name: u8[..12], weight: f32 }

fn format(out: u8[>..]&, g: Goose) { format(out,
    g.name, " (", g.weight, " kg)"); }

fn heavier(g: Goose, kg: f32) -> Goose { Goose {
    name: g.name, weight: g.weight + kg } }

fn ::main() { print(geese::Goose { "Test", 1.0
    }); }
```

Newer than v2026.3

> This example uses a feature added to Goose after the v2026.3 release, so it is not checked against the compiler the rest of the book is.

```
import geese;

fn main() {
    let g = geese::Goose { "Ada", 4.5 };
    print(g, " ", g.heavier(0.5));
}
```

```
Ada (4.5 kg) Ada (5.0 kg)
```

2. The `fn ::main()` above prints `Test (1.0 kg)` when you run `geese.goose` directly. When the root file imports `geese.goose`, the library's `main` is ignored, and only the root's `main` runs, as the output above shows. (A plain `fn main()` inside the namespace would be `geese::main`, which is not an entry point at all.)

3. Both functions end up in the global namespace with the same signature, so every call is ambiguous:

```
fn load(path: u8[:]) -> i64 { 1 }
```

```
fn load(path: u8[:]) -> i64 { 2 }
```

```
import a;
import b;

fn main() { print(load("x")); }
```

```
error: ambiguous call to load: multiple
    overloads match equally well
```

Putting each library in a namespace of its own avoids the clash.

Chapter 20: Threads That Share Nothing C.19

1. Pass each worker its half of the range as arguments. The two results can arrive in either order, which does not matter for a sum:

```
struct Part { sum: i64 }

thread_fn summer(lo: i64, hi: i64) {
    var s = 0;
    for i in lo..hi { s += i; }
    qput(Part { sum: s });
}

fn main() {
    let a = thread_spawn(summer, 1, 500001);
    let b = thread_spawn(summer, 500001,
        1000001);
    let total = qget<Part>().sum + qget<Part>
        ().sum;
    thread_wait(a);
    thread_wait(b);
    print(total);
}
```

```
500000500000
```

2. The weights travel inside the message, as part of one flat value:

```
struct Batch { weights: f32[] }
struct Avg { value: f32 }

thread_fn averager() {
    let b = qget<Batch>();
    var t: f32 = 0.0;
    for w in b.weights { t += w; }
    qput(Avg { value: t / b.weights.len });
}

fn main() {
    let w = thread_spawn(averager);
    qput(Batch { weights: [4.5, 5.25, 3.0, 4.0,
        4.75] });
    print(qget<Avg>().value);
    thread_wait(w);
}
```

```
4.3
```

3. The worker increments its own copy, taken when it was spawned, so it reports one more than `main` had, and `main`'s global is unchanged:

```
struct Count { value: i64 }

var counter = 0;

thread_fn bumper() {
    counter += 1;
    qput(Count { value: counter });
}

fn main() {
    counter = 41;
    let c = thread_spawn(bumper);
    print("worker says ", qget<Count>().value,
        ", main still has ", counter);
    thread_wait(c);
}
```

```
worker says 42, main still has 41
```

1. The loaded diary is an ordinary grow-only array, so it can be pushed to like any other:

```
import os;

struct Entry { day: varint, note: u8[varint] }

fn main() {
    var diary: Entry[>..] = [];
    diary.push(Entry { day: 1, note: "first egg"
        });
    diary.push(Entry { day: 3, note: "two eggs"
        });
    print(write_file_atomic("diary.bin",
        diary.to_bytes()));
    var disk: u8[>..] = [];
    read_file("diary.bin", disk);
    var back, ok = from_bytes<Entry[>..]>(disk);
    back.push(Entry { day: 4, note: "a goose
        stole the bread" });
    print(ok, " ", back);
    delete_file("diary.bin");
}
```

```
true
true [Entry { 1, "first egg" }, Entry { 3, "two
    eggs" }, Entry { 4, "a goose stole the
    bread" }]
```

2. The first byte, 22, is the image's length. Each word follows as a four-byte little-endian length (`4 0 0 0`, `2 0 0 0`) and then its characters, because the elements are `u8[]`, whose length is a `u32`:

```
fn main() {
    var words: u8[][>..] = [];
    for w in ["honk", "hi", "flap"] {
        words.push(w); }
    var out: u8[>..] = [];
    for b in words.to_bytes() { format(out, b, "
        "); }
    print(out);
}
```

```
22 4 0 0 0 104 111 110 107 2 0 0 0 104 105 4 0 0
    0 102 108 97 112
```

3. An empty file is not a valid image (it does not even have a length), so `from_bytes` rejects it. A file that cannot be read gives `false` too. Since `from_bytes` already returns an array and a `bool`, the function can return its results directly:

```
import os;

struct Goose { name: u8[..8], weight: i32 }

fn load_geese(path: u8[:]) -> Goose[>..], bool {
    var disk: u8[>..] = [];
    if !read_file(path, disk) { return [],
        false; }
    return from_bytes<Goose[>..]>(disk);
}

fn main() {
    write_file("empty.bin", "");
    let e, eok = load_geese("empty.bin");
    let m, mok = load_geese("missing.bin");
    var flock: Goose[>..] = [Goose { "Ada", 45
        }];
    write_file("geese.bin", flock.to_bytes());
    let g, gok = load_geese("geese.bin");
    print(eok, " ", mok, " ", gok, " ", g);
    delete_file("empty.bin");
    delete_file("geese.bin");
}
```

```
false false true [Goose { "Ada", 45 }]
```

Chapter 22: Talking to C C.21

1. C's `fmod` truncates, so its result has the sign of the dividend, and Goose's float `%` is `fmod`. Integer `%` is different: it is Euclidean (Chapter 3), so `-7 % 3` is `2`, the value that pairs with rounding `-7 / 3` *down*, as `floor` does:

```
extern fn floor(x: f64) -> f64;
extern fn fmod(x: f64, y: f64) -> f64;

fn main() {
    print(fmod(-7.0, 3.0), " ", -7.0 % 3.0, " ",
        -7 % 3);
    print(floor(-7.0 / 3.0));
}
```

```
-1.0 -1.0 2
-3.0
```

2. The C file declares the slice struct and writes the count through the pointer:

```
#include <stdint.h>

typedef struct { const uint8_t *data; int64_t
    len; } u8_slice;

void count_vowels(u8_slice s, int64_t *out) {
    int64_t n = 0;
    for (int64_t i = 0; i < s.len; i++) {
        switch (s.data[i] | 0x20) {
            case 'a': case 'e': case 'i': case
                'o': case 'u': n++;
        }
    }
    *out = n;
}
```

```
c_sources = ["vowels.c"];
```

On the Goose side, an `i64&` crosses as an `int64_t *`:

```
extern fn count_vowels(s: const u8[:], out:
    i64&);

fn main() {
    for w in ["goose", "Honk", "aeiou",
        "rhythm"] {
        var n: i64 = 0;
        count_vowels(w, n);
        print(w, ": ", n);
    }
}
```

```
goose: 3
Honk: 1
aeiou: 5
rhythm: 0
```

3. Neither has a plain C shape. A `Goose?` has a presence flag whose layout is the compiler's business, and a `u8[]` stores its bytes inline after a length, so its size is not known to C. Pass the parts instead: a `bool` and a `Goose` (or a `Goose&`) for the optional, and a `const u8[:]` slice for the string, which C sees as `{ data, len }`.

Chapter 23: Performance C.22

1. Whether the version without an `assert` keeps checks depends on its callers. Called only with arrays whose lengths the compiler can see are equal, it keeps none, since facts cross calls. Called with arrays from `fill(n)` (as in the

chapter), it keeps the checks on `a[i]` and `b[i]`. With the `assert`, it keeps none whoever calls it:

```
fn add(a: f32[:], b: f32[:], out: f32[:]) {
    assert(a.len == out.len && b.len ==
        out.len);
    for i in out.len { out[i] = a[i] + b[i]; }
}

fn main() {
    var a: f32[>..] = [1.0, 2.0, 3.0];
    var b: f32[>..] = [10.0, 20.0, 30.0];
    var out: f32[>..] = [0.0, 0.0, 0.0];
    add(a, b, out);
    print(out);
}
```

```
bce-line: main.goose:3: 3 elided, 0 kept
```

```
[11.0, 22.0, 33.0]
```

2. With the `assert`, `checkfree fn add` compiles. Without it, it does not, even though every call in the program passes three arrays of the same length, because a `checkfree` function must prove its checks unnecessary on its own:

```
checkfree fn add(a: f32[:], b: f32[:], out:
    f32[:]) {
    for i in out.len { out[i] = a[i] + b[i]; }
}

fn main() {
    let a: f32[3] = [1.0, 2.0, 3.0];
    let b: f32[3] = [10.0, 20.0, 30.0];
    var out: f32[3] = [0.0, 0.0, 0.0];
    add(a, b, out);
    print(out);
}
```

```
main.goose:2: error: checkfree fn add keeps a
    bounds check: nothing shows i < a.len
main.goose:2: error: checkfree fn add keeps a
    bounds check: nothing shows i < b.len
```

3. Each node keeps the range of its edges as two bound types, and each edge is a `NodeId`. All the nodes are pushed first, so that every edge can be converted to a `NodeId` when it is pushed:

```
struct Node { weight: i32, lo: u32 upto edges,
    hi: u32 upto edges }

var nodes: Node[>..] = [];
type NodeId = u32 in nodes;
var edges: NodeId[>..] = [];

checkfree fn neighbour_weight(n: NodeId) -> i64
    {
    let node = nodes[n];
    var t = 0;
    for k in node.lo..node.hi { t +=
        nodes[edges[k]].weight; }
    return t;
}

fn main() {
    let none = 0 as (u32 upto edges);
    for w in [5, 3, 8, 1] {
        nodes.push(Node { weight: w as i32, lo:
            none, hi: none });
    }
    let links: i64[2][4] = [[1, 2], [2, 3], [3,
        0], [0, 1]];
    for v in nodes.len {
        nodes[v].lo = edges.len as (u32 upto
            edges);
        for t in links[v] { edges.push(t as
            NodeId); }
        nodes[v].hi = edges.len as (u32 upto
            edges);
    }
    for v in nodes.len {
        print("node ", v, ": ",
            neighbour_weight(v as NodeId));
    }
}
```

```
node 0: 11
node 1: 9
node 2: 6
node 3: 8
```

The loop `for k in node.lo..node.hi` gives `k` the index type of `edges`, since both of its bounds are bound types of `edges`, so `edges[k]` needs no

check, and `edges[k]` is a `NodeId`, so `nodes[edges[k]]` needs none either.

Chapter 24: A Word Counter C.23

The solutions for this chapter and the next show only the parts of the program that change. The rest is as printed in the chapter.

1. Only the test in `count_words` changes. `total` now counts the words that were kept:

```
fn count_words(text: const u8[:], counts:
    Counts&) -> i64 {
    var total = 0;
    var i = 0;
    while i < text.len {
        while i < text.len &&
            !is_word_byte(text[i]) { i += 1; }
        let start = i;
        while i < text.len &&
            is_word_byte(text[i]) { i += 1; }
        if i - start > 2 {
            counts.update(text[start..i], 0) {
                it += 1; };
            total += 1;
        }
    }
    return total;
}
```

```
47 words, 21 different
   11  the
    7  goose
    4  and
    3  eggs
    3  farmer
    3  kept
```

2. Every entry of `list` has its count, so a loop after the table finds them. The output below shows the top three:

```
fn main() {
    let a = args();
    guard a.len >= 2 else {
        print("usage: words file [how-many]");
        exit(2);
    }
    var top = 10;
    if a.len >= 3 {
        let n, ok = parse_int(a[2]);
        guard ok && n > 0 else {
            print("not a count: ", a[2]);
            exit(2);
        }
        top = n;
    }
    var text: u8[>..] = [];
    guard read_file(a[1], text) else {
        print("cannot read ", a[1]);
        exit(1);
    }
    to_lower(text);

    var counts = Counts {};
    let total = count_words(text, counts);
    let list = ranked(counts);
    print(total, " words, ", counts.count, "
        different");
    for c in list[..min(top, list.len)] {
        var line: u8[>..] = [];
        format_int(line, c.n, 10, 5, ' ');
        format(line, "  ", c.word);
        print(line);
    }
    var once = 0;
    for c in list { if c.n == 1 { once += 1; } }
    print(once, " words occur once");
}
```

```
53 words, 24 different
   11  the
    7  goose
    4  and
15 words occur once
```

3. Nothing else has to change. `main` passes `text`, which is writable, so it can be passed as either

kind of slice. Inside `count_words`, the slices `text[start..i]` are now writable, and a writable slice converts to the dictionary's read-only key type where it is inserted. The conversion only goes that way: a `const u8[:]` cannot be passed where a `u8[:]` is expected. So the `const` version is the better signature, since it accepts both kinds of text.

```
fn count_words(text: u8[:], counts: Counts&) ->
    i64 {
    var total = 0;
    var i = 0;
    while i < text.len {
        while i < text.len &&
            !is_word_byte(text[i]) { i += 1; }
        let start = i;
        while i < text.len &&
            is_word_byte(text[i]) { i += 1; }
        if i > start {
            counts.update(text[start..i], 0) {
                it += 1; };
            total += 1;
        }
    }
    return total;
}
```

```
53 words, 24 different
   11  the
    7  goose
```

4. The program now reads standard input when there is no file name, or when the name is `-`, the usual convention, which also lets a count be given with standard input: `goose words.goose -- - 3 < geese.txt`.

```
fn main() {
    let a = args();
    var top = 10;
    if a.len >= 3 {
        let n, ok = parse_int(a[2]);
        guard ok && n > 0 else {
            print("not a count: ", a[2]);
            exit(2);
        }
        top = n;
    }
    var text: u8[>..] = [];
    if a.len < 2 || a[1] == "-" {
        read_stdin(text);
    } else {
        guard read_file(a[1], text) else {
            print("cannot read ", a[1]);
            exit(1);
        }
    }
    to_lower(text);

    var counts = Counts {};
    let total = count_words(text, counts);
    let list = ranked(counts);
    print(total, " words, ", counts.count, "
        different");
    for c in list[..min(top, list.len)] {
        var line: u8[>..] = [];
        format_int(line, c.n, 10, 5, ' ');
        format(line, "  ", c.word);
        print(line);
    }
}
```

```
53 words, 24 different
   11  the
    7  goose
    4  and
```

Chapter 25: A Calculator C.24

1. `%` joins `/` in `product`, sharing its check for zero. The version below uses Goose's float `%`, which is C's `fmod`: the result has the sign of the dividend,

so `-7 % 3` is `-1`. A calculator for people rather than programmers might prefer the Euclidean result, `2`, which `v - d * floor(v / d)` gives for a positive `d`. Either is defensible, as long as the choice is documented.

```
7 % 3
-7 % 3
7.5 % 2
2 * 7 % 4
1 % 0
```

```
    recursive fn sum() -> f64 {
        var v = product();
        loop {
            if eat('+') { v += product(); }
            else if eat('-') { v -= product(); }
            else { return v; }
        }
    }
    fn product() -> f64 {
        var v = power();
        loop {
            if eat('*') { v *= power(); }
            else if eat('/') || eat('%') {
                let op = src[pos - 1];
                let at = here();
                let d = power();
                if d == 0.0 {
                    return fail(at, "division by
                        zero");
                }
                if op == '/' { v /= d; } else {
                    v = v % d; }
            }
            else { return v; }
        }
    }
```

```
1.0
-1.0
1.5
2.0
error: column 5: division by zero
```

2. A call gets a function of its own, `call`, which reads one argument or two. `group` is still used for plain parentheses, so `(1, 2)` remains an error. `call` is part of the recursion cycle through `sum`, and it keeps the depth count, like `group`:

```
max(2, 3) * min(4, 5)
max(1, -max(2, 3))
sqrt(4, 9)
max(1, 2
```

```
    fn atom() -> f64 {
        let c = peek();
        let at = pos;                          //
            after the spaces
        if eat('(') { return group(at); }
        if is_digit(c) || c == '.' { return
            number(); }
        if c == 0 { return fail(at, "unexpected
            end of line"); }
        if !is_letter(c) {
            return fail(at, str("unexpected ",
                src[at..at + 1]));
        }
        let n = word();
        if eat('(') { return call(at, n); }
        let v = vars.get(n);
        if v { return v; }
        return fail(at, str("unknown name ",
            n));
    }
    // After the '(' of f(: one argument or two,
        then ')'.
    fn call(at: i64, f: u8[:]) -> f64 {
```

```
12.0
1.0
error: column 1: no function of two arguments
    called sqrt
error: column 9: expected )
```

3. `main` stores each value it prints. A line that fails leaves `ans` as it was, which is what a user

would expect:

```
6 * 7
ans / 2
ans + ans
let x = 1 / 0
ans
```

```
fn main() {
    var vars = Vars {};
    vars.insert("pi", PI);
    var line: u8[>..] = [];
    while read_line(line) {
        if trim(line).len > 0 {
            let v, err = evaluate(line, vars);
            if err.len > 0 {
                print("error: ", err);
            } else {
                print(v);
                vars.insert("ans", v);
            }
        }
        line.clear();
    }
}
```

```
42.0
21.0
42.0
error: column 13: division by zero
42.0
```

4. The column becomes a result of its own instead of part of the text, so that `main` can use it. `fail` returns the position, and `main` prints the line, then a `^` under that position:

```
1 / (2 - 2)
2 * (3 + 4
wings + 1
```

```
// One line of input: `let name = expr`, or an
    expression. Returns
// the value, -1 and "", or 0.0, the position of
    the error and
// what went wrong.
fn evaluate(src: const u8[:], vars: Vars&) ->
    f64, i64, u8[] {
    fn fail(at: i64, what: u8[:]) -> f64 {
        return 0.0, at, str(what) from evaluate;
    }
    return statement(), -1, "";
fn main() {
    var vars = Vars {};
    vars.insert("pi", PI);
    var line: u8[>..] = [];
    while read_line(line) {
        if trim(line).len > 0 {
            let v, at, err = evaluate(line,
                vars);
            if err.len == 0 { print(v); }
            else {
                var mark: u8[>..] = [];
                for k in at { mark.push(' '); }
                mark.push('^');
                print(line, "\n", mark, " ",
                    err);
            }
        }
        line.clear();
    }
}
```

```
1 / (2 - 2)
    ^ division by zero
2 * (3 + 4
          ^ expected )
wings + 1
^ unknown name wings
```