



The Cromo Guide

The Cromo programming language

The Cromo Guide

Version 0.27.1 · October 2026

cromolang.org

github.com/cromolang/cromo

The text and the examples are part of the sources of Cromo, under the MIT or the Apache 2.0 license.

Every example of this guide was compiled and run with cromolang 0.27.1, and its output checked, when the guide was made.

Contents

	Introduction	8
	Why Cromo	8
	Who this guide is for	8
	How to use this guide	9
	The state of Cromo	9
	The name	10
1	Getting started	11
	1.1 Installation	11
	1.2 Hello, world!	12
	1.3 Building an executable	13
	1.4 A project	13
	1.5 Formatting	14
	1.6 Visual Studio Code	14
	1.7 Summary	14
2	A first program: counting words	16
	2.1 The program	16
	2.2 How it works	17
	2.3 Summary	20
3	Common concepts	21
	3.1 Variables	21
	3.2 Types	22
	3.3 Operators	25
	3.4 Functions	26
	3.5 Comments	26
	3.6 Statements and lines	26
	3.7 Conditions	27
	3.8 Loops	27
	3.9 switch	28
	3.10 Summary	29

4	Ownership	30
4.1	Every value has one owner	30
4.2	Simple values are copied	31
4.3	Values that own resources are moved	32
4.4	Passing values to functions	34
4.5	References live only during a call	37
4.6	Exclusivity	37
4.7	const	37
4.8	Summary	38
5	Structs	39
5.1	Defining a struct	39
5.2	Constructors	40
5.3	Methods	41
5.4	Destructors	43
5.5	Printing a struct	44
5.6	Structs inside structs	44
5.7	Summary	45
6	Enums and switch	46
6.1	Cases without data	46
6.2	Cases with data	47
6.3	switch covers every case	48
6.4	Modifying the data of a case	49
6.5	Methods, and generic enums	50
6.6	Summary	51
7	Lists, maps and text	53
7.1	Lists	53
7.2	Fixed-size arrays	54
7.3	Maps	55
7.4	Strings	56
7.5	Formatting	57
7.6	Summary	58

8	Errors	59
8.1	Throwing and catching exceptions	59
8.2	Exception types	60
8.3	finally	61
8.4	Exceptions and destructors	62
8.5	Throwing again	63
8.6	Uncaught exceptions	63
8.7	Panics	64
8.8	Choosing between them	65
8.9	Summary	65
9	Interfaces and generics	66
9.1	Interfaces	66
9.2	Generics	68
9.3	Generic structs	70
9.4	Summary	71
10	Anonymous functions	72
10.1	Writing a lambda	72
10.2	Captures	73
10.3	Lambdas as values	74
10.4	const func	75
10.5	Summary	75
11	Modules	76
11.1	A file is a module	76
11.2	Visibility	77
11.3	Using the names	78
11.4	A folder is a module	78
11.5	The standard library	79
11.6	Code for each system	79
11.7	Packages	80
11.8	Summary	80
12	Tests	81
12.1	Writing a test	81
12.2	Running the tests	82
12.3	assert	83
12.4	Testing errors	83
12.5	Summary	84

13	Threads, tasks and channels	85
13.1	Threads	85
13.2	Mutex	86
13.3	Tasks	87
13.4	Channels	88
13.5	Atomics	89
13.6	Summary	89
14	Files, processes and time	90
14.1	Files	90
14.2	The environment	92
14.3	Other programs	92
14.4	Time	93
14.5	Summary	93
15	Networking and HTTP	95
15.1	TCP	95
15.2	TLS	96
15.3	An HTTP server	97
15.4	An HTTP client	98
15.5	WebSocket	99
15.6	Cryptography and compression	100
15.7	Summary	101
16	JSON	102
16.1	From structs to JSON and back	102
16.2	Names and fields to skip	103
16.3	Enums	104
16.4	Values of any shape	104
16.5	Errors	105
16.6	Summary	106
17	Calling C	107
17.1	Declaring C functions	107
17.2	Types at the boundary	108
17.3	Wrapping a resource	108
17.4	The unsafe dialect	109
17.5	Summary	109

18	Final project: a web service	111
18.1	The data	111
18.2	The store	112
18.3	The handler	113
18.4	The whole program	114
18.5	Making it a real service	115
18.6	Where to go next	116
A	Keywords and operators	117
A.1	Keywords	117
A.2	Built-in types	118
A.3	Operators	119
A.4	Literals	120
A.5	Attributes	120
B	The cromu command	121
B.1	Commands	121
B.2	Options of build and run	122
B.3	Debug and release	122
B.4	Exit codes	122
B.5	Files	123
C	Cromu for C++, Go and Rust programmers	124
C.1	From C++	124
C.2	From Go	125
C.3	From Rust	126
C.4	In short	127

Introduction

Welcome to *The Cromo Guide*, an introduction to the Cromo programming language.

Cromo is a general-purpose, compiled language. It has the syntax of C and C++, no pointers and no garbage collector. Every value has exactly one owner, and the resources a value holds, such as memory, files and connections, are released by its destructor when the owner goes out of scope. The compiler checks ownership with a small set of rules, and LLVM turns the program into fast native code.

Why Cromo

Cromo brings together ideas that have proven themselves in other languages:

- the **syntax**, the constructors and the **destructors** of C and C++, so that a C or C++ programmer can read Cromo code from the first day;
- **ownership checked by the compiler**, as in Rust, with simpler rules: references exist only as parameters, so no lifetime ever needs to be written;
- **lightweight tasks and channels**, as in Go: thousands of tasks wait for the network at the same time, with ordinary blocking code and without `async` or `await`;
- **one tool** that builds, runs, tests and formats, as in Go and Zig;
- the **LLVM** optimizer and linker, which Clang, Rust and Swift also use.

The guiding criterion of the design is simplicity: when two rules give the same safety, Cromo chooses the one that is easier to explain to a C or C++ programmer.

Who this guide is for

This guide assumes that you have written programs in at least one other language. It does not assume C or C++: when Cromo follows them, the guide says so, and it explains everything that a C++ programmer would take for granted.

If you already know C++, Go or Rust, [Appendix C](#) summarizes the differences, and you can skim the first chapters.

How to use this guide

The chapters are meant to be read in order: each one builds on the previous ones. Chapter 2 is a small but complete program, written before the details are explained; chapters 3 to 12 present the language; chapters 13 to 16 the standard library; chapter 17 shows how to call C, and chapter 18 puts everything together in a web service.

Every program in this guide is compiled and run when the guide is built, and its output is the one shown below it:

```
i32 main() {  
    println("Hello from the guide!")  
    ret 0  
}
```

Hello from the guide!

A block with a red border is a program that does **not** compile, shown on purpose, followed by the error of the compiler:

```
i32 main() {  
    string name = "Cromo"  
    string other = name  
    ret 0  
}
```

error: values of type string are not copied implicitly

The guide is also available as a PDF, in English and in Italian: use the **PDF** button at the top of the page.

The state of Cromo

Cromo is young. The current version is 0.27, and the language can still change before version 1.0. Every design decision is recorded, with its reasons, in the file `docs/decisions.md` of the [repository](#), next to the draft specification

`docs/language.md`.

Cromo runs on Windows x64 and Linux x64, and each version compiles programs for both systems.

The name

Cromo is the Italian word for chromium, element 24 of the periodic table, whose symbol Cr appears in the logo. On the web the language is *cromolang*, as in cromolang.org.

Getting started

In this chapter you install Cromo, write your first program, and meet `cromo`, the command that builds, runs, tests and formats Cromo code.

1.1 Installation

Cromo is installed for your user, without administrator rights, with one line. On Windows, in PowerShell:

```
irm https://cromolang.org/install.ps1 | iex
```

On Linux:

```
curl -sSf https://cromolang.org/install.sh | sh
```

The script downloads the archive of the latest version, checks its SHA-256, and unpacks it in `~/.cromo` (`%USERPROFILE%\cromo` on Windows). It adds `~/.cromo/current/bin` to the `PATH` and, if it finds Visual Studio Code, installs the Cromo extension. Open a new terminal, then check the installation:

```
cromo version
```

```
cromo 0.27.1 (LLVM 23.1.4)
```

That is all. `cromo` contains the compiler and the linker: it needs no C compiler, no Visual Studio and no Windows SDK, and the programs it writes do not depend on the C runtime of the system.

Note: the archive can also be installed by hand: download it from cromolang.org, unpack it anywhere and add its `bin` folder to the `PATH`. `cromo` finds the library in the `lib` folder next to `bin`.

1.2 Hello, world!

Create a folder for your programs and, in it, a file named `hello.cro`:

```
hello.cro
i32 main() {
    println("Hello, world!")
    ret 0
}
```

Then run it:

```
cromo run hello.cro

Hello, world!
```

Let us look at the program:

- `i32 main()` declares the function `main`, where every program starts. The type comes before the name, as in C: `main` returns an `i32`, a 32-bit integer, which becomes the exit code of the program.
- `println` prints a line. Its first argument is a format string, which can contain `{}` for the values that follow it: `println("{} + {} = {}", a, b, a + b)`.
- `ret` returns from a function. `Cromo` writes `ret` where C writes `return`.
- There are no semicolons: a statement ends at the end of its line.

A `main` that returns nothing is also allowed, and the exit code is then 0:

```
void main() {  
    println("Hello again!")  
}
```

```
Hello again!
```

1.3 Building an executable

`cromo run` compiles the program into a temporary folder, runs it and deletes it. To keep the executable, use `cromo build`:

```
cromo build hello.cro          # writes hello.exe on Windows, hello on Linux  
./hello
```

By default `cromo` builds a **debug** executable: it is optimized, but it checks every integer operation for overflow, and it contains the information that a debugger needs. With `--release` integers wrap around on overflow, as in C, and the executable is smaller:

```
cromo build --release hello.cro
```

The executable runs on any machine with the same system, without installing anything. `cromo` can also write an executable for the other system:

```
cromo build --target linux hello.cro    # on Windows: an executable for Linux  
cromo build --target windows hello.cro  # on Linux: hello.exe for Windows
```

1.4 A project

A project is a folder with a file `main.cro`. Without a file name, `cromo build`, `cromo run` and `cromo test` use the `main.cro` of the current folder, and the executable is named after the folder:

```
mkdir greeter  
cd greeter  
# write main.cro  
cromo run
```

The other files of the folder are modules, which `main.cro` imports by name: chapter 11 explains them.

1.5 Formatting

Cromo code has one style, and `cromo fmt` writes it: four spaces of indentation, the opening brace on the same line, one space around operators. There are no options to discuss:

```
cromo fmt           # every .cro file of this folder and its subfolders
cromo fmt hello.cro # one file
cromo fmt --check   # only lists the files that are not formatted
```

1.6 Visual Studio Code

The Cromo extension, which the installation adds to Visual Studio Code, colors the code and uses `cromo lsp`, the language server of Cromo, to show:

- the errors while you write, without compiling by hand;
- the type and the declaration of a name, when the mouse is over it;
- the definition of a name, with F12;
- the members of a value, after a dot;
- the formatting of the file, when it is saved.

Any editor that speaks the Language Server Protocol can use `cromo lsp` in the same way.

1.7 Summary

- `cromo run file.cro` compiles and runs a program; `cromo build` writes the executable.
- A program starts from `main`, which returns its exit code.
- The type comes before the name; `ret` returns; statements end at the end of the line.
- `cromo fmt` formats the code, and the extension for Visual Studio Code shows the errors while you write.

In the next chapter you will write a complete program, before learning the details.

A first program: counting words

Before the details, a complete program: it reads a text file and prints the words it uses most often. It shows functions, loops, strings, lists, maps and structs at work; the next chapters explain each of them.

2.1 The program

Here is the text we will count:

```
poem.txt
The fog comes on little cat feet.
It sits looking over harbor and city
on silent haunches and then moves on.
The fog, the cat, the city: on and on.
```

And here is the heart of the program, `words.cro`: its `main`. The functions that it calls stand in place of `// ...`, and the rest of the chapter shows them one at a time; on the site, *Show all* opens the whole file.

```
words.cro
import std.fs

// ...

i32 main(list<string> args) {
    if (args.len() < 2) {
        eprintln("usage: words <file>")
        ret 1
    }
}
```

```

}
string text = fs.readText(args[1])

map<string, i64> seen
for (string word in words(text)) {
    seen[word] += 1
}

list<Count> counts
for (string word, i64 times in seen) {
    Count c
    c.word = word.copy()
    c.times = times
    counts.add(move c)
}
sort(ref counts)

for (i64 i = 0; i < counts.len() && i < 5; i++) {
    println("{:>3}  {}", counts[i].times, counts[i].word)
}
ret 0
}

```

Run it with the name of the file after `--` :

```
cromo run words.cro -- poem.txt
```

```

5  on
4  the
3  and
2  cat
2  city

```

2.2 How it works

Importing a module

```
import std.fs
```

The standard library is made of a core, which every program sees, and of modules, which a program imports. `std.fs` works with files: `fs.readText(path)` returns the whole content of a file as a `string`. Lists, maps and strings are in the core and need no import.

Functions and types

```
bool isLetter(u8 c) {  
    ret (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || c >= 128  
}
```

A function is written as in C: the type of the result, the name, and the parameters, each with its type before its name. `u8` is an unsigned byte; `'a'` is a character, which in Cromo is a byte. A `string` is text in UTF-8, so a letter such as `è` is made of two bytes above 127, which the program counts as letters.

Loops

```
for (i64 i = 0; i <= text.len(); i++) {
```

Cromo has one loop, `for`, with the forms of C and some more: here the classic one, with an `i64` that goes from 0 to the length of the text. `text.len()` is the length in bytes and `text[i]` is the byte at position `i`. Lengths and positions are always `i64`. An index out of the text stops the program with an error, instead of reading memory that is not there.

Lists

```
list<string> found  
found.add(text.slice(start, i).lower())  
ret found
```

`list<string>` is a list of strings that grows as needed. Declared without a value, it is empty: in Cromo every variable has a value, and the default of a list is the empty list. `ret found` gives the list to the caller without copying it.

Structs

```
struct Count {  
    string word  
    i64 times  
}
```

A struct groups values, as in C. `Count c` declares one with its default value, an empty `word` and `times` equal to 0; the fields are then assigned one by one.

Ownership: `ref`, `move` and `copy()`

Three words of the program deserve attention, because they are the heart of Cromo:

- `sort(ref counts)`: the function **modifies** the list, and the call says so with `ref`, as the declaration `void sort(ref list<Count> counts)` does.
- `counts.add(move c)`: the struct **moves** into the list. After the `move`, `c` can no longer be used: its content now belongs to the list.
- `c.word = word.copy()`: a string is never copied without saying so. `word` belongs to the map, so the program asks for a copy.

Without `ref`, a parameter is read-only, as `text` in `words(string text)`: the function can read the string, but not change it, and the string is not copied. Chapter 4 explains these rules, which are the core of the language.

Maps

```
map<string, i64> seen
seen[word] += 1
for (string word, i64 times in seen) {
```

A `map` associates keys with values, as a hash table. `seen[word]` reads or writes the value of a key; `seen[word] += 1` adds a missing key with the value 0 first, as `operator[]` does in C++. A `for` with two variables visits every key with its value, in no particular order: that is why the program sorts the counts.

Printing

```
println("{:>3} {}", counts[i].times, counts[i].word)
```

`{:>3}` prints a value aligned to the right in three characters. The formats are those of `std::format` in C++ and of Python: chapter 7 lists them.

Errors

If the file does not exist, `fs.readText` throws an exception. Nobody catches it, so the program stops and prints what happened, with the calls that led there (shortened here):

```
cromo run words.cro -- missing.txt
```

```
uncaught exception: cannot read 'missing.txt': no such file or folder  
  at std.fs.readText (cromo:std/fs.cro:34)  
  at main (words.cro:52)
```

Chapter 8 shows how to catch exceptions with `try` and `catch`.

2.3 Summary

In a few lines the program used most of what you will meet in this guide: functions, the `for` loop, strings, lists, maps, a struct, and the three ways of passing a value, `read`, `ref` and `move`. Now let us look at the language from the beginning.

Common concepts

This chapter presents what every language has: variables, types, functions, comments and control flow. A C or C++ programmer will recognize most of it; the differences are few, and they matter.

3.1 Variables

A variable is declared with its type before its name, always:

```
i32 main() {  
    i32 apples = 5  
    i64 total = apples + 10  
    string name = "Cromo"  
    bool ready = true  
    println("{} {} {} {}", apples, total, name, ready)  
    ret 0  
}
```

```
5 15 Cromo true
```

There is no type inference: the type is written even when the value would say it. The code is longer by a few characters and easier to read, because the type of every variable is in front of you.

Every variable has a value

A variable declared without a value takes the **default value** of its type: 0 for numbers, `false` for `bool`, the empty string and the empty list.

```
i32 main() {
    i32 count
    f64 average
    string label
    list<i32> numbers
    println("{} {} {} {}", count, average, label, numbers.len())
    ret 0
}
```

[0] [0] [] [0]

There are no uninitialized variables in Cromo, and so no undefined behavior that comes from them.

Constants

Variables can change. `const` makes one immutable:

```
i32 main() {
    const f64 PI = 3.14159
    PI = 3.0
    ret 0
}
```

error: 'PI' is const

A `const` must be given its value where it is declared.

3.2 Types

Integers

Type	Size	Range
<code>i8</code> , <code>u8</code>	8 bits	-128 to 127, 0 to 255
<code>i16</code> , <code>u16</code>	16 bits	-32 768 to 32 767, 0 to 65 535
<code>i32</code> , <code>u32</code>	32 bits	about ±2.1 billion, 0 to 4.2 billion
<code>i64</code> , <code>u64</code>	64 bits	about ±9.2 × 10 ¹⁸ , 0 to 1.8 × 10 ¹⁹

Every type has the same size on every system: there is no `int` whose size depends on the machine, and no `size_t`. Lengths and positions are `i64`.

Integer literals can be written in decimal, hexadecimal and binary, with `_` to separate the digits: `1_000_000`, `0xFF`, `0b1010`.

Overflow

An integer operation whose result does not fit in its type is an error. In a debug build, the default, the program stops:

```
overflow.cro
i32 main() {
    i8 small = 120
    for (i32 i = 0; i < 10; i++) {
        small += 1
    }
    println!("{}", small)
    ret 0
}

panic: integer overflow at overflow.cro:4:9
       at main (overflow.cro:4)
```

A *panic* stops the program, prints where it happened and the calls that led there, and gives the exit code 101. In a release build, with `--release`, integers wrap around instead, as in C: the check costs a little, and a tested program can do without it. Division by zero is always a panic.

Floating point, booleans and characters

- `f32` and `f64` are the floating-point numbers of IEEE 754: `3.14`, `1e-9`.
- `bool` is `true` or `false`.
- `char` is a byte, a synonym of `u8`. A character literal is written between single quotes and must be ASCII: `'a'`, `'\n'`, `'\x41'`. Printed with `{}`, a `char` is a number.

Strings

`string` is text in UTF-8. Its length is in bytes, and `s[i]` is a byte:

```
i32 main() {
    string city = "Torino"
    string greeting = "Hello, " + city + "!"
    println("{} has {} bytes, the first is {}", greeting, greeting.len(),
greeting[0])
    ret 0
}
```

```
Hello, Torino! has 14 bytes, the first is 72
```

Chapter 7 shows the functions of text: searching, splitting, replacing.

Conversions

A number is converted implicitly only when no information can be lost: an `i32` becomes an `i64` or an `f64` by itself. In every other case the conversion is written as a call with the name of the type:

```
i32 main() {
    i32 x = 7
    i64 big = x
    f64 exact = f64(x) / 2.0
    i32 back = i32(exact)
    i32 wide = 300
    u8 byte = u8(wide)
    println("{} {} {} {}", big, exact, back, byte)

    i32 parsed = i32("42") + 1
    println!("{}", parsed)
    ret 0
}
```

```
7 3.5 3 44
43
```

`i32(exact)` drops the fraction; `u8(wide)` keeps the lowest 8 bits. `i32("42")` reads a number from a text, and throws an exception if the text is not a number.

Literals take the type of their context

A literal has no type of its own: it takes the one that its place requires. In `i64 total = apples + 10`, `10` is an `i32` like `apples`. Where nothing says the type, the compiler asks for it:

```
i32 main() {
    println!("{}", 10 / 3)
    ret 0
}
```

error: the type of this expression cannot be deduced

Writing `i32(10) / 3`, or using a variable, solves it.

3.3 Operators

The operators are those of C, with one difference in precedence: the bitwise operators bind more than the comparisons, so `flags & MASK == 0` means `(flags & MASK) == 0`, as one would expect. From the strongest:

Operators	Meaning
<code>*</code> <code>/</code> <code>%</code>	multiplication, division, remainder
<code>+</code> <code>-</code>	addition, subtraction
<code><<</code> <code>>></code>	shifts
<code>&</code>	bitwise and
<code>^</code>	bitwise exclusive or
<code> </code>	bitwise or
<code>==</code> <code>!=</code> <code><</code> <code><=</code> <code>></code> <code>>=</code>	comparisons
<code>&&</code>	logical and
<code> </code>	logical or

`!` negates a `bool` and inverts the bits of an integer. `i++` and `i--` exist, but only as statements on their own, never inside an expression: `a[i++]` is not Croma.

3.4 Functions

```
i64 square(i64 x) {  
    ret x * x  
}  
  
void greet(string name, i32 times) {  
    for (i32 i = 0; i < times; i++) {  
        println("Hello, {}!", name)  
    }  
}  
  
i32 main() {  
    println!("{}", square(12))  
    greet("Ada", 2)  
    ret 0  
}
```

```
144  
Hello, Ada!  
Hello, Ada!
```

A function that returns a value must reach a `ret` on every path, or the program does not compile. A `void` function can end without `ret`. The order of the functions in a file does not matter: `main` can come first and call functions written after it.

A function has one result. When it may have nothing to give, it returns a conventional value, throws an exception, or returns a `bool` and writes the result in a parameter passed with `ref`, which chapter 4 explains. There is no `null`.

3.5 Comments

```
// a comment until the end of the line  
/* a comment  
   on several lines */
```

3.6 Statements and lines

A statement ends at the end of its line, without a semicolon. It continues on the next line when the line ends with an operator, a comma or an open parenthesis:

```
i32 main() {
    i64 total = 1 +
        2 +
        3
    println!("{}",
        total)
    ret 0
}
```

6

There is one statement for each line.

3.7 Conditions

```
i32 main() {
    i32 temperature = 23
    if (temperature > 30) {
        println("hot")
    } else if (temperature < 10) {
        println("cold")
    } else {
        println("mild")
    }
    ret 0
}
```

mild

The braces are always required, and the condition must be a `bool`: `if (count)` is an error, `if (count != 0)` is correct.

3.8 Loops

Cromo has one loop, `for`, in four forms:

```
i32 main() {
    // The classic form of C.
    for (i32 i = 0; i < 3; i++) {
        print("{} ", i)
    }
    println("")
}
```

```

// Only a condition, where C has while.
i32 n = 1
for (n < 100) {
    n *= 3
}
println!("{}", n)

// Without anything, until a break.
for {
    n -= 50
    if (n < 0) {
        break
    }
}
println!("{}", n)

// Over the elements of a list.
list<string> names = ["Ada", "Alan", "Grace"]
for (string name in names) {
    if (name == "Alan") {
        continue
    }
    println!("{}", name)
}
ret 0
}

```

```

0 1 2
243
-7
Ada
Grace

```

`break` leaves the loop and `continue` goes to the next round. There is no `while`, no `do`, and no `goto`.

3.9 switch

`switch` chooses among integers or strings. There is no fall-through: each `case` has its own block, and a `case` can list several values. `default` takes the values not listed:

```

string describe(i32 status) {
    switch (status) {
        case 200 {
            ret "ok"
        }
        case 301, 302 {
            ret "moved"
        }
        case 404 {
            ret "not found"
        }
        default {
            ret "other"
        }
    }
}

i32 main() {
    println("{}, {}, {}", describe(200), describe(302), describe(500))
    ret 0
}

```

ok, moved, other

`switch` is most useful with enums, as chapter 6 shows: there the compiler checks that every case is handled.

3.10 Summary

- The type comes before the name; every variable has a value; `const` makes it immutable.
- Integers have a fixed size; overflow and division by zero stop the program.
- Conversions that can lose information are written: `i32(x)`.
- Functions return with `ret`; there is no `null`.
- `if` wants braces and a `bool`; `for` is the only loop; `switch` has no fall-through.

Ownership

Ownership is how Cromo manages memory and every other resource without a garbage collector and without pointers. The rules are few, the compiler checks them, and once learned they make the cost of every line visible.

4.1 Every value has one owner

A value belongs to the variable, the field or the element of a list that holds it: its **owner**. When the owner ends, the value ends with it. For a local variable, that happens at the end of the block where it is declared, in the reverse order of the declarations.

A struct can say what to do when it ends, with a **destructor**: a method named `~` followed by the name of the struct. The program below uses one to show when each value ends:

```
struct Log {
    string name

    Log(move string label) {
        name = move label
        println("open {}", name)
    }

    ~Log() {
        println("close {}", name)
    }
}
```

```

void inner() {
    Log b = Log("b")
    println("inner")
}

i32 main() {
    Log a = Log("a")
    inner()
    Log d = Log("d")
    println("end of main")
    ret 0
}

```

```

open a
open b
inner
close b
open d
end of main
close d
close a

```

`b` ends when `inner` returns; `d` and `a` at the end of `main`, the last declared first. This is RAIL, as in C++: a struct that owns a file closes it in its destructor, one that owns a connection closes the connection, and the code that uses them cannot forget it, even when an exception goes through. A `string` and a `list` free their memory in the same way.

4.2 Simple values are copied

Numbers, `bool` and the structs made only of them are **simple** types: they are copied, as in C.

```

struct Point {
    f64 x
    f64 y
}

i32 main() {
    Point p
    p.x = 1.5
    Point q = p
    q.x = 9.0
    println("{} {}", p.x, q.x)
    ret 0
}

```

1.5 9

4.3 Values that own resources are moved

A `string`, a `list`, a `map` and the structs that contain them, or that have a destructor, own something that a copy would have to duplicate. Cromo never duplicates it without saying so. Assigning one to another variable does not compile:

```

i32 main() {
    string first = "hello"
    string second = first
    ret 0
}

```

error: values of type string are not copied implicitly

There are two ways to say what you mean. With `move`, the value passes to the new owner, without copying anything, and the old variable can no longer be used:

```

i32 main() {
    string first = "hello"
    string second = move first
    println!("{}", first)
    ret 0
}

```

error: 'first' has been moved and can no longer be used

With `copy()`, the value is duplicated and both variables stay usable:

```
i32 main() {  
    string first = "hello"  
    string second = first.copy()  
    second += "!"  
    println("{} {}", first, second)  
    ret 0  
}
```

```
hello hello!
```

A struct has its `copy()` generated by the compiler, when all its fields can be copied and it has no destructor of its own.

The compiler follows every path of the function: a variable moved only in one branch of an `if` cannot be used after the `if` either.

Temporaries and returned values

A value without a name, as the result of a call or a literal, is moved without writing `move`. So is a local variable returned with `ret`:

```
list<string> names() {  
    list<string> result = ["Ada", "Grace"]  
    ret result  
}  
  
i32 main() {  
    list<string> all = names()  
    all.add("Alan")  
    println("{} ", all.len())  
    ret 0  
}
```

```
3
```

Nothing is copied here: the list built in `names` becomes `all`.

Moving a field

Moving a field, or an element of a list, leaves the default value in its place, so the struct stays usable:

```

struct Message {
    string text
    i32 priority
}

i32 main() {
    Message m
    m.text = "urgent"
    string taken = move m.text
    println("[{}] [{}]", taken, m.text)
    ret 0
}

```

```
[urgent] []
```

4.4 Passing values to functions

A parameter is passed in one of three ways, and **the call repeats the word of the declaration**, so that reading a call is enough to know what happens to its arguments:

Mode	Declaration	Call	The function
read	<code>list<i32> xs</code>	<code>f(numbers)</code>	reads the value; the caller keeps it
modify	<code>ref list<i32> xs</code>	<code>f(ref numbers)</code>	modifies the caller's value
take	<code>move list<i32> xs</code>	<code>f(move numbers)</code>	becomes the owner; the caller loses it

Read

A parameter without a word is read-only. The function cannot modify it:

```

void addOne(list<i32> numbers) {
    numbers.add(1)
}

i32 main() {
    list<i32> values
    addOne(values)
    ret 0
}

```

```
error: cannot call 'add', which is not const
parameter 'numbers' cannot be modified: parameters are read-only
```

A read parameter is never copied: a large value is passed by its address, a small one, up to 16 bytes, by copy, because that is faster. The compiler chooses, and it can, because the value cannot change during the call. In C++ terms, a read parameter is a `const T&` that you never have to write.

Modify: `ref`

With `ref`, the function works on the caller's value. The call writes `ref` too:

```
void addOne(ref list<i32> numbers) {
    numbers.add(1)
}

i32 main() {
    list<i32> values
    addOne(ref values)
    addOne(ref values)
    println("{} ", values.len())
    ret 0
}
```

2

Forgetting the `ref` in the call is an error, not a silent copy:

```
void addOne(ref list<i32> numbers) {
    numbers.add(1)
}

i32 main() {
    list<i32> values
    addOne(values)
    ret 0
}
```

```
error: parameter 'numbers' is 'ref': write 'ref' before the argument
```

`ref` is also how a function gives more than one result:

```

bool find(list<string> items, string wanted, ref i64 position) {
    for (i64 i = 0; i < items.len(); i++) {
        if (items[i] == wanted) {
            position = i
            ret true
        }
    }
    ret false
}

i32 main() {
    list<string> planets = ["Mercury", "Venus", "Earth"]
    i64 at
    if (find(planets, "Earth", ref at)) {
        println("found at {}", at)
    }
    ret 0
}

```

found at 2

Take: **move**

A **move** parameter receives the ownership of the value. The function can keep it, for example in a list, without copying it:

```

struct Library {
    list<string> titles

    void add(move string title) {
        titles.add(move title)
    }
}

i32 main() {
    Library library
    string book = "The C Programming Language"
    library.add(move book)
    library.add("Structure and Interpretation of Computer Programs")
    println("{} books", library.titles.len())
    ret 0
}

```

2 books

4.5 References live only during a call

`ref` exists only for parameters. There are no references in variables, in fields or in returned values: a reference cannot outlive the object it refers to, because it ends with the call. This single rule replaces the lifetimes and the borrow checker of other languages.

When a program needs to point to another object, as in a graph, it keeps an index into a list instead of a pointer.

4.6 Exclusivity

During a call, an object passed with `ref` cannot be reached through any other argument of the same call. Otherwise a function could see a value change under its feet:

```
void swap(ref i32 a, ref i32 b) {  
    i32 t = a  
    a = b  
    b = t  
}  
  
i32 main() {  
    i32 x = 1  
    swap(ref x, ref x)  
    ret 0  
}
```

error: 'x' overlaps 'x' in the same call, and the call can modify both

When the compiler cannot know, as in `swap(ref v[i], ref v[j])`, the program checks it while it runs, and a call with `i` equal to `j` is a panic. The same rule allows the compiler to tell LLVM that two `ref` parameters never overlap, which makes the code faster, as `restrict` in C.

4.7 const

A `const` variable cannot be modified, and `const` is deep: the strings and lists inside a `const` value are `const` too. A method marked `const` does not modify its object, as chapter 5 shows; only `const` methods can be called on a read parameter.

4.8 Summary

- Every value has one owner, and ends with it: destructors release the resources.
- Simple values are copied; the others are moved with `move` or duplicated with `copy()`.
- A parameter is read, modified with `ref`, or taken with `move`, and the call says which.
- References exist only during a call: no lifetimes to write, no dangling pointers.
- Two arguments of a call that modifies them are never the same object.

Structs

A struct groups related values under one name, as in C. In Cromo it can also have a constructor, methods and a destructor, as a class of C++, but there is no inheritance: what different types have in common is expressed with interfaces, in chapter 9.

5.1 Defining a struct

```
struct Rectangle {  
    f64 width  
    f64 height  
}  
  
i32 main() {  
    Rectangle r  
    r.width = 3.0  
    r.height = 2.0  
    println("{} x {}", r.width, r.height)  
    ret 0  
}
```

3 x 2

`Rectangle r` declares a rectangle with its default value: each field has the default of its type, here 0. A field can declare its own initial value:

```

struct Settings {
    string host = "localhost"
    i32 port = 8080
    list<string> paths = ["/", "/health"]
    bool verbose
}

i32 main() {
    Settings s
    println("{}: {} {} {}", s.host, s.port, s.paths.len(), s.verbose)
    ret 0
}

```

```
localhost:8080 2 false
```

The initial value can be any expression, and it is computed every time a default `Settings` is made.

5.2 Constructors

A constructor is written as in C++, with the name of the struct and no type. It assigns the fields, which are used by their name: there is no `this`.

```

struct Rectangle {
    f64 width
    f64 height

    Rectangle(f64 w, f64 h) {
        width = w
        height = h
    }
}

i32 main() {
    Rectangle r = Rectangle(3.0, 2.0)
    println("{} x {}", r.width, r.height)
    ret 0
}

```

```
3 x 2
```

A struct has at most one constructor. A parameter cannot have the name of a field, so that a name inside the struct always means one thing: `width = w`, never `width = width`.

When a constructor cannot do its job, for example because a file does not open, it throws an exception: the object then does not exist, and the fields already made are destroyed.

A struct without a constructor can still be made as a value, with `Rectangle()`: it is its default value. A variable declared without a value always gets the default value, even when the struct has a constructor: the constructor runs only when it is called.

5.3 Methods

Functions written inside the struct are its methods. They see the fields by name and can call the other methods by name:

```
struct Rectangle {
    f64 width
    f64 height

    Rectangle(f64 w, f64 h) {
        width = w
        height = h
    }

    f64 area() const {
        ret width * height
    }

    bool isSquare() const {
        ret width == height
    }

    void scale(f64 factor) {
        width *= factor
        height *= factor
    }

    bool contains(Rectangle other) const {
        ret width >= other.width && height >= other.height
    }
}
```

```
i32 main() {
    Rectangle a = Rectangle(3.0, 2.0)
    Rectangle b = Rectangle(1.0, 1.0)
    println("area {}, square {}", a.area(), b.isSquare())
    b.scale(4.0)
    println("a contains b: {}", a.contains(b))
    ret 0
}
```

```
area 6, square true
a contains b: false
```

const methods

`const` after the parameters, as in C++, says that the method does not modify the object. Only `const` methods can be called on a value that cannot be modified: a read parameter, a `const` variable, the element of a `for` over a list.

```
struct Counter {
    i64 value

    void increment() {
        value++
    }
}

void show(Counter c) {
    c.increment()
}

i32 main() {
    Counter c
    show(c)
    ret 0
}
```

```
error: cannot call 'increment', which is not const
```

Without pointers, `const` is deep: a `const` method cannot modify even the strings and the lists inside the object.

5.4 Destructors

The destructor, `~Name()`, runs when the object ends: at the end of its scope, when it is removed from a list, when the struct that contains it ends. It releases what the object holds. After the destructor, the fields are destroyed by themselves, the last declared first, so a destructor only deals with what the fields cannot release alone: a handle of the system, a temporary file, a message to print.

```
struct Timer {
    string label

    Timer(move string name) {
        label = move name
        println("{}: start", label)
    }

    ~Timer() {
        println("{}: end", label)
    }
}

void work() {
    Timer t = Timer("work")
    println("working")
}

i32 main() {
    work()
    Timer outer = Timer("main")
    ret 0
}
```

```
work: start
working
work: end
main: start
main: end
```

A destructor is never called by hand, and it cannot let an exception out: the compiler checks it.

A struct with a destructor is not a simple type: it is moved, and copied only if it writes its own `copy()` method.

5.5 Printing a struct

`{:?}` prints any value in a form meant for debugging, with the names of the fields. To print a struct with `{}`, give it a method `string text() const`:

```
struct Point {
    i64 x
    i64 y

    string text() const {
        ret format("({}, {})", x, y)
    }
}

i32 main() {
    Point p
    p.x = 1
    p.y = 2
    println("{} {:?}", p, p)
    ret 0
}
```

```
(1, 2) Point { x: 1, y: 2 }
```

`format` works as `println`, but returns the text instead of printing it.

5.6 Structs inside structs

A struct can contain other structs, lists of structs, maps: everything it contains is destroyed with it. It cannot contain itself directly, since it would have an infinite size, but it can through a list:

```
struct Folder {
    string name
    list<Folder> children

    i64 count() const {
        i64 total = 1
        for (Folder child in children) {
            total += child.count()
        }
        ret total
    }
}
```

```

i32 main() {
    Folder root
    root.name = "/"
    Folder docs
    docs.name = "docs"
    docs.children.add(Folder())
    root.children.add(move docs)
    root.children.add(Folder())
    println("{} folders", root.count())
    ret 0
}

```

4 folders

5.7 Summary

- A struct has fields, at most one constructor, methods and a destructor; inside it, the fields are used by name.
- Every field has a default value, or the initial value that it declares.
- `const` methods do not modify the object, and they are the only ones that can be called on a read-only value.
- The destructor releases what the object holds, automatically, at the right moment.
- `text()` prints a struct with `{}`; `{:?}` prints any value for debugging.

Enums and switch

An enum is a type whose value is one of a list of cases. In Cromo a case can carry data, so an enum can say not only *which* case a value is, but also *what* comes with it. Together with `switch`, which the compiler checks against every case, enums are the safest way to represent alternatives.

6.1 Cases without data

```
enum Direction {  
    North  
    East  
    South  
    West  
}  
  
Direction turnRight(Direction d) {  
    switch (d) {  
        case North {  
            ret Direction.East  
        }  
        case East {  
            ret Direction.South  
        }  
        case South {  
            ret Direction.West  
        }  
        case West {  
            ret Direction.North  
        }  
    }  
}
```

```

}

i32 main() {
    Direction d = Direction.North
    d = turnRight(turnRight(d))
    println("{:?} {}", d, d == Direction.South)
    ret 0
}

```

South true

A value is written with the name of the enum, `Direction.North`; inside a `switch` on the enum, the case is enough. The default value of an enum is its first case.

The cases of an enum without data can have a value, which `i32(...)` reads and `Color(n)` turns back into a case:

```

enum Color {
    Red = 1
    Green = 2
    Blue = 4
}

i32 main() {
    Color c = Color.Green
    println("{} {:?}", i32(c), Color(4))
    ret 0
}

```

2 Blue

`Color(3)` would be a panic, since no case has the value 3.

6.2 Cases with data

A case can carry values, written as the parameters of a function:

```

enum Shape {
    Circle(f64 radius)
    Rect(f64 width, f64 height)
    Empty
}

f64 area(Shape shape) {

```

```

switch (shape) {
    case Circle(f64 r) {
        ret 3.14159 * r * r
    }
    case Rect(f64 w, f64 h) {
        ret w * h
    }
    case Empty {
        ret 0.0
    }
}

i32 main() {
    list<Shape> shapes = [Shape.Circle(1.0), Shape.Rect(2.0, 3.0), Shape.Empty]
    for (Shape s in shapes) {
        println("{:>12} {:.2f}", format("{:?}", s), area(s))
    }
    ret 0
}

```

```

Circle(1.0) 3.14
Rect(2.0, 3.0) 6.00
Empty 0.00

```

In `case Circle(f64 r)` the data of the case get a name, read-only and without a copy. A case can ignore its data by leaving out the parentheses.

6.3 switch covers every case

On an enum, `switch` must handle every case, or have a `default`. Forgetting one is a compile error, not a bug found later:

```

enum Shape {
    Circle(f64 radius)
    Rect(f64 width, f64 height)
}

f64 area(Shape s) {
    switch (s) {
        case Circle(f64 r) {
            ret 3.14159 * r * r
        }
    }
}

i32 main() {
    ret 0
}

```

error: 'switch' does not cover Rect of Shape: add the cases, or 'default'

When a case is added to an enum, the compiler lists every `switch` that must learn about it.

A `switch` has no fall-through: each case runs its own block and the `switch` ends. Several cases without data can share a block: `case North, South { ... }`. A `break` inside a case is an error, because in C it would leave the `switch`, while in Cromo it would leave the loop around it.

6.4 Modifying the data of a case

With `ref`, a case gives its data for modification:

```

enum Shape {
    Circle(f64 radius)
    Rect(f64 width, f64 height)
}

void grow(ref Shape s) {
    switch (s) {
        case Circle(ref f64 r) {
            r *= 2.0
        }
        case Rect(ref f64 w, ref f64 h) {
            w *= 2.0
            h *= 2.0
        }
    }
}

```

```

    }
}

i32 main() {
    Shape s = Shape.Circle(1.5)
    grow(ref s)
    println("{:?}", s)
    ret 0
}

```

```
Circle(3.0)
```

6.5 Methods, and generic enums

An enum can have methods, as a struct. Inside a method, `switch` without a value works on the enum itself:

```

enum Token {
    Number(i64 value)
    Word(string text)
    End

    string describe() const {
        switch {
            case Number(i64 n) {
                ret format("the number {}", n)
            }
            case Word(string w) {
                ret format("the word '{}'", w)
            }
            case End {
                ret "the end"
            }
        }
    }
}

i32 main() {
    List<Token> tokens = [Token.Number(42), Token.Word("hello"), Token.End]
    for (Token t in tokens) {
        println!("{}", t.describe())
    }
    ret 0
}

```

```
the number 42
the word 'hello'
the end
```

An enum can also have type parameters, as the generic structs of chapter 9. This one holds either a result or the reason of a failure:

```
enum Outcome<T> {
    Done(T value)
    Failed(string reason)
}

Outcome<i64> half(i64 n) {
    if (n % 2 != 0) {
        ret Outcome<i64>.Failed(format("{} is odd", n))
    }
    ret Outcome<i64>.Done(n / 2)
}

i32 main() {
    list<i64> inputs = [10, 7]
    for (i64 n in inputs) {
        switch (half(n)) {
            case Done(i64 v) {
                println("half of {} is {}", n, v)
            }
            case Failed(string why) {
                println("cannot: {}", why)
            }
        }
    }
    ret 0
}
```

```
half of 10 is 5
cannot: 7 is odd
```

6.6 Summary

- An enum lists cases; a case can carry data, written as parameters.
- `switch` on an enum must cover every case or have `default`, and the compiler checks it.
- `case Circle(f64 r)` names the data of a case; with `ref` they can be modified.

- Enums have methods and type parameters, as structs; the default value is the first case.

Lists, maps and text

The core of the library, which every program sees without importing it, has the containers that almost every program needs: `list`, fixed-size arrays, `map` and `string`. Each one owns its elements and frees them when it ends.

7.1 Lists

`list<T>` is a sequence of values of type `T` that grows as needed, as `std::vector` in C++:

```
i32 main() {
    list<i64> numbers = [5, 3, 8]
    numbers.add(13)
    numbers.insert(0, 1)
    i64 removed = numbers.remove(2)
    println("{:?}", removed {}, length {}, numbers, removed, numbers.len())

    numbers[0] = 10
    for (ref i64 n in numbers) {
        n *= 2
    }
    println("{:?}", numbers)
    ret 0
}
```

```
[1, 5, 8, 13], removed 3, length 4
[20, 10, 16, 26]
```

- `[a, b, c]` makes a list; an empty one is `list<i64>()`, or simply a variable without a value.
- `add`, `insert(i, x)`, `remove(i)`, `clear()` and `reserve(n)` change the list; `len()` gives its length.
- `numbers[i]` is the element itself: it can be read, assigned, passed with `ref` and moved. An index out of the list is a panic.
- `for (ref i64 n in numbers)` visits the elements and can modify them; without `ref` they are read-only.

While a `for` visits a list, the list cannot change its length: adding an element in the loop would move the elements in memory under the loop. The compiler refuses it.

map and filter

`map` makes a new list from each element, and `filter` keeps the elements that pass a test. Both take a function, here an anonymous function, explained in chapter 10:

```
i32 main() {
    list<i64> numbers = [1, 2, 3, 4, 5, 6]
    list<i64> even = numbers.filter([](i64 n) bool { ret n % 2 == 0 })
    list<string> labels = even.map<string>([](i64 n) string { ret format("#{}",
n) })
    println("{:?}", labels)
    ret 0
}
```

```
["#2", "#4", "#6"]
```

7.2 Fixed-size arrays

An array has a size fixed in its type, and lives where its variable is, without memory on the heap:

```

i32 main() {
    i32[4] scores
    scores[1] = 7
    f64[2][3] grid
    grid[1][2] = 0.5
    println("{} has {} elements", scores, scores.len())
    println("{}:", grid)
    ret 0
}

```

```

[0, 7, 0, 0] has 4 elements
[[0.0, 0.0, 0.0], [0.0, 0.0, 0.5]]

```

`f64[2][3]` is two arrays of three. An array is a value: it is copied when its elements are simple, moved otherwise.

7.3 Maps

`map<K, V>` associates keys with values, with a hash table:

```

i32 main() {
    map<string, i32> stock
    stock["apples"] = 10
    stock["pears"] = 4
    stock["apples"] -= 3
    stock["plums"] += 6

    println("{} apples, plums known: {}", stock["apples"],
stock.contains("plums"))
    bool removed = stock.remove("pears")
    println("removed: {}, {} kinds left", removed, stock.len())

    i32 total = 0
    for (string fruit, i32 count in stock) {
        total += count
    }
    println("{} fruits", total)
    ret 0
}

```

```

7 apples, plums known: true
removed: true, 2 kinds left
13 fruits

```

- `m[k] = v`, `m[k] += v` and `m[k]++` add a missing key, with the default value, before changing it, as `operator[]` of C++. Reading a missing key in any other way is a panic: ask first with `contains`.
- `for (K key, V value in m)` visits the map in no particular order; with `ref V value` it can modify the values.
- The keys can be numbers, `bool`, `string`, and the structs that implement the interface `Key`, with `u64 hash() const` and `bool equals(Self other) const`.

7.4 Strings

A `string` is text in UTF-8. It owns its bytes, and is moved or copied as a list:

```
i32 main() {
    string text = " Hello, World "
    string clean = text.trim()
    println("[{}]", clean)
    println("{} {} {}", clean.find("World"), clean.contains("lo"),
clean.startsWith("Hello"))
    println("{} / {}", clean.upper(), clean.replace("World", "Cromo"))
    println("{}:?}", "a,b,c".split(","))
    println("{}{}", join(["x", "y", "z"], " - "))
    println("{}{}", "ab".repeat(3))
    ret 0
}
```

```
[Hello, World]
7 true true
HELLO, WORLD / Hello, Cromo
["a", "b", "", "c"]
x - y - z
ababab
```

The functions of text are methods of `string`: `find`, `findFrom`, `findLast`, `contains`, `startsWith`, `endsWith`, `split`, `trim`, `trimStart`, `trimEnd`, `replace`, `upper`, `lower`, `repeat`, and the function `join(parts, separator)`. Positions are in bytes, and a text that is not found gives -1. `upper` and `lower` change only the letters of ASCII.

Strings and bytes

The length of a string is in bytes, and a letter outside ASCII takes more than one:

```
i32 main() {
    string word = "città"
    list<u8> bytes = word.bytes()
    println("{} bytes, the first three are '{}'", word.len(), word.slice(0, 3))
    string again = string(bytes)
    println("{} {}", again, again == word)
    ret 0
}
```

```
6 bytes, the first three are 'cit'
città true
```

`slice(from, to)` takes the bytes from `from` to `to`, excluded. `string(bytes)` checks that the bytes are valid UTF-8, and throws an exception otherwise.

Strings are compared with `==`, `!=`, `<`, `<=`, `>`, `>=`, byte by byte, and joined with `+` and `+=`.

7.5 Formatting

`println`, `print`, `eprintln` (to the standard error) and `format` (which returns a `string`) share the format of `std::format` in C++ and of Python. Each `{}` takes the next value; inside the braces, after `:`, a specification says how to write it:

```
i32 main() {
    f64 ratio = 2.0 / 3.0
    i64 n = 1234567
    println("[{:3f}] [{:8.2f}] [{:e}]", ratio, ratio, ratio)
    println("[{:<6}] [{:^7}] [{:>6}]", "ab", "mid", "r")
    println("[{:+}] [{:09}] [{:x}] [{:#x}] [{:b}]", n, n, n, n, n)
    println("{0} and {1}, then {0} again", "first", "second")
    println("{ { and } are braces")
    ret 0
}
```

```
[0.667] [    0.67] [6.666667e-01]
[ab    ] [  mid  ] [      r]
[+1234567] [001234567] [12d687] [0x12d687] [100101101011010000111]
first and second, then first again
{ and } are braces
```

Specification	Meaning
< ^ >	align to the left, the center, the right, in the width that follows
a number	the minimum width; with a 0 before it, filled with zeros
.n	digits after the point, or the maximum length of a text
+	the sign also for positive numbers
x X b o	hexadecimal, binary, octal; # adds the prefix
f e g	fixed point, exponent, the shorter of the two
?	the form for debugging, for any value

The format string must be a literal: the compiler checks the number and the types of the values against it.

7.6 Summary

- `list<T>` grows as needed; `[a, b]` makes one; `for` visits it, with `ref` to modify the elements.
- `T[n]` is an array of fixed size, without the heap.
- `map<K, V>` is a hash table; `m[k] += v` adds a missing key; the order is not defined.
- `string` is UTF-8, measured in bytes, with methods for searching, splitting and replacing.
- `format` and `println` use the format of `std::format`, checked by the compiler.

Errors

Cromo separates two kinds of errors:

- **Errors that a program expects**, as a file that does not exist or a connection that drops: they are **exceptions**, which a caller can catch and handle.
- **Programming errors**, as an index out of a list or an integer overflow: they are **panics**, which stop the program, because a program that has broken its own rules cannot know what to do next.

8.1 Throwing and catching exceptions

```
void checkAge(i64 age) {  
    if (age < 0) {  
        throw Error(format("invalid age {}", age))  
    }  
}  
  
i32 main() {  
    try {  
        checkAge(30)  
        println("30 is fine")  
        checkAge(-3)  
        println("not reached")  
    } catch (Exception e) {  
        println("error: {}", e.message())  
    }  
    ret 0  
}
```

```
30 is fine
error: invalid age -3
```

`throw` stops the function and goes back through the callers until a `try` whose `catch` accepts the exception. `Error` is a ready-made exception with a message; `catch (Exception e)` accepts any exception, and `e.message()` gives its text. Exceptions are not declared in the signature of a function.

8.2 Exception types

An exception is a struct that implements the interface `Exception`, which has one method, `string message() const`. A type of its own lets a caller recognize the error and read its fields:

```
struct NotFound : Exception {
    string name

    NotFound(move string what) {
        name = move what
    }

    string message() const {
        ret format("{} was not found", name)
    }
}

i64 position(list<string> items, string wanted) {
    for (i64 i = 0; i < items.len(); i++) {
        if (items[i] == wanted) {
            ret i
        }
    }
    throw NotFound(wanted.copy())
}

i32 main() {
    list<string> names = ["Ada", "Alan"]
    try {
        println!("{}", position(names, "Alan"))
        println!("{}", position(names, "Grace"))
    } catch (NotFound e) {
```

```

        println("missing: {}", e.name)
    } catch (Exception e) {
        println("other error: {}", e.message())
    }
    ret 0
}

```

```

1
missing: Grace

```

The `catch` blocks are tried in order; the first that accepts the type of the exception handles it. The exception of a `catch` is read-only.

8.3 finally

`finally` runs whenever the `try` ends: normally, with `ret`, with `break` or `continue`, or with an exception, caught or not:

```

i32 main() {
    for (i32 i = 0; i < 2; i++) {
        try {
            if (i == 1) {
                throw Error("second round")
            }
            println("round {}", i)
        } catch (Exception e) {
            println("caught: {}", e.message())
        } finally {
            println("end of round {}", i)
        }
    }
    ret 0
}

```

```

round 0
end of round 0
caught: second round
end of round 1

```

8.4 Exceptions and destructors

While an exception goes back through the functions, every object of the scopes it leaves is destroyed, as at the end of a normal block. This is what makes RAII reliable: a resource held by an object is released even when something goes wrong.

```
struct Resource {
    string name

    Resource(move string label) {
        name = move label
        println("acquire {}", name)
    }

    ~Resource() {
        println("release {}", name)
    }
}

void work() {
    Resource a = Resource("a")
    Resource b = Resource("b")
    throw Error("failure in work")
}

i32 main() {
    try {
        work()
    } catch (Exception e) {
        println("caught: {}", e.message())
    }
    ret 0
}
```

```
acquire a
acquire b
release b
release a
caught: failure in work
```

A destructor, and a `finally`, cannot let an exception out: the compiler requires them to catch it inside.

8.5 Throwing again

`throw` alone, inside a `catch`, throws the same exception again, for example after writing it in a log:

```
void load() {
    throw Error("disk full")
}

i32 main() {
    try {
        try {
            load()
        } catch (Exception e) {
            println("load failed, passing it on")
            throw
        }
    } catch (Exception e) {
        println("main: {}", e.message())
    }
    ret 0
}
```

```
load failed, passing it on
main: disk full
```

8.6 Uncaught exceptions

An exception that leaves `main` stops the program with exit code 1, and prints the message and the calls from the `throw` up to `main`:

```
deep.cro

void deep(i64 n) {
    if (n == 0) {
        throw Error("too deep")
    }
    deep(n - 1)
}

i32 main() {
    deep(3)
    ret 0
}
```

```
uncaught exception: too deep
  at deep (deep.cro:3)
  at main (deep.cro:9)
```

The trace is printed also in release builds, with the names, files and lines of the source. The executable does not need its debug information for it: the compiler writes a compact table of the code inside it.

8.7 Panics

A panic is a programming error: an index out of bounds, an overflow, a division by zero, a key missing from a map, a stack overflow. It is not an exception and it cannot be caught: the program stops with exit code 101 and prints the calls that led there.

```
bounds.cro
i64 third(list<i64> values) {
  ret values[2]
}

i32 main() {
  list<i64> short = [1, 2]
  println("{} ", third(short))
  ret 0
}
```

```
panic: index out of bounds: the index is 2 but the length is 2 at
bounds.cro:2:15
  at third (bounds.cro:2)
  at main (bounds.cro:7)
```

`panic("reason")` stops the program on purpose, when it reaches a state that should be impossible.

assert

`assert(condition)` is a panic when its condition is false. The message repeats the condition as written, and for a comparison of numbers the values of its sides:

```
check.cro
```

```
i32 main() {  
    i64 x = 3  
    assert(x * 2 == 7)  
    ret 0  
}
```

```
panic: assertion failed: x * 2 == 7 (x * 2 is 6) at check.cro:3:5  
      at main (check.cro:3)
```

`assert` works in every build, also with `--release`. It is also the tool of the tests of chapter 12.

8.8 Choosing between them

- If the caller can do something about the error, as asking for another file name, throw an **exception**.
- If the error means that the program is wrong, as an index computed badly, it is a **panic**: fix the program.
- If the absence of a result is normal, as a word not found in a text, return a conventional value or a `bool`, as `find` does with `-1`.

8.9 Summary

- `throw`, `try`, `catch` and `finally` work as in C++, Java and C#; exceptions implement `Exception`.
- Destructors run while an exception goes back, so resources are always released.
- An uncaught exception, a panic and a stack overflow print the calls that led there, also in release.
- Panics are for programming errors; `assert` checks what must be true.

Interfaces and generics

Different types often share a behavior: a circle and a square both have an area. In Croma the behavior is described by an **interface**, and code that works with any type that has it is written once, either as a function that takes the interface, decided while the program runs, or as a **generic**, decided by the compiler. There is no inheritance between structs.

9.1 Interfaces

An interface lists methods without a body. A struct implements it by naming it after a colon and writing those methods:

```
interface Shape {
    f64 area() const
    string name() const
}

struct Circle : Shape {
    f64 radius

    Circle(f64 r) {
        radius = r
    }

    f64 area() const {
        ret 3.14159 * radius * radius
    }

    string name() const {
        ret "circle"
    }
}
```

```

    }
}

// ...

void describe(Shape s) {
    println("{} with area {:.2f}", s.name(), s.area())
}

i32 main() {
    Circle c = Circle(1.0)
    describe(c)
    describe(Square(2.0))
    ret 0
}

```

```

circle with area 3.14
square with area 4.00

```

A parameter of type `Shape` accepts any struct that implements `Shape`, without copying it and without allocating anything: the struct stays where it is, and the call goes through a table of its methods, as a virtual call in C++.

Values of an interface

A variable, a field or a list can hold values of an interface. A struct becomes one by moving to the heap, together with its table of methods; when the value ends, the struct is destroyed:

```

interface Shape {
    f64 area() const
}

struct Circle : Shape {
    f64 radius

    f64 area() const {
        ret 3.14159 * radius * radius
    }
}

// ...

i32 main() {
    Circle small
    small.radius = 1.0
}

```

```

Square big
big.side = 3.0
list<Shape> shapes = [small, big]
f64 total = 0.0
for (Shape s in shapes) {
    total += s.area()
}
println("{:.2f}", total)
ret 0
}

```

12.14

Here `small` and `big` are simple values, so the list receives copies; a struct that owns resources would be added with `move`. A value of an interface is never copied, only moved, and only the methods of the interface can be called on it. Its default value is empty, and calling a method on an empty value is a panic.

9.2 Generics

A generic function has type parameters, written between `<` and `>` after its name. Each type parameter is constrained by an interface, which says what the function can do with it:

```

interface Comparable {
    bool less(Self other) const
}

struct Version : Comparable {
    i32 major
    i32 minor

    Version(i32 a, i32 b) {
        major = a
        minor = b
    }

    bool less(Version other) const {
        if (major != other.major) {
            ret major < other.major
        }
        ret minor < other.minor
    }
}

```

```

T largest<T : Comparable>(list<T> items) {
    T best = items[0]
    for (T item in items) {
        if (best.less(item)) {
            best = item
        }
    }
    ret best
}

i32 main() {
    list<Version> versions = [Version(1, 2), Version(2, 0), Version(1, 9)]
    Version newest = largest(versions)
    println("{}.{}", newest.major, newest.minor)
    ret 0
}

```

2.0

- `T : Comparable` says that `T` can be any type that implements `Comparable`. In an interface, `Self` is the type that implements it: `less` compares a `Version` with another `Version`.
- The type arguments are deduced from the arguments of the call: `largest(versions)` is `largest<Version>`. When they cannot be deduced, they are written: `largest<Version>(versions)`.
- A generic is checked once, where it is written: on a `T`, only the methods of its constraints can be called, and the errors speak of the generic, not of its uses.
- The compiler makes a version of the function for each type it is used with, as the templates of C++: the calls are direct, and as fast as with a function written for that type.

An interface that uses `Self`, as `Comparable`, can only be a constraint, not the type of a parameter or of a value.

Interface or generic?

	<code>void f(Shape s)</code>	<code>void f<T : Shape>(T s)</code>
The call	through the table of methods	direct, often inlined
The code	one function	one version for each type

	<code>void f(Shape s)</code>	<code>void f<T : Shape>(T s)</code>
Different types in one list	yes, <code>list<Shape></code>	no, a <code>list<T></code> has one type

Use an interface for collections of different types and for code that does not need the speed; use a generic for algorithms over one type at a time.

9.3 Generic structs

A struct can have type parameters too, and so can its methods:

```
struct Stack<T> {
    list<T> items

    void push(move T value) {
        items.add(move value)
    }

    T pop() {
        ret items.remove(items.len() - 1)
    }

    bool empty() const {
        ret items.len() == 0
    }
}

struct Pair<A, B> {
    A first
    B second
}

i32 main() {
    Stack<string> stack
    stack.push("first")
    stack.push("second")
    println("{} {}", stack.pop(), stack.empty())

    Pair<string, i64> entry
    entry.first = "answer"
    entry.second = 42
    println("{:?}", entry)
    ret 0
}
```

```
second false
Pair { first: "answer", second: 42 }
```

A generic struct is always written with its type arguments: `Stack<string>`, `Pair<string, i64>`. The containers of the library, `list<T>` and `map<K, V>`, are generic structs written in Cromo in the same way.

9.4 Summary

- An interface lists methods; a struct implements it with `struct Name : Interface`.
- A parameter of an interface type takes any struct that implements it, without copying it.
- Values of an interface, as `list<Shape>`, own their struct on the heap.
- Generics have type parameters constrained by interfaces, are checked once, and are compiled for each type.
- There is no inheritance: shared behavior is an interface, shared data is a field.

Anonymous functions

An anonymous function, or *lambda*, is a function written where it is used, as an argument or a value. It can use the variables around it, and the way it takes them follows the rules of ownership of chapter 4.

10.1 Writing a lambda

```
i64 count(list<i64> values, func(i64) bool keep) {
    i64 n = 0
    for (i64 v in values) {
        if (keep(v)) {
            n++
        }
    }
    ret n
}

i32 main() {
    list<i64> values = [3, 8, 12, 5, 20]
    i64 limit = 6
    i64 big = count(values, [limit](i64 x) bool { ret x > limit })
    println("{} values are above {}", big, limit)
    ret 0
}
```

```
3 values are above 6
```

The lambda is `[limit](i64 x) bool { ret x > limit }`:

- between square brackets, the variables it **captures**, here `limit`;

- its parameters, as a function's;
- the type it returns, after the parameters; without it, the lambda returns nothing;
- its body.

`func(i64) bool` is the type of a function that takes an `i64` and returns a `bool`. A parameter of a function type is called as a function: `keep(v)`.

10.2 Captures

A lambda says how it uses each variable it captures, with the same words as a parameter:

Capture	The lambda
<code>[limit]</code>	reads the variable
<code>[ref total]</code>	modifies the variable
<code>[move name]</code>	takes the value, which the variable loses

```
i64 count(list<i64> values, func(i64) bool keep) {
    i64 n = 0
    for (i64 v in values) {
        if (keep(v)) {
            n++
        }
    }
    ret n
}
```

```
i32 main() {
    list<i64> values = [3, 8, 12, 5, 20]
    i64 total = 0
    count(values, [ref total](i64 x) bool {
        total += x
        ret true
    })
    println("total {}", total)
    ret 0
}
```

total 48

Without `ref`, a lambda cannot modify what it captures, and the compiler says so.

10.3 Lambdas as values

A lambda that captures by address, with `ref` or by reading a value that is not simple, can only be passed as an argument: the variables it refers to live in the function that calls, and the lambda ends with the call. A lambda that captures only simple values, or takes its values with `move`, is a value: it can be kept in a variable, a field or a list, returned and moved.

```
func(i64) i64 adder(i64 amount) {
    ret [amount](i64 x) i64 { ret x + amount }
}

struct Button {
    string label
    func() onClick

    void click() {
        onClick()
    }
}

i32 main() {
    func(i64) i64 plus3 = adder(3)
    println("{} ", plus3(4))

    list<func(i64) i64> steps = [adder(1), adder(10), adder(100)]
    i64 x = 0
    for (func(i64) i64 step in steps) {
        x = step(x)
    }
    println("{} ", x)

    string file = "notes.txt"
    Button save
    save.label = "Save"
    save.onClick = [move file]() { println("saving {} ", file) }
    save.click()
    ret 0
}
```

```
7
111
saving notes.txt
```

Returning a lambda that captures with `ref` does not compile: the variable would end with the function, and the lambda would refer to nothing.

```
func() makeCounter() {
    i64 n = 0
    ret [ref n]() { n++ }
}

i32 main() {
    ret 0
}
```

error: this lambda captures by address

A function value is moved, not copied. An empty one, as a field never assigned, is a panic when called.

10.4 const func

A parameter `const func(...)` accepts only a function that does not modify what it captures. Several threads can call such a function at the same time, safely: this is how the HTTP server of chapter 15 calls the same handler for many requests at once.

10.5 Summary

- A lambda is `[captures](parameters) type { body }`; its type is `func(parameters) type`.
- Captures use the words of parameters: read, `ref`, `move`.
- A lambda that refers to variables of its caller lives only during the call; one that owns what it captures is a value.
- `const func` is a function that can be called from several threads at once.

Modules

A program grows beyond one file by dividing its code into **modules**. Cromo has no header files and no preprocessor: a file is a module, a folder is a module, and `import` makes the public names of a module visible.

11.1 A file is a module

Next to `main.cro`, a file `geometry.cro`:

```
geometry.cro
public struct Point {
    public f64 x
    public f64 y
    i32 visits
}

public f64 distance(Point a, Point b) {
    ret root(square(a.x - b.x) + square(a.y - b.y))
}

f64 square(f64 v) {
    ret v * v
}

// The square root with Newton's method, enough for an example.
f64 root(f64 v) {
```

```

f64 guess = v / 2.0
for (i32 i = 0; i < 20; i++) {
    guess = (guess + v / guess) / 2.0
}
ret guess
}

```

The program imports it by its name:

```

main.cro
import geometry

i32 main() {
    Point a
    Point b
    b.x = 3.0
    b.y = 4.0
    println!("{}", distance(a, b))
    ret 0
}

5

```

`cromo build` or `cromo run` in the folder compiles `main.cro` and the modules it imports, and the modules they import, all together.

11.2 Visibility

Everything in a module is private to it, unless it is marked `public`: functions, structs, and also the fields, methods and constructors of a struct. Above, `Point` and `distance` are public; `square`, `root` and the field `visits` are not:

```

import geometry

i32 main() {
    println!("{}", square(2.0))
    Point p
    println!("{}", p.visits)
    ret 0
}

```

```

error: 'square' of geometry is private to its module: it is not marked public
error: field 'visits' of Point is private to its module: it is not marked public

```

What a public declaration shows must be public too: a public function cannot return a private struct. The destructor has no `public`: it runs wherever the object ends.

11.3 Using the names

After `import geometry`, its public names are used directly: `distance(a, b)`, `Point p`. The name of the module in front, `geometry.distance(a, b)`, is needed only when two imported modules give the same name, or to make the code clearer. A name is looked for first in the module itself, then in the imported modules, then in the standard library, so a name added to the library never breaks an existing program.

The imports are written at the top of the file, and each file writes its own: importing is not transitive. Two modules can import each other.

11.4 A folder is a module

`import shapes` imports the folder `shapes`, with all its `.cro` files, as one module:

```
shapes/circle.cro
public f64 circleArea(f64 r) {
    ret 3.14159 * r * r
}
```

```
shapes/square.cro
public f64 squareArea(f64 s) {
    ret s * s
}
```

```
areas.cro
import shapes

i32 main() {
    println("{:.2f} {}", circleArea(1.0), squareArea(2.0))
    ret 0
}
```

```
3.14 4
```

`import a.b` is the file `a/b.cro`, or the folder `a/b`. The paths start from the folder of the main file, so a module has the same name wherever it is imported. The files of a folder share their private names, as parts of one module; the subfolders are separate modules.

11.5 The standard library

The core of the library, with `list`, `map`, `string`, `format` and `Exception`, is visible everywhere. The other modules are imported with the prefix `std`:

Module	Content	Chapter
<code>std.fs</code>	files and folders	14
<code>std.os</code>	the environment of the process	14
<code>std.process</code>	running other programs	14
<code>std.time</code>	clocks, dates, pauses	14
<code>std.thread</code>	threads, tasks, mutexes, channels, atomics	13
<code>std.net</code>	TCP, UDP, TLS	15
<code>std.http</code>	HTTP client and server	15
<code>std.websocket</code>	WebSocket client and server	15
<code>std.json</code>	JSON	16
<code>std.crypto</code>	hashes, random bytes, encryption, signatures	15
<code>std.gzip</code>	compression	15

After `import std.fs`, the functions of the module are written with its last name: `fs.readText(path)`.

11.6 Code for each system

A module can have parts written for one system: next to `files.cro`, the files `files_windows.cro` and `files_linux.cro`. Only the file of the system the program is compiled for is used, and it is part of the module, with access to its private names. The standard library uses this to call the system of Windows and the one of Linux behind the same functions.

11.7 Packages

The libraries of other people will be **packages**: Git repositories, named by their URL, with versions given by their tags, listed in a file `cromo.mod` and downloaded by `cromo`. The design is decided, and it is described in `docs/language.md`; it is not available yet.

11.8 Summary

- A file is a module; a folder is a module made of its files; `import` makes their public names visible.
- Everything is private unless it is `public`, also the fields and methods of a struct.
- The names of an imported module are used directly, with the module in front when two clash.
- `std.x` are the modules of the standard library; `_windows` and `_linux` files hold code for one system.

Tests

Cromo has tests built in: a test is a function marked `[[test]]`, `assert` checks a condition, and `cromo test` runs the tests and reports the results. No library or framework is needed.

12.1 Writing a test

A test is a function without parameters that returns nothing, with the attribute `[[test]]` before it. It passes if it ends normally, and fails with a panic, an exception that it does not catch, or a stack overflow:

```
stats.cro
public f64 mean(list<f64> values) {
    f64 sum = 0.0
    for (f64 v in values) {
        sum += v
    }
    ret sum / f64(values.len())
}

[[test]]
void meanOfThree() {
    list<f64> values = [1.0, 2.0, 3.0]
    assert(mean(values) == 2.0)
}
```

Tests can be in the file of the code, or in a file next to it whose name ends with `_test.cro`. That file is part of the same module, so its tests see the private names too, and `cromo build` leaves it out of the executable:

```
stats_test.cro

[[test]]
void meanOfOne() {
    list<f64> values = [5.0]
    assert(mean(values) == 5.0)
}

[[test]]
void meanOfTwo() {
    list<f64> values = [1.0, 2.0]
    f64 m = mean(values)
    assert(m == 2.0)
}
```

12.2 Running the tests

```
main.cro

import stats

i32 main() {
    list<f64> values = [2.0, 4.0]
    println!("{}", mean(values))
    ret 0
}
```

`cromo test` in the folder of `main.cro` runs the tests of the program and of all its modules. The second test above is wrong on purpose, since the mean of 1 and 2 is 1.5:

```
cromo test

ok    stats.meanOfThree
ok    stats.meanOfOne
FAIL  stats.meanOfTwo
      panic: assertion failed: m == 2.0 (m is 1.5) at stats_test.cro:11:5
            at stats.meanOfTwo (stats_test.cro:11)
2 passed, 1 failed
```

- Each test runs in a process of its own, as many at a time as the processors: a test that crashes stops only itself.
- The output of a test is shown only when it fails.
- The exit code of `cromo test` is 1 if a test failed, so that a script or a continuous integration can stop.
- The names of the tests have their module in front: `stats.meanOfThree`.

Names after the file choose the tests to run: `cromo test main.cro meanOf` runs the tests whose name contains `meanOf`.

12.3 assert

`assert(condition)` stops the test with a panic when the condition is false. For a comparison of numbers or `bool`, the message gives the value of each side that is not a literal, computed once, as `m is 1.5` above: most of the time it is enough to understand what went wrong without a debugger.

`assert` works in every function, not only in tests, and also in release builds.

12.4 Testing errors

A test that expects an exception catches it:

```
i64 parsePort(string text) {
    i64 port = i64(text)
    if (port < 1 || port > 65535) {
        throw Error(format("{} is not a port", port))
    }
    ret port
}

[[test]]
void rejectsZero() {
    bool thrown = false
    try {
        parsePort("0")
    } catch (Exception e) {
        thrown = true
    }
    assert(thrown)
```

```
}  
  
i32 main() {  
    println!("{}", parsePort("8080"))  
    ret 0  
}
```

```
8080
```

12.5 Summary

- `[[test]] void name()` is a test; `assert(condition)` checks.
- Tests live in the code or in `name_test.cro`, which sees the private names of the module.
- `cromo test` runs each test in its own process, in parallel, and reports the failures with their cause.

Threads, tasks and channels

Cromo runs code at the same time in two ways: **threads** of the system, and lightweight **tasks**, many of which share a few threads. Both are in the module `std.thread`, with the same interface, and the compiler checks what they share: two threads never write the same data at the same time.

13.1 Threads

A `thread.Thread` is made from a lambda, started with `start()` and waited for with `join()`:

```
import std.thread

i64 sum(list<i64> values, i64 from, i64 to) {
    i64 total = 0
    for (i64 i = from; i < to; i++) {
        total += values[i]
    }
    ret total
}

i32 main() {
    list<i64> values
    for (i64 i = 1; i <= 1000000; i++) {
        values.add(i)
    }
    i64 low = 0
    thread.Thread worker = thread.Thread([values, ref low]() {
        low = sum(values, 0, 500000)
    })
}
```

```

worker.start()
i64 high = sum(values, 500000, 1000000)
worker.join()
println("{} ", low + high)
ret 0
}

```

```
500000500000
```

The lambda reads `values` and modifies `low`, both variables of `main`. While the thread runs, the compiler does not let `main` modify what the thread reads, nor touch what it modifies: here `main` only reads `values`, which is allowed. A thread that captures by address is a thread of its block: if `join()` is not called, the block waits for it at its end, so the variables it uses always outlive it.

A thread that captures only with `move` owns everything it uses, and is free: it can be kept in a list and outlive the block that started it.

An exception that ends a thread is thrown again by `join()`, in the thread that waits.

13.2 Mutex

When threads must modify the same data, the data go in a `thread.Mutex<T>`. They can be reached only inside `lock`, which gives them with `ref` to one thread at a time:

```

import std.thread

i32 main() {
    thread.Mutex<i64> counter
    thread.Thread a = thread.Thread([counter]() {
        for (i32 i = 0; i < 1000; i++) {
            counter.lock([&ref i64 n] { n++ })
        }
    })
    thread.Thread b = thread.Thread([counter]() {
        for (i32 i = 0; i < 1000; i++) {
            counter.lock([&ref i64 n] { n += 2 })
        }
    })
    a.start()
}

```

```

    b.start()
    a.join()
    b.join()
    counter.lock([])(ref i64 n) { println!("{}", n) }
    ret 0
}

```

3000

There is no way to reach the `i64` without the lock, so there is no way to forget it. `lock` is released at the end of the lambda, even when an exception goes through.

13.3 Tasks

A `thread.Task` has the interface and the rules of a `thread.Thread`, but it is not a thread of the system: the runtime runs many tasks on a few threads, one for each processor. When a task waits, for the network, for `time.sleep`, for a mutex or for `join()`, its thread runs other tasks in the meantime. The code is the same as with threads: there is no `async`, no `await`, and no function of a different color.

A task costs a few kilobytes: a program can have hundreds of thousands. `thread.Group` starts many of them and waits for all:

```

import std.thread
import std.time

i32 main() {
    time.Instant start = time.now()
    thread.Atomic<i64> done
    thread.Group jobs
    for (i64 id = 0; id < 10000; id++) {
        jobs.start([done]() {
            time.msleep(100)
            done.add(1)
        })
    }
    jobs.wait()
    println("{} tasks waited 100 ms each", done.load())
    println("in less than a second: {}", start.elapsed().milliseconds() < 1000)
    ret 0
}

```

```
10000 tasks waited 100 ms each
in less than a second: true
```

Ten thousand threads of the system would need gigabytes of stacks; ten thousand tasks need a few tens of megabytes.

- The tasks of a group can read the variables declared before the group, which stay constant until the end of its block, and take others with `move`; they cannot capture with `ref`.
- `wait()` waits for all the tasks, and throws again the first exception of a task. At the end of its block, a group waits for its tasks by itself.
- A task that calls a function that blocks its thread, as reading a file or a C function, keeps that thread; after about a millisecond the runtime gives the other tasks another thread.

Use tasks for work that waits, as servers and clients of the network; use threads for long computations and for C code that blocks.

13.4 Channels

A `thread.Channel<T>` passes values from one task or thread to another, in order:

```
import std.thread

i32 main() {
    thread.Channel<string> lines = thread.Channel<string>(16)
    thread.Atomic<i64> letters
    thread.Group readers
    for (i32 r = 0; r < 4; r++) {
        readers.start([lines, letters]() {
            for (string line in lines) {
                letters.add(line.len())
            }
        })
    }
    list<string> words = ["channels", "pass", "values", "between", "tasks"]
    for (string word in words) {
        lines.send(word.copy())
    }
}
```

```

    }
    lines.close()
    readers.wait()
    println("{} letters", letters.load())
    ret 0
}

```

30 letters

- `thread.Channel<T>(n)` holds up to `n` values: `send` waits while it is full, `receive` while it is empty, and a task that waits leaves its thread to the others. With capacity 0, the sender waits until a receiver has taken the value.
- `close()` ends the channel: the values already sent can still be received, then `receive()` throws an exception, and `for (T v in channel)` ends.
- `next(ref v)` receives without the exception: it returns `false` once the channel is closed and empty.

13.5 Atomics

`thread.Atomic<T>` holds an integer or a `bool` that several threads change without a lock: `load`, `store`, `add`, `exchange` and `compareExchange`. They are sequentially consistent, the simplest model to reason about. An atomic cannot be copied, and the tasks and threads that use it capture it by reading it, as `done` and `letters` above.

13.6 Summary

- `thread.Thread` is a thread of the system; `thread.Task` a lightweight task with the same interface.
- The compiler checks what threads and tasks capture: shared data that change go in a `thread.Mutex`.
- `thread.Group` starts many tasks and waits for them; a task that waits leaves its thread to others.
- `thread.Channel` passes values between tasks; `thread.Atomic` counts without locks.

Files, processes and time

This chapter is a tour of the modules that talk to the operating system: `std.fs` for files and folders, `std.os` for the environment, `std.process` for other programs and `std.time` for clocks and dates. They work in the same way on Windows and on Linux, and they report errors with exceptions.

14.1 Files

```
import std.fs

i32 main() {
    fs.createFolders("data/logs")
    fs.writeText("data/notes.txt", "first line\n")
    fs.appendText("data/notes.txt", "second line\n")

    string text = fs.readText("data/notes.txt")
    for (string line in text.trimEnd().split("\n")) {
        println("> {}", line)
    }
    println("{} bytes", fs.size("data/notes.txt"))

    fs.copy("data/notes.txt", "data/copy.txt")
    fs.rename("data/copy.txt", "data/old.txt")
    println("{} {}", fs.isFile("data/old.txt"), fs.isFolder("data/logs"))

    fs.removeAll("data")
    println("{} ", fs.exists("data"))
    ret 0
}
```

```

> first line
> second line
23 bytes
true true
false

```

Function	Does
<code>readText(path)</code> , <code>readBytes(path)</code>	the whole content, as a <code>string</code> (checked as UTF-8) or as bytes
<code>writeText</code> , <code>writeBytes</code> , <code>appendText</code>	write a file, creating or replacing it, or add to its end
<code>exists</code> , <code>isFile</code> , <code>isFolder</code> , <code>isLink</code>	ask what a path is
<code>size</code> , <code>modified</code>	the size in bytes, and the time of the last change in seconds from 1970
<code>copy</code> , <code>rename</code> , <code>remove</code>	copy, move or delete a file
<code>createFolder</code> , <code>createFolders</code>	create a folder, or a folder with the folders above it
<code>listFolder(path)</code>	the names in a folder
<code>removeAll(path)</code>	delete a folder with all it contains, without following links
<code>joinPath</code> , <code>parent</code> , <code>fileName</code> , <code>extension</code>	the parts of a path

A file that cannot be read or written is an exception, with a message that says which and why:

```

import std.fs

i32 main() {
    try {
        string text = fs.readText("missing.txt")
    } catch (Exception e) {
        println("{} ", e.message())
    }
    ret 0
}

```

```
cannot read 'missing.txt': no such file or folder
```

14.2 The environment

`std.os` gives the name of the system, `"windows"` or `"linux"`, the variables of the environment and the current folder; `os.exit(code)` ends the program at once. The arguments of the program arrive in `main`:

```
import std.os

i32 main(list<string> args) {
    for (i64 i = 1; i < args.len(); i++) {
        println("argument {}: {}", i, args[i])
    }
    if (os.hasEnv("CROMO_GUIDE_EXAMPLE")) {
        println!("{}", os.env("CROMO_GUIDE_EXAMPLE"))
    }
    ret 0
}
```

```
argument 1: alpha
argument 2: beta
```

`args[0]` is the name of the program, as in C.

14.3 Other programs

`process.run` runs a program, waits for it, and gives its exit code, its output and its errors:

```
import std.process

i32 main() {
    process.Output result = process.run("git", ["status", "--short"])
    if (result.code != 0) {
        eprintln("git failed: {}", result.err)
        ret 1
    }
    print!("{}", result.out)
    ret 0
}
```

The program is looked for in the folders of the `PATH`, and the arguments are passed as they are, without a shell that interprets them. An exit code other than 0 is a value to check; a program that cannot start is an exception.

`process.start` starts a program without waiting, and gives a `process.Process` to talk with it: `write` and `closeInput` for its standard input, `read` and `readAll` for its output, `wait` for its end and `kill` to stop it.

14.4 Time

```
import std.time

i32 main() {
    time.Instant start = time.now()
    time.msleep(50)
    time.Duration waited = start.elapsed()
    println("at least 50 ms: {}", waited.milliseconds() >= 50)

    time.DateTime now = time.utcnow()
    println("the year has four digits: {}", now.year >= 1000)
    ret 0
}
```

```
at least 50 ms: true
the year has four digits: true
```

- `time.now()` reads a clock that only goes forward, for measuring; `elapsed()` and `since()` give a `Duration`, with `seconds()`, `milliseconds()`, `microseconds()` and the field `nanoseconds`.
- `utcNow()` and `localNow()` give a `DateTime`, with its fields from the year to the nanosecond, the offset from UTC in minutes, and `text()` in the format of ISO 8601: `2026-10-30T18:21:33+01:00`.
- `sleep`, `msleep` and `usleep` pause; in a task, the thread runs other tasks meanwhile.

14.5 Summary

- `std.fs` reads and writes files and folders; errors are exceptions with a clear message.

- `std.os` gives the environment; the arguments arrive in `main(list<string> args)`.
- `std.process` runs other programs, without a shell.
- `std.time` measures durations and gives the date and time.

Networking and HTTP

The standard library has everything a network program needs: TCP and UDP in `std.net`, TLS for encrypted connections, HTTP clients and servers in `std.http`, WebSocket in `std.websocket`, and the cryptography and compression they rely on. It needs no other library: TLS comes from Mbed TLS, compiled into the runtime of Cromo, so the executables stay self-contained.

The calls block until they are done, as in C. In a task, a call that waits for the network leaves its thread to the other tasks, so a server written in the simplest way, one task for each connection, serves thousands of clients at once.

15.1 TCP

A server listens with `net.listen`, accepts connections with `accept()`, and reads and writes on each `net.Connection`. A client connects with `net.connect`. The program below is both: its server runs in a task, and its client talks to it.

```
import std.net
import std.thread

i32 main() {
    net.Listener listener = net.listen("127.0.0.1", 0) // port 0: any free
    port
    i32 port = listener.port()
    thread.Task server = thread.Task([move listener]() {
        net.Connection client = listener.accept()
        list<u8> request = client.read(1024)
        client.writeText("echo: " + string(request))
    })
}
```

```

server.start()

net.Connection connection = net.connect("127.0.0.1", port)
connection.writeText("hello")
println("{} ", string(connection.read(1024)))
server.join()
ret 0
}

```

echo: hello

- `read(most)` waits for data and returns up to `most` bytes; an empty list means that the other side has closed.
- `timeout(ms)` limits every following operation; a timeout, as any network error, is an exception.
- The destructor of a connection closes it.
- `net.udp(host, port)` opens a UDP socket, with `send` and `receive`; `net.resolve(host)` gives the addresses of a name. IPv4 and IPv6 work on both systems.

15.2 TLS

`net.connectTls` opens an encrypted connection, with the same `read` and `write`:

```

import std.net

i32 main() {
    net.Connection site = net.connectTls("example.com", 443)
    site.writeText("GET / HTTP/1.1\r\nHost: example.com\r\nConnection:
close\r\n\r\n")
    println("{} ", string(site.read(200)))
    ret 0
}

```

The certificate of the server must be signed by an authority that the system trusts, and name the host; otherwise the connection fails with an exception that says why, as `the certificate has expired`. A server uses `net.listenTls(host, port, certificate, key)`, with the certificate and the key as texts in PEM.

15.3 An HTTP server

`http.serve` gives every request to one function, the handler, which fills the response:

```
import std.http

i32 main() {
    http.serve("0.0.0.0", 8080, [(ref http.Request req, ref http.Response res)
    {
        if (req.matches("GET /")) {
            res.write("Welcome!")
        } else if (req.matches("GET /hello/{name}")) {
            res.write("Hello, " + req.param("name") + "!")
        } else {
            res.status = 404
            res.write("not found")
        }
    })
    ret 0
}
```

Run it and open `http://localhost:8080/hello/world` in a browser.

- Each connection is served by a task of its own, so the handler runs on several threads at once. It is a `const func`: what the requests share must be in a `thread.Mutex` or a `thread.Atomic`.
- `req.matches("GET /users/{id}")` checks the method and the path, and `req.param("id")` reads the part in braces; `{rest...}` takes the rest of the path. `req.queryValue("page")` reads the query.
- `res.write`, `res.writeJson` and `res.writeBytes` write the body and set its `Content-Type`; `res.status` is 200 by default; `res.headers` holds the headers.
- An exception in the handler gives the status 500, and the server goes on.
- The server keeps the connections alive between requests, and reads and writes bodies in chunks.

`http.listen(host, port)` gives a `http.Server` without starting it, whose `port()` is known before `serve(handler)`: useful with port 0, and in tests.

15.4 An HTTP client

```
import std.http

i32 main() {
    http.Response r = http.get("https://example.com/")
    println("{} {}", r.status, r.headers.get("content-type"))
    println("{} bytes", r.body.len())
    ret 0
}
```

`http.get(url)`, `http.post(url, body, type)` and `http.send(request)` follow redirects and use TLS for `https`. A status of error, as 404, is a response like the others; a failure of the network is an exception. The client asks for gzip and decompresses the body by itself.

An `http.Client` keeps the connections open between requests to the same host, which makes many requests much faster, and can be used by several tasks at once.

The server and the client together, in one program:

```
import std.http
import std.os
import std.thread

i32 main() {
    http.Server server = http.listen("127.0.0.1", 0)
    i32 port = server.port()
    thread.Task task = thread.Task([move server]() {
        server.serve([](ref http.Request req, ref http.Response res) {
            if (req.matches("GET /hello/{name}")) {
                res.write("Hello, " + req.param("name") + "!")
            } else {
                res.status = 404
            }
        })
    })
    task.start()

    http.Client client
    http.Response ok = client.get(format("http://127.0.0.1:{}/hello/Ada", port))
    http.Response missing = client.get(format("http://127.0.0.1:{}/other",
```

```

port))
    println("{} {}", ok.status, ok.text())
    println("{} ", missing.status)
    os.exit(0) // the server would run forever
    ret 0
}

```

```

200 Hello, Ada!
404

```

15.5 WebSocket

A WebSocket keeps a connection open in both directions, for messages. The server takes it from a request with `req.websocket`, and a client opens one with `websocket.connect`:

```

import std.http
import std.websocket

i32 main() {
    http.serve("0.0.0.0", 8080, [](ref http.Request req, ref http.Response res)
    {
        if (req.matches("GET /echo")) {
            req.websocket([](ref websocket.Socket ws) {
                for {
                    switch (ws.receive()) {
                        case Text(string text) {
                            ws.send("echo: " + text)
                        }
                        case Binary(list<u8> bytes) {
                            ws.sendBytes(bytes)
                        }
                    }
                }
            })
        }
    })
    ret 0
}

```

`receive()` throws `websocket.Closed` when the other side closes, which ends the loop above. `websocket.connect("wss://...")` opens one with TLS.

15.6 Cryptography and compression

`std.crypto` has the building blocks that network programs need, and `std.gzip` compresses:

```
import std.crypto
import std.gzip

i32 main() {
    list<u8> data = "hello".bytes()
    println!("{}", crypto.hex(crypto.sha256(data)))
    println!("{}", crypto.base64(data))

    list<u8> key = crypto.random(32)
    list<u8> nonce = crypto.random(12)
    list<u8> sealed = crypto.encrypt(crypto.Cipher.ChaCha20Poly1305, key, nonce,
data, list<u8>())
    list<u8> opened = crypto.decrypt(crypto.Cipher.ChaCha20Poly1305, key, nonce,
sealed, list<u8>())
    println("{} bytes sealed, opened: {}", sealed.len(), string(opened))

    list<u8> text = "abc".repeat(1000).bytes()
    list<u8> packed = gzip.compress(text)
    println("{} bytes, compressed below 100: {}", text.len(), packed.len() <
100)
    println("{} bytes again", gzip.decompress(packed).len())
    ret 0
}
```

```
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
aGVsbG8=
21 bytes sealed, opened: hello
3000 bytes, compressed below 100: true
3000 bytes again
```

- Hashes: `md5`, `sha1`, `sha256`, `sha512`, and `hmac`; `random(n)` takes bytes from the generator of the system.
- `encrypt` and `decrypt` with AES-GCM or ChaCha20-Poly1305: `decrypt` throws if the data were changed.
- Ed25519 signatures and X25519 key exchange; `hex` and `base64` with their inverses.
- `gzip.decompressUpTo(data, limit)` refuses data that would grow beyond a limit, for data from the network.

15.7 Summary

- `std.net` has TCP, UDP and TLS, with blocking calls that leave the thread to other tasks while they wait.
- `http.serve` runs a handler for each request, on many tasks at once; `http.get` and `http.Client` make requests.
- `std.websocket`, `std.crypto` and `std.gzip` complete the toolbox, with no external library.

JSON is how most services exchange data. In Cromo, the compiler writes the conversion between a struct and JSON by itself, the first time it is needed: no annotation is required on the struct, no code is generated by a separate tool, and the conversion is as fast as code written by hand.

16.1 From structs to JSON and back

```
import std.json

struct Address {
    string city
    string country
}

struct Person {
    string name
    i64 born
    list<string> languages
    Address address
}

i32 main() {
    Person p
    p.name = "Grace"
    p.born = 1906
    p.languages = ["COBOL", "FLOW-MATIC"]
    p.address.city = "New York"
    p.address.country = "USA"
```

```

string text = json.encode(p)
println("{} ", text)

Person back = json.decode<Person>(text)
println("{} was born in {}", back.name, back.born)
ret 0
}

```

```

{"name":"Grace","born":1906,"languages":["COBOL","FLOW-MATIC"],"address":
{"city":"New York","country":"USA"}}
Grace was born in 1906

```

- `json.encode(value)` writes compact JSON; `json.pretty(value)` writes it on several lines, indented.
- `json.decode<T>(text)` reads a `T`. A member that is missing, or `null`, leaves the default value of its field; a member without a field is ignored.
- The conversion works for numbers, `bool`, `string`, lists, maps with string keys, fixed arrays, enums and structs whose fields can be converted. A field that cannot, as a function, is a compile error that names it.

16.2 Names and fields to skip

The attribute `[[json(...)]]` before a field changes its name in JSON, or leaves it out:

```

import std.json

struct Account {
    [[json("user_id")]] i64 id
    string name
    [[json(skip)]] string password
}

i32 main() {
    Account a
    a.id = 7
    a.name = "Ada"
    a.password = "secret"
    println("{} ", json.encode(a))
    ret 0
}

```

```
{"user_id":7, "name":"Ada"}
```

16.3 Enums

A case without data is written as its name; a case with data as an object whose only member is the case:

```
import std.json

enum Shape {
    Empty
    Circle(f64 radius)
    [[json("rect")]] Rectangle(f64 width, f64 height)
}

i32 main() {
    list<Shape> shapes = [Shape.Empty, Shape.Circle(2.5), Shape.Rectangle(1.0,
2.0)]
    string text = json.encode(shapes)
    println("{} ", text)
    list<Shape> back = json.decode<list<Shape>>(text)
    println("{:?}", back[2])
    ret 0
}
```

```
["Empty", {"Circle":{"radius":2.5}}, {"rect":{"width":1.0, "height":2.0}}]
Rectangle(1.0, 2.0)
```

16.4 Values of any shape

When the shape of the data is not known in advance, `json.parse` gives a `json.Value`, an enum with the cases `Null`, `Bool`, `Int`, `UInt`, `Float`, `Text`, `Array` and `Object`:

```
import std.json

i32 main() {
    json.Value v = json.parse("{\"items\": [1, 2.5, \"x\", null], \"ok\": true}")
    json.Value items = v.get("items")
    println("{} items, the first is {}", items.len(), items.at(0).asInt())
    println("{} {}", items.at(2).asText(), items.at(3).isNull())
}
```

```

switch (items.at(1)) {
    case Int(i64 n) {
        println("an integer: {}", n)
    }
    case Float(f64 f) {
        println("a float: {}", f)
    }
    default {
        println("something else")
    }
}

json.Value made = json.object()
made.set("language", json.Value.Text("Cromo"))
made.set("version", json.Value.Int(27))
println!("{}", json.encode(made))
ret 0
}

```

```

4 items, the first is 1
x true
a float: 2.5
{"language":"Cromo","version":27}

```

Integers and floating-point numbers are kept apart: an identifier of 64 bits keeps all its digits.

16.5 Errors

A text that is not JSON, or JSON that does not fit the type, is an exception that says where:

```

import std.json

struct Point {
    i64 x
    i64 y
}

i32 main() {
    try {
        Point p = json.decode<Point>("{\"x\": 1, \"y\": \"two\"}")
    } catch (Exception e) {
        println!("{}", e.message())
    }
}

```

```

    }
    try {
        json.Value v = json.parse("{\"x\\\": 1,, }")
    } catch (Exception e) {
        println("{} ", e.message())
    }
    ret 0
}

```

cannot decode JSON: at y: expected an integer, found a string
cannot read JSON: line 1, column 9: expected the name of a member, in double quotes

16.6 Summary

- `json.encode`, `json.pretty` and `json.decode<T>` convert structs, lists, maps and enums, with code written by the compiler.
- `[[json("name")]]` renames a field or a case; `[[json(skip)]]` leaves a field out.
- `json.parse` gives a `json.Value` for data of any shape.
- Errors say where the problem is.

Calling C

Every system offers its services through C functions, and many libraries are written in C. Cromo calls them directly, without a C compiler and without header files, from modules that declare it with `import unsafe`. The rest of the program uses those modules through safe functions and structs, so the unsafe part stays small and easy to find.

17.1 Declaring C functions

A module that writes `import unsafe` can declare C functions in an `extern` block, which names their library:

```
import unsafe

extern "kernel32" {
    i32 MulDiv(i32 number, i32 numerator, i32 denominator)
    i32 QueryPerformanceFrequency(ref i64 frequency)
}

i32 main() {
    println("{} ", MulDiv(10, 3, 4))
    i64 ticks
    QueryPerformanceFrequency(ref ticks)
    println("the counter runs: {} ", ticks > 0)
    ret 0
}

8
the counter runs: true
```

This program is for Windows, whose `kernel32.dll` has these functions.

- On Windows, `extern "name"` is `name.dll`. `cromo` reads the functions that the DLL exports while it compiles, so a misspelled name is an error of the compiler, not a crash at start-up. It writes the import library by itself: the Windows SDK is not needed. A `name.lib` next to the module is used instead, for example a static C library.
- On Linux, `extern "name"` is the static library `libname.a` next to the module.
- A DLL next to a module is copied next to the executable.

```
import unsafe

extern "kernel32" {
    i32 MulDvi(i32 number, i32 numerator, i32 denominator)
}

i32 main() {
    ret MulDvi(1, 2, 3)
}
```

```
error: kernel32.dll has no function 'MulDvi'
```

17.2 Types at the boundary

C functions take and return numbers, `bool` and `addr`, an address of C that Cromo code cannot follow. A `ref` parameter is a C pointer to a number or to a struct of numbers, which has the layout of C. The `B00L` of Windows is an `i32`.

The C functions keep their name, and are private to their module: other modules cannot call them.

17.3 Wrapping a resource

A resource of C, as a handle or a descriptor, belongs in a struct with a destructor. The rest of the program then uses the struct, and the resource is always released, as any other in Cromo:

```

import unsafe

extern "kernel32" {
    i32 QueryPerformanceCounter(ref i64 count)
    i32 QueryPerformanceFrequency(ref i64 frequency)
}

// Measures the time from its creation.
public struct Stopwatch {
    i64 start
    i64 frequency

    public Stopwatch() {
        QueryPerformanceFrequency(ref frequency)
        QueryPerformanceCounter(ref start)
    }

    public i64 micros() const {
        i64 now
        QueryPerformanceCounter(ref now)
        ret (now - start) * 1000000 / frequency
    }
}

```

The module offers `Stopwatch`; the modules that import it do not write `import unsafe` and cannot touch the C functions.

17.4 The unsafe dialect

`import unsafe` also gives the type `addr` and the built-in functions that work with raw memory, as `alloc` and `free`. The standard library is written in this dialect: `list`, `string` and `map` are structs of Cromo that manage their memory with these functions, behind a safe interface.

A program normally has no reason to use the dialect, except to call C. A search for `import unsafe` finds every place where the safety of the program depends on the care of its author.

17.5 Summary

- `import unsafe` allows `extern` blocks of C functions, the type `addr` and raw memory.

- `extern "name"` is `name.dll` on Windows, `libname.a` on Linux; the functions are checked while compiling.
- C resources go in structs with destructors, and the rest of the program uses those.

Final project: a web service

The last chapter puts together what the guide has shown: a small service that keeps a list of things to do, with a JSON API over HTTP. It uses structs and ownership, a `thread.Mutex` shared by the requests, the JSON written by the compiler, the HTTP server and client, and a test.

The service answers four requests:

Request	Answer
<code>GET /todos</code>	the list, in JSON
<code>POST /todos</code> with <code>{"title": "..."} </code>	201 and the new todo, or 400 without a title
<code>POST /todos/{id}/done</code>	204, or 404 if the todo does not exist
anything else	404

18.1 The data

```
struct Todo {
    i64 id
    string title
    bool done
}

struct NewTodo {
    string title
}
```

`Todo` is what the service keeps and sends; `NewTodo` is what a client sends to create one. Both are converted to and from JSON by the compiler, without a line of code.

18.2 The store

The todos live in a `Store`, a struct with its methods:

```
struct Store {
    list<Todo> items
    i64 next = 1

    Todo add(string title) {
        Todo todo
        todo.id = next
        todo.title = title.copy()
        next++
        items.add(todo.copy())
        ret todo
    }

    bool complete(i64 id) {
        for (ref Todo todo in items) {
            if (todo.id == id) {
                todo.done = true
                ret true
            }
        }
        ret false
    }
}
```

`add` keeps a copy of the new todo in the list and returns the other one, which the handler sends back to the client. `complete` visits the list with `ref`, to modify the todo it finds.

The store has nothing to do with HTTP, so it is easy to test:

```

[[test]]
void addGivesNewIds() {
    Store store
    Todo first = store.add("a")
    Todo second = store.add("b")
    assert(second.id == first.id + 1)
    assert(store.complete(first.id))
    assert(!store.complete(99))
}

```

18.3 The handler

The server calls the handler for each request, on many tasks at once. The store is shared by all the requests, so it lives in a `thread.Mutex<Store>`, and the handler reaches it only inside `lock`:

```

void handle(thread.Mutex<Store> store, ref http.Request req, ref http.Response
res) {
    if (req.matches("GET /todos")) {
        store.lock([ref res](ref Store s) { res.writeJson(json.encode(s.items))
    })
    } else if (req.matches("POST /todos")) {
        NewTodo input = json.decode<NewTodo>(req.text())
        if (input.title == "") {
            res.status = 400
            res.write("a todo needs a title")
            ret
        }
        store.lock([ref res, input](ref Store s) {
            res.status = 201
            res.writeJson(json.encode(s.add(input.title)))
        })
    } else if (req.matches("POST /todos/{id}/done")) {
        i64 id = i64(req.param("id"))
        bool found = false
        store.lock([ref found, id](ref Store s) { found = s.complete(id) })
        res.status = 404
        if (found) {
            res.status = 204
        }
    } else {
        res.status = 404
    }
}

```

Some details:

- The lambdas given to `lock` capture `res` with `ref`, to write the response, and the values they need by reading them.
- A body that is not JSON makes `json.decode` throw; the server turns the uncaught exception into a response with status 500, and goes on with the other requests.
- `i64(req.param("id"))` throws in the same way for an id that is not a number.

18.4 The whole program

Here is the `main` of the service: the declarations of the previous sections stand in place of `// ...`, and on the site *Show all* opens the whole file. To try the service without a browser, `main` also starts a client that sends some requests and prints the answers; a real service would only call `serve`.

```
todo.cro

import std.http
import std.json
import std.os
import std.thread

// ...

i32 main() {
    thread.Mutex<Store> store
    http.Server server = http.listen("127.0.0.1", 0)
    string base = format("http://127.0.0.1:{}", server.port())
    thread.Task service = thread.Task([store, move server]() {
        server.serve([store](ref http.Request req, ref http.Response res) {
handle(store, ref req, ref res) })
    })
    service.start()

    // A client that tries the service.
    http.Client client
    http.Response created = client.post(base + "/todos", "{\"title\": \"write
the guide\"}", "application/json")
    println("{} {}", created.status, created.text())
    client.post(base + "/todos", "{\"title\": \"publish it\"}",
"application/json")
}
```

```

http.Response done = client.post(base + "/todos/1/done", "", "text/plain")
println!("{}", done.status)
// ...
os.exit(0)
ret 0
}

```

```

201 {"id":1,"title":"write the guide","done":false}
204
[{"id":1,"title":"write the guide","done":true},{"id":2,"title":"publish
it","done":false}]
[x] write the guide
[ ] publish it
400 a todo needs a title

```

And its test:

```
cromo test todo.cro
```

```

ok    addGivesNewIds
1 passed, 0 failed

```

18.5 Making it a real service

To serve real clients, replace the client in `main` with the server alone, on a fixed port:

```

i32 main() {
    thread.Mutex<Store> store
    http.serve("0.0.0.0", 8080, [store](ref http.Request req, ref http.Response
res) { handle(store, ref req, ref res) })
    ret 0
}

```

Then build it for the server, which can be Linux even when you work on Windows:

```
cromo build --release --target linux todo.cro
```

The result is one executable, without dependencies, that runs on any Linux x64.

From here, the service could keep its data in a file with `std.fs`, serve HTTPS with `http.listenTls`, or check a token in the headers of each request. The library has what is needed for each, and the compiler checks that the requests, running at the same time, never touch the store without the lock.

18.6 Where to go next

- `docs/language.md` in the [repository](#) is the draft specification, with every rule of the language.
- `docs/decisions.md` explains why each rule is as it is.
- The folder `examples/` has more programs, and `library/` the standard library, written in Cromo.

Thank you for reading, and welcome to Cromo.

A

Keywords and operators

A.1 Keywords

Keyword	Use
<code>break</code>	leaves a loop
<code>case</code>	a branch of a <code>switch</code>
<code>catch</code>	handles an exception
<code>const</code>	an immutable variable; a method that does not modify its object; a <code>const func</code>
<code>continue</code>	goes to the next round of a loop
<code>default</code>	the branch of a <code>switch</code> for the values not listed
<code>else</code>	the alternative of an <code>if</code>
<code>enum</code>	declares an enum
<code>extern</code>	declares C functions, in a module with <code>import unsafe</code>
<code>false</code> , <code>true</code>	the values of <code>bool</code>
<code>finally</code>	runs at every exit of a <code>try</code>
<code>for</code>	the loop, in all its forms
<code>func</code>	the type of a function
<code>if</code>	a condition

Keyword	Use
<code>import</code>	imports a module
<code>in</code>	in a <code>for</code> over the elements
<code>interface</code>	declares an interface
<code>move</code>	moves a value; a parameter or a capture that takes ownership
<code>public</code>	makes a declaration visible outside its module
<code>ref</code>	a parameter or a capture that modifies; the elements of a <code>for</code> that can be modified
<code>ret</code>	returns from a function
<code>struct</code>	declares a struct
<code>switch</code>	chooses among cases
<code>throw</code>	throws an exception
<code>try</code>	the block whose exceptions are handled
<code>unsafe</code>	<code>import unsafe</code> : the dialect of C functions and raw memory
<code>void</code>	no result

`print`, `println`, `eprint`, `eprintln`, `format`, `panic` and `assert` are built-in functions, not keywords.

A.2 Built-in types

Type	Meaning
<code>i8</code> , <code>i16</code> , <code>i32</code> , <code>i64</code>	signed integers
<code>u8</code> , <code>u16</code> , <code>u32</code> , <code>u64</code>	unsigned integers
<code>f32</code> , <code>f64</code>	floating-point numbers
<code>bool</code>	<code>true</code> or <code>false</code>
<code>char</code>	a byte, a synonym of <code>u8</code>
<code>string</code>	text in UTF-8

Type	Meaning
<code>list<T></code>	a list that grows
<code>map<K, V></code>	a hash table
<code>T[n]</code>	an array of fixed size
<code>func(A, B) R</code>	a function
<code>addr</code>	an address of C, only with <code>import unsafe</code>

A.3 Operators

From the strongest to the weakest:

Operators	Meaning
<code>.</code> <code>()</code> <code>[]</code>	member, call, index
<code>!</code> <code>-</code>	not, inversion of the bits, negation
<code>*</code> <code>/</code> <code>%</code>	multiplication, division, remainder
<code>+</code> <code>-</code>	addition, subtraction; <code>+</code> also joins strings
<code><<</code> <code>>></code>	shifts
<code>&</code>	bitwise and
<code>^</code>	bitwise exclusive or
<code> </code>	bitwise or
<code>==</code> <code>!=</code> <code><</code> <code><=</code> <code>></code> <code>>=</code>	comparisons
<code>&&</code>	logical and, short-circuit
<code> </code>	logical or, short-circuit
<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code> <code>&=</code> <code> =</code> <code>^=</code> <code><<=</code> <code>>>=</code>	assignments, statements on their own

- `i++` and `i--` are statements on their own, never parts of an expression.
- There is no conditional operator `?:`: write an `if`.
- A shift beyond the width of the type is a panic in debug builds.
- Assignments are statements: `if (a = b)` is not Croma.

A.4 Literals

Literal	Example
integer	<code>42</code> , <code>1_000_000</code> , <code>0xFF</code> , <code>0b1010</code>
floating point	<code>3.14</code> , <code>1e-9</code> , <code>2.5e3</code>
character	<code>'a'</code> , <code>'\n'</code> , <code>'\t'</code> , <code>'\\'</code> , <code>'\''</code> , <code>'\x41'</code>
string	<code>"text"</code> , with the same escapes and <code>\</code>
list	<code>[1, 2, 3]</code>

A.5 Attributes

Attributes are written as in C++11, before what they apply to:

Attribute	Meaning
<code>[[test]]</code>	a test function
<code>[[json("name")]]</code>	the name of a field or a case in JSON
<code>[[json(skip)]]</code>	a field left out of JSON

B

The cromos command

`cromo` is the only tool of Cromo: the compiler, the linker, the test runner, the formatter and the language server, in one executable.

```
cromo <command> [options] [file.cro] [-- arguments]
```

Without a file, the commands use `main.cro` of the current folder.

B.1 Commands

Command	Does
<code>cromo build [file.cro]</code>	compiles the program into an executable
<code>cromo run [file.cro] [-- arguments]</code>	compiles the program into a temporary folder, runs it with the arguments that follow <code>--</code> , and ends with its exit code
<code>cromo test [file.cro] [names]</code>	runs the <code>[[test]]</code> functions of the program and of its modules; the names choose the tests that contain them
<code>cromo fmt [--check] [files and folders -]</code>	formats the code, by default of the current folder and its subfolders; <code>--check</code> only lists the files to format; <code>-</code> formats the standard input
<code>cromo lsp</code>	the language server, for editors, on the standard input and output
<code>cromo version</code>	prints the version of <code>cromo</code> and of LLVM
<code>cromo help</code>	prints the commands

B.2 Options of build and run

Option	Meaning
<code>-o <file></code>	the name of the executable; by default the name of the source, or of the folder without a file, with <code>.exe</code> for Windows
<code>--release</code>	integers wrap around on overflow, instead of stopping the program
<code>--target windows, --target linux</code>	the system of the executable; by default the one <code>cromo</code> runs on
<code>-O0 ... -O3</code>	the level of optimization; the default is <code>-O2</code>
<code>--emit-ir</code>	prints the LLVM IR of the program instead of writing the executable

B.3 Debug and release

	Debug (default)	<code>--release</code>
Optimization	<code>-O2</code>	<code>-O2</code>
Integer overflow	panic	wraps around
Bounds, division by zero, exclusivity	checked	checked
Stack traces	yes	yes
Information for a debugger	yes: PDB on Windows, DWARF on Linux	no

With `-O0` a debugger shows every variable; with the optimizations, some can be *optimized out*.

B.4 Exit codes

Code	Meaning
0	the program ended normally, or <code>main</code> returned 0
1	an exception left <code>main</code> ; or <code>cromo</code> found errors
101	a panic, or a stack overflow
other	the value returned by <code>main</code> , or given to <code>os.exit</code>

B.5 Files

Path	Content
<code>~/.cromo/current/bin/cromo</code>	the active version, which the <code>PATH</code> reaches
<code>~/.cromo/versions/<version></code>	the installed versions
<code>lib/</code> next to <code>bin/</code>	the standard library and the runtime, which <code>cromo</code> reads

On Windows, `~` is `%USERPROFILE%`.

C

Cromo for C++, Go and Rust programmers

Cromo borrows from each of these languages. This appendix lists, for each, what will feel familiar and what is different.

C.1 From C++

Familiar: the syntax, braces and types before names; structs with constructors, methods, `const` methods and destructors; RAII and the order of destruction; exceptions with `try`, `catch` and `throw`; generics compiled for each type, as templates; `std::format` for formatting.

Different:

C++	Cromo
<code>return</code>	<code>ret</code>
<code>;</code> at the end of statements	the end of the line
pointers, <code>nullptr</code> , <code>new</code> , <code>delete</code>	no pointers and no <code>null</code> ; the heap only through containers
<code>T&</code> , <code>const T&</code>	a parameter is read by default; <code>ref</code> modifies; no references outside parameters
copy constructors, implicit copies	<code>copy()</code> , always explicit for types that own resources

C++	Cromo
<code>std::move</code>	<code>move</code> , checked: a moved variable cannot be used
<code>class</code> , inheritance, <code>virtual</code>	<code>struct</code> and interfaces
templates with any type	generics constrained by interfaces, checked once
<code>while</code> , <code>do</code>	<code>for (condition)</code> , <code>for { }</code>
<code>switch</code> with fall-through	no fall-through; on enums every case must be handled
<code>#include</code> , headers	<code>import</code> of modules; no preprocessor
<code>int</code> , <code>long</code> , <code>size_t</code>	<code>i32</code> , <code>i64</code> , <code>u64</code> : fixed sizes; lengths are <code>i64</code>
undefined behavior on overflow, bounds	a panic in debug builds; bounds always checked
uninitialized variables	every variable has its default value
<code>std::thread</code> , <code>std::mutex</code>	<code>thread.Thread</code> , <code>thread.Mutex<T></code> that holds its data

C.2 From Go

Familiar: one tool that builds, runs, tests and formats; lightweight tasks that block without `async`; channels; a standard library for networks and HTTP; static executables that cross-compile; and, once packages arrive, dependencies named by the URL of their repository, with minimal version selection.

Different:

Go	Cromo
garbage collector	ownership and destructors; memory is freed at the end of the owner
<code>go f()</code>	<code>thread.Task</code> or <code>thread.Group</code> , whose captures the compiler checks
<code>defer</code>	destructors, and <code>finally</code>
<code>error</code> values and <code>if err != nil</code>	exceptions
<code>nil</code>	no <code>null</code> ; every variable has a value

Go	Cromo
<code>x := 1</code>	<code>i64 x = 1</code> : the type is always written
capital letters for exported names	<code>public</code>
<code>select</code>	not yet
slices that share memory	lists that own their elements; <code>slice()</code> makes a copy

C.3 From Rust

Familiar: ownership checked by the compiler, moves by default for types that own resources, no `null`, no data races; enums with data and exhaustive matching; generics constrained by interfaces, as traits; LLVM; tests built into the tool.

Different:

Rust	Cromo
references with lifetimes, the borrow checker	<code>ref</code> only for parameters: a reference never outlives a call, and there are no lifetimes
<code>&T</code> , <code>&mut T</code>	a read parameter, a <code>ref</code> parameter
<code>Clone</code> , <code>.clone()</code>	<code>copy()</code>
<code>Copy</code> types	simple types: numbers, <code>bool</code> , structs made only of them
<code>Result<T, E></code> and <code>?</code>	exceptions; enums with data for results that are values
<code>Option<T></code>	a conventional value, a <code>bool</code> with a <code>ref</code> parameter, or an enum
<code>match</code> with patterns	<code>switch</code> on cases, with the data named; no nested patterns
<code>Drop</code>	destructors
<code>impl Trait for Type</code>	<code>struct Type : Interface</code> , with the methods in the struct
<code>async</code> / <code>.await</code>	tasks that block, as in Go
<code>Arc<Mutex<T>></code>	<code>thread.Mutex<T></code> captured by the tasks that use it
macros	not available

C.4 In short

Cromo writes like C++, manages memory like Rust with fewer rules, and runs concurrent code like Go. When in doubt, the simplest rule wins.