



La guida di Cromo

Il linguaggio di programmazione Cromo

La guida di Cromo

Versione 0.27.1 · ottobre 2026

cromolang.org

github.com/cromolang/cromo

Il testo e gli esempi fanno parte dei sorgenti di Cromo, con la licenza MIT o Apache 2.0.

Ogni esempio di questa guida è stato compilato ed eseguito con cromolang 0.27.1, e il suo output verificato, quando la guida è stata generata.

Sommario

	Introduzione	8
	Perché Cromo	8
	A chi è rivolta questa guida	9
	Come usare questa guida	9
	Lo stato di Cromo	10
	Il nome	10
1	Primi passi	11
	1.1 Installazione	11
	1.2 Hello, world!	12
	1.3 Produrre un eseguibile	13
	1.4 Un progetto	13
	1.5 Formattazione	14
	1.6 Visual Studio Code	14
	1.7 Riepilogo	14
2	Un primo programma: contare le parole	16
	2.1 Il programma	16
	2.2 Come funziona	17
	2.3 Riepilogo	20
3	Concetti comuni	21
	3.1 Variabili	21
	3.2 Tipi	22
	3.3 Operatori	25
	3.4 Funzioni	26
	3.5 Commenti	26
	3.6 Istruzioni e righe	27
	3.7 Condizioni	27
	3.8 Cicli	27
	3.9 switch	29
	3.10 Riepilogo	29

4	Possesso	30
4.1	Ogni valore ha un solo proprietario	30
4.2	I valori semplici si copiano	31
4.3	I valori che possiedono risorse si spostano	32
4.4	Passare valori alle funzioni	34
4.5	I riferimenti vivono solo durante una chiamata	37
4.6	Esclusività	37
4.7	const	38
4.8	Riepilogo	38
5	Struct	39
5.1	Definire una struct	39
5.2	Costruttori	40
5.3	Metodi	41
5.4	Distruttori	43
5.5	Stampare una struct	44
5.6	Struct dentro altre struct	44
5.7	Riepilogo	45
6	Enum e switch	46
6.1	Casi senza dati	46
6.2	Casi con dati	47
6.3	switch copre ogni caso	48
6.4	Modificare i dati di un caso	49
6.5	Metodi ed enum generici	50
6.6	Riepilogo	51
7	Liste, mappe e testo	53
7.1	Liste	53
7.2	Array di dimensione fissa	54
7.3	Mappe	55
7.4	Stringhe	56
7.5	Formattazione	57
7.6	Riepilogo	58

8	Errori	59
8.1	Lanciare e catturare eccezioni	59
8.2	Tipi di eccezione	60
8.3	finally	61
8.4	Eccezioni e distruttori	62
8.5	Rilanciare un'eccezione	63
8.6	Eccezioni non catturate	63
8.7	Panic	64
8.8	Come scegliere	65
8.9	Riepilogo	65
9	Interfacce e generici	66
9.1	Interfacce	66
9.2	Generici	68
9.3	Struct generiche	70
9.4	Riepilogo	71
10	Funzioni anonime	72
10.1	Scrivere una lambda	72
10.2	Catture	73
10.3	Le lambda come valori	74
10.4	const func	75
10.5	Riepilogo	75
11	Moduli	76
11.1	Un file è un modulo	76
11.2	Visibilità	77
11.3	Usare i nomi	78
11.4	Una cartella è un modulo	78
11.5	La libreria standard	79
11.6	Codice per ogni sistema	80
11.7	Pacchetti	80
11.8	Riepilogo	80
12	Test	81
12.1	Scrivere un test	81
12.2	Eseguire i test	82
12.3	assert	83
12.4	Testare gli errori	83
12.5	Riepilogo	84

13	Thread, task e canali	85
13.1	Thread	85
13.2	Mutex	86
13.3	Task	87
13.4	Canali	88
13.5	Atomici	89
13.6	Riepilogo	89
14	File, processi e tempo	90
14.1	File	90
14.2	L'ambiente	92
14.3	Altri programmi	92
14.4	Il tempo	93
14.5	Riepilogo	94
15	Rete e HTTP	95
15.1	TCP	95
15.2	TLS	96
15.3	Un server HTTP	97
15.4	Un client HTTP	98
15.5	WebSocket	99
15.6	Crittografia e compressione	100
15.7	Riepilogo	101
16	JSON	102
16.1	Da struct a JSON e viceversa	102
16.2	Nomi e campi da escludere	103
16.3	Enum	104
16.4	Valori di forma qualsiasi	105
16.5	Errori	106
16.6	Riepilogo	106
17	Chiamare il C	107
17.1	Dichiarare funzioni C	107
17.2	I tipi al confine	108
17.3	Incapsulare una risorsa	108
17.4	Il dialetto unsafe	109
17.5	Riepilogo	109

18	Progetto finale: un servizio web	111
18.1	I dati	111
18.2	L'archivio	112
18.3	L'handler	113
18.4	Il programma completo	114
18.5	Farne un servizio vero	115
18.6	Per proseguire	116
A	Parole chiave e operatori	117
A.1	Parole chiave	117
A.2	Tipi integrati	118
A.3	Operatori	119
A.4	Letterali	120
A.5	Attributi	120
B	Il comando crom	121
B.1	Comandi	121
B.2	Opzioni di build e run	122
B.3	Debug e release	122
B.4	Codici di uscita	123
B.5	File	123
C	Cromo per programmatori C++, Go e Rust	124
C.1	Dal C++	124
C.2	Da Go	125
C.3	Da Rust	126
C.4	In breve	127

Introduzione

Ti diamo il benvenuto nella *Guida di Cromo*, un'introduzione al linguaggio di programmazione Cromo.

Cromo è un linguaggio compilato e di uso generale. Ha la sintassi del C e del C++, niente puntatori e niente garbage collector. Ogni valore ha uno e un solo proprietario, e le risorse che un valore detiene, come la memoria, i file e le connessioni, vengono rilasciate dal suo distruttore quando il proprietario esce dal proprio ambito. Il compilatore controlla il possesso con poche regole, e LLVM trasforma il programma in codice nativo veloce.

Perché Cromo

Cromo mette insieme idee che hanno già dato buona prova di sé in altri linguaggi:

- la **sintassi**, i costruttori e i **distruttori** del C e del C++, così che un programmatore C o C++ possa leggere codice Cromo fin dal primo giorno;
- il **possesso controllato dal compilatore**, come in Rust, ma con regole più semplici: i riferimenti esistono solo come parametri, quindi non serve mai scrivere un lifetime;
- **task leggeri e canali**, come in Go: migliaia di task aspettano la rete nello stesso momento, con normale codice bloccante e senza `async` né `await`;
- **un solo strumento** che compila, esegue, testa e formatta, come in Go e Zig;
- l'ottimizzatore e il linker di **LLVM**, gli stessi che usano anche Clang, Rust e Swift.

Il criterio che guida il progetto è la semplicità: quando due regole offrono la stessa sicurezza, Cromo sceglie quella più facile da spiegare a un programmatore C o C++.

A chi è rivolta questa guida

Questa guida presuppone che tu abbia già scritto programmi in almeno un altro linguaggio. Non presuppone il C o il C++: quando Cromo li segue, la guida lo dice, e spiega tutto ciò che un programmatore C++ darebbe per scontato.

Se conosci già il C++, Go o Rust, l'[Appendice C](#) riassume le differenze, e puoi scorrere rapidamente i primi capitoli.

Come usare questa guida

I capitoli sono pensati per essere letti in ordine: ognuno si basa sui precedenti. Il capitolo 2 è un programma piccolo ma completo, scritto prima di spiegarne i dettagli; i capitoli dal 3 al 12 presentano il linguaggio, quelli dal 13 al 16 la libreria standard; il capitolo 17 mostra come chiamare il C, e il capitolo 18 mette tutto insieme in un servizio web.

Ogni programma di questa guida viene compilato ed eseguito quando si genera la guida, e il suo output è quello riportato subito sotto:

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

```
Hello from the guide!
```

Un blocco con il bordo rosso è un programma che **non** compila, mostrato di proposito, seguito dall'errore del compilatore:

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

```
error: values of type string are not copied implicitly
```

La guida è disponibile anche in PDF, in inglese e in italiano: usa il pulsante **PDF** in cima alla pagina.

Lo stato di Cromo

Cromo è giovane. La versione attuale è la 0.27, e il linguaggio può ancora cambiare prima della versione 1.0. Ogni decisione di progetto è registrata, con le sue motivazioni, nel file `docs/decisions.md` del [repository](#), accanto alla bozza di specifica `docs/language.md`.

Cromo funziona su Windows x64 e Linux x64, e ogni versione compila programmi per entrambi i sistemi.

Il nome

Cromo è il nome italiano dell'elemento 24 della tavola periodica (in inglese *chromium*), il cui simbolo, Cr, compare nel logo. Sul web il linguaggio si chiama *cromolang*, come in cromolang.org.

Primi passi

In questo capitolo installi Cromo, scrivi il tuo primo programma e fai la conoscenza di `cromo`, il comando che compila, esegue, testa e formatta il codice Cromo.

1.1 Installazione

Cromo si installa per il tuo utente, senza diritti di amministratore, con una sola riga. Su Windows, in PowerShell:

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

Su Linux:

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

Lo script scarica l'archivio dell'ultima versione, ne verifica lo SHA-256 e lo estrae in `~/.cromo` (`%USERPROFILE%\cromo` su Windows). Aggiunge `~/.cromo/current/bin` al `PATH` e, se trova Visual Studio Code, installa l'estensione di Cromo. Apri un nuovo terminale, poi verifica l'installazione:

```
cromo version
```

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

Tutto qui. `cromo` contiene il compilatore e il linker: non ha bisogno di un compilatore C, di Visual Studio né del Windows SDK, e i programmi che produce non dipendono dal runtime C del sistema.

Nota: l'archivio si può installare anche a mano: scaricalo da cromolang.org, estrailo dove preferisci e aggiungi la sua cartella `bin` al `PATH`. `cromo` trova la libreria nella cartella `lib` accanto a `bin`.

1.2 Hello, world!

Crea una cartella per i tuoi programmi e, al suo interno, un file chiamato `hello.cro`:

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

Poi esegui:

```
cromo run hello.cro
```

```
Hello, world!
```

Vediamo il programma da vicino:

- `i32 main()` dichiara la funzione `main`, da cui parte ogni programma. Il tipo viene prima del nome, come in C: `main` restituisce un `i32`, un intero a 32 bit, che diventa il codice di uscita del programma.
- `println` stampa una riga. Il suo primo argomento è una stringa di formato, che può contenere `{}` al posto dei valori che la seguono: `println("{} + {} = {}", a, b, a + b)`.
- `ret` esce da una funzione. `Cromo` scrive `ret` dove il C scrive `return`.
- Non ci sono punti e virgola: un'istruzione finisce alla fine della sua riga.

È ammesso anche un `main` che non restituisce nulla, e in quel caso il codice di uscita è 0:

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

```
Hello again!
```

1.3 Produrre un eseguibile

`cromo run` compila il programma in una cartella temporanea, lo esegue e lo cancella. Per tenere l'eseguibile, usa `cromo build`:

```
cromo build hello.cro          # scrivi hello.exe su Windows, hello su Linux  
./hello
```

Per impostazione predefinita `cromo` produce un eseguibile di **debug**: è ottimizzato, ma controlla l'overflow di ogni operazione sugli interi e contiene le informazioni che servono a un debugger. Con `--release`, in caso di overflow gli interi ricominciano dall'estremo opposto, come in C, e l'eseguibile è più piccolo:

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

L'eseguibile funziona su qualsiasi macchina con lo stesso sistema, senza installare nulla. `cromo` può anche produrre un eseguibile per l'altro sistema:

```
cromo build --target linux hello.cro    # su Windows: un eseguibile per Linux  
cromo build --target windows hello.cro  # su Linux: hello.exe per Windows
```

1.4 Un progetto

Un progetto è una cartella con un file `main.cro`. Se non indichi un nome di file, `cromo build`, `cromo run` e `cromo test` usano il `main.cro` della cartella corrente, e l'eseguibile prende il nome della cartella:

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

Gli altri file della cartella sono moduli, che `main.cro` importa per nome: li spiega il capitolo 11.

1.5 Formattazione

Il codice Cromo ha un solo stile, e `cromo fmt` lo applica: quattro spazi di rientro, la graffa aperta sulla stessa riga, uno spazio attorno agli operatori. Non ci sono opzioni su cui discutere:

```
cromo fmt                # tutti i file .cro di questa cartella e delle sue
sottocartelle
cromo fmt hello.cro      # un solo file
cromo fmt --check        # elenca soltanto i file non formattati
```

1.6 Visual Studio Code

L'estensione di Cromo, che l'installazione aggiunge a Visual Studio Code, colora il codice e usa `cromo lsp`, il language server di Cromo, per mostrare:

- gli errori mentre scrivi, senza compilare a mano;
- il tipo e la dichiarazione di un nome, quando ci passi sopra con il mouse;
- la definizione di un nome, con F12;
- i membri di un valore, dopo un punto;
- la formattazione del file, al salvataggio.

Qualsiasi editor che supporti il Language Server Protocol può usare `cromo lsp` allo stesso modo.

1.7 Riepilogo

- `cromo run file.cro` compila ed esegue un programma; `cromo build` produce l'eseguibile.
- Un programma parte da `main`, che restituisce il suo codice di uscita.

- Il tipo viene prima del nome; `ret` esce dalla funzione; le istruzioni finiscono a fine riga.
- `chroma fmt` formatta il codice, e l'estensione per Visual Studio Code mostra gli errori mentre scrivi.

Nel prossimo capitolo scriverai un programma completo, prima ancora di conoscerne i dettagli.

2

Un primo programma: contare le parole

Prima dei dettagli, un programma completo: legge un file di testo e stampa le parole che vi compaiono più spesso. Mostra al lavoro funzioni, cicli, stringhe, liste, mappe e struct; i capitoli successivi li spiegano uno per uno.

2.1 Il programma

Ecco il testo di cui conteremo le parole:

```
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.
```

Ed ecco il cuore del programma, `words.cro`: il suo `main`. Le funzioni che chiama stanno al posto di `// ...`, e il resto del capitolo le mostra una alla volta; sul sito, *Mostra tutto* apre il file intero.

```
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
}

```

Eseguiilo passando il nome del file dopo `--` :

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

```

5  on
4  the
3  and
2  cat
2  city

```

2.2 Come funziona

Importare un modulo

```
import std.fs
```

La libreria standard è fatta di un nucleo, che ogni programma vede, e di moduli, che un programma importa. `std.fs` lavora con i file: `fs.readText(path)` restituisce l'intero contenuto di un file come `string`. Liste, mappe e stringhe stanno nel nucleo e non serve importarle.

Funzioni e tipi

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

Una funzione si scrive come in C: il tipo del risultato, il nome e i parametri, ciascuno con il tipo prima del nome. `u8` è un byte senza segno; `'a'` è un carattere, che in Cromo è un byte. Una `string` è testo in UTF-8, quindi una lettera come `è` è fatta di due byte sopra 127, che il programma conta come lettere.

Cicli

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

Cromo ha un solo ciclo, `for`, con le forme del C e qualcuna in più: qui c'è quella classica, con un `i64` che va da 0 alla lunghezza del testo. `text.len()` è la lunghezza in byte e `text[i]` è il byte in posizione `i`. Lunghezze e posizioni sono sempre `i64`. Un indice fuori dal testo ferma il programma con un errore, invece di leggere memoria che non c'è.

Liste

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

`list<string>` è una lista di stringhe che cresce secondo necessità. Dichiarata senza valore, è vuota: in Cromo ogni variabile ha un valore, e quello predefinito di una lista è la lista vuota. `ret found` consegna la lista al chiamante senza copiarla.

Struct

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

Una struct raggruppa dei valori, come in C. `Count c` ne dichiara una con il suo valore predefinito, cioè `word` vuota e `times` uguale a 0; poi i campi vengono assegnati uno per uno.

Possesso: `ref`, `move` e `copy()`

Tre parole del programma meritano attenzione, perché sono il cuore di Cromo:

- `sort(ref counts)`: la funzione **modifica** la lista, e la chiamata lo dice con `ref`, come fa la dichiarazione `void sort(ref list<Count> counts)`.
- `counts.add(move c)`: la struct si **sposta** nella lista. Dopo il `move`, `c` non si può più usare: il suo contenuto ora appartiene alla lista.
- `c.word = word.copy()`: una stringa non viene mai copiata senza dirlo. `word` appartiene alla mappa, quindi il programma ne chiede una copia.

Senza `ref`, un parametro è in sola lettura, come `text` in `words(string text)`: la funzione può leggere la stringa ma non cambiarla, e la stringa non viene copiata. Il capitolo 4 spiega queste regole, che sono il nucleo del linguaggio.

Mappe

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

Una `map` associa chiavi a valori, come una tabella hash. `seen[word]` legge o scrive il valore di una chiave; `seen[word] += 1`, se la chiave manca, prima la aggiunge con il valore 0, come fa `operator[]` in C++. Un `for` con due variabili visita ogni chiave insieme al suo valore, senza un ordine particolare: per questo il programma ordina i conteggi.

Stampa

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

`{:>3}` stampa un valore allineato a destra su tre caratteri. I formati sono quelli di `std::format` del C++ e di Python: il capitolo 7 li elenca.

Errori

Se il file non esiste, `fs.readText` lancia un'eccezione. Nessuno la cattura, quindi il programma si ferma e stampa che cosa è successo, con le chiamate che hanno portato fin lì (qui abbreviate):

```
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)
```

Il capitolo 8 mostra come catturare le eccezioni con `try` e `catch`.

2.3 Riepilogo

In poche righe il programma ha usato gran parte di ciò che incontrerai in questa guida: funzioni, il ciclo `for`, stringhe, liste, mappe, una struct e i tre modi di passare un valore: in lettura, con `ref` e con `move`. Ora vediamo il linguaggio dal principio.

3

Concetti comuni

Questo capitolo presenta ciò che ogni linguaggio ha: variabili, tipi, funzioni, commenti e controllo del flusso. Un programmatore C o C++ ne riconoscerà la maggior parte; le differenze sono poche, ma contano.

3.1 Variabili

Una variabile si dichiara con il tipo prima del nome, sempre:

```
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
```

Non c'è inferenza di tipo: il tipo si scrive anche quando il valore basterebbe a dedurlo. Il codice è più lungo di qualche carattere e più facile da leggere, perché hai sempre davanti il tipo di ogni variabile.

Ogni variabile ha un valore

Una variabile dichiarata senza valore prende il **valore predefinito** del suo tipo: 0 per i numeri, `false` per `bool`, la stringa vuota e la lista vuota.

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

[0] [0] [] [0]

In Cromo non esistono variabili non inizializzate, e quindi nemmeno il comportamento indefinito che ne deriva.

Costanti

Le variabili possono cambiare. `const` ne rende una immutabile:

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

error: 'PI' is const

Una variabile `const` deve ricevere il suo valore nel punto in cui viene dichiarata.

3.2 Tipi

Interi

Tipo	Dimensione	Intervallo
<code>i8</code> , <code>u8</code>	8 bit	da -128 a 127, da 0 a 255
<code>i16</code> , <code>u16</code>	16 bit	da -32 768 a 32 767, da 0 a 65 535
<code>i32</code> , <code>u32</code>	32 bit	circa $\pm 2,1$ miliardi, da 0 a 4,2 miliardi
<code>i64</code> , <code>u64</code>	64 bit	circa $\pm 9,2 \times 10^{18}$, da 0 a $1,8 \times 10^{19}$

Ogni tipo ha la stessa dimensione su tutti i sistemi: non c'è un `int` la cui dimensione dipende dalla macchina, né un `size_t`. Lunghezze e posizioni sono `i64`.

I literali interi si possono scrivere in decimale, esadecimale e binario, con `_` per separare le cifre: `1_000_000`, `0xFF`, `0b1010`.

Overflow

Un'operazione tra interi il cui risultato non sta nel suo tipo è un errore. In una build di debug, quella predefinita, il programma si ferma:

```
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)
```

Un *panic* ferma il programma, stampa dove è avvenuto e le chiamate che hanno portato fin lì, e restituisce il codice di uscita 101. In una build di release, con `--release`, gli interi invece ricominciano dall'estremo opposto, come in C: il controllo ha un piccolo costo, e un programma già testato può farne a meno. La divisione per zero è sempre un panic.

Virgola mobile, booleani e caratteri

- `f32` e `f64` sono i numeri in virgola mobile dello standard IEEE 754: `3.14`, `1e-9`.
- `bool` vale `true` o `false`.
- `char` è un byte, sinonimo di `u8`. Un letterale carattere si scrive tra apici singoli e deve essere ASCII: `'a'`, `'\n'`, `'\x41'`. Stampato con `{}`, un `char` appare come numero.

Stringhe

`string` è testo in UTF-8. La sua lunghezza è in byte, e `s[i]` è un 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
```

Il capitolo 7 mostra le funzioni sul testo: cercare, dividere, sostituire.

Conversioni

Un numero viene convertito implicitamente solo quando non si può perdere informazione: un `i32` diventa da solo un `i64` o un `f64`. In tutti gli altri casi la conversione si scrive come una chiamata con il nome del tipo:

```
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)` scarta la parte frazionaria; `u8(wide)` tiene gli 8 bit più bassi. `i32("42")` legge un numero da un testo, e lancia un'eccezione se il testo non è un numero.

I letterali prendono il tipo dal contesto

Un letterale non ha un tipo suo: prende quello richiesto dal punto in cui si trova. In `i64 total = apples + 10`, `10` è un `i32` come `apples`. Dove nulla indica il tipo, il compilatore chiede di specificarlo:

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

error: the type of this expression cannot be deduced

Basta scrivere `i32(10) / 3`, o usare una variabile, per risolvere.

3.3 Operatori

Gli operatori sono quelli del C, con una differenza nella precedenza: gli operatori bit a bit legano più dei confronti, quindi `flags & MASK == 0` significa `(flags & MASK) == 0`, come ci si aspetterebbe. Dal più forte:

Operatori	Significato
<code>*</code> <code>/</code> <code>%</code>	moltiplicazione, divisione, resto
<code>+</code> <code>-</code>	addizione, sottrazione
<code><<</code> <code>>></code>	shift
<code>&</code>	and bit a bit
<code>^</code>	or esclusivo bit a bit
<code> </code>	or bit a bit
<code>==</code> <code>!=</code> <code><</code> <code><=</code> <code>></code> <code>>=</code>	confronti
<code>&&</code>	and logico
<code> </code>	or logico

`!` nega un `bool` e inverte i bit di un intero. `i++` e `i--` esistono, ma solo come istruzioni a sé stanti, mai dentro un'espressione: `a[i++]` non è Croma.

3.4 Funzioni

```
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!
```

Una funzione che restituisce un valore deve arrivare a un `ret` su ogni percorso, altrimenti il programma non compila. Una funzione `void` può terminare senza `ret`. L'ordine delle funzioni in un file non conta: `main` può venire per prima e chiamare funzioni scritte dopo.

Una funzione ha un solo risultato. Quando potrebbe non avere nulla da dare, restituisce un valore convenzionale, lancia un'eccezione, oppure restituisce un `bool` e scrive il risultato in un parametro passato con `ref`, come spiega il capitolo 4. Non esiste `null`.

3.5 Commenti

```
// un commento fino alla fine della riga  
/* un commento  
   su più righe */
```

3.6 Istruzioni e righe

Un'istruzione finisce alla fine della sua riga, senza punto e virgola. Prosegue sulla riga successiva quando la riga termina con un operatore, una virgola o una parentesi aperta:

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

6

C'è un'istruzione per riga.

3.7 Condizioni

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

mild

Le graffe sono sempre obbligatorie, e la condizione deve essere un `bool`: `if (count)` è un errore, `if (count != 0)` è corretto.

3.8 Cicli

Cromo ha un solo ciclo, `for`, in quattro forme:

```

i32 main() {
    // La forma classica del C.
    for (i32 i = 0; i < 3; i++) {
        print("{} ", i)
    }
    println("")

    // Solo una condizione, dove il C usa while.
    i32 n = 1
    for (n < 100) {
        n *= 3
    }
    println!("{}", n)

    // Senza niente, fino a un break.
    for {
        n -= 50
        if (n < 0) {
            break
        }
    }
    println!("{}", n)

    // Sugli elementi di una lista.
    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` esce dal ciclo e `continue` passa al giro successivo. Non esistono `while`, `do` né `goto`.

3.9 switch

`switch` sceglie tra interi o stringhe. Non c'è fall-through: ogni `case` ha il suo blocco, e un `case` può elencare più valori. `default` prende i valori non elencati:

```
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` dà il meglio di sé con gli enum, come mostra il capitolo 6: lì il compilatore controlla che ogni caso sia gestito.

3.10 Riepilogo

- Il tipo viene prima del nome; ogni variabile ha un valore; `const` la rende immutabile.
- Gli interi hanno una dimensione fissa; l'overflow e la divisione per zero fermano il programma.
- Le conversioni che possono perdere informazione vanno scritte: `i32(x)`.
- Le funzioni restituiscono il risultato con `ret`; non esiste `null`.
- `if` vuole le graffe e un `bool`; `for` è l'unico ciclo; `switch` non ha fall-through.

4

Possesso

Il possesso è il modo in cui Croma gestisce la memoria e ogni altra risorsa senza garbage collector e senza puntatori. Le regole sono poche, il compilatore le controlla e, una volta imparate, rendono visibile il costo di ogni riga.

4.1 Ogni valore ha un solo proprietario

Un valore appartiene alla variabile, al campo o all'elemento di lista che lo contiene: il suo **proprietario**. Quando il proprietario finisce, il valore finisce con lui. Per una variabile locale questo succede alla fine del blocco in cui è dichiarata, nell'ordine inverso delle dichiarazioni.

Una struct può stabilire che cosa fare quando finisce, con un **distruttore**: un metodo che si chiama `~` seguito dal nome della struct. Il programma qui sotto ne usa uno per mostrare quando finisce ogni valore:

```
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` finisce quando `inner` termina; `d` e `a` alla fine di `main`, a partire dall'ultima dichiarata. È il RAI, come in C++: una struct che possiede un file lo chiude nel suo distruttore, una che possiede una connessione chiude la connessione, e il codice che le usa non può dimenticarsene, nemmeno quando lo attraversa un'eccezione. Una `string` e una `list` liberano la loro memoria allo stesso modo.

4.2 I valori semplici si copiano

I numeri, `bool` e le struct fatte solo di questi sono tipi **semplici**: si copiano, come 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 I valori che possiedono risorse si spostano

Una `string`, una `list`, una `map` e le struct che le contengono, o che hanno un distruttore, possiedono qualcosa che una copia dovrebbe duplicare. Croma non lo duplica mai senza che tu lo chiedi. Assegnarne una a un'altra variabile non compila:

```

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

```

error: values of type string are not copied implicitly

Ci sono due modi per dire che cosa intendi. Con `move` il valore passa al nuovo proprietario, senza copiare nulla, e la vecchia variabile non si può più usare:

```

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

```

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

Con `copy()` il valore viene duplicato ed entrambe le variabili restano utilizzabili:

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

```
hello hello!
```

Per una struct il compilatore genera `copy()` da sé, quando tutti i suoi campi si possono copiare e la struct non ha un distruttore proprio.

Il compilatore segue ogni percorso della funzione: anche una variabile spostata in un solo ramo di un `if` non si può più usare dopo l'`if`.

Temporanei e valori restituiti

Un valore senza nome, come il risultato di una chiamata o un letterale, si sposta senza bisogno di scrivere `move`. Lo stesso vale per una variabile locale restituita con `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
```

Qui non si copia nulla: la lista costruita in `names` diventa `all`.

Spostare un campo

Spostare un campo, o un elemento di una lista, lascia al suo posto il valore predefinito, quindi la struct resta utilizzabile:

```
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 Passare valori alle funzioni

Un parametro si passa in uno di tre modi, e la **chiamata ripete la parola della dichiarazione**, così basta leggere una chiamata per sapere che cosa succede ai suoi argomenti:

Modo	Dichiarazione	Chiamata	La funzione
leggere	<code>list<i32> xs</code>	<code>f(numbers)</code>	legge il valore; il chiamante lo conserva
modificare	<code>ref list<i32> xs</code>	<code>f(ref numbers)</code>	modifica il valore del chiamante
prendere	<code>move list<i32> xs</code>	<code>f(move numbers)</code>	ne diventa proprietaria; il chiamante lo perde

Leggere

Un parametro senza parola è in sola lettura. La funzione non può modificarlo:

```
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

Un parametro in lettura non viene mai copiato: un valore grande si passa per indirizzo, uno piccolo, fino a 16 byte, per copia, perché così è più veloce. Sceglie il compilatore, e può farlo perché il valore non può cambiare durante la chiamata. In termini di C++, un parametro in lettura è un `const T&` che non devi mai scrivere.

Modificare: `ref`

Con `ref` la funzione lavora sul valore del chiamante. Anche la chiamata scrive `ref`:

```
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

Dimenticare il `ref` nella chiamata è un errore, non una copia silenziosa:

```

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 è anche il modo in cui una funzione restituisce più di un risultato:

```

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

Prendere: **move**

Un parametro **move** riceve il possesso del valore. La funzione può tenerlo, per esempio in una lista, senza copiarlo:

```

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 I riferimenti vivono solo durante una chiamata

`ref` esiste solo per i parametri. Non ci sono riferimenti nelle variabili, nei campi o nei valori restituiti: un riferimento non può sopravvivere all'oggetto a cui si riferisce, perché finisce con la chiamata. Questa sola regola sostituisce i lifetime e il borrow checker di altri linguaggi.

Quando un programma deve puntare a un altro oggetto, come in un grafo, tiene un indice in una lista al posto di un puntatore.

4.6 Esclusività

Durante una chiamata, un oggetto passato con `ref` non può essere raggiunto attraverso nessun altro argomento della stessa chiamata. Altrimenti una funzione potrebbe vedersi cambiare un valore sotto i piedi:

```

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

Quando il compilatore non può saperlo, come in `swap(ref v[i], ref v[j])`, il programma lo controlla durante l'esecuzione, e una chiamata con `i` uguale a `j` è un panic. La stessa regola permette al compilatore di dire a LLVM che due parametri `ref` non si sovrappongono mai, il che rende il codice più veloce, come `restrict` in C.

4.7 const

Una variabile `const` non si può modificare, e `const` è profondo: anche le stringhe e le liste dentro un valore `const` sono `const`. Un metodo marcato `const` non modifica il suo oggetto, come mostra il capitolo 5; su un parametro in lettura si possono chiamare solo metodi `const`.

4.8 Riepilogo

- Ogni valore ha un solo proprietario e finisce con lui: i distruttori rilasciano le risorse.
- I valori semplici si copiano; gli altri si spostano con `move` o si duplicano con `copy()`.
- Un parametro viene letto, modificato con `ref` o preso con `move`, e la chiamata dice quale dei tre.
- I riferimenti esistono solo durante una chiamata: nessun lifetime da scrivere, nessun puntatore pendente.
- Due argomenti di una chiamata che li modifica non sono mai lo stesso oggetto.

5

Struct

Una struct raggruppa sotto un unico nome dei valori correlati, come in C. In Croma può avere anche un costruttore, dei metodi e un distruttore, come una classe del C++, ma non c'è ereditarietà: ciò che tipi diversi hanno in comune si esprime con le interfacce, nel capitolo 9.

5.1 Definire una 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` dichiara un rettangolo con il suo valore predefinito: ogni campo ha il valore predefinito del suo tipo, qui 0. Un campo può dichiarare un proprio valore iniziale:

```

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
```

Il valore iniziale può essere un'espressione qualsiasi, e viene calcolato ogni volta che si crea un `Settings` predefinito.

5.2 Costruttori

Un costruttore si scrive come in C++, con il nome della struct e senza tipo. Assegna i campi, che si usano con il loro nome: non esiste `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
```

Una struct ha al massimo un costruttore. Un parametro non può avere il nome di un campo, così un nome dentro la struct indica sempre una cosa sola: `width = w`, mai `width = width`.

Quando un costruttore non riesce a fare il suo lavoro, per esempio perché un file non si apre, lancia un'eccezione: l'oggetto allora non esiste, e i campi già costruiti vengono distrutti.

Una struct senza costruttore si può comunque creare come valore, con `Rectangle()`: è il suo valore predefinito. Una variabile dichiarata senza valore riceve sempre il valore predefinito, anche quando la struct ha un costruttore: il costruttore viene eseguito solo quando lo si chiama.

5.3 Metodi

Le funzioni scritte dentro la struct sono i suoi metodi. Vedono i campi per nome e possono chiamare gli altri metodi per nome:

```
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
```

Metodi const

`const` dopo i parametri, come in C++, dice che il metodo non modifica l'oggetto. Solo i metodi `const` si possono chiamare su un valore che non può essere modificato: un parametro in lettura, una variabile `const`, l'elemento di un `for` su una lista.

```
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
```

Senza puntatori, `const` è profondo: un metodo `const` non può modificare nemmeno le stringhe e le liste contenute nell'oggetto.

5.4 Distruttori

Il distruttore, `~Name()`, viene eseguito quando l'oggetto finisce: alla fine del suo blocco, quando viene rimosso da una lista, quando finisce la struct che lo contiene. Rilascia ciò che l'oggetto detiene. Dopo il distruttore, i campi vengono distrutti da soli, a partire dall'ultimo dichiarato, quindi un distruttore si occupa solo di ciò che i campi non sanno rilasciare da soli: un handle di sistema, un file temporaneo, un messaggio da stampare.

```
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
```

Un distruttore non si chiama mai a mano, e non può lasciar uscire un'eccezione: il compilatore lo controlla.

Una struct con un distruttore non è un tipo semplice: viene spostata, e copiata solo se scrive un proprio metodo `copy()`.

5.5 Stampare una struct

`{:?}` stampa qualsiasi valore in una forma pensata per il debug, con i nomi dei campi. Per stampare una struct con `{}`, dalle un metodo `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` funziona come `println`, ma restituisce il testo invece di stamparlo.

5.6 Struct dentro altre struct

Una struct può contenere altre struct, liste di struct, mappe: tutto ciò che contiene viene distrutto insieme a lei. Non può contenere direttamente se stessa, perché avrebbe una dimensione infinita, ma può farlo attraverso una lista:

```
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 Riepilogo

- Una struct ha dei campi, al massimo un costruttore, dei metodi e un distruttore; al suo interno, i campi si usano per nome.
- Ogni campo ha un valore predefinito, oppure il valore iniziale che dichiara.
- I metodi `const` non modificano l'oggetto, e sono gli unici che si possono chiamare su un valore in sola lettura.
- Il distruttore rilascia ciò che l'oggetto detiene, in automatico, al momento giusto.
- `text()` permette di stampare una struct con `{}`; `{:?}` stampa qualsiasi valore per il debug.

6

Enum e switch

Un enum è un tipo il cui valore è uno tra i casi di un elenco. In Cromo un caso può portare con sé dei dati, quindi un enum può dire non solo *quale* caso è un valore, ma anche *che cosa* lo accompagna. Insieme a `switch`, che il compilatore controlla rispetto a ogni caso, gli enum sono il modo più sicuro di rappresentare delle alternative.

6.1 Casi senza dati

```
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

Un valore si scrive con il nome dell'enum, `Direction.North`; dentro uno `switch` sull'enum basta il nome del caso. Il valore predefinito di un enum è il suo primo caso.

I casi di un enum senza dati possono avere un valore, che `i32(...)` legge e `Color(n)` riconverte in un caso:

```

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)` provocherebbe un panic, perché nessun caso ha il valore 3.

6.2 Casi con dati

Un caso può portare con sé dei valori, scritti come i parametri di una funzione:

```

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)` i dati del caso ricevono un nome, in sola lettura e senza copia. Per ignorare i dati di un caso basta omettere le parentesi.

6.3 switch copre ogni caso

Su un enum, `switch` deve gestire ogni caso, oppure avere un `default`. Dimenticarne uno è un errore di compilazione, non un bug da scoprire più tardi:

```
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'

Quando si aggiunge un caso a un enum, il compilatore elenca tutti gli `switch` che devono tenerne conto.

Uno `switch` non ha fall-through: ogni case esegue il proprio blocco e lo `switch` termina. Più casi senza dati possono condividere un blocco: `case North, South { ... }`. Un `break` dentro un case è un errore, perché in C uscirebbe dallo `switch`, mentre in C++ uscirebbe dal ciclo che lo racchiude.

6.4 Modificare i dati di un caso

Con `ref`, i dati di un caso si possono modificare:

```
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 Metodi ed enum generici

Un enum può avere dei metodi, come una struct. Dentro un metodo, `switch` senza valore lavora sull'enum stesso:

```

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
```

Un enum può avere anche dei parametri di tipo, come le struct generiche del capitolo 9. Questo contiene un risultato oppure il motivo di un fallimento:

```
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 Riepilogo

- Un enum elenca dei casi; un caso può portare con sé dei dati, scritti come parametri.
- `switch` su un enum deve coprire ogni caso o avere `default`, e il compilatore lo controlla.

- `case Circle(f64 r)` dà un nome ai dati di un caso; con `ref` si possono modificare.
- Gli enum hanno metodi e parametri di tipo, come le struct; il valore predefinito è il primo caso.

7

Liste, mappe e testo

Il nucleo della libreria, che ogni programma vede senza importarlo, contiene i contenitori di cui quasi ogni programma ha bisogno: `list`, gli array di dimensione fissa, `map` e `string`. Ognuno possiede i propri elementi e li libera quando finisce.

7.1 Liste

`list<T>` è una sequenza di valori di tipo `T` che cresce quando serve, come `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]` crea una lista; una lista vuota è `list<i64>()`, o semplicemente una variabile senza valore.
- `add`, `insert(i, x)`, `remove(i)`, `clear()` e `reserve(n)` modificano la lista; `len()` ne restituisce la lunghezza.
- `numbers[i]` è l'elemento stesso: si può leggere, assegnare, passare con `ref` e spostare. Un indice fuori dalla lista è un panic.
- `for (ref i64 n in numbers)` visita gli elementi e può modificarli; senza `ref` sono in sola lettura.

Mentre un `for` visita una lista, la lista non può cambiare lunghezza: aggiungere un elemento nel ciclo sposterebbe gli elementi in memoria sotto i piedi del ciclo. Il compilatore lo rifiuta.

map e filter

`map` crea una nuova lista trasformando ogni elemento, e `filter` tiene gli elementi che superano un test. Entrambi prendono una funzione, qui una funzione anonima, spiegata nel capitolo 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 Array di dimensione fissa

Un array ha una dimensione fissata nel suo tipo, e vive dove si trova la sua variabile, senza memoria sullo heap:

```
i32 main() {
    i32[4] scores
    scores[1] = 7
    f64[2][3] grid
    grid[1][2] = 0.5
    println("{:?}", 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]` sono due array di tre elementi. Un array è un valore: viene copiato quando i suoi elementi sono semplici, spostato altrimenti.

7.3 Mappe

`map<K, V>` associa chiavi a valori, con una tabella hash:

```
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` e `m[k]++` aggiungono una chiave mancante, con il valore predefinito, prima di modificarne il valore, come `operator[]` del C++. Leggere una chiave mancante in qualsiasi altro modo è un panic: chiedi prima con `contains`.
- `for (K key, V value in m)` visita la mappa senza un ordine preciso; con `ref V value` può modificare i valori.
- Le chiavi possono essere numeri, `bool`, `string` e le struct che implementano l'interfaccia `Key`, con `u64 hash() const` e `bool equals(Self other) const`.

7.4 Stringhe

Una `string` è testo in UTF-8. Possiede i propri byte, e si sposta o si copia come una lista:

```
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
```

Le funzioni sul testo sono metodi di `string`: `find`, `findFrom`, `findLast`, `contains`, `startsWith`, `endsWith`, `split`, `trim`, `trimStart`, `trimEnd`, `replace`, `upper`, `lower`, `repeat`, più la funzione `join(parts, separator)`. Le posizioni sono in byte, e un testo non trovato dà -1. `upper` e `lower` cambiano solo le lettere ASCII.

Stringhe e byte

La lunghezza di una stringa è in byte, e una lettera fuori dall'ASCII ne occupa più di uno:

```
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)` prende i byte da `from` a `to` escluso. `string(bytes)` controlla che i byte siano UTF-8 valido, e altrimenti lancia un'eccezione.

Le stringhe si confrontano con `==`, `!=`, `<`, `<=`, `>`, `>=`, byte per byte, e si concatenano con `+` e `+=`.

7.5 Formattazione

`println`, `print`, `eprintln` (sullo standard error) e `format` (che restituisce una `string`) condividono il formato di `std::format` del C++ e di Python. Ogni `{}` prende il valore successivo; dentro le graffe, dopo `:`, una specifica dice come scriverlo:

```
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
```

Specifica	Significato
< ^ >	allinea a sinistra, al centro, a destra, nella larghezza che segue
un numero	la larghezza minima; se preceduto da 0, riempita di zeri
.n	le cifre decimali, o la lunghezza massima di un testo
+	il segno anche per i numeri positivi
x X b o	esadecimale, binario, ottale; # aggiunge il prefisso
f e g	virgola fissa, notazione esponenziale, la più breve delle due
?	la forma per il debug, per qualsiasi valore

La stringa di formato deve essere un letterale: il compilatore controlla il numero e i tipi dei valori in base a essa.

7.6 Riepilogo

- `list<T>` cresce quando serve; `[a, b]` ne crea una; `for` la visita, con `ref` per modificarne gli elementi.
- `T[n]` è un array di dimensione fissa, senza heap.
- `map<K, V>` è una tabella hash; `m[k] += v` aggiunge una chiave mancante; l'ordine non è definito.
- `string` è UTF-8, misurata in byte, con metodi per cercare, dividere e sostituire.
- `format` e `println` usano il formato di `std::format`, controllato dal compilatore.

8

Errori

Cromo distingue due tipi di errori:

- **Errori che un programma si aspetta**, come un file che non esiste o una connessione che cade: sono **eccezioni**, che un chiamante può catturare e gestire.
- **Errori di programmazione**, come un indice fuori da una lista o l'overflow di un intero: sono **panic**, che fermano il programma, perché un programma che ha violato le proprie regole non può sapere che cosa fare dopo.

8.1 Lanciare e catturare eccezioni

```
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` interrompe la funzione e risale attraverso i chiamanti fino a un `try` il cui `catch` accetta l'eccezione. `Error` è un'eccezione già pronta con un messaggio; `catch (Exception e)` accetta qualsiasi eccezione, e `e.message()` ne restituisce il testo. Le eccezioni non si dichiarano nella firma di una funzione.

8.2 Tipi di eccezione

Un'eccezione è una struct che implementa l'interfaccia `Exception`, che ha un solo metodo, `string message() const`. Un tipo dedicato permette a un chiamante di riconoscere l'errore e di leggerne i campi:

```
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

```

I blocchi `catch` vengono provati in ordine; il primo che accetta il tipo dell'eccezione la gestisce. L'eccezione di un `catch` è in sola lettura.

8.3 finally

`finally` viene eseguito in qualunque modo finisca il `try`: normalmente, con `ret`, con `break` o `continue`, oppure con un'eccezione, catturata o no:

```

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 Eccezioni e distruttori

Mentre un'eccezione risale attraverso le funzioni, ogni oggetto degli ambiti che abbandona viene distrutto, come alla fine di un blocco normale. È questo che rende affidabile il RAI: una risorsa detenuta da un oggetto viene rilasciata anche quando qualcosa va storto.

```
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
```

Un distruttore, così come un `finally`, non può lasciar uscire un'eccezione: il compilatore richiede che la catturi al proprio interno.

8.5 Rilanciare un'eccezione

`throw` da solo, dentro un `catch`, rilancia la stessa eccezione, per esempio dopo averla scritta in un 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 Eccezioni non catturate

Un'eccezione che esce da `main` ferma il programma con codice di uscita 1, e stampa il messaggio e le chiamate dal `throw` fino a `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)
```

La traccia viene stampata anche nelle build di release, con i nomi, i file e le righe del sorgente. Per farlo, all'eseguibile non servono le informazioni di debug: il compilatore vi scrive dentro una tabella compatta del codice.

8.7 Panic

Un panic è un errore di programmazione: un indice fuori dai limiti, un overflow, una divisione per zero, una chiave mancante in una mappa, uno stack overflow. Non è un'eccezione e non si può catturare: il programma si ferma con codice di uscita 101 e stampa le chiamate che lo hanno portato fin lì.

```
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")` ferma il programma di proposito, quando raggiunge uno stato che dovrebbe essere impossibile.

assert

`assert(condition)` provoca un panic quando la sua condizione è falsa. Il messaggio ripete la condizione così come è scritta e, per un confronto tra numeri, i valori dei due lati:

```
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` funziona in ogni build, anche con `--release`. È anche lo strumento dei test del capitolo 12.

8.8 Come scegliere

- Se il chiamante può fare qualcosa per rimediare all'errore, come chiedere un altro nome di file, lancia un'eccezione.
- Se l'errore significa che il programma è sbagliato, come un indice calcolato male, è un **panic**: correggi il programma.
- Se l'assenza di un risultato è normale, come una parola non trovata in un testo, restituisci un valore convenzionale o un `bool`, come fa `find` con `-l`.

8.9 Riepilogo

- `throw`, `try`, `catch` e `finally` funzionano come in C++, Java e C#; le eccezioni implementano `Exception`.
- I distruttori vengono eseguiti mentre un'eccezione risale, quindi le risorse vengono sempre rilasciate.
- Un'eccezione non catturata, un `panic` e uno `stack overflow` stampano le chiamate che hanno portato fin lì, anche in `release`.
- I `panic` servono per gli errori di programmazione; `assert` controlla ciò che deve essere vero.

9

Interfacce e generici

Spesso tipi diversi condividono un comportamento: un cerchio e un quadrato hanno entrambi un'area. In Croma il comportamento è descritto da un'**interfaccia**, e il codice che funziona con qualsiasi tipo che lo possiede si scrive una volta sola: come funzione che prende l'interfaccia, risolta mentre il programma è in esecuzione, oppure come **generico**, risolto dal compilatore. Non c'è ereditarietà tra struct.

9.1 Interfacce

Un'interfaccia elenca dei metodi senza corpo. Una struct la implementa indicandone il nome dopo i due punti e scrivendo quei metodi:

```
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

```

Un parametro di tipo `Shape` accetta qualsiasi struct che implementa `Shape`, senza copiarla e senza allocare nulla: la struct resta dov'è, e la chiamata passa per una tabella dei suoi metodi, come una chiamata virtuale in C++.

Valori di un'interfaccia

Una variabile, un campo o una lista possono contenere valori di un'interfaccia. Una struct lo diventa spostandosi sullo heap, insieme alla sua tabella dei metodi; quando il valore finisce, la struct viene distrutta:

```

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

Qui `small` e `big` sono valori semplici, quindi la lista ne riceve delle copie; una struct che possiede delle risorse andrebbe aggiunta con `move`. Un valore di un'interfaccia non viene mai copiato, solo spostato, e su di esso si possono chiamare solo i metodi dell'interfaccia. Il suo valore predefinito è vuoto, e chiamare un metodo su un valore vuoto è un panic.

9.2 Generici

Una funzione generica ha dei parametri di tipo, scritti tra `<` e `>` dopo il suo nome. Ogni parametro di tipo è vincolato da un'interfaccia, che dice che cosa la funzione può farci:

```

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` dice che `T` può essere qualsiasi tipo che implementa `Comparable`. In un'interfaccia, `Self` è il tipo che la implementa: `less` confronta una `Version` con un'altra `Version`.
- Gli argomenti di tipo vengono dedotti dagli argomenti della chiamata: `largest(versions)` è `largest<Version>`. Quando non si possono dedurre, si scrivono esplicitamente: `largest<Version>(versions)`.
- Un generico viene controllato una volta sola, dove è scritto: su un `T` si possono chiamare solo i metodi dei suoi vincoli, e gli errori parlano del generico, non dei suoi usi.
- Il compilatore crea una versione della funzione per ogni tipo con cui viene usata, come i template del C++: le chiamate sono dirette, e veloci quanto con una funzione scritta apposta per quel tipo.

Un'interfaccia che usa `Self`, come `Comparable`, può essere solo un vincolo, non il tipo di un parametro o di un valore.

Interfaccia o generico?

	<code>void f(Shape s)</code>	<code>void f<T : Shape>(T s)</code>
La chiamata	attraverso la tabella dei metodi	diretta, spesso inline
Il codice	una sola funzione	una versione per ogni tipo
Tipi diversi in una lista	sì, <code>List<Shape></code>	no, una <code>List<T></code> ha un solo tipo

Usa un'interfaccia per le collezioni di tipi diversi e per il codice che non ha bisogno della massima velocità; usa un generico per gli algoritmi che lavorano su un tipo alla volta.

9.3 Struct generiche

Anche una struct può avere dei parametri di tipo, e lo stesso vale per i suoi metodi:

```
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 }
```

Una struct generica si scrive sempre con i suoi argomenti di tipo: `Stack<string>`, `Pair<string, i64>`. I contenitori della libreria, `list<T>` e `map<K, V>`, sono struct generiche scritte in Cromo allo stesso modo.

9.4 Riepilogo

- Un'interfaccia elenca dei metodi; una struct la implementa con `struct Name : Interface`.
- Un parametro di tipo interfaccia accetta qualsiasi struct che la implementa, senza copiarla.
- I valori di un'interfaccia, come quelli di una `list<Shape>`, possiedono la loro struct sullo heap.
- I generici hanno parametri di tipo vincolati da interfacce, vengono controllati una volta sola e compilati per ogni tipo.
- Non c'è ereditarietà: un comportamento condiviso è un'interfaccia, dei dati condivisi sono un campo.

Funzioni anonime

Una funzione anonima, o *lambda*, è una funzione scritta nel punto in cui serve, come argomento o come valore. Può usare le variabili che ha intorno, e il modo in cui le prende segue le regole del possesso viste nel capitolo 4.

10.1 Scrivere una 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
```

La lambda è `[limit](i64 x) bool { ret x > limit }`:

- tra parentesi quadre, le variabili che **cattura**, qui `limit`;

- i suoi parametri, come quelli di una funzione;
- il tipo che restituisce, dopo i parametri; se manca, la lambda non restituisce nulla;
- il suo corpo.

`func(i64) bool` è il tipo di una funzione che riceve un `i64` e restituisce un `bool`. Un parametro di tipo funzione si chiama proprio come una funzione: `keep(v)`.

10.2 Catture

Una lambda dichiara come usa ogni variabile che cattura, con le stesse parole che si usano per i parametri:

Cattura	La lambda
<code>[limit]</code>	legge la variabile
<code>[ref total]</code>	modifica la variabile
<code>[move name]</code>	prende il valore, che la variabile perde

```
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
```

Senza `ref`, una lambda non può modificare ciò che cattura, e il compilatore te lo segnala.

10.3 Le lambda come valori

Una lambda che cattura per indirizzo, con `ref` oppure leggendo un valore di tipo non semplice, può essere passata solo come argomento: le variabili a cui fa riferimento vivono nella funzione chiamante, e la lambda finisce insieme alla chiamata. Una lambda che cattura solo valori di tipo semplice, o che prende i suoi valori con `move`, è invece un valore: si può tenere in una variabile, in un campo o in una lista, restituire e spostare.

```
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
```

Restituire una lambda che cattura con `ref` non compila: la variabile smetterebbe di esistere al termine della funzione, e la lambda non farebbe più riferimento a niente.

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

i32 main() {
    ret 0
}
```

```
error: this lambda captures by address
```

Un valore di tipo funzione viene spostato, non copiato. Se è vuoto, come un campo mai assegnato, chiamarlo provoca un panic.

10.4 const func

Un parametro `const func(...)` accetta solo una funzione che non modifica ciò che cattura. Più thread possono chiamare una funzione del genere nello stesso momento, in sicurezza: è così che il server HTTP del capitolo 15 chiama lo stesso handler per molte richieste contemporaneamente.

10.5 Riepilogo

- Una lambda è `[captures](parameters) type { body }`; il suo tipo è `func(parameters) type`.
- Le catture usano le stesse parole dei parametri: lettura, `ref`, `move`.
- Una lambda che fa riferimento a variabili del chiamante vive solo per la durata della chiamata; una che possiede ciò che cattura è un valore.
- `const func` è una funzione che si può chiamare da più thread contemporaneamente.

Moduli

Quando un programma cresce oltre un singolo file, il suo codice si divide in **moduli**. Croma non ha file header né preprocessore: un file è un modulo, una cartella è un modulo, e `import` rende visibili i nomi pubblici di un modulo.

11.1 Un file è un modulo

Accanto a `main.cro`, un 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
}

// La radice quadrata con il metodo di Newton, sufficiente per un esempio.
f64 root(f64 v) {
```

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

Il programma lo importa con il suo nome:

```
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` o `cromo run` nella cartella compilano tutti insieme `main.cro`, i moduli che importa e, a loro volta, i moduli importati da questi.

11.2 Visibilità

Tutto ciò che sta in un modulo è privato del modulo, a meno che non sia marcato `public`: le funzioni, le struct, e anche i campi, i metodi e i costruttori di una struct. Qui sopra, `Point` e `distance` sono pubblici; `square`, `root` e il campo `visits` no:

```
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
```

Ciò che una dichiarazione pubblica espone deve essere pubblico a sua volta: una funzione pubblica non può restituire una struct privata. Il distruttore non ha `public`: viene eseguito ovunque l'oggetto finisca la sua vita.

11.3 Usare i nomi

Dopo `import geometry`, i suoi nomi pubblici si usano direttamente: `distance(a, b)`, `Point p`. Il nome del modulo davanti, `geometry.distance(a, b)`, serve solo quando due moduli importati esportano lo stesso nome, oppure per rendere il codice più chiaro. Un nome viene cercato prima nel modulo stesso, poi nei moduli importati e infine nella libreria standard: così un nome aggiunto alla libreria non rompe mai un programma esistente.

Gli import si scrivono all'inizio del file, e ogni file scrive i propri: l'importazione non è transitiva. Due moduli possono importarsi a vicenda.

11.4 Una cartella è un modulo

`import shapes` importa la cartella `shapes`, con tutti i suoi file `.cro`, come un unico modulo:

```
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
}
```

`import a.b` indica il file `a/b.cro`, oppure la cartella `a/b`. I percorsi partono dalla cartella del file principale, così un modulo ha lo stesso nome ovunque venga importato. I file di una cartella condividono i loro nomi privati, come parti di un unico modulo; le sottocartelle sono moduli a sé.

11.5 La libreria standard

Il nucleo della libreria, con `list`, `map`, `string`, `format` ed `Exception`, è visibile ovunque. Gli altri moduli si importano con il prefisso `std`:

Modulo	Contenuto	Capitolo
<code>std.fs</code>	file e cartelle	14
<code>std.os</code>	l'ambiente del processo	14
<code>std.process</code>	eseguire altri programmi	14
<code>std.time</code>	orologi, date, pause	14
<code>std.thread</code>	thread, task, mutex, canali, atomici	13
<code>std.net</code>	TCP, UDP, TLS	15
<code>std.http</code>	client e server HTTP	15
<code>std.websocket</code>	client e server WebSocket	15
<code>std.json</code>	JSON	16
<code>std.crypto</code>	hash, byte casuali, cifratura, firme	15
<code>std.gzip</code>	compressione	15

Dopo `import std.fs`, le funzioni del modulo si scrivono precedute dall'ultima parte del suo nome: `fs.readText(path)`.

11.6 Codice per ogni sistema

Un modulo può avere parti scritte per un solo sistema: accanto a `files.cro`, i file `files_windows.cro` e `files_linux.cro`. Viene usato solo il file del sistema per cui il programma è compilato, e fa parte del modulo, con accesso ai suoi nomi privati. La libreria standard sfrutta questo meccanismo per chiamare il sistema di Windows e quello di Linux dietro le stesse funzioni.

11.7 Pacchetti

Le librerie scritte da altri saranno **pacchetti**: repository Git, identificati dal loro URL, con le versioni date dai loro tag, elencati in un file `cromo.mod` e scaricati da `cromo`. Il loro design è deciso ed è descritto in `docs/language.md`, ma non sono ancora disponibili.

11.8 Riepilogo

- Un file è un modulo; una cartella è un modulo formato dai suoi file; `import` rende visibili i loro nomi pubblici.
- Tutto è privato, a meno che non sia `public`: anche i campi e i metodi di una struct.
- I nomi di un modulo importato si usano direttamente, con il modulo davanti quando due nomi entrano in conflitto.
- `std.x` sono i moduli della libreria standard; i file `_windows` e `_linux` contengono il codice per un solo sistema.

Test

Cromo ha i test integrati: un test è una funzione marcata `[[test]]`, `assert` controlla una condizione, e `cromo test` esegue i test e ne riporta i risultati. Non servono librerie né framework.

12.1 Scrivere un test

Un test è una funzione senza parametri che non restituisce nulla, preceduta dall'attributo `[[test]]`. Passa se termina normalmente, e fallisce con un panic, con un'eccezione che non cattura o con uno 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)
}
```

I test possono stare nel file del codice, oppure in un file accanto il cui nome finisce con `_test.cro`. Quel file fa parte dello stesso modulo, quindi i suoi test vedono anche i nomi privati, e `cromo build` lo esclude dall'eseguibile:

```
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 Eseguire i test

```
main.cro

import stats

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

`cromo test` nella cartella di `main.cro` esegue i test del programma e di tutti i suoi moduli. Il secondo test qui sopra è sbagliato di proposito, perché la media di 1 e 2 è 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
```

- Ogni test gira in un processo tutto suo, tanti alla volta quanti sono i processori: un test che va in crash ferma solo sé stesso.
- L'output di un test viene mostrato solo quando il test fallisce.
- Il codice di uscita di `cromo test` è 1 se un test è fallito, così uno script o un sistema di integrazione continua possono fermarsi.
- I nomi dei test sono preceduti dal loro modulo: `stats.meanOfThree`.

I nomi scritti dopo il file scelgono quali test eseguire: `cromo test main.cro meanOf` esegue i test il cui nome contiene `meanOf`.

12.3 assert

`assert(condition)` ferma il test con un panic quando la condizione è falsa. In un confronto tra numeri o tra `bool`, il messaggio riporta il valore di ciascun lato che non sia un letterale, calcolato una volta sola, come `m is 1.5` qui sopra: il più delle volte basta per capire cosa è andato storto senza ricorrere a un debugger.

`assert` funziona in qualsiasi funzione, non solo nei test, e anche nelle build di release.

12.4 Testare gli errori

Un test che si aspetta un'eccezione la cattura:

```
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 Riepilogo

- `[[test]] void name()` è un test; `assert(condition)` esegue il controllo.
- I test stanno nel codice o in `name_test.cro`, che vede i nomi privati del modulo.
- `cromo test` esegue ogni test in un processo a sé, in parallelo, e riporta i fallimenti con la loro causa.

Thread, task e canali

Cromo può eseguire codice contemporaneamente in due modi: con i **thread** del sistema e con i **task** leggeri, molti dei quali condividono pochi thread. Entrambi si trovano nel modulo `std.thread`, con la stessa interfaccia, e il compilatore controlla ciò che condividono: due thread non scrivono mai gli stessi dati nello stesso momento.

13.1 Thread

Un `thread.Thread` si crea da una lambda, si avvia con `start()` e si attende con `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
```

La lambda legge `values` e modifica `low`, entrambe variabili di `main`. Finché il thread è in esecuzione, il compilatore non permette a `main` di modificare ciò che il thread legge, né di toccare ciò che modifica: qui `main` si limita a leggere `values`, e questo è consentito. Un thread che cattura per indirizzo appartiene al suo blocco: se `join()` non viene chiamato, il blocco lo attende alla propria fine, così le variabili che il thread usa vivono sempre più a lungo di lui.

Un thread che cattura solo con `move` possiede tutto ciò che usa, ed è libero: si può tenere in una lista e può sopravvivere al blocco che lo ha avviato.

Un'eccezione che termina un thread viene rilanciata da `join()`, nel thread che attende.

13.2 Mutex

Quando più thread devono modificare gli stessi dati, i dati vanno messi in un `thread.Mutex<T>`. Si possono raggiungere solo dentro `lock`, che li passa con `ref` a un thread alla volta:

```

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

Non c'è modo di raggiungere l'`i64` senza il lock, e quindi non c'è modo di dimenticarselo. Il `lock` viene rilasciato alla fine della lambda, anche quando la attraversa un'eccezione.

13.3 Task

Un `thread.Task` ha l'interfaccia e le regole di un `thread.Thread`, ma non è un thread del sistema: il runtime esegue molti task su pochi thread, uno per processore. Quando un task aspetta la rete, `time.sleep`, un mutex o `join()`, il suo thread nel frattempo esegue altri task. Il codice è lo stesso che si scrive con i thread: niente `async`, niente `await`, e nessuna funzione di un colore diverso.

Un task costa pochi kilobyte: un programma ne può avere centinaia di migliaia. `thread.Group` ne avvia molti e li attende tutti:

```

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
```

Diecimila thread del sistema richiederebbero gigabyte di stack; diecimila task richiedono qualche decina di megabyte.

- I task di un gruppo possono leggere le variabili dichiarate prima del gruppo, che restano costanti fino alla fine del suo blocco, e prenderne altre con `move`; non possono catturare con `ref`.
- `wait()` attende tutti i task, e rilancia la prima eccezione sollevata da un task. Alla fine del suo blocco, un gruppo attende da solo i propri task.
- Un task che chiama una funzione che blocca il suo thread, come la lettura di un file o una funzione C, tiene occupato quel thread; dopo circa un millisecondo il runtime assegna agli altri task un altro thread.

Usa i task per il lavoro che passa il tempo ad aspettare, come i server e i client di rete; usa i thread per i calcoli lunghi e per il codice C che blocca.

13.4 Canali

Un `thread.Channel<T>` passa valori da un task o da un thread a un altro, in ordine:

```
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)` contiene fino a `n` valori: `send` attende finché il canale è pieno, `receive` finché è vuoto, e un task che attende lascia il suo thread agli altri. Con capacità 0, chi invia attende finché un ricevente non ha preso il valore.
- `close()` chiude il canale: i valori già inviati si possono ancora ricevere, dopodiché `receive()` lancia un'eccezione e `for (T v in channel)` termina.
- `next(ref v)` riceve senza eccezione: restituisce `false` quando il canale è chiuso e vuoto.

13.5 Atomici

`thread.Atomic<T>` contiene un intero o un `bool` che più thread modificano senza lock: `load`, `store`, `add`, `exchange` e `compareExchange`. Le operazioni sono sequenzialmente consistenti, il modello su cui è più semplice ragionare. Un atomico non si può copiare, e i task e i thread che lo usano lo catturano in lettura, come `done` e `letters` qui sopra.

13.6 Riepilogo

- `thread.Thread` è un thread del sistema; `thread.Task` è un task leggero con la stessa interfaccia.
- Il compilatore controlla ciò che thread e task catturano: i dati condivisi che cambiano vanno in un `thread.Mutex`.
- `thread.Group` avvia molti task e li attende; un task che attende lascia il suo thread agli altri.
- `thread.Channel` passa valori tra i task; `thread.Atomic` conta senza lock.

File, processi e tempo

Questo capitolo è una panoramica dei moduli che dialogano con il sistema operativo: `std.fs` per file e cartelle, `std.os` per l'ambiente, `std.process` per gli altri programmi e `std.time` per orologi e date. Funzionano allo stesso modo su Windows e su Linux, e segnalano gli errori con le eccezioni.

14.1 File

```
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
```

Funzione	Cosa fa
<code>readText(path)</code> , <code>readBytes(path)</code>	l'intero contenuto, come <code>string</code> (verificata come UTF-8) o come byte
<code>writeText</code> , <code>writeBytes</code> , <code>appendText</code>	scrive un file, creandolo o sostituendolo, oppure aggiunge in fondo
<code>exists</code> , <code>isFile</code> , <code>isFolder</code> , <code>isLink</code>	dice che cosa è un percorso
<code>size</code> , <code>modified</code>	la dimensione in byte, e l'ora dell'ultima modifica in secondi dal 1970
<code>copy</code> , <code>rename</code> , <code>remove</code>	copia, sposta o elimina un file
<code>createFolder</code> , <code>createFolders</code>	crea una cartella, oppure una cartella insieme a quelle che la contengono
<code>listFolder(path)</code>	i nomi contenuti in una cartella
<code>removeAll(path)</code>	elimina una cartella con tutto ciò che contiene, senza seguire i link
<code>joinPath</code> , <code>parent</code> , <code>fileName</code> , <code>extension</code>	le parti di un percorso

Un file che non si può leggere o scrivere provoca un'eccezione, con un messaggio che dice quale file e perché:

```
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 L'ambiente

`std.os` fornisce il nome del sistema, `"windows"` o `"linux"`, le variabili d'ambiente e la cartella corrente; `os.exit(code)` termina subito il programma. Gli argomenti del programma arrivano a `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]` è il nome del programma, come in C.

14.3 Altri programmi

`process.run` esegue un programma, ne attende la fine e restituisce il suo codice di uscita, il suo output e i suoi errori:

```
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
}
```

Il programma viene cercato nelle cartelle del `PATH`, e gli argomenti vengono passati così come sono, senza una shell che li interpreti. Un codice di uscita diverso da 0 è un valore da controllare; un programma che non riesce a partire provoca un'eccezione.

`process.start` avvia un programma senza attenderlo, e restituisce un `process.Process` con cui comunicare: `write` e `closeInput` per il suo standard input, `read` e `readAll` per il suo output, `wait` per attenderne la fine e `kill` per fermarlo.

14.4 Il tempo

```
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()` legge un orologio che va solo in avanti, adatto alle misure; `elapsed()` e `since()` restituiscono una `Duration`, con `seconds()`, `milliseconds()`, `microseconds()` e il campo `nanoseconds`.
- `utcNow()` e `localNow()` restituiscono un `DateTime`, con i suoi campi dall'anno al nanosecondo, lo scostamento da UTC in minuti, e `text()` nel formato ISO 8601: `2026-10-30T18:21:33+01:00`.
- `sleep`, `msleep` e `usleep` mettono in pausa; in un task, nel frattempo il thread esegue altri task.

14.5 Riepilogo

- `std.fs` legge e scrive file e cartelle; gli errori sono eccezioni con un messaggio chiaro.
- `std.os` fornisce l'ambiente; gli argomenti arrivano in `main(list<string> args)`.
- `std.process` esegue altri programmi, senza passare da una shell.
- `std.time` misura le durate e fornisce la data e l'ora.

Rete e HTTP

La libreria standard ha tutto ciò che serve a un programma di rete: TCP e UDP in `std.net`, TLS per le connessioni cifrate, client e server HTTP in `std.http`, WebSocket in `std.websocket`, e la crittografia e la compressione su cui si appoggiano. Non richiede altre librerie: TLS viene da Mbed TLS, compilato nel runtime di Cromo, così gli eseguibili restano autosufficienti.

Le chiamate bloccano finché non hanno finito, come in C. In un task, una chiamata che aspetta la rete lascia il suo thread agli altri task, così un server scritto nel modo più semplice, con un task per ogni connessione, serve migliaia di client contemporaneamente.

15.1 TCP

Un server si mette in ascolto con `net.listen`, accetta le connessioni con `accept()`, e legge e scrive su ogni `net.Connection`. Un client si connette con `net.connect`. Il programma qui sotto fa entrambe le cose: il suo server gira in un task, e il suo client ci dialoga.

```
import std.net
import std.thread

i32 main() {
    net.Listener listener = net.listen("127.0.0.1", 0) // porta 0: una porta
libera qualsiasi
    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)` attende dei dati e restituisce fino a `most` byte; una lista vuota significa che l'altro capo ha chiuso.
- `timeout(ms)` pone un limite a ogni operazione successiva; un timeout, come qualsiasi errore di rete, è un'eccezione.
- Il distruttore di una connessione la chiude.
- `net.udp(host, port)` apre un socket UDP, con `send` e `receive`; `net.resolve(host)` restituisce gli indirizzi associati a un nome. IPv4 e IPv6 funzionano su entrambi i sistemi.

15.2 TLS

`net.connectTls` apre una connessione cifrata, con gli stessi `read` e `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
}

```

Il certificato del server deve essere firmato da un'autorità di cui il sistema si fida, e riportare il nome dell'host; altrimenti la connessione fallisce con un'eccezione che spiega il motivo, come `the certificate has expired`. Un server usa

`net.ListenTls(host, port, certificate, key)`, con il certificato e la chiave come testi in formato PEM.

15.3 Un server HTTP

`http.Serve` passa ogni richiesta a un'unica funzione, l'handler, che riempie la risposta:

```
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
}
```

Eseguilo e apri `http://localhost:8080/hello/world` in un browser.

- Ogni connessione è servita da un task tutto suo, quindi l'handler gira su più thread contemporaneamente. È una `const func`: ciò che le richieste condividono deve stare in un `thread.Mutex` o in un `thread.Atomic`.
- `req.matches("GET /users/{id}")` controlla il metodo e il percorso, e `req.param("id")` legge la parte tra graffe; `{rest...}` prende il resto del percorso. `req.queryValue("page")` legge la query.
- `res.write`, `res.writeJson` e `res.writeBytes` scrivono il corpo e ne impostano il `Content-Type`; `res.status` vale 200 per impostazione predefinita; `res.headers` contiene gli header.
- Un'eccezione nell'handler produce lo stato 500, e il server va avanti.
- Il server mantiene attive le connessioni tra una richiesta e l'altra, e legge e scrive i corpi a blocchi.

`http.Listen(host, port)` restituisce un `http.Server` senza avviarlo, la cui `port()` è nota prima di `serve(handler)`: è utile con la porta 0, e nei test.

15.4 Un client HTTP

```
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)` e `http.send(request)` seguono i redirect e usano TLS per `https`. Uno stato di errore, come 404, è una risposta come le altre; un guasto della rete è un'eccezione. Il client chiede gzip e decompime il corpo da solo.

Un `http.Client` tiene aperte le connessioni tra richieste allo stesso host, il che rende molto più veloci le richieste numerose, e può essere usato da più task contemporaneamente.

Il server e il client insieme, in un solo programma:

```
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:{}", port))
http.Response missing = client.get(format("http://127.0.0.1:{}",
port))
println("{} {}", ok.status, ok.text())
println!("{}", missing.status)
os.exit(0) // il server andrebbe avanti all'infinito
ret 0
}

```

```

200 Hello, Ada!
404

```

15.5 WebSocket

Un WebSocket tiene aperta una connessione in entrambe le direzioni, per lo scambio di messaggi. Il server lo ottiene da una richiesta con `req.websocket`, e un client ne apre uno con `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()` lancia `websocket.Closed` quando l'altro capo chiude, e questo fa terminare il ciclo qui sopra. `websocket.connect("wss://...")` ne apre uno con TLS.

15.6 Crittografia e compressione

`std.crypto` offre i mattoni di base che servono ai programmi di rete, e `std.gzip` comprime:

```
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
```

- Hash: `md5`, `sha1`, `sha256`, `sha512` e `hmac`; `random(n)` prende i byte dal generatore del sistema.
- `encrypt` e `decrypt` con AES-GCM o ChaCha20-Poly1305: `decrypt` lancia un'eccezione se i dati sono stati alterati.
- Firme Ed25519 e scambio di chiavi X25519; `hex` e `base64` con le rispettive inverse.
- `gzip.decompressUpTo(data, limit)` rifiuta i dati che supererebbero un certo limite una volta decompressi, per i dati che arrivano dalla rete.

15.7 Riepilogo

- `std.net` ha TCP, UDP e TLS, con chiamate bloccanti che, mentre attendono, lasciano il thread agli altri task.
- `http.serve` esegue un handler per ogni richiesta, su molti task contemporaneamente; `http.get` e `http.Client` effettuano le richieste.
- `std.websocket`, `std.crypto` e `std.zip` completano la cassetta degli attrezzi, senza librerie esterne.

JSON è il formato con cui la maggior parte dei servizi si scambia i dati. In Croma la conversione tra una struct e il JSON la scrive il compilatore da sé, la prima volta che serve: non occorre alcuna annotazione sulla struct, nessuno strumento esterno genera codice, e la conversione è veloce quanto quella scritta a mano.

16.1 Da struct a JSON e viceversa

```
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)` scrive un JSON compatto; `json.pretty(value)` lo scrive su più righe, indentato.
- `json.decode<T>(text)` legge un `T`. Un membro assente, o `null`, lascia al suo campo il valore predefinito; un membro che non corrisponde ad alcun campo viene ignorato.
- La conversione funziona per i numeri, `bool`, `string`, le liste, le mappe con chiavi stringa, gli array fissi, gli enum e le struct i cui campi siano a loro volta convertibili. Un campo che non lo è, come una funzione, dà un errore di compilazione che lo indica per nome.

16.2 Nomi e campi da escludere

L'attributo `[[json(...)]]` davanti a un campo ne cambia il nome nel JSON, oppure lo esclude:

```
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 Enum

Un case senza dati si scrive con il suo nome; un case con dati, come un oggetto che ha per unico membro il case stesso:

```
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 Valori di forma qualsiasi

Quando la forma dei dati non è nota in anticipo, `json.parse` restituisce un `json.Value`, un enum con i case `Null`, `Bool`, `Int`, `UInt`, `Float`, `Text`, `Array` e `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}
```

Gli interi e i numeri in virgola mobile restano distinti: un identificatore a 64 bit conserva tutte le sue cifre.

16.5 Errori

Un testo che non è JSON, o un JSON che non corrisponde al tipo, genera un'eccezione che indica dove si trova il problema:

```
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 Riepilogo

- `json.encode`, `json.pretty` e `json.decode<T>` convertono struct, liste, mappe ed enum, con codice scritto dal compilatore.
- `[[json("name")]]` rinomina un campo o un case; `[[json(skip)]]` esclude un campo.
- `json.parse` restituisce un `json.Value` per dati di forma qualsiasi.
- Gli errori indicano dove si trova il problema.

Chiamare il C

Ogni sistema offre i suoi servizi attraverso funzioni C, e molte librerie sono scritte in C. Crome le chiama direttamente, senza un compilatore C e senza file header, dai moduli che lo dichiarano con `import unsafe`. Il resto del programma usa quei moduli attraverso funzioni e struct sicure, così la parte non sicura resta piccola e facile da individuare.

17.1 Dichiarare funzioni C

Un modulo che scrive `import unsafe` può dichiarare funzioni C in un blocco `extern`, che ne indica la libreria:

```
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
```

Questo programma è per Windows, la cui `kernel32.dll` contiene queste funzioni.

- Su Windows, `extern "name"` è `name.dll`. `cromo` legge le funzioni esportate dalla DLL mentre compila, quindi un nome scritto male è un errore di compilazione, non un crash all'avvio. La libreria di importazione la scrive da sé: il Windows SDK non serve. Se accanto al modulo c'è una `name.lib`, viene usata quella, per esempio una libreria C statica.
- Su Linux, `extern "name"` è la libreria statica `libname.a` accanto al modulo.
- Una DLL accanto a un modulo viene copiata accanto all'eseguibile.

```
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 I tipi al confine

Le funzioni C prendono e restituiscono numeri, `bool` e `addr`, un indirizzo C che il codice Cromo non può seguire. Un parametro `ref` è un puntatore C a un numero o a una struct di numeri, che ha lo stesso layout del C. Il `BOOL` di Windows è un `i32`.

Le funzioni C mantengono il loro nome e sono private al loro modulo: gli altri moduli non possono chiamarle.

17.3 Incapsulare una risorsa

Una risorsa C, come un handle o un descrittore, va messa in una struct con un distruttore. Il resto del programma usa poi la struct, e la risorsa viene sempre rilasciata, come ogni altra risorsa in Cromo:

```

import unsafe

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

// Misura il tempo trascorso dalla sua creazione.
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
    }
}

```

Il modulo offre `Stopwatch`; i moduli che lo importano non scrivono `import unsafe` e non possono toccare le funzioni C.

17.4 Il dialetto unsafe

`import unsafe` dà accesso anche al tipo `addr` e alle funzioni integrate che lavorano sulla memoria grezza, come `alloc` e `free`. La libreria standard è scritta in questo dialetto: `list`, `string` e `map` sono struct di Cromo che gestiscono la propria memoria con queste funzioni, dietro un'interfaccia sicura.

Di norma un programma non ha motivo di usare il dialetto, se non per chiamare il C. Cercando `import unsafe` si trovano tutti i punti in cui la sicurezza del programma dipende dalla cura di chi l'ha scritto.

17.5 Riepilogo

- `import unsafe` permette i blocchi `extern` di funzioni C, il tipo `addr` e la memoria grezza.

- `extern "name"` è `name.dll` su Windows, `libname.a` su Linux; le funzioni vengono controllate in fase di compilazione.
- Le risorse C vanno in struct con distruttori, e il resto del programma usa quelle.

Progetto finale: un servizio web

L'ultimo capitolo mette insieme quanto visto nella guida: un piccolo servizio che tiene una lista di cose da fare, con un'API JSON su HTTP. Usa le struct e il possesso, un `thread.Mutex` condiviso tra le richieste, il JSON scritto dal compilatore, il server e il client HTTP, e un test.

Il servizio risponde a quattro richieste:

Richiesta	Risposta
GET /todos	la lista, in JSON
POST /todos con {"title": "..."} 	201 e la nuova attività, oppure 400 se manca il titolo
POST /todos/{id}/done	204, oppure 404 se l'attività non esiste
tutto il resto	404

18.1 I dati

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

struct NewTodo {
    string title
}
```

`Todo` è ciò che il servizio conserva e invia; `NewTodo` è ciò che un client invia per crearne una nuova. Entrambe vengono convertite da e verso JSON dal compilatore, senza scrivere una riga di codice.

18.2 L'archivio

Le attività stanno in uno `Store`, una struct con i suoi metodi:

```
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` conserva nella lista una copia della nuova attività e restituisce l'altra, che l'handler rimanda al client. `complete` scorre la lista con `ref`, per modificare l'attività che trova.

L'archivio non ha niente a che fare con HTTP, quindi è facile da testare:

```

[[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 L'handler

Il server chiama l'handler per ogni richiesta, su molti task contemporaneamente. L'archivio è condiviso da tutte le richieste, quindi sta in un `thread.Mutex<Store>`, e l'handler vi accede solo dentro `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
    }
}

```

Alcuni dettagli:

- Le lambda passate a `lock` catturano `res` con `ref`, per scrivere la risposta, e in lettura i valori che servono loro.
- Un corpo che non è JSON fa lanciare un'eccezione a `json.decode`; il server trasforma l'eccezione non gestita in una risposta con stato 500 e prosegue con le altre richieste.
- `i64(req.param("id"))` lancia un'eccezione allo stesso modo se l'id non è un numero.

18.4 Il programma completo

Ecco il `main` del servizio: le dichiarazioni delle sezioni precedenti stanno al posto di `// ...`, e sul sito *Mostra tutto* apre il file intero. Per provare il servizio senza un browser, il `main` avvia anche un client che invia alcune richieste e stampa le risposte; un servizio vero si limiterebbe a chiamare `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()

    // Un client che prova il servizio.
    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

```

E il suo test:

```
cromo test todo.cro
```

```

ok    addGivesNewIds
1 passed, 0 failed

```

18.5 Farne un servizio vero

Per servire client reali, sostituisci il client in `main` con il solo server, su una porta fissa:

```

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
}

```

Poi compilalo per il server, che può essere Linux anche se lavori su Windows:

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

Il risultato è un unico eseguibile, senza dipendenze, che gira su qualsiasi Linux x64.

Da qui, il servizio potrebbe salvare i suoi dati in un file con `std.fs`, servire HTTPS con `http.listenTls`, o controllare un token negli header di ogni richiesta. La libreria ha quanto serve per ciascuna di queste cose, e il compilatore controlla che le richieste, che girano contemporaneamente, non tocchino mai l'archivio senza il lock.

18.6 Per proseguire

- `docs/language.md` nel [repository](#) è la bozza della specifica, con tutte le regole del linguaggio.
- `docs/decisions.md` spiega perché ogni regola è fatta così.
- La cartella `examples/` contiene altri programmi, e `library/` la libreria standard, scritta in Cromo.

Grazie per aver letto questa guida, e benvenuto in Cromo.

A

Parole chiave e operatori

A.1 Parole chiave

Parola chiave	Uso
<code>break</code>	esce da un ciclo
<code>case</code>	un ramo di uno <code>switch</code>
<code>catch</code>	gestisce un'eccezione
<code>const</code>	una variabile immutabile; un metodo che non modifica il suo oggetto; una <code>const func</code>
<code>continue</code>	passa all'iterazione successiva di un ciclo
<code>default</code>	il ramo di uno <code>switch</code> per i valori non elencati
<code>else</code>	l'alternativa di un <code>if</code>
<code>enum</code>	dichiara un enum
<code>extern</code>	dichiara funzioni C, in un modulo con <code>import unsafe</code>
<code>false</code> , <code>true</code>	i valori di <code>bool</code>
<code>finally</code>	viene eseguito a ogni uscita da un <code>try</code>
<code>for</code>	il ciclo, in tutte le sue forme
<code>func</code>	il tipo di una funzione

Parola chiave	Uso
<code>if</code>	una condizione
<code>import</code>	importa un modulo
<code>in</code>	in un <code>for</code> sugli elementi
<code>interface</code>	dichiara un'interfaccia
<code>move</code>	sposta un valore; un parametro o una cattura che prende il possesso
<code>public</code>	rende una dichiarazione visibile fuori dal suo modulo
<code>ref</code>	un parametro o una cattura che modifica; gli elementi di un <code>for</code> che si possono modificare
<code>ret</code>	ritorna da una funzione
<code>struct</code>	dichiara una struct
<code>switch</code>	sceglie tra più case
<code>throw</code>	lancia un'eccezione
<code>try</code>	il blocco di cui si gestiscono le eccezioni
<code>unsafe</code>	<code>import unsafe</code> : il dialetto delle funzioni C e della memoria grezza
<code>void</code>	nessun risultato

`print`, `println`, `eprint`, `eprintln`, `format`, `panic` e `assert` sono funzioni integrate, non parole chiave.

A.2 Tipi integrati

Tipo	Significato
<code>i8</code> , <code>i16</code> , <code>i32</code> , <code>i64</code>	interi con segno
<code>u8</code> , <code>u16</code> , <code>u32</code> , <code>u64</code>	interi senza segno
<code>f32</code> , <code>f64</code>	numeri in virgola mobile
<code>bool</code>	<code>true</code> o <code>false</code>
<code>char</code>	un byte, sinonimo di <code>u8</code>

Tipo	Significato
<code>string</code>	testo in UTF-8
<code>list<T></code>	una lista che cresce
<code>map<K, V></code>	una tabella hash
<code>T[n]</code>	un array di dimensione fissa
<code>func(A, B) R</code>	una funzione
<code>addr</code>	un indirizzo C, solo con <code>import unsafe</code>

A.3 Operatori

Dalla precedenza più alta alla più bassa:

Operatori	Significato
<code>.</code> <code>()</code> <code>[]</code>	membro, chiamata, indice
<code>!</code> <code>-</code>	not, inversione dei bit, negazione
<code>*</code> <code>/</code> <code>%</code>	moltiplicazione, divisione, resto
<code>+</code> <code>-</code>	addizione, sottrazione; <code>+</code> concatena anche le stringhe
<code><<</code> <code>>></code>	shift
<code>&</code>	and bit a bit
<code>^</code>	or esclusivo bit a bit
<code> </code>	or bit a bit
<code>==</code> <code>!=</code> <code><</code> <code><=</code> <code>></code> <code>>=</code>	confronti
<code>&&</code>	and logico, valutato in cortocircuito
<code> </code>	or logico, valutato in cortocircuito
<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code> <code>&=</code> <code> =</code> <code>^=</code> <code><<=</code> <code>>>=</code>	assegnamenti, istruzioni a sé stanti

- `i++` e `i--` sono istruzioni a sé stanti, mai parti di un'espressione.

- Non esiste l'operatore condizionale `?:` : scrivi un `if`.
- Uno shift oltre la larghezza del tipo provoca un panic nelle build di debug.
- Gli assegnamenti sono istruzioni: `if (a = b)` non è Croma.

A.4 Letterali

Letterale	Esempio
intero	<code>42</code> , <code>1_000_000</code> , <code>0xFF</code> , <code>0b1010</code>
virgola mobile	<code>3.14</code> , <code>1e-9</code> , <code>2.5e3</code>
carattere	<code>'a'</code> , <code>'\n'</code> , <code>'\t'</code> , <code>'\\'</code> , <code>'\''</code> , <code>'\x41'</code>
stringa	<code>"text"</code> , con le stesse sequenze di escape e <code>\</code>
lista	<code>[1, 2, 3]</code>

A.5 Attributi

Gli attributi si scrivono come in C++11, prima di ciò a cui si applicano:

Attributo	Significato
<code>[[test]]</code>	una funzione di test
<code>[[json("name")]]</code>	il nome di un campo o di un case nel JSON
<code>[[json(skip)]]</code>	un campo escluso dal JSON

B

Il comando `cromo`

`cromo` è l'unico strumento di Cromo: il compilatore, il linker, il test runner, il formattatore e il language server, in un solo eseguibile.

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

Se non indichi un file, i comandi usano il `main.cro` della cartella corrente.

B.1 Comandi

Comando	Cosa fa
<code>cromo build</code> <code>[file.cro]</code>	compila il programma in un eseguibile
<code>cromo run [file.cro]</code> <code>[-- arguments]</code>	compila il programma in una cartella temporanea, lo esegue con gli argomenti che seguono <code>--</code> e termina con il suo codice di uscita
<code>cromo test [file.cro]</code> <code>[names]</code>	esegue le funzioni <code>[[test]]</code> del programma e dei suoi moduli; i nomi selezionano i test che li contengono
<code>cromo fmt [--check]</code> <code>[files and folders </code> <code>-]</code>	formatta il codice, di default quello della cartella corrente e delle sue sottocartelle; <code>--check</code> si limita a elencare i file da formattare; <code>-</code> formatta lo standard input
<code>cromo lsp</code>	il language server, per gli editor, su standard input e standard output
<code>cromo version</code>	stampa la versione di <code>cromo</code> e di LLVM

Comando	Cosa fa
<code>cromo help</code>	elenca i comandi

B.2 Opzioni di build e run

Opzione	Significato
<code>-o <file></code>	il nome dell'eseguibile; di default il nome del sorgente, o della cartella se non c'è un file, con <code>.exe</code> per Windows
<code>--release</code>	in caso di overflow gli interi ricominciano dall'estremo opposto, invece di fermare il programma
<code>--target windows</code> , <code>--target linux</code>	il sistema dell'eseguibile; di default quello su cui gira <code>cromo</code>
<code>-O0 ... -O3</code>	il livello di ottimizzazione; il default è <code>-O2</code>
<code>--emit-ir</code>	stampa l'LLVM IR del programma invece di scrivere l'eseguibile

B.3 Debug e release

	Debug (default)	<code>--release</code>
Ottimizzazione	<code>-O2</code>	<code>-O2</code>
Overflow degli interi	panic	ricominciano dall'estremo opposto
Limiti degli indici, divisione per zero, esclusività	controllati	controllati
Tracce dello stack	sì	sì
Informazioni per il debugger	sì: PDB su Windows, DWARF su Linux	no

Con `-O0` un debugger mostra ogni variabile; con le ottimizzazioni, alcune possono risultare *optimized out*.

B.4 Codici di uscita

Codice	Significato
0	il programma è terminato normalmente, o <code>main</code> ha restituito 0
1	un'eccezione è uscita da <code>main</code> , oppure <code>cromo</code> ha trovato degli errori
101	un panic, o uno stack overflow
altro	il valore restituito da <code>main</code> , o passato a <code>os.exit</code>

B.5 File

Percorso	Contenuto
<code>~/.cromo/current/bin/cromo</code>	la versione attiva, raggiunta dal <code>PATH</code>
<code>~/.cromo/versions/<version></code>	le versioni installate
<code>lib/</code> accanto a <code>bin/</code>	la libreria standard e il runtime, che <code>cromo</code> legge

Su Windows, `~` è `%USERPROFILE%`.

C

Cromo per programmatori C++, Go e Rust

Cromo prende qualcosa da ciascuno di questi linguaggi. Questa appendice elenca, per ognuno, che cosa ti risulterà familiare e che cosa invece è diverso.

C.1 Dal C++

Familiare: la sintassi, le graffe e i tipi prima dei nomi; le struct con costruttori, metodi, metodi `const` e distruttori; RAII e l'ordine di distruzione; le eccezioni con `try`, `catch` e `throw`; i generici compilati per ogni tipo, come i template; `std::format` per la formattazione.

Diverso:

C++	Cromo
<code>return</code>	<code>ret</code>
<code>;</code> alla fine delle istruzioni	la fine della riga
puntatori, <code>nullptr</code> , <code>new</code> , <code>delete</code>	niente puntatori e niente <code>null</code> ; l'heap solo attraverso i contenitori
<code>T&</code> , <code>const T&</code>	un parametro è in lettura per default; <code>ref</code> modifica; nessun riferimento al di fuori dei parametri
costruttori di copia, copie implicite	<code>copy()</code> , sempre esplicito per i tipi che possiedono risorse

C++	Cromo
<code>std::move</code>	<code>move</code> , controllato: una variabile spostata non può più essere usata
<code>class</code> , ereditarietà, <code>virtual</code>	<code>struct</code> e interfacce
template con qualsiasi tipo	generici vincolati da interfacce, controllati una volta sola
<code>while</code> , <code>do</code>	<code>for (condition)</code> , <code>for { }</code>
<code>switch</code> con fall-through	niente fall-through; sugli enum ogni case va gestito
<code>#include</code> , header	<code>import</code> di moduli; nessun preprocessore
<code>int</code> , <code>long</code> , <code>size_t</code>	<code>i32</code> , <code>i64</code> , <code>u64</code> : dimensioni fisse; le lunghezze sono <code>i64</code>
comportamento indefinito per overflow e accessi fuori dai limiti	un panic nelle build di debug; i limiti sono sempre controllati
variabili non inizializzate	ogni variabile ha il suo valore predefinito
<code>std::thread</code> , <code>std::mutex</code>	<code>thread.Thread</code> , <code>thread.Mutex<T></code> che contiene i suoi dati

C.2 Da Go

Familiare: un solo strumento che compila, esegue, testa e formatta; task leggeri che si bloccano senza bisogno di `async`; i canali; una libreria standard per le reti e per HTTP; eseguibili statici con la cross-compilazione; e, quando arriveranno i pacchetti, dipendenze indicate dall'URL del loro repository, con la selezione della versione minima.

Diverso:

Go	Cromo
garbage collector	possezzo e distruttori; la memoria viene liberata quando finisce la vita del proprietario
<code>go f()</code>	<code>thread.Task</code> o <code>thread.Group</code> , di cui il compilatore controlla le catture
<code>defer</code>	distruttori, e <code>finally</code>

Go	Cromo
valori <code>error</code> e <code>if err != nil</code>	eccezioni
<code>nil</code>	niente <code>null</code> ; ogni variabile ha un valore
<code>x := 1</code>	<code>i64 x = 1</code> : il tipo si scrive sempre
iniziali maiuscole per i nomi esportati	<code>public</code>
<code>select</code>	non ancora
slice che condividono la memoria	liste che possiedono i loro elementi; <code>slice()</code> crea una copia

C.3 Da Rust

Familiare: il possesso controllato dal compilatore, lo spostamento come comportamento predefinito per i tipi che possiedono risorse, niente `null`, niente data race; gli enum con dati e il match esaustivo; i generici vincolati da interfacce, come i trait; LLVM; i test integrati nello strumento.

Diverso:

Rust	Cromo
riferimenti con lifetime, il borrow checker	<code>ref</code> solo per i parametri: un riferimento non sopravvive mai a una chiamata, e non ci sono lifetime
<code>&T</code> , <code>&mut T</code>	un parametro in lettura, un parametro <code>ref</code>
<code>Clone</code> , <code>.clone()</code>	<code>copy()</code>
tipi <code>Copy</code>	tipi semplici: numeri, <code>bool</code> , struct composte solo da questi
<code>Result<T, E></code> e <code>?</code>	eccezioni; enum con dati per i risultati che sono valori
<code>Option<T></code>	un valore convenzionale, un <code>bool</code> con un parametro <code>ref</code> , oppure un enum
<code>match</code> con pattern	<code>switch</code> sui case, dando un nome ai dati; niente pattern annidati
<code>Drop</code>	distruttori

Rust	Cromo
<code>impl Trait for Type</code>	<code>struct Type : Interface</code> , con i metodi dentro la struct
<code>async / .await</code>	task che si bloccano, come in Go
<code>Arc<Mutex<T>></code>	<code>thread.Mutex<T></code> catturato dai task che lo usano
macro	non disponibili

C.4 In breve

Cromo si scrive come il C++, gestisce la memoria come Rust ma con meno regole, ed esegue il codice concorrente come Go. Nel dubbio, vince la regola più semplice.