Azzolini Riccardo 2021-02-24

Il linguaggio C

1 Storia

Il linguaggio **C** fu inizialmente progettato nel 1972 da Dennis Ritchie, che lo sviluppò come evoluzione del linguaggio *B*, a sua volta derivato da *BCPL*. Una delle prime applicazioni del C fu la riscrittura del kernel di Unix, dopodiché esso si diffuse sempre più ampiamente (in ambito prima industriale, e poi anche accademico).

Nel 1990, l'ANSI (American National Standards Institute) pubblicò la prima specifica **standard** del linguaggio C. La standardizzazione di un linguaggio di programmazione è importante perché favorisce la portabilità del codice, facendo sì che compilatori diversi possano compilare lo stesso codice, invece di avere ciascuno un proprio "dialetto" del linguaggio.

2 Caratteristiche principali

Il C è un linguaggio:

- **tipizzato**;
- con **binding statico**: l'indirizzo associato al nome di una procedura (ovvero quali siano le istruzioni che la implementano) viene determinato in fase di compilazione, e non può cambiare durante l'esecuzione (mentre ciò è possibile con il **binding dinamico**);
- che pone la massima attenzione alle prestazioni, sia dal punto di vista della velocità di esecuzione che della compattezza del codice sorgente (ciò si manifesta nella scelta della sintassi dei vari costrutti, come ad esempio l'uso delle parentesi graffe al posto delle parole chiave `BEGIN` e `END`).

Queste caratteristiche sono sostanzialmente motivate dallo stato della tecnologia informatica all'epoca dello sviluppo del C: i calcolatori erano ingombranti, costosi, e di potenza molto limitata — avevano decine di kilobyte di memoria RAM, frequenze di clock nell'ordine dei kilohertz, hard disk di pochi megabyte grandi come elettrodomestici, archivi a nastro ad accesso sequenziale (con tempi di accesso misurati in secondi), ecc.

3 Struttura di un programma C

Ad alto livello, un programma C è costituito da un insieme non vuoto di funzioni, definite tutte allo stesso livello (il nesting non è ammesso). In C, una **funzione** è una sequenza di istruzioni associata a un nome, che elabora eventuali dati ricevuti in ingresso e restituisce eventualmente un risultato. In altre parole, le funzioni C corrispondono alle funzione e alle procedure della programmazione procedurale in generale; il C non fa alcuna distinzione sintattica tra funzioni e procedure.

In ogni programma deve essere obbligatoriamente presente almeno una funzione chiamata `main`, che costituisce il punto di inizio del programma: all'avvio, l'esecuzione inizia appunto dalla prima istruzione di `main`.

4 Tipi di dati predefiniti

Il C mette a disposizione una serie di **tipi di dati predefiniti** (*built-in*); essi sono tutti tipi numerici, classificabili in **tipi interi** e **tipi reali**.

I tipi interi sono `char`, `short`, `int` e `long`, ciascuno disponibile in due versioni:

- *con segno* (indicata dal modificatore `signed`), che può rappresentare numeri sia positivi che negativi;
- *senza segno* (`unsigned`), che può rappresentare solo valori positivi.

Il nome del tipo `char`, in particolare, deriva dal fatto che i suoi valori possono essere interpretati come caratteri, ad esempio secondo il codice ASCII.

I quattro interi tipi si distinguono per il numero di byte occupati dalle variabili, e dunque per i range di valori che possono essere rappresentati. Lo standard C non fissa un numero di byte preciso per ciascun tipo (fare ciò limiterebbe le architetture hardware sulle quali il linguaggio potrebbe essere impiegato in modo efficiente), ma si limita a specificare che ciascun tipo deve essere grande *almeno quanto il tipo precedente* nell'ordine `char`, `short`, `int`, `long`:

$$\texttt{char} \leq \texttt{short} \leq \texttt{int} \leq \texttt{long}$$

Le dimensioni esatte variano poi in base all'architettura per cui si compila il codice e allo specifico compilatore impiegato; un esempio tipico di dimensioni dei tipi e intervalli di valori rappresentabili è riportato nella seguente tabella:

Tipo	Byte	Intervallo `signed`	Intervallo `unsigned`
`char`	1	$[-128, +127]$	$[0,\ 255]$
`short`	2	$[-32\,768, +32\,767]$	$[0,\ 65\,535]$
`int`	4	$[-2\,147\,483\,648, +2\,147\,483\,647]$	$[0,\ 4\,294\,967\,295]$
`long`	8	$[-9\,223\,372\,036\,854\,775\,808,$ $+9\,223\,372\,036\,854\,775\,807]$	$[0,\ 18\,446\,744\,073\,709\,551\,615]$

Un discorso analogo vale per i tipi reali, che sono `float`, `double` e `long double`, anch'essi in ordine crescente di grandezza:

$$\texttt{float} \leq \texttt{double} \leq \texttt{long double}$$

Tipicamente, lo spazio dedicato alla rappresentazione di una variabile di tipo `float` è di 4 byte (con i quali si ha una precisione di circa 6 cifre significative), mentre lo spazio occupato da una variabile `double` è di 8 byte (corrispondenti a circa 15 cifre significative), e infine alle variabili `long double` sono dedicati 16 byte.

5 Determinare le dimensioni dei tipi predefiniti

Per determinare il numero di byte occupato dalle variabili dei vari tipi predefiniti sulla propria macchina e con il proprio compilatore, si può scrivere un semplice programma C:

```c
#include <stdio.h>

int main(int argc, const char *argv[]) {
    printf("Dimensione di char: %lu byte\n", sizeof(char));
    printf("Dimensione di short: %lu byte\n", sizeof(short));
    printf("Dimensione di int: %lu byte\n", sizeof(int));
    printf("Dimensione di long: %lu byte\n", sizeof(long));
    printf("Dimensione di float: %lu byte\n", sizeof(float));
    printf("Dimensione di double: %lu byte\n", sizeof(double));
    printf("Dimensione di long double: %lu byte\n", sizeof(long double));
    return 0;
}
```

- La prima riga di questo programma, `#include <stdio.h>`, è una *direttiva al preprocessore* che, sostanzialmente, annuncia il fatto di voler usare le funzioni che la libreria standard mette a disposizione per l'input e output ("`stdio`" sta appunto per "standard input output"). Più nel dettaglio, `stdio.h` è il nome di un file (fornito con il compilatore C) che contiene i *prototipi* di tali funzioni di libreria (ovvero i loro nomi, argomenti e tipi restituiti), e la direttiva `#include` viene sostituita, prima della compilazione, con il contenuto di tale file, dando così al compilatore le informazioni necessarie per permettere l'uso delle suddette funzioni.
- Successivamente, viene definita la funzione `main` (che, come detto prima, costituisce il punto di inizio del programma ed è obbligatoria). Essa ha alcuni argomenti, che verranno spiegati più avanti (ma, in sintesi, servono ad accedere agli argomenti specificati sulla riga di comando quando si avvia il programma dalla shell), e restituisce un valore intero (anch'esso usato per comunicare con la shell, in particolare per dare un'indicazione sull'esito dell'esecuzione del programma).

- Le principali istruzioni che costituiscono la funzione `main` sono una serie di chiamate della funzione di libreria `printf`, che serve a stampare una stringa sullo *standard output* (che tipicamente è lo schermo). Anche questa verrà ripresa più avanti, ma sostanzialmente funziona in questo modo:

  1. il primo argomento è una *stringa di controllo* che può contenere delle *specifiche di conversione*, sequenze di caratteri precedute da `%` che indicano tipi di valori stampare e come formattarli;
  2. gli zero o più argomenti successivi sono le variabili/espressioni che forniscono i valori da stampare.

  In questo esempio, `printf` viene sempre chiamata con una stringa di controllo contenente una singola specifica di conversione, `%lu`, che serve a stampare il valore di tipo `unsigned long` restituito dall'operatore `sizeof`. Tale valore è il numero di byte occupati da una variabile del tipo indicato tra parentesi.

- All'interno delle stringhe, si usa la sintassi `\n` per indicare il carattere "a capo".
- L'ultima istruzione di `main` è `return 0;`, che specifica il valore intero restituito dalla funzione (e come detto prima, nel caso di `main` tale valore viene mandato alla shell).

6 Costruttori di tipo

Il linguaggio C permette al programmatore di definire nuovi tipi tramite i **costruttori di tipo**: array e record.

6.1 Array

Una variabile di tipo **array** (o *vettore*) rappresenta una *sequenza di lunghezza fissa di dati omogenei* (ovvero dello stesso tipo) disposti in celle di memoria *adiacenti*. Ad esempio, la riga di codice

```
unsigned int voti[20];
```

definisce una variabile chiamata `voti` che rappresenta una sequenza di 20 numeri interi. Se un intero occupa 4 byte, la variabile `voti` occuperà (almeno[1]) $20 \cdot 4 = 80$ byte.

L'accesso agli elementi di un array avviene tramite indici di tipo intero, e l'indice del primo elemento è **0** (anche questo per motivi di efficienza). Sui valori degli indici usati

[1] La dimensione della variabile potrebbe essere maggiore a causa di un eventuale *allineamento* effettuato dal compilatore: su molte architetture, l'accesso a un dato in memoria è più efficiente quando l'indirizzo di tale dato è un multiplo della dimensione del dato stesso, dunque i compilatori spesso inseriscono dei byte di riempimento tra i dati "veri e propri" per aumentare l'efficienza del programma compilato.

non viene effettuato alcun controllo di validità: ad esempio, dato l'array `voti` definito prima, l'istruzione

```
voti[20] = 18;
```

scrive il valore 18 nei byte immediatamente successivi alla fine dell'array, perché l'indice 20 si riferirebbe al ventunesimo elemento, che non esiste. Ciò potrebbe provocare un errore in fase di esecuzione (se ad esempio i byte successivi all'array ricadono in una zona di memoria protetta dal sistema operativo), o, peggio, potrebbe modificare il valore di un altro oggetto a cui tali byte sono associati: allora, il programma si comporterebbe probabilmente in modo anomalo, magari anche nello svolgere operazioni che non coinvolgono questo array, e sarebbe molto difficile risalire alla causa del problema.

Quando si definisce una variabile di tipo array, è possibile elencare direttamente la sequenza di valori con cui inizializzarla, e in tal caso si può anche omettere la lunghezza, che il compilatore ricava dal numero di valori specificati. Ad esempio:

```
char nome[] = {'p', 'i', 'p', 'p', 'o', '\0'};
```

Questo esempio mostra la definizione di un array di caratteri, che può essere interpretato come una stringa: i valori indicati tra singoli apici sono caratteri (corrispondenti al tipo `char` dell'array), e in particolare `'\0'` indica il carattere la cui codifica è il valore 0,[2] che in C viene usato, per convenzione, per indicare la fine di una stringa (così, le funzioni di libreria possono operare sulle stringhe senza bisogno di conoscerne la lunghezza). Esiste anche una sintassi abbreviata, del tutto equivalente, per inizializzare array di caratteri terminati da `'\0'`:

```
char nome[] = "pippo";
```

Infine, si possono definire array pluridimensionali (a due o più dimensioni), che verranno trattati più avanti. Ad esempio, con la riga di codice

```
int mat[10][10];
```

si definisce un array bidimensionale di 10×10 interi.

6.2 Record

Un **record**, chiamato **struttura** (`struct`) in C, è un *contenitore di dati eterogenei* (cioè che possono avere tipi differenti), che dal punto di vista matematico corrisponde al *prodotto cartesiano*. Una struttura viene definita specificando un *nome etichetta* e un elenco di *campi*. Ad esempio:

[2] Il carattere `'\0'` non deve essere confuso con il carattere `'0'`, la cifra 0, la cui codifica in ASCII è 48 (e non 0).

```
struct persona {
    char nome[10];
    char cognome[20];
    unsigned int peso;
};
```

Per definire una variabile di tipo struttura, si usa poi la sintassi illustrata nel seguente esempio, nel quale il nome della variabile è `io`:

```
struct persona io;
```

Se si vuole evitare di dover ripetere ogni volta la parola chiave `struct`, si può ricorrere al costrutto `typedef` per definire un tipo, chiamato ad esempio `persona`, che funga da *alias* di `struct persona`:

```
typedef struct persona persona;
```

Adesso, per dichiarare una variabile di tipo `struct persona` basta scrivere:

```
persona altro;
```

Si osservi che il tipo `persona` è distinto dalla struttura `struct persona`, poiché in C i tipi e le strutture hanno spazi dei nomi diversi: è consentito definire un tipo e una struttura con lo stesso nome, ma non due tipi con lo stesso nome o due strutture con lo stesso nome. Il fatto di aver scelto per il tipo lo stesso nome della struttura è arbitrario, e sarebbe invece stato possibile usare un qualunque altro nome, come ad esempio:

```
typedef struct persona individuo;
individuo altro;
```

L'accesso ai campi di una variabile di tipo struttura avviene attraverso l'operatore binario infisso `.`, che deve avere come operando sinistro la variabile, e come operando destro il nome di un campo della struttura. Ad esempio:

```
io.peso = 75;
```