Azzolini Riccardo 2021-03-08

Espressioni e funzioni

1 Espressioni

In C un'**espressione** è costituita da una combinazione di *operandi* e *operatori* che rispetta le regole sintattiche del linguaggio. Il valore di un'espressione è univocamente determinato dai valori assunti dagli operandi, dal significato degli operatori e dalle regole di *precedenza* e *associatività* di questi ultimi.

Ad esempio, l'espressione

```
i = a + 3 * b
```

è formata da 4 operandi (`i`, `a`, `3`, `b`) e 3 operatori binari infissi (`=`, `+`, `*`). Siccome l'operatore `*` ha precedenza maggiore di `*`, che a sua volta ha precedenza maggiore di `=`, quest'espressione viene interpretata come:

```
i = (a + (3 * b))
```

Allora, per valutare l'espressione, il compilatore emette del codice che:

1. calcola il prodotto `3 * b` e memorizza il risultato in una variabile temporanea;
2. calcola la somma `a + (3 * b)`, e se necessario (a seconda delle caratteristiche della macchina sottostante) memorizza anche questo risultato in una variabile temporanea;
3. assegna alla variabile `i` il risultato della somma.

Si osservi che in C, a differenza di altri linguaggi, l'assegnamento è appunto un operatore (con precedenza molto bassa e associatività da destra a sinistra), che come ogni operatore restituisce un valore: il valore assunto dall'operando destro, ovvero il valore assegnato all'operando sinistro. Ciò permette la scrittura di assegnamenti a cascata: ad esempio, l'espressione

```
a = b = c = 3
```

viene interpretata come

```
a = (b = (c = 3))
```

e la sua valutazione avviene come segue:

1. si valuta l'assegnamento `c = 3`, che assegna il valore `3` alla variabile `c` e restituisce lo stesso valore;

2. si valuta l'assegnamento `b = (c = 3)`, che assegna a `b` il valore restituito da `c = 3`, ovvero `3`, e restituisce ancora lo stesso valore;
3. si valuta infine `a = (b = (c = 3))`, che ancora una volta assegna (ad `a`) e restituisce il valore `3`, poiché questo è il valore restituito dall'operando destro, `b = (c = 3)`.

Un altro esempio è

```
x = a = b - 6 < c
```

che viene interpretata come

```
x = (a = ((b - 6) < c))
```

dunque la sua valutazione consiste nel determinare se `b - 6` è minore di `c` e assegnare il risultato del confronto (0 oppure 1) ad `a` e a `x`.

È importante fare attenzione a scrivere espressioni leggibili, non troppo complicate. A tale scopo, due consigli utili sono:

- aggiungere delle parentesi, anche se tecnicamente superflue, ogni volta che si ha un dubbio sulla precedenza e/o sull'associatività degli operatori impiegati;
- evitare un uso eccessivo degli operatori che hanno *side effect* (`=`, `++` e `--`), cioè che modificano i valori dei loro operandi.

Inoltre, bisogna stare attenti anche a non confondere la sintassi delle espressioni C con la notazione matematica. Ad esempio, si potrebbe pensare che l'espressione

```
3 < i < 5
```

dia risultato 1 (cioè vero) se e solo se il valore di `i` è compreso tra 3 e 5, estremi esclusi. Invece, essa viene interpretata come

```
(3 < i) < 5
```

ovvero:

1. si confronta il valore di `i` con 3, dando un risultato che può essere 0 (falso) oppure 1 (vero);
2. sia 0 che 1 sono minori di 5, quindi l'espressione restituisce *sempre* il valore 1 (vero).

2 Funzioni

Il C non presenta la distinzione sintattica tra funzioni e procedure che esiste in altri linguaggi di programmazione (ad esempio Pascal): la sintassi per definire funzioni e procedure è la stessa, e dunque si usa spesso il termine *funzione* per riferirsi a entrambe.

Una funzione viene definita specificando, in quest'ordine:

1. il **tipo del valore restituito**, che può essere vuoto, `void`, per indicare che non viene restituito alcun valore, cioè che la funzione è una procedura;
2. il **nome** della funzione;
3. l'elenco dei **parametri formali**;
4. il **corpo** della funzione.

2.1 Corpo

Il corpo di una funzione è un'istruzione composta, costituita da una sequenza di *dichiarazioni di variabili locali* e istruzioni racchiusa tra parentesi graffe.

Nelle prime versioni del linguaggio C (il C "tradizionale" pre-standardizzazione, e poi lo standard ANSI C del 1990) era obbligatorio mettere tutte le dichiarazioni all'inizio del corpo di una funzione, prima delle istruzioni, principalmente perché si pensava che ciò migliorasse la leggibilità del codice. Tuttavia, ci si è poi accorti che in realtà il codice risulta più leggibile senza questo vincolo, perché le variabili possono essere definite più vicino a dove vengono usate (il che aiuta soprattutto nelle funzioni grandi). L'esempio classico è un ciclo `for`:

- Se le tutte dichiarazioni devono essere all'inizio del corpo, bisogna scrivere, ad esempio,

  ```c
  int f(int n) {
      int i;
      /* ... */
      for (i = 0; i < n; i++) { /* ... */ }
      /* ... */
  }
  ```

  dove le righe di codice tra la dichiarazione di `i` e il suo uso nel ciclo potrebbero magari essere anche qualche decina.

- Invece, le versioni moderne del C consentono di scrivere direttamente:

```c
int f(int n) {
    // ...
    for (int i = 0; i < n; i++) { /* ... */ }
    // ...
}
```

Così, quando nella lettura del codice si incontra la dichiarazione di `i`, è immediato capire quale sia lo scopo di tale variabile.

2.2 Istruzione `return`

L'istruzione `return` arresta l'esecuzione della funzione in cui viene eseguita, trasferendo il controllo al chiamante ed eventualmente restituendo a esso un valore.

- Una funzione che deve restituire un valore (cioè una funzione "vera e propria", con un tipo di valore restituito diverso da `void`) deve *sempre* eseguire un'istruzione `return`, scritta con la sintassi

  `return` *`expr`*`;`

  dove *`expr`* è un'espressione che determina il valore restituito. Ad esempio, la seguente è una funzione che restituisce un valore di tipo `float*` (puntatore a `float`):

  ```c
  float *max(float vet[], int dim) {
      float *pmax;
      // ...
      return pmax;
  }
  ```

- Una funzione con tipo di valore restituito `void`, ovvero una procedura, non deve restituire alcun valore, dunque al suo interno l'istruzione `return` assume la sintassi:

  `return;`

  Inoltre, non è obbligatoria l'esecuzione di un'istruzione `return` esplicita: se non ne viene eseguita una, il controllo torna implicitamente al chiamante al termine delle istruzioni presenti nel corpo della funzione. Ad esempio, la funzione/procedura

  ```c
  void print_pwr_2(unsigned int n) {
      unsigned int i, j;
      j = 1;
      for (i = 0; i <= n; i++) {
          printf("2^%u = %u\n", i, j);
          j = 2 * j;
      }
  ```

```
    return;
}
```

funzionerebbe allo stesso modo se si rimuovesse il `return` finale.

2.3 Prototipo

Il tipo del valore restituito da una funzione, il nome della funzione e l'elenco dei suoi parametri formali costituiscono un **prototipo di funzione**, che contiene tutte le informazioni necessarie al compilatore per gestire correttamente l'istruzione di chiamata di tale funzione.

È possibile scrivere anche solo il prototipo di una funzione, senza specificare il corpo. Ad esempio:

```
void bubblesort(int vet[], int dim);
```

Queste informazioni sono sufficienti a generare il codice che chiama la funzione, ma non il codice della funzione stessa (mancano le istruzioni da eseguire, specificate nel corpo), quindi da qualche parte deve esserci una definizione completa. Tuttavia, se è presente un prototipo nel file dove la funzione viene usata, la definizione completa può trovarsi "da un'altra parte", ad esempio in una libreria separata.

3 File di intestazione

I prototipi delle funzioni che una libreria mette a disposizione sono riportati in un **file di intestazione** (**header file**), che serve a separare l'interfaccia della libreria dal codice che ne fornisce l'implementazione. Quando si vuole usare la libreria in un file di codice, bisogna includere in questo file i contenuti del file di intestazione, il che viene fatto tramite la **direttiva al preprocessore** `#include`, la cui sintassi è

`#include <`*filename*`>`

oppure

`#include "`*filename*`"`

Il **preprocessore** è uno strumento che viene eseguito automaticamente quando si invoca il compilatore C, prima che inizi il processo di compilazione vero e proprio. Esso legge i file da compilare, ed effettua al loro interno alcune sostituzioni testuali. In particolare, quando il preprocessore trova una direttiva `#include`, la sostituisce con l'intero contenuto del file specificato in tale direttiva. La ricerca del file dipende da quale tra le due forme della direttiva sia stata impiegata:

- con `#include <`*filename*`>` viene cercato un file di nome *filename* nella directory di sistema dedicata ai file di intestazione;

- con `#include "filename"` viene cercato un file di nome *filename* nella directory corrente.

Di conseguenza, per i file di intestazione delle librerie si usa tipicamente la forma `#include <filename>`.

Ad esempio, come già visto, per usare le funzioni fornite dalla libreria standard di input/output bisogna includere il file di intestazione `stdio.h`, scrivendo la direttiva

```
#include <stdio.h>
```

all'inizio del proprio codice.