Azzolini Riccardo 2021-03-10

Funzioni e regole di scope

1 Parametri formali

L'elenco dei parametri formali di una funzione è una sequenza della forma

$$\mathit{type}_1\ \mathit{name}_1,\ \ldots,\ \mathit{type}_n\ \mathit{name}_n$$

dove type_i è il tipo del parametro name_i (per $i = 1, \ldots, n$). Ad esempio, il prototipo di funzione

```
int f(int voti[], int dim, char l);
```

definisce una funzione con tre parametri formali, rispettivamente di tipo vettore di interi (`voti`), intero (`dim`) e carattere (`l`).

Quando si scrive solo il prototipo di una funzione, senza specificare il corpo, i nomi dei parametri possono essere omessi, poiché essi servono solo all'interno del corpo per riferirsi ai valori passati come parametri attuali al momento di una chiamata. Ad esempio, il prototipo mostrato prima può essere scritto equivalentemente in questo modo:

```
int f(int[], int, char);
```

Per indicare invece che una funzione non ha parametri si usa la parola riservata `void`:

```
int g(void);
```

2 Passaggio dei parametri

Il C supporta una sola regola per il *passaggio dei parametri*, cioè per stabilire la corrispondenza tra parametri attuali e parametri formali: il **passaggio per valore** (o **copia**).[1] Per tradurre un'espressione di chiamata di funzione

$$f(\mathit{expr}_1,\ \ldots,\ \mathit{expr}_n)$$

il compilatore genera del codice che:

[1] Quando il C fu progettato, esistevano linguaggi (ad esempio Pascal) che permettevano al programmatore di scegliere tra il passaggio per valore e il passaggio per riferimento. Implementare solo il passaggio per valore è allora una semplificazione rispetto a tali linguaggi. Tuttavia, le operazioni che sarebbero possibili con il passaggio per riferimento possono essere eseguite usando invece i puntatori, come spiegato in seguito.

1. valuta tutte le espressioni (parametri attuali) $expr_i$;
2. copia i valori risultanti nella parte dedicata ai parametri formali del nuovo record di attivazione creato per eseguire la funzione f.

Il significato dell'operazione di copia dei valori risultanti è immediato per i valori di tipi predefiniti, mentre è opportuno dire cosa succeda nei casi di parametri attuali di tipo struttura e di tipo vettore:

- se un parametro è di tipo struttura, allora viene eseguita una *copia componente per componente* della struttura (in pratica, vengono copiati nel record di attivazione tutti i bit che costituiscono la rappresentazione in memoria della struttura);
- se invece un parametro è di tipo array, esso viene interpretato come un puntatore al primo elemento, dunque viene copiato appunto tale indirizzo, e non vengono invece copiati gli elementi dell'array.

Ad esempio, si consideri il seguente frammento di codice:

```
int f(int a[], int d) {
    // ...
}

void g(void) {
    int x = 5;
    int vet[20];
    x = f(vet, x);
}
```

Quando viene eseguita la chiamata `f(vet, x)`, nel record di attivazione di `f` vengono inseriti i valori `&vet[0]` e `5`.

2.1 Simulazione del passaggio per riferimento

Si supponga di voler scrivere una procedura per scambiare i valori di due variabili. Un primo tentativo potrebbe essere questo:

```
#include <stdio.h>

void scambio(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

int main(int argc, const char *argv[]) {
    int a = 11, b = 22;
```

```
    printf("Valore di a prima di scambio: %d\n", a);
    printf("Valore di b prima di scambio: %d\n", b);
    scambio(a, b);
    printf("Valore di a dopo scambio: %d\n", a);
    printf("Valore di b dopo scambio: %d\n", b);
    return 0;
}
```

Eseguendo tale programma, però, ci si accorge che la procedura `scambio` non ha alcun effetto:

```
Valore di a prima di scambio: 11
Valore di b prima di scambio: 22
Valore di a dopo scambio: 11
Valore di b dopo scambio: 22
```

Il motivo per cui lo scambio non funziona è che i parametri sono passati per valore: i valori di `a` e `b` vengono copiati nel record di attivazione di `scambio`, e poi la procedura scambia tali valori all'interno del suo record di attivazione, mentre i valori nelle variabili originali rimangono inalterati.

Lo scambio funzionerebbe correttamente se i parametri fossero passati per riferimento, ma il C supporta solo il passaggio per valore. Allora, l'unica soluzione è simulare il passaggio per riferimento, mettendo come parametri di `scambio` dei puntatori alle variabili su cui operare: in questo modo, ciò che viene copiato nel record di attivazione siano gli *indirizzi* delle variabili, e non i loro valori, A tale scopo, è necessario modificare sia la definizione che l'invocazione di `scambio`:

```
#include <stdio.h>

void scambio(int *xp, int *yp) {
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

int main(int argc, const char *argv[]) {
    int a = 11, b = 22;
    printf("Valore di a prima di scambio: %d\n", a);
    printf("Valore di b prima di scambio: %d\n", b);
    scambio(&a, &b);
    printf("Valore di a dopo scambio: %d\n", a);
    printf("Valore di b dopo scambio: %d\n", b);
    return 0;
}
```

Con queste modifiche lo scambio viene eseguito correttamente:

```
Valore di a prima di scambio: 11
Valore di b prima di scambio: 22
Valore di a dopo scambio: 22
Valore di b dopo scambio: 11
```

3 Record di attivazione

Ogni volta che si chiama una funzione, viene allocato un nuovo **record di attivazione** (o **frame**) in un'area di memoria chiamata **stack**, che come suggerisce il nome viene gestita in modo LIFO (Last In First Out). In ogni istante, il record di attivazione presente in cima allo stack corrisponde alla funzione attualmente in esecuzione, mentre i record sottostanti corrispondono a funzioni momentaneamente sospese, ciascuna in attesa che termini l'esecuzione della funzione che ha chiamato (quella corrispondente al record immediatamente soprastante nello stack).

Le informazioni che un record di attivazione contiene, a livello qualitativo,[2] sono:

- i parametri formali, ai quali sono assegnati i valori dei parametri attuali;
- le variabili locali (che saranno spiegate meglio a breve);
- eventuali variabili temporanee, gestite dal compilatore in modo automatico e trasparente al programmatore, che possono ad esempio servire a memorizzare i risultati intermedi della valutazione delle espressioni;
- l'area di salvataggio dello stato della macchina, nella quale viene salvato lo stato dell'ambiente di esecuzione (in particolare il contenuto dei registri) al momento della chiamata della funzione, in modo da poterlo ripristinare quando si restituisce il controllo al chiamante (altrimenti, la funzione chiamante potrebbe non funzionare correttamente, perché perderebbe i dati che aveva posto nei registri prima della chiamata).

L'accesso ai contenuti del record della funzione in esecuzione (cioè quello in cima allo stack) avviene tipicamente tramite indirizzi della forma $\mathit{frame\ pointer} + \mathit{displacement}$, dove $\mathit{frame\ pointer}$ punta all'inizio del record, mentre $\mathit{displacement}$ è uno spiazzamento (offset) fissato dal compilatore in base alla posizione nel record del dato a cui si vuole accedere.

[2] L'organizzazione concreta dei record di attivazione dipende dall'architettura e dal compilatore, perché questo è uno dei (molti) casi in cui lo standard C non specifica esattamente come un determinato aspetto del linguaggio debba essere implementato, in modo da consentire la realizzazione di compilatori per architetture anche molto diverse e lo sfruttamento degli avanzamenti tecnologici.

4 Regole di scope

Le **regole di visiblità** (**scope rules**) determinano quali identificatori siano visibili in quali parti del codice. Nel caso del C, siccome non è ammesso l'annidamento delle funzioni, tali regole sono particolarmente semplici. In sostanza, *gli identificatori sono accessibili solamente all'interno del blocco nel quale sono dichiarati.*

Un **blocco** è un'istruzione composta (racchiusa tra parentesi graffe) che contiene una o più dichiarazioni di variabili, come ad esempio:

```
{
    int i;
    char nome[10];
    // ...
    for (int i = 0; i < 10 && nome[i] != '\0'; i++)
        printf("%c", nome[i]);
}
```

Un blocco può contenere altri blocchi, cioè l'annidamento dei blocchi è ammesso. Ad esempio:

```
{
    int i;
    {
        char nome[10];
        // ...
        for (int i = 0; i < 10 && nome[i] != '\0'; i++)
            printf("%c", nome[i]);
    }
}
```

Allora, un identificatore rimane visibile *nel blocco in cui è dichiarato e in tutti i blocchi in esso contenuti* (in pratica, la visibilità procede "dall'esterno all'interno"), *a meno di ridefinizioni.*

Si ha una ridefinizione quando si definisce in un blocco una variabile avente lo stesso nome di una già definita all'esterno del blocco. Ad esempio:

```
int x;
// ...
{
    float x;
    // ...
}
// ...
```

Allora, dentro il blocco è visibile la variabile dichiarata all'interno (che "nasconde", "oscura" la variabile esterna), mentre fuori dal blocco (sia prima che dopo) è visibile solo quella dichiarata all'esterno:

```
int x;
// Visibile solo int x
{
    float x;
    // Visibile solo float x
}
// Visibile solo int x
```

(qui le due variabili hanno tipi diversi, ma lo stesso succederebbe se fossero entrambe dello stesso tipo).

4.1 Esempio

Si consideri il seguente frammento di codice:

```
{ // Blocco 1
    int i;
    char nome[10];
    // ...
    { // Blocco 2
        int nome;
        float b;
        i = 4;
    }
    // ...
    { // Blocco 3
        nome[0] = 'p';
        b = 9.7;
    }
}
```

Nel blocco 2:

- il vettore `char nome[10]` dichiarato nel blocco 1 non è accessibile, perché è nascosto dall'omonima variabile `int nome`;
- la variabile `int i` è visibile, perché appartiene a un blocco esterno e non ci sono ridefinizioni.

Invece, nel blocco 3:

- il vettore `char nome[10]` è visibile, perché appartiene a un blocco esterno e non ci sono ridefinizioni;

- non è visibile alcuna variabile di nome `b`, e in particolare non è visibile la variabile `float b`, perché essa è dichiarata nel blocco 2: siccome i blocchi 2 e 3 sono "paralleli", situati allo stesso livello, e non uno dentro l'altro, nessuno di questi blocchi può vedere le variabili dell'altro.

4.2 Variabili locali

Le variabili dichiarate all'interno di una funzione prendono il nome di **variabili locali**, e sono visibili solo all'interno della funzione stessa: una funzione è di fatto un blocco dotato di nome, e tutte le funzioni sono definite allo stesso livello (non essendo ammesso il nesting), quindi nessuna funzione può accedere alle variabili locali di un'altra funzione.

Le variabili locali sono normalmente rappresentate all'interno del record di attivazione della funzione a cui appartengono, ma si vedrà più avanti che è possibile richiederne la memorizzazione in luoghi diversi.

Si potrebbe pensare di poter "condividere" le variabili locali di una funzione restituendone l'indirizzo. Ad esempio:

```
int *f(void) {
    int tmp = 7;
    return &tmp;
}
```

Questo è lecito dal punto di vista dei tipi, perché `&tmp` è effettivamente l'indirizzo di una variabile intera, dunque non si ha un errore in compilazione (al massimo, il compilatore potrebbe emettere un'avvertenza, a meno che non sia stato configurato in modo da trattare le avvertenze come errori). Tuttavia, ciò che avverrebbe se si provasse ad accedere all'indirizzo restituito da `f` è imprevedibile, poiché tale indirizzo fa riferimento al record di attivazione di `f`, che viene deallocato quando `f` termina, e quasi sicuramente sovrascritto da chiamate successive. Allora, il frammento di codice

```
int *pt = f();
printf("Valore di tmp: %d\n", *pt);
```

potrebbe, ad esempio:

- stampare il valore `7`, cioè il valore della variabile locale `tmp`;
- stampare un valore diverso, perché il record di attivazione di `f` è stato sovrascritto;
- generare un errore in esecuzione, se il sistema operativo blocca l'accesso alla zona di memoria corrispondente al record di attivazione deallocato.

4.3 Variabili globali

Si chiamano **variabili globali** le variabili dichiarate all'esterno delle funzioni, cioè allo stesso livello delle dichiarazioni di funzione. Una variabile globale è accessibile, salvo ridefinizioni, in tutte le funzioni:

- definite nello stesso file sorgente, purché siano scritte dopo il punto in cui la variabile viene definita;
- presenti in altri file sorgenti dello stesso programma[3] (salvo restrizioni di visibilità, che si vedrà più avanti come specificare).

In sostanza, è come se ci fosse un unico grande blocco che racchiude tutto il codice, cioè tutte le definizioni di funzioni e variabili globali. Allora, il blocco corrispondente a una funzione è contenuto nel blocco in cui si definiscono le variabili globali, dunque può accedere a esse.

Un esempio è il seguente:

```
void f(void) {
    pippo = 1; // Errore
}

int pippo;

void g(void) {
    pippo = 2; // OK
}

void h(void) {
    pippo = 3; // OK
}
```

Qui la variabile locale `pippo` non è visibile nella funzione `f`, scritta prima che tale variabile venga definita, ma è invece visibile nelle funzioni scritte in seguito, cioè `g` e `h`.

La memoria occupata dalle variabili globali rimane impegnata per tutta l'esecuzione del programma (a differenza della memoria dedicata alle variabili locali di una funzione, che — essendo parte del record di attivazione — viene deallocata appena la funzione termina).

[3] Tipicamente, il codice di un programma grande viene suddiviso in più file di dimensioni e complessità minori.

5 Layout della memoria

Durante l'esecuzione di un programma C, la memoria è logicamente divisa in 4 aree con funzioni diverse:

- l'**area codice**, che contiene le istruzioni del programma;
- l'**area dati globali**, che contiene le variabili globali (e non solo, come si vedrà più avanti);
- l'**area stack**, che contiene appunto lo stack di record di attivazione delle funzioni;
- il **free store** ("memoria libera"), detto anche **heap**, è una zona di memoria dinamica che viene gestita dal programma durante l'esecuzione, tramite apposite funzioni di libreria.

L'area codice e l'area dati globali hanno dimensioni fisse, note al momento della compilazione, mentre l'area stack e il free store hanno dimensioni che variano dinamicamente (ma, ad esempio, molti sistemi operativi impongono un limite massimo alla dimensione dell'area stack).

Il fatto che le variabili locali e quelle globali siano collocate in aree di memoria diverse può essere dimostrato con un programma come il seguente,

```c
#include <stdio.h>

int g;
double h;

int main(int argc, const char *argv[]) {
    float l;
    char m;
    printf("Indirizzo di g: %p\n", &g);
    printf("Indirizzo di h: %p\n", &h);
    printf("Indirizzo di l: %p\n", &l);
    printf("Indirizzo di m: %p\n", &m);
    return 0;
}
```

che potrebbe ad esempio produrre il seguente output (variabile a seconda di architettura, sistema operativo, compilatore, ecc.):

```
Indirizzo di g: 0x55cbede94040
Indirizzo di h: 0x55cbede94048
Indirizzo di l: 0x7ffc22cb7ce4
Indirizzo di m: 0x7ffc22cb7ce3
```

le variabili globali hanno indirizzi tra loro simili, e lo stesso vale per quelle locali, ma invece gli indirizzi delle variabili globali sono molto diversi da quelli delle variabili locali.