Azzolini Riccardo 2021-03-15

Classi di memorizzazione

1 Classi di memorizzazione

Le variabili e le funzioni hanno un attributo che specifica una tra 4 possibili **classi di memorizzazione**. Una classe di memorizzazione può avere significati diversi, a seconda che sia applicata a una variabile locale/globale o a una funzione.

2 Classe di memorizzazione automatica

La classe di memorizzazione **automatica** è quella di default per le variabili locali, ma può essere specificata esplicitamente con la parola riservata `auto`. Lo spazio per memorizzare una variabile automatica viene allocato all'interno del record di attivazione della funzione a cui appartiene la variabile. Di conseguenza, tale spazio rimane occupato solo per la durata dell'esecuzione della funzione, dopodiché viene rilasciato.

Ad esempio, il frammento di codice

```
void f(void) {
    int tmp;
    // ...
}
```

equivale a

```
void f(void) {
    auto int tmp;
    // ...
}
```

e specifica che la variabile `tmp` viene memorizzata nel record di attivazione della procedura `f`.

3 Classe di memorizzazione registro

La classe di memorizzazione **registro**, specificata con la parola riservata `register`, si applica a una variabile locale per suggerire al compilatore di memorizzare tale variabile in un registro del processore, invece che nel record di attivazione (nell'area di stack della memoria). Ciò può essere utile, ad esempio, per accelerare l'accesso a variabili che vengono lette e/o modificate di frequente, come ad esempio il contatore di un ciclo:

```
void f(void) {
    for (register int i = 0; i < SIZE; i++) {
        // ...
    }
}
```

Se il compilatore determina che i registri sono già tutti occupati, oppure se la dimensione della variabile non è compatibile con quella dei registri disponibili — i registri di un processore sono pochi e hanno dimensioni molto limitate — allora la variabile viene gestita come se fosse automatica, ovvero memorizzata nel record di attivazione. Viceversa, il compilatore può decidere autonomamente di memorizzare alcune variabili automatiche nei registri del processore, al fine di ottimizzare le prestazioni del codice. Ad esempio, nel caso del ciclo `for` mostrato prima, la variabile `i` sarebbe molto probabilmente memorizzata in un registro anche se non si specificasse la classe di memorizzazione `register`.

Siccome una variabile di classe registro potrebbe non avere un indirizzo (i registri del processore non hanno appunto indirizzi), non è ammesso applicare a essa l'operatore di estrazione di indirizzo, a prescindere dal fatto che essa sia effettivamente memorizzata in un registro o meno. Ad esempio, il seguente frammento di codice dà un errore in compilazione:

```
void f(void) {
    register int x;
    int *pt = &x; // Errore in compilazione
    // ...
}
```

4 Classe di memorizzazione esterna

La classe di memorizzazione **esterna**, indicata con la parola riservata `extern`, si applica sia alle variabili che alle funzioni.

Nel caso di una variabile globale o di un prototipo di funzione, la classe di memorizzazione esterna indica al compilatore che la definizione completa di tale variabile o funzione è

situata in un altro file (ad esempio, potrebbe essere una definizione di libreria). Ad esempio, nel frammento di codice

```
extern void f(void);
extern int x;

void g(void) {
    // ...
    x = /* ... */;
    // ...
    f();
    // ...
}
```

si indica al compilatore che la procedura `f` e la variabile globale `x`, usate all'interno della procedura `g`, sono definite in un altro file.

Anche nel caso di una variabile locale, la classe di memorizzazione esterna indica che tale variabile è in realtà una variabile globale definita in un altro file, e quindi non è memorizzata nel record di attivazione della funzione in cui compare la definizione. Tuttavia, una variabile esterna così dichiarata è visibile solo all'interno della funzione in cui compare la dichiarazione, e non nel resto del file. Ad esempio, nel frammento di codice

```
extern int x;

void f(void) {
    extern int y;
    // ...
}

void g(void) {
    // ...
}
```

la variabile `x` è visibile sia in `f` che in `g`, mentre la variabile `y` è visibile solo nel corpo di `f`.

5 Classe di memorizzazione statica

La classe di memorizzazione **statica**, indicata con la parola riservata `static`, può essere applicata alle variabili locali, oppure alle variabili globali e alle funzioni.

5.1 Variabili locali

Quando viene applicata a una variabile locale, la classe di memorizzazione **statica** indica che la variabile conserva il proprio valore tra una chiamata e l'altra della funzione in cui è definita. Ad esempio, il programma

```c
#include <stdio.h>

void f(void) {
    static int count = 0;
    printf("f chiamata %d volte\n", ++count);
}

int main(int argc, const char *argv[]) {
    for (int i = 0; i < 3; i++)
        f();
}
```

tiene traccia del numero di chiamate della procedura `f`:

```
f chiamata 1 volte
f chiamata 2 volte
f chiamata 3 volte
```

Da questo esempio si può osservare che l'azione di inizializzazione delle variabili statiche viene eseguita una sola volta, altrimenti il valore di `count` sarebbe reimpostato a 0 a ogni chiamata di `f`.

Lo spazio dedicato a rappresentare queste variabili è occupato per l'intera durata dell'esecuzione del programma, ed è situato nell'*area dati globali*, insieme alle variabili globali. È però importante sottolineare che una variabile locale statica è appunto locale, ovvero rimane inaccessibile al di fuori della funzione che la definisce. Tuttavia, siccome lo spazio dedicato alla variabile non viene deallocato quando la funzione termina, si può restituire l'indirizzo della variabile per consentirne l'accesso dall'esterno (senza i problemi che si presentano nel fare ciò con le variabili automatiche). Ad esempio, il programma

```c
#include <stdio.h>

int *f(void) {
    static int count = 0;
    printf("f chiamata %d volte\n", ++count);
    return &count;
}

int main(int argc, const char *argv[]) {
```

```
    int *pt = f();
    printf("Valore di count: %d\n", *pt);
    for (int i = 0; i < 3; i++) {
        f();
        printf("Valore di count: %d\n", *pt);
    }
}
```

produce il seguente output:

```
f chiamata 1 volte
Valore di count: 1
f chiamata 2 volte
Valore di count: 2
f chiamata 3 volte
Valore di count: 3
f chiamata 4 volte
Valore di count: 4
```

5.2 Variabili globali e funzioni

Nel caso delle variabili globali e delle funzioni, la parola riservata `static` funge da restrizione di visibilità: una variabile globale o funzione con l'attributo `static` è visibile solo all'interno del file in cui è definita (a partire dal punto della definizione), mentre non è possibile accedervi (tramite dichiarazioni `extern`) dagli eventuali altri file dello stesso programma.

Ad esempio, se un file `f1.c` definisce le seguenti variabili globali e funzioni,

```
int x;
static int y;

void f(void) {
    // ...
}

static void g(void) {
    // ...
}
```

allora in un altro file `f2.c` si può accedere alla variabile `x` e alla funzione `f`, ma non alla variabile `y` e alla funzione `g`:

```c
extern int x; // OK
extern void f(void); // OK

extern int y; // Errore
extern void g(void); // Errore
```