12.10 Classificazione delle variabili

Oltre a essere classificata in base al tipo (fondamentale, derivato e derivato composto), in C una variabile può essere anche classificata secondo la visibilità all'interno del programma e secondo la classe di memoria.
Del concetto di visibilità di una variabile abbiamo già trattato a proposito delle funzioni, dove abbiamo imparato a distinguere tra variabili locali e globali. L'ambito di definizione o *scope* di un nome globale si estende dal punto di definizione sino alla fine del file, mentre l'ambito di validità di un nome locale si estende dal punto di definizione sino alla fine del blocco in cui è contenuto.

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

#define DIM 31
#define TAPPO "THE END"

/* semplice struttura che modella una persona */
struct per {
  char cognome[DIM];
  char nome[DIM];
  char ind[DIM];
  int eta;
};
```

```
/* vettore di persone */
struct per anag[] = {
  {"Edison", "Thomas", "Vicolo della Lampadina, 8", 30},
  {"Alighieri", "Dante", "Via del Purgatorio, 13", 21},
  {"More", "Thomas", "Viale Utopia, 48", 39},
  {TAPPO, TAPPO, TAPPO, 0}
};

void vis_per(void);

main()
{
  vis_per();
}

void vis_per(void)
{
char pausa; int i;

for(i=0; strcmp(anag[i].cognome, TAPPO)!=0; i++) {
  printf("\n\n--------------------\n");
  printf("\n\t\tCognome : %s", anag[i].cognome);
  printf("\n\t\tNome : %s", anag[i].nome);
  printf("\n\t\tIndirizzo : %s", anag[i].ind);
  printf("\n\t\tEtà : %d", anag[i].eta);
  printf("\n\n--------------------\n");

  scanf("%c", &pausa);
}
}
```

Listato 12.5 Variabili locali e globali

Consideriamo il Listato 12.5. In questo programma la variabile `anag` è globale mentre le variabili `i` e `pausa` sono locali rispettivamente alla funzione `vis_per`. Finora abbiamo sempre considerato programmi C contenuti in un solo file sorgente. In generale, però, un programma C di media-alta complessità si estende su più file. Per esempio, il programma precedente potrebbe essere diviso su due file, uno contenente il `main` e l'altro la funzione `vis_per` e tutte le altre eventuali funzioni che vorremo far operare su `anag` (Listati 12.6 e 12.7).

```
/* programma principale: MAIN.C */

extern void vis_per(void);

main()
{
  vis_per();
}
```

Listato 12.6 File `MAIN.C`

```
/* file delle funzioni: VIS_PER.C */

#include <stdio.h>
#include <string.h>

#define DIM 31
#define TAPPO "THE END"

/* semplice struttura che modella una persona */
```

```
struct per {
  char cognome[DIM];
  char nome[DIM];
  char ind[DIM];
  int eta;
};

/* vettore di persone */
struct per anag[] = {
  {"Edison", "Thomas", "Vicolo della Lampadina, 8", 30},
  {"Alighieri", "Dante", "Via del Purgatorio, 13", 21},
  {"More", "Thomas", "Viale Utopia, 48", 39},
  {TAPPO, TAPPO, TAPPO, 0}
};

void vis_per(void)
{
char pausa; int i;

for(i=0; strcmp(anag[i].cognome, TAPPO)!=0; i++) {
  printf("\n\n--------------------\n");
  printf("\n\t\tCognome : %s", anag[i].cognome);
  printf("\n\t\tNome : %s", anag[i].nome);
  printf("\n\t\tIndirizzo : %s", anag[i].ind);
  printf("\n\t\tEtà : %d", anag[i].eta);
  printf("\n\n--------------------\n");

  scanf("%c", &pausa);
}
}
```

Listato 12.7 File `VIS_PER.C`

Il nome simbolico `vis_per` dichiarato nel file `MAIN.C` con

```
extern void vis_per(void);
```

non ha all'interno del file una corrispondente definizione. Per tale motivo il nome è dichiarato come `extern`. La parola chiave `extern` si usa per informare il compilatore del fatto che la definizione del simbolo dichiarato `extern` è presente in qualche altro file. Nel nostro esempio la definizione della funzione `vis_per` si trova nel file `VIS_PER.C`.

Possiamo ora definire con esattezza il concetto di variabile globale.
In C una variabile globale è visibile in tutti i file sorgenti che compongono il programma. Per poter far riferimento a una variabile globale che si trova in un altro file occorre però usare la dichiarazione `extern`. Sfortunatamente spesso il *linker* non controlla la congruità fra il tipo e la dimensione della variabile `extern` e quello della corrispondente variabile globale: tale congruenza deve essere mantenuta dal programmatore. Se per esempio avessimo scritto per errore in `MAIN.C`

```
extern char vis_per(int); /* invece di void vis_per(void) */
```

il compilatore e il *linker* non avrebbero avvertito e in esecuzione il programma avrebbe avuto dei problemi.

Se vogliamo fare in modo che una variabile globale – cioè definita fuori di un blocco – rimanga globale ma solo all'interno del file in cui è definita senza diventare nota a tutti gli altri file, allora la variabile globale deve essere dichiarata `static`. Per esempio, se volessimo rendere visibile solo al file `VIS_PER.C` la variabile globale `anag` dovremmo scrivere come nel Listato 12.8.

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

#define DIM 31
```

```
#define TAPPO "X_Y_Z"

struct per {
  char cognome[DIM];
  char nome[DIM];
  char ind[DIM];
  int eta;
};

/* vettore di persone */
static struct per anag[] = {
  {"Edison", "Thomas", "Vicolo della Lampadina, 8", 30},
  {"Alighieri", "Dante", "Via del Purgatorio, 13", 21},
  {"More", "Thomas", "Viale Utopia, 48", 39},
  {TAPPO, TAPPO, TAPPO, 0}
};

void vis_per(void)
{
char pausa; int i;

for(i=0; strcmp(anag[i].cognome, TAPPO)!=0; i++) {
  printf("\n\n--------------------\n");
  printf("\n\t\tCognome : %s", anag[i].cognome);
  printf("\n\t\tNome : %s", anag[i].nome);
  printf("\n\t\tIndirizzo : %s", anag[i].ind);
  printf("\n\t\tEtà : %d", anag[i].eta);
  printf("\n\n--------------------\n");

  scanf("%c", &pausa);
}
}
```

Listato 12.8 Esempio di variabili `static`

```
static struct per anag[];
```

Allora se nel file `MAIN.C` tentassimo di far riferimento alla variabile `anag` mediante una dichiarazione del tipo:

```
extern struct per anag[];
```

non otterremmo altro effetto se non quello di provocare un errore in fase di link.

Come abbiamo anticipato, oltre a essere classificate in base al tipo le variabili C sono classificate anche in base alla *classe di memoria*. La classe di memoria è un attributo dell'oggetto, cioè dell'area di memoria, associato a ogni variabile il quale stabilisce quando una variabile nasce e quando muore, cioè quando è deallocato il corrispondente oggetto. A questo proposito in C esistono due classi di memoria che si riferiscono a due variabili:

- variabili automatiche;
- variabili non automatiche.

Le variabili automatiche sono tutte le variabile locali, cioè dichiarate all'interno di un blocco, che non possiedono l'attributo `static`. Tipicamente variabili automatiche sono gli argomenti di una funzione e le variabili locali di una funzione. Gli oggetti relativi alle variabili automatiche vengono allocati all'atto dell'invocazione della funzione e deallocati quando la funzione termina. Così, nell'esempio:

```
int f(int x, char b)
     {
       int a, b, c;
       char *pc;
       ...
     }
```

le variabili `x`, `y`, `a`, `b`, `c` e `pc` sono tutte automatiche, ovvero vengono create all'atto della chiamata di `f` e sono distrutte quando `f` termina e ritorna il controllo al programma chiamante. Le variabili automatiche, essendo in un certo senso "provvisorie", sono allocate nello *stack*. Poiché le dimensioni dello stack sono limitate, si sconsiglia di utilizzare funzioni con molte variabili automatiche. Per esempio è conveniente evitare funzioni che definiscono al proprio interno strutture dati che allocano grosse fette di memoria. Se invece si vuole estendere il ciclo di vita di una variabile locale, magari definita all'interno di una funzione, anche a dopo che la funzione è ritornata, basta dichiararla `static`. Nel frammento di codice seguente:

```
int f(int x, char b)
     {
        static int a;
        int b, c;
        char *pc;
        ...
     }
```

la variabile `a` è "statica" o – come si dice – non automatica. Essa nasce all'atto dell'esecuzione dell'intero programma e resta attiva per tutto il tempo in cui il programma è in esecuzione.

Le variabili non automatiche sono molto usate per tutte le funzioni che hanno necessità di mantenere memoria di un valore che esse stesse hanno definito. Si consideri per esempio il caso di un semplice generatore di numeri casuali (Listato 12.9). L'esecuzione del programma produce il seguente risultato:

```
Il numero casuale 1 è 18625
     Il numero casuale 2 è 16430
     Il numero casuale 3 è 8543
     Il numero casuale 4 è 43172
     Il numero casuale 5 è 64653
     Il numero casuale 6 è 2794
     Il numero casuale 7 è 27083
     Il numero casuale 8 è 3200
     Il numero casuale 9 è 23705
     Il numero casuale 10 è 34534
```

```
#include <stdio.h>

#define FATTORE 25173
#define MODULO         65535
#define INCREMENTO     13849
#define SEME_INIZIALE      8
#define LOOP 10

unsigned rand(void);

void main()
{
int i;
for(i=0; i<LOOP; i++)
   printf("Il numero casuale %d è %6u\n\n", i+1, rand());
}

unsigned rand(void)
{
static unsigned seme = SEME_INIZIALE;
seme = (FATTORE*seme+INCREMENTO) % MODULO;
return(seme);
}
```

Listato 12.9 Un generatore di numeri casuali

La funzione `rand` è eseguita per 10 volte. La prima volta la variabile locale `seme` è inizializzata con il valore `SEME_INIZIALE`. Poi la funzione calcola il nuovo valore di `seme` pari a 18625 e lo restituisce alla funzione `printf`

che lo visualizza. Quando `rand` è eseguita per la seconda volta, essendo rimasta in vita la variabile `seme` anche dopo il ritorno della funzione, il valore di `seme` è 18625 e il successivo valore è calcolato in 16430. Il procedimento prosegue per tutte le altre iterazioni. Se la variabile `seme` non fosse stata dichiarata `static` la funzione `rand` avrebbe ricalcolato per 10 volte il medesimo valore 18625 (provare per credere).
In generale, si considerano variabili non automatiche le variabili che hanno un ciclo di vita che si estende per tutta la durata dell'esecuzione del programma. Secondo tale definizione, allora, variabili non automatiche sono non soltanto le variabili locali dichiarate `static` ma anche tutte le variabili globali.
Il lettore presti attenzione a come viene usata la parola chiave `static` in C. Infatti abbiamo visto che ne esistono due possibili usi:

1. 1. relativamente alle variabili globali l'attributo `static` fa sì che la variabile sia locale, ma limitatamente al file in cui è definita senza essere visibile a tutti gli altri file del programma;
2. 2. relativamente alle variabili locali l'attributo `static` fa sì che il valore della variabile venga mantenuto anche dopo l'uscita del blocco in cui la variabile è definita, e si conservi sino alla fine del programma.

Per concludere con la classificazione delle variabili, osserviamo che esiste un altro attributo che può caratterizzare una variabile: l'attributo `register`. Questo è sempre riferito a variabili automatiche e dice al compilatore di allocare il relativo oggetto direttamente nei registri macchina dell'unità di elaborazione centrale (CPU). Per mezzo della direttiva `register`, in teoria, si velocizzano le operazioni sulle relative variabili. Per esempio la funzione:

```
void strcpy(register char *S, register char *t)
     {
       while(*s++ = *t++);
     }
```

viene in teoria eseguita più velocemente di

```
void strcpy(char *s, char *t)
     {
       while(*s++ = *t++);
     }
```

In realtà i moderni compilatori sono talmente ottimizzati che da soli provvedono ad allocare nei registri di memoria le variabili più frequentemente usate. Addirittura talvolta, esagerando con il numero delle variabili dichiarate `register`, si può sortire l'effetto contrario e rendere più inefficiente il programma. Infatti il numero dei registri macchina è limitato, e spingere il compilatore ad allocare molte variabili con pochi registri significa perdere tempo in una gestione di scarsa utilità.