Regole di Visibilità

Classi di Memorizzazione

Regole di visibilità

- Gli identificatori sono visibili (accessibili) all'interno del blocco nel quale sono dichiarati
  - –fenomeno di *oscuramento* di variabili esterne
- Utilità per variabili di uso strettamente locale
  - – efficienza nell'uso della memoria
  - – debugging

Regole di visibilità
Blocchi innestati

- Un identificatore dichiarato nel blocco esterno è valido in un blocco interno a meno che il blocco interno lo ridefinisca
  - – I blocchi interni possono essere innestati fino ad una profondità che dipende dal sistema
- Esempio:

```
main()
{
 int a=3; b=5;
 {
  float a=4.6;
  printf("%f %d%", a,b); /*stampa 4.6 e 5*/
 }
}
```

Un esempio di visibilità

```
/*File: visib.c */
main()
{
int count;
if(count < 3) {
  int count = 7;
  printf("%d\n", count);
   count++;
  printf("%d\n", count);
}
printf("%d\n", count);
}
```

- Questa variabile count ha visibilità all'interno della funzione *main*
- Qui si accede a count del *main*
- Questa variabile count ha visibilità nel blocco all'interno del ciclo di *if*
- Queste 3 istruzioni accedono a count dell' *if*
- Qui si accede a count del *main*

Regole di visibilità
Blocchi paralleli

- Un identificatore dichiarato in un certo blocco non è visibile negli altri blocchi paralleli.

Esempio:

```
main()
{
  int a , b;
   ....
 { float a;
   ....
 }
 { float b;
   ....
 }
 .....
}
```

Due blocchi paralleli

Qui a è di tipo *float* e b è di tipo *int*

Qui b è di tipo *float* e a è di tipo *int*

Classi di memorizzazione

- funzioni e variabili hanno 2 attributi:
  - – tipo
  - – classe di memorizzazione
- classi di memorizzazione
  - – auto
  - – extern
  - – register
  - – static

Classe di memorizzazione *auto*

- Le variabili dichiarate all'interno dei corpi delle funzioni sono per *default* automatiche
- Caratteristiche:
  - visibilità all'interno del blocco
  - il sistema alloca memoria per le variabili automatiche all'ingresso del blocco e la rilascia in uscita
- Dichiarazione tipica *(auto* si può omettere):

```
auto int a,b,c;
auto float f;
```

Classe di memorizzazione *extern*

- Le variabili dichiarate all'esterno delle funzioni hanno classe di memorizzazione *extern*
- Usata per trasferire informazioni tra i blocchi e tra funzioni
  - Le variabili extern mantengono il loro valore all'uscita dai blocchi
- Caratteristiche:
  - variabile allocata permanentemente
  - accesso globale da parte delle funzioni dichiarate successivamente
- Possono essere *oscurate* da variabili locali

I esempio della classe *extern*

```
/* File: extern.c */
int a = 1, b = 2, c = 3;
int prod (void);
main()
{
printf ("%d\n", prod());
printf ("%d %d %d \n", a, b, c);
}
int prod(void)
{
int b;
b = 8;
return (a * b * c);
}
```

- Variabili globali
- Prototipo di prod()
- Stampa 24
- Stampa 1, 2, 3
- La variabile locale b nasconde b globale
- prod() restituisce la somma di 1*8*3

II esempio della classe *extern*

```
/* File: file1.c */
int a = 1, b = 2, c = 3;
int prod (void);
main()
{
printf ("%d\n", prod());
printf ("%d %d %d \n", a, b, c);
}
/* File: file2.c */
int prod(void)
{
extern int a;
int b;
b = 8;
return (a + b + c);
}
```

- Questo programma composto da due file: *file1.c* e *file2.c* ha lo stesso comportamento di quello precedente
- Variabili globali
- In *file2.c* la variabile a è *extern* (si trova definita altrove)

Classe di memorizzazione *register*

- Comunica al compilatore di *cercare* di ottimizzare l'accesso alla variabile
  - il compilatore cerca di memorizzarla in un registro ad alta velocita'
  - il compilatore ha pochi registri a disposizione
- Se il compilatore non ci riesce viene allocata nella classe *auto*
- Utile per:
  - variabili indice dei cicli
  - parametri di funzioni
- Esempio:

```
register int i;
```

Classe di memorizzazione *static*

- Una variabile *static* viene allocata una sola volta e non viene deallocata all'uscita dal blocco
  - mantiene il valore precedente al rientro nel blocco
- Utile per
  - le variabili interne ad un blocco
  - le variabili *extern*

Un esempio della classe *static*

```
/* File: static.c */
void f(void)
main()
{
int i;
for(i=1:i<=10;i++)
  f();
}
void f(void)
{
static int count = 1;
count ++ ;
if (count > 4) {
printf ("f eseguita piu' di 4 volte");
}
```

- count e' *static*
- l'inizializzazione di count a 1 viene eseguita solo la prima volta che è eseguita la funzione f
- Ad ogni chiamata successiva alla quarta, la funzione f stampa "f eseguita piu' di 4 volte"

Variabili *statiche esterne*

- Variabili esterne con visibilita' ristretta
  - non sono visibili alle funzioni precedentemente definite nel file o definite in file differenti
- Utili per
  - fornire un meccanismo di privatezza tra i moduli di un programma

Esempio di variabili statiche esterne

```
.......
int f1(void)
{
.....
}
static int v;

void f2(void)
{
.......
}
```

- La funzione f1 non può accedere a v
- Dichiarazione di v come *static*
- La funzione f2 può accedere a v
- v non e' visibile in altri file.

Inizializzazione di *default*

- le variabili *extern* e le variabili *static* non inizializzate esplicitamente nel programma, vengono inizializzate a zero dal sistema
- le variabili *auto* e *register* non vengono inizializzate dal sistema
  - alcuni sistemi inizializzano anche queste variabili ma e' bene non fare affidamento su questa possibilita'

Ricorsione

- Una funzione che richiami se stessa viene detta *ricorsiva*
- Il C permette di richiamare ricorsivamente qualsiasi funzione
- Un esempio di funzione ricorsiva:

```
void forever(void);
main()
{
forever();
}
void forever(void)
{
printf ("Vado avanti per sempre");
forever();
}
```

Esempio di ricorsione

```
/* File: sommatoria.c */
int sum (int);
main()
{
printf("%d\n", sum(4));
}

int sum (int n)
{
if (n == 1)
return (1);
else
return (n + sum (n -1) );
}
```

Stampa 4+3+2+1=10

- Se n è 1 restituisce 1
- Altrimenti restituisce la somma di n e del risultato della chiamata di sum(n-1)

Esempio di ricorsione

```
/* File: sommatoria.c */
int sum (int);
main()
{
printf("%d\n", sum(4));
}

int sum (int n)
{
if (n == 1)
return (1);
else
return (n + sum (n -1) );
}
```

• La variabile n è un parametro ed e quindi di classe *auto*

• La variabile n viene "creata" ogni volta ad ogni chiamata della funzione

Come funziona

```
/* File: sommatoria.c */
int sum (int);
main()
{
printf("%d\n", sum(4));
}

int sum (int n)
{
if (n == 1)
return (1);
else
return (n + sum (n -1) );
}
```

main ()
printf("%d\n", sum(4));

sum(4)
... return (4 + sum (3));

sum(3)
... return (3 + sum 2));

sum(2)
... return (2 + sum (1));

sum(1)
... return (1);

Efficienza della ricorsione

Vantaggio:

• programmi più eleganti, piu' facili da scrivere e da comprendere

• numero inferiore di variabili

Svantaggio:

• i valori dei parametri e delle variabili relativi a ciascuna chiamata ricorsiva sono conservati in uno *stack*

– costoso sia in termini di spazio che in termini di tempo

Efficienza della ricorsione

```
/* File: fibonacci_r.c */
int a = 1, b = 2, c = 3;
long fib_ric(int);
main()
{
.....
printf ("%d\n", fib_ric(5));
}
long fib_ric(int n)
{
If (n<=1)
  return(n);
return (fib_ric(n-1) + fib_ric(n-2));
}
```

Questo programma stampa il numero di Fibonacci f_5

La funzione fib_ric calcola il numero di Fibonacci f_n ricorsivamente

Efficienza della ricorsione