Le funzioni

Una **funzione** può dirsi l'astrazione o generalizzazione del concetto di variabile.

```
a = b * f (parametri di f) – c / g(parametri di g) + d
```

La definizione di una funzione può servire per:

- mettere in evidenza il risultato della funzione e non le operazioni che ne consentono la determinazione;
- riusare la stessa funzione in più occasioni, anche se con diversi valori degli argomenti (o parametri).

Ogni funzione è caratterizzata da:

- ✓ numero, tipo e dominio degli argomenti;
- ✓ tipo e co-dominio del risultato.

Esempio: ***funzione che calcola il cubo di un numero***

```
#include   <stdio.h>

double cubo(float);


main()
{
float      a;
double     b;
printf     ("Inserisci un numero: ");
scanf("%f", &a);

b = cubo(a);
printf("%f elevato al cubo è uguale a %f", a, b);
}


double cubo(float c)
{
  return (c*c*c);
}
```

dichiarazione o ***prototipo***

Presuppone esista da qualche parte la definizione della funzione

invocazione o **chiamata**

definizione

ritorno

La dichiarazione (*prototipo*) di funzione

All'interno di un programma in C una funzione può essere chiamata se è *definita* o se è ***dichiarata***.

La **dichiarazione** (o **prototipo**) di una funzione non è altro che la definizione della funzione, seguita dal simbolo ;

```
double cubo   (float);
```

La presenza del prototipo prima della chiamata facilita l'operazione di compilazione.

In particolare l'attività del compilatore può essere facilitata dall'introduzione del prototipo nella parte dichiarativa globale del programma o nella parte dichiarativa del main o nella parte dichiarativa dei sottoprogrammi in cui viene chiamata la funzione.

La definizione di una funzione

Poiché non è possibile predefinire tutte le funzioni di cui si può aver bisogno per trattare i vari tipi di dati, il C consente di costruire (o ***definire***) una funzione ed invocare (o ***chiamare***) una funzione.

È possibile perciò definire nuove funzioni da "chiamare" nelle espressioni.

La ***definizione di una funzione*** si compone di:

- un *FunctionHeader* (o *Testata*), che a sua volta contiene:
  - il tipo del risultato
  - il nome o identificatore della funzione
  - la lista degli argomenti ai quali la funzione si applica
- una *Local Declarative Part*, che contiene la dichiarazione degli elementi disponibili nella funzione
- una *Executable Part* (o *corpo della funzione*), racchiusa tra i simboli **{ }**, che contiene le istruzioni eseguibili che costituiscono la funzione.

La struttura di una funzione è molto simile a quella del **main**. È questo il motivo per cui essa si dice anche che è un ***sottoprogramma di tipo funzione***.

La dichiarazione dei parametri

La lista degli argomenti di una funzione è racchiusa tra parentesi tonde e consiste di una sequenza di **parametri formali**, separati da una virgola, ciascuno costituito da un identificatore di tipo e da un identificatore di oggetto di quel tipo.

Gli argomenti si dicono parametri formali, perchè essi non hanno un valore proprio, ma servono solo per identificare le caratteristiche degli argomenti che la funzione richiede per poter determinare un risultato.

Ogni parametro formale è un identificatore di tipo seguito da un identificatore.

Una funzione non può restituire array o funzioni, ma può restituire un puntatore a qualsiasi tipo.

Le dichiarazioni locali di una funzione (*Local Declarative Part*)

... specificano tutti gli oggetti (tipi, costanti, variabili, ***prototipi*** di funzioni) che saranno utilizzati nella parte eseguibile della funzione.

La parte eseguibile (corpo) di una funzione (*ExecutablePart*)

... è costruita con le stesse regole sintattiche del corpo del main.

Nel suo interno si trova l'istruzione **`return`**, che indica la fine dell'esecuzione della funzione e assegna un valore alla variabile risultato. Tale valore è quello dell'*`espressione`* specificata nell'istruzione `return`.

`return espressione;`

La *`espressione`* è ovviamente dello stesso tipo del risultato definito per la funzione.

Nel corpo di una funzione ci possono essere più istruzioni `return`. La prima che viene eseguita determina con la sua *`espressione`* il valore del risultato della funzione e provoca la fine dell'esecuzione.

Se nel corpo di una funzione non c'è nessuna istruzione `return` o se comunque nessuna istruzione `return` viene eseguita, la funzione termina in corrispondenza del simbolo `}` che segna la fine del corpo della funzione.
In tal caso il risultato della funzione è indefinito e viene segnalato un messaggio di errore.

La chiamata di una funzione

L'uso di una funzione - e quindi la sua "chiamata" - avviene nell'ambito di una espressione e consiste in:

- ✓ **nome della funzione**
- ✓ la **lista dei parametri attuali**, ossia i valori effettivi degli argomenti in base ai quali deve essere determinato il risultato della funzione.

Il numero dei *parametri attuali* è esattamente uguale al numero dei *parametri formali* contenuti nella definizione della funzione. Il tipo di ciascun parametro attuale deve essere compatibile con il tipo del corrispondente parametro formale.

Ogni parametro attuale può essere rappresentato da un'espressione, che può eventualmente far riferimento ad un'altra chiamata di funzione.

Ogni parametro attuale viene associato al corrispondente parametro formale contenuto nella definizione della funzione.

Errori possibili: ordine, numero o discordanza di tipo.

Passaggio dei parametri "per valore"

Il passaggio dei parametri avviene ***per valore***: all'invocazione ogni parametro formale è inizializzato con il valore del parametro attuale.

Non sempre occorre la perfetta corrispondenza di tipo tra parametri formali e attuali, avvengono infatti *conversioni implicite di tipo*:

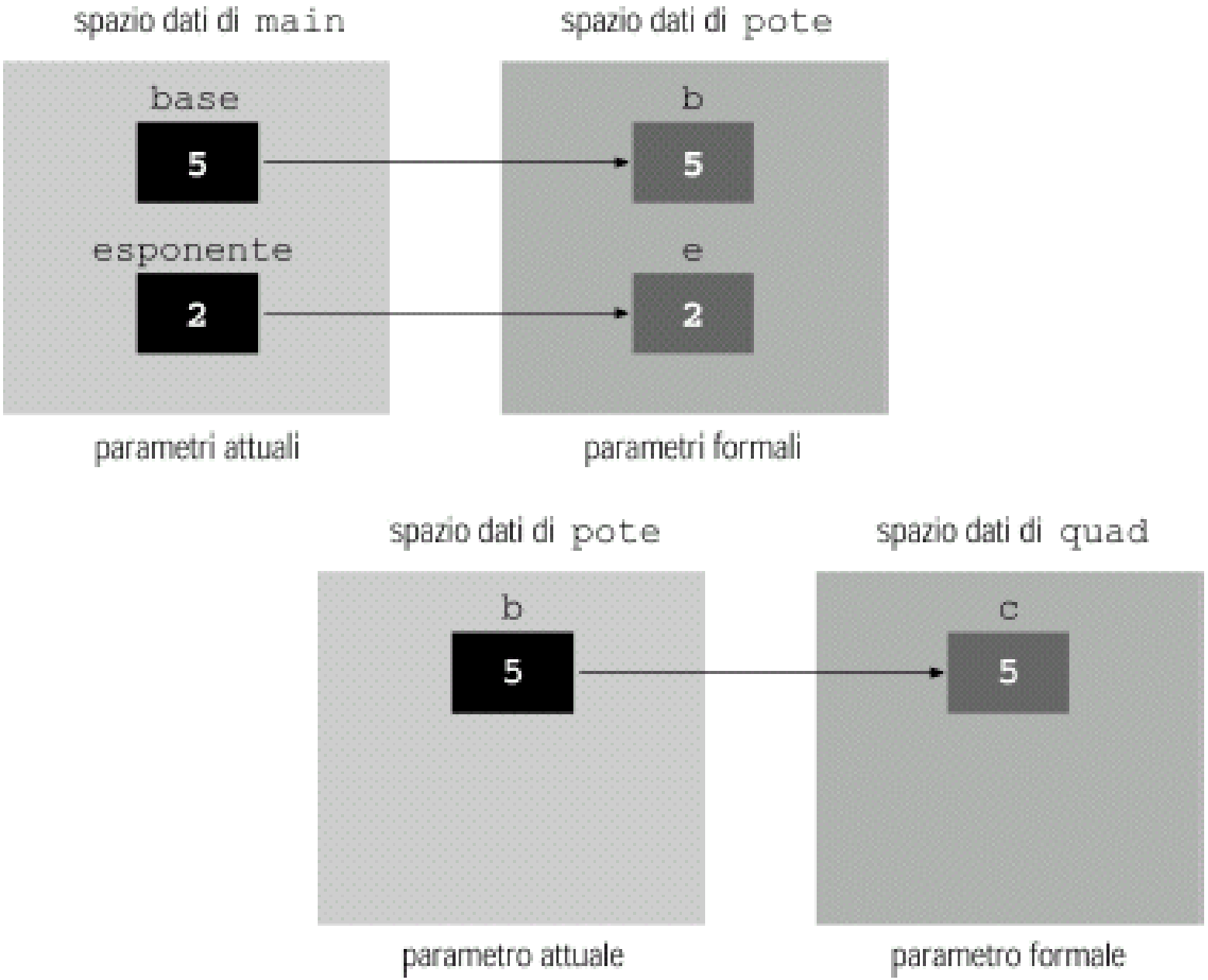
I parametri attuali float sono convertiti in double prima di essere passati alla funzione.
I parametri attuali char e short int sono convertiti in int.

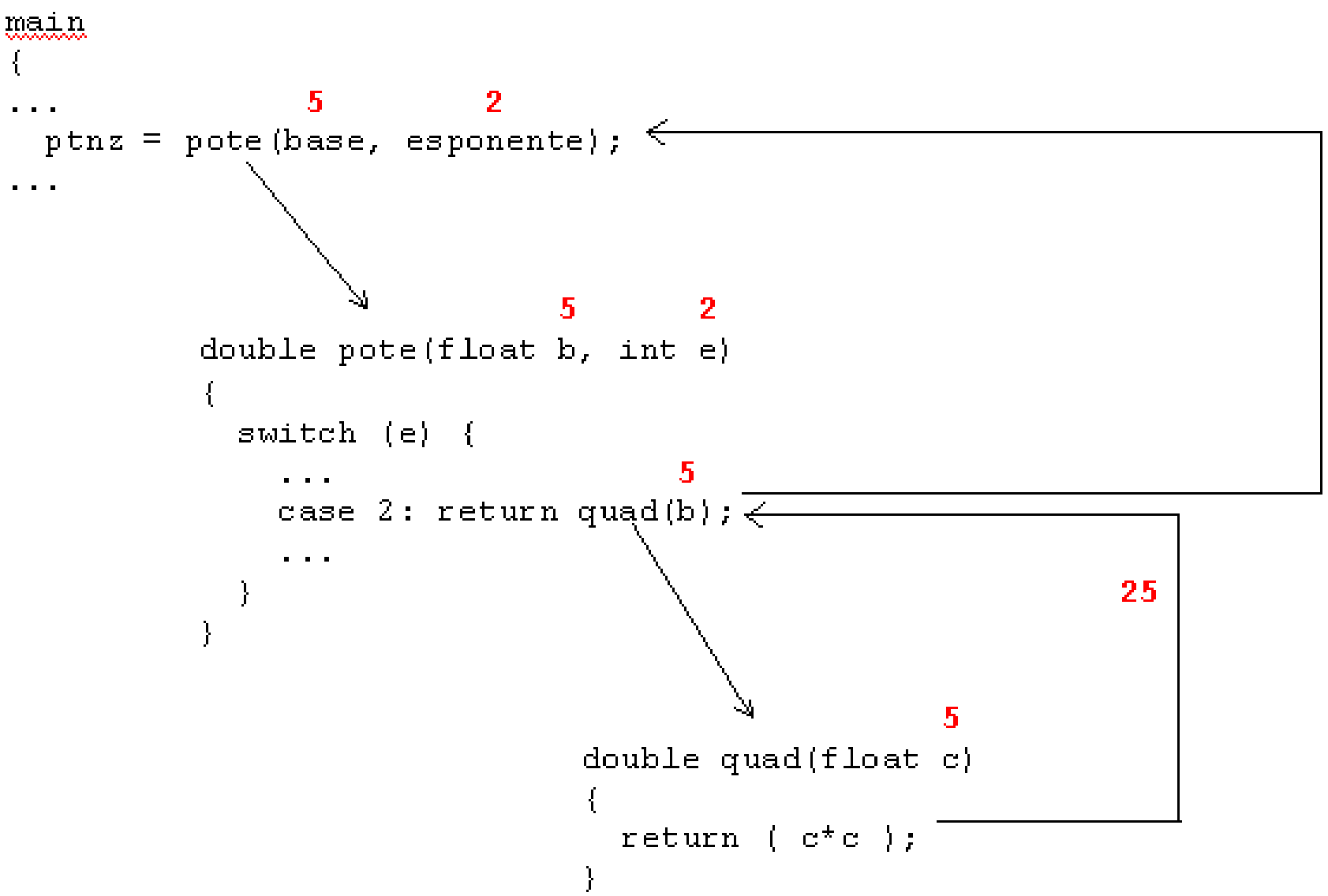
Non è consentito il passaggio di parametri di tipo array.

Passaggio implicito: definire una variabile globale sia alla funzione chiamante sia a quella chiamata.

Esempio del passaggio dei parametri "per valore"

```
#include <stdio.h>
double quad (float);  double cubo (float);
double quar (float);  double quin (float);
double pote (float, int);
main ()   {
            int base, esponente; double ptnz;
            printf (" Inserire base: " ); scanf ("%d", &base);
            printf (" Inserire esponente (0-5): ");scanf ("%d", &esponente);
            ptnz = pote( base, esponente);
            if (ptnz == -1) printf ("Potenza non prevista\n");
            else printf ("La potenza %d di %d e' %f\n", esponente, base, ptnz);
double quad(float c){return(c*c);}
double cubo(float c){return(c*c*c);}
double quar(float c){return(c*c*c*c);}
double quin(float c){return(c*c*c*c*c);}
double pote(float b, int e){
switch (e) {
  case   0: return (1);
  case   1: return (b);
  case   2: return (quad( b ));
  case   3: return (cubo( b ));
  case   4: return (quar( b ));
  case   5: return (quin( b ));
  default : return (-1);
}}
```


spazio dati di main
spazio dati di pote
base
5
esponente
2
b
5
e
2
parametri attuali
parametri formali
spazio dati di pote
spazio dati di quad
b
5
c
5
parametro attuale
parametro formale


main
{
...
ptnz = pote(base, esponente);
...
5
2
double pote(float b, int e)
{
switch (e) {
...
case 2: return quad(b);
...
}
}
5
2
5
25
double quad(float c)
{
return (c*c);
}
5

void

Per le funzioni che non restituiscono alcun valore, e funzioni che non hanno parametri si ha a disposizione un "tipo" speciale detto **void**.

```
#include <stdio.h>
#define DIM_INT 16
void stampa_bin ( int );
main(){
          char resp[2];   int  num;   resp[0] = 's';
             while( resp[0] == 's' ) {
             printf("\nInserisci un intero positivo: ");
             scanf("%d", &num);
             printf("La sua rappresentazione binaria: ");
             stampa_bin( num );
             printf("\nVuoi continuare? (s/n): ");
             scanf("%s",resp);
             }
      }
void stampa_bin( int v ){
  int i, j; char a[DIM_INT];
  if (v == 0)  printf("%d", v); else
                {  for( i=0; v != 0; i++)
                   { a[i] = v % 2;
                         v /= 2;
                   }
                   for(j = i-1 ; j >= 0; j--) printf("%d", a[j]);
                 }
                       }
```

void

void è anche usato per specificare l'assenza di parametri

Dunque `main` dovrebbe essere dichiarata `void main(void)`

Le procedure

Una **procedura** (o sottoprogramma procedurale) può dirsi l'astrazione o generalizzazione del concetto di istruzione.

L'esecuzione della procedura può cioè non limitarsi a produrre un risultato da utilizzare per il calcolo di un'espressione, ma comportare la ***modifica dello stato*** di celle di memoria dello stesso programma chiamante.

La "chiamata" di una procedura ha lo scopo di richiedere l'esecuzione di un blocco di istruzioni, che faranno uso degli elementi *visibili* alla procedura (***contesto***):

- i parametri attuali contenuti nella chiamata, i quali sostituiranno i corrispondenti parametri formali contenuti nella definizione della procedura;
- gli elementi dichiarati nella sua parte dichiarativa;
- gli elementi dichiarati nella parte dichiarativa globale del programma.

Una procedura serve per riusare le stesse operazioni in più occasioni, anche se con diversi valori degli argomenti (o parametri).

Definizione e chiamata di procedure

La ***definizione di una procedura*** è identica a quella di una funzione, salvo che il tipo di risultato indicato nella testata è **void**, indicante appunto l'assenza di un risultato.

```
void        InsertElem (int  elemento)
```

La ***chiamata di una procedura*** è un'istruzione contenente il nome della procedura e, tra parentesi tonde, i parametri attuali "passati" alla procedura, separati tra loro da virgole. L'istruzione termina con un **;**

```
InsertElem ( i );
```

Il numero dei *parametri attuali* è esattamente uguale al numero dei *parametri formali* contenuti nella definizione della procedura. Il tipo di ciascun parametro attuale deve essere compatibile con il tipo del corrispondente parametro formale.

Una procedura, una volta terminata, produce l'effetto di restituire il controllo all'istruzione immediatamente successiva alla chiamata, senza produrre alcun risultato oltre quello già prodotto dalle istruzioni della sua parte eseguibile.

Esempio

```
#include <stdio.h>
#define     num_max     20
.......
int      alfa [num_max];
int      num_elem;
.......
main ( )
{
     num_elem = 0;
     .......
     InsertElem (i);
     .......
}

void     InsertElem (int     elemento)
     {
     if (num_elem = = num_max) printf (“lista piena”);
               else     {   num_elem++;
                         alfa[num_elem – 1] = elemento;
                         }
     }
```

Passaggio dei parametri "per indirizzo"

Perché una procedura possa modificare gli elementi del suo contesto e per far sì che i parametri attuali identifichino un particolare elemento del contesto su cui operare, è necessario un meccanismo di passaggio dei parametri diverso da quello della copia.

Il meccanismo opportuno è quello che va sotto il nome di **passaggio dei parametri "per indirizzo"**.

Esso utilizza il costruttore di tipo puntatore per la definizione dei parametri formali.

Esempio

`scambia(x, y);` con passaggio dei parametri per **valore**

x e **y** sono copiati nei parametri formali **a** e **b** che sono scambiati ma i valori originali di **x** e **y** rimangono inalterati poiché il passaggio è per valore!

```
#include <stdio.h>
void scambia(int, int);
main()
{
int x, y; x = 8; y = 16;
printf("Prima dello scambio\n");
printf("x = %d, y = %d\n", x, y);
scambia(x, y);
printf("Dopo lo scambio\n");
printf("x = %d, y = %d\n", x, y);
}

void scambia(int a, int b)
{
int temp;
temp = a; a = b; b = temp;
}
```

Per ottenere un effetto su **x** e **y**, la procedura **scambia** deve ricevere gli indirizzi, anziché i valori, delle variabili.

Esempio

`scambia(&x, &y);` con passaggio dei parametri per **indirizzo**

L'invocazione è modificata in modo da passare l'indirizzo delle variabili da scambiare.
Vengono passati come parametri attuali gli indirizzi di **`x`** e **`y`**.

```
#include <stdio.h>
void scambia(int *, int *);
main()
{
int x, y;
x = 8; y = 16;
printf("Prima dello scambio\n");
printf("x = %d, y = %d\n", x, y);
scambia(&x, &y);
printf("Dopo lo scambio\n");
printf("x = %d, y = %d\n", x, y);
}

void scambia(int *a, int *b)
{
int temp;
temp = *a; *a = *b; *b = temp;
}
```

`(int *a, int *b)`

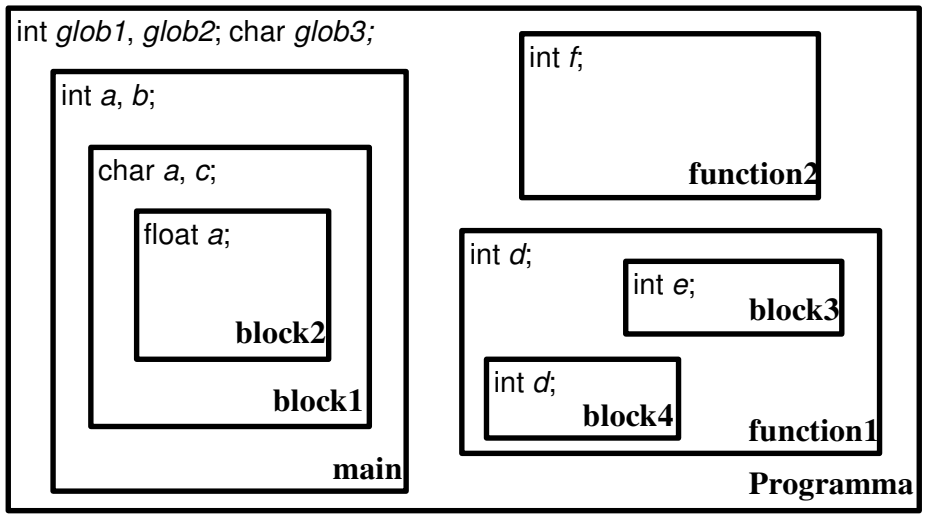
I parametri formali **`a`** e **`b`** sono puntatori. Nell'esempio puntatori a **`x`** e **`y`**.

Regole di visibilità delle variabili

Un *programma globale* in C è un insieme di variabili globali del programma (***ambiente globale***) e di funzioni, tra le quali una deve essere identificata come **main**. L'esecuzione ha inizio con la prima istruzione eseguibile della funzione **main**.

Ogni funzione o procedura possiede poi un ***ambiente locale***. Al di dentro di una funzione o procedura vi possono essere poi diversi blocchi, paralleli o annidati, che possono al loro interno contenere una parte dichiarativa, che costituisce l'***ambiente di blocco***.

Il **modello a contorni** fornisce una forma grafica per comprendere le regole secondo cui possono essere usati i vari oggetti (costanti, variabili, tipi e funzioni) dichiarati nei diversi ambienti e per comprendere le ***regole di visibilità*** degli oggetti presenti nei diversi ambienti.



Esempio di programma per rappresentare il "modello a contorni" *(1/2)*

```
/* parte dichiarativa globale */
#include <stdio.h>
int     glob1, glob2;
char        glob3;
int     function1(int param1, float param2);

    main ( )
{
/* parte dichiarativa locale del main */
    int a, b;
    int function2(int param3, float param2);
    {
/* parte dichiarativa locale di block1 del main */
        char        a, c;
        {
/* dichiarativa locale del blocco annidato block2*/
            float   a;
        }
    }
}

.......
.......
```

Esempio di programma per rappresentare il "modello a contorni" *(2/2)*

```
.......
.......
    int function1(int param1, float param2)
{
/* parte dichiarativa locale di function1 */
    int d;
    {
/* parte dichiarativa locale di block3 di function1 */
        int e;
    }
    {
/* dichiarativa locale di block4 parallelo di function1 */
        int d;
    }
}


    int function2(int param3, float param2)
{
/* parte dichiarativa locale di function2 */
    int f;
}
```

Le regole di visibilità

Si può dichiarare più volte lo stesso identificatore, anche con significati diversi, purché in ambienti diversi.

Gli ***oggetti dell'ambiente globale*** possono essere visti da tutte le funzioni ed i blocchi che costituiscono il programma.

Se in una funzione o in un blocco esistono più definizioni di un identificatore, vale quella dell'ambiente più vicino al punto di utilizzo.

Gli ***oggetti dell'ambiente locale*** possono essere visti dalle istruzioni della funzione e da quelle dei blocchi contenuti nella funzione.

Gli ***oggetti dell'ambiente di un blocco*** possono essere visti dalle istruzioni del blocco e da quelle dei blocchi annidati.

Uso di parametri di tipo array

L'uso di un array come parametro formale comporta in realtà il passaggio per valore dell'indirizzo di base (i.e. del primo elemento) dell'array.

In altri termini non viene effettuato, come ci si attenderebbe, il passaggio per valore degli elementi attuali costituenti l'array.

Ciò è dovuto al fatto che il nome di una variabile di tipo array è di fatto un puntatore al primo elemento dell'array.

Quindi sono equivalenti le tre definizioni di funzione:

```
1) typedef int vettore[max-num];
   ......
   int sommatoria(vettore  alfa, int num_elem)

2) int sommatoria(int  *alfa, int num_elem)

3) int sommatoria(int  alfa[], int  num_elem)
```

Sono possibili perciò chiamate che sommino solo alcuni elementi (ovviamente consecutivi) dell'array.

Si noti come le operazioni sul parametro formale che sono effettuate nella funzione in realtà siano svolte sul parametro attuale.

Tempo di vita (o durata) delle variabili

Quando viene creata e quando viene distrutta una variabile?

Variabili fisse o statiche: i loro valori sono mantenuti anche all'esterno del loro ambito di visibilità per tutta la durata del programma.

Variabili globali. Variabili del main.

Variabili automatiche: i loro valori non sono mantenuti all'esterno del loro ambito di visibilità per tutta la durata del programma.

Variabili locali e variabili di blocco.

Tali variabili possono essere fisse se le si fa precedere dalla parola chiave **static**.

Esempio:

```
int funzione (int  param1, float  param2)
{   static  int somma;
........
........
}
```

Lo spazio di memoria delle variabili **static** è allocato quando viene chiamata per la prima volta la funzione e non viene rilasciato al termine dell'esecuzione della funzione. Una variabile di tal tipo può svolgere il ruolo di contatore o sommatore.

Gestione della memoria a tempo di esecuzione (run-time)

Il codice macchina e i dati risiedono entrambi in memoria, ma in zone separate:

- ✓ la memoria per il codice macchina è fissata a tempo di compilazione
- ✓ la memoria per i dati (in particolare per le variabili automatiche) cresce e decresce dinamicamente durante l'esecuzione: viene gestita a **pila**

Una **pila** (o **stack**) è una struttura dati con accesso LIFO: Last In First Out = l'ultimo entrato è il primo ad uscire (Es.: pila di piatti).

Il sistema gestisce in memoria la **pila dei Record Di Attivazione** (RDA):

- ⇨ per **ogni chiamata di funzione** viene creato un nuovo RDA in cima alla pila
- ⇨ al **termine della chiamata** della funzione il RDA viene rimosso dalla pila

Ogni RDA contiene:

- ✓ le locazioni di memoria per i parametri formali (se presenti)
- ✓ le locazioni di memoria per le variabili locali (se presenti)
- ✓ l'indirizzo dell'istruzione di ritorno

Anche gli ambienti locali dei blocchi vengono allocati/deallocati sulla pila.

Record di attivazione

Esempio:

```
    int f(int);
    main()
{
    int x, y, z;
    x=10;
    y=20;       /* PUNTO 1: blocco principale */
    z = f(x);   /* PUNTO 2: prima chiamata di f */
{               /* PUNTO 3: uscita da f e ingresso nel blocco annidato*/
    int x=50;
    y=f(x);     /* PUNTO 4: seconda chiamata di f */
    z=y;        /* PUNTO 5: uscita da f */
}
    ...         /* PUNTO 6: uscita dal blocco */
}
    int f(int a)
{
    int z;
    z = a + 1;
    return z;
}
```

Record di attivazione

Evoluzione della pila:

PUNTO 1

x	10
y	20
z	?

PUNTO 2

a	10
z	?
x	10
y	20
z	?

PUNTO 3

x	50
x	10
y	20
z	11

PUNTO 4

a	50
z	?
x	50
x	10
y	20
z	11

PUNTO 5

x	50
x	10
y	51
z	11

PUNTO 6

x	10
y	51
z	51