# Dal Pascal al C

Parte 4ª

# **Record** e **strutture**

Il tipo **record** del Pascal ha il suo analogo C nel tipo **struttura**.

Nel Pascal un record viene dichiarato tramite la parola chiave **record**.

Una **struttura** è un aggregato di campi, ognuno con un suo tipo ed identificatore.

La dichiarazione di una **struttura** viene indicata nel C con la parola chiave **struct** seguita dall'elenco dei campi, tra parentesi graffe, e dal nome assegnato alla struttura.

# Dichiarazione di variabili di tipo **record**

```
program dichiarazionerecord;
var  compleanno:  record
                      giorno: integer;
                      mese:   integer;
                      anno:    integer
                    end;
      poligonale:     array [1..100]  of  record
                                            ascissa:   real;
                                            ordinata: real
                                          end;
begin
    compleanno.giorno:=25;
    poligonale[5].ascissa:=6.0;
    poligonale[5].ordinata:=6.4;
end.
```

# Dichiarazione di variabili di tipo **struttura**

```
main ()  {
struct {
            int  giorno;
            int  mese;
            int  anno;
          } compleanno;
struct {
            double  ascissa;
            double  ordinata;
          } poligonale [100];
     compleanno.giorno=25;
     poligonale[4].ascissa=6.0;
     poligonale[4].ordinata=6.4;
}
```

Nella dichiarazione della variabile **compleanno**, il tipo precede l'identificazione di variabile.

La sintassi del C per accedere ai campi delle strutture è analoga alla sintassi del Pascal per accedere ai campi del record.

La variabile **poligonale** è un array di record nel Pascal e un array di strutture nel C.

# Dichiarazione dei tipi **record** e **struttura**

```
program dichiarazionerecord2;
type  data =  record
                  giorno: integer;
                  mese:   integer;
                  anno:    integer
                end;
var   compleanno:  data;
begin
    compleanno.giorno:=25
end.
```

```
main ()  {
typedef struct {
                  int  giorno;
                  int  mese;
                  int  anno;
                  } data;
data  compleanno;
    compleanno.giorno=25;
}
```

- Le variabili di tipo struttura possono essere inizializzate con le stesse modalità usate per gli array.
- Come in Pascal, se **a** e **b** sono due variabili di tipo struttura dichiarate tramite lo stesso identificatore di tipo è possibile scrivere l'istruzione **a=b**. La compatibilità è nominale e non strutturale.
- La forma sintattica senza la lista dei campi, e quindi con la sola parola chiave **struct** (o **union**) e l'identificatore, può essere utilizzata per definire il solo nome di una struttura prima che essa venga definita completamente. Ciò è utile se si deve utilizzare l'identificatore di struttura prima che la struttura sia definita completamente.

# Modello di struttura

Si è visto come definire variabili di tipo struttura e tipi struttura, quest'ultimi attraverso l'uso della parola chiave **typedef**.

Il C fornisce un ulteriore strumento per definire strutture. Nel C è possibile definire una struttura ed associare ad essa un nome senza che tale nome sia un nome di variabile o un identificatore di tipo. Il nome così definito è una sorta di **modello di struttura** lo si può riutilizzare sia per definire tipi che per definire variabili.

# **Modello di struttura** per definire una **variabile**

```
main ()  {
struct  modellodidata {
            int  giorno;
            int  mese;
            int  anno;
            } ;
struct  modellodidata  compleanno;

      compleanno.giorno=25;

}
```

La parola **struct** viene usata una prima volta per definire la composizione di una struttura (la struttura **modellodidata**), senza attribuirle un tipo e senza associarla direttamente ad una variabile. Poiché **modellodidata** non è un identificatore di tipo, nella dichiarazione della variable è necessario usare nuovamente la parola chiave **struct**.

La dichiarazione di variabile dell'esempio può essere così interpretata: definisci una variabile, avente nome **compleanno**, di tipo struttura, la cui composizione in termini di campi è quella indicata in **modellodidata**.

# **Modello di struttura** per definire un **tipo**

```
main ()  {
struct  modellodidata {
                int  giorno;
                int  mese;
                int  anno;
                } ;
typedef  struct  modellodidata  data;
data   compleanno;

      compleanno.giorno=25;

}
```

I dati vengono definiti in tre fasi successive: (1) si definisce la composizione della struttura **modellodidata**; (2) usando la **typedef** si definisce il tipo **data** che è una struttura la cui composizione è descritta da **modellodidata**; (3) si definisce la variabile compleanno di tipo **data**.

Generalizzando, l'identificatore di un **modello di struttura**, preceduto dalla parola chiave **struct**, può essere utilizzato ovunque sia necessario specificare un tipo.

# Strutture di strutture

```
main ()  {
struct  modellodidata {
                int  giorno;
                int  mese;
                int  anno;
                 } ;
struct   {
                char  nome[20];
                char  luogonascita[20];
                struct  modellodidata  datanascita;
          }  p;

      p.datanascita.mese = 2;
}
```

# **Record varianti** e **union**

Nella definizione di tipo struttura esiste nel C la possibilità di utilizzare la parola chiave **union** al posto della parola chiave **struct**.

La parola chiave **union** consente di definire strutture simili ai **record varianti** del Pascal.

# Strutture definite tramite **union**

```
main ()
 {
union  {
            short    s;
            float     f;
            double d;
          }  sfd;

      sfd.s=25;
      sfd.d=6.02E23;
}
```

Al momento della dichiarazione della variabile **sfd** il compilatore riserva per tale variabile uno spazio di memoria pari al massimo spazio tra quelli necessari per rappresentare il solo campo **s** o il solo campo **f** o il solo campo **d**. In pratica i tre campi vivono nella variabile in alternativa. Il programmatore usa la variabile come di tipo **short**, **float** o **double** in funzione del riferimento esplicito al nome del campo. Quindi **sfd.s=25** assegna a **sfd** il valore **25** considerando **sfd** come una variabile di tipo **short**; l'istruzione successiva modifica il valore di **sfd** assegnandole il valore **6.02** per **10** elevato a **23** considerando **sfd** come **double**. Dopo la seconda assegnazione se accediamo nuovamente a **sfd.s** otteniamo un valore non significativo.

Le strutture dichiarate con la parola chiave **union** servono a fare economia di spazio.

# Strutture con campi di bit

Il C consente di definire strutture i cui campi sono specificati in termini della loro dimensione espressa in numero di bit (**campi di bit**).

```
struct  {
        unsigned int voto: 5;
        unsigned int giorno: 5;
        unsigned int mese: 4;
         int anno;
         }  esame;
```

La variabile esame di tipo struttura ha quattro campi. I campi **voto** e **giorno** sono composti da **5 bit**, il campo **mese** ha **4 bit** e il campo **anno** è un normale campo di tipo **intero**.

La possibilità di definire campi di bit consente spesso di risparmiare memoria. D'altro canto l'accesso a campi bit può diminuire l'efficienza del programma.

# Puntatori

- Nel C si ritrovano tutte le caratteristiche del tipo puntatore del Pascal.
- Viceversa il tipo puntatore del C ha varie potenzialità che non hanno analogie nel Pascal.
- Nel Pascal il tipo puntatore è utilizzato solo per gestire indirizzi di locazioni della zona di memoria usata come **heap** mentre nel C un puntatore può essere utilizzato per indirizzare una zona qualsiasi della memoria.
- Nel C per i puntatori è definita una aritmetica ed essi sono in stretto rapporto con gli array.

# I puntatori

Con **strutture dinamiche di dati** si intende, in generale, la possibilità di utilizzare strutture di dati, alle quali viene assegnato lo spazio necessario in memoria centrale quando serve e durante l'esecuzione del programma, incrementando e decrementando tale spazio in base alle esigenze dell'esecuzione in corso.

Le tecniche di programmazione per la gestione dinamica delle strutture di dati si basano sul concetto di **puntatore**.

Una variabile di tipo **puntatore** contiene l'indirizzo di memoria di un'altra variabile.

L'utilizzo dei puntatori è molto importante, ma richiede molta attenzione in quanto può essere facilmente causa di errori di compilazione e di esecuzione.

# Gli operatori * e &

Per capire l'uso dei puntatori nel linguaggio C è necessario chiarire il significato degli operatori unari ***** e **&**.

L'operatore ***** applicato ad una espressione restituisce il contenuto della variabile il cui indirizzo di memoria è dato dal risultato dell'espressione.

L'operatore ***** può essere applicato solo ad una espressione che restituisca un valore di tipo puntatore.

Nella definizione del tipo puntatore l'asterisco del C svolge un ruolo analogo al simbolo **^** del Pascal.

L'operatore **&**, applicato ad una variabile, restituisce l'indirizzo di memoria (un puntatore) in cui tale variabile è allocata. L'operatore **&** può essere applicato ad una variabile di tipo qualunque.

La funzione **scanf("%d",&num)**, ad esempio, richiede la lettura di un dato nella variabile **num**; perché questo sia possibile è necessario fornire allo **scanf** l'indirizzo della locazione di memoria in cui memorizzare il dato letto.

# L’operatore *

Per i tipi di variabili, sia predefiniti nel linguaggio che definiti dall’utente, è possibile creare un puntatore anteponendo il carattere ***** (**asterisco** o **star**) al nome del puntatore nella dichiarazione.

```
int *puntatore;

char *s;

struct data *ricorrenza;
```

Il primo è un puntatore in grado di contenere l’indirizzo di una variabile intera e per tale ragione è detto puntatore di tipo intero, il secondo è un puntatore di tipo carattere e il  terzo è un puntatore per una struttura definita dall’utente.
Indipendentemente dal tipo, tutti i puntatori occupano uno spazio di memoria della stessa dimensione, quella necessaria per la memorizzazione di un indirizzo.

# L’operatore &

Per conoscere l’indirizzo di memoria di una variabile si usa l’**operatore &** (**ampersand** o **e commerciale**).

```
int  num = 5;
int  *pinteri;
pinteri = &num;
```

**pinteri** è un puntatore di tipo intero, **&num** restituisce l’indirizzo di memoria dove è memorizzata la variabile **num**. Dopo l’assegnazione **pinteri** punta alla memoria occupata da **num**. L’uso della lettera **p** come iniziale del nome del puntatore serve ad evidenziare megliole variabili di questo tipo all’interno del codice.
L’operatore **&** sottintende la frase **“indirizzo di”**. Supponiamo che l’indirizzo della variabile **num** sia **126** e contenga il numero **5**; il puntatore **pinteri**, dopo l’istruzione di assegnamento, contiene il valore **126**.

# Il tipo puntatore nel Pascal

```
program usodeipuntatori;
type
    puntrecord = ^recordlista;
    recordlista = record
                        info:  real;
                        next:  puntorecord
                  end;
var piniz :  puntorecord;
begin
    new(piniz);
    piniz^.info := 5.5;
    piniz^.next := nil
end.
```

# Il tipo puntatore nel C

```
#include<malloc.h>
main()  {
struct modellorecordlista  {
                        double info;
                        struct modellorecordlista  *next;
                        } ;
tydef struct modellorecordlista recordlista;
typedef recordlista  *puntrecord;
puntrecord  piniz;
            piniz=(puntrecord)malloc(sizeof(recordlista));
            piniz->info=5.5;
            piniz->next=NULL;
}
```

# Confronto dei due programmi precedenti (1)

I due programmi descrivono una struttura di dati lista semplice e costruiscono una lista formata da un unico elemento.

- Le dichiarazioni sono sostanzialmente simili. Nel programma in C si ricorre al concetto di modello di struttura.
- La primitiva del Pascal **new(piniz)**, che alloca dinamicamente un record al programma e lo fa puntare da **piniz**, è sostituita nel C dalla ben più complessa **piniz=(puntrecord)malloc(sizeof(recordlista))**. La funzione **malloc** svolge lo stesso ruolo della **new**, è necessario però specificare come argomento la dimensione della porzione di memoria che si intende allocare. L'operatore **sizeof**, applicato all'identificatore di tipo **recordlista**, restituisce come risultato il numero di byte occupati da una struttura di tipo **recordlista**. Rimane da chiarire il ruolo della coppia di parentesi a sinistra della **malloc**. La funzione **malloc** restituisce un puntatore di tipo indefinito (tipo puntatore a **void**); per poterlo utilizzare è necessario convertire tale tipo al tipo che ci interessa, nel caso dell'esempio il tipo **puntrecord**.

# Confronto dei due programmi precedenti (2)

- La direttiva **#include<malloc.h>** è necessaria in ogni programma in cui si usa la funzione **malloc**.
- I simboli **^.** del Pascal sono sostituiti dai simboli **->** per cui per denotare il campo **info** del record puntato dalla variabile **piniz** occorre scrivere **piniz->info**. Naturalmente si può anche scrivere **(*piniz).info**, utilizzando l'operatore ***** (le parentesi sono necessarie a causa della priorità per il fatto che il punto ha la priorità maggiore rispetto all'asterisco).
- La costante **nil** del Pascal è sostituita in C dalla costante **NULL**.

# Esempio puntatore

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

int main()
{
   int alfa = 4;
   int beta = 7;
   int *pointer;

   pointer = &alfa;
   printf("alfa -> %d, beta -> %d, pointer -> %dn \n", alfa, beta, pointer);
   beta = *pointer;
   printf("alfa -> %d, beta -> %d, pointer -> %dn \n", alfa, beta, pointer);
   alfa = pointer;
   printf("alfa -> %d, beta -> %d, pointer -> %dn \n", alfa, beta, pointer);
   *pointer = 5;
   printf("alfa -> %d, beta -> %d, pointer -> %dn \n", alfa, beta, pointer);
   system("pause");
}
```

# Commento all’esempio precedente

Assumiamo di avere due variabili, **alfa** e **beta** ed un puntatore di nome **pointer**; assumiamo anche che **alfa** risieda alla locazione di memoria "**100**", **beta** alla locazione "**200**" e **pointer** alla locazione "**1000**" e vediamo, eseguendo il codice proposto nell’esempio precedente, che si ottengono i valori corretti di **alfa** e **beta** (**4** e **7**) e l'**indirizzo** di **alfa** memorizzato nel **puntatore** (ad es. **100**):

**alfa -> 4, beta -> 7, pointer -> 100**

poi assegniamo a **beta** il valore dell'oggetto puntato da **pointer** (**alfa**), e quindi il valore **4**:

**alfa -> 4, beta -> 4, pointer -> 100**

successivamente assegniamo ad **alfa** il valore memorizzato in **pointer** e quindi l'**indirizzo** di memoria di **alfa** stesso,

**alfa -> 100, beta -> 4, pointer -> 100**

continuiamo memorizzando in **alfa** (l'oggetto puntato da **pointer**) il valore **5**,

**alfa -> 5, beta -> 4, pointer -> 100**

# Le funzioni

Il C, come la maggior parte dei linguaggi di programmazione, permette di definire ed utilizzare sottoprogrammi. Lo strumento che il C mette a disposizione per questo scopo è la funzione. Le caratteristiche principali delle funzioni del C sono:

- Nel C, contrariamente a quanto accade nel Pascal, non esistono procedure. L'unico strumento per scrivere sottoprogrammi è la funzione.
- Le funzioni del C non possono essere dichiarate in modo annidato. Tutte le dichiarazioni di funzione sono sullo stesso livello. Esse precedono o seguono il **main**.
- L'attribuzione di un valore ad una funzione non avviene come nel Pascal attraverso una assegnazione all'identificatore della funzione ma attraverso una istruzione specifica: l'istruzione **return**. Quando viene eseguita una istruzione **return**, il controllo viene restituito al programma chiamante.
- I parametri formali di una funzione C possono essere soltanto parametri valore, non esistono parametri variabile/riferimento. Lo strumento utilizzato in C per ovviare a questa limitazione è il puntatore. Ogni volta che una variabile **a** deve essere modificata all'internodi una funzione, viene passato alla funzione il puntatore ad **a**. La funzione opera su **a** attraverso il puntatore. L'unica eccezione riguarda gli **array**: nel C un **array** è sempre un parametro variabile.

# Sintassi delle funzioni C (1)

La sintassi generale di una funzione (function) nel C è:

***tipo di dato restituito*** **nome della funzione (*elenco dei parametri*) {**

**istruzioni;**

**return *valore restituito*;**

**}**

Poiché la funzione restituisce un valore, occorre specificare, prima del nome che identifica la funzione, il **tipo** del valore restituito. Dopo il nome della funzione, le parentesi tonde servono a contenere l'elenco degli argomenti passati alla funzione, detti **parametri**. Le istruzioni che formano la funzione sono racchiuse tra parentesi graffe e rappresentano il codice che viene eseguito alla chiamata della funzione.

Il corpo della funzione contiene come ultima istruzione, la parola **return** seguita dal valore o dalla variabile contenente il risultato restituito dalla funzione. L'istruzione **return** provoca il ritorno del controllo alla funzione principale o, in generale, alla funzione chiamante.

Se il tipo restituito dalla funzione non è specificato, si assume, per default, che il tipo sia **int**.

# Sintassi delle funzioni C (2)

Se si vuole invece che la funzione non restituisca alcun valore, bisogna specificare, come tipo di dato restituito, il tipo **void** (*vuoto* o *privo*):

**void** nome della funzione (***elenco dei parametri***)

L'elenco dei parametri segue la sintassi della dichiarazione delle variabili; i parametri sono separati dalla virgola e per ciascun parametro deve essere indicato il tipo (diversamente dalla dichiarazione delle variabili):

***tipo*** **nome della funzione (*tipo1* nome1, *tipo2* nome2, …)**

Il tipo **void** può essere usato anche nella dichiarazione dei parametri: il tipo **void** scritto tra parentesi tonde indica che nessun parametro viene passato alla funzione. Questo è il valore di default; infatti **void f (void);** si può anche scrivere **void f ();**.

Per rendere più efficace la lettura e l'interpretazione del codice, può essere utile ripetere, dopo la parentesi graffa che chiude il corpo della funzione, il nome della funzione stessa come commento (racchiuso tra i delimitatori ***/*…*/***) in modo da rendere più evidente dove inizia e dove termina la funzione.

# Esempi di funzioni C

```
int Somma (int a, int b) {
    int x;
    x = a + b;
    return x;
}
```

```
void StampaSomma (int a, int b) {
    int x;
    x = a + b;
    printf("Tot. =  %d  \n", x);
}
```

```
void Stampa (void) {
    printf("stampa di prova  \n");
    printf("fine della funzione \n");
}
```

Codifica di una funzione che esegue la somma di due numeri interi, ricevuti come parametri, e restituisce il valore calcolato.

Se la una funzione ha come tipo di ritorno il tipo **void**, l'istruzione **return** non è seguita da alcun valore o espressione, in quanto il tipo **void** non restituisce nulla alla funzione chiamante; in questo caso l'istruzione **return**, in fondo alla implementazione, può essere omessa.

Ecco un esempio di una funzione che non restituisce alcun valore e che non riceve alcun parametro come argomento: il tipo restituito è **void** e il tipo **void** scritto tra parentesi tonde indica che nessun parametro viene passato alla funzione.

# Chiamata di una funzione

La chiamata di una funzione può essere effettuata da un qualunque punto del programma, dopo averla dichiarata e definita: occorre specificare il nome della funzione seguito dall'elenco, tra parentesi tonde, dei valori da assegnare ai parametri della funzione:

**nome della funzione (*elenco dei valori da passare ai parametri*);**

Al momento della chiamata, il compilatore esegue il controllo di corrispondenza tra i parametri specificati nella definizione della funzione e i valori passati alla funzione. Un esempio di chiamata alla funzione **Somma**:

```
int totale;
int subtot1;
int subtot2;
…
totale = Somma (subtot1, subtot2);
…
```

# La funzione **main**

- **main ()** è a tutti gli effetti una funzione: questo è il motivo della presenza della coppia di parentesi tonde dopo la parola **main**. Tuttavia la funzione **main** ha una caratteristica specifica: è la funzione che viene eseguita per prima all'avvio del programma.
- L'istruzione **return** può essere presente anche nella funzione **main**, in tal caso il valore viene restituito direttamente al sistema operativo, che è il programma chiamante della funzione.
- Per completezza quindi, sarebbe meglio scrivere la parola **int** prima di **main** per indicare il tipo di valore restituito, e alla fine del **main** scrivere l'istruzione **return** seguita dal valore **0** (zero).
- L'uso di **return** con il valore **0** è il modo più comune per indicare la terminazione di un programma che non ha trovato errori durante la **esecuzione**.
- Le variabili dichiarate all'esterno del **main** e delle funzioni si chiamano **variabili globali**, in contrapposizione di quelle dichiarate all'interno delle funzioni o all'interno del **main**, che prendono il nome di **variabili locali**.

# La ricorsione

```
function fatt(n: integer):
    integer;
begin
    if n=0
    then fatt := 1
    else
        fatt := fatt(n-1)*n
end;
```

```
long int fatt (int n) {
    if (n==0)
        return 1;
    else
        return fatt(n-1)*n;
}
```

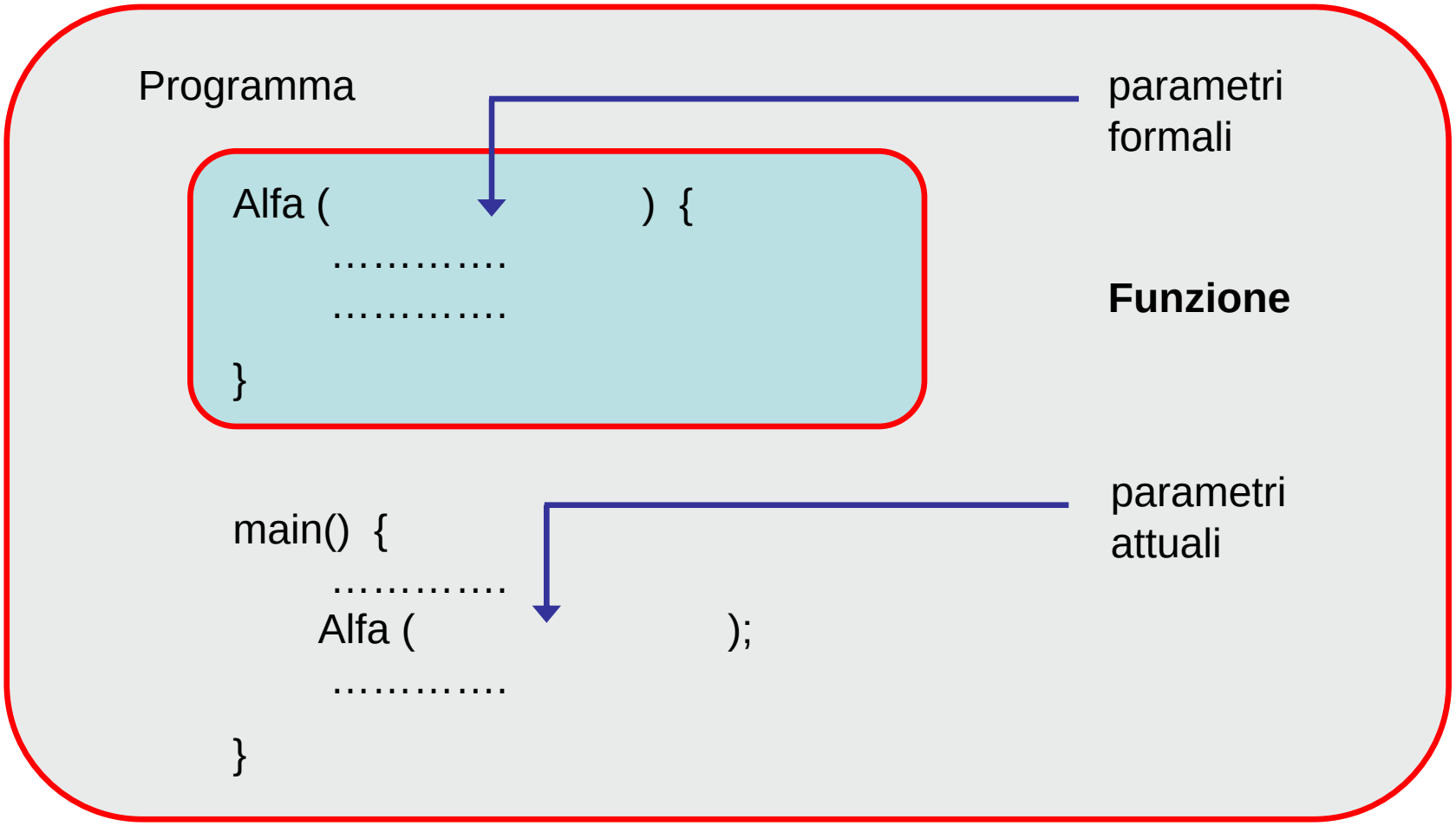
Il risultato del prodotto fattoriale è di tipo **long int** e si ottiene moltiplicando un **long int** per un **int**. Il prodotto fattoriale **fatt(n-1)*n** restituisce un **long int** secondo le regole del C sulle conversioni implicite di tipo.

# Funzioni con parametri

L’uso delle funzioni risponde all’esigenza fondamentale di costruire programmi ben organizzati e strutturati secondo la metodologia **top-down**.

Grazie alle funzioni si può utilizzare uno stesso gruppo di istruzioni in programmi diversi, proprio come accade con le funzioni predefinite. Conviene quindi rendere parametrici i valori utilizzati dalle funzioni, in modo da ricevere come argomenti i valori passati dalla funzione **main** o da un’altra funzione chiamante. Le variabili dichiarate all’interno del **main** o di una funzione vengono dette **variabili locali**; le funzioni operano sui parametri che vengono loro passati dal **main** o dalla funzione chiamante (questa operazione viene detta **passaggio di parametri**).

Le variabili indicate nella intestazione della funzione si chiamano **parametri formali**; le variabili che forniscono i valori ai parametri si chiamano **parametri attuali**.


Programma
parametri formali
Alfa ( ) {
……
……
}
Funzione
parametri attuali
main() {
……
Alfa ( );
……
}

# Passaggio dei parametri

L’operazione, con la quale vengono associate le due liste dei parametri attuali e formali, detta **passaggio dei parametri**, può avvenire in due modi diversi:

- **passaggio dei parametri per valore**, quando i valori delle variabili della funzione principale vengono ricopiati nei parametri della funzione; i cambiamenti effettuati sui parametri formali durante l’esecuzione della funzione non influenzano i valori delle variabili nella funzione chiamante (**main** o altra funzione). Nel linguaggio C il passaggio di parametri per valore avviene indicando nell’intestazione della funzione il tipo e il nome di ciascun parametro formale.
- **passaggio dei parametri per referenza** (o **per indirizzo**), quando i parametri attuali e formali fanno riferimento alla stessa cella di memoria centrale, cioè allo stesso **indirizzo** di memoria; questo è il caso in cui il programmatore vuole che i cambiamenti di valore ai parametri, durante l’esecuzione della funzione, influenzino i valori delle variabili corrispondenti nella funzione chiamante. Per passare l’indirizzo di una variabile è necessario utilizzare l’**operatore &**, che restituisce appunto l’indirizzo della variabile alla quale è applicato.

# Le procedure del Pascal

```
program sommanumeri;
var a, b, c: integer;
    procedure Somma2 (x,y:integer; var z:integer);
    begin
        z:=x+y
    end;
begin
    a:=5;
    b:=23;
    Somma2(a,b,c);
    writeln(c)
    end.
```

Nel C, non essendo possibile definire parametri formali di tipo variabile/riferimento, per modificare il valore di variabili del programma chiamante è necessario usare i puntatori.

# Come le funzioni del C simulano le procedure

```
#include<stdio.h>
void Somma2 (int x, int y, int *z) {
    *z=x+y;
}
int main() {
int a, b, c;
    a=5;
    b=23;
    Somma2(a,b,&c);
    printf("%d",c);
    return 0;
}
```

Il comportamento della procedura **Somma2** del Pascal è ottenuto nel C con la seguente tecnica. Il tipo della funzione è **void**, il che vuol dire che la funzione non restituisce alcun valore. Il parametro formale **z** è definito come puntatore ad un intero. Così **z** consente di accedere alla locazione di memoria in cui è allocata la variabile **c** (variabile del programma chiamante) e quindi consente di scrivere direttamente in quella locazione di memoria il risultato della somma. Si noti a questo l'uso dell'operatore ***** nella istruzione ***z=x+y;**, il significato dell'istruzione è: calcola la somma di **x** e **y** e metti il risultato nella locazione di memoria il cui indirizzo è contenuto in **z** (puntata da **z**).

Si noti come nella istruzione che chiama la funzione (l'istruzione **Somma2(a,b,&c);**) l'indirizzo della (il puntatore alla) variabile **c** sia ottenuto tramite l'operatore **&**.

# Prototipi di funzione

Nel C è possibile dichiarare una funzione senza specificare l'insieme delle istruzioni che la compongono; tale dichiarazione consente di rendere nota l'intestazione di funzione lasciando ad un secondo momento la specifica dell'intera funzione. La sintassi è la seguente:

***tipo di dato restituito*** **nome della funzione (*elenco dei parametri*)**

La **dichiarazione della funzione**, che si intende utilizzare nel programma, di norma va posta in testa al programma, immediatamente dopo la sezione dedicata alle direttive e alla dichiarazione delle variabili globali del programma.

Tutte le funzioni devono essere dichiarate, fatta eccezione per la funzione **main** che non necessita di dichiarazione, in quanto, essendo la prima funzione ad essere eseguita, non richiede controlli alla chiamata.

La **definizione** (o **implementazione**) **della funzione**, con le istruzioni che la compongono, viene normalmente posta dopo la funzione **main**.