
Struttura di un Programma C

Un programma C ha in linea di principio la seguente forma:

- **Direttive per il preprocessore**
- **Definizione di tipi**
- **Prototipi di funzioni**, con dichiarazione dei tipi delle funzioni e delle variabili passate alle funzioni)
- **Dichiarazione delle Variabili Globali**
- **Dichiarazione Funzioni**, dove ogni dichiarazione di una funzione ha la forma:

```
Tipo NomeFunzione(Parametri)
{
   Dichiarazione Variabili Locali
   Istruzioni C
}
```

```
#include <stdio.h>

typedef struct point {
            int   x;    int   y;
            };
int       f1(void);
void    f2(int i, double g);


int   sum;


int main(void)
{
      int j;
      double g=0.0;
      for(j=0;j<2;j++)
            f2(j,g);
      return(2);
}

void f2(int i, double g)
{
      sum = sum + g*i;
}
```

Identificatori

- Sono **identificatori** per il C che possono essere usati per indicare variabili, funzioni, etichette e dati di tipo definito dall'utente.

- Un identificatore deve essere costuituito da **uno o più** caratteri.

- Il **primo carattere** di un identificatore (quello più a sinistra) deve essere una **lettera** o una **sottolineatura** (underscore _).

- I **caratteri successivi** al primo possono essere **numeri o lettere o sottolineature**.

Identificatori ammessi	Identificatori **NON** ammessi
Num	1Num
_Num	
num	num;pippo
n_um	num$!

- Non sono ammessi caratteri di punteggiatura o altro, che hanno significati speciali.

- Il C, a differenza del Pascal, è **case-sensitive**, ovvero **considera caratteri minuscoli e caratteri maiuscoli come differenti**. Quindi l'identificatore "NUMERO" è diverso dall'identificatore "numero" e da "Numero".

- Non sono ammessi caratteri di punteggiatura o altro, che hanno significati speciali.

Ogni identificatore, usato sia per dentificare variabili sia per indicare funzioni, deve essere diverso dalle parole riservate utilizzate per il linguaggio C, e deve essere diverso anche dai nomi di variabili o funzioni delle librerie utilizzate durante la fase di linking.

Tipi di Dati Semplici

Il C mette a disposizione i seguenti dati di tipo semplice:

Tipo	Size (byte)	Valore minimo	Valore Massimo	Equivalente in Pascal
signed char	1	-128	+127	char
unsigned char	1	0	255	---
short int	2	-32768	+32767	integer
unsigned short int	2	0	65535	---
long int	4	-2^{31}	$+2^{31}-1$	longint
unsigned long int	4	0	$+2^{32}-1$	---
float	4	$-3.2 \times 10^{\pm 38}$	$+3.2 \times 10^{\pm 38}$	real
double	8	$-1.7 \times 10^{\pm 308}$	$+1.7 \times 10^{\pm 308}$	extended
void	0	non definito	non definito	---

Osservazioni importante:

- Sui sistemi UNIX tutte le variabili dichiarate "**int**" sono considerate **long int**", mentre "short int" deve essere dichiarato esplicitamente.
- In alcune architetture, inoltre, il dato di tipo int è costituito da un numero maggiore di byte.

- E' importante notare che in C **non esiste un tipo di variabile booleano.**
- Come variabili di tipo booleano si possono utilizzare variabili "char", "int", "long int" sia signed che unsigned.
- Ciascuno di questi dati, quando viene valutato dal punto di vista booleano, è valutato **FALSO quando assume valore 0** (ZERO), se invece assume **un valore diverso da zero è valuato come VERO**.
  Per esempio, la seguente istruzione condizionale **if(-1)** i=1;
  esegue l'assegnamento perchè il valore **-1** è considerato VERO.

Il tipo void rappresenta un tipo di dato indefinito, e ha due funzioni: serve ad indicare che una funzione non restituisce nessun valore, e serve per definire un puntatore che punta ad un dato generico.

Variabili

Tutte le variabili devono essere dichiarate prima di essere usate.
La **dichiarazione delle variabili** è così fatta:

Tipo ElencoVariabili;

dove Tipo è uno dei tipi di dati ammessi dal C, e ElencoVariabili è composto da uno o più identificatori validi separati da una virgola.
In questo modo ogni identificatore che compare in ElencoVariabili diventa una variabile di tipo Tipo.

esempi:

```
int       i;        /* i è una variabile di tipo int. */
long int  l1, l2;   /* l1 ed l2 sono long int */
float     f,g,x,y;  /* f, g, x, y sono variabile in virgola mobile */
```

Le variabili assumono caratteristiche diverse, in particolare caratteristiche di visibilità (scope) da parte delle funzioni , in dipendenza della posizione in cui avviene la dichiarazione.

A seconda della posizione in cui avviene la dichiarazione, si distinguono tre tipi di variabili:

- Variabili **Locali**.
- Parametri **Formali**.
- Variabili **Globali**.

Variabili Locali

Definiamo **Blocco di Istruzioni** una sequenza di istruzioni C racchiusa tra una parentesi graffa aperta ed una parentesi graffa chiusa.
Il corpo di una funzione (il codice C che implementa una funzione) è un caso particolare di Blocco.

Esempi di Blocchi:

Corpo di Funzione

```
int funcA (double f)
{
   int j;    /* corretto */
   printf("corpo di funcA")
   int K;   /* ERRORE */
}
```

Interno di ciclo for

```
if ( 1 )
{
   int j;
    printf("ciclo for")
}

func(J);    ERRORE
qui J NON e' visibile
```

Ovunque, usando il Trucco aperta-chiusa {}

```
printf("codice C");
{
   int j;
    printf("ciclo for")
}
```

Una variabile Locale può essere dichiarata **dentro un qualunque blocco**, ma in questo caso sempre e solo **all'inizio** del blocco, cioè mai dopo che nel blocco sia stata scritta un'istruzione diversa da una dichiarazione), ed in tal caso:

- la variabile verrà detta **Locale al blocco**,
- potrà essere acceduta solo dall'interno del blocco stesso,
- cioè non è visibile fuori dal blocco,
- e avrà un ciclo di vita che inizierà nel momento in cui il controllo entra nel blocco, e terminerà nel momento in cui il controllo esce dal blocco.

Le variabili locali sono caricate sullo stack quando il controllo entra nel blocco considerato, e vengono eliminate quando il controllo esce dal blocco in cui sono state dichiarate.
Quando una variabile è dichiarata nel corpo di una funzione, è locale alla funzione, e assomiglia alle variabili locali del Pascal.

Variabili come Parametri Formali

Sono le variabili che definiscono, nell'implementazione di una funzione, i parametri passati come argomenti alla funzione.
Sono esattamente equivalenti ai parametri formali delle funzioni o procedure del Pascal.
Per default i dati di tipo semplice sono passati per valore, come in Pascal.
Invece i dati di tipo matrice sono passati per puntatore (la modalità **var** del pascal).

C	Pascal

int func(float f , int i)
{
  printf ("param: f=%f i=%d\n,f,i);
}
	func(real : f, i : integer) :integer ;
	begin
	  writeln('param: f=',f, ' i=', i);
	end;

Come per le variabili locali, anche i Parametri Formali

- potrà essere acceduta solo dall'interno della funzione in cui è stata dichiarata,
- cioè non è visibile fuori dalla funzione,
- avrà un ciclo di vita che inizierà nel momento in cui il controllo entra nella funzione, e terminerà nel momento in cui il controllo esce dal blocco.
-

I parametri Formali vengono caricati sullo stack quando il controllo entra nel blocco considerato, e vengono eliminati quando il controllo esce dal blocco in cui sono state dichiarate. Se il Parametro è passato per puntatore, è il puntatore ad essere caricato sullo stack.

Variabili Globali e Specificatore extern

Le variabili Globali sono quelle variabili che sono dichiarate **fuori da tutte le funzioni**, in una posizione qualsiasi del file.
Una tale variabile allora verrà detta **globale**, perchè

- potrà essere acceduta da tutte le funzioni che stanno **nello stesso file** ma sempre **sotto alla dichiarazione della variabile stessa**,
- potrà essere acceduta da tutte le funzioni che stanno **in altri file in cui esiste una dichiarazione extern per la stessa variabile,** ma sempre **sotto alla dichiarazione extern** della variabile stessa,
- e avrà durata pari alla durata in esecuzione del programma.

Per default, una variabile globale NomeVariabile è visibile da tutti i moduli in cui esiste una dichiarazione di variabile extern di NomeVariabile, ovvero una dichiarazione siffatta:

extern tipo NomeVariabile;

che è la solita dichiarazione di variabile preceduta però dalla parola extern.

Una tale dichiarazione dice al compilatore che:

1. nel modulo in cui la dichiarazione extern è presente, la variabile NomeVariabile non esiste,
2. ma esiste in qualche altro modulo,
3. e che il modulo con la dichiarazione extern è autorizzato ad usare la variabile,
4. e quindi il compilatore non si deve preoccupare se non la trova in questo file,
5. perchè la variabile esiste da qualche altra parte.
6. Sarà il Linker a cercare in tutti i moduli fino a trovare il modulo in cui esiste la dichiarazione **senza extern** per la variabile NomeVariabile.

La **variabile NomeVariabile viene fisicamente collocata solo nel modulo in cui compare la dichiarazione senza extern**, (che deve essere uno solo altrimenti il Linker non sa cosa scegliere) e precisamente nel punto in cui compare la dichiarazione. Nei moduli con la dichiarazione extern invece rimane solo un riferimento per il linker.

Protezione dagli Accessi esterni al modulo: **Variabili Globali** e specificatore **Static**

Se vogliamo che una certa variabile globale NomeVariabile, collocata in un certo file, non sia accedibile da nessun altro modulo, dobbiamo modificare la sua dichiarazione in quel modulo, facendola precedere dalla keyword **static** ottenendo una dichiarazione di questo tipo.

static tipo NomeVariabile;

In tal modo, quella variabile potrà ancora essere acceduta dalle funzioni nel suo modulo, ma da nessun altro modulo.

Esempio, Problemi con variabili globali, all'interno dello stesso file in cui le variabili globali sono definite

```
#include <stdio.h>

int K=2;                                /* variabile globale visibile      */
                                        /* da tutte le funzioni            */

int main(void)
{
        int         i=34;

        printf("i = %d \n",  i  );      /* stampa i cioè 34, corretto      */
        int         J=0;            /* ERRORE, dichiarazione dopo istr. */
        printf("K = %d \n", K  );      /* stampa K cioè 2, corretto       */
        printf("g = %f \n", g  );      /* NON "VEDE" g, ERRORE            */
        funzione1();
        exit(0);
}

double      g=13;

void funzione1(void)
{
        printf("g = %f \n", g  );      /* stampa g, cioè 13, corretto     */
        printf("i = %d \n",  i  );      /* NON VEDE i, ERRORE              */
}
```

Esempio, Problemi tipici in programmi con più moduli.

Il nostro programma è costituito da due moduli, var.c e main.c.

- main.c contiene il main del programma, ed alcune funzioni, tra cui la funzione f , che accetta come parametro formale un intero e lo stampa.
- var.c contiene alcune variabili intere, alcune (A)globali, altre (C) globali ma statiche e quindi visibili solo dentro il modulo var.c.
- Non esiste una variabile B da nessuna parte.

```
/* file var.c */
int A=1;
static int C;
```

```
#include <stdio.h>          /* file main.c */
extern int A;
extern int C;
void f(int c){    printf("c=%d\n",c);  } /*stampa intero */
void main(void)
{
   f(A);   /* corretto                                  */
   f(B);   /* error C2065: 'B' : undeclared identifier*/
   f(C);   /* error LNK2001:unresolved external symbol _C*/
}
```

Il modulo main.c contiene due errori, perchè:

1) con l'istruzione f(C) main tenta di accedere alla variabile C che non può vedere perchè è protetta dallo specificatore static che la rende visibile solo dentro var.c.

- Il compilatore non si accorge dell'errore perchè main.c ha una dichiarazione extern per C, e il compilatore si fida e fa finta che C esista e sia accedibile in un qualche altro modulo.
- Il linker invece, che deve far tornare i conti, non riesce a rintracciare una variabile C accedibile, e segnala l'errore indicando un "**error LNK2001: unresolved external symbol**" perche non trova C.

2) con l'istruzione f(B) main tenta di accedere alla variabile B che non è definita nel modulo main.c, nemmeno da una dichiarazione extern.

- il compilatore si accorge dell'errore e lo segnala con il messaggio "**error C2065: 'B' : undeclared identifier**".

Protezione dagli Accessi esterni alla funzione per variabili che debbono mantenere un valore tra una chiamata alla funzione e la successiva ad una funzione.

Lo specificatore static, appliccato ad una variable locale ordina al compilatore di collocare la variabile non più nello stack all'atto della chiamata alla funzione, ma in una locazione di memoria permanente (per tutta la durata del programma), come se fosse una varabile locale.
Ma a differenza della variabile globale, la variabile locale static sarà visibile solo all'interno del blocco in cui è stata dichiarata.
L'effetto è che la variabile locale static:

- viene inizializzata una sola volta, la prima volta che la funzione viene chiamata.
- mantiene il valore assunto anche dopo che il controllo è uscito dalla funzione, e fino a che non viene di nuovo chiamata la stessa funzione.

vediamo un esempio di utilizzo, per contare il numero delle volte che una data funzione viene eseguita.

```
#include <stdio.h>          /* file contaf.c */
void f(void)
{
      static int     contatore=0;      /* viene inizializzato solo una volta */
      contatore = contatore + 1;
      printf("contatore =%d\n", contatore);     /*stampa contatore */
}

void main()             /* per vedere cosa succede in f*/
{
      int i;
      for( i=0; i<100; i++ )
            f();
}
```

Istruzioni di Assegnamento, Semplice e Multiplo

Mentre in Pascal l'operatore di assegnamento era **:=** , in C è **=** e basta.
Quindi l'istruzione di assegnamento diventa:

NomeVariabile = espressione ;

dove:

- espressione può essere semplice come una singola costante, o essere una combinazione di variabili, operatori, funzioni e costanti.
- NomeVariabile deve essere una variabile, e mai una funzione.

esempi:

```
        dati
int func( float f); /* prototipo della funzione func */
int i , j;

i = 0;                          corretta
i = func( 3.8 );                corretta
j = i ;                         corretta
func( 3.8 ) =  100;             NON AMMESSA, errore in compilazione
```

L'assegnamento in C produce un risultato, che è **il valore assegnato alla variabile a sinistra dell'*uguale*, ed è del tipo della variabile**. Tale risultato può essere utilizzato:

- **per un ulteriore assegnamento**, tenendolo alla destra di un *uguale*, valgono perciò istruzioni composite del tipo:
  ```
  int i, j , k ;
   k = j = i = 10;
  ```
  che assegnano alla variabile a sinistra il valore assegnato alla variabile subito a destra, cioè si assegna prima il valore 10 ad i, poi dato che i vale 10 si assegna il valore 10 a j, poi visto che j vale 10 si assegna 10 a k.
  Il vantaggio è un'esecuzione più veloce delle istruzioni di assegnamento separate, perchè il dato da assegnare è già caricato sui registri.
- **come espressione** ad esempio come condizione in un if, eventualmente tenendolo dentro parentesi per evitare confusioni
  ```
  if ( i=10 ) func(9);      i=10 vale 10 che è diverso da zero e vale VERO
  ```

Problema: Conversioni di Tipo nelle Istruzioni di Assegnamento

```
void main(void)
{
int i,j;      double g,x;
x = i = g = 10.3;
printf("x=%f  i=%d  g=%f\n",x, i, g );
}
```

Tale programma stampa la stringa:

x=10.0 i=10 g=10.3

```
void main(void)
{
int i,j;      double g,x;
i = x = g = 10.3;
printf("x=%f  i=%d  g=%f\n",x, i, g);
}
```

Tale programma stampa la stringa:

x=10.3 i=10 g=10.3

Valutiamo cosa succede nell'assegnamento multiplo del programma a sinistra.
l'istruzione di assegnamento g=10.3; assegna un valore floating point 10.3 alla variabile floating point g, e 10.3 è il risultato dell'assegnamento più a destra.
Tale risultato (10.3) viene assegnato ad i che però è una variabile intera, e quindi viene persa la parte decimale e ad i viene assegnato il valore 10, che è il risultato dell'assegnamento ad i.
Questo risultato (10) viene quindi assegnato a x che lo memorizza come floating point 10.0 , che è il risultato dell'assegnamento ad x, ed è anche il risultato finale di tutto l'assegnamento multiplo.

Nell'assegnamento multiplo del programma a destra, invece, l'assegnamento ad x viene fatto dandogli il valore ottenuto dall'assegnamento di 10.3 a g che è floating point, e quindi memorizza 10.3.

N.B.
In entrambi i casi il compilatore si accorge che c'e' una perdita di dati nel passaggio da double a int e avverte il programmatore con un warning di questo tipo:
warning: '=' : conversion from 'double ' to 'int ', possible loss of data

---

Conversioni di Tipo nelle Istruzioni di Assegnamento

Il problema delle conversioni di tipo esiste quando l'assegnamento coinvolge variabili (o una variabile ed una costante) di tipo diverso.

La **regola di conversione** che vale in C è la seguente:

Il C converte il valore alla destra del segno di uguale (il risultato della valutazione dell'espressione) nel tipo del lato sinistro (la variabile destinazione dell'assegnamento.

La seguente tabella illustra la eventuale perdita di dati durante la conversione:

Tipo Destinazione	Tipo Espressione	Possibile Perdita di Informazione	Informazione Conservata
signed char	char	se valore è > 127	7 bit meno significativi
char	short int	byte + significativo	byte - significativo
char	long int (4 byte)	3 byte + significativi	byte - significativo
short int	long int	2 byte + significativi	byte - significativo
short int	float, double	almeno la parte frazionaria	la parte intera se minore del valore massimo per short int, un valore senza senso altrimenti
float	double	arrotondamento del risultato	risultato arrotondato alla precisione del float

vediamo un esempio, conversione di un short int in un char:

short int i; char ch; **ch = i ;**