




# Funzioni

Francesco Isgrò



# Funzioni

- Scomporre un problema in sotto-problemi semplici è molto spesso il miglior metodo per la risoluzione di problemi
- Prendere un problema e dividerlo in piccoli pezzi più facili è vitale per scrivere programmi *grandi*
- In C questo approccio è implementato usando il costrutto delle funzioni

- 
- Un programma è costituito da uno o più file
    - Ogni file contiene zero (header file) o più funzioni
    - Un solo file contiene la funzione main
  - La funzione main può chiamare altre funzioni
  - Le funzioni operano su variabili del programma


# Definizione delle funzioni

- Il codice che descrive cosa fa una funzione viene detto definizione della funzione

- La forma generale è

*type function\_name(parameters) { declarations  
statements }*

- La parte prima della parentesi graffa è denominata *header*
- La parte fra le parentesi graffe è detta *corpo* della funzione
- I parametri sono una lista di dichiarazioni di variabile separate da virgola




```
int factorial ( int n )           //header
{                                // body starts here
    int i , product = 1;

    for ( i =2; i <= n ; i ++ ) {
        product *= i ;
    }


    return product ;
}
```

- Il valore restituito, se necessario, sarà convertito in intero
- La lista dei parametri dice che il parametro, se necessario, sarà convertito in intero




```
void wrt_address(void)
{
    printf("%s \n%s \n%s \n%s \n%s \n\n",
        "          ****",
        "      **  SANTA CLAUS  **",
        "      **  NORTH POLE   **",
        "      **   EARTH        **");
}
```

- Il primo void informa il compilatore che la funzione non restituisce alcun valore
- Il secondo void informa il compilatore che la funzione non ha parametri




```
void wrt_address(void)
{
    printf("%s \n%s \n%s \n%s \n%s \n\n",
        "*****",
        "**  SANTA CLAUS  **",
        "**  NORTH POLE   **",
        "**  EARTH         **");
}
```

- La funzione viene chiamata con l'istruzione  
wrt\_address();
- Cosa succede se scriviamo l'istruzione  
a = wrt\_address();

- 
- La definizione inizia con il tipo della funzione
  - Se la funzione non restituisce alcun valore il tipo è void
  - Se il tipo è diverso da void il valore restituito, se necessario, viene convertito a quel tipo
  - I parametri sono anche chiamati *formal parameters*
  - Il corpo della funzione è un blocco di statement che può anche contenere dichiarazioni di variabili
  - Le variabili dichiarate nel corpo di una funzione sono visibili solo all'interno della funzione






```
void nothing ( void )  
{  
  
}
```


```
double twice ( double x )  
{  
    return (2.0 * x );  
}
```

```
int all_add ( int a , int b , int c )  
{  
    return ( a + b + c );  
}
```

- 
- Se il tipo della funzione non è definito viene considerato int
  - Ad esempio, le due forme sono equivalenti

```
all_add ( int a , int b , int c )  
{  
    return ( a + b + c );  
}
```


```
int all_add ( int a , int b , int c )  
{  
    return ( a + b + c );  
}
```

- 
- Se il tipo della funzione non è definito viene considerato `int`
  - Ad esempio, le due forme sono equivalenti

```
all_add ( int a , int b , int c )  
{  
    return ( a + b + c );  
}
```

```
int all_add ( int a , int b , int c )  
{  
    return ( a + b + c );  
}
```

- Comunque è considerata buona pratica di programmazione specificare sempre il tipo della funzione



```
#include <stdio.h>

int a = 33;

int all_add(int a, int b, int c)
{
    return (a+b+c);
}


int main(void)
{
    int b = 15, c = 34;
    printf("%d \n", all_add(a,b,c));
}
```

- Tutte le variabili dichiarate nel corpo di una funzione sono dette *locali*
- Le variabili dichiarate esternamente al corpo di una funzione sono dette *globali*

# Definizione nel C tradizionale

- Nel C tradizionale la definizione delle funzioni ha una sintassi differente
- La dichiarazione di una variabile nella lista dei parametri avviene dopo la lista e prima della prima parentesi graffa

```
void f(a, b, c, x, y)
int  a, b, c;
double x, y;
{
    ....
}
```

- 
- L'ordine in cui vengono dichiarati i parametri non ha importanza
  - Se non ci sono parametri si mettono solo le parentesi

```
void f()  
{  
    ....  
}
```




# Perché scrivere programmi usando le funzioni?

- È più facile scrivere funzioni brevi che siano corrette
- Scrittura e debugging sono più semplici
- In generale maggiore semplicità nelle modifiche riscrivendo solo le funzioni che devono essere modificate
- Maggiore comprensibilità del codice: funzioni brevi si leggono e capiscono più facilmente

# L'istruzione return

- Lo statement return non deve necessariamente includere un'espressione
- Ad esempio
  - `return;`
  - `return ++a;`
  - `return (a*b);`
- Con l'istruzione return la funzione termina e il controllo passa all'ambiente in cui è stata chiamata



- 
- Se c'è una espressione il valore di questa è passato all'ambiente in cui è stata chiamata
  - Se necessario il valore sarà convertito al tipo della funzione
  - Ci possono essere zero o più return statement
  - Se non ci sono return il controllo è restituito quando si raggiunge la parentesi di chiusura della funzione



## Due return

```
double absolute_value(double x)
{
    if (x >= 0)
        return x;
    else
        return -x;
}
```

## Non sempre si usa il valore restituito

```
while (.....) {
    getchar();
    c = getchar();
    .....
}
```

# Prototipi delle funzioni


- Le funzioni devono essere dichiarate prima che possano essere usate
- ANSI C fornisce un meccanismo chiamato *function prototype*
- Il prototipo istruisce il compilatore sul numero e tipo di parametri che sono passati alla funzione e sul tipo del valore restituito dalla funzione


- 
- Il prototipo


`double sqrt(double);`

informa che `sqrt()`

- è una funzione
- prende un solo argomento di tipo `double`
- restituisce un valore di tipo `double`

- 
- La forma generale del prototipo è  
    < type > function\_name ( parameter type list );
  - Gli identificatori dei parametri sono opzionali.
  - I due prototipi  
    void f ( char c , int i );  
    void f ( char , int );  
sono equivalenti

- 
- Gli identificatori non sono usati dal compilatore
  - Servono a fornire documentazione sul codice, specialmente se i nomi scelti sono significativi
  - Suggerimento: metterli sempre

- 
- La parola chiave `void` è usata se la funzione non ha nessun parametro
  - Se la funzione accetta un numero variabile di parametri si usano .... (ellipses)


```
int printf ( const char *format , ...);
```


- 
- Nel C tradizionale la dichiarazione non permetteva la lista dei tipi dei parametri

`double sqrt ();      // traditional C style`

- L'uso dei prototipi è preferibile
- I valori passati alle funzioni sono forzati opportunamente



- 
- Consideriamo la chiamata
    - `sqrt(4);`
  - senza il prototipo il compilatore non sa che la funzione accetta un `double` e il risultato non sarà corretto
  - con il prototipo il risultato ottenuto è corretto



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

```
double sqrt();
```

```
int main()
```

```
{
```

```
    int a = 4;
```


```
    printf("sqrt(4) = %f \n", sqrt(a));  
}
```

## Compilazione

```
gcc -o prova prova.c -lm
```

## Output

```
sqrt(4) = 0.0000
```



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

```
double sqrt(double);
```

```
int main()
```

```
{
```

```
    int a = 4;
```

```
    printf("sqrt(4) = %f \n", sqrt(a));  
}
```

## Compilazione

```
gcc -o prova prova.c -lm
```


## Output


```
sqrt(4) = 0.0000
```




# Dichiarazione delle funzioni vista dal compilatore

- Per il compilatore la dichiarazione di una funzione è generata in varie maniere
  - chiamata della funzione
  - definizione della funzione
  - dichiarazione esplicita
  - prototipo della funzione

- 
- Se viene incontrata una chiamata  $f(x)$  prima di un altro tipo di dichiarazione il compilatore assume una dichiarazione
    - `int f();`
  - Non si assume niente sulla lista dei parametri
  - Il programmatore deve passare i tipi corretti

- 
- Vediamo ora come si comporta il compilatore con la definizione di una funzione
  - Consideriamo prima la definizione con C tradizionale per poi passare a ANSI C




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

```
float f(x)
long double x;
{
    return sqrt(x);
}
```

```
int main()
{
    int a = 4;
    double b = 4;
    long double c = 4;
```

```
    printf("parametro intero: %f \n", f(a));
    printf("parametro double: %f \n", f(b));
    printf("parametro long double: %f \n", f(c));
}
```



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

```
float f(x)
long double x;
{
    return sqrt(x);
}
```

```
int main()
{
    int a = 4;
    double b = 4;
    long double c = 4;

    printf("parametro intero: %f \n", f(a));
    printf("parametro double: %f \n", f(b));
    printf("parametro long double: %f \n", f(c));
}
```

## Output

```
parametro intero: 0.000000
parametro double: 0.000000
parametro long double: 2.000000
```




- 
- Il codice include la definizione della funzione f
  - Solo la linea

`float f(x)`

è presa come dichiarazione della funzione

- Non si assume niente sui parametri
  - devono essere passati correttamente
  - il compilatore non può fare nessuna conversione




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

```
float f(x)
{
    return sqrt(x);
}
```

```
int main()
{
    int a = 4;
    double b = 4;
    long double c = 4;

    printf("parametro intero: %f \n", f(a));
    printf("parametro double: %f \n", f(b));
    printf("parametro long double: %f \n", f(c));
}
```



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


```
float f(x)
{
    return sqrt(x);
}
```


```
int main()
{
    int a = 4;
    double b = 4;
    long double c = 4;

    printf("parametro intero: %f \n", f(a));
    printf("parametro double: %f \n", f(b));
    printf("parametro long double: %f \n", f(c));
}
```

## Output

```
parametro intero: 2.000000
parametro double: 0.000000
```

- 
- Il tipo del parametro `x` non è dichiarato
  - Il compilatore assume la dichiarazione  
`int x;`




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

```
float f(long double x)
{
    return sqrt(x);
}
```

```
int main()
{
    int a = 4;
    double b = 4;
    long double c = 4;

    printf("parametro intero: %f \n", f(a));
    printf("parametro double: %f \n", f(b));
    printf("parametro long double: %f \n", f(c));
}
```



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

```
float f(long double x)
{
    return sqrt(x);
}
```

```
int main()
{
    int a = 4;
    double b = 4;
    long double c = 4;

    printf("parametro intero: %f \n", f(a));
    printf("parametro double: %f \n", f(b));
    printf("parametro long double: %f \n", f(c));
}
```

## Output

```
parametro intero: 2.000000
parametro double: 2.000000
parametro long double: 2.000000
```

- 
- La linea

`float f(long double x)`

è presa come dichiarazione.


- Questa volta è un prototipo.
- Il compilatore conosce la lista dei parametri quindi può convertire int in long double
- In definitiva è buona norma la definizione della funzione (stile ANSI C ) o il prototipo prima che la funzione sia usata
- Una delle ragioni di includere gli header file è che contengono i prototipi delle funzioni



# Passaggio per valore

- Gli argomenti delle funzioni sono passati per valore
- Questo significa che ogni argomento è calcolato e il suo valore è usato localmente al posto del corrispondente parametro
- Se una variabile è passata a una funzione, il suo valore nell'ambiente che chiama la funzione non è modificato, anche se la funzione modifica il valore che riceve





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


```
int compute_sum(int n)    /* sum the integers from 1 to n */
{
    int sum = 0;

    for ( ; n > 0; --n)    /* stored value of n is changed */
        sum += n;
    return sum;
}
```

```
int main(void)
{
    int n = 3, sum, compute_sum(int);

    printf("%d\n", n);
    sum = compute_sum(n);
    printf("%d\n", n);
    printf("%d\n", sum);

    return 0;
}
```



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

```
int compute_sum(int n)    /* sum the integers from 1 to n */
{
    int sum = 0;

    for ( ; n > 0; --n)    /* stored value of n is changed */
        sum += n;
    return sum;
}
```


```
int main(void)
{
    int n = 3, sum, compute_sum(int);

    printf("%d\n", n);
    sum = compute_sum(n);
    printf("%d\n", n);
    printf("%d\n", sum);

    return 0;
}
```

**Output**

3  
3  
6

- 
- La variabile `n` è passata alla funzione `compute_sum()`
  - Il valore di `n` è modificato nel corpo della funzione `compute_sum()`
  - Il valore di `n` nell'ambiente di partenza non è modificato
  - Se si vuole che `n` sia modificata nell'ambiente di partenza si può
    - restituire il valore come risultato della funzione
    - passare l'indirizzo della variabile che si vuole modificare (puntatori)




# Storage class

- Ogni variabile e funzione in C ha due attributi
  - tipo
  - storage class
- Ci sono 4 storage class corrispondenti alle parole chiave
  - auto
  - extern
  - register
  - static



# Storage class auto

- Un blocco è una parte di codice delimitata da parentesi graffe che inizia con dichiarazioni di variabili
- Le variabili hanno visibilità solo all'interno del blocco
- Le variabili dichiarate all'interno di un blocco sono auto di default
- Poiché è la classe più comune la parola chiave auto è usata raramente

- 
- Quando si entra in un blocco il sistema alloca la memoria per le variabili locali del blocco
  - Quando si esce dal blocco il sistema libera la memoria che era stata occupata

# Storage class extern

- Una maniera semplice di passare informazioni fra blocchi e funzioni è l'uso di variabili esterne
- Una variabile è esterna se non è definita all'interno di un blocco
- Alle variabili esterne viene automaticamente assegnata la classe extern, senza esplicitare la storage class
- La parola chiave extern viene usata per dire al compilatore di guardare da un'altra parte per quella variabile
- Chiariamo con un esempio di un programma su due file



file1.c

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

```
int a =1, b =2, c = 3; //variabili esterne  
int f(void);
```


```
int main(void)  
{  
    printf("`%3d\n", f());  
    printf("`%3d %3d %3d\n", a, b, c);  
  
    return 0;  
}
```


file2.c


```
extern int a; //cercala da qualche altra parte
```

```
int f(void)  
{  
    int b, c;    //variabili locali  
  
    a = b = c = 4;  
    return (a +b +c);  
}
```




- 
- I due file sono compilati separatamente
  - Durante la compilazione del file1 alle variabili a , b e c viene assegnata la classe extern
  - Durante la compilazione del file2 il compilatore viene istruito che la variabile a è globale e dichiarata in un altro file
  - La fase di linking mette insieme tutte le informazioni e fa sì che si usi la locazione di memoria giusta

- 
- Le informazioni si possono passare anche come parametri delle funzioni
  - Va precisato che è preferibile cercare di evitare le variabili globali
  - Meglio trasferire informazioni tramite i parametri delle funzioni
  - Codice più chiaro e si capisce meglio dove i valori sono modificati


- 
- Tutte le funzioni hanno storage class extern
  - Questo significa che le funzioni sono visibili ovunque in tutto il programma, anche se scritto su più file
  - Si può quindi aggiungere la parola chiave extern alla definizione e ai prototipi delle funzioni  
extern double sin ( double );  
ma in generale non è necessario

# Storage class register

- La storage class register chiede al compilatore di associare la variabile, se possibile, a un registro di memoria ad accesso rapido
- Siccome si tratta di memoria limitata, che è anche usata dal sistema, non sempre è possibile soddisfare le richieste
- In caso negativo la storage class diventa auto
- Le variabili globali non possono essere register
- Le funzioni non possono essere register

- 
- Register è un tentativo di velocizzare l'esecuzione del programma
  - Le variabili register vanno selezionate accuratamente
  - Poche variabili a cui si accede molto spesso
  - Candidati comune sono le variabili su cui si fa un ciclo

```
int main(void)
{
    register int i;
    .....
    for (i = 0; i < LIMIT; i++) {
    }
}
```

- 
- In questa maniera la variabile blocca il registro per tutta l'esecuzione della funzione
  - Più efficiente spostare la dichiarazione e il loop in un blocco


```
int main(void)
{
    .....
    {
        register int i;

        for (i = 0; i < LIMIT; i++) {
        }
    }
}
```



# Storage class static

- Le dichiarazioni static hanno due usi distinti
  - Il primo serve a mantenere il valore di una variabile anche dopo l'uscita dal blocco in cui è visibile
  - Il secondo è in connessione con la parola chiave `extern`




```
void f(void)
{
    static int cnt = 0;

    ++cnt;
    if (cnt % 2 == 0)
        .....
    else
        .....
}
```

- La prima volta cnt è inizializzata a 0 e poi incrementata a 1
- La seconda volta che f viene chiamata cnt non viene inizializzata e mantiene il valore 1 e esce con il valore 2.



- 
- Variabili static extern hanno una visibilità ristretta rispetto alle variabili globali
  - Variabili di questa classe non sono visibili a funzioni definite prima nello stesso file o a funzioni in altri file
  - Il tentativo di accedervi usando il meccanismo della parola chiave extern non serve




//A family of pseudo random number generators


```
#define INITIAL_SEED    17
#define MULTIPLIER      25173
#define INCREMENT       13849
#define MODULUS         65536
#define FLOATING_MODULUS 65536.0
```

```
static unsigned seed = INITIAL_SEED; //esterna ma privata a
                                     //questo file
```

```
unsigned random(void)
{
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
    return seed;
}
```

```
double probability(void)
{
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
    return (seed / FLOATING_MODULUS);
}
```

- 
- `random()` produce un numero intero fra 0 e MODULUS
  - `probability()` produce un numero fra 0 e 1
  - Le due funzioni usano il vecchio valore di seed e lo modificano
  - Se seed fosse modificata accidentalmente verrebbe modificato l'algoritmo
  - Siccome seed è privata di questo file non può essere modificata da nessun altro modulo

- 
- L'uso della classe `static` per le funzioni restringe la visibilità delle funzioni
  - Funzioni `static` sono visibili solo nel file in cui sono definite
  - Quindi sono accessibili solo a funzioni dello stesso file
  - Utili per definire dei moduli privati di funzioni