

Funzioni



- Programmi che possono essere mandati in esecuzione da altri programmi
- Evitano ripetizioni di codice e facilitano la programmazione rendendola modulare
 - Rendono possibile riutilizzare infinite volte programmi già fatti all'interno di programmi nuovi
- Ogni funzione ha un *nome* che serve al programma chiamante per mandarla in esecuzione
- Funzioni definite all'interno di un programma possono essere mandate in esecuzione anche da altri programmi
- Raggruppando funzioni (collaudate) in librerie tematiche, altri programmi potranno utilizzarle facilmente comprimendo i tempi di sviluppo del software e rendendolo più affidabile

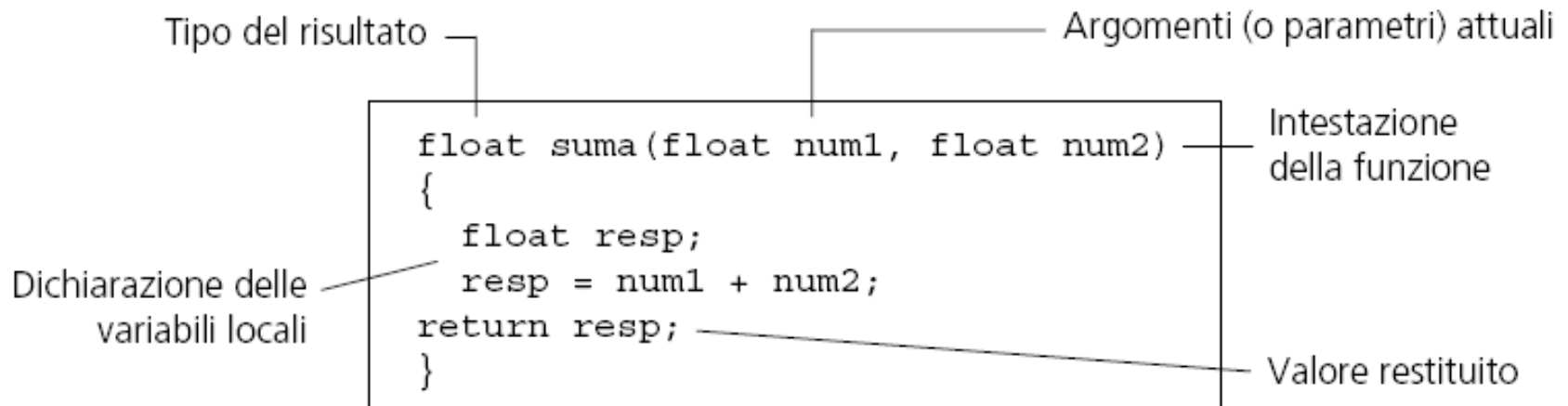
il concetto di funzione

un programma per leggere una lista di caratteri dalla tastiera, metterli in ordine alfabetico e visualizzarli sullo schermo, si può scrivere come una funzione (la `main()`) che chiama altre funzioni indipendenti per realizzare ogni sottocompito:

```
int main()
{
    legge_caratteri(); // Chiama la funzione che legge i caratteri
    ordinare();        // Chiama la funzione che li ordina alfabeticamente
    scrivi_caratteri(); // Chiama la funzione che li scrive sullo schermo
    return 0;          // restituisce il controllo al sistema operativo
}
int legge_caratteri()
{
    ...                // Codice per leggere una sequenza di caratteri dalla tastiera
    return 0;          // restituisce il controllo al main()
}
int ordinare()
{
    ...                // Codice per ordinare alfabeticamente la sequenza dei caratteri
    return 0;          // restituisce il controllo al main()
}
int scrivi_caratteri()
{
    ...                // Codice per visualizzare sullo schermo la sequenza ordinata
    return 0;          // restituisce il controllo al main()
}
```

struttura di una funzione

- una funzione è un programma che può essere mandato in esecuzione in qualunque punto di un altro programma
- in C++ le funzioni non si possono annidare, cioè non possono essere dichiarate dentro altre funzioni, ma sono globali, cioè, possono essere chiamate da qualunque punto del programma



caratteristiche di una funzione



- ***tipo del risultato***: tipo del dato che la funzione restituisce al programma che l'ha mandata in esecuzione
- ***argomenti formali***: lista dei parametri tipizzati che la funzione richiede al programma che la chiama; vengono scritti nel formato:
tipo1 parametro1, tipo2 parametro2, ...
- ***corpo della funzione***: è il sottoprogramma vero e proprio; si racchiude tra parentesi graffe senza punto e virgola dopo quella di chiusura
- ***passaggio di parametri***: quando viene mandata in esecuzione una funzione le si passano i suoi argomenti "attuali" e, come vedremo, questo passaggio può avvenire o "*per valore*" o "*per riferimento*"
- ***dichiarazioni locali***: gli argomenti formali, le costanti e le variabili definite dentro la funzione sono ad essa locali, cioè esistono solo mentre la funzione è in esecuzione e non sono accessibili fuori di essa
- ***valore restituito dalla funzione***: mediante la parola riservata `return` si può ritornare il valore restituito dalla funzione al programma chiamante
- non si possono dichiarare funzioni annidate, ma *una funzione può mandare in esecuzione un'altra funzione*

nome di una funzione

- comincia con una lettera o un underscore (_) e può contenere lettere, cifre o underscores
- C++ è "case sensitive", il che significa che le lettere maiuscole e minuscole sono caratteri diversi

```
// il nome di questa funzione è "max"  
int max (int x, int e)  
{  
    ...  
}
```

```
// il nome di questa funzione è "media"  
double media (double x1, double x2)  
{  
    ...  
}
```

tipo del dato di ritorno

- se non si specifica il tipo di dato restituito dalla funzione si sottintende che essa restituisce un valore di tipo `int`
- il tipo può essere uno dei tipi semplici, come `int`, `char` o `float`, un puntatore a qualunque tipo C++, o un tipo `struct`

```
int max(int x, int y) // ritorna un tipo int
double media(double x1, double x2) // ritorna un tipo double
float funz0() {...} //ritorna un float
char* funz1() {...} //ritorna un puntatore a char
int* funz3() {...} //ritorna un puntatore ad int
struct InfoPersona CercareRegistro(int num_registro);
```

- molte funzioni non restituiscono risultati e si utilizzano solo come *subroutines* per realizzare compiti concreti; esse vengono dette *procedure* e si specificano indicando la parola riservata `void` come tipo di dato restituito

```
void scrivi_risultati(float totale, int num_elementi)
```

risultati di una funzione

- una funzione *può* restituire un valore mediante l'istruzione `return` la cui sintassi è:

```
return (espressione) ;
```

```
return espressione;
```

```
return; // caso di una procedura, si può omettere
```

- `espressione` deve essere ovviamente del tipo definito come restituito dalla funzione; ad esempio, non si può restituire un valore `int` se il tipo di ritorno è un puntatore; tuttavia, se si restituisce un `int` e il tipo di ritorno è un `float`, il compilatore lo converte automaticamente
- una funzione può avere più di un'istruzione `return` e terminerà s'esegue la prima di esse
- se non s'incontra alcun'istruzione `return` l'esecuzione continuerà alla parentesi graffa finale del corpo della funzione
- un errore tipico è quello di dimenticare l'istruzione `return` o metterla dentro una sezione di codice che non verrà eseguita; in questi casi il risultato della funzione è imprevedibile e probabilmente porterà a risultati scorretti

Tipicamente C++
segnala l'errore

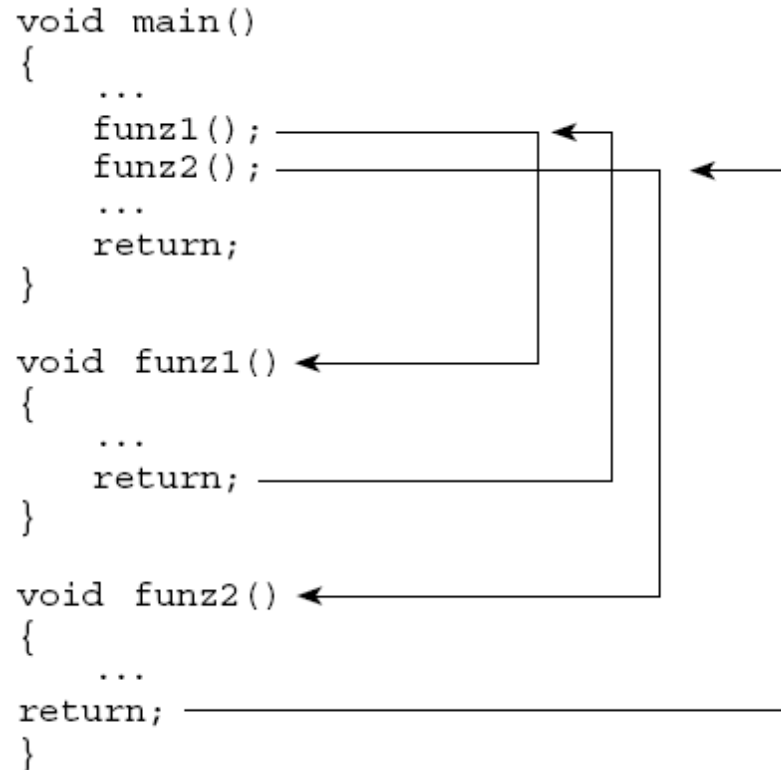
chiamata di una funzione

- una funzione va in esecuzione quando viene *chiamata* (o *invocata*) dal programma principale `main()` o da un'altra funzione
- la funzione che chiama un'altra funzione si denomina *funzione chiamante* e la funzione controllata si denomina *funzione chiamata*.

```
void main()
{
    ...
    funz1();
    funz2();
    ...
    return;
}

void funz1()
{
    ...
    return;
}

void funz2()
{
    ...
    return;
}
```



prototipi di funzioni

- a volte la funzione potrebbe essere definita in altri programmi che vengono poi collegati a quello che la chiama
- perché il compilatore possa accettare la chiamata ad una funzione definita altrove bisogna che essa sia *dichiarata*
 - Tipicamente prima di main
- la dichiarazione di una funzione si dice *prototipo* della funzione. Un prototipo
 - non ha il corpo (verrà definito altrove)
 - deve specificare il tipo dei parametri formali (ma non necessariamente il nome)
 - deve terminare con il punto e virgola ;

```
tipo_restituito nome_funzione (tipi_parametri_formali);
```

```
double DollaroEuro (double dollaro);  
int max(int,int);
```

Dichiarare vs Definire

- **Dichiarazione:** si fornisce il nome e si elencano le caratteristiche della funzione.
- **Definizione:** si riserva anche la memoria per essa.
 - La definizione specifica anche dove trovare la funzione

Suggerimento: Usate i nomi degli argomenti

```
int elabora (int classe, char sezione, char*cognome)
```

è equivalente a

```
int elab (int, char, char*)
```

passaggio di argomenti ad una funzione

- in C++ ci sono tre modi per passare variabili come argomenti attuali alle funzioni:

- **"per valore"**: è il sistema per default: non viene passata alla funzione la variabile, ma solo il valore in essa contenuto
- **"per riferimento tramite i puntatori"**: non viene passata alla funzione la variabile, ma il suo indirizzo di memoria contenuto in un puntatore; la funzione poi risalirà dall'indirizzo alla variabile per eventualmente modificarne il contenuto
- **"per riferimento tramite i riferimenti"**: (non presente nel vecchio linguaggio "C") con questa tecnica non viene passata alla funzione la variabile, ma un suo altro riferimento (cioè un *alias*); questo altro nominativo non è altro che l'indirizzo "mascherato" della variabile, che sarà poi utilizzato dalla funzione per risalire alla variabile di cui modificare eventualmente il contenuto

passaggio di argomenti "*per valore*"

- per default non viene passata alla funzione la variabile, ma solo il valore in essa contenuto; per esempio, si consideri la seguente funzione:

```
void scambia_valori_variabili(int a, int b)
{
    int aux; // definizione della variabile locale ausiliaria
    aux = a; // aux prende il valore del parametro a
    a = b;    // a prende il valore del parametro b
    b = aux; // b prende il valore della variabile locale aux
}
```

la chiamata:

```
int x=4, y=5;
scambia_valori(x, y);
```

non scambia i valori delle variabili x ed y che sono servite solo per passare ad a e b i loro rispettivi valori

passaggio *per riferimento* tramite *puntatori*

- i *puntatori* saranno trattati estesamente in seguito; ne anticipiamo qui l'uso come strumento per passare variabili alle funzioni

```
void scambia_valori_variabili(int* a, int* b)
{
    int aux; // definizione della variabile locale ausiliaria
    aux = *a; // aux prende il valore della variabile che si trova
              // all'indirizzo contenuto nel parametro a
    *a = *b; // la variabile che si trova all'indirizzo contenuto nel
              // parametro a prende come valore quello della variabile
              // all'indirizzo contenuto nel parametro b
    *b = aux; // la variabile che si trova all'indirizzo contenuto nel
              // parametro b prende come valore quello della variabile
              // ausiliare locale aux
}
```

la chiamata:

```
int x=4, y=5;
scambia_valori(&x, &y);
```

passa alla funzione non già le variabili *x* ed *y* bensì i loro indirizzi, specificati dall'operatore **&** denominato "*indirizzo di*"; ciò fa sì che la funzione sopra descritta acceda poi realmente alle due variabili *x* e *y* per scambiarne i valori

passaggio *per riferimento* tramite i *riferimenti*

- i *riferimenti* saranno trattati anche in seguito; ne anticipiamo qui l'uso principale come strumento per passare variabili alle funzioni

```
void scambia_valori_variabili(int& a, int& b)
{
    int aux;    // definizione della variabile locale ausiliaria
    aux = a;    // aux prende il valore della variabile che si chiama "a"
    a = b;      // la variabile che si chiama "a" prende come valore quello
                // della variabile che si chiama "b"
    b = aux;    // la variabile che si chiama "b" prende come valore quello
                // della variabile ausiliare locale aux
}
```

la chiamata:

```
int x=4, y=5;
scambia_valori(x, y);
```

passa alla funzione proprio le variabili *x* ed *y*; i parametri formali *a* e *b*, definiti come "*riferimenti ad intero*" mediante l'operatore *&*; essi sono cioè due potenziali nomi alternativi per variabili di tipo intero; alla chiamata diventano alias per le variabili *x* e *y* rispettivamente

argomenti di default

- in C++ è possibile definire funzioni in cui alcuni argomenti assumono un valore di *default*. Se all'atto della chiamata non viene passato alcun valore per quel parametro allora la funzione assumerà per lui il valore di default stabilito nell'intestazione
- gli argomenti di default devono raggrupparsi a destra nell'intestazione
- il valore di default deve essere un'espressione costante

```
char funzdef(int arg1=1, char c='A', float f_val=45.7f);
```

si può chiamare `funzdef` con qualunque delle seguenti istruzioni:

```
funzdef(9, 'Z', 91.5); //Annulla i tre argomenti di default
funzdef(25, 'W'); //Annulla i due primi argomenti di default
funzdef(50); //Annulla il primo argomento di default
funzdef(); //Utilizza i tre argomenti di default
```

se si omette un argomento bisogna omettere anche tutti quelli alla sua destra; la seguente chiamata non è corretta:

```
funzdef( , 'Z', 99.99);
```

funzioni *inline*

- servono per aumentare la velocità del programma
- convenienti quando la funzione si richiama parecchie volte nel programma e il suo codice è breve
- il compilatore ricopia realmente il codice della funzione in ogni punto in cui essa viene invocata
- il programma verrà così eseguito più velocemente perché non si dovrà eseguire il codice associato alla chiamata alla funzione
- tuttavia, ogni ripetizione della funzione richiede memoria, perciò il programma aumenta la sua dimensione
- per creare una funzione in linea si deve inserire la parola riservata `inline` all'inizio dell'intestazione

```
inline int sommare15(int n) {return (n+15);}
```

```
funzdef( , 'Z', 99.99);
```


visibilità

- la *visibilità* di una variabile è la zona del programma in cui essa è accessibile
- esistono quattro tipi di visibilità:
 - le variabili che hanno *visibilità di programma* si dicono *globali* e possono essere referenziate da qualunque funzione del programma; sono definite all'inizio del programma, fuori di qualunque funzione e sono visibili in tutto il programma a partire dal loro punto di definizione nel file sorgente
 - le variabili che hanno *visibilità di file sorgente* sono definite, fuori di qualunque funzione, mediante la parola riservata `static`; sono visibili dal punto in cui sono dichiarate fino alla fine del file sorgente
 - le variabili dichiarate dentro il corpo della funzione si dicono *locali* alla funzione ed hanno *visibilità di funzione*: si possono referenziare in qualunque parte della funzione ma non al di fuori di essa
 - una variabile dichiarata in un blocco ha *visibilità di blocco* e può essere referenziata dal punto in cui è dichiarata fino alla fine del blocco; una variabile locale dichiarata in un blocco annidato è visibile solo all'interno di quel blocco

classi di immagazzinamento

- gli specificatori `extern`, `register`, `static` e `typedef` modificano la visibilità di una variabile
- una funzione può utilizzare una variabile globale definita in un altro file sorgente *dichiarandola* localmente usando `extern`;
 - `extern` indica al compilatore che la variabile è *definita* in un altro file sorgente che sarà *linkato* assieme

```
#include<iostream>
using namespace std;
void leggiIntero(void);
```

```
int intero;
```

```
int main()
{ leggiIntero();
  cout << "Valore di Inter0:" << intero;
  return 0;
}
```

```
#include<iostream>
using namespace std;
```

```
void leggiIntero(void);
{ extern int intero;
  cout << "Introduci un intero:" << intero;
  cin >> intero;}
}
```

- per default le funzioni sono `extern` e quindi visibili da altri moduli di programma, ma si possono dichiarare `static`, così da impedire di utilizzarle in altri moduli sorgente del programma

classi di immagazzinamento

- gli specificatori `extern`, `register`, `static` e `typedef` modificano la visibilità di una variabile
 - *variabili registro*: con la parola riservata `register` si chiede al compilatore di porre la variabile in uno dei registri hardware del microprocessore (riducendo il tempo che la CPU richiede per cercare il valore della variabile in memoria);
 - richiesta non ordine!
 - non possono essere variabili globali
 - utilizzo tipico: variabili di controllo di cicli
 - *variabili statiche*: con la parola riservata `static` si chiede al compilatore di mantenere i valori delle variabili locali fra diverse chiamate di una funzione.
 - Una variabile statica s'inizializza una volta per tutte.

```
void SommaTotale(int add);  
{    static int Somma;  
    ...  
    Somma = Somma + add;  
}
```

funzioni di libreria

- tutte le versioni del linguaggio C++ contengono una grande raccolta di funzioni di libreria per operazioni comuni; esse sono raccolte in gruppi definite in uno stesso *header file*
- alcuni dei gruppi di funzioni di libreria più usuali sono:
 - I/O standard (per operazioni di Input/Output);
 - matematiche (per operazioni matematiche);
 - routines standard (per operazioni standard di programmi);
 - visualizzare finestra di testo;
 - di conversione (routines di conversione di caratteri e stringhe);
 - di diagnostico (forniscono routines di debugging incorporato);
 - di manipolazione di memoria;
 - controllo del processo;
 - classificazione (ordinamento);
 - cartelle;
 - data e ora;
 - di interfaccia;
 - ricerca;
 - manipolazione di stringhe;
 - grafici.

funzioni numeriche

- **matematiche:**

`ceil(x)` arrotonda all'intero più vicino

`fabs(x)` restituisce il valore assoluto di `x` (un valore positivo).

`floor(x)` arrotonda per difetto all'intero più vicino

`pow(x, y)` calcola `x` elevato ad `y`

`sqrt(x)` restituisce la radice quadrata di `x`

- **trigonometriche:**

`acos(x)` calcola l'arco coseno di `x`

`asin(x)` calcola l'arco seno di `x`

`atan(x)` calcola l'arco tangente di `x`

`atan2(x, e)` calcola l'arco tangente di `x` diviso `e`

`cos(x)` calcola il coseno dell'angolo `x` (`x` si esprime in radianti)

`sin(x)` calcola il seno dell'angolo `x` (`x` si esprime in radianti)

`tan(x)` restituisce la tangente dell'angolo `x` (`x` si esprime in radianti)

- **logaritmiche ed esponenziali:**

`exp(x)` calcola l'esponenziale e^x

`log(x)` calcola il logaritmo naturale di `x`

`log10(x)` calcola il logaritmo decimale di `x`

funzioni varie

- aleatorie:

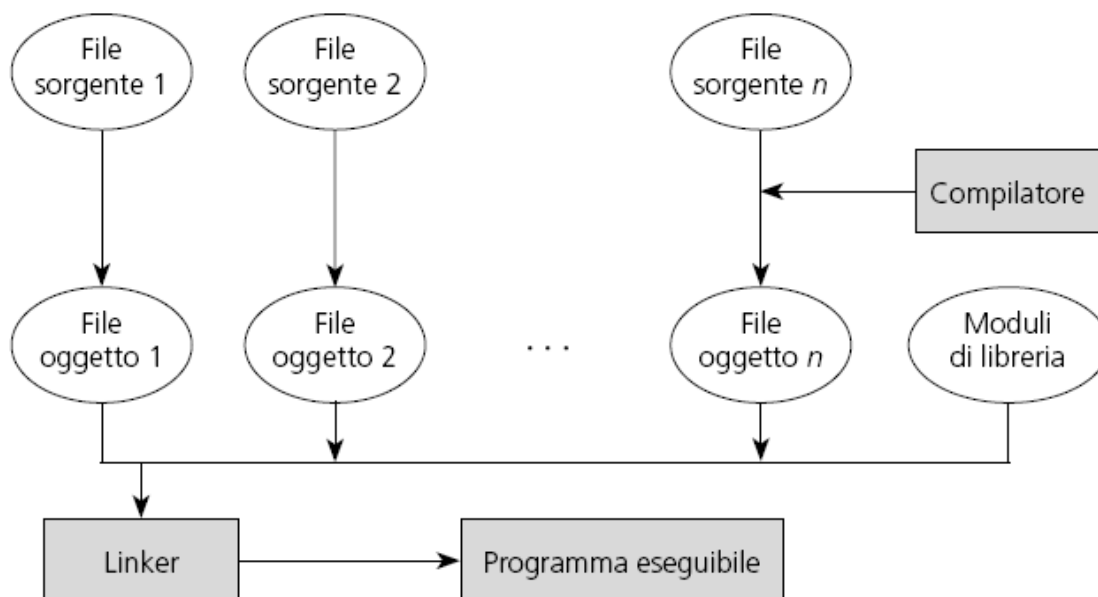
`rand()` genera un numero aleatorio fra 0 e `RAND_MAX`
`randomize()` inizializza il generatore di numeri aleatori con un seme aleatorio ottenuto a partire da una chiamata alla funzione `time`
`srand(seme)` inizializza il generatore di numeri aleatori in base al valore dell'argomento `seme`
`random(num)` restituisce un numero aleatorio da 0 a `num-1`

- di data ed ora:

`clock(void)` restituisce il tempo di CPU in secondi trascorso dall'inizio dell'esecuzione del programma
`time(ora)` ottiene l'ora attuale; restituisce il numero di secondi trascorsi dalla mezzanotte (00:00:00) del primo gennaio 1970; questo valore di tempo si mette nella posizione puntata dall'argomento `ora`

compilazione modulare

- i programmi grandi sono più facili da gestire se si dividono in vari files sorgenti, anche chiamati *moduli*, ognuno dei quali può contenere una o più funzioni; questi moduli verranno poi compilati separatamente ma linkati assieme
- ad ogni ricompilazione verranno in realtà ricompilati solo i moduli che sono stati modificati per ridurre il tempo di compilazione



sovraccaricamento delle funzioni



- *l'overheading* permette di dare lo stesso nome a funzioni con almeno un argomento di tipo diverso e/o con un diverso numero di argomenti
- C++ determina quale tra le funzioni sovraccaricate deve chiamare, in funzione del numero e del tipo di parametri passati

Le regole che il C++ segue per selezionare una funzione sovraccaricata sono

- se esiste, si seleziona la funzione che mostra la corrispondenza esatta tra il numero ed i tipi dei parametri formali ed attuali
- se tale funzione non esiste, si seleziona una funzione in cui il matching dei parametri formali ed attuali avviene tramite una conversione automatica di tipo (come da int a long, o da float a double)
- la corrispondenza dei tipi degli argomenti può venire anche forzata mediante *casting*
- se una funzione sovraccaricata possiede un numero variabile di parametri (tramite l'uso di punti sospensivi [...]), può venire selezionata in mancanza di corrispondenze più specifiche

Funzioni Ricorsive (intro)

- Funzioni che chiamano se stesse
- Necessitano di una condizione di arresto
- Se usate correttamente semplificano la scrittura del codice

Esempio: $\text{Fattoriale}(n) = n!$

```
double fattoriale(int n)
{
    if (n > 1) return (n*fattoriale(n-1));
    else return 1;
}
```

Versione non ricorsiva

Esempio: $\text{Fattoriale}(n) = n!$

```
double fattoriale(int n)
{   int val=1;

    for(int i=n; i>1; i--)
        val=val*i;
    return val;
}
```

template di funzioni

- forniscono un meccanismo per creare *funzioni* generiche, che possa cioè supportare simultaneamente differenti tipi di dato
- sono molto utili per i programmatori quando bisogna utilizzare la stessa funzione con differenti tipi di argomenti
- un template di funzione ha il seguente formato:

```
template <class tipo> tipo funzione (tipo arg1, tipo arg2,...)
{
    // Corpo della funzione
}
```

una dichiarazione tipica è:

```
template <class T> T max (T a, T b)
{
    return a > b ? a : b;
}
```

Macro vs Template

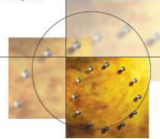
- In C è possibile scrivere codice indipendente dal tipo: usando macro
 - Si riscrive il codice “sostituendo” le variabili
- ```
#define max(a,b) ((a) < (b) ? (b) : (a))
```

## Contro:

- Il codice deve stare su una linea (logica)
- Non e' type safe (il compilatore non avverte di errori di tipo).
- Sono possibili errori inaspettati

## Pro:

- Non e' una funzione separata ma una espansione inline (fattibile anche con i templates dichiarandoli inline) => piu' efficiente in termini di tempo



# Differenza overheading/templates

! "\$%&' ( ) \* "+, - . ( / 0 1

2+" ' ) 34"#- ) "56\$+&- \* \* 7"% 2/ %& . ( ) 89) : \* \* " ; <  
 2+" ' ) 34\$=/ . ) "56\$+&- \* \* 7"% \$/ . / - - ( . ( ) 89) : \* \* " ; <  
 - ( 0>% - ( \* \$% , , ) ? 1) 2+" ' ) @ 4 ? ) "56\$+&- \* \* 7 2/ % 9) : \* \* " ; <

" #- ) 0/ " #456  
 ) ) " #- ) / ABC;  
 ) ) \$=/ . ) DAEFG;

) ) 34/ 5; HH - / 0>/ 9) "% 2/ %& . ( ) 89) BC  
 ) ) 34D5; HH - / 0>/ 9) "% \$/ . / - - ( . ( ) 89) F

) ) @ 4/ 5; HH - / 0>/ 9) 2/ % 9) BC  
 ) ) @ 4D5; HH - / 0>/ 9) 2/ % 9) F

) ) . ( - & #) J;  
 <