

Sperimentazioni di Fisica I

mod. A – Lezione 11

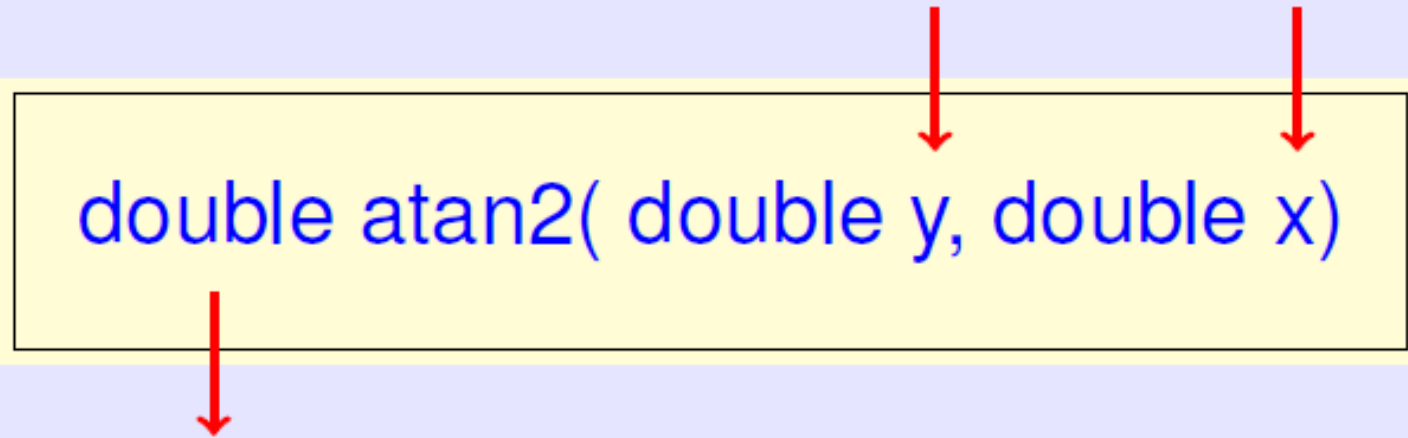
Le Funzioni (CAP. 6)

*Dipartimento di Fisica e Astronomia “G. Galilei”,
Università degli Studi di Padova*

Le Funzioni

- Sono i **mattoni fondamentali** per la **costruzione dei programmi** in C++ (e C)
- Il **prototipo** della funzione **descrive l'interfaccia** con l'utilizzatore.
- Es. funzione trigonometrica **arcotangente**, (**atan2**):

INPUT: 2 parametri



```
double atan2( double y, double x)
```

OUTPUT: risultato del calcolo

man page

- Tutte le [funzioni della libreria standard](#) del C e C++ hanno [pagine di documentazione](#) disponibile [online](#)
- Esempio: funzione trigonometrica [arcotangente](#), ([atan2](#))

```
man atan2
```

```
ATAN2(3)           Linux Programmer's Manual           ATAN2(3)
```

```
NAME
```

```
atan2 - arc tangent function of two variables
```

```
SYNOPSIS
```

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

```
double atan2(double y, double x);
```

```
DESCRIPTION
```

The `atan2()` function calculates the arc tangent of the two variables `x` and `y`. It is similar to calculating the arc tangent of `y/x`, except that the signs of both arguments are used to determine the quadrant of the result.

```
RETURN VALUE
```

The `atan2()` function returns the result in radians, which is between `-PI` and `PI` (inclusive).

Funzioni Definite dall'Utente

- Per utilizzare una funzione in C++ occorre seguire i seguenti passi:

1. fornire il **prototipo** della funzione
2. scrivere la **definizione** della funzione
3. invocare la funzione

```
1 double cube( double );  
int main( )  
{  
    using namespace std;  
  
    double R0 = 12.;  
    cout << "Cube of " << R0 << " is " << 3cube( R0 ) << endl;  
    return 0;  
}
```

```
2 double cube( double x ) //definizione  
{  
    return (x * x * x);  
}
```

Definizione di una Funzione

Esistono **due categorie** di funzioni

1. quelle che **NON ritornano** nessun **valore**

```
void functionName( parameterList )  
{  
    statement(s);  
    return; // opzionale  
}
```

2. quelle che **ritornano** un **valore**

```
typeName functionName( parameterList )  
{  
    statement(s);  
    return value; // value is cast to typeName  
}
```

Prototipo di una Funzione

Il prototipo di una funzione è un'istruzione → deve terminare con ' ; '

Il metodo più semplice per scrivere il prototipo → copiare la prima riga di definizione della funzione e aggiungere il carattere ' ; ':

```
double cube( double x ); //prototipo
double cube( double x ) // definizione funzione
{
    return ( x * x * x );
}
```

tuttavia è obbligatorio specificare soltanto il tipo dei parametri, il nome è opzionale (ignorato dal compilatore)

```
double cube( double ); //prototipo
double cube( double x ) // definizione funzione
{
    return ( x * x * x );
}
```

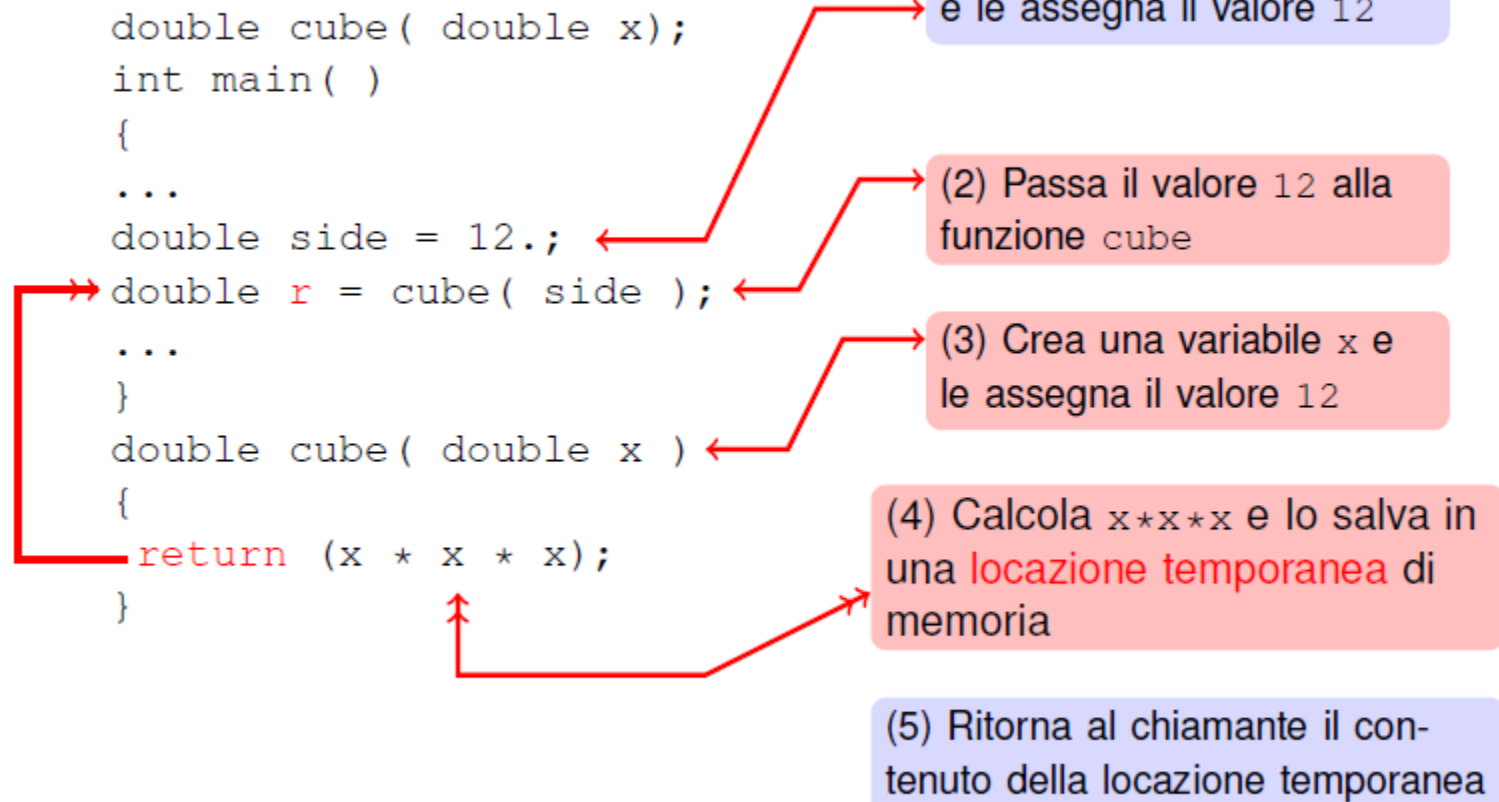
Utilizzo del Prototipo

- I **prototipi** delle funzioni aiutano il compilatore, **riducendo** le possibilità di errore.
- permettono al compilatore di
 - **trattare correttamente** il **valore di ritorno** della funzione
 - controllare che la **funzione** sia **invocata** con un **numero congruo di argomenti**
 - controllare che la **funzione** sia **invocata** con **tipi** di argomenti **corretti**
 - **se necessario** : tenta di convertire gli argomenti al tipo corretto (genera un errore in fase di compilazione se fallisce).

N.B. Il prototipo non è necessario se la funzione è definita prima della `main()`.

La Chiamata di una Funzione

- Studiamo la procedura con cui viene invocata una funzione



Funzioni e Variabili Locali

```
...  
void cheers( int n);  
...
```

```
int main( )  
{
```

```
    int n = 20;
```

```
    double a = 3.5;
```

```
    ...
```

```
    cheers( n );
```

```
    ...  
}
```

```
void cheers( int n )
```

```
{
```

```
    for (int i=0; i< n; i++)
```

```
        ...
```

```
}
```

n	a
20	3.5

Ogni funzione ha le proprie variabili con il loro nome e la loro *visibilità (scope)*

n	i
20	0

Argomenti Multipli

- Le funzioni possono avere più di un argomento
- Nella chiamata della funzione, gli argomenti vanno separati da una virgola:

```
x = integrazione( xmin, xmax, step );
```

- similmente, nella definizione, i parametri devono essere separati da virgole, (' , ')

```
double integrazione( double a, double b, double step )  
{  
    ...  
}
```

- anche se i parametri fossero dello stesso tipo, la definizione seguente non è permessa:

```
double integrazione( double a, b, step )  
{  
    ...  
}
```

Calcolo della Media (I)

- Supponiamo di avere un `vector` di `double` contenente alcune misure effettuate in laboratorio.
- Vogliamo scrivere una `funzione che ne calcola il valor medio`.
- Dobbiamo passare in input alla funzione il `vector`.
 - Possibile `prototipo`:
`double Media (vector <double>)`

- Definizione della `funzione`:

```
double Media (vector <double> dati)
{
    double somma = 0.0;
    for (int i=0; i < dati.size(); i++)
        somma += dati[i];
    return somma/dati.size();
}
```

Calcolo della Media (II)

- Possiamo eventualmente utilizzare il `range for`.
- Il `prototipo` sarà lo stesso di prima:
`double Media (vector <double>)`

- Definizione della `funzione`:

```
double Media (vector <double> dati)
{
    double somma = 0.0;
    for (auto c : dati)
        somma += c;
    return somma/dati.size();
}
```

Media di un Intervallo di Dati

- Supponiamo di dover calcolare la media solo di un intervallo dei dati salvati all'interno del nostro `vector`.
- Dovremo passare in input alla nostra funzione non solo il `vector` contenente le misure, ma anche l'indice **iniziale (a)** e l'indice **one-past-the-end (b)**.
- Possibile **prototipo**:
`double Media2 (vector <double>, int, int)`

- Definizione della **funzione**:

```
double Media2 (vector <double> dati, int a, int b)
{
    double somma = 0.0;
    for (int i = a; i < b; i++)
        somma += dati.at(i);
    return somma/(b-a);
}
```

Funzioni e Strutture

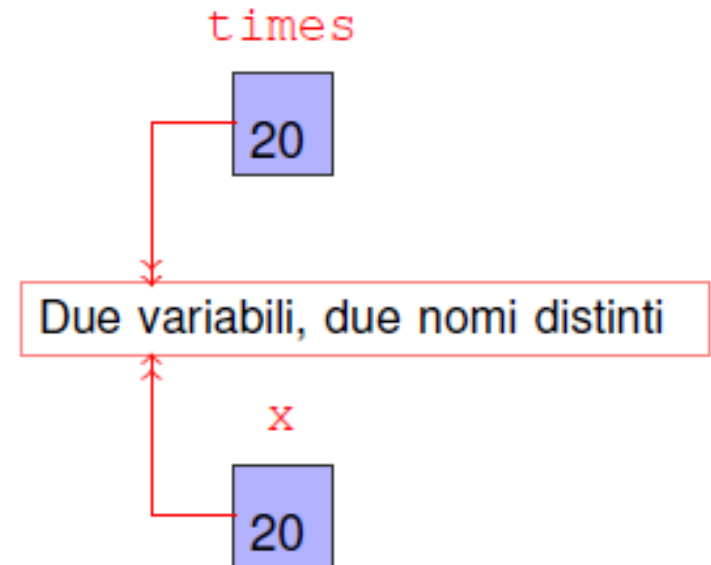
- Le **strutture** si comportano come i **tipi semplici** (variabili scalari).
 - Normalmente vengono passate alle funzioni **by-value**.
- le funzioni lavorano quindi con una copia della struttura originale.
- Se la struttura ha dimensione ragguardevoli, per ottimizzare il programma è consigliabile passare un puntatore alla struttura.
- Il C++ fornisce un'**alternativa** che è il **call by-reference**

Pass-by-value

```
void sneezy( int )  
int main( )  
{  
    int times = 20;  
    ...  
    sneezy( times );  
    ...  
}
```

```
void sneezy( int x )  
{  
    ...  
    ...  
}
```

Crea una variabile `times` e assegna il valore 20



Crea una variabile `x` e le passa il valore 20

Pass-by-reference

```
void sneezy( int & )  
int main( )  
{  
    int times = 20;  
    ...  
    sneezy( times );  
    ...  
}
```

Crea una variabile `times` e
assegna il valore 20

`times`

20

Una variabile,
due nomi

```
void sneezy( int & x )  
{  
    ...  
    ...  
}
```

Crea `x` come alias a `times`

Swap di due Variabili

Si vuole scambiare il contenuto di due variabili:

```
int a = 7;  
int b = 12;
```

a = 7 b = 12

```
cout << "a = " << a << " b = " << b << endl;
```

```
int temp;  
temp = a;  
a = b;  
b = temp;
```

a = 12 b = 7

```
cout << "a = " << a << " b = " << b << endl;
```

Swap: *pass-by-value*

Utilizziamo una funzione: `void swap_value(int, int)`

```
int v1 = 7;  
int v2 = -12;
```

`v1 = 7 v2 = -12`

```
cout << "v1 = " << v1 << " v2 = " << v2 << endl;
```

```
swap_value( v1, v2);
```

`v1 = 7 v2 = -12`

```
cout << "v1 = " << v1 << " v2 = " << v2 << endl;
```

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

Swap: *pass-by-reference*

Nuovo prototipo: `void swap_ref(int &, int &)`

```
int v1 = 7;
```

```
int v2 = -12;
```

`v1 = 7 v2 = -12`

```
cout << "v1 = " << v1 << " v2 = " << v2 << endl;
```

```
swap_ref( v1, v2);
```

`v1 = -12 v2 = 7`

```
cout << "v1 = " << v1 << " v2 = " << v2 << endl;
```

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

Swap: *pass-by-pointer*

Nuovo prototipo: `void swap_pointer(int *, int *)`

```
int v1 = 7;
```

```
int v2 = -12;
```

`v1 = 7 v2 = -12`

```
cout << "v1 = " << v1 << " v2 = " << v2 << endl;
```

```
swap_pointer( &v1, &v2);
```

`v1 = -12 v2 = 7`

```
cout << "v1 = " << v1 << " v2 = " << v2 << endl;
```

```
void swap_pointer( int * p, int * q)
{
    int temp = *p;
    *p = *q;
    *q = temp;
}
```

Referenze e Strutture

- Le referenze funzionano molto bene con le **strutture** e le **classi**.
- Possono essere utilizzate come **argomenti di funzioni** quando si vuole modificarne il contenuto oppure quando si vuole **passare una struttura od oggetto pesante** e non si vogliono usare i puntatori.
- **ATTENZIONE:** Quando si ritorna una referenza **evitare di ritornare una referenza** ad una variabile od oggetto **automatico**, che cessa di esistere quando si esce dalla funzione.

```
struct complex{ double x; double y};
```

```
complex & add(complex z1,complex z2)
{
    complex ztot;
    ztot = z1 + z2;

    return ztot;
}
```

```
complex add(complex z1,complex z2)
{
    complex ztot;
    ztot = z1 + z2;

    return ztot;
}
```

Funzioni che ritornano Strutture

- Per evitare problemi, è possibile **ritornare** una **referenza passata come argomento** alla funzione.

```
complex & add( complex & z1, complex & z2)
{
    z1 = z1 + z2;
    return z1;
}
```

- Oppure utilizzare **new** per **creare nuovi spazi di storage**

```
complex & add( complex & z1, complex & z2)
{
    complex * ztot = new complex;
    *ztot = z1 + z2;
    return *ztot;
}
```

Quando usare le Referenze

- Per permettere ad una funzione di **modificare l'oggetto** passato come **argomento**.
- Per **velocizzare** il programma, passando una referenza invece di copiare tutti i dati dell'oggetto passato.
- Ho una funzione alla quale devo passare dei dati.
- Cosa utilizzo: **referenza**, **puntatore** oppure **pass-by-value** ?
 1. se l'**oggetto** è **piccolo** (struttura o tipi semplici) \Rightarrow **pass-by-value**
 2. l'argomento è un **array** \Rightarrow usare un **puntatore** (unica possibilità)
 3. l'argomento è una struttura di dimensioni ragguardevoli: usare un **puntatore** o una **referenza**
 4. l'argomento è un **oggetto** di una **classe** \Rightarrow usare una referenza.
(Questa è una delle motivazioni per cui sono state introdotte nel C++)

Function Overloading

È possibile avere più funzioni con lo stesso nome, ma con una differente lista di parametri

```
void print(const char *message);  
void print(string message);  
void print(const int * start, const int * end);  
void print(cont int misure[], size_t dim);
```

- le quattro funzioni, come dice il nome, stampano qualcosa sullo schermo, quello che cambia sono gli argomenti che vengono passati

```
void print(const char * m)  
{  
    cout << m << endl;  
}
```

```
void print(const int a[], int n)  
{  
    for (int i=0; i<n; i++)  
        cout << a[i] << " ";  
    cout << endl;  
}
```

```
void print(string s)  
{  
    if (! s.empty() )  
        cout << s << endl;  
}
```

```
void print(const int *s, cont int *e)  
{  
    while (s != e)  
        cout << *s++ << " ";  
    cout << endl;  
}
```


Default Arguments

- È possibile specificare **valori di default** per gli argomenti di una funzione
- Il valore di **default** è **utilizzato automaticamente** se viene **omesso il parametro** corrispondente nella **chiamata** ad una funzione.
- I valori dei default vengono specificati nel **prototipo** della funzione.

```
int test1( int a, int b, double z = 0.0);
```

```
double & ref( int & a, int b=0, char *p); // Errato!
```

```
double & ref( int & a, int b=0, char *p=NULL); // OK!
```

- Una volta specificato un argomento di default, tutti i successivi (da sx a dx) devono essere specificati.

Esempi :

```
test1( 5, 10, -12.0);
```

```
test1( 5, 10); // Terzo argomento preso di default
```