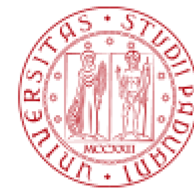
Dipartimento
di Fisica
e Astronomia
Galileo Galilei

800 1222·2022 ANNI



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

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.;                                   3
     cout << "Cube of " << R0 << " is " << cube( R0 ) << endl;
     return 0;
   }

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

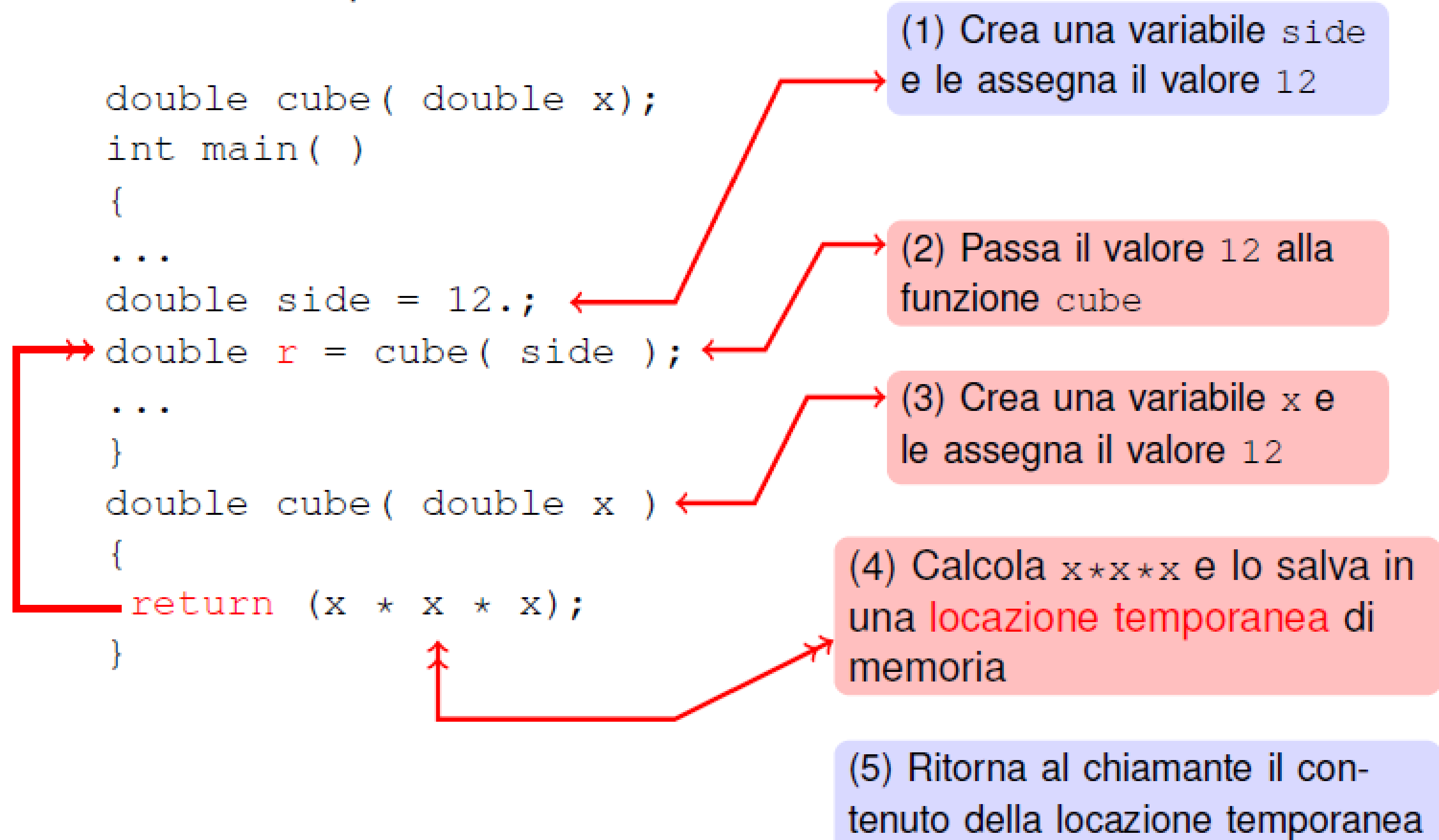
Utilizzo del Prototipo

- 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()`.

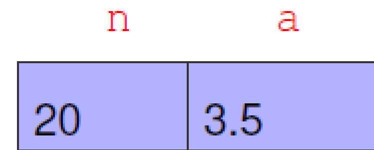
La Chiamata di una Funzione

- Studiamo la procedura con cui viene invocata una funzione

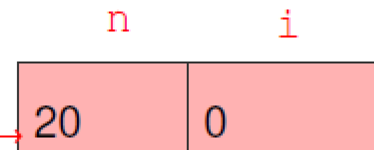


Funzioni e Variabili Locali

```
...
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++)
      ...
}
```



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



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.

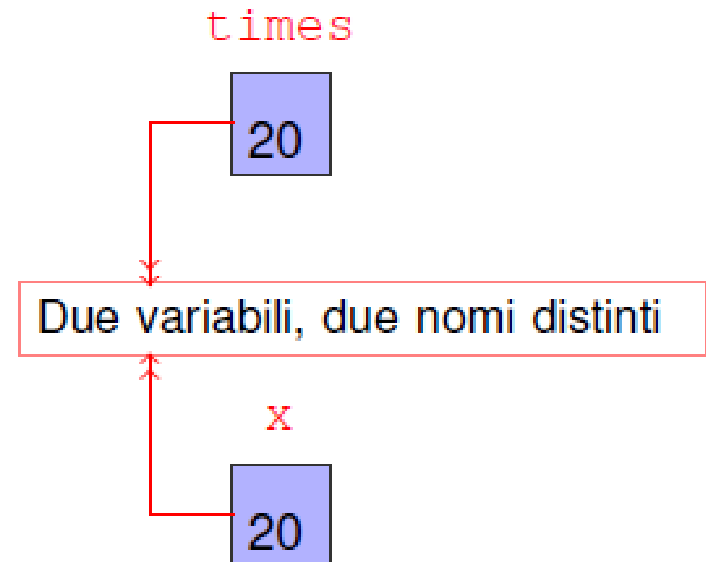
- Il C++ fornisce un'alternativa che è il call by-reference

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 );
 ...
}
```

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

Crea una variabile `times` e assegna il valore 20

`times`

20

Una variabile, due nomi

Crea `x` come alias a `times`

Swap di due Variabili

Si vuole scambiare il contenuto di due variabili:

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

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

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

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

```
a = 7 b = 12
```

```
a = 12 b = 7
```

Swap: *pass-by-value*

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

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

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

swap_value( v1, v2);

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

v1 = 7 v2 = -12

v1 = 7 v2 = -12

```
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;

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

swap_ref( v1, v2);

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

v1 = 7 v2 = -12

v1 = -12 v2 = 7

```
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
```

Dipartimento di Fisica e Astronomia Galileo Galilei

1222 · 2022 800 ANNI

Università degli Studi di Padova

Sperimentazioni di Fisica I mod. A – Lezione 12

Lettura e Scrittura da/su File

Dipartimento di Fisica e Astronomia "G. Galilei",
Università degli Studi di Padova

#include <fstream>

Per gestire la scrittura e la lettura da file si deve includere il file di libreria fstream con il comando:

```
#include <fstream>
```

Vengono quindi abilitati gli oggetti:

`ifstream` : per la gestione del/dei file di lettura

`ofstream` : per la gestione del/dei file di scrittura

Scrittura su File: `ofstream`

```
#include <fstream>
…
string nome1 = "nomeoutputfile.txt";          1
ofstream outputfile1(nome1);
…
ofstream outputfile2("nome2.txt");
…
int a, b, c;
string d, e, f;
…
outputfile1 << a << endl;
```

(analogo a `cout << a << endl;`)

```
outputfile2 << b << d << endl;
outputfile1 << d << e << endl;
outputfile2 << f << endl;
```

Esempio di Scrittura

```
// Programma che apre un file in scrittura, genera numeri casuali e li scrive
in un file
#include <iostream>
#include <fstream>
#include <random>
using namespace std;

int main(){
    int esami;
    cout << "Quanti voti nell'intervallo [18,30] vuoi generare? ";
    cin >> esami;
    ofstream libretto("Voti_OutputFile.txt"); //apro un file di scrittura
    if(!libretto){    //verifica corretta apertura del file
        cout << "Errore in apertura del file di output." << endl;
        return -1;
        }
    uniform_int_distribution<unsigned> u(18,30);
    default_random_engine e;
    for (int q=0; q < esami; q++)
        libretto << u(e) << endl;
return 0;
}
```

Lettura da File: `ifstream`

```
#include <fstream>
…
string input1 = "nomeinputfile.txt";
ifstream inputfile1(input1);
…
ifstream inputfile2("input2.txt");
…
int a, b, c;
string d, e, f;
…
inputfile1 >> a;   (analogo a cin >> a;)
inputfile2 >> b >> c;
getline(inputfile1, d); (analogo a getline(cin,d);)
inputfile2 >> e >> f;
```

Esempio di Lettura

```
// Programma che legge i voti del libretto
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;
int main() {

    ifstream iFile("Voti_OutputFile.txt");
    if (! iFile) // Always test file open
        {
        cout << "Error opening input file" << endl;
        return -1;
        }
    vector<int> v1;
    int value;
    while (iFile >> value)
        v1.push_back(value);

cout << "Il vettore ha " << v1.size() << " elementi:" << endl;
return 0;
}
```

Lettura/scrittura file multipli

```
#include <iostream>
#include <fstream> //file stream

using namespace std;

// for running you must have a series of input files with name: input_0.txt input_1.txt etc etc

int main() {

        string ifilename, ofilename;
        string in="input_", out="output_", postfix=".txt" ; //names
        int dummy;

        for (int i=0; i<1;++i){


        // USE of string
                ifilename= in+ to_string(i) + postfix;
//              cout << ifilename << endl;
                ifstream ifile (ifilename);

                ofilename = out + to_string(i) + "_proc" + postfix;
                cout << ofilename << endl;
                ofstream ofile (ofilename);

                        while (ifile>>dummy){

                                ofile << ++dummy << endl;

                        }


        }

        return 0;

}
```

Coding vs debugging

Debug

In caso di crash, segmentation faults o errori logici, cosa fare?

1) cout << " test"<< endl;

2) debugger (GDB):
– ricompilare con il flag '-g'
(i.e. g++ -g prova.cxx -o prova).
mantiene il numero di riga nell'eseguibile

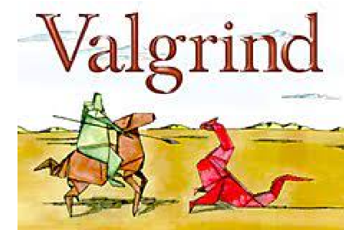
– eseguire il programma nel debugger:
linux> gdb prova

gdb> run <argomenti da riga di comando nel caso vanno qui>
(aspettare per il crash)

gdb> where
(mostra la linea di codice dove è avvenuto il crash)

gdb> quit
(esce debugger)

APPROFONDIMENTO: possibile inserire breakpoints e stampare valori variabili



Debug

Perdite/corruzione di memoria (e.g. scrittura oltre la dimensione di un array) e profiling –> VALGRIND

– ricompilare con il flag ‘-g’
(i.e. g++ -g prova.cxx -o prova).
mantiene il numero di riga nell’eseguibile

– Rilanciare il programma con valgrind
linux> valgrind prova

– in caso di memoria corrotta, gli errori vengono stampati su schermo (stack), insieme al numero di riga
– in caso di perdite di memoria, il totale viene mostrato. Per un report dettagliato (dove c’è stata allocazione ma non distruzione) rilanciare:
linux> valgrind –-leak-check=full prova

Ambienti integrati di sviluppo IDE

- Coding, building, debugging in the same environment
- *Eclipse, Kdevelop, Anjuta, Visual studio, Xcode* e molti altri
- Diversi lunguaggi supportati da ciascun IDE

Consigli per un buon esame

- Studiare (... in modo critico)
- Ripetere! Ripetere! Ripetere (a voce alta! e meglio se a qualcuno)
- Esercitarsi a programmare (l’informatica non è teorica): *segment. faults* aiutano più di mille esempi. Non scoraggiatevi è così per tutti.
- Esercitarsi a risolvere problemi (sopratutto che troverete nel laboratorio di misure) tramite la scrittura di codice. Ciò valorizza quanto avete imparato più del voto (che vi potra ± soddisfare)

Email

- daniele.mengoni@unipd.it

Inoltre solo per esercizi di programmazione potete scrivere anche a:

- jakub.skrowonski@phd.unipd.it
- sara.pigliapoco@phd.unipd.it
- stefano.tornamienti@studenti.unipd.it