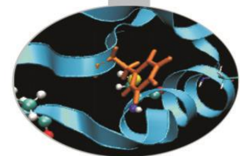
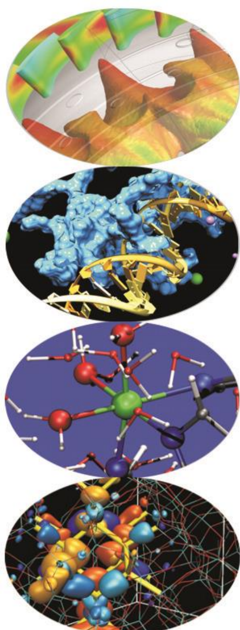
CINECA SCAI
SuperComputing Applications and Innovation

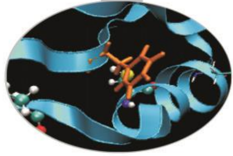


Funzioni I Parte

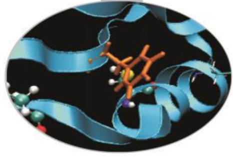


CINECA

Indice



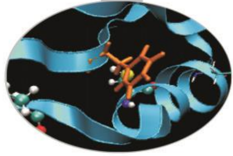
- **Le funzioni: dichiarazione, definizione e chiamata**
- **Il passaggio degli argomenti per valore e riferimento**
- **La funzione *main()***
- **Le regole di visibilità**



Le funzioni

- Tutti i linguaggi hanno costrutti che consentono di separare e impacchettare istruzioni di codice.
- Il C utilizza il concetto di funzione per impacchettare porzioni di codice che possono essere eseguite più volte nel programma
- La funzione ha un nome, un tipo di dato restituito e una lista di argomenti che vengono passati quando la funzione viene chiamata.

Le funzioni: la dichiarazione



- La *dichiarazione* di una funzione viene detta *prototipo* della funzione e serve per informare il compilatore del nome della funzione, del tipo di dato restituito e del numero, tipo ed ordine degli argomenti passati alla stessa. Gli argomenti passati ad una funzione prendono anche il nome di *parametri formali*.

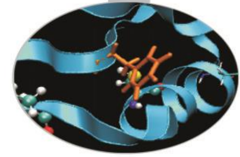
tipo_restituito nome_funzione(lista_argomenti);

- esempio:

```
int max_part_int( double, double ) ;
```

Questa è la dichiarazione della funzione max_part_int che riceve due parametri di tipo double e restituisce un dato di tipo int.

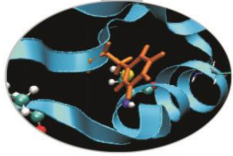
Le funzioni: la definizione



- La *definizione* di una funzione richiede a sua volta il tipo di dato restituito, il nome della funzione, e la lista degli argomenti passati alla funzione (cioè il suo prototipo). Il prototipo della funzione deve, inoltre, essere seguito da una serie di dichiarazioni ed istruzioni incluse tra parentesi graffe (*corpo della funzione* o *inizializzatore*).

```
tipo_restituito   nome_funzione(lista_argomenti){
        statement1;
        statement2;
        …
        statementN;
}
```

Esempio

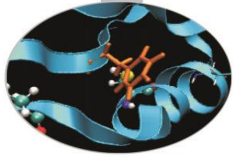


```
int max_part_int(double x, double y) {
    int a, b, max ;
    a=x ;
    b=y ;
    if( a >= b )
       max = a;
    else
       max = b;
    return max; }
```

Questa è la definizione della funzione max_part_int: essa riceve come argomenti le variabili di tipo double x ed y e restituisce, tramite il comando ***return*** la variabile max di tipo intero.

Il prototipo di una funzione è, in generale, obbligatorio. Se la definizione della funzione compare nel codice prima di ogni sua chiamata, il prototipo non è più necessario in quanto contenuto nella definizione stessa.

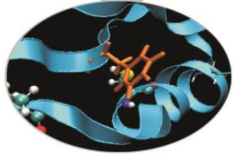
Le funzioni: la chiamata



- Una funzione viene chiamata da un’altra porzione di codice, per es. dal main(), semplicemente scrivendo il suo *nome* accompagnato dalla *lista degli argomenti* passati:

```
#include<iostream>
using namespace std;
int max_part_int(double, double) ;  // prototipo della
                                     // funzione
int main(){
   double doub_1, doub_2;
   cout << "Enter two doubles" << endl;
   cin >> doub_1 >> doub_2 ;
   cout << "The largest integer part is:"
        << max_part_int(doub_1, doub_2);
   cout << endl;
   return 0;
}
```

Le funzioni: void type



- *void* è un tipo di dato formalizzato nell' ANSI C che significa 'niente'. Può essere usato sia come return value, sia come argomento di una funzione, per indicare che non ritorna nulla o che non riceve argomenti in ingresso.

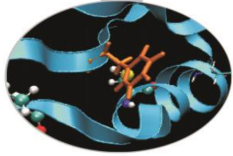
```
// Riceve nulla e ritorna nulla
void test (void);

// Riceve nulla e ritorna un float
float test(void);

// Ritorna nulla e riceve un float
void test(float a);

// Riceve nulla e ritorna nulla
void test();
```

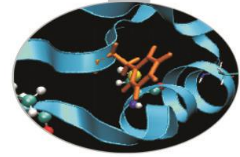
Le funzioni: il passaggio degli argomenti



- Il passaggio degli argomenti ad una funzione può avvenire in tre modi differenti: per valore, per puntatore, per reference (C++).

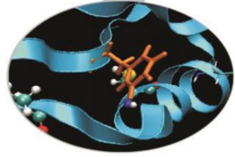
- Le ultime due modalità sono sostanzialmente identiche e vanno entrambe sotto il nome di passaggio per *riferimento*.

Le funzioni: il passaggio degli argomenti



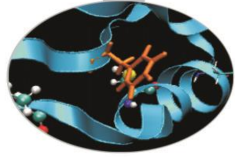
- Il passaggio per *valore* implica l'allocazione di nuove aree di memoria destinate a contenere *copie* degli argomenti passati sulle quali la funzione agisce e che vengono, poi, distrutte al termine dell'esecuzione della funzione stessa. Questa procedura è utile quando non si vogliono modificare i valori che gli argomenti della funzione assumono nella sezione chiamante.

Le funzioni: il passaggio degli argomenti



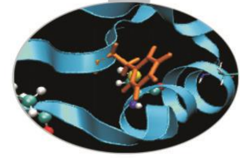
•Il passaggio per *puntatore* e per *reference*, invece, comporta un notevole risparmio di tempo e di memoria. Esso richiede, infatti, solo una quantità di memoria delle dimensioni di 4 o 8 byte, quanto basta cioè per contenere un *indirizzo* di memoria: per questa ragione tale modalità è consigliata qualora si debba passare alle funzioni una grande quantità di dati. A differenza di quanto accade nel passaggio per valore, la funzione può ora accedere *direttamente* ai parametri formali specificati nella chiamata, attraverso i loro indirizzi: in questo modo il valore degli argomenti risulta modificato anche nella sezione chiamante.

Le funzioni: il passaggio degli argomenti



•Per evitare che gli argomenti passati alla funzione vengano modificati all'interno della stessa, bisogna dichiararli ***const***. Quando è necessario trasmettere una cospicua quantità di byte ad una funzione, è preferibile utilizzare il passaggio per reference o per puntatore, con il rischio, però, che i dati possano risultare modificati nella sezione chiamante: è questo un caso in cui è consigliabile passare gli argomenti come reference costanti o puntatori a locazione costante.

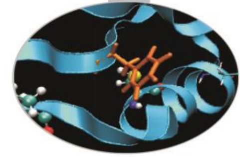
Il passaggio per valore: la funzione swap



•Proviamo ora a scrivere un programma che scambi il valore di due numeri dati passandoli per valore alla funzione swap.

```
#include<iostream>
using namespace std;
void swap(int, int);    // prototipo della funzione
int main(){
   int a=12, b=20;
   cout << " value of a:" << a << endl;
   cout << " value of b:" << b << endl;
   swap(a,b);              // passaggio per valore
   cout << " value of a:" << a << endl;
   cout << " value of b:" << b << endl;
   return 0;
}
```

Il passaggio per valore: la funzione swap



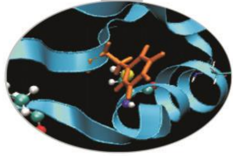
```
void swap (int a_s, int b_s){  // significa a_s = a ...
    int tmp;
    tmp = a_s;
    a_s = b_s;
    b_s = tmp;
}
```

L'output che otteniamo è il seguente:

```
 value of a: 12
value of b: 20
value of a: 12
value of b: 20
```

ovvero i valori *non* sono stati scambiati. In realtà, lo scambio è stato fatto sulle copie di a e b create dalla funzione swap (a_s e b_s), poi distrutte in uscita dalla stessa, senza intaccare i valori di a e b nel main().

SCAI SuperComputing Applications and Innovation



Il passaggio per puntatore: la funzione swap

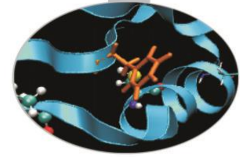
- Modifichiamo il programma precedente in modo che i valori di a e di b siano passati per puntatore alla funzione swap:

```
#include<iostream>
using namespace std;
void swap(int*, int*);  // prototipo della funzione
                        // con il passaggio di puntatori
int main()
{
   int a=12, b=20;
   cout << " value of a:" << a << endl;
   cout << " value of b:" << b << endl;

   swap(&a, &b);     // nota: passaggio di indirizzi
   cout << " value of a:" << a << endl;
   cout << " value of b:" << b << endl;
   return 0;
}
```

CINECA

Il passaggio per puntatore: la funzione swap



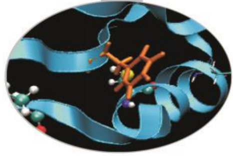
```
void swap(int* ptr_a, int* ptr_b) // int* ptr_a = &a
{
     int tmp;
     tmp=*ptr_a;
     *ptr_a=*ptr_b;
     *ptr_b=tmp;
}
```

Questa volta otteniamo l'output desiderato, ovvero:

```
value of a:12
value of b:20
value of a:20
value of b:12
```

Infatti, utilizzando i puntatori ad a e b, i valori delle due variabili sono stati modificati anche nel main().

Il passaggio per reference: la funzione swap

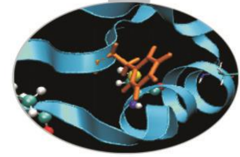


- Possiamo ottenere uno scambio corretto dei valori di a e di b utilizzando i reference al posto dei puntatori.

```
#include<iostream>
using namespace std;
void swap(int&, int&);     // prototipo della funzione con
                           // il passaggio di reference
int main() {
   int a=12, b=20;
   cout << " value of a:" << a << endl;
   cout << " value of b:" << b << endl;

   swap(a, b);             // passaggio di indirizzi, benché non
                           // compaia l'operatore &
   cout << " value of a:" << a << endl;
   cout << " value of b:" << b << endl;
   return 0;
}
```

SCAI
SuperComputing Applications and Innovation



Il passaggio per reference: la funzione swap

```
void swap (int& ref_a, int& ref_b){
                    // significa int& ref_a = a ecc.
    int tmp;
    tmp = ref_a;    // la dereferenziazione è automatica
    ref_a = ref_b;
    ref_b = tmp;
}
```

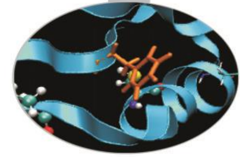
Ancora una volta otteniamo in uscita dal programma:

```
value of a:12
value of b:20
value of a:20
value of b:12
```

Con l'uso dei reference il programma funziona correttamente ed appare più leggibile rispetto alla versione con i puntatori.

CINECA

Il qualificatore const



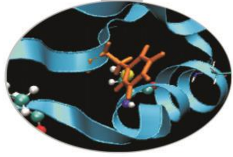
•Il qualificatore *const* può essere utilizzato nella dichiarazione degli argomenti di una funzione per indicare che il codice che utilizzerà quella variabile non andrà a modificarne il valore.

```
void foo(const int a );        // prototipo della funzione con
                             // parametro costante
```

Il qualificatore non è obbligatorio ma:

- chiarisce il codice per chi lo legge
- aiuta il compilatore nella creazione dell’eseguibile

La funzione main() ed i suoi argomenti



•La funzione main(), che rappresenta l'entry point di ogni programma, può ricevere in ingresso *due* argomenti: un *intero* ed un *array* di puntatori a caratteri, generalmente indicati con **argc** ed **argv[ ]**.

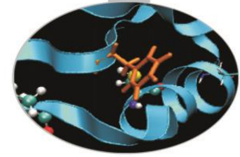
Il prototipo del main() è pertanto:

*int main(int **argc**, char* **argv[]**)*

•L'intero **argc** rappresenta il numero di argomenti passati al programma sulla riga di lancio. Viene contato anche il nome del programma.

•Le componenti dell'array **argv[ ]** sono puntatori agli argomenti passati al programma. Tali argomenti sono tutti trattati come stringhe di caratteri.

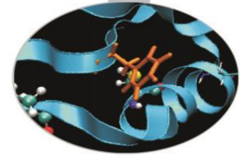
La funzione main() ed i suoi argomenti



•Scriviamo un semplice programma che calcoli la somma di due numeri passati come argomenti:

```
// file mainEx.cpp compilato come mainEx.x
#include<iostream>
#include<stdlib.h>
using namespace std;
int main(int argc, char* argv[ ]){
  if(argc != 3){
    cout << "Usare la sintassi: " <<
           " nomeProgramma int1 int2" << endl;
    exit(1);
  }
  cout << "Il nome del programma e': " << argv[0] << endl;
  int int1 = atoi(argv[1]);  // trasformare una
                            // stringa di char in un int
```

La funzione main() ed i suoi argomenti

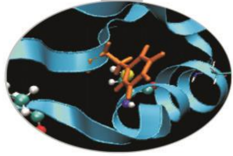


```
  int int2 = atoi(argv[2]);
  int sum = int1 + int2;
  cout << "La somma di " << argv[1] << " e " << argv[2]
       << " e': " << sum << endl;
  return 0;
}
```

- Il programma può generare i seguenti output:

```
>> mainEx.x
Usare la sintassi: nomeProgramma int1 int2
>> mainEx.x 4
Usare la sintassi: nomeProgramma int1 int2
>> mainEx.x 20 12
Il nome del programma e': mainEx.x
La somma di 20 e 12 e': 32
```

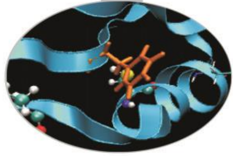
Le regole di visibilità (scope)



Ciascuna variabile definita all'interno di un programma ha un proprio scope di appartenenza che ne determina la visibilità.
Le variabili non possono essere utilizzate al di fuori dello scope in cui sono definite.

- Il body di una function definisce uno scope locale.
- Variabili definite nel corpo di una funzione hanno un tempo di vita determinato dal corpo della funzione.
- Ciascuna chiamata a funzione alloca uno stack frame con le variabili locali definite all'interno della funzione

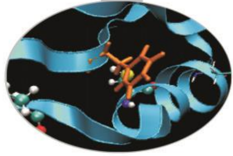
Le regole di visibilità (scope)



Distinguiamo 3 casi diversi:

1. Il programma è costituito da un solo file composto dalla funzione main().
2. Il programma è costituito da **un solo file** che contiene varie funzioni oltre al main().
3. Il programma è costituito da **più file**.

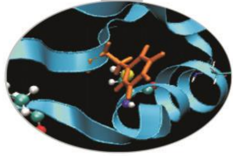
Le regole di visibilità (scope)



***Caso 1**: Il programma è costituito da **un solo file** composto dalla funzione main():*

• La visibilità di ciascuna variabile è a livello di blocco, ovverosia una variabile non è accessibile dall'esterno del blocco in cui è definita (variabile **locale**).

• Una variabile definita esternamente al main() è invece accessibile da ogni blocco: la sua visibilità è a livello di file (variabile **globale**).

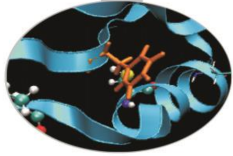


Esempio

```
#include<iostream>
using namespace std;
int varA=1;            // variabile globale
int main(){
  {
    int varB=2;        // variabile locale
    int varA=3;        // variabile locale,
                       // occulta la variabile globale varA
    cout << varB << "  " << varA << endl;
  }
  int varB=4; // variabile locale, la varB definita in
                   // precedenza non esiste più
  cout << varB << "  " << varA << endl;
  return 0;
}
```

L'output è il seguente:

```
2  3
4  1
```



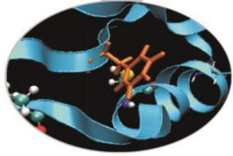
Regole di visibilità

Caso 2*: Il programma è costituito da* ***un solo file*** *che contiene varie funzioni oltre al main().*

La visibilità è, più in generale, a livello di funzione (all'interno di ogni funzione la visibilità è a livello di blocco).

Una variabile definita al di fuori del main() e di ogni altra funzione è **globale**: il suo valore è accessibile da ogni sezione del programma.

Regole di visibilità

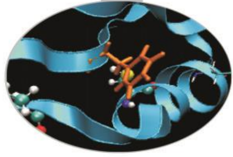


Ogni variabile definita all'interno di una funzione è **locale**, ovvero non è visibile dalle altre funzioni.

In uscita dalla funzione, il valore di ogni variabile locale viene cancellato dalla memoria.

Se, all'interno di una funzione, una variabile è definita **static**, la sua visibilità è ancora limitata alla funzione, ma il suo valore viene conservato in memoria e sarà disponibile ad una successiva chiamata alla funzione.

Esempio



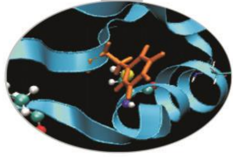
```
#include<iostream>
using namespace std;

void stampa(){
  int varS1 = 0;
  static int varS2 = 0;
  varS2++;
  varS1++;
  cout      << "varS1 = " << varS1 << "   " <<
          "varS2 = " << varS2 << endl;
}

int main(){
  for(int i=0; i<5; i++){
    stampa();
  }
  return 0; }
```

Output:

```
varS1 = 1  varS2 = 1
varS1 = 1  varS2 = 2
varS1 = 1  varS2 = 3
varS1 = 1  varS2 = 4
varS1 = 1  varS2 = 5
```



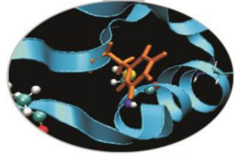
Regole di visibilità

Caso 3*: Il programma è costituito da* ***più file****.*

Le variabili **globali** vengono definite (o semplicemente dichiarate) come tali in un solo file, mentre vengono dichiarate come **extern** in tutti gli altri file: la loro visibilità è, così, effettivamente a livello di programma.

Una variabile **globale** definita come **static** ha, invece, visibilità ridotta a livello di file: è cioè visibile solamente dalle funzioni presenti nel file in cui è stata definita.

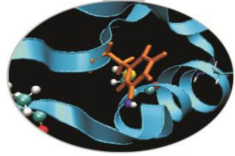
Esempio



```
// file main.cpp
#include<iostream>
using namespace std;
int esp = 2;                    // variabile globale
int sum(int num);
double radq();

int main() {
  int num;
  cout << "Calcolo della somma dei quadrati dei primi N"
       << " numeri naturali" << endl;
  cout << "Inserisci N: ";
  cin >> num;
  cout << "La somma dei quadrati dei primi " << num
       << " numeri naturali e' " << sum(num) << endl;
  cout << "La sua radice quadrata vale: " <<
            radq() << endl;
  return 0;
}
```

Esempio



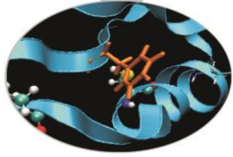
```
// file funzioni.cpp
#include<iostream>
#include<math.h>
using namespace std;
static int somma=0;  // variabile globale
                         // con scope di file
extern int esp;      // variabile globale

int sum(int num){

   for (int i=1; i<num+1; i++)
      somma += (int)pow((double)i, (double)esp );
   return somma;

}

double radq(){
   return(sqrt((double)somma ));
}
```



Esempio

output:

```
>> Calcolo della somma dei quadrati dei primi N numeri naturali
>> Inserisci N: 10
>> La somma dei quadrati dei primi 10 numeri naturali e' 385
>> La sua radice quadrata vale: 19.6214
```