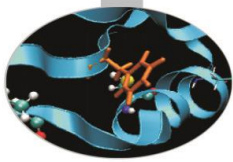
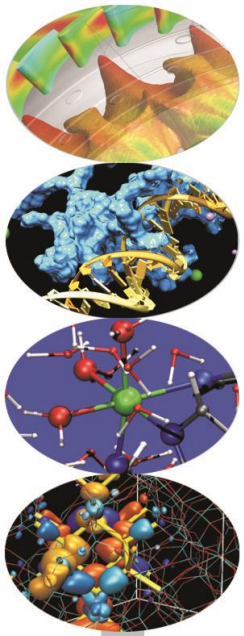
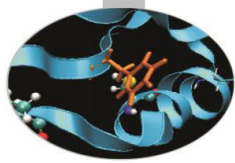


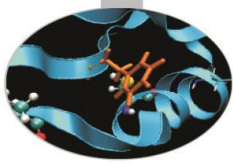
Sintassi III Parte





Indice

- Gli array
- Array multidimensionali
- I puntatori
- L'aritmetica dei puntatori
- Puntatori ed array
- I reference
- L'attributo const a puntatori e reference

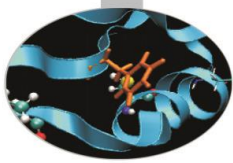


Array

- Per un tipo T t[size] è un array di size elementi del tipo T.

```
type name [elements];
```

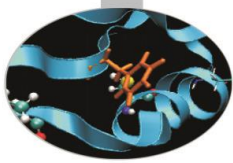
- Gli elementi sono indicizzati da 0 a size-1.
- Il numero di elementi che costituiscono un array deve essere un'espressione costante
- Gli array sono solo monodimensionali, per rappresentare array multidimensionali si usano gli array di array.
- In C/C++ non è consentito assegnare un vettore ad un altro, bisogna assegnare individualmente ogni elemento.



Array

Esempi:

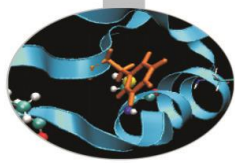
```
double a[3];      /* array di 3 double: a[0], a[1], a[2]*/


int k=6;
int w1[k];    /* errore l'array deve avere una dimensione costante*/


double w_w[3][6]; /* w_w è un array di 3 array di 6 double. Di
fatto                                          è una matrice 3 x 6.*/
int v1[5], v2[5];
v1 = v2;          // errore in compilazione !!!!!
```



Array

L'inizializzazione può essere fatta in diversi modi e se esplicita in altro modo può essere omessa la dimensione che viene rilevata in fase di compilazione. Tuttavia in C++ come in C non è presente alcun controllo da parte del compilatore sul superamento del limite dell'array. Ecco alcuni esempi:

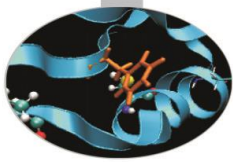
```
int v[3];
double w[ ] = {3.5, -65.7, 8.0, 0.0} ;
bool b[2] = {true, false};   /* l'array contiene size elementi
                                del tipo T*/
int h[1] = { 2,3,4};   /* ERRORE!! Numero elementi > size*/
int m[3] = {23, 12}; /* il compilatore aggiunge gli zeri  necessari
                        per inizializzare il vettore*/
int m[3] = { 23, 12, 0} ; /* è equivalente alla
                             precedente inizializzazione*/
```



Array

Possiamo accedere ai singoli elementi dell'array in lettura e in scrittura con l'operatore []

```
#include <iostream>
using namespace std;
int billy [] = {16, 2, 77, 40, 12071};
int n, result=0;
int main () {
for ( n=0 ; n<5 ; n++ )  result += billy[n];
cout << result;
return 0;
}
```



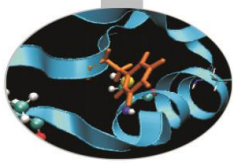
Nota

La mancanza di controllo sui limiti degli array è molto pericolosa poiché può mandare in crash il programma in fase di esecuzione.

Questo è uno dei prezzi da pagare per avere un codice veloce ed efficiente quale è quello scritto in C/C++.

E' compito del programmatore tutelare il codice con attenzione o creando un nuovo tipo di array che abbiano un controllo di limite.

```
int main (){
int boomm[3];
for(int i; i<1000; i++) boomm[i] = i;
return 0;
}
//l'errore sul limite dell'array porta al crash il
//  codice a run-time
```



Array bidimensionali

- E' possibile definire degli array multidimensionali.

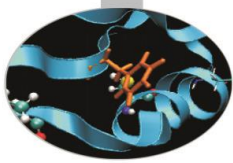
Un array bidimensionale avente *n* righe ed *m* colonne con elementi di tipo *T* può essere definito come *T[n][m]* dove l'indice di riga varia tra 0 e *n-1* e l'indice di riga e l'indice di colonna varia tra 0 e m-1

```
double mt[5][2];  //array 2D 5 per 2
mt[0][0]=1;     //accesso riga 0 colonna 0
double a=mt[0][1]; //accesso riga 0 colonna 1
```

- Un array bidimensionale può essere pensato come un array monodimensionale di array monodimensionali. Per esempio l'array *int [3][4]* si può pensare come un array di 3 elementi ciascuno dei quali è a sua volta un array di 4 elementi (le colonne). Nell'inizializzazione di un array multidimensionale solamente la prima dimensione può essere omessa

```
int u[][3]={{1,2,3},{4,5,6}};  OK
int u[2][];              //NO!!
```

Puntatori(1)



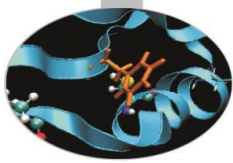
Ogni variabile occupa in memoria, rappresentata come un nastro ordinato di byte, un certo numero di celle contigue.

```
char c='e';
int x;
```

1-byte

4-byte

Symbol	Addr	Value
	0	
	1	
c	2	‘e’
	3	
x	4	
	5	
	6	
	7	



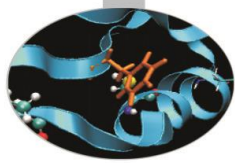
Puntatori(2)

Per un tipo T, T* è il tipo puntatore ad una variabile di tipo T; una variabile di questo tipo è detta puntatore e può contenere l'indirizzo in memoria di un oggetto del tipo T.

L'operazione inversa all'indirizzamento è la dereferenziazione. Per accedere all'indirizzo in memoria di una variabile di qualunque tipo si utilizza l'operatore &.

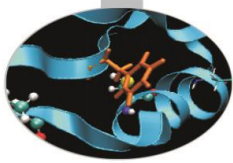
```
char a='f' ;

char *pc = &a;        // pc contiene l'indirizzo di a.

 double x = 23.7;

double *pd = &x;      // pd contiene l'indirizzo di x.

*pd =30.7;            /*il valore contenuto nella variabile
                        puntata da pd è 30.7*/
```

Puntatori (2)



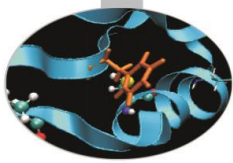
- Notiamo la differenza tra l'operatore di indirizzamento e di dereferenziazione:

- & operatore di indirizzamento : 'indirizzo di '
- * operatore di dereferenziazione: 'valore puntato da'



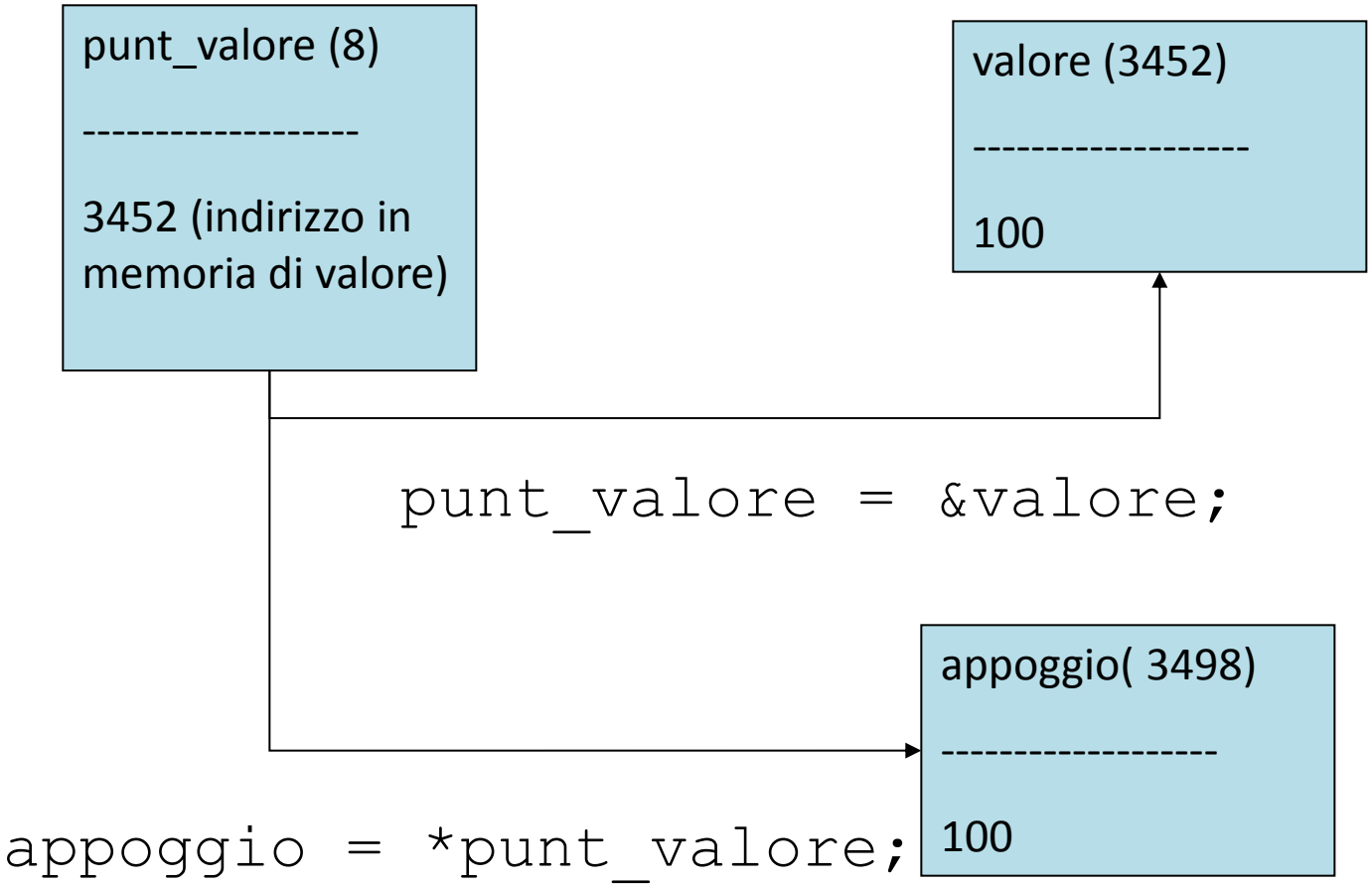
Esempio

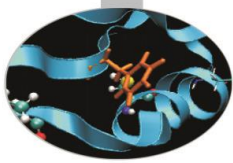
```
#include <iostream>
using namespace std;
int main (){
      int valore = 100;
      int *punt_valore;
      int appoggio;
      punt_valore = &valore;
      appoggio = *punt_valore;
      cout << valore;
      cout << appoggio;
      return 0;
}
```



Esempio

graficamente:





Puntatori (3)

- Un puntatore è legato al tipo di variabile cui punta, per cui:

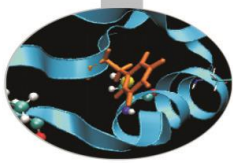
```
int x= 3;
double *pd = &x; // ERRORE non c'è coerenza di tipo
```

- Tuttavia la conversione può essere imposta:

```
int x= 3;
double *pd = (double*) &x; // questo è accettato dal compilatore.
```

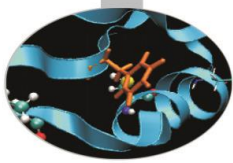
- Non esiste in fase di compilazione un controllo se non sul tipo della variabile puntata. Per questo e poiché nessun oggetto occupa la posizione zero in memoria, viene scelto come convenzione di inizializzare i puntatori a 0 una volta istanziati.

```
int *pi = 0;          // in questo momento pi punta 0.
int i;
pi = &i;      // ora pi punta l'indirizzo occupato da i in memoria.
```



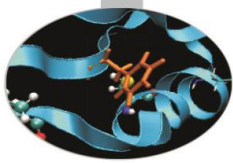
Aritmetica dei puntatori

- È possibile sommare un puntatore ed un offset int, incrementare un puntatore e decrementare un puntatore. È anche sensato fare la differenza tra due puntatori.
- Puntatore T* + offset int = puntatore T* che punta offset elementi oltre nella memoria.
- Incremento di T *t; t++ → puntatore T* che punta un elemento oltre. Analogamente per il decremento (t--).
- La differenza tra puntatori restituisce un intero che rappresenta la distanza (offset) tra i due puntatori espressa come numero di elementi di quel tipo.



Esempio

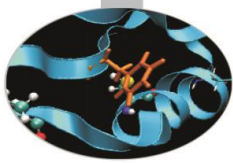
```
char a;


char *pc = &a;      // pc punta all'indirizzo di a

pc++;               // pc punta "un sizeof(char) caselle "di memoria oltre a

pc += 3;  /* pc punta "3 x sizeof(char) caselle "di memoria oltre
             a dove puntava prima*/

--pc;       /* pc punta "un sizeof (char) caselle" di memoria
                          prima rispetto a dove puntava*/

char *pc1 = pc;  /* definisco un nuovo puntatore allo stesso
                                    tipo che punta alla stessa zona */

cout << pc1- pc << endl;  // la differenza tra questi == 0
```



Puntatori e array

- Puntatori e array sono intimamente legati tra loro. Dato un array, il nome dell'array senza le [ ] restituisce un puntatore (dello stesso tipo degli elementi dell'array) al primo elemento dell'array (posizione 0).
- In questo senso l'aritmetica dei puntatori trova un'applicazione immediata per lo scorrimento di un vettore.

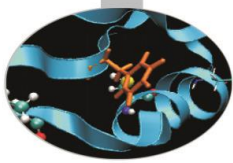
```
int s[ ] ={1,2,3,4,5};
 int *ps = s; // conversione implicita da int[ ] a int*
cout<< *(ps + 3) << endl;
 cout << s[3] << endl ; // queste due righe danno lo stesso output
s = ps;        /* ERRORE questo assegnamento non è consentito poiché si
                  perde la dimensione del vettore */
```



Const e puntatori

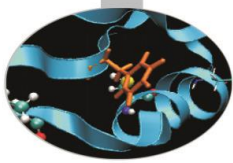
- Nell'utilizzo di un puntatore vengono coinvolte due entità, il puntatore e l'oggetto puntato. Utilizzando il qualificatore prefisso const nella dichiarazione di un puntatore rendiamo costante l'oggetto puntato, non modificabile quindi tramite il puntatore, ma non il puntatore in sé.
- Per dichiarare un puntatore in se stesso come const dobbiamo utilizzare la notazione (**const*) al posto del semplice (*).
- Osservazione: l'operatore per dichiarare un puntatore costante impone di essere usato post posto al tipo a cui punta il puntatore, usato scorrettamente fornisce solo un puntatore ad una costante.

```
char *const pc //puntatore costante OK!
char const* pc; /*è equivalente a */ const char* pc1;
```



Const e puntatori

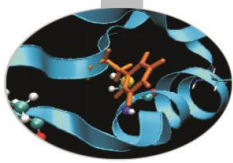
```
char s [ ] = "abcd" ;
char *p;
const char *pc =s;  //puntatore alla  stringa costante s
pc [2] = 'e' ;    /* ERRORE l'oggetto puntato da pc è
                          dichiarato costante */
pc = p;                   /* va bene perché pc in sé non deve
                                restare  costante*/

char *const cp = s ;     // qui il puntatore è costante
cp [3] = 'm' ;           /*qui va bene perché l'oggetto
                                puntato non è costante */
cp = p;          // ERRORE non posso modificare il puntatore.

const char *const cpc = s ;   /* puntatore costante a
                                costante*/
cpc [3] = 'm' ;               // errore
cpc = p;                      // errore
```



Reference

- Un reference è un nome alternativo per un oggetto. L'uso di questo strumento è principalmente nel passaggio e nel ritorno di valori a e da funzioni.
- È obbligatoria l'inizializzazione.
- Deve essere inizializzato con il nome di una variabile.
- Ogni variazione fatta utilizzando il nome della variabile si riflette sul contenuto della "variabile" reference (in realtà non esiste un'altra variabile è solo un nome alternativo della stessa variabile) e viceversa.

```
long alfa = 4300000 ;
long &r  = alfa ;
alfa = 5000000 ;        // anche r vale 5000000
r += 1000000 ;          // anche alfa vale 6000000.
```



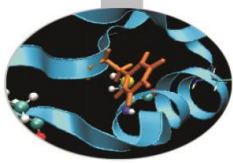
Const e reference

Anche un reference può essere dichiarato costante, in questo modo non si può utilizzare quel nome per modificare l'oggetto cui si riferisce.

A differenza del reference semplice il const & può essere inizializzato con una costante.

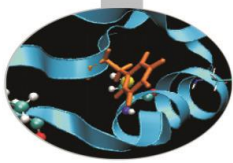
```
const double pi = 3.1415 ;     //costante double
const double &P = pi; // reference costante a costante
double &b = 3.3 ; //ERRORE rvalue const con reference non const
const double &k = 3.3 ; //ok
```

Const e reference



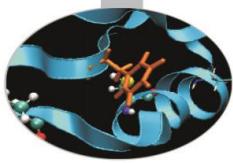
```
int a=5;
const int &b=a;
a=7;                        //OK
printf("%d %d\n",a,b);
b=7;                        //Errore b è const!
                    //assignment of read-only
                    //reference
```

Note su puntatori e reference



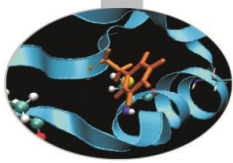
```
int main(){
      int i = 1;
       int *pi;
       int &ri = i;
       pi=&i;
       cout << "valore di i: " << i << endl;
       cout << "valore di &i: " << &i << endl;
       cout << "valore di pi: " << pi<< endl;
       cout << "valore di *pi: " << *pi<< endl;
       cout << "valore di &(*pi): " << &(*pi)<< endl;
       cout << "valore di &pi: " << &pi<< endl;
       cout << "valore di ri: " << ri<< endl;
       cout << "valore di &ri: " << &ri<< endl;
return 0;}
```

Note su puntatori e reference



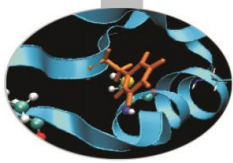
OUTPUT:

```
valore di i: 1
valore di &i: 0x7aff0910
valore di pi: 0x7aff0910
valore di *pi: 1
valore di &(*pi): 0x7aff0910
valore di &pi: 0x7aff0918
valore di ri: 1
valore di &ri: 0x7aff0910
```



Reference Commenti

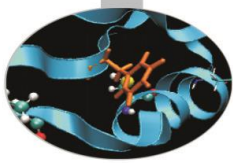
- Tipicamente un reference viene usato in C++ congiuntamente alla parola chiave const per il passaggio di variabile “voluminose” a funzioni; questo aspetto verrà ampiamente discusso nel prossimo modulo.
- In C per ottenere lo stesso risultato si usano i puntatori costanti.
- Essenzialmente all’interno del compilatore un reference è costituito da un puntatore e quindi in ultima analisi il lavoro che possono svolgere è identico. La vera differenza risiede dunque ad un livello più alto, cioè alla rappresentazione formalmente diversa che ha un reference agli occhi di un programmatore (facilità d’uso, immediatezza).
- l vantaggio, facilmente visibile, del tipo reference rispetto ai puntatori è rappresentato dal fatto che una variabile reference, dopo la sua definizione, va trattata esattamente allo stesso modo di una variabile normale e non necessita degli operatori di indirizzamento e deindirizzamento utilizzati dai puntatori.



Reference Commenti

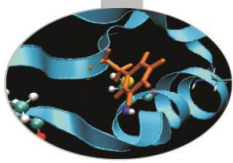
```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int val=3;
    int * pointer=&val;
    int &ref=val;
    printf("step 0 val=%d\n",val);
    *pointer=5;
    printf("step 1 val=%d\n",val);
    ref=100;
    printf("step 2 val=%d\n",val);
    val=500;
    printf("step 3 *pointer=%d\n", *pointer);
    printf("step 3 ref=%d\n",ref);
    return 0;
}
```

Stringhe C-like e string



- Una stringa letterale è una sequenza di caratteri racchiusi tra doppi apici: *“ecco una stringa”*.
- Il C++ fornisce due rappresentazioni per le stringhe letterali: la stringa di caratteri in stile C e il tipo string vero e proprio introdotto nel C++.
- Di seguito introdurremo entrambe queste rappresentazioni in quanto anche l’uso delle stringhe di caratteri alla C si rende ancora indispensabile in alcuni casi pratici (es: gestione delle opzioni da riga di comando).

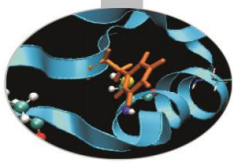
Stringhe C-like e string



- Un array di un appropriato numero di caratteri costanti è una stringa letterale. Ogni stringa letterale è terminata dal carattere speciale '\0', il carattere nullo, che ha valore zero.
- Ogni stringa letterale può essere dichiarata/inizializzata in qualunque dei seguenti modi:

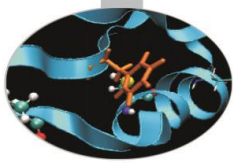
```
char parola[]={'c','i','a','o','\0'};
char parola[5]={'c','i','a','o'};
char parola[ ]="ciao";
char parola[5]="ciao"
char* parola="ciao";
```

Stringhe C-like e string



- Una stringa letterale è, di solito, allocata staticamente e può essere ritornata da una funzione; la maggior parte dei compilatori richiede che essa venga associata ad un array di caratteri perché possa essere modificata nel corso di un programma:

```
char parola[ ] = "roma";
parola[2] = 's' ;
/* char* parola="roma";
   *(parola+2)='s';  errore a run-time */
```



Stringhe C-like e string

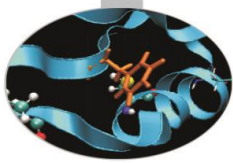
- La scrittura e la lettura di stringhe letterali dalle device standard sono gestite rispettivamente dalle funzioni printf() e scanf():

```
printf("%s \n",<nome_stringa>);
scanf("%s",<nome_stringa>);
```

- La funzione scanf() non richiede l'operatore & davanti a <nome_stringa> perché quest'ultimo rappresenta di per sé un indirizzo di memoria.

- La funzione scanf() presenta l'inconveniente di terminare la lettura di una stringa al primo spazio bianco (oltre a quello della bufferizzazione visto in precedenza); è allora preferibile sostituirla con gets() contenuta, naturalmente, all'interno di stdio.h

```
gets(<nome_stringa>);
```



Stringhe C-like e string

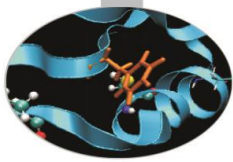
- esempio1:

```
#include<stdio.h>
int main(){
        char ciao[15];
        printf("Scrivi: Ciao Mondo! \n");
        scanf("%s",ciao);
        printf("%s \n",ciao);
        return 0;
}
```

- output:

```
Scrivi: Ciao Mondo!
Ciao Mondo!
Ciao
```

Stringhe C-like e string

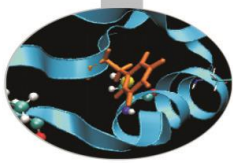


- esempio2:

```
#include<stdio.h>
int main(){
        char ciao[15];
        printf("Scrivi: Ciao Mondo! \n");
        gets(ciao);
        printf("%s \n",ciao);
        return 0;
}
```

- output:

```
Scrivi: Ciao Mondo!
Ciao Mondo!
Ciao Mondo!
```



String

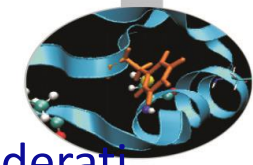
Oltre alle stringhe di tipo C-like, cioè gli array di char, la libreria **<string>** del C++ mette a disposizione il tipo ***string***.

Le variabili di tipo string sono più semplici da maneggiare rispetto alle stringhe C-like: è consentito l'assegnamento di una stringa ad un'altra, il confronto tramite gli operatori logici tradizionali ==,<,<=, !=,>,>= e la concatenazione attraverso l'operatore matematico + .

L'input e l'output delle string dallo stream sono gestiti dagli operatori << e >> rispettivamente.

```
string avverbio="ecco" ;
string articolo="una" ;
string sostantivo="frase" ;
string sentenza;
sentenza = avverbio + " " + articolo + " " + sostantivo;
cout << sentenza << endl;  // output: "ecco una frase"
```

String



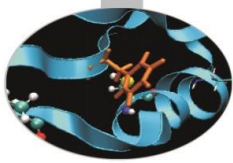
- La lettura di string da tastiera può, tuttavia, portare a risultati indesiderati anche con l'operatore **>>**. Per ovviare a questi problemi si ricorre alla funzione getline() contenuta nella libreria iostream.
- esempio1

```
#include<iostream>
#include<string>
using namespace std;
int main(){
        string str;
        int matricola;
        cout << "Nome e cognome:" << endl;
        cin >> str;
        cout << "Numero di matricola:" << endl;
        cin >> matricola;
        cout << str << " " << matricola << endl;
        return 0;
}
```

- output:

```
Nome e cognome:
Giuseppe Verdi
Numero di matricola:
Giuseppe 134515310
```

String



- esempio2

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

int main(){
        string str;
        int matricola;
        cout << "Nome e cognome:" << endl;
        getline(cin,str);
        cout << "Numero di matricola:" << endl;
        cin >> matricola;
        cout << str << " " << matricola << endl;
        return 0;
}
```

- output:

```
Nome e cognome:
Giuseppe Verdi
Numero di matricola:
641
Giuseppe Verdi 641
```