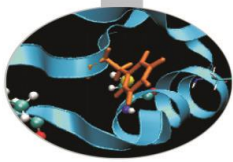
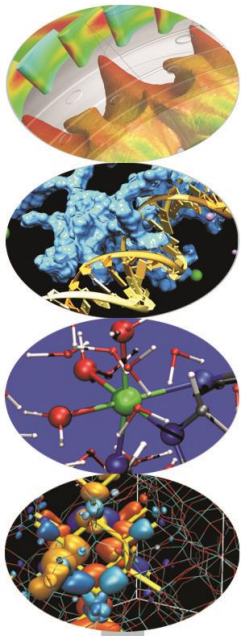
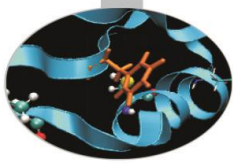


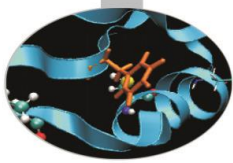
Sintassi I Parte





Indice

- Ciao Mondo! (in C e C++)
- Tipi di dato
- Variabili e costanti
- Operatori aritmetici e sui bit
- Espressioni miste
- Conversione di tipo
- L'operatore condizionale ternario
- Precedenza ed associatività degli operatori
- Input ed output da device standard



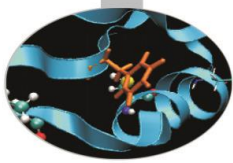
Ciao Mondo! (in C)

- Introduciamo i concetti di base del C e del C++ mettendo a confronto le rispettive versioni di un semplice programma:

```
/* file ciao_mondo.c */
#include <stdio.h>

int main(){
        printf("Ciao Mondo! \n");
   return 0;
}
```

- Il programma produce la stampa su video del messaggio:

```
Ciao Mondo!
```



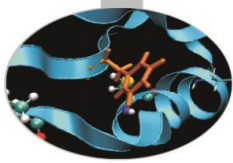
Ciao Mondo! (in C++)

```
//file ciaoMondo.cpp

#include <iostream>

using namespace std;

int main(){
        cout << "Ciao Mondo!" << endl;
        return 0;
}
```

Ancora una volta, l'output generato dal programma è il seguente:

```
Ciao Mondo!
```

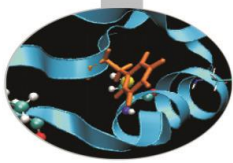
Ciao Mondo! (in C)



- Le istruzioni precedute dal simbolo **#** sono rivolte ad un componente software chiamato preprocessore, che interviene sul programma prima della compilazione. L'istruzione ***#include***, in particolare, consente di inserire un file in quello da compilare.

- Dopo la specifica dello spazio dei nomi da utilizzare, incontriamo il cuore del programma, ovvero la funzione ***main( )***, che contiene le istruzioni principali del codice. Da qui è anche possibile chiamare altre funzioni, definite eventualmente in file differenti.

Ciao Mondo! (in C)



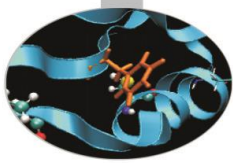
- Il corpo della funzione è compreso tra le due graffe: costituisce, cioè, un cosiddetto *blocco* ovvero una collezione di *statement.*
- Per *statement* intendiamo ogni porzione di codice che termina con un **;** vedi, ad esempio:

```
printf("Ciao Mondo!");
```

e

```
cout << "Ciao Mondo!" << endl;
```

- Lo *statement* è, di fatto, l'unità fondamentale di un codice in C/C++.
- A differenza degli statement, i blocchi non devono mai terminare con il **;** , salvo nella definizione di una *classe* o di una *struct.*

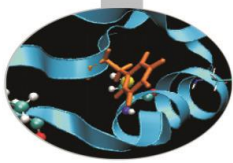


I commenti

Per commentare una o più linee di codice si adottano in C i simboli di apertura **/*** e chiusura ***/** di commento:

```
/* file circonferenza.c */
double circonferenza(int raggio, double
due_pi){
   /*const double pi_greco=3.14;
     if(raggio <=0) exit;
     double circ=raggio*2*pi;
     Attenzione: parte vecchia del
programma,
     non piu' necessaria */
            ...
}
```

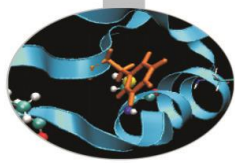
I commenti



In C++ è inoltre possibile inserire un commento su una linea utilizzando il simbolo **//**:

```
const double pi=3.14159; // valore del pi greco
```

Per commentare più linee di codice è invece più comodo usare la sintassi: **/* …*/** ereditata dal C.

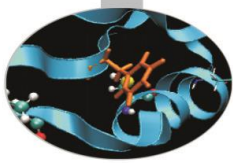


I tipi

- I *tipi* in C/C++ determinano l'insieme delle operazioni che possono essere eseguite sulle variabili e su ogni altra entità del linguaggio, come le funzioni o gli oggetti (strong type checking).

- Ogni variabile (entità) deve essere associata ad un tipo noto al compilatore.

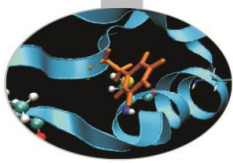
- Esistono tre categorie distinte di tipi:
  - tipi predefiniti o fondamentali;
  - tipi costruiti sui fondamentali (es.: puntatori, array, reference);
  - tipi definiti dall'utente (es.: strutture, classi).



I tipi predefiniti in C

Il C mette a disposizione i seguenti tipi fondamentali:

- char (unsigned, signed);
- int (signed, unsigned, short, long);
- float;
- double (long);
- void.



I tipi predefiniti in C++

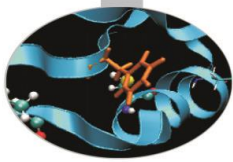
Il C++ mette a disposizione i seguenti tipi fondamentali:

- bool;
- char (unsigned, signed, wchar_t);
- int (signed, unsigned, short, long);
- float;
- double (long);
- void.

Il tipo wchar_t è definito per il supporto delle lingue straniere.

In entrambi i linguaggi usando lo specificatore da solo il tipo int è assunto di default:

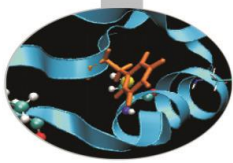
```
unsigned int i;      // int è esplicitato
unsigned i;          // int implicitamente assunto
```



Dimensione dei tipi

- La dimensione dei tipi è espressa in byte ed il suo valore è accessibile tramite l'operatore ***sizeof()***.
- In generale abbiamo che:

TIPO	***DIMENSIONE***
char	1 byte
bool	1 byte
wchar_t	2 byte
short	2 byte
int	4 byte
float	4 byte
long	8 byte
double	8 byte
long double	16 byte



La dimensione di un char è uguale ad 1 byte ed è presa come unità di misura.

Valgono, inoltre, le seguenti regole tra tipi “compatibili”:

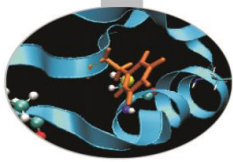
sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long);

sizeof(char) <= sizeof(wchar_t) <= sizeof(long);

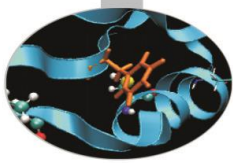
sizeof(bool) <= sizeof(long);

sizeof(float) <= sizeof(double) <= sizeof(long double).

Nota: dichiarazioni, definizioni & Co



- Dichiarazione: associa una variabile ad un tipo di dato
- Definizione: indica come una variabile viene costruita
- Istanza:momento di costituzione della variabile; per le variabili di tipo built-in questo coincide con la definizione
- Inizializzazione: definizione + valore associato
- Assegnamento: nuovo valore per una variabile già definita



Variabili e costanti

- La dichiarazione ed eventuale inizializzazione di una variabile richiede la seguente notazione:

tipo nome_variabile1(=valore1), nome_variabile2(=valore2);

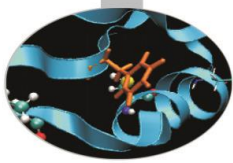
```
int x,y;                    // dichiarazione
double z=4.5;               // inizializzazione
```

- Per inizializzare una costante è necessario usare l'istruzione *const*:

```
const double pi_greco=3.14159;
```

in questo modo il valore di pi_greco non potrà più essere modificato nel resto del programma.

Variabili e costanti

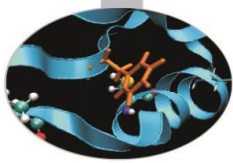


- Le dichiarazioni C++ sono dichiarazioni C, con una grande differenza formale: le dichiarazioni C++ sono istruzioni.

- Essendo istruzioni possono essere inserite in qualsivoglia punto del codice

- Questo non è possibile in C in cui vige l'obbligo di inserire tutte le dichiarazioni di variabile in uno stesso blocco di codice prima della prima "vera" istruzione.

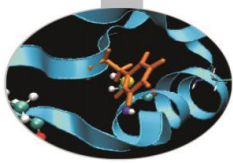
- Ne risulta che scrivendo codice C++ è possibile, e consigliato, dichiarare una variabile quando si sta per farne effettivamente uso nel codice.



Gli operatori aritmetici

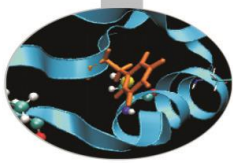
Operatore binario: Espressione:	**Operazione:**	
+	addizione	x + y
-	sottrazione	x - y
*	moltiplicazione	x * y
/	divisione	x / y
%	modulo	x % y

N.B.: l'operazione di modulo restituisce il resto della divisione.



Gli operatori aritmetici

Operatore unario:	**Operazione:**	**Espressione:**
++ (prefisso)	incremento di 1	++x
++ (postfisso)	incremento di 1	x++
-- (prefisso)	decremento di 1	--x
-- (postfisso)	decremento di 1	x--
+	segno positivo	z = + x / y
-	cambio di segno	z = - x / y



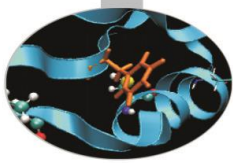
Gli operatori aritmetici

esempio:

```
int x, b = 3;

x = ++b;          /* inizialmente b viene incrementato
                     a 4 e quindi ad x è assegnato il
                     valore di b, 4 */
```

se invece avessimo:

```
x = b++;          /* inizialmente ad x viene assegnato
                     il valore di b, cioè 3, quindi b è
                     incrementato a 4 */
```

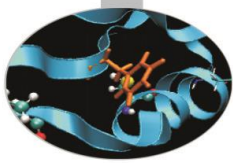


Gli operatori relazionali

Gli operatori relazionale e logici in C++ ritornano valori bool, mentre in C, non esistendo i bool vengono restituiti interi con la seguente convenzione: qualunque valore non-nullo è considerato vero, lo zero è considerato falso.

Operatore binario:	**Relazione:**	**Espressione:**
<	minore di	x < y
<=	minore o uguale a	x <= y
>	maggiore di	x > y
>=	maggiore o uguale a	x >= y
==	uguale a	x == y
!=	diverso da	x != y

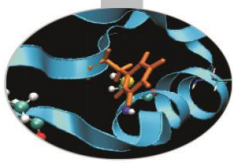
Gli operatori logici



Operatore binario:	Operazione:	Espressione:
&&	AND	(x>=2) && (x<5)
||	OR	(x < 3) || (x > 7)

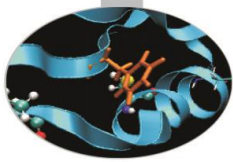
Operatore unario:	Operazione:	Espressione:
!	NOT	!(x > 5)

Gli operatori di assegnamento

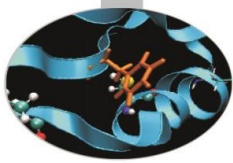


Operatore:	Relazione:	Espressione:	Significato:
=	assegnamento	x = y	
+=	incremento	x += y	x = x + y
-=	decremento	x -= y	x = x -y
*=	moltiplicazione	x *= y	x = x * y
/=	divisione	x /= y	x = x / y
%=	modulo	x %= y	x = x % y

Gli operatori di assegnamento



- Un'espressione di assegnamento del tipo *x = 6* associa ad una precisa locazione di memoria occupata dalla variabile x (left value) un determinato valore (right value), in questo caso il numero 6. Alla luce di quanto detto è chiaro che un'espressione come *3 = y* è del tutto priva di significato.



Espressioni miste

Un'espressione può contenere entità di differenti tipi: in questo caso il compilatore cerca di ricondurle tutte al medesimo tipo che sarà il più grande possibile fra quelli presenti, al fine di limitare la perdita di informazioni.

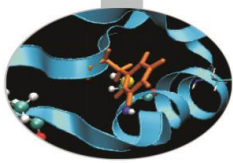
Esempio:

```
double x = 2.04;

double y = x * 5;   /* 5 è una costante intera che viene promossa
                       a 5.0 cioè a costante double, y assume il
          valore 10.2 */
```

attenzione:

```
double x = 2.04;

int y = x * 5;     /*  5 viene ancora promossa a 5.0 per essere moltiplicata per la
                     variabile double x, ma y può assumere solo valori interi,
                      dunque il valore di y è troncato a 10 */
```

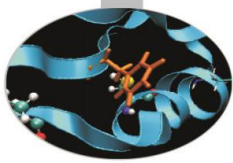
Conversione automatica di tipo



Le conversioni effettuate dal compilatore senza perdita di informazioni possono essere riassunte nella seguente sequenza:

`char` → `short` → `int` → `long` → `float` → `double`.

È inoltre possibile la conversione da int a bool: ogni valore intero (anche negativo) diverso da zero viene convertito in *true*; lo zero in *false*.



Conversione forzata di tipo

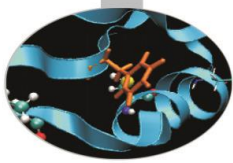
In C e C++ è possibile effettuare la conversione forzata di un tipo di dato tramite un cast:

```
double d;
d = (double)10 / 3; /*   converte 10 in double prima
                         dell'operazione, che viene
                         eseguita in precisione doppia */
d = (double) (10/3); /*  il risultato dell'operazione
                         è un intero, convertito in precisione
                         doppia prima dell'assegnazione */
```

in C++è possibile forzare la conversione del tipo di un'entità anche attraverso l'operatore ***static_cast<type>()***:

```
double d;
d = static_cast<double>(10)/3;  // d assume il valore 3.333
```

L'operatore condizionale ternario



L'operatore ternario ***?:*** permette di valutare un'espressione condizionale del tipo:

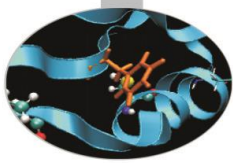
operando1 **?** *operando2* **:** *operando3*

dove il primo operando rappresenta la condizione ed il secondo ed il terzo operando sono i valori assunti dall'espressione se la condizione risulta vera o falsa rispettivamente.

Esempio:

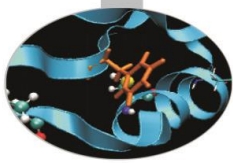
```
int x = 2, sign;

sign = (x < 0) ? -1 : 1;  // sign assume il valore 1
```

Operatori: precedenza ed associatività

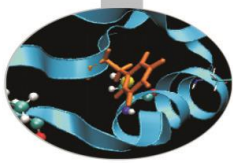


Operatori	**Associatività**	**Tipo**
()	*sx verso dx*	parentesi
++ -- + - ! static_cast<*type*>()	*dx verso sx*	unario
* / %	*sx verso dx*	moltiplicativo
+ -	*sx verso dx*	additivo
<< >>	*sx verso dx*	inserzione/estrazione
< <= > >=	*sx verso dx*	relazionale

Operatori: precedenza ed associatività



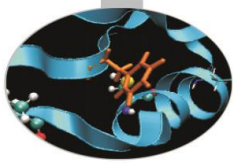
Operatori	Associatività	Tipo
== !=	sx verso dx	uguaglianza
&&	sx verso dx	AND
||	sx verso dx	OR
?:	dx verso sx	condizionale
= += -= *= /= %=	dx verso sx	assegnamento
,	sx verso dx	virgola



Gli operatori a livello di bit

Operatore binario:	Operazione:	Espressione:
&	AND	x & y
|	OR	x | y
^	XOR	x ^ y
<<	Left Shift	x << 2
>>	Right Shift	x >> 3

Operatore unario:	Operazione:	Espressione:
~	complemento a 1	~x

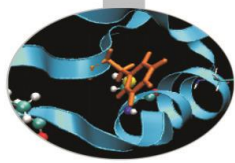


Gli operatori a livello di bit

Gli operatori a livello di bit ammettono, come operandi, ogni tipo di variabile intera (int, short, long, signed ed unsigned), insieme con i char. La loro azione si esplicita nella manipolazione diretta delle sequenze di bit in cui possono essere convertiti gli operandi stessi.

- L'operazione XOR restituisce 0 se i due operandi sono entrambi uguali ad 1 o a 0; 1, invece, se sono diversi tra loro.

- Il complemento ad uno è conosciuto anche come negazione (NOT).



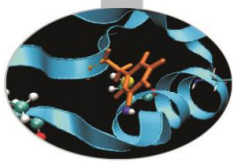
Gli operatori a livello di bit

Consideriamo due interi x ed y codificati con 8 bit, il cui bit più significativo (il più a sinistra) rappresenta il segno del numero. Se esso è uguale ad uno il numero è negativo.

```
x=5 ovvero x=00000101;

y=9 ovvero y=00001001

~x=11111010 ovvero ~x= -6  (ottenuto sommando 1 a x)

x & y  = 00000001 = 1

x | y  = 00001101 = 11

x ^ y  = 00001100 = 10

x << 2 = 00010100 = 20

y >> 2 = 00000010 = 2
```

calcolo della forma binaria di –x con la regola del complemento a due:

```
−x = ~x + 1 = 11111010 + 1 = 11111011 = -5
```

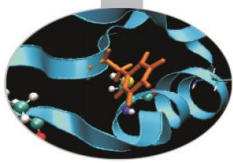


Gli operatori a livello di bit

Gli operatori binari a livello di bit possono essere combinati con l'operatore di assegnamento: &=, |=, ^=, <<= e >>= .

Le regole di precedenza ed associatività fra gli operatori a livello di bit sono le seguenti:

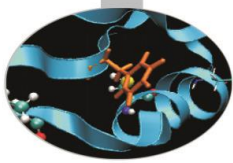
Operatori	**Associatività**	**Tipo**
~	dx verso sx	unario
<< >>	sx verso dx	binario
&	sx verso dx	binario
^	sx verso dx	binario
|	sx verso dx	binario
&=, ^=, |=	sx verso dx	binario



Input Output

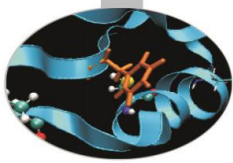
- L'input da tastiera e l'output su video sono gestiti in C da funzioni che richiedono di specificare, tutte le volte che vengono invocate, il formato degli argomenti da leggere o scrivere. In particolare, ***printf()*** e ***scanf()*** sono le due funzioni maggiormente utilizzate rispettivamente per le operazioni di output ed input. Il loro utilizzo richiede l'inclusione, all'interno del programma, della libreria ***stdio.h*** .
- Sintassi:

```
printf("<stringa_di_commento>,%<formato_var>",nome_var);
scanf("%<formato_var>", &nome_var);
```



Stati di formattazione in C

Formato	Tipo	Esempio
c	char	a
d or i	digit (int)	20
e	notazione scientifica	20e2
E	notazione scientifica	20E2
f	float	20.26
g	il più compatto tra e ed f	2000
G	il più compatto tra E ed f	2E10
o	octal	712
s	string	ciao
u	unsigned int	20
x	unsigned hexadecimal	7fa
X	unsigned hexadecimal	7FA
p	address	0x7aff0918



Stati di formattazione in C

Tra **%** e <formato_var> si possono inoltre inserire:

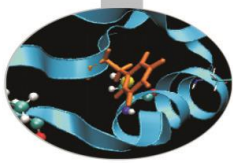
- **(segno -)**: giustifica a sinistra;

- **(segno +):** impone la scrittura del segno, positivo o negativo, di una variabile o costante numerica;

- **m.d** : m = numero minimo di cifre della parte intera; d = precisione, ovvero massimo numero di cifre della parte decimale;

- **#**: con i formati o, x, X impone rispettivamente 0, 0x, 0X davanti al valore numerico.

Stati di formattazione in C

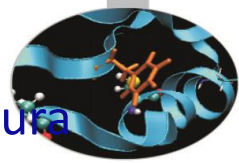


- esempio:

```
printf("%2.3f",95.23472);   //output: 95.235
printf("%#o", 100);         //output: 0144
```

- All'interno della <stringa_di_commento> possono apparire dei caratteri speciali. Fra i più usati ricordiamo:
  - **\n**, newline
  - **\t**, tab
  - **\b**, backspace
  - **\r**, carriage return.

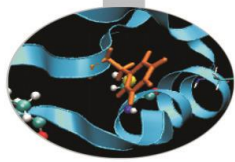
Esempio



Esempio: lettura di variabili di diverso tipo da standard input e loro scrittura su standard output.

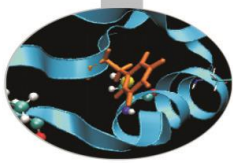
```
#include<stdio.h>
int main(){
   char var_c;
   int var_i;
   float var_f;
   printf("Inserisci un char: ");
   scanf("%c",&var_c);
   printf("Inserisci un intero: ");
   scanf("%d",&var_i);
   printf("Inserisci un float: ");
   scanf("%f",&var_f);
   printf("Hai inserito: %c, %d, %2.2f \n",
                var_c,var_i,var_f);
   return 0;
}
```

Esempio



output:

```
Inserisci un char: a
Inserisci un intero: 23
Inserisci un float: 432.729875
Hai inserito: a, 23, 432.73
```



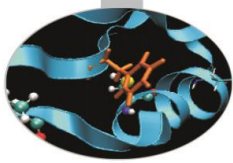
Note

L'uso di scanf() per la lettura di un char può generare dei problemi a causa della bufferizzazione del carattere di a capo, corrispondente alla pressione del tasto return al termine di una precedente operazione di input. La funzione fflush() della libreria stdio.h permette di svuotare il buffer, ma non rappresenta una soluzione affidabile poiché il suo funzionamento dipende dal compilatore:

```
#include<stdio.h>
int main(){
   int var_i; char var_c;
   printf("Inserisci un intero: ");
   scanf("%d",&var_i);
   fflush(stdin);
   printf("Inserisci un char: ");
   scanf("%c",&var_c);
   printf("\n");
   printf("Hai inserito: %d, %c. \n",var_i,var_c);
   return 0;}
```

- output:

```
Inserisci un intero: 28
Inserisci un char:
Hai inserito: 28,
.
```



Note

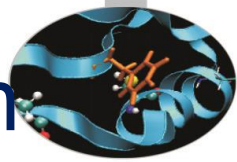
E' preferibile far ricorso alla funzione getchar(), presente anch'essa all'interno di stdio.h, che semplicemente legge un char per volta:

```
#include<stdio.h>
int main(){
   int var_i;
   char var_c;
   printf("Inserisci un intero: ");
   scanf("%d",&var_i);
   printf("Inserisci un char: ");
   getchar(); /* oppure while(getchar() != '\n'); */
   var_c=getchar();
   printf("Hai inserito: %d, %c. \n",var_i,var_c);
   return 0;
}
```

output:

```
Inserisci un intero: 24
Inserisci un char: d
Hai inserito: 24, d.
```

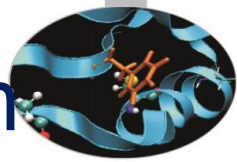
Inserimento ed estrazione dallo stream



Gli operatori di inserimento e estrazione dallo stream

- Per eseguire istruzioni di input/output analoghe a quelle viste finora, il C++ mette a disposizione gli operatori di estrazione ***>>*** ed inserimento ***<<*** nello stream (con stream si intende la sequenza di byte prodotta o ricevuta da un programma). Questi operatori vanno usati, rispettivamente, insieme agli oggetti ***cin*** e ***cout*** e perciò richiedono l'inclusione della libreria ***iostream***.

Inserimento ed estrazione dallo stream



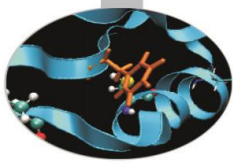
Esempio:

```
int age;
cout << "Insert your age"; // scrittura su standard
                           //   output
cin >> age;          // lettura da standard input
cout << "Your age is" << age << endl;
```

Esiste anche l'oggetto ***cerr*** che può essere usato, ad es., per la scrittura su standard error:

```
cerr  << "Error!" ;   //  scrive su standard error
```

Esempio



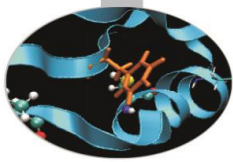
Lettura da standard input e scrittura su standard output di più variabili di tipi differenti

```
#include<iostream>
using namespace std;
int main(){
   int var_i; char var_c; double var_d;
   cout << "Inserisci un intero: ";
   cin >> var_i;
   cout << "Inserisci un double: ";
   cin >> var_d;
   cout << "Inserisci un char: ";
   cin >> var_c;
   cout << "Hai inserito: " << var_i << ", " << var_d
        << ", " << var_c << "." << endl;
   return 0;}
```

output:

```
Inserisci un intero: 21
Inserisci un double: 42.432
Inserisci un char: m
Hai inserito: 21, 42.432, m.
```

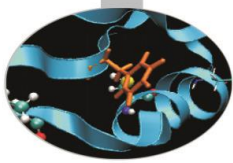
Stati di formattazione in C++



- Gli operatori di immissione ed estrazione (cin >>, cout <<) sono molto comodi ed immediati per gestire l'I/O.

- Esistono però casi in cui il formato col quale si devono scrivere i dati è centrale (ad esempio dovendo interagire con altri codici che si aspettano determinati formati).

- In questi casi dobbiamo usare le funzioni che permettono di definire, per il dato trattato, la dimensione occupata in termini di caratteri (larghezza), il formato (esadecimale, esponenziale, ecc.). Queste funzioni vengono dette *manipolatori* e sono contenuti nella libreria iomanip.h.



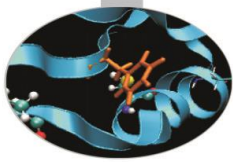
I manipolatori del C++

- **dec**, **hex**, **oct:** visualizzano una cifra in base decimale, esadecimale ed ottale rispettivamente.

```
int num = 14;
cout << num << “esadecimale:” << hex << num << “\n”
     << dec << num << “ottale” << oct << num << endl;
```

- **setbase(int b):** permette di settare la base di rappresentazione. Poiché riceve un argomento (che può assumere soltanto i valori 8, 10 e 16), è detto manipolatore *parametrizzato*.

```
cout << num << “in ottale:” << setbase(8) << num
          << endl;
```



I manipolatori del C++

- **setprecision(int width):** permette di definire con **quanti decimali** vanno rappresentati i numeri in virgola mobile:

  double dbl=static_cast<double>(11)/3;

  for(int i=0; i<10; i++)

  cout << setprecision(i) << dbl;

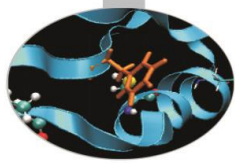
- **setw(int w):** definisce l'ampiezza del campo, ovvero il numero minimo di caratteri con cui viene scritto o letto un dato:

  cout << setw(5) << dbl << endl;

- **setfill(int c):** permette di definire quale carattere utilizzare per riempire il campo, se il numero è troppo piccolo per usare tutto lo spazio che ha a disposizione:

  cout << setw(10) << setfill('*') << num << endl;

Esempio



- esempio: uso di alcuni manipolatori di stream

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


int main(){
    int num = 32;
    cout << num << " hex: " << hex << num << "\n"
         << setbase(10) << num << " oct: " << oct
          << num << endl;
    cout << setw(10) << num << endl;
    cout << setw(10) << setfill('*') << num << endl;
    return 0;
}
```

- output:

```
32 hex: 20
32 oct: 40
        40
********40
```