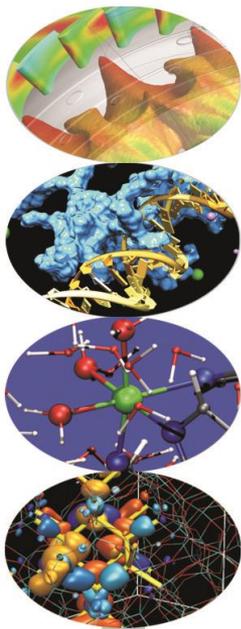
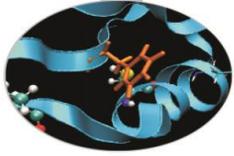


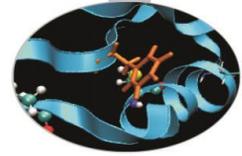
# Sintassi I Parte



# Indice



- **Ciao Mondo!**
- **Tipi di dato**
- **Limiti**
- **Variabili e costanti**
- **Operatori**
- **Espressioni miste**
- **Conversione di tipo**
- **L'operatore condizionale ternario**
- **Input ed output da standard device**



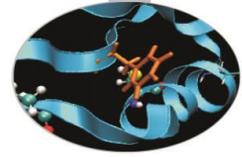
# Ciao Mondo! (in C)

- Introduciamo i concetti di base del C attraverso un semplice programma:

```
/* file ciao_mondo.c */  
#include <stdio.h>  
  
int main() {  
    printf("Ciao Mondo! \n");  
    return 0;  
}
```

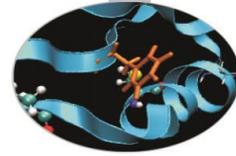
- Il programma produce la stampa su video del messaggio:

Ciao Mondo!



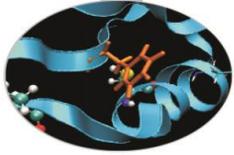
# Ciao Mondo! (in C)

- 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.
- Molte funzionalità sono disponibili tramite una libreria di funzioni il cui contenuto è definito dallo Standard del linguaggio.
- Per utilizzare tali funzioni il compilatore ha necessità di conoscerne il prototipo, dove sono definiti gli argomenti ed il tipo ritornato.



# Ciao Mondo! (in C)

- Le informazioni sulle funzioni sono contenute negli header files.
- Gli header files sono raggruppati per categorie, devono essere inseriti nel codice sorgente prima di utilizzare una particolare funzione la cui definizione è contenuta al loro interno.
- Quindi 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.



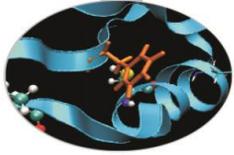
# Ciao Mondo! (in C)

- 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!");
```

Lo *statement* è, di fatto, l'unità fondamentale di un codice in C/C++.

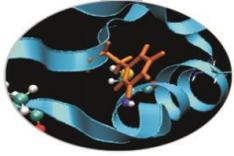
- A differenza degli *statement*, i blocchi non devono mai terminare con il `;`, salvo nella definizione di una *struct*.



# 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 */  
    ...  
}
```

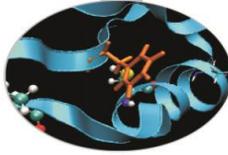


# I commenti

A partire dallo standard c99 è inoltre possibile inserire un commento su una linea utilizzando il simbolo `//`:

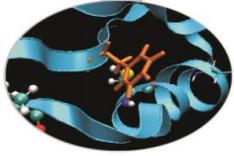
```
const double pi=3.14159; // valore del pi greco
```

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 (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).



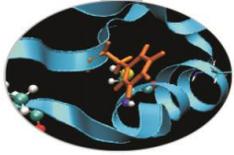
# I tipi predefiniti in C

Il C mette a disposizione i seguenti tipi fondamentali:

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

Usando lo specificatore da solo il tipo `int` è assunto di default:

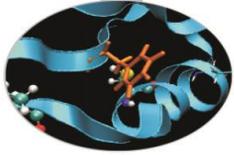
- `unsigned int i; // int è esplicitato`
- `unsigned i; // int implicitamente assunto`



# I tipi predefiniti in C

Con il C99 è stato introdotto il tipo `long long`, il tipo `complex` ed il tipo `bool`.

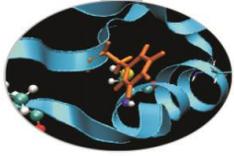
- Occorre includere l'header **`complex.h`**
- Si possono estrarre parte reale ed immaginaria con le funzioni **`creal(complex)`** e **`cimag(complex)`**
- Per utilizzare il tipo `bool` occorre includere l'header **`stdbool.h`**



# 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:

<u>TIPO</u>	<u>DIMENSIONE</u>
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.

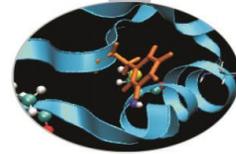
Valgono, inoltre, le seguenti regole tra tipi “compatibili”:

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

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

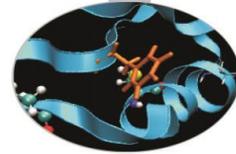
`sizeof(bool) <= sizeof(long);`

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



# Limiti numerici <limits.h>

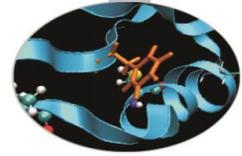
Name	Meaning	Value
<b>CHAR_BIT</b>	width of any char type	$\geq 8$
<b>SCHAR_MIN</b>	minimum value of signed char	$\leq -127$
<b>SCHAR_MAX</b>	maximum value of signed char	$\geq 127$
<b>UCHAR_MAX</b>	maximum value of unsigned char type	$\geq 255$
<b>SHRT_MIN</b>	minimum value of short	$\leq -32767$
<b>SHRT_MAX</b>	maximum value of short	$\geq 32767$
<b>USHRT_MAX</b>	maximum value of unsigned short	$\geq 65535$
<b>INT_MIN</b>	minimum value of int	$\leq -32767$
<b>INT_MAX</b>	maximum value of int	$\geq 32767$
<b>UINT_MAX</b>	maximum value of unsigned	$\geq 65535$
<b>LONG_MIN</b>	minimum value of long	$\leq -2147483647$
<b>LONG_MAX</b>	maximum value of long	$\geq 2147483647$
<b>ULONG_MAX</b>	maximum value of unsigned long	$\geq 4294967295$
<b>LLONG_MIN</b>	minimum value of long long	$\leq -9223372036854775807$
<b>LLONG_MAX</b>	maximum value of long long	$\geq 9223372036854775807$
<b>ULLONG_MAX</b>	maximum value of unsigned long long	$\geq 18446744073709551615$



# Limiti numerici <float.h>

Name	Meaning	Value
FLT_EPSILON	$\min\{x 1.0 + x > 1.0\}$ in <b>float</b> type	$\leq 10^{-6}$
DBL_EPSILON	$\min\{x 1.0 + x > 1.0\}$ in <b>double</b> type	$\leq 10^{-9}$
LDBL_EPSILON	$\min\{x 1.0 + x > 1.0\}$ in <b>long double</b> type	$\leq 10^{-9}$
FLT_DIG	decimal digits of precision in <b>float</b> type	$\geq 6$
DBL_DIG	decimal digits of precision in <b>double</b> type	$\geq 10$
LDBL_DIG	decimal digits of precision in <b>long double</b> type	$\geq 10$
FLT_MIN	minimum normalized <b>positive</b> number in <b>float</b> range	$\leq 10^{-37}$
DBL_MIN	minimum normalized <b>positive</b> number in <b>long</b> range	$\leq 10^{-37}$
LDBL_MIN	minimum normalized <b>positive</b> number in <b>long double</b> range	$\leq 10^{-37}$
FLT_MAX	maximum finite number in <b>float</b> range	$\geq 10^{37}$
DBL_MAX	maximum finite number in <b>long</b> range	$\geq 10^{37}$
LDBL_MAX	maximum finite number in <b>long double</b> range	$\geq 10^{37}$
FLT_MIN_10_EXP	minimum $x$ such that $10^x$ is in <b>float</b> range and normalized	$\leq -37$
DBL_MIN_10_EXP	minimum $x$ such that $10^x$ is in <b>double</b> range and normalized	$\leq -37$
LDBL_MIN_10_EXP	minimum $x$ such that $10^x$ is in <b>long double</b> range and normalized	$\leq -37$
FLT_MAX_10_EXP	maximum $x$ such that $10^x$ is in <b>float</b> range and finite	$\geq 37$
DBL_MAX_10_EXP	maximum $x$ such that $10^x$ is in <b>double</b> range and finite	$\geq 37$
LDBL_MAX_10_EXP	maximum $x$ such that $10^x$ is in <b>long double</b> range and finite	$\geq 37$

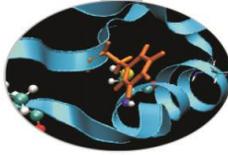




# Limiti numerici

```
printf("Floating-point types: \n");
printf("Type      \tSize \tMinimum \tMaximum\n");
printf("float      \t%-5d\t%-G\t%-G\n", sizeof(float), FLT_MIN, FLT_MAX);
printf("double     \t%-5d\t%-1G\t%-1G\n", sizeof(double), DBL_MIN, DBL_MAX);
printf("long double\t%-5d\t%-11G\t%-11G\n", sizeof(long double), LDBL_MIN,
                                             LDBL_MAX);

printf("\n");
printf("Type          \tSignificant digits\n");
printf("float          \t%d\n", FLT_DIG);
printf("double         \t%d\n", DBL_DIG);
printf("long double    \t%d\n", LDBL_DIG);
printf("\n");
}
```



# Gestione Inf - NaN

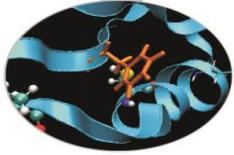
A partire dal c99 il linguaggio mette a disposizione le macro "NaN" e "INFINITY" ed alcune funzioni di utilità per l'individuazione e la gestione delle eccezioni numeriche:

- **isfinite()** Ritorna true se il valore dell'argomento della funzione è finito
- **isinf()** Ritorna true se il valore dell'argomento della funzione è infinito
- **isnan()** Ritorna true se il valore dell'argomento della funzione è un NaN

```
#include <stdio.h>
#include <limits.h>
#include <math.h>
int main(int argc, char *argv[]){
float a=12.0;
    if (isfinite(a/10)) printf ("Finite value \n");
    if (isinf(a/0)) printf ("Inf \n");
    if (isnan(sqrt(-a))) printf ("NaN \n");
return 0; }
```

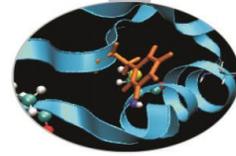
• output:

```
Finite value
Inf
NaN.
```



# 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:

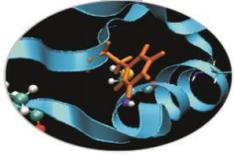
*tipo nome\_variabile1(=valore1), nome\_variabile2(=valore2);*

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

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

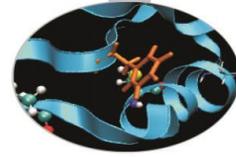
```
const double pi_greco=3.14159;
```

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



# Variabili e costanti

- Le variabili devono avere nomi o identificativi validi, ovvero che rispettano le regole previste dallo standard del linguaggio quali:
  - I caratteri ammessi sono: **a-z, A-Z, 0-9, \_**
  - Il primo carattere non può essere un numero
  - (per esempio **x1 è ammesso, 1x no**)
  - L'estensione massima di una variabile non dovrebbe superare i 31 caratteri
- Il linguaggio è case-sensitive: **var1 non è uguale a VAr1 !**
- Una convenzione comune: evitare di dichiarare variabili scritte solo con lettere maiuscole



# Gli operatori aritmetici

**Operatore binario:**  
**Espressione:**

**Operazione:**

+

addizione

$x + y$

-

sottrazione

$x - y$

\*

moltiplicazione

$x * y$

/

divisione

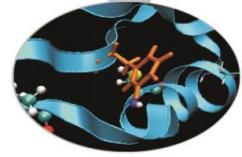
$x / y$

%

modulo

$x \% y$

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



# Gli operatori aritmetici

## Operatore unario:

++ (prefisso)

++ (postfisso)

-- (prefisso)

-- (postfisso)

+

-

## Operazione:

incremento di 1

incremento di 1

decremento di 1

decremento di 1

segno positivo

cambio di segno

## Espressione:

++x

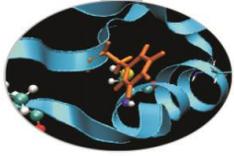
x++

--x

x--

$z = + x / y$

$z = - x / y$



# Gli operatori aritmetici

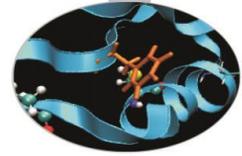
esempio:

```
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:

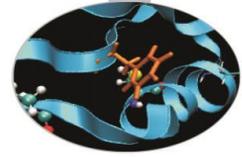
```
x = b++;          /* inizialmente ad x viene assegnato  
                  il valore di b, cioè 3, quindi b è  
                  incrementato a 4 */
```



# Gli operatori relazionali

Gli operatori relazionale e logici **ritornano valori bool**. Se si usa lo standard c89 dove il tipo bool non è definito, vengono restituiti interi con la seguente convenzione: qualunque valore non-nullo è considerato vero, lo zero è considerato falso.

<b>Operatore binario:</b>	<b>Relazione:</b>	<b>Espressione:</b>
<	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:

&&

AND

||

OR

## Espressione:

$(x \geq 2) \ \&\& \ (x < 5)$

$(x < 3) \ || \ (x > 7)$

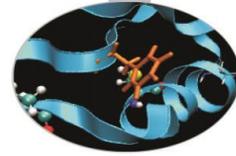
## Operatore unario: Operazione:

!

NOT

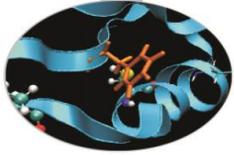
## Espressione:

$!(x > 5)$



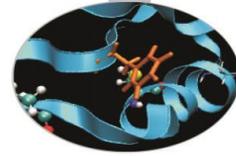
# Gli operatori di assegnamento

<b>Operatore:</b>	<b>Relazione:</b>	<b>Espressione:</b>	<b>Significato:</b>
=	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:

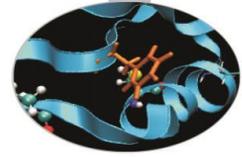
```
double x = 2.04;
```

```
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 */
```

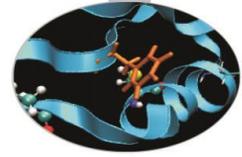


# Conversione automatica di tipo

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:

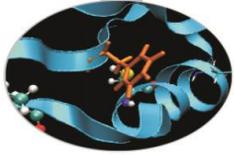
```
double d;
```

```
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 \*/



# L'operatore condizionale ternario

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

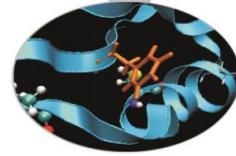
*operando1 ? operando2 : operando3*

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;
```

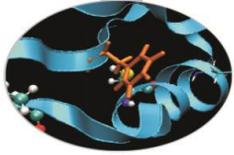
```
sign = (x < 0) ? -1 : 1; // sign assume il valore 1
```



# Gli operatori a livello di bit

<b>Operatore binario:</b>	<b>Operazione:</b>	<b>Espressione:</b>
&	AND	$x \& y$
	OR	$x   y$
^	XOR	$x \wedge y$
<<	Left Shift	$x \ll 2$
>>	Right Shift	$x \gg 3$

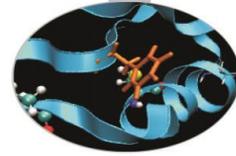
<b>Operatore unario:</b>	<b>Operazione:</b>	<b>Espressione:</b>
~	complemento a 1	$\sim x$



# 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  $y=00001001$

$\sim x=11111010$  ovvero  $\sim x= -6$  (ottenuto invertendo i bit di  $x$ )

$x \& y = 00000001 = 1$

$x | y = 00001101 = 11$

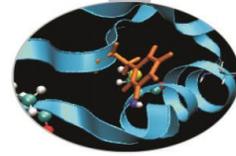
$x \wedge y = 00001100 = 10$

$x \ll 2 = 00010100 = 20$

$y \gg 2 = 00000010 = 2$

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

$-x = \sim x + 1 = 11111010 + 00000001 = 11111011 = -5$

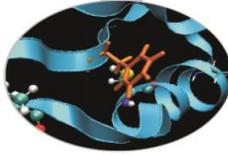


# Gli operatori a livello di bit

Gli operatori binari a livello di bit possono essere combinati con l'operatore di assegnamento:  $\&=$ ,  $|=$ ,  $\wedge=$ ,  $\ll=$  e  $\gg=$ .

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

<b>Operatori</b>	<b>Associatività</b>	<b>Tipo</b>
$\sim$	dx verso sx	unario
$\ll$ $\gg$	sx verso dx	binario
$\&$	sx verso dx	binario
$\wedge$	sx verso dx	binario
$ $	sx verso dx	binario
$\&=$ , $\wedge=$ , $ =$	sx verso dx	binario

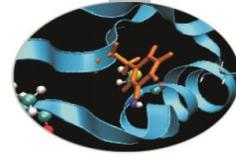


# Input Output

- 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:

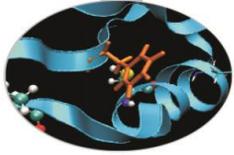
```
printf("<stringa_di_commento>, %<formato_var>", nome_var);
```

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



# Stati di formattazione in C

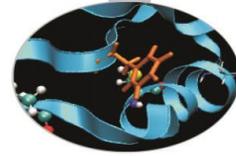
<b>Formato</b>	<b>Tipo</b>	<b>Esempio</b>
c	char	a
d or i	digit (int)	20
e	notazione scientifica	2e+03
E	notazione scientifica	2E+03
f	float	20.26
g	il più compatto tra e ed f	2000
G	il più compatto tra E ed f	2E+10
o	octal	712
s	string	ciao
u	unsigned int	20
x	unsigned hexadecimal	7fa
X	unsigned hexadecimal	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.



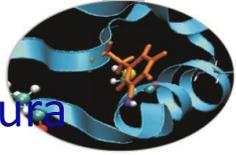
# Stati di formattazione in C

- 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.

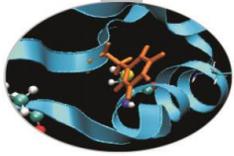
# Esempio



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

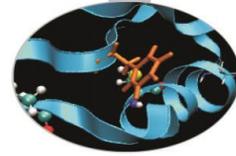
## output:

```
Inserisci un char: a
```

```
Inserisci un intero: 23
```

```
Inserisci un float: 432.729875
```

```
Hai inserito: a, 23, 432.73
```



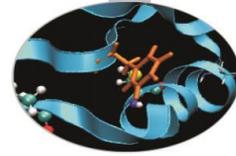
# Note

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,
.
```



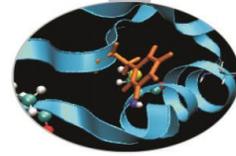
# Note

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();
    var_c=getchar();
    printf("Hai inserito: %d, %c. \n",var_i,var_c);
    return 0;
}
```

## output:

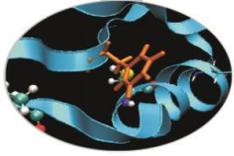
```
Inserisci un intero: 24
Inserisci un char: d
Hai inserito: 24, d.
```



# Input Output in C++

- Per effettuare operazioni di lettura da standard input e scrittura su standard output il C++ mette a disposizione gli oggetti `cin` e `cout`.
- Tali oggetti sono correlati ai corrispettivi operatori di immissione “>>” ed estrazione “<<” dallo stream.
- Al posto del carattere di newline “\n” è disponibile l’oggetto `endl`.
- Con `cin` e `cout` non sono più presenti i problemi di bufferizzazione visti per le funzioni del C poiché il buffer dell’oggetto `cout` viene svuotato (flushed) prima di ogni operazione eseguita con `cin`.

# Esempio



- Esempio: uso di cin, cout, endl.

```
#include<iostream>

using namespace std;

int main(){
    int num;

    cout << " Insert integer number"<< endl;
    cin >> num;
    cout << "The number inserted is:  "<< num << endl;
    return 0;
}
```

**•output:**

```
Insert integer number
23
The number inserted is:  23
```