7. I tipi
Oggetti dello stesso tipo utilizzano lo stesso spazio in memoria e la stessa codifica; sono soggetti alle stesse operazioni, con lo stesso significato.

7.1 Tipi di tipi.
Nel C++ i tipi sono distinti in:

- **i tipi fondamentali**, che servono a rappresentare informazioni semplici. Ad esempio: i numeri interi o i caratteri (int, char, …);
- **i tipi derivati,** permettono di costruire strutture dati complesse. Si costruiscono a partire dai tipi fondamentali, mediante opportuni costruttori (array, puntatori, …) .

Tipi fondamentali:

- i tipi interi: *int, short, long, long long*
- i tipi Booleani: *bool*
- i tipi enumerativi: *enum*
- il tipo carattere: *char*
- i tipi reali: *float, double, long double*

I primi quattro sono detti tipi discreti (hanno un dominio finito).

7.2 Tipi interi (con segno)
I tipi interi (con segno) sono costituiti da numeri interi compresi tra due valori estremi.
Vengono tipicamente codificati in complemento a 2 con 16, 32 o 64 bit.
Appartengono all'intervallo $[-2^{N-1},2^{N-1}-1]$.

Quattro tipi, in ordine crescente di dimensione (che dipendono dalla macchina e dal sistema operativo. Short ha tipicamente massimo 16 bit, long almeno 32):
short [int], int, long [int], long long [int].
short < int < long < long long.

L'operatore **sizeof()** restituisce in byte la dimensione del tipo.
cout << sizeof(int);

! I valori ciclano. Il valore di un'espressione intera non esce dal range prestabilito (con 16 bit è [-32768,32767]) il successore di 32767 è -32768.

7.3 Tipi interi (senza segno)
Il tipo *unsigned* ... rappresenta numeri interi non negativi di varie dimensioni (poco usati, qualche applicazione di sequenze di bit in elettronica).
Range $[0, 2^N-1]$, dove N è il numero di bit.

unsigned int x=1232;
unsigned short int x=567;
unsigned long int x=878678687;

Il valore di un'espressione unsigned non esce mai dal range. Il successivo di 2^N-1 è 0 (il primo valore del range), il successivo è 1.

7.4 Operazioni aritmetiche sugli interi
Operatori binari: + - * / (è la divisione tra interi, quindi 5/2=2) % (5%2=1 ovvero il resto della divisione intera tra 5 e 2).
Operatore unario: - (posto davanti al valore cambia il segno).

Operazioni bit-a-bit (su interi SENZA segno)

Operatore	**Significato**	**Esempio**
<<	x>>n	Shift a destra di n posizioni
>>	x<<n	Shift a sinistra di n posizioni
&	x&n	AND bit a bit tra x e y
|	x|n	OR bit a bit tra x e y
^	x^n	XOR bit a bit tra x e y
~	x~n	NOT, complemento bit a bit

~: restituisce un intero signed anche se l'input è unsigned

Siano x e y rappresentati su 16 bit
x: 0000000000001100 (12)
y: 0000000000001010 (10)

AND, OR e XOR
x|y: 0000000000001110 (14)
x&y: 0000000000001000 (8)
x^y: 0000000000000110 (6)

NOT (al posto di 0 mette 1 e al posto di 1 mette 0)
~x: 1111111111110011 (65523 oppure -13, a seconda che la codifica è a complemento a 2 o segno-valore)
x>>2: 0000000000000011 (3)
x<<2: 0000000000110000 (48)

Tabelle di verità

	1 e 1	1 e 0	0 e 1	0 e 0
AND	1	0	0	0
OR	1	1	1	0
XOR	0	1	1	0

Le operazioni bit-a-bit sono operazioni primitive supportate direttamente dal processore. Vengono usate per concatenare condizioni, ma anche per svolgere operazioni tra interi (come visto sopra) e tra variabili di tipo bool (come vedremo).

8. Operatori di assegnazione.
Sintassi dell'operatore di assegnazione: *exp1 = exp2*

- exp1 deve essere un'espressione dotata di indirizzo (l-value, cioè una variabile.
- exp1 e exp2 devono essere di tipo compatibile

Il valore di exp2 viene valutato e poi assegnato a exp1.

Un'assegnazione può occorrere dentro un'altra espressione. Il valore di un'espressione di assegnazione è il valore di exp2. L'operazione di assegnazione, '=', associa a destra.
Ad esempio: *a=b=c=d=5;* equivale a scrivere *d=5; c=d; b=c; a=b;*

Ad una variabile di tipo *char* si assegna un carattere in questo modo:
char carattere='a'; //si mette tra apici

Ad una variabile di tipo *int* si assegna un numero; può essere scritto anche in notazione binaria o esadecimale.
int num=63; oppure
int num=0b111111; (dopo 0b si inserisce il numero in binario) oppure
int num=0x3f; (dopo 0x si scrive il numero in esadecimale)
Dopo un'istruzione input, l'utente deve però scrivere il numero solo in decimale.

8.1 Operatori misti di assegnazione/aritmetica

Forma compatta	Forma estesa
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y

L'uso della forma compatta è tipicamente più efficiente (su alcune architetture consente di utilizzare in modo ottimale funzionalità della CPU).

8.2 Operatori di pre e post incremento e decremento unitario.
x++ (x=x+1):
incrementa x di un'unità
denota il valore di x **prima** dell'incremento

x-- (x=x-1):
decrementa x di un'unità
denota il valore di x **prima** del decremento

++*x* (x=x+1):
incrementa x di un'unità
denota il valore di x **dopo** l'incremento

--*x* (x=x-1):
decrementa x di un'unità
denota il valore di x **dopo** il decremento

Per fare un calcolo è indifferente usare l'operatore di pre o post, ma se faccio un'assegnazione oppure uso questo operatore in una condizione attenzione a come voglio che sia considerata la mia variabile.

Nel caso di *x++* e *x--* si usano i valori di x prima dell'incremento. Quindi:

```
x=5;
x++; //x=6
y=x++; //x è stata incrementata di 1, quindi vale 7, ma in y è stato assegnato il valore di x prima dell'incremento, quindi y=6

z=--x;  //in questa istruzione prima è stato fatto il decremento, poi è stato assegnato il valore di x in
        z, quindi z vale 6

x=0, y=2, z=3.
x=z++;          //x=3, y=2, z=4
x=++z;          //x=5, y=2, z=5
x=y--;          //x=2, y=1, z=5
x=--z;          //x=4, y=1, z=4
```

! Prestare attenzione !

8.3. Operatori relazionali

Usati per confrontare valori.

Operatore	Significato
==	uguale (per confrontare)
!=	diverso
<=	minore uguale
>=	maggiore uguale
<	minore
>	maggiore

8.4 Ordine di valutazione di un'espressione

In C++ non è specificato l'ordine di valutazione degli operandi di ogni operatore, in particolare:

- l'ordine di valutazione di sottoespressioni in un'espressione
- (l'ordine di valutazione degli argomenti di una funzione)

Ad esempio nel valutare *expr1 <op> expr2*, non è specificato se *expr1* venga valutata prima di *expr2* o viceversa.
(Con alcune importanti eccezioni, vedi operatori Booleani).

Questo è problematico quando sotto-espressioni contengono operatori con "side-effects" (gli operatori di pre e post incremento). Esempio:

```
j = i++ * i++; // undefined behavior
i = ++i + i++; // undefined behavior
```

Evitare l'uso di operatori con side-effects in sotto-espressioni!

9. Il tipo Booleano

Il C++ prevede un tipo Booleano *bool* (mantiene il valore di verità in una variabile). Una variabile di tipo bool è una variabile che può assumere due valori:

- **il valore falso**: rappresentato dalla costante *false* (equivalente a *0*)
- **il valore vero**: è rappresentato dalla costante *true* (equivalente ad un valore intero diverso da 0, generalmente *1*).

Si può usare a tal scopo anche il tipo *int.*

```
bool a=true;
bool b=false;
bool c=-2; //true
int d=0; //false
bool e = 5>6; //siccome 5 non è maggiore di 6, e assumerà il valore di false (0).
```

Se si fa *cout << a << b << c << d;* verrà stampato a video: 1 0 1 0.

9.1 Operatori booleani

Operatori booleani: **!** (not), **&&** (and) e || (or).

Not

```
bool x=true;
bool y=false;
bool z;

z=!x;   //z=false
!!y;    //y=false
```

Tricks

!(x || y) è *(!x && !y)*
!(x && y) è *(!x || !y)*

Priorità

! ha priorità su *&&* e ||.

!x && y equivale a *(!x) && y*.

9.1.1 Tabella di verità degli operatori AND, OR e NOT

! *false=0 true=1*

x	y	!x	x && y	x || y
false	false	true	false	false
false	true	true	false	true
true	false	false	false	true
true	true	false	true	true

9.1.2 Lazy evaluation

In C++ && e || sono valutati in modi “pigro” (lazy evaluation):

(exp1 && exp2): se *exp1* è valutata false, allora *exp2* non viene valutata (perché *false* AND *false* dà *false* e *false* AND *true* dà comunque *false*)

(exp1 || exp2): se *exp1* è valutata *true*, allora *exp2* non viene valutata (perché *true* OR false dà *true* e *true* OR *true* dà comunque *true*)

L'ordine di valutazione tra *exp1* e *exp2* è da sinistra a destra

Evitare di usare costrutti side-effects all'interno di espressioni Booleane.

9.2 Uso degli operatori booleani al posto di if-then o comunque per semplificare i codici
Gli operatori booleani possono essere usati come interi. Infatti è importante ricordarsi che false=0 e true=1.

Ad esempio se devi stampare la tabella di verità di un'implicazione, puoi usare un if e mettere come condizione le due parti dell'implicazione e poi stampare a video il risultato. Oppure è anche possibile studiare l'implicazione e cercare di ricavare una formula.
Se ad esempio abbiamo che P→Q restituisce:

P	**Q**	**P→Q**
T	F	F
T	T	T
F	F	T
F	T	T

Si può osservare che l'implicazione è la seguente: !P || Q. Quindi si può scrivere solo una linea di codice.

Esempio: calcolare il valore assoluto senza la funzione if
valoreassoluto = (a-b)((a>b) - (a<b));*
In questo modo il programma calcolerà a-b e se b<a, lo renderà positivo moltiplicandolo per -1. In questa linea di codice viene fatta, oltre ad un'assegnazione, anche due verifiche (a>b e a<b). Nel caso in cui a>b, la prima condizione risulterebbe vera (cioè 1) e la seconda falsa (cioè 0) e quindi moltiplicheremo la sottrazione iniziale per 1.
Nel caso contrario, la moltiplicheremo per -1 in modo tale da cambiarne il segno.

Esempio: dati due numeri interi, inserirli nella variabile max e min senza usare la funzione if
Dati due numeri interi *a* e *b.* Salviamo il maggiore in *max* e il minore in *min*.
max=a-(a-b)(a<b);*
//se a<b è true, al posto di questa disequazione ci sarebbe 1 e quindi avremo che max=a-a+b, ovvero max=b
min=b-(b-a)(b>a);* //stessa logica per l'assegnazione sopra

Metodo 2:
max=a(a>b)+b*(b>a);* //se a>b, b>a è falsa e quindi equivale a 0, e max avrà valore di a
min=(a+b)-max;

! Alla base di tutto ciò bisogna ricordarsi che una variabile bool può essere inizializzata anche con un'espressione: *bool b=5<4; //b=false*.

N.B.! Nelle operazioni di assegnazione se il compilatore incontra una disequazione, ne verificherà la veridicità per poi assegnarne come valore 1 o 0.

10. I tipi reali

I tipi reali hanno come insieme di valori un sottoinsieme dei numeri reali, ovvero quelli rappresentabili all'interno del computer in un formato prefissato (sono in realtà dei razionali e chiaramente hanno range finito).

Ne esistono vari tipi, in ordine crescente di precisione (che dipende dalla macchina):

- i *float*
- i *double*
- i *long double*

Operatori aritmetici: +,-,*,/. ("/" diverso da divisione tra interi: 7.0/2.0 = 3.5)

Esempi di definizione

```
double a = 2.2;
double b = -14.12e-2;          //notazione esponenziale
double c = .57, d = 6.;        //c=0.57, d=6

float g = -3.4F;
float h = g-.89F;
```

A volte si usa il suffisso 'f' o 'F' dopo un numero per specificare il float.

```
long double i = +0.001;
long double j = 1.23e+12L; //mi stampa a video 1.23e+012 perché il numero è troppo lungo
```

Anche qua a volte si usa il suffiso 'l' o 'L'.

10.1 Precisione dei tipi reali

La rappresentazione dei numeri reali ha dei limiti di precisione, dovuti a:

- limitato numero di bit nella rappresentazione della mantissa (vedi codifica floating-point)
- uso di codifica binaria nei decimali (alcuni numeri non hanno rappresentazione esatta (es: 0.1, 11.1,…);

! Talvolta non visualizzabili con "cout << ..."; e confronto con "... == ..." tra tipi reali spesso problematico.

11. Il tipo enumerato

Un tipo enumerato è un insieme finito di costanti intere, definite dal programmatore, ciascuna individuata da un identificatore.
Sintassi: *enum typeid { id_or_init1, ..., id_or_initn };*

Se non specificato esplicitamente, i valori sono equivalenti rispettivamente agli interi 0, 1, 2, … (i valori vengono stampati come interi).
Ad una variabile di tipo enumerato è possibile assegnare solo un valore del tipo enumerato.

Vengono usati per verificare che una certa variabile abbia un certo valore.

Definizione d'esempio

enum Giorno {LUN,MAR,MER,GIO,VEN,SAB,DOM}; //quelle tra le {} sono costanti che hanno valori 0, 1, 2 fino a 6 che sarà il valore di DOM

Giorno oggi = LUN; //giorno è una variabile enumerata e l'assegnazione deve essere fatta in questo modo
//oggi = 3; //ERRORE! Bisogna assegnare solo valori di tipo enumerato

cout << ***"oggi = "*** << ***oggi*** << ***endl;*** //stampa a video "oggi= 0"

La variabile oggi ha valore 0 e può essere tranquillamente usata nel programma, come nell'esempio che segue.

enum boolean { FALSE, TRUE};
boolean X = TRUE; //variabile enumerata X

cout << ***"X = "*** << ***X*** << ***endl;*** //stampa 1

bool y=true;
cout << ***X || y;*** //stampa 1

In questo esempio diamo noi un valore alle nostre costanti.

enum colore {ROSSO=10, GIALLO=15, BLU=20};
colore sfondo= GIALLO, testo= BLU;

cout << ***"sfondo = "*** << ***sfondo*** << ***endl;***

12. Il tipo carattere
Il tipo *char* ha come insieme di valori i caratteri stampabili (’a’, ’Y’, ’6’, ’+’, ’ ’, …) e generalmente un carattere occupa 1 byte (8 bit).
Il tipo *char* è un sottoinsieme del tipo *int.*

Sintassi: *char var;*
char c=’a’;

Il valore numerico associato ad un carattere è detto codice e dipende dalla codifica utilizzata dal computer (es. ASCII, EBCDIC, BCD, …). La più usata è la codifica ASCII.

12.1 Regole generali di codifica

Il tipo *char* è indipendente dalla particolare codifica adottata!
Un programma deve funzionare sempre nello stesso modo, indipendentemente dalla codifica usata nella macchine su cui è eseguito!!E
Evitare di far riferimento al valore ASCII di un carattere:
char c;
c = 65; /NO
c = ’A’; // SI

cout << (int) c << endl; // stampa 97
cout << c << endl; //stampa A

Su una macchina con una codifica diversa da quella ASCII, stampando *c* è probabile che non venga stampato il carattere ‘A’. Quindi è meglio usare la seconda scrittura di asssegnazione.

Tuttavia qualunque codifica deve soddisfare le seguenti regole.
Precedenza: ’a’ < ’b’ < ... < ’z’
’A’ < ’B’ < ... < ’Z’
’0’ < ’1’ < ... < ’9’
Consecutività: tra lettere minuscole, lettere maiuscole, numeri
’a’, ’b’, ..., ’z’
A’, ’B’, ..., ’Z’
’0’, ’1’, ..., ’9’

Non è fissa la relazione tra maiuscole e minuscole o fra i caratteri non alfabetici. Se in ASCII la distanza tra un carattere minuscolo e il rispettivo maiuscolo è di 22, in altre codifiche potrebbe non essere così.

12.2 Operazioni coi caratteri
È possibile applicare operatori aritmetici agli oggetti di tipo *char*.

char l=’a’;

l += 3; // l diventa ’d’. Infatti il valore di a viene aumentato di tre unità e ciò corrisponde a ‘d’.
l--; // l diventa ’c’

12.2.1 Verificare cos’è un carattere stampandolo a video
Test: l e' una lettera minuscola?
cout << ((l >= 'a') && (l <= 'z')) << endl; //stamperà *1* (cioè) *true* se *l* è una lettera minuscola

Test: l e' una lettera maiuscola?
cout << ((l >= 'A') && (l <= 'Z')) << endl; //stamperà *1* (cioè) *true* se *l* è una lettera maiuscola

Test: l e' una cifra?
cout << ((l >= '0') && (l <= '9')) << endl; //stamperà *1* (cioè) *true* se *l* è un numero

12.2.2 Convertire una lettera da minuscolo a maiuscolo

Da minuscolo a maiuscolo

char l=’c’;

l-=’a’-’A’; // l diventa ’C’; l=l-(‘a’-A’), ovvero l – la distanza tra ‘a’ e ‘A’, in questo modo si converte il carattere in maiuscolo

Da maiuscolo a minuscolo

char l=’c’;

l+=’a’-’A’; // l diventa ’C’; l=l+(‘a’-A’), ovvero l + la distanza tra ‘a’ e ‘A’, in questo modo si converte il carattere in minuscolo

13. L'operatore *sizeof()*
L'operatore *sizeof()*, può avere come argomento: una variabile (*sizeof (x)*), una costante (*sizeof ('a')*) o il nome di un qualsiasi tipo (*sizeof (double)*). Può essere usato in alcuni casi con o senza parentesi.

Restituisce un intero rappresentante la dimensione in byte dell'elemento considerato, che può essere stampato a video o memorizzato in una variabile.

14. Operazioni miste e conversioni di tipo
Spesso si usano operandi di tipo diverso in una stessa espressione o si assegna ad una variabile un valore di tipo diverso della variabile stessa.
Esempio operazioni miste:
int prezzo=2765;
double peso=0.3;
*int costo=prezzo * peso;* //costo=829, se costo fosse stato double costo=829.5

In ogni operazione mista è sempre necessaria una conversione di tipo che può essere implicita o esplicita. In questo caso siccome *costo* una variabile intera si è dovuto troncare il risultato.

14.1 Conversioni implicite
Le conversioni implicite vengono effettuate dal compilatore. Le conversioni implicite più significative sono:
- nella valutazione di espressioni numeriche (gli operandi sono convertiti al tipo di quello di dimensione maggiore)
- nell'assegnazione, un'espressione viene sempre convertita al tipo della variabile

float x = 3; //equivale a: x = 3.0
*int y = 2*3.6;* //equivale a: y = 7

ATTENZIONE
1 / 2 per il compilatore è una divisione fra interi quindi darà come risultato 0.
Affinchè dia 0.5 bisogna scrivere in questo modo: *1.0 / 2.0*.

14.2 Conversioni esplicite
Il programmatore può richiedere una conversione esplicita di un valore da un tipo ad un altro (**casting**).

Esistono due notazioni:
- **prefissa**.
  Esempio: *int i = (int) 3.14;* //in questa riga di codice si definisce *i* (quindi non bisogna definirla prima) e le si fa assumere il valore 3 Se avessimo avuto *float i = (int) 3.14*; //*i* sarebbe stata una variabile di tipo float ma con valore di 3 (abbiamo scritto int vicino a 3.14). diversamente da prima però se ora facciamo *i/2* (cioè 3/2) avremo 1.5 e non 1 perché *i* è float.

- **funzionale**. Esempio: *double f = double(3);* // in questo modo si fa assumere a *f* il valore 3 non considerato come intero, ma come double. Potevamo anche scrivere *double f = 3.;*

14.2.1 Conversioni tra tipi numerici
Promozione: conversione da un tipo ad uno simile più grande.
short → int → long → long long
float → double → long double

Questo modo garantisce di mantenere lo stesso valore.

short int a=42;
cout << sizeof(a); //2
int a= (short) a;
cout << sizeof(a); //4

Conversioni tra tipi compatibili:

- Conversione da tipo reale a tipo intero: il valore viene **troncato** (non arrotondato)
  int x = (int) 4.7; *// x=4*
  int x = (int) -4.7; *//x==-4*
- Conversione da tipo intero a reale: il valore può perdere precisione
  float y = float(2147483600); // 2^31-48
  cout << *y;* // 2^31=2147483648.0

Esempi di conversione
int a=65;
char c = (int) a;
cout << c << (int) c; //Stampa 'A' e a seguire 65

La conversione può essere fatta non solo in fase di definizione di una variabile, ma anche in fase di assegnazione.
double x=10.3;
int y;
y= int (x); oppure *y= (int) x;*

15. L'operatore typeid
typeid (espressione); permette di verificare il tipo di un'espressione.
È necessario prima includere la libreria *#include <typeinfo>*

Esempio:
int a;

cout << "a è del seguente tipo: " << ***typeid(a).name()*** *;*