



Fondamenti di Programmazione - Tommaso Molesti Anno 2020-2021

Consiglio l'uso di questa dispensa a persone che hanno seguito il corso del professore e che hanno già un'idea degli argomenti trattati.

Non mi assumo alcuna responsabilità nel caso alcune informazioni siano incorrette.

Buono studio :)

Rappresentazione dell'informazione

Algoritmo Div & Mod

Basi particolari

Potenze di due

Base otto

Base sedici

Somme di numeri naturali in binario

Rappresentazione dei numeri Z

Modulo e segno

Complemento a 2

Rappresentazione con BIAS

Rappresentazione numeri R

Virgola fissa

Virgola mobile

Proprietà degli operatori C++

Oggetti (variabili) C++

I tipi del C++

Il tipo intero

Il tipo unsigned

Il tipo reale

Il tipo booleano

Operatori di confronto

Il tipo carattere

Il tipo enumerato

Conversioni implicite

Conversioni esplicite

Costanti

Operatore sizeof

Espresioni di assegnamento

Espresioni aritmetiche e logiche

Regola del cortocircuito ("Shortcut rule")

Istruzione if

Operatore condizionale

Operatore ;

Istruzione switch e break

Istruzioni iterative

Istruzione while

Istruzione do

- [Istruzione for](#)
- [Istruzioni di salto](#)
 - [Istruzione break](#)
 - [Istruzione continue](#)
- [Tipi derivati](#)
 - [Riferimenti](#)
 - [Riferimenti costanti](#)
 - [Riferimenti come argomenti](#)
- [Puntatori](#)
- [Variabili di blocco e variabili globali](#)
 - [Variabili di blocco](#)
 - [Variabili globali](#)
- [Concetto di funzione](#)
 - [Istruzione return](#)
 - [Overloading di funzioni](#)
 - [Parametri default](#)
- [Strutture](#)
 - [Strutture dentro strutture](#)
- [Tipi e oggetti array](#)
 - [Array e puntatori](#)
 - [Array come argomenti di funzioni](#)
 - [Array statici](#)
 - [Array automatici](#)
 - [Array dinamici](#)
 - [Memoria Heap](#)
 - [Funzioni su vettori dinamici](#)
 - [dealloc\(\)](#)
 - [alloc\(\)](#)
 - [aggiungiNuovoElementoInFondo\(\)](#)
- [Stringhe](#)
 - [myStrlen\(\)](#)
 - [myStrcpy\(\)](#)
 - [myStrcmp\(\)](#)
 - [Libreria cstring](#)
- [Ordinamento di vettori](#)
 - [Selection sort](#)
 - [Bubble sort](#)
 - [Bubble sort ottimizzato](#)
- [Ricerca in un vettore](#)
 - [Ricerca lineare](#)
 - [Ricerca binaria](#)
- [Matrici](#)
 - [Passaggio di matrice a funzione](#)
 - [Vettore trattato come matrice](#)
 - [Matrici dinamiche](#)
- [Dichiarazioni typedef](#)
- [Liste](#)
 - [Lista di interi letti da tastiera terminati da !](#)
 - [Stampa lista](#)
 - [Inserimento in testa](#)
 - [Estrazione in testa](#)
 - [Inserimento in coda](#)

[Estrazione dal fondo](#)
[Dealloca lista](#)
[Inserimento \(ordinato\) in una lista ordinata](#)
[Estrazione di un elemento](#)
[Estrazione di un elemento da una lista ordinata](#)
[Liste con un puntatore ausiliario](#)

[Visibilità](#)
[Unità di compilazione](#)
 [Operatore di risoluzione di visibilità \(::\)](#)
 [Spazio dei nomi](#)
 [Collegamento](#)
 [Moduli](#)

[Pila](#)
[Tipi classe](#)
 [Operazioni su oggetti classe](#)
 [Somma di due oggetti](#)
 [Puntatore this](#)
 [Visibilità a livello di classe](#)
 [Modularità e ricompilazione](#)
 [Funzioni globali](#)
 [Costruttori](#)
 [Costruttori default](#)
 [Costruttori per allocare memoria dinamica](#)
 [Costruttori per oggetti dinamici](#)
 [Distruttori](#)
 [Regole di chiamata](#)
 [Costruttori](#)
 [Distruttori](#)
 [Costruttori di copia](#)
 [Costruttore di copia per allocare memoria](#)
 [Funzioni friend](#)
 [Array di oggetti classe](#)

[Overloading di operatori](#)
 [Overloading di operatori di uscita](#)
 [Overloading di operatori di ingresso](#)
 [Overloading di operatori di assegnamento](#)
 [Operatori che si possono ridefinire](#)

[Controlli sugli stream](#)
[Manipolazione dei file](#)
[Espressioni letterali](#)
[Costanti e riferimenti nelle classi](#)
[Membro classe all'interno di classi](#)
[Usi della parola static](#)
 [Primo uso - regole di collegamento](#)
 [Secondo uso - variabili locali di una funzione per cambiare tempo di vita](#)
 [Terzo uso - utilizzo nelle classi](#)
 [Membri dato](#)
 [Membri funzione](#)

[Funzioni const](#)
[Conversione mediante costruttori](#)
[Preprocessore](#)
[Cenni sulla ricorsione](#)

sintassi : grammatica di un linguaggio

semantica : significato delle istruzioni scritte, correttezza di un programma

Compito del Compilatore & Linker : traduce il programma sorgente in programma eseguibile

- analisi programma sorgente
 - analisi lessicale
 - analisi sintattica
- traduzione
 - generazione del codice
 - ottimizzazione del codice

Rappresentazione dell'informazione

Algoritmo Div & Mod

Dato un numero N in base dieci e una base scelta β , con questo algoritmo posso convertire il numero N in base β .

Svolgo la divisione fino a che non trovo uno zero.

Esempio $N = 38, \beta = 6$

$$\begin{array}{lll} q_0 = 38 & & \\ q_1 = 38 \text{ DIV } 6 = 6 & a_0 = 38 \text{ MOD } 6 = 2 & \\ q_2 = 6 \text{ DIV } 6 = 1 & a_1 = 6 \text{ MOD } 6 = 0 & \\ q_3 = 1 \text{ DIV } 6 = 0 & a_2 = 1 \text{ MOD } 6 = 1 & \\ a_2 a_1 a_0 = 102 & & \end{array}$$

Basi particolari

Potenze di due

I numeri che sono potenze della base due possono essere individuati così

$$2^p = (1(0\ldots0)_{p \text{ volte}})_2$$

$$\text{Esempio 1: } 2^2 = 4 = (100)_2 \quad \text{Esempio 2: } 2^8 = 256 = (100000000)_2$$

Base otto

Prendo il numero 100101 in base due, lo divido in terne e ottengo :

n1=100

n2=101

Li converto in base dieci e ottengo:

$$n_1=4$$

$$n_2=5$$

$$(100101)_2 \Rightarrow (45)_8$$

Se l'ultima terna ha meno di tre cifre, basta aggiungere degli zeri.

Altro esempio :

$$(532)_8 = (101\ 011\ 010)_2$$

Base sedici

Superando le dieci cifre si usano anche le prime sei lettere maiuscole dell'alfabeto.

In questo caso divido il numero in quaterne.

$$(2A)_{16} \Rightarrow (0010\ 1010)_2 \Rightarrow (42)_{10}$$

Convertiamo da base due a base sedici $(01011011)_2$

Divido in quaterne:

$$n_1 = 0101 = (5)_{16}$$

$$n_2 = 1011 = (B)_{16}$$

Quindi:

$$(01011011)_2 = (5B)_{16}$$

Somme di numeri naturali in binario

Somma fra due numeri naturali espressi in binario.

Per sommare 10011 e 101101, basta metterli in colonna allineandoli a destra (come si fa con i numeri in base 10) e poi tenere presenti le seguenti regole:

0 +	0 +	1 +	1 +	1 +
0 =	1 =	0 =	1 =	1 =
0	1	1	10	11

genera un riporto

genera un riporto

eventuale riporto generatosi precedentemente

Esempio: calcolare la somma di 101100 e 111010

11 ← riporti generati

$$\begin{array}{r} 101100 \\ + 111010 \\ \hline 1100110 \end{array}$$

Attenzione all'overflow!

Rappresentazione dei numeri Z

Modulo e segno

$$[-(2^{p-1}-1); + (2^{p-1}-1)]$$

1° bit di segno (0 positivo)
 (due rappresentazioni +0 e -0)
EX $a = +5 \quad p = 4$
 $|+5| = 5 \quad \text{segno} = 0 \Rightarrow 0101$
EX $a = -5 \quad p = 4$
 $|+5| = 5 \quad \text{segno} = 1 \Rightarrow 1101$
EX $A = 01000 \quad p = 5$
 $\text{segno} = + \Rightarrow +8$
EX $A = 11001 \quad p = 5$
 $\text{segno} = - \Rightarrow -9$

Complemento a 2

$$[-z]^{p-1}; + (z^{p-1} - 1)$$

$a \geq 0$ lo rappresento normalmente

$$a < 0 - (2^r - |a|)$$

EX $a = +7 \quad r=4 \Rightarrow (0111)_2$

EX $(0111)_2 \quad r=4 \quad \text{segno} = + \Rightarrow +7$

EX $a = -8 \quad r=4$

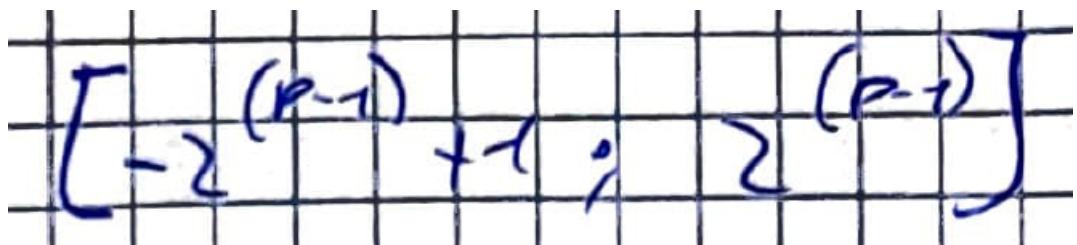
$$|-8| = 8 \quad 2^4 - 8 = 8 \Rightarrow (1000)_2$$

EX $(1111)_2 \quad r=4$

$$\text{segno} = - \quad (1111)_2 = (15)_{10}$$

$$-(2^4 - 15) = -1$$

Rappresentazione con BIAS



$$a + 2^{r-1} - 1$$

EX $a = 8 \quad r=4 \quad 8 + (2^{4-1} - 1) = 15 \Rightarrow (1111)_2$

EX $a = -7 \quad r=4 \quad -7 + (2^{4-1} - 1) = 0 \Rightarrow (0000)_2$

EX $A = 0111 \quad r=4 \quad 7 + (2^{4-1} - 1) = 0$

EX $A = 0101 \quad r=4 \quad 5 + (2^{4-1} - 1) = -2$

Rappresentazione numeri R

Virgola fissa

ALGORITMO PARTE INTERA E PARTE FRAZIONARIA

EX $p=16$ $A = 5$ (n° bit pt. frz.)

$r = +331,6875$

\Rightarrow PARTE INT. $331 = (101001011)_2$

\Rightarrow PARTE FRZ. $0,6875$

- MOLTIPLICO LA PRECEDENTE PT. FRZ. $\times 2$
- DEL RISULTATO NUOVO PT. FRZ. È PT. INT.
QUELLO INTERO SARÀ 1 o 0
- MI FERMO SE TROVO 0 o 0 SE FINISCO i bit

$A_0 = 0,6875$

$A_{-1} = 0,6875 \cdot 2 = 1,375 \Rightarrow 0,375$

$\bar{r}_1 = \textcircled{1} \quad \overbrace{}$

$A_{-2} = 0,375 \cdot 2 = 0,75$

$\bar{r}_2 = \textcircled{0} \quad \overbrace{}$

$A_{-3} = 0,75 \cdot 2 = 1,5 \Rightarrow 0,5$

$\bar{r}_3 = \textcircled{1} \quad \overbrace{}$

$A_{-4} = 0,5 \cdot 2 = 1,0 \Rightarrow 0 \quad \Rightarrow \begin{array}{l} \text{AGGIUNGO 2 ZERI IN ADE} \\ \text{E 1 IN FONDO} \end{array}$

$\bar{r}_4 = \textcircled{1} \quad \overbrace{}$

$\Rightarrow (001010010110110)_2$

Virgola mobile

$$r = \pm m \cdot b^e$$

m : PARTE IN VIRGOGLA FISSA (PARTE INFERIORE OCCUPA)
1 bit

e = ESPONENTE

$$\pm 1,11001 \cdot 2^{+11}$$

$$R = \{S, E, F\}$$

S = SEGNO

E = ESPONENTE

F = PT. FR.

NUMERI CON
MODULO MAX
 $A \pm \infty$

$$R = \{0, 1111, 111111\}$$

$$R = \{1, 1111, 111111\}$$

NUMERI PIÙ VICINI
ALLO 0 ($0^{\pm 0}$)

$$R = \{0, 00000, 00000000\}$$

$$R = \{1, 00000, 00000000\}$$

$$R_1 = 12,31 \cdot 10^{-4}$$

$$R_2 = -0,31 \cdot 10^{-4}$$

VARIAZIONE bit DISTRIBUITI

• HALF PRECISION : 16 bit { 5 ESP.

{ 10 PT. FR.

• SINGLE PRECISION : 32 bit { 8 ESP.

{ 23 PT. FR.

• DOUBLE PRECISION : 64 bit { 15 ESP.

{ 52 PT. FR.

• QUADDOUBLE PRECISION : 128 bit { 15 ESP.

{ 112 PT. FR.

INDIVIDUA IL SEGNO

- E = RAPPRESENTAZIONE CON BIAS

$$e = E - \text{BIAS} = E - (2^{e-1} - 1)$$

- TROVO M PARTENDO DA F

$$F = 2^{-6} \quad m = 1 + 2^{-6}$$

(LO \otimes NON SI RAPPRESENTA)

EX $R = \{-1, 10011, 1110100101\}$

$S = -$

$$e = E - \text{BIAS} = 19 - (2^{e-1} - 1) = 4$$

$$F = F \cdot 2^{-10} = 1110100101$$

$$r = -(1.1110100101 \cdot 2^4) = -(11110,100101)$$

$$\Rightarrow -30,578125$$

EX

$r = 2,3$ HALF PRECISION

$s = 0$

VIRGOGLA $F \times = 10.01001 \rightarrow 1,001001 \cdot 2^{-1}$

MIC SERVEMO 10 CIFRE X 09 PT. FR.

$1,001001001 \cdot 2^{-1}$

$F = 0,001001001 \cdot 2^{-1} = 0010011001$

$E = e + b_12^5 = 1 + (2^{51}-1) = 16 \quad E = (0000)_2$

$R = \{0,10000,0010011001\}$

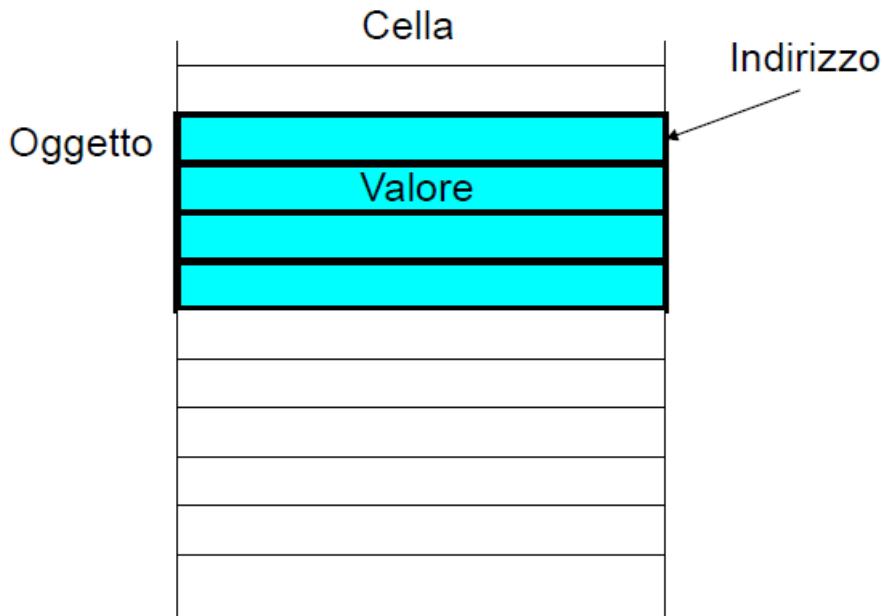
Proprietà degli operatori C++

- posizione rispetto ai suoi operandi (argomenti)
 - prefisso (+5) op arg
 - postfisso (x++) arg op
 - infisso (4+5) arg1 op arg2
- numero di argomenti (arietà)
 - op arg (arietà 1) (operatore unario)
 - arg1 op arg2 (arietà 2) (operatore binario)
- precedenza (priorità) nell'ordine di esecuzione
 - gli operatori con priorità più alta vengono eseguiti per primi
 - arg1 + arg2 * arg3 (il prodotto viene eseguito prima della somma)
- associatività (ordine di esecuzione di operatori con stessa priorità)
 - arg1 + arg2 + arg3 \Rightarrow (arg1 + arg2) + arg3 (assoc. da sinistra)
 - arg1 = arg2 = arg3 \Rightarrow arg1 = (arg2 = arg3) (assoc. da destra)

Oggetti (variabili) C++

Un oggetto è una variabile, cioè un contenitore di informazione tipizzato.

Le variabili vengono salvate in memoria RAM (insieme di celle). Una cella in genere ha una dimensione di 1 byte (8 bit).



In memoria RAM un oggetto è un gruppo di celle consecutive che vengono considerate dal programmatore come un'unica cella informativa.

Attributi di un oggetto:

- Indirizzo della prima cella
- Valore (contenuto di tutte le celle)

Gli oggetti possono essere costanti, cioè il loro valore una volta definito non è più modificabile.

Come abbiamo detto sopra un oggetto è tipizzato, cioè ha un tipo. Il tipo, una volta definito non può essere cambiato (non si può dichiarare un oggetto senza definirne prima il tipo).

Tipo di un oggetto:

- insieme di valori rappresentabili con quel tipo
- insieme delle operazioni definite su quel tipo
- occupazione di memoria

I tipi del C++

- Tipi fondamentali (tipi aritmetici o discreti)
 - tipi predefiniti
 - tipo intero e naturale (tipi numerici)
 - tipo reale
 - tipo booleano
 - tipo carattere
 - tipi enumerazione
- Tipi derivati

- si ottengono dai tipi fondamentali e permettono di costruire strutture dati più complesse

Il tipo intero

Occupà 4 byte. Serve per memorizzare numeri senza parte frazionaria.

```
int a = 5;
int b = 4.5; //prende solo il 4
```

Posso anche inizializzare alcune variabili intere con un numero in base 8 o 16.

Per la base 8 devo mettere uno 0 come prima cifra, per la base 16 uso 0x.

```
int ott = 011; //ott = 11 base 8 => 9 in base 10
int ex = 0xF; //ex = 0xf base 16 => 15 in base 10
```

Il tipo unsigned

E' effettivamente un intero senza il segno. Occupa come l'intero 4 byte.

Si può usare anche questa scrittura:

```
unsigned u1 = 5u;
unsigned u2 = 15u;

unsigned short //16 bit
unsigned int   //32 bit
unsigned long  //64 bit
```

Se N è il numero di bit impiegati per rappresentare gli interi, i valori vanno da 0 a $2^N - 1$. Il tipo unsigned viene usato soprattutto per operazioni a basso livello. Laddove il contenuto di alcune celle di memoria viene gestito come una configurazione di bit.

Operatori bit a bit:

- | OR bit a bit
- & AND bit a bit
- ^ OR esclusivo bit a bit
- ~ complemento bit a bit
- << traslazione a sinistra
- >> traslazione a destra

```
unsigned short a = 0xFFFF9; //1111 1111 1111 1001 (65529)
unsigned short b = ~a;      //0000 0000 0000 0110 (6)
unsigned short c = 0x0013; //0000 0000 0001 0011 (19)
unsigned short d = 0b0000000000010011 //come sopra ma esplicitandolo direttamente

unsigned short c1, c2, c3;
c1 = b | c;    //0000 0000 0001 0111 (23)
c2 = b & c;    //0000 0000 0000 0010 (2)
c3 = b ^ c;    //0000 0000 0001 0101 (21)

unsigned short b1, b2;
```

```
b1 = b << 2; //0000 0000 0001 1000 (24)
b2 = b >> 1; //0000 0000 0000 0011 (3)
```

Il tipo reale

Tipo che occupa 64 bit. Può anche esprimere esponenziali.

```
double d1 = 3.3;
double d2 = -12.14e-3; //-12.14*10^(-3)

float f = -2.2;
float g = f - 12.12F;
```

La parte intera o la parte frazionaria, se valgono zero, possono essere omesse.

Le operazioni sugli interi e sui reali si indicano con gli stessi simboli, ma sono operazioni diverse.

Il tipo booleano

Tipo discreto avente due costanti predefinite: *true* (1) o *false* (0).

Occupà 1 byte.

```
bool a = true;
bool b = 0;
```

Operazioni:

- || OR logico
- && AND logico
- ! NOT logico

L'operatore AND ha priorità maggiore rispetto all'OR.

Operatori di confronto

- == uguale
- != diverso
- > maggiore
- < minore
- >= maggiore o uguale
- <= minore o uguale

Il risultato è un booleano, vale *false* se la condizione non è verificata, *true* altrimenti. Gli operatori di uguale e diverso hanno priorità maggiore rispetto agli altri.

Il tipo carattere

Caratteri opportunatamente codificati.

Sono possibili tutte le operazioni definite sugli interi, che agiscono sulle loro codifiche. La codifica più comune è quella ASCII.

```
char c = 'a';
```

Caratteri di controllo: rappresentati da combinazioni speciali che iniziano con \

- nuova riga \n
- tabulazione \t
- ritorno carrello \r
- barra invertita \\
- apice \'
- virgolette \"

Ordinamento:

- le codifiche rispettano l'ordine alfabetico delle lettere
- in ASCII le lettere maiuscole vengono prima di quelle minuscole 'A' < 'a'

Possono anche essere scritti usando il loro valore nella codifica adottata dall'implementazione (es. ASCII). Il valore può essere espresso in decimale, ottale o esadecimale.

```
char c1 = 'c';
char t = '\t';
char d = '\n';

char c2 = '\x63'; // 'c' in esadecimale
char c3 = '\143'; // 'c' in ottale
char c4 = 99; // 'c' in decimale

char c5 = c1 + 1; // 'd'
char c6 = c1 - 2; // 'a'
int i = c1 - 'a'; // 2
```

Il tipo enumerato

E' un tipo fondamentale che può essere customizzato dal programmatore.

- Costituito da insiemi di costanti intere, ciascuna individuata da un identificatore e detta enumeratore
- Viene usato per variabili che assumono un numero limitato di valori
- Servono a rappresentare informazioni non numeriche
- Non predefinito, ma definito dal programmatore

Le operazioni definite sono di solito quelle di confronto, sono comunque possibili tutte le operazioni degli interi.

```
enum Semaforo {ROSSO, GIALLO, VERDE};
Semaforo s = ROSSO;
int i = ROSSO; // i vale 0

// NON SI POSSONO FARE ASSEGNAZIMENTI INVERSI
s = GIALLO - ROSSO; // NO
s = 1; // NO
s = i; // NO
```

```
enum {PROVA1=10, PROVA2, PROVA3=9, PROVA4}
//PROVA1 -> 10
//PROVA2 -> 11
//PROVA3 -> 9
//PROVA4 -> 10
```

Conversioni implicite

```
int i=10, j;
float f=2.5, h;
j=f+3.1; //5
h=f+1; //3.5
```

Nella conversione da double a int c'è una perdita di informazione.

In alcuni casi si può avere anche una perdita passando da int a double, poiché gli interi sono rappresentati in forma esatta e i reali sono rappresentati in forma approssimata (di solito succede con i numeri grandi).

Ad una variabile intera posso assegnare un valore di tipo reale, booleano, carattere o enumerazione.

A una variabile di tipo reale può essere assegnato un valore di tipo intero.

A una variabile di tipo carattere può essere assegnato un valore di tipo intero, booleano o enumerazione.

Il linguaggio gestisce tali assegnamenti, se non riesce a gestirli non li fa eseguire.

Conversioni esplicite

Fa la conversione quando esiste la conversione implicita inversa.

```
int j=(int) 1.1; //cast
float f=float(2) //notazione funzionale
```

Può essere usato per assegnare un intero ad enumerato, per velocizzare in qualche modo l'algoritmo (non si fa quasi mai).

Costanti

Un tipo di variabile che una volta inizializzato non si può più modificare. Obbligatoria l'inizializzazione quando viene dichiarata

```
const float pi = 3.14;
const int a; //ERRORE
pi=2; //ERRORE
```

Operatore sizeof

Operatore che ritorna la dimensione in byte di una variabile.

```
cout<<sizeof(char)<<endl; //1
cout<<sizeof(int)<<endl; //4
cout<<sizeof(float)<<endl;//4
```

Espressioni di assegnamento

Serve a calcolare un valore. Opzionale perchè in certi casi può essere utile usare una istruzione vuota usando direttamente ";".

```
variableName = expression;
(left value)      (right value)
```

Ha una priorità bassa, per far sì che vengano fatte prima le operazioni alla sua destra.

Left value : una variabile (un qualcosa che ha un indirizzo in memoria)

Right value : un valore (ma anche un'altra variabile)

```
int k,j,i;
k=i=j=5; //mette in tutte e tre 5 (associativo da destra)
```

Per aggiornare la solita variabile invece:

```
int i=0;
i++; //aggiungo 1
i+=5;//aggiungo 5
//funziona anche con i--
//e con gli altri simboli

i=0;
int j;
//incremento postfisso
j=i++; //j=0
//prima assegna e poi incrementa

i=0; j=0;
//incremento prefisso
j=++i; //j=1
//prima incrementa e poi assegna
```

Alcune eccezioni:

```
j=++++i;    //OK
j=++i++;    //NO
j=(++i)++; //OK
j=i++++;   //NO
```

Espressioni aritmetiche e logiche

Calcolo delle espressioni: vengono rispettate le precedenze e le associatività degli operatori che compaiono.

- prima vengono valutati i fattori, calcolando i valori delle funzioni e applicando gli operatori unari (prima increm e decr postfissi, poi quelli prefissi, NOT logico (!), meno unario (-) e più unario (+))
- poi vengono valutati i termini applicando gli operatori binari
 - moltiplicativi (*, / , %)
 - additivi (+, -)
 - relazione (<,, ==, !=)
 - uguaglianza (==, !=)
 - logici (nell'ordine, &&, ||)
 - assegnamento (=,, +=, -=, *=, /=, etc)

Le parentesi tonde fanno diventare qualsiasi espressione un fattore, che viene calcolato per primo.

Gli operatori aritmetici binari sono associativi a sinistra, quelli unari e di assegnamento invece a destra.

Regola del cortocircuito ("Shortcut rule")

```
bool a=false;
int b=5;
a && (++b);
cout<<b; //5

a=true;
a || (++b);
cout<<b; //5
```

Visto che la prima espressione dell'AND logico è già falsa, la condizione totale sarà sicuramente falsa, non ha senso valutare la seconda, la variabile i non viene quindi incrementata.

Nella seconda parte invece nell'OR logico mi basta che una delle due sia vera per far sì che la condizione totale sarà sicuramente vera, quindi non viene valutata la seconda, come conseguenza anche in questo caso i rimarrà 5.

Istruzione if

La condizione deve essere un operatore booleano, altrimenti non verrà effettuata alcuna operazione. Vengono usati gli operatori di confronto.

```
if( condizione ){ //se condizione vera..
  //..fai questo
}
else{ //altrimenti..
  //..fai quest'altro
}
```

Le condizioni possono anche essere composte, usando gli operatori logici AND e OR.

E' possibile mettere infinite condizioni in cascata (if dentro if...).

Il ramo *else* può anche essere omesso.

Operatore condizionale

e1 ? e2 : e3

Unico operatore ternario del C++

e1 è un'espressione logica che viene valutata, se è vera viene valutata e2, altrimenti verrà valutata e3

Scorciatoia per usare l'istruzione if, l'unica differenza è che questo operatore restituisce un valore (if non restituisce niente), come nel caso sotto.

```
//calcolo minimo 2 valori
int i,j,min;
cin>>i>>j;
min = (i<j ? i : j);
cout<<min;
```

```
//calcolo minimo 3 valori
int i,j,z,min;
cin>>i>>j>>z;
min = i<j ? (i<z ? i++ : z++) : (j<z ? j++ : z++);
cout<<min;
//espressione molto compatta ma poco leggibile
```

Operatore ","

```
int a=2, b=3;
a=b++, 5;
cout<<b<<endl; //4
cout<<a<<endl; //5
//l'operatore "," ha la priorità più bassa di tutti gli altri
```

Usato ogni volta che la grammatica richiede una singola espressione ma noi vogliamo mettere cene più di una.

Istruzione switch e break

Permette di selezionare diversi casi, rispetto all'if che permette solo "due strade"

L'espressione è di solito costituita da una variabile a valori discreti (int, char, enum ecc)

Case-label: contengono espressioni costanti il cui valore possa essere stabilito a tempo di compilazione (35, const int, sizeof(), ROSSO).

Si usa il break per uscire dallo switch.

```
switch(expr){
    case ST1:
        ...
        break;
    case ST2:
        ...
        break;
    default: //eseguita se le altre non sono eseguite, può anche mancare
        ...
```

```
    break;  
}
```

Posso anche combinare più statement insieme, si comportano come un OR dentro a un if.

Istruzioni iterative

Istruzione while

Il ciclo while cicla finchè l'istruzione tra parentesi tonde è vera, se falsa esce dal corpo del while, oppure non ci entra nemmeno. La condizione può anche essere composta.

```
while( condizione ){  
    //finchè la condizione rimane vera fai qualcosa  
}  
  
//ESEMPIO  
int n=10;  
while(n>0){  
    cout<<n;  
    n--;  
}
```

Istruzione do

Al contrario del while, prima si esegue il ciclo una volta e poi si valuta la condizione, si esce dal ciclo non appena la condizione è falsa.

```
do{  
    //fai qualcosa finchè la condizione rimane vera  
}while( condizione );  
  
//ESEMPIO  
do{  
    cin>>n;  
}while(n>0);
```

Istruzione for

Un ciclo simile al while, che richiede un inizializzazione, una condizione e un passo

```
for(inizializzazione; condizione; passo){  
    //finchè la condizione è vera fai qualcosa  
    //il passo serve per cambiare lo stato a una variabile  
    //quindi per (a un certo punto) non far valere la condizione  
}  
  
//ESEMPIO  
int n=10;  
for(int i=0; i<n; i++){  
    cout<<n;  
}
```

Istruzioni di salto

Istruzione break

Salto all'istruzione immediatamente successiva al corpo del ciclo o dell'istruzione switch.

Istruzione continue

Provoca la terminazione di un'iterazione del ciclo che la contiene.

Tipi derivati

- riferimenti
- puntatori
- array
- strutture
- unioni
- classi

Sono tipi che derivano da quelli fondamentali

Riferimenti

Identificatore che individua un oggetto, il riferimento di default è il nome di un oggetto; oltre a quello si possono definire altri riferimenti per un oggetto (alias).

Non si possono definire riferimenti di riferimenti.

Sono opzioni diverse che si riferiscono al solito oggetto.

```
int i=10;
int &r=i;
int &s=r;
cout<<i<<" "<<r<<" "<<s; //10 10 10
r++;
cout<<i<<" "<<r<<" "<<s; //11 11 11

int &c=10; //NO
//il right value deve essere qualcosa che occupa uno spazio in memoria
```

Mi posso riferire al solito valore 10 usando altre variabili che operano sullo stesso valore.

I riferimenti, una volta dichiarati, vanno sempre inizializzati.

Riferimenti costanti

```
int i=1;
const int &r=i; //anche se i non è costante
i++;
//come se potessi solo accedere in lettura ad i
r++; //NO

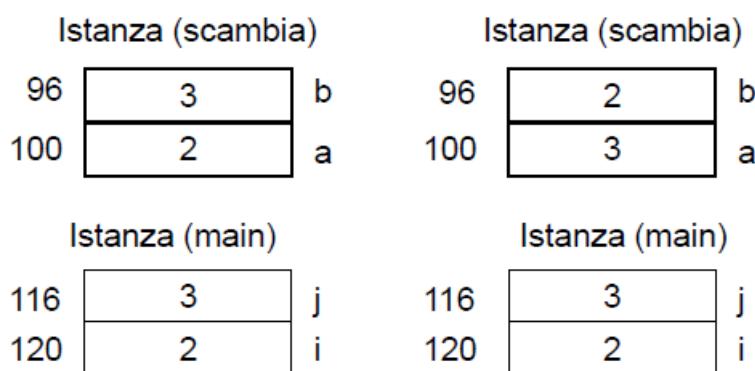
int &c=r; //NO
```

Riferimenti come argomenti

```

void scambia(int a, int b){
    //ERRATO
    int c=a;
    a=b;
    b=c;
    //dentro la funzione i valori sono stati effettivamente scambiati
}
int main(){
    int i,j;
    cin>>i>>j; //2 3
    scambia(i,j);
    cout<<i<<" "<<j; //2 3
    //nel main non sono però scambiati
}

```

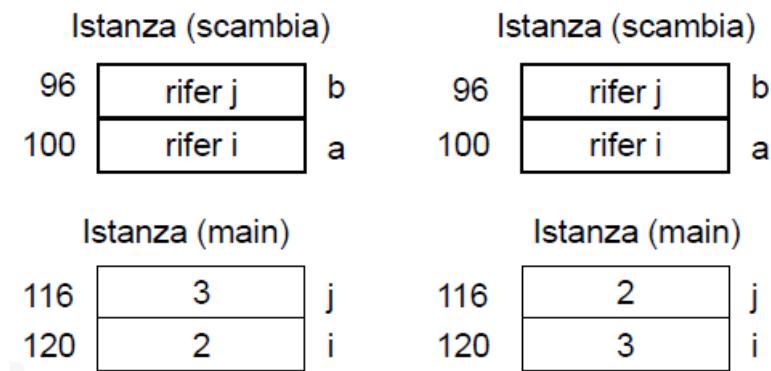


Vorrei ritrovarmi invece i due valori i e j scambiati anche nel main, passo alla funzione i parametri come riferimento.

```

void scambia(int &a, int &b){
    int c=a;
    a=b;
    b=c;
}
int main(){
    int i,j;
    cin>>i>>j; //2 3
    scambia(i,j);
    cout<<i<<" "<<j; //3 2
}

```



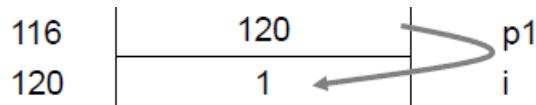
L'argomento formale corrisponde a un contenitore senza nome, che ha per valore il riferimento. Nel corpo della funzione, ogni operazione che coinvolge l'argomento formale agisce sull'entità riferita.

Puntatori

Oggetto il cui valore rappresenta l'indirizzo di un altro oggetto o di un'altra funzione. Come valore destro accetta solo indirizzi di memoria, la variabile a cui punta deve essere dello stesso tipo del puntatore stesso.

Occupava sempre 4 o 8 byte (in base all'architettura) indifferentemente dal tipo puntato.

```
int *p; //puntatore a intero
int i=1;
int *p1=&i; //operatore indirizzo
int *p2; //se non inizializzato, punta a un indirizzo casuale in memoria
```



```
int i=1;
int *p1=&i;
*p1=10; //come fare i=10 (i ora vale 10)
int *p2=p1; //p1 è un indirizzo
cout<<i<<" "<<*p1<<" "<<*p2; //10 10 10
*p2=20; //i ora vale 20
cout<<i<<" "<<*p1<<" "<<*p2; //20 20 20
cout<<&i<<" "<<p1<<" "<<p2; //0x22ff6c 0x22ff6c 0x22ff6c
```

Tramite puntatori posso aggiornare il valore a cui puntano.

E' sempre buona cosa inizializzare un puntatore quando viene dichiarato, poiché può puntare casualmente a un valore di un'altra variabile, se non me ne accorgo rischio di modificare involontariamente il valore di tale variabile.

Variabili di blocco e variabili globali

Variabili di blocco

```
int main(){
    int a; //var di blocco main
    {
        int b; //var di blocco
        b=10;
        cout<<b;
    } //quando esco dal blocco b viene distrutta
    cout<<b; //NO
    int c=7; //var di blocco main
} //quando esco dal blocco main a e c vengono distrutte
```

Le variabili di blocco vengono impilate nello stack (pila). Man mano che vengono distrutte il loro posto nello stack viene liberato.

Variabili globali

```

int n; //var globale
int main(){
    n=10;
    int a=7;
    int b,c; //a b c variabili di blocco
}

```

Le variabili globali iniziano ad esistere non appena il programma entra in esecuzione, sono visibili in tutti i blocchi del programma. Una variabile globale verrà deallocata quando il programma termina.

Concetto di funzione

PROBLEMA : potrei aver bisogno di calcolare più volte il fattoriale di un numero nel main, non voglio però replicare sempre tutto il codice per n volte.

Una funzione è un gruppo di istruzioni che può essere chiamato n volte, scrivendo il codice una sola volta.

Posso utilizzare una funzione che mi calcola il fattoriale e la richiamo ogni volta che ne ho bisogno.

```

#include <iostream>
using namespace std;
int n; //anche la funzione fatt() vede la var n
int fatt(){
    int ris=1;
    for(int i=2;i<=n;i++)
        ris*=i;
    return ris;
}
int main(){
    int f;
    cin>>n;
    f=fatt();
    cout<<f;

    cin>>n;
    f=fatt();
    cout<<f;

    cin>>n;
    f=fatt();
    cout<<f;
}
//faccio la stessa operazione 3 volte ma scrivendo il codice 1 volta sola

```

Posso anche riscrivere il mio programma anche senza la variabile globale n.

```

#include <iostream>
using namespace std;
int fatt(int n){ //passo come argomento alla funzione una variabile (argomento formale)
    int ris=1;
    for(int i=2;i<=n;i++)
        ris*=i;
    return ris;
}
int main(){
    int i, f;
    cin>>i;
    f=fatt(i); //ogni volta passo la variabile presa in ingresso alla funzione
    cout<<f;

    cin>>i;
}

```

```

f=fatt(i); //i è l'argomento attuale
cout<<f;

cin>>i;
f=fatt(i);
cout<<f;
}

```

Devo definire (o solamente dichiarare e poi definire dopo) una funzione prima del main per far sapere al compilatore il tipo degli argomenti, così quando nel main vado a chiamarla il compilatore controlla che gli argomenti siano almeno del tipo corretto.

```

#include <iostream>
using namespace std;
//firma della funzione
int fatt(int); //dichiarazione senza definizione
//posso anche specificare il tipo senza la variabile

int main(){
    int i, f;
    cin>>i;
    f=fatt(i);
    cout<<f;

    cin>>i;
    f=fatt(i);
    cout<<f;

    cin>>i;
    f=fatt(i);
    cout<<f;
}
int fatt(int n){
    int ris=1;
    for(int i=2;i<=n;i++)
        ris*=i;
    return ris;
}

```

Le variabili di blocco definite dentro a una funzione vengono anche dette variabili locali. In funzioni diverse possiamo usare nomi di variabili locali uguali, ogni variabile si riferirà alla rispettiva funzione.

Chiamata :

- gli argomenti formali e le var locali vengono allocati in memoria
- gli argomenti formali vengono inizializzati con i valori degli argomenti attuali (passaggio per valori)
- gli argomenti formali e le var locali vengono usati per le dovute elaborazioni
- al termine della funzione essi vengono deallocati

Istanza di funzione :

- nuova copia degli argomenti formali e delle var locali
- il valore di una variabile locale non viene conservato per istanze successive
- quando una funzione viene chiamata, viene creata in memoria un'istanza della funzione
- l'istanza ha un tempo di vita pari al tempo di esecuzione della funzione

Istruzione return

Serve per "ritornare" un valore che assume la funzione poi nel main.

La funzione ritorna un valore del suo stesso tipo.

```
unsigned int fibo(unsigned int n){  
    unsigned int i=0, j=1, s;  
    for(;;){  
        if((s=i+j)>n)  
            return j;  
        i=j;  
        j=s;  
    }  
}  
int main(){  
    unsigned int n;  
    cin>>n;  
    cout<<fibo(n);  
}
```

Overloading di funzioni

Posso definire funzioni con lo stesso nome, a patto che abbiano argomenti diversi. Non possono differire solo dal tipo di ritorno.

```
int sqrt(const int n){...};  
int sqrt(int n, int x){...};  
int sqrt(int n){...};
```

Parametri default

```
void stampaVettore(int *v, int n, char sep=' ', char par='['){  
    cout<<par;  
    for(int i=0;i<n-1;i++)  
        cout<<v[i]<<sep;  
    cout<<v[i]<<(par+2);  
}  
//se non passo gli ultimi due parametri, di default lascia quelli sopra
```

Strutture

n-upla ordinata di elementi, detti campi, ciascuno con uno specifico tipo e uno specifico nome.
Rappresenta una collezione di informazioni su un dato oggetto.

```
struct punto{  
    double x;  
    double y;  
};  
int main(){  
    punto r,s;  
    r.x=3; //selettore di membro  
    r.y=10.5;  
    s.x=r.x;  
    s.y=r.y+10.0;  
    punto t={1.0,2.0}; //inizializzazione diretta  
    //posso inizializzare anche con variabili o r.x, r.y  
    cout<<r.x<<" "<<r.y<<endl; //3 10.5
```

```

cout<<s.x<<" "<<s.y<<endl; //3 20.5
cout<<t.x<<" "<<t.y<<endl; //1.0 2.0

punto *p=&r;
cout<<p->x<<" "<<p->y<<endl; //3 10.5 (selettore di membro per puntatore)
}

```

Strutture dentro strutture

```

struct punto{
/*...*/
punto s; //ERRORE
};

struct punto{
/*...*/
punto *p; //OK
};

```

Tipi e oggetti array

n-upla ordinata di oggetti dello stesso tipo, ai quali ci si riferisce tramite indice, che rappresenta la loro posizione dentro all'array

```

const int N=5;
int v[N];
for(int i=0;i<N;i++) //inserimento
    cin>>v[i];
for(int i=0;i<N;i++) //stampa
    cout<<v[i];

cout<<v; //come fare cout<<&v[0] prima componente del vettore

```

104	0	v[0]
108	1	v[1]
112	2	v[2]
116	3	v[3]
120	4	v[4]

Se per caso dovessi andare a leggere fuori dal vettore o in celle che non sono state inizializzate, stampo valori casuali.

Array e puntatori

Posso calcolarmi gli indirizzi delle celle del vettore.

Se p punta alla prima cella del vettore p+i punta alla i-esima cella.

```

int v[4];
int *p=v; //*p=&v[0]
//il puntatore deve essere dello stesso tipo del vettore
for(int i=0;i<4;i++) //stampa normalmente

```

```

cout<<*(p+i); //lo posso fare perchè i è intero

//aritmetica dei puntatori
cout<<p+1-p; //1
cout<<int(p+1)-int(p); //4 (byte)

//posso fare anche così
for(int i=0;i<4;i++)
    cout<<*(v+i); //usando direttamente il vettore

```

Array come argomenti di funzioni

```

void stampa(const double*,int);
//void stampa(const double,int);
int main(){
    double v[]={3.0,4.0,5.0,6.0};
    double *p=v;
    //le tre sotto sono equivalenti
    stampa(p,4);
    stampa(&v[0],4);
    stampa(v,4);
}
//bisogna per forza passare alla funzione la lunghezza del vettore
//altrimenti impossibile farci un ciclo sopra

//le due sotto sono equivalenti
void stampa(const double *q, int lun){
    for(int i=0;i<lun;i++){
        cout<<*(q+i)<<endl; //alternativa 1
        cout<<q[i]<<endl;   //alternativa 2
    }
}
//non metto tra [] la dimensione perchè verrà ignorata
void stampa(const double v[], int lun){
    for(int i=0;i<lun;i++){
        cout<<*(q+i)<<endl; //alternativa 1
        cout<<q[i]<<endl;   //alternativa 2
    }
}

```

Array statici

```

const int n1=10; //dimensione costante
int v1[n1]; //vettore globale ("statico")
int main(){
    //...
}

```

Array automatici

```

int main(){
    int n2;
    cin>>n2;
    int v2[n2]; //vettore automatico, allocato sullo stack ("semi dinamico")
}

```

Mi consente di allocare il vettore della lunghezza giusta o a run-time, o dopo aver fatto la cin.

Non è un vero vettore dinamico perchè:

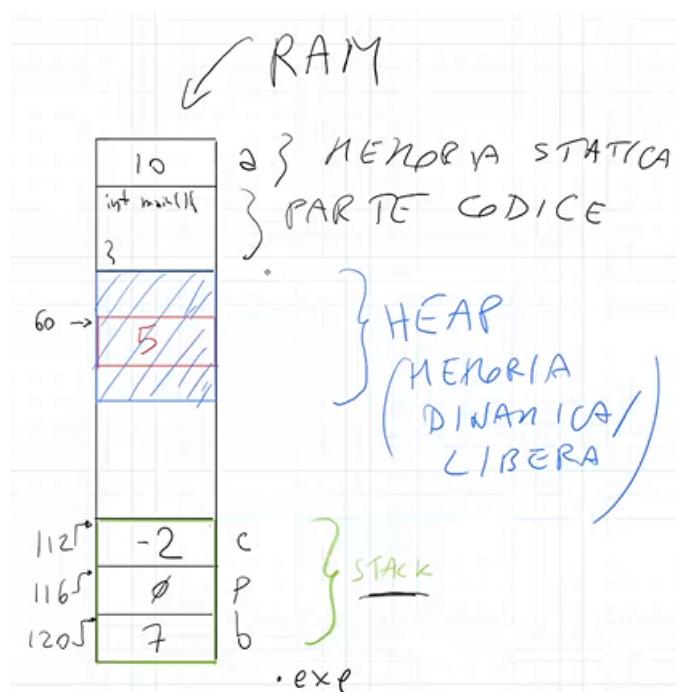
1. Va dimensionato in fase di dichiarazione
2. Da quando inizia ad esistere, fino alla sua distruzione avrà sempre la stessa lunghezza

Array dinamici

```

int a=10;
int main(){
    int b=7;
    int *p=nullptr;
    int c=-2;
    p=new int; //variabile intera allocata sullo heap (memoria dinamica)
    *p=5;
    cout<<*p;
    delete q; //dealloco la memoria
}

```



Memoria Heap

Il programmatore non è in grado di stabilire a priori il numero di variabili di un certo tipo che serviranno durante l'esecuzione del programma.

Per variabili array infatti dover specificare la dimensione è limitativo.

Possiamo dimensionare un array dopo aver scoperto durante l'esecuzione del programma quanto deve occupare.

Risulta possibile allocare delle aree di memoria durante l'esecuzione del programma, ed accedere a tali aree con puntatori.

Il puntatore punta ad una parte di memoria "occupata" e dimensionata in base al tipo specificato dopo la new, ma non ha effettivamente un nome.

La memoria viene associata via via durante l'esecuzione del programma, lo heap sicuramente avrà dei buchi, cosa impossibile nello stack.

```

int n3;
cin>>n3;
//rinuncio alla conoscenza del nome del vettore
int *p3=nullptr; //dichiarazione "vettore senza nome"
//...
p=new int [n3]; //definizione "vettore senza nome"
//dichiarazione e definizione separate
for(int i=0;i<n3;i++)
    cin>>p3[i]; //posso usare solo il puntatore per accederci
delete [] p3;

```

Funzioni su vettori dinamici

```

int *p6=nullptr; //dichiarazione vettore dinamico
cin>>lun6;
alloca(p6,lun6); //definizione o allocazione
leggi(p6,lun6); //riempimento
bubbleSort(p6,lun6); //operazione
//altre operazioni
stampa(p6,lun6); //stampo
dealloca(p6); //deallocazione

```

dealloca()

```

void dealloc(int* &vet){
    delete [] vet;
    vet=nullptr; //avrebbe senso per non creare errori
}
//devo passare il vettore per riferimento perchè vado a scriverci

```

alloca()

```

void alloca(int* &vet, int n){
    vet=new int[n];
}
//devo passare il vettore per riferimento perchè vado a scriverci

```

aggiungiNuovoElementoInFondo()

```

//posso aggiungere un nuovo elemento perchè la memoria dinamica
//non è "stretta" come la statica
void aggiungiNuovoElementoInFondo(int* &vd, int a, int &len){
    int* nuovoVD=new int[len+1];
    for(int i=0;i<len;i++)
        nuovoVD[i]=vd[i];
    nuovoVD[len]=a;
    delete [] vd;
    vd=nuovoVD;
    len++;
}
//passo il vettore per riferimento perchè cambio l'indirizzo del puntatore

```

Stringhe

Sequenza di caratteri, in C++ non esiste il tipo string.

Sono array di caratteri che memorizzano stringhe con carattere '\0' finale.

Una C-stringa è un array di caratteri con al suo interno almeno un carattere '\0' (marca di fine stringa). Una caratteristica principale è che posso fare direttamente la cout senza dover fare un for, si ferma da sola quando trova '\0'.

Con un semplice array di caratteri invece devo per forza usare un for, se provassi a fare un cout diretto stamperà roba a caso finché casualmente non viene trovato un '\0'.

```
char vc[]={'C','+', '+', '\0'}; //C-stringa
cout<<vc; //C++

char vc2[]={'C','+', '+'}; //array di caratteri
for(int i=0;i<3;i++){
    cout<<vc2[i];
}
while((*vc2)!='\0'){
    cout<<*vc2;
    vc2++;
}

char str[]="C++";
//la dimensione del vettore viene messa a 4, contando il \0 finale
cout<<str; //C++
char str2[10];
cin>>str2; //si può fare, aggiunge da solo \0
//non devo superare la dimensione, altrimenti rischio di scrivere dove non devo
```

Stampa tramite funzione

```
void stampaCStringa(const char *p){
    while(*p]!='\0'){
        cout<<*p;
        p++;
    }
    cout<<endl;
}
//possibile implementazione di una normalissima cout<<cStr;
//posso anche evitare di comunicare la dimensione del vettore
```

myStrlen()

```
int myStrlen(const char *p){
    int i=0;
    while(*p]!='\0'){
        i++;
        p++;
    }
    return i;
}
//ritorna l'occupazione logica in memoria
//occupazione fisica = occupazione logica + 1 ('\0')
```

myStrcpy()

```
//copia il valore di una stringa in un'altra
void myStrcpy(char *d, const char *s){
    int i=0;
```

```

while(s[i]!='\0'){
    d[i]=s[i];
    i++;
}
d[i]='\0';
}

```

mystrcmp()

```

//ritorna -1, 0, +1 in base al confronto tra le stringhe
int mystrcmp(const char *s1, const char *s2){
    int i=0;
    while(s1[i]!='\0' && s2[i]!='\0'){
        if(s1[i]<s2[i])
            return -1;
        else{
            if(s1[i]>s2[i])
                return +1;
        }
        i++;
    }
    if(s1[i]!='\0' && s2[i]=='\0')
        return +1;
    else{
        if(s1[i]=='\0' && s2[i]!='\0')
            return -1;
    }
    return 0;
}

```

Libreria cstring

strlen() → ritorna la lunghezza della stringa

strcat() → concatena le due stringhe

strcmp() → ritorna 0 se sono uguali, altri numeri in base all'ordine alfabetico

strcpy() → copia il contenuto del secondo argomento nel primo

```

#include <cstring>
using namespace std;
int main(){
    const char str[]="Ciao";
    cout<<strlen(str); //4

    const char str2[]="Miao";
    strcpy(str,str2);
    cout<<str; //Miao

    cout<<strcmp("scintilla","scivolo"); //<0
    cout<<strcmp("sci","sci"); //0
    cout<<strcmp("sci","ape"); //>0
}

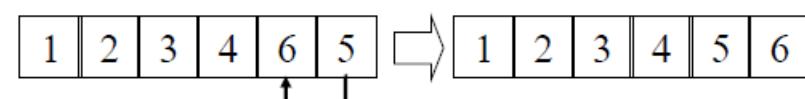
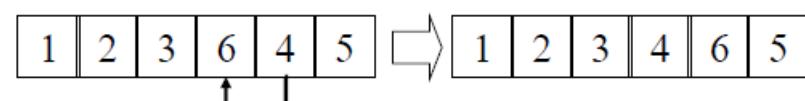
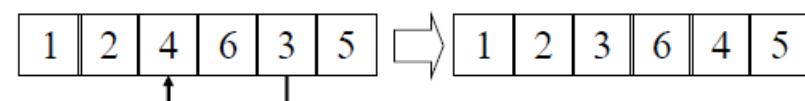
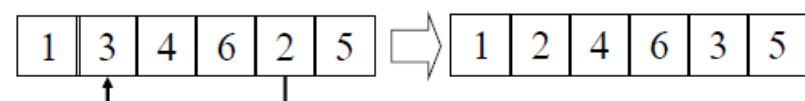
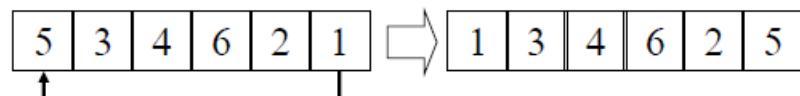
```

Ordinamento di vettori

Selection sort

Si cerca l'elemento più piccolo e lo si scambia col primo elemento, ad ogni ciclo non considero l'elemento con cui ho scambiato il più piccolo.

```
void selectionSort(T v[], int n){
    for(int i=0; i<n; i++){
        for(int j=i+1; j<n; j++)
            if(v[j]<v[i]){
                int aux=v[i];
                v[i]=v[j];
                v[j]=aux;
            }
    }
}
```



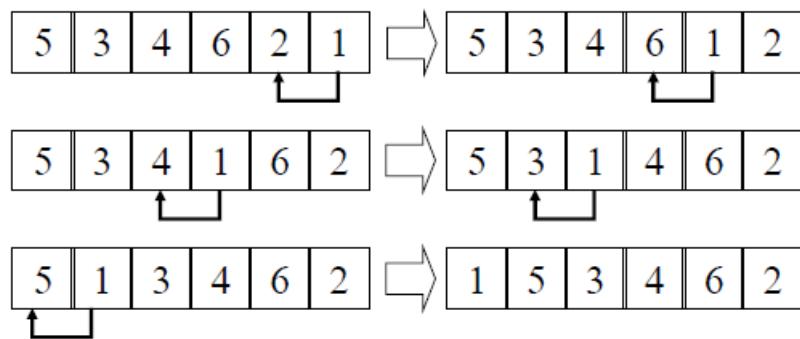
Complessità dell'algoritmo : $O(n^2)$

Bubble sort

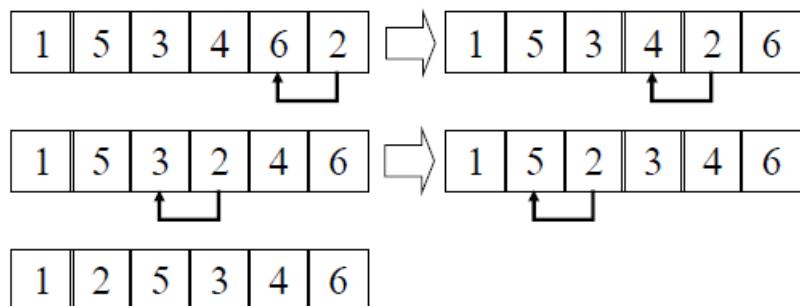
Scorro il vettore $n-1$ volte, da destra a sinistra, scambiando due elementi contigui se non sono in ordine.

```
void bubbleSort(T v[], int n){
    for(int i=0; i<n-1; i++)
        for(int j=n-1; j>i; j--)
            if(v[j]<v[j-1]){
                int aux=v[j];
                v[j]=v[j-1];
                v[j-1]=aux;
            }
}
```

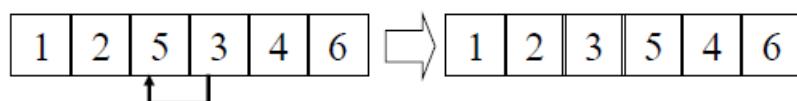
- **Prima volta**



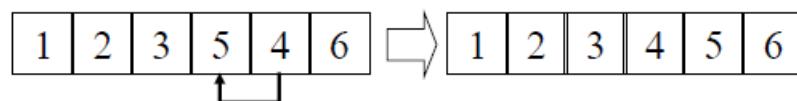
- **Seconda volta**



- **Terza volta**



- **Quarta volta**



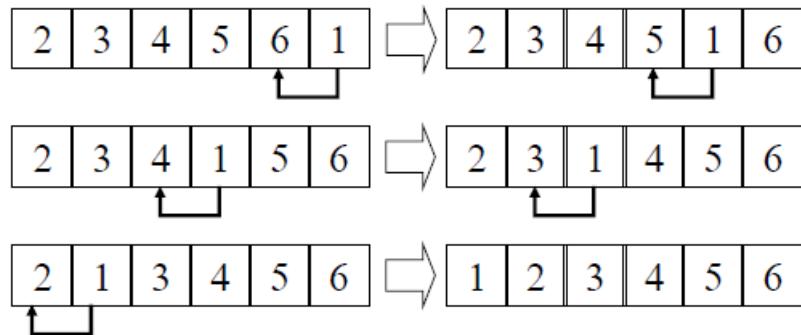
Complessità algoritmo : $O(n^2)$

Bubble sort ottimizzato

Ogni volta controllo se il vettore per caso già ordinato, nel caso mi fermo.

```
void bubbleSort(T vettore[], int n){
    bool ordinato=false;
    for(int i=n-1;i>0&&!ordinato;i--){
        ordinato=true;
        for(int j=0;j<i;j++){
            if(vet[j]<vet[j-1]){
                int aux=v[j];
                v[j]=v[j-1];
                v[j-1]=aux;
                ordinato=false;
            }
        }
    }
}
```

- **Prima volta**



La complessità può dipendere da caso a caso, nel caso peggiore $O(n^2)$

Ricerca in un vettore

Ricerca lineare

Scorro il vettore (o sottovettore) da *inf* a *sup* e se trovo l'elemento ritorno la posizione in *pos*.

```
bool ricercaLin(T v[], int inf, int sup, T k, int &pos){
    bool trovato=false;
    while(!trovato)&&(inf<=sup)){
        if(v[sup]==k){
            pos=sup;
            trovato=true;
        }
        sup--;
    }
    return trovato;
}
```

Complessità $O(n)$

Ricerca binaria

Prerequisito : vettore ordinato

Si confronta l'elemento cercato con l'elemento in posizione centrale, se sono uguali la ricerca finisce, altrimenti se l'elemento cercato è minore dell'elemento in posizione centrale la ricerca prosegue nella prima metà del vettore, altrimenti prosegue nella seconda metà.

```
bool ricBin(T ordVet[], int inf, int sup, T k, int &pos){
    while(inf<=sup){
        int medio((inf+sup)/2);
        if(ordVet[medio]==k){
            pos=medio;
            return true;
        }
        else{
            if(k<ordVet[medio])
                sup=medio-1;
            else
                inf=medio+1;
        }
    }
}
```

```

        }
    }
    return false;
}

```

Complessità O(logn)

Matrici

```

const int R=2;
const int C=3;
int m[R][C];
//inserimento
for(int i=0;i<R;i++)
    for(int j=0;j<C;j++)
        cin>>m[i][j];

int *p=&m[0][0]; //anche int *p=m;
//stampa
for(int i=0;i<R;i++){
    for(int j=0;j<C;j++)
        cout<<*(p+i*C+j)<<'\t';
    //cout<<p[i*C+j]; anche così
    //cout<<m[i][j]; ma anche così
    cout<<endl;
}
/*
1   2   3
4   5   6
*/
//altra inizializzazione
int m2[3][3]={ {0,1,2}, {10,11}, {100,101,102} };

```

Una matrice in memoria viene memorizzata per riga.

104	1	m[0][0]
108	2	m[0][1]
112	3	m[0][2]
116	4	m[1][0]
120	5	m[1][1]
124	6	m[1][2]

Passaggio di matrice a funzione

```

//stampa quando il numero di colonne è costante
void stampa(int m[][C], int R){//omento il numero di righe dentro []
    for(int i=0;i<R;i++){
        for(int j=0;j<C;j++)
            cout<<m[i][j]<<'\t';
        cout<<endl;
    }
}

//stampa per qualsiasi matrice

```

```

void stampa(int *mat, int r, int c){//sfrutto l'aritmetica dei puntatori
    for(int i=0;i<r;i++){
        for(int j=0;j<c;j++){
            cout<<mat[i*c+j]<<'\t';
            cout<<endl;
        }
    }
    //chiamata
    stampa(&m[0][0], numR, numC);
}

```

Vettore trattato come matrice

```

int r4=5;
int c4=6;
int m4[r4*c4]; //30 componenti
for(int i=0;i<r4;i++)
    for(int j=0;j<c4;j++)
        cin>>m4[i*c4+j];
stampa(&m4[0][0],r4,c4);

```

Matrici dinamiche

```

int r5;
cin>>r5;
int c5;
cin>>c5;
int *m5;
m5=new int[r5*c5];
for(int i=0;i<r5;i++)
    for(int j=0;j<c5;j++)
        cin>>m5[i*c5+j];
stampa(&m5[0][0],r5,c5);
stampa(m5,r5,c5);

```

Dichiarazioni `typedef`

Definisce degli identificatori che vengono usati per riferirsi a tipi nelle dichiarazioni, non creano nuovi tipi.

Crea un sinonimo di un tipo

```

typedef int T;
T a=5;
...
//se ho molte istruzioni che usano un tipo che voglio poi cambiare,
//basta che lo cambio nella prima riga e sarà cambiato per tutti

```

Liste

Problema : memorizzare numeri inseriti da tastiera finché non viene inserito il carattere '!'.

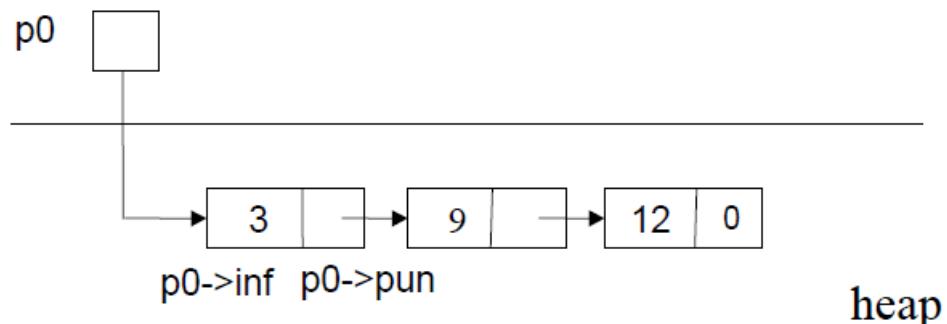
Struttura dati, formata da elementi dello stesso tipo collegati in catena, la cui lunghezza varia dinamicamente.

Ogni elemento è una struct, costituita da uno o più campi informazione e da un campo puntatore che contiene l'indirizzo dell'elemento successivo.

Il primo elemento possiede un puntatore detto "puntatore della lista".

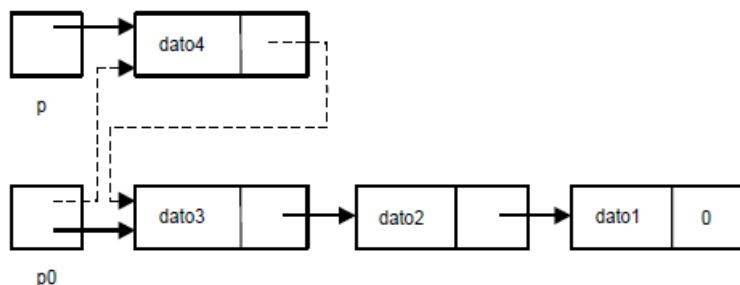
Il puntatore dell'ultimo elemento punta a `nullptr`.

```
typedef int T;
struct elem{
    T inf;
    elem* pun;
};
```



Lista di interi letti da tastiera terminati da '!'.

```
lista costruisceListaInteri(){
    int aux;
    lista p0=nullptr;
    while(cin<<"Inserisci il numero, ('.' per uscire) : ", cin>>aux){
        lista p=new elem;
        p->inf=aux;
        p->pun=p0;
        p0=p;
    }
    return p0;
}
```



Stampa lista

```
void stampa(const lista p0){
    const lista p=p0;
    while(p!=nullptr){
        cout<<p->inf<<" ";
        p=p->pun;
    }
}
```

```

    cout<<endl;
}

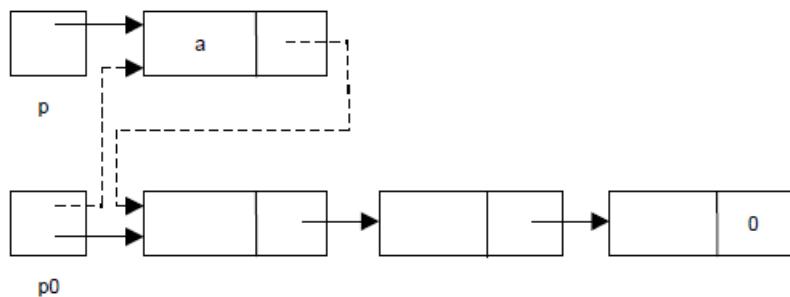
```

Inserimento in testa

```

void insTesta(lista &p0, T a){
    lista p=new elem;
    p->inf=a;
    p->pun=p0;
    p0=p;
}

```

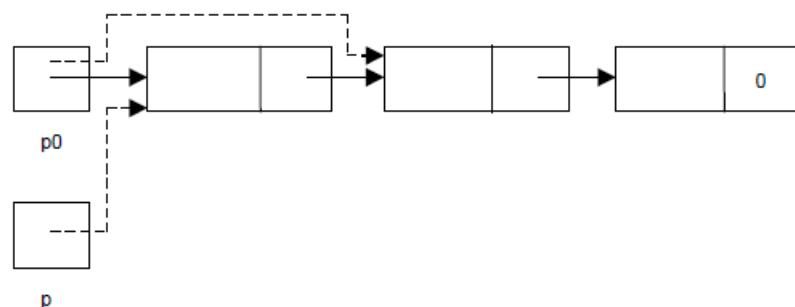


Estrazione in testa

```

bool esTesta(lista &p0, T &a){
    lista p=p0;
    if(p0==nullptr)
        return false;
    a=p0->inf;
    p0=p0->pun;
    delete p;
    return true;
}

```



Inserimento in coda

```

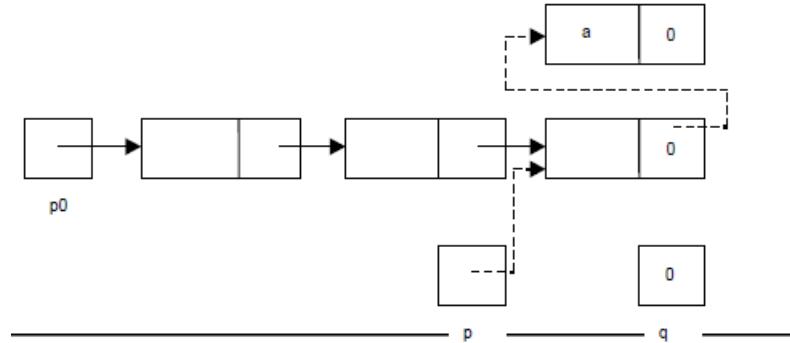
void insFondo(lista &p0, T a){
    lista p=nullptr, q=nullptr;
    q=p0;
    while(q!=nullptr){
        p=q;
        q=q->pun;
    }
}

```

```

q=new elem;
q->inf=a;
q->pun=nullptr;
if(p0==nullptr)
    p0=q;
else
    p->punt=q;
}

```

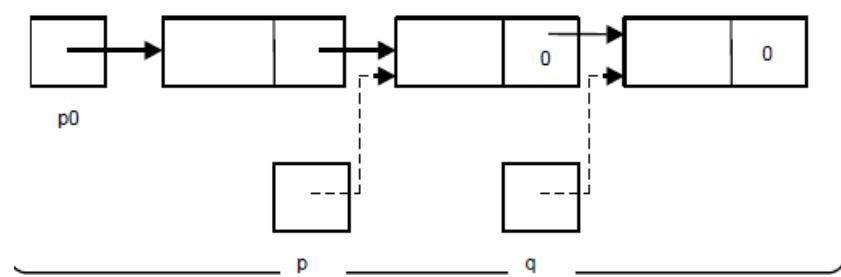


Estrazione dal fondo

```

bool estFondo(lista &p0, T &a){
    lista p=nullptr, q=nullptr;
    if(p0==nullptr)
        return false;
    q=p0;
    while(q->pun!=nullptr){
        p=q;
        q=q->pun;
    }
    a=q->inf;
    if(q==p0)
        p0=nullptr;
    else
        p->pun=nullptr;
    delete q;
    return true;
}

```



Dealloca lista

```

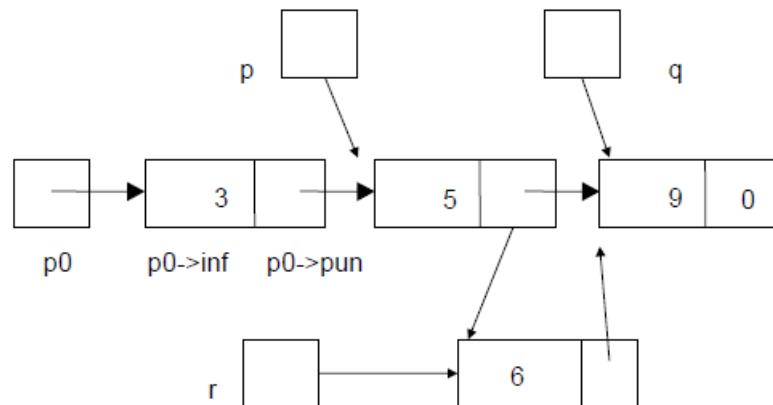
void deallocaLista(lista p0){
    lista p=p0;
    while(p!=nullptr){
        lista q=p->pun;
        delete p;
        p=q;
    }
}

```

```
}
```

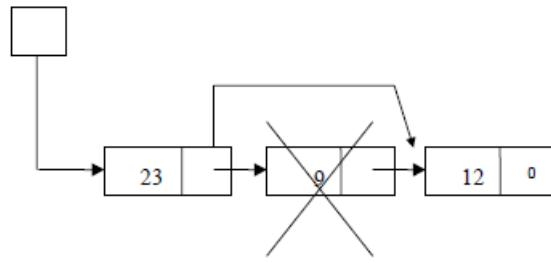
Inserimento (ordinato) in una lista ordinata

```
void insOrdinato(lista &p0, T a){
    lista p=nullptr, q, r;
    q=p0;
    while(q!=nullptr && q->inf<a){
        p=q;
        q=q->pun
    }
    r=new elem;
    r->inf=a;
    r->pun=q;
    if(q==p0)
        p0=r;
    else
        p->pun=r;
}
```



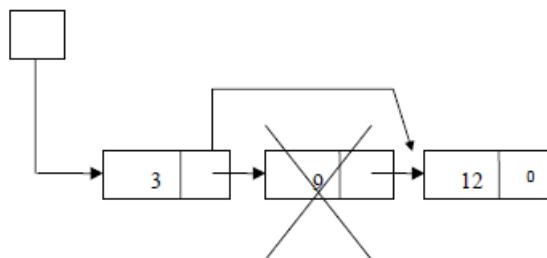
Estrazione di un elemento

```
bool estrazione(lista &p0, T a){
    lista p=nullptr, q;
    q=p0;
    while(q!=nullptr && q->inf!=a){
        p=q;
        q=q->pun;
    }
    if(q==nullptr)
        return false;
    if(q==p0)
        p0=q->pun;
    else
        p->pun=q->pun;
    delete q;
    return true;
}
```

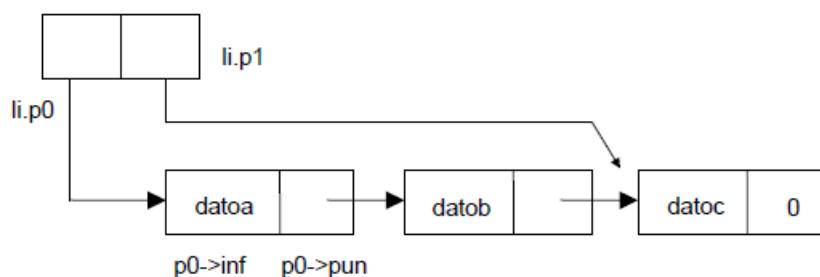


Estrazione di un elemento da una lista ordinata

```
bool estrazioneOrd(lista &p0, T a){
    lista p=nullptr, q;
    q=p0;
    while(q!=nullptr && q->inf<a){
        p=q;
        q=q->pun;
    }
    if((q==nullptr)|| (q->inf>a))
        return false;
    if(q==p0)
        p0=q->pun;
    else
        p->pun=q->pun;
    delete q;
    return true;
}
```



Liste con un puntatore ausiliario



```
struct lista_n{
    lista p0;
    lista p1;
};
lista_n crealistai(int n){
    lista p;
```

```

lista_n li={nullptr,nullptr};
if(n>=1){
    p=new elem;
    cin>>p->inf;
    p->pun=nullptr;
    li.p0=p;
    li.p1=p;
    for(int i=2;i<=n;i++){
        p=new elem;
        cin>>p->inf;
        p->pun=li.p0;
        li.p0=p;
    }
}
return li;
}
bool esttestat1(lista_n &li, T &a){
    lista p=li.p0;
    if(li.p0==nullptr)
        return false;
    a=li.p0->inf;
    li.p0=li.p0->pun;
    delete p;
    if(li.p0==nullptr)
        li.p1=nullptr;
    return true;
}
void insfondo1(lista_n &li, T a){
    lista p=new elem;
    p->inf=a;
    p->pun=nullptr;
    if(li.p0==nullptr){
        li.p0=p;
        li.p1=p;
    }
    else{
        li.p1->pun=p;
        li.p1=p;
    }
}

```

Visibilità

Nei programmi più complessi si usano tecniche di programmazione modulare.

Si suddivide il programma in diverse parti che vengono scritte e compilate separatamente.

Scambio di informazioni fra funzioni usando oggetti comuni.

Scope (visibilità) : campo di visibilità di un identificatore (parte di programma in cui l'identificatore può essere usato)

Regole di visibilità : permettono a più parti del programma di riferirsi ad una stessa entità; impediscono ad alcune parti di un programma di riferirsi ad un'entità

Unità di compilazione

Costituita da un file sorgente e dai file inclusi mediante #include

Se il file da includere non è di libreria, il suo nome va racchiuso tra virgolette

```

//header.h
const double PI_GRECO=3.14;
int f1(int);

```

```

int f2(int);

//header.cpp
#include "header.h"
int f1(int a){
    return PI_GRECO+a;
}
int f2(int a){
    return PI_GRECO-a;
}

//main
#include <iostream>
using namespace std;
#include "header.h"
int main(){
    cout<<PI_GRECO<<endl;
    cout<<f1(4)<<endl;
    cout<<f2(4)<<endl;
}

```

Il compilatore quando va a compilare il main sostituisce alla riga `#include "header.h"` tutto il contenuto del file header.h e compila quello.

Operatore di risoluzione di visibilità (::)

```

#include <iostream>
using namespace;
int i=1;
int main(){
    cout<<i<<endl; //1
    {
        int i=5; //visibilità locale
        cout<<::i<<'t'<<i<<endl; //1 5
    }
    int i=10;
    cout<<::i<<'t'<<i<<endl; //1 10
}
cout<<::i<<endl; //1
}

```

Spazio dei nomi

Aiutano il programmatore qualora volesse usare librerie diverse che svolgono le stesse funzionalità.

Insieme di dichiarazione e definizioni racchiuse tra parentesi graffe, ognuna delle quali introduce determinate entità dette membri.

```

//libreriaVettori1.h
namespace lib1{
    int stampa(int*, int);
    int ordina(int*,int);
}
//libreriaVettori1.cpp
#include "libreriaVettori1.h"
const int N_MAX=1000;
int stampa(int* vet,int N){ ... };
int ordina(int* vet,int N){ ... };

//libreriaVettori2.h
namespace lib2{
    int stampa(int* arr, int dim);
}

```

```

int ordina(int* arr,int dim);
}

//libreriaVettori2.cpp
#include "libreriaVettori2.h"
const int DIM_MAX=2000;
int stampa(int*,int){ ... };
int ordina(int*,int){ ... };

//main
#include "libreriaVettori1.h"
#include "libreriaVettori2.h"
int main(){
    int vettore[]={1,5,12};
    using namespace lib1;
    stampa(); //lib1::stampa()
    ordina(); //lib1::ordina()
    using namespace lib2;
    stampa(); //lib2::stampa()
    ordina(); //lib2::ordina()

    using namespace lib1;
    using namespace lib2;
    //ERRORE, se chiamo una funzione genera ambiguità
}

```

Collegamento

Un programma può essere formato da più unità di compilazione, che vengono sviluppate separatamente e poi collegate insieme.

Un identificatore ha collegamento interno se si riferisce a un'entità accessibile solo da quell'unità di compilazione. Ha collegamento esterno se si riferisce a un'entità accessibile anche da altra unità di compilazione.

Gli identificatori con visibilità locale hanno collegamento interno.

Gli identificatori con visibilità a livello di file hanno collegamento esterno (a meno che non siano dichiarati const).

In ciascuna unità in cui vengono usati devono essere dichiarati.

Viene solo dichiarato un oggetto se si usa la parola chiave extern, se non si usa viene anche definito.

Le funzioni vengono solo dichiarate se si specifica solo l'intestazione, si può usare extern, quando la definizione si trova in un altro file.

```

//file1.cpp
int a=1; //collegamento esterno
const int N=0; //collegamento interno
static int b=10; //collegamento interno
void f1(int a){ //a - collegamento interno
    int k; //k - collegamento interno
    /* ... */
}
static void f2(){ //collegamento interno
    /*...*/
}
struct punto{ //collegamento interno
    double x;
    double y;
};
punto p1; //collegamento esterno

//file2.cpp

```

```

extern int a; //solo dichiarazione
void f1(int); //solo dichiarazione
void f2(); //solo dichiarazione
void f3(); //solo dichiarazione
double f4(double, double); //definizione mancante, OK perchè non usata
int main(){
    cout<<a<<endl; //1      OK
    extern int b; //ERRORE
    f1(a); //OK
    f2(); //ERRORE
    f3(); //ERRORE
    punto p2; //ERRORE
}

```

Moduli

Parte di programma che svolge una funzionalità e che risiede su uno o più file.

Modulo servitore:

- offre risorse di varia natura, funzioni, variabili globali
- costituito di solito da un file .h e un file .cpp

Modulo cliente:

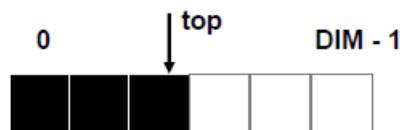
- usa risorse offerte dai moduli servitori
- viene scritto senza conoscere i dettagli della realizzazione dei moduli servitori

Separazione fra interfaccia e realizzazione:

- ha per scopo l'occultamento dell'informazioni
- semplifica le dipendenze tra i moduli
- permette la modifica della realizzazione di un modulo senza influenzare il funzionamento dei suoi clienti

Pila

Insieme ordinato di dati di tipo uguale, in cui è possibile effettuare operazioni di inserimento e di estrazione usando la politica LIFO (Last In First Out).



Se $\text{top} == -1$, la pila è vuota. Se $\text{top} == \text{DIM}-1$, la pila è piena.

```

//pila.h
typedef int T;
const int DIM=5;
struct pila{
    int top;
    T stack[DIM];
};
void inip(pila &);
bool empty(const pila &);
bool full(const pila &);

```

```

bool push(pila &, T);
bool pop(pila &, T &);
void stampa(const pila &);

//pila.cpp
#include "pila.h"
void inip(pila &pp){
    pp.top=-1;
}
bool empty(const pila &pp){
    if(pp.top== -1) return true;
    return false;
}
bool full(const pila &pp){
    if(pp.top==DIM-1) return true;
    return false;
}
bool push(pila &pp, T s){
    if(full(pp)) return false;
    pp.stack[++(pp.top)]=s;
    return true;
}
bool pop(pila &pp, T &s){
    if(empty(pp)) return false;
    s=pp.stack[(pp.top)--];
    return true;
}
void stampa(const pila &pp){
    cout<<"Elementi nella pila : "<<endl;
    for(int i=pp.top;i>=0;i--)
        cout<<'['<<i<<']'<<pp.stack[i]<<endl;
}

//main
#include "pila.h"
int main(){
    pila p;
    //...
}

```

Tipi classe

Nell'esempio sopra la struttura della pila è visibile ai moduli che usano istanze della pila. Non si riesce in questo modo a realizzare compiutamente il tipo di dato astratto.

Il C++ mette quindi a disposizione le classi.

Classe : costrutto che consente di dichiararsi un proprio tipo.

```

class NomeClasse{
private:
    //membri privati
public:
    //membri pubblici
};
//private può anche essere omesso, di default viene preso per privato

```

```

typedef int T;
const int DIM=5;
class pila{
    int top;      //membri dato privati
    T stack[DIM];
public:
    void inip();
    bool empty();

```

```

    bool full();
    bool push(T s);
    bool pop(T &s);
    void stampa();
};

int main(){
    pila st;
    //...
    st.top=-1; //ERRORE, i membri sono di default privati
}

```

Un membro di una classe può essere:

- un tipo (enum o struct)
- un campo dati
- una funzione
- una classe (diversa da quella a cui appartiene)

Solo funzioni membro pubbliche possono accedere ai membri privati.

Alla base delle classi c'è il concetto di information hiding.

```

//CLASSE COMPLESSO (parte reale, parte immaginaria)
//voglio riuscire a definire un nuovo tipo complesso con tutte le operazioni
class complesso{
    double re, im;
public:
    void iniz_compl(double r, double i){ re=r; im=i; }
    double reale(){ return re; }
    double immag(){ return im; }
    //...
    void scrivi(){ cout<<'('<<re<<','<<im<<')'<<endl; }
};

int main(){
    complesso c1, c2;
    c1.iniz_compl(1.0,-1.0); //inizializzazione
    c1.scrivi(); //(1,-1)

    c2.iniz_compl(10.0,-10.0); //inizializzazione
    c2.scrivi(); //(10,-10)

    complesso *cp=&c1;
    cp->scrivi(); //(1,-1)

    //accedo ai membri privati tramite funzioni pubbliche
    cout<<c1.reale()<<endl; //1.0
    cout<<c1.immag()<<endl; //-1.0
}

```

```

//CLASSE COMPLESSO con modulo e argomento
//Cambio implementazione ma l'utente non se ne accorge

#include <cmath>
class complesso{
    double mod, arg;
public:
    void iniz_compl(double r, double i){
        mod=sqrt(r*r+i*i);
        arg=atan(i/r);
    }
    double reale(){return mod*cos(arg);}
    double immag(){return mod*sin(arg);}
    //...
}

```

```

void scrivi(){
    cout<<'(<<reale()<<','<<immag()<<')';
}
};

int main(){
    complesso c1, c2;
    c1.iniz_compl(1.0,-1.0); //inizializzazione
    c1.scrivi(); //(1,-1)

    c2.iniz_compl(10.0,-10.0); //inizializzazione
    c2.scrivi(); //(10,-10)

    complesso *cp=&c1;
    cp->scrivi(); //(1,-1)

    //accedo ai membri privati tramite funzioni pubbliche
    cout<<c1.reale()<<endl; //1.0
    cout<<c1.immag()<<endl; //-1.0
}

```

Le funzioni membro possono anche essere definite esternamente usando l'operatore di risoluzione di visibilità

```

#include <cmath>
class complesso{
    double re, im;
public:
    void iniz_compl(double r, double i);
    double reale();
    double immag();
    //...
    void scrivi();
};
void complesso::iniz_compl(double r, double i){
    re=r;
    im=i;
}
double complesso::reale(){return re;}
double complesso::immag(){return im;}
//...
void complesso::scrivi(){
    cout<<'(<<reale()<<','<<immag()<<')';
}

```

Operazioni su oggetti classe

```

int main(){
    complesso c1;
    c1.iniz_compl(1.0,-1.0);
    //inizializzazione, ricopiatura membro a membro
    complesso c2=c1;
    complesso c3(c2); //solo se c3 e c2 sono dello stesso tipo

    c1.scrivi(); //(1,-1)
    c2.scrivi(); //(1,-1)
    c3.scrivi(); //(1,-1)

    complesso *pc1=new complesso(c1);
    pc1->scrivi(); //(1,-1)

    complesso *pc2=&c1;
    pc2->scrivi(); //(1,-1)

    //funziona anche con le variabili normali
    int i=7;
}

```

```
    int k(i);
}
```

Somma di due oggetti

```
complesso somma(compleSSo a, compleSSo b){
    compleSSo s;
    s.iniz_compl(a.reale()+b.reale(),a.immag()+b.immag());
    return s;
} //somma è una funzione globale, non membro della classe
//usa infatti solo funzioni pubbliche della classe, non
//può accedere ai membri privati
int main(){
    compleSSo c1, c2, c3;
    c1.iniz_compl(1.0,-1.0);
    c2=c1;
    c1.scrivi(); //(1.0,-1.0)
    c2.scrivi(); //(1.0,-1.0)

    c3=somma(c1,c2);
    c3.scrivi(); //(2.0,-2.0)
}
```

Puntatore this

Puntatore predefinito che esiste in tutte le funzioni membro della classe.

Puntatore che mantiene l'indirizzo dell'istanza della classe sulla quale è stata chiamata la funzione membro (*compleSSo * const this*).

Il suo indirizzo non può essere modificato.

Viene implicitamente passato ad ogni funzione membro come primo parametro.

```
class compleSSo{
    /*...
     * ...
     */
    compleSSo scala(double s){
        re*=s; //implicito this->re*=s
        im*=s; // this->im*=s;
        return *this;
    }
}
int main(){
    compleSSo c1;
    c1.iniz_compl(1.0,-1.0);
    c1.scala(2);
    c1.scrivi(); //(2,-2)

    compleSSo c2;
    c2.iniz_compl(1.0,-1.0);
    c2.scala(2).scala(2);
    c2.scrivi(); //(2,-2)
}
/*
la concatenazione non funziona perchè la seconda chiamata della scala
viene chiamata sulla variabile temporanea
*/
```

```
//Per far funzionare la concatenazione basta modificare il tipo di
//ritorno della funzione scala
class compleSSo{
    /*...*/
```

```

    complesso& scala(double s){
        re*=s; //implicito this->re*=s
        im*=s; // this->im*=s;
        return *this;
    }
}
int main(){
    complesso c1;
    c1.iniz_compl(1.0,-1.0);
    c1.scala(2);
    c1.scrivi(); //(2,-2)

    complesso c2;
    c2.iniz_compl(1.0,-1.0);
    c2.scala(2).scala(2);
    c2.scrivi(); //(4,-4)
}

```

Visibilità a livello di classe

Gli identificatori dichiarati dentro la classe sono visibili dal punto della loro dichiarazione fino alla fine della classe e in tutte le funzioni membro.

Se nella classe viene riusato un identificatore dichiarato all'esterno, la dichiarazione fatta nella classe nasconde quella più esterna.

All'esterno della classe possono essere resi visibili (con l'operatore di risoluzione di visibilità):

- le funzioni membro
- un tipo o un enumarato, se pubblici
- membri statici

L'operatore di visibilità non può essere usato per campi dati non statici.

```

class grafica{
public:
    enum colore{rosso, verde, blu};
};

class traffico{
public:
    enum colore{rosso, giallo, verde};
    colore stato();
};

traffico::colore traffico::stato()/*...*/
grafica::colore punto;
traffico::colore semaforo;
int main(){
    punto=grafica::rosso;
    semaforo=traffico::rosso;
}

```

Nella dichiarazione di una classe si può dichiarare un'altra classe (annidata). Per riferirsi ai membri della classe annidata esternamente alla classe che la contiene si devono usare due risolutori di visibilità. Nessuna delle due classi ha diritti di accesso alla parte privata dell'altra.

```

class A{
    class B{
        int x;
    public:
        void iniz_B(int);
    }
}

```

```

    int y;
public:
    void iniz_A();
}
void A::iniz_A(){y=0;}
void A::B::iniz_B(int n){x=n;}

void A::iniz_A(){x=3;} //ERRORE, A non può accedere alla parte privata di B
void A::B::iniz_B(int n){y=n;} //ERRORE, B non può accedere alla parte privata di A

```

Modularità e ricompilazione

L'uso delle classi permette di scrivere programmi in cui l'interazione tra moduli è limitata dalle interfacce.

Quando si modifica la parte dati di una classe, anche se gli altri moduli usano solo l'interfaccia, bisogna ricompilare i moduli.

Ai fini della ricompilazione non è appropriato separare una classe in interfaccia e struttura interna, ma piuttosto fare una divisione in file.

Un file di intestazione che contiene la dichiarazione della classe e deve essere incluso in tutti i moduli cliente.

Un file di realizzazione che contiene tutte le definizioni delle funzioni membro.

La modifica di un file di intestazione richiede la ricompilazione di tutti i file che lo includono.

La modifica di un file di realizzazione richiede solo la ricompilazione di questo file.

```

//complesso.h
class complesso{
    double re, im;
public:
    void iniz_compl(double r, double i);
    double reale();
    double immag();
    void scrivi();
    complesso& scala(double s);
};

//complesso.cpp
void complesso::iniz_compl(double r, double i){
    re=r;
    im=i;
}
double complesso::reale(){return re;}
double complesso::immag(){return im;}
void complesso::scrivi(){
    cout<<'(<<reale()<<','<<immag()<<')';
}
compleSSO& complesso::scala(double s){
    re*=s;
    im*=s;
    return *this;
}

//main
#include "compleSSO.h"
int main(){
    complesso c1;
    c1.iniz_compl(1.0,-1.0);
    c1.scala(2);
    c1.scrivi(); //((2,-2)

    complesso c2;

```

```

c2.iniz_compl(1.0,-1.0);
c2.scala(2).scala(2);
c2.scrivi(); //(4,-4)
}

```

Funzioni globali

Funzioni non membro che non possono accedere alla parte privata e non possono usare il puntatore this.

```

complesso somma(const complesso& a, const complesso& b){
    complesso s;
    s.iniz_compl(a.reale()+b.reale(),a.immag()+b.immag());
    return s;
}
int main(){
    complesso c1, c2, c3;
    c1.iniz_compl(1.0,-1.0);
    c2=c1;
    c1.scrivi(); //(1.0,-1.0)
    c2.scrivi(); //(2.0,-2.0)

    c3=somma(c1,c2);
    c3.scrivi(); //(3.0,-3.0)
}

```

Costruttori

Funzione membro il cui nome è il nome della classe. Se viene definita viene invocata automaticamente tutte le volte che viene creata un'istanza di classe.

Serve per portare in uno stato consistente l'istanza della classe e (nel caso) allocare memoria.

```

class complesso{
    /*...*/
    complesso(double r, double i);
};

compleSSO::compleSSO(double r, double i){
    re=r;
    im=i;
}

int main(){
    complesso c1(1.0, -1.0);
    c1.scrivi(); //(1.0,-1.0)

    complesso c2; //ERRORE, non esiste complesso()
    complesso c3(1); //ERRORE, non esiste complesso(int)
}

```

Una volta che la classe è dotata di un costruttore, l'istanza della classe è impossibile da dichiarare senza prima passare da esso.

Costruttori default

Per definire oggetti senza inizializzatore occorre definire un costruttore di default che può essere chiamato senza argomenti.

```

//MECCANISMO DELL'OVERLOADING
class complesso{

```

```

    double re, im;
public:
    complesso();
    complesso(double r ,double i);
    double reale();
    double immag();
    void scrivi();
};

//complesso.cpp
complesso::complesso(){ //costruttore di default
    re=0;
    im=0;
}
complesso::complesso(double r,double i){
    re=r;
    im=i;
}

//main
int main(){
    complesso c1(1.0,-1.0);
    c1.scrivi(); //(1,-1)
    complesso c2;
    complesso c3(3.0); //ERRATO
}

```

```

//MECCANISMO DEGLI ARGOMENTI DI DEFAULT
class complesso{
    double re, im;
public:
    complesso();
    complesso(double r=0,double i=0);
    double reale();
    double immag();
    void scrivi();
};

//complesso.cpp
complesso::complesso(double r,double i){
    re=r;
    im=i;
}

//main
int main(){
    complesso c1(1.0,-1.0); //(1,-1)
    complesso c2;
    complesso c3(3.0); //(3,0)
    complesso c4=3.0; //(3,0)
}

```

I due metodi usati sopra non possono essere usati contemporaneamente.

Costruttori per allocare memoria dinamica

Spesso il costruttore ha bisogno di allocare memoria libera per alcuni oggetti da costruire

```

//stringa.h
class stringa{
    char *str;
public:
    stringa(const char s[]);
    /*...*/
};

```

```

//stringa.cpp
stringa::stringa(const char s[]){
    str=new char[strlen(s)+1];
    strcpy(str,s);
}

//main
int main(){
    stringa inf("Fondamenti di Programmazione");
    /*...*/
}

```

Riesco a costruire un oggetto di lunghezza precisa (senza spreco o senza non riuscire a farcelo entrare) per la mia stringa.

Costruttori per oggetti dinamici

I costruttori definiti per una classe vengono chiamati implicitamente anche quando un oggetto viene allocato dinamicamente.

```

int main(){
    complesso *pc1=new complesso(3.0,4.0);
    complesso *pc2=new complesso(3.0);
    complesso *pc3=new complesso;
    /*...*/
}

```

Distruttori

Funzione membro che viene invocata automaticamente quando un oggetto termina il suo tempo di vita. Non ha argomenti. Da definire obbligatoriamente quando il costruttore alloca memoria dinamica.

```

class stringa{
    char *str;
public:
    stringa(const char s[]);
    ~stringa(); //distruttore
};
stringa::~stringa(){
    delete[] str;
}
int main(){
    stringa *ps=new stringa("Fondamenti");
    delete ps; //verrà chiamato il distruttore
}
//anche se non faccio la delete di ps, il distruttore verrà chiamato
//lo stesso quando esco dal blocco

```

Regole di chiamata

Costruttori

1. Per gli oggetti statici, all'inizio del programma
2. Per gli oggetti automatici, quando viene incontrata la definizione
3. Per gli oggetti dinamici, quando viene incontrato *new*

4. Per gli oggetti membri di altri oggetti, quando questi ultimi vengono costruiti

Distruttori

1. Per gli oggetti statici, alla fine del programma
2. Per gli oggetti automatici, all'uscita dal blocco in cui sono definiti
3. Per gli oggetti dinamici, quando viene incontrato *delete*
4. Per gli oggetti membri di altri oggetti, quando questi ultimi vengono distrutti

Costruttori di copia

Costruttore che agisce fra due oggetti della stessa classe effettuando una ricopiatura membro a membro dei campi dati.

Viene invocato in 3 casi :

1. Quando un oggetto classe viene inizializzato con un altro oggetto della stessa classe
2. Quando un oggetto classe viene passato ad una funzione come argomento valore
3. Quando una funzione restituisce come valore un oggetto classe (mediante return)

```
complesso(const complesso &c){
    re=c.re;
    im=c.im;
}
//se passassi c per valore ci sarebbe una ricorsione infinita
```

Per tutte le classi che non usano memoria dinamica, va benissimo l'uso del costruttore di copia predefinito (punto 1 sopra)

```
int main(){
    complesso c1(3.2,4);

    complesso c2(c1); //chiamata costr di copia punto 1 sopra

    complesso c3=c1; //chiamata costr di copia punto 1 sopra

    //visto che la classe complesso non alloca memoria dinamica
    //non c'è bisogno di ridefinire il costr di copia
}
```

Se il costr di copia non viene ridefinito, viene usato quello predefinito.

Per impedirne l'uso, occorre inserire la sua dichiarazione nella parte privata della classe stessa senza alcuna ridefinizione.

Costruttore di copia per allocare memoria

```
class stringa{
    char *str;
public:
    stringa(const char s[]);
    ~stringa();
    stringa(const stringa &);

    //ridefinizione costr di copia
```

```

stringa::stringa(const stringa &s){
    str=new char[strlen(s.str)+1];
    strcpy(str,s.str);
}

```

Funzioni friend

Una funzione è friend (amica) di una classe se nella sua dichiarazione, appare nelle dichiarazione di tale classe. Una funzione friend può accedere ai membri pubblici e privati della classe.

```

class complesso{
    double re,im;
public:
    double reale();
    double immag();
    friend complesso somma(const complesso &a, const complesso &b);
}
compleSSO somma(const complesso &a, const complesso &b){
    complesso s(a.re+b.re,a.im+b.im);
    return s;
}
int main(){
    complesso c1(3.2,4);
    complesso c2(c1);
    complesso c3=somma(c1,c2);
}

```

La funzione somma è una funzione globale, ma che può accedere ai membri privati della classe re e im, senza dover usare le due funzioni reale() e immag().

Anche un'intera classe può essere amica di un'altra classe.

Array di oggetti classe

Un array può avere come elemento oggetti classe, se nella classe sono definiti costruttore e distruttore, questi vengono richiamati per ogni elemento dell'array

```

int main(){
    complesso vc[5]={compleSSO(1.1,2.2),
                     compleSSO(1.5,1.5), compleSSO(4.1,2.5)};
    for(int i=0;i<5;i++){
        vc[i].scrivi();
        cout<<endl;
    }
}
/*
(1.1,2.2)
(1.5,1.5)
(4.1,2.5)
(0,0)
(0,0)
*/

```

Se la lista di costruttori non è completa, nella classe deve essere definito un costruttore default, che viene richiamato per gli elementi non esplicitamente dichiarati.

Overloading di operatori

Tipo di dato astratto : può richiedere che vengono definite delle operazioni tipo somma, prodotto...

Necessaria la ridefinizione degli operatori.

La ridefinizione di un operatore ha la forma di una definizione di funzione, il cui identificatore è costituito dalla parola chiave *operator* seguita dall'operatore che si vuole ridefinire, e ogni occorrenza dell'operatore equivale ad una chiamata alla funzione.

```
class complesso{
    double re,im;
public:
    complesso(double r=0,double i=0);
    complesso operator+(const complesso&);
}
compleSSO complesso::operator+(const complesso &x){
    complesso x(re+x.re,im+x.im);
    return x;
}
//definita come funzione membro della classe
int main(){
    complesso c1(3.2,4);
    complesso c2(c1);
    complesso c3=c1+c2; //c3=c1.operator+(c2)
}
```

```
class complesso{
    double re,im;
public:
    complesso(double r=0,double i=0);
    friend complesso operator+(const complesso&);
}
compleSSO complesso::operator+(const complesso &x){
    complesso x(re+x.re,im+x.im);
    return x;
}
//definita come funzione globale, friend della classe
int main(){
    complesso c1(3.2,4);
    complesso c2(c1);
    complesso c3=c1+c2; //c3=operator+(c1,c2)
}
```

Overloading di operatori di uscita

```
ostream& operator<<(ostream &os, const complesso &c){
    os<<'('<<c.reale()<<','<<c.immag()<<')';
    return os;
}
int main(){
    complesso c1(1.0,-2.0);
    cout<<c1; //(1.0,-2.0)
}
```

Overloading di operatori di ingresso

```
istream& operator>>(istream &is, complesso &z){
    double re=0, im=0;
    char c=0;
    is>>c;
    if(c!='(')
        is.clear(ios::failbit);
    else{
```

```

    is>>re>>c;
    if(c!=',')
        is.clear(ios::failbit);
    else{
        is>>im>>c;
        if(c!=')')
            is.clear(ios::failbit);
    }
}
z=complesso(re,im);
return is;
}

```

Oveloading di operatori di assegnamento

1. deallocate la memoria dinamica dell'operando a sinistra
2. allocare la memoria della dimensione uguale all'operando destro
3. copiare i membri dato e gli elementi dello heap

ALIASING : argomento implicito uguale all'argomento esplicito

```

class stringa{
    char *str;
public:
    stringa(const char s[]);
    ~stringa();
    stringa(const stringa &);

    stringa& operator=(const stringa &s);
};

stringa& stringa::operator=(const stringa &s){
    if(this!=&s){      //controllo aliasing
        delete[] str;
        str=new char[strlen(s.str)+1];
        strcpy(str,s.str);
    }
    return *this;
}

//OTTIMIZZAZIONE
stringa& stringa::operator=(const stringa &s){
    if(this!=&s){
        if(strlen(str)!=strlen(s.str)){
            delete[] str;
            str=new char[strlen(s.str)+1];
        }
        strcpy(str,s.str);
    }
    return *this;
}

```

Se non faccio un controllo anti aliasing potrei avere problemi seri, visto che potrei avere due puntatori che puntano allo stesso oggetto.

Operatori che si possono ridefinire

- operatore di visibilità (::)
- operatore di selezione di membro (.)
- operatore selezione di membro attraverso un puntatore a membro (.*)

Controlli sugli stream

Lo stato di uno stream è la configurazione di bit, detti bit di stato. Lo stato è good se tutti i bit di stato valgono 0.

- Bit fail - messo a 1 quando si verifica un errore recuperabile
- Bit bad - discrimina se l'errore è recuperabile (0) o no (1)
- Bit eof - posto ad 1 quando viene letta la marca di fine stream

I bit possono essere esaminati con le funzioni seguenti:

- fail() - restituisce true se almeno uno dei due bit fail e bad sono ad 1
- bad() - restituisce true se il bit bad è ad 1
- eof() - restituisce true se tutti il bit eof è ad 1
- good() - restituisce true se tutti i bit sono a 0

I bit possono essere manipolati tramite la funzione clear() (che ha per argomento default 0). Ma tramite ios::failbit, ios::badbit, ios::eofbit passati come argomento posso settare i rispettivi bit ad 1.

Manipolazione dei file

Stream associati ai file: gestiti da una apposita libreria, occorre includere <fstream>

Funzione open():

- associa uno stream ad un file
- apre lo stream secondo alcune modalità
 - lettura ⇒ ios::in
 - scrittura ⇒ ios::out
 - scrittura alla fine ⇒ ios::out | ios::app
- il nome del file viene specificato come stringa

Stream aperto in lettura:

- il file deve essere già presente
- il puntatore si sposta sulla prima casella

Stream aperto in scrittura:

- il file, se non presente, viene creato
- il puntatore si posiziona sulla prima casella
- il file, se presente, viene sovrascritto

Stream aperto in append:

- il file associato, se non presente viene creato
- il puntatore si sposta alla fine dello stream, eventuali dati presenti nel file non vengono sovrascritti

Scrittura su file : file<<10

Lettura da file : file>>x

Funzione close():

- chiude uno stream aperto una volta che è stato usato

Alla fine del programma tutti gli stream aperti vengono automaticamente chiusi.

```
#include <fstream>
#include <iostream>
using namespace std;
int main(){
    fstream ff;
    int num;
    ff.open("esempio",ios::out);
    if(!ff){
        cerr<<"Il file non può essere aperto"=<<endl;
        exit(1);
    }
    for(int i=0;i<4;i++)
        ff<<i<<endl;
    ff.close();
    ff.open("esempio",ios::in);
    if(!ff){
        cerr<<"Il file non può essere aperto"=<<endl;
        exit(1);
    }
    while(ff>>num)
        cout<<num<<'t';
    cout<<endl;
    return 0;
}
```

Si possono usare due keyword diverse per dichiarare file in lettura o in scrittura: ifstream, ofstream.

Espressioni letterali

```
class complesso{
    double re, im;
public:
    complesso(double r=0, double i=0);
    complesso operator+(const complesso &x); //...
};
compleSSO complessP::operator+(const complesso &x){
    complesso z(re+x.re,im+x.im);
    return z;
}
int main(){
    complesso c1(3.2,4), c2;
    c2=complesso(0.3,3.0)+c1;
    c2.scrivi(); // (3.5,7)
}
```

Costanti e riferimenti nelle classi

```
class complesso{
    const double re;
    double im;
public:
    complesso(double r=0, double i=0);
    complesso operator+(const complesso&);
```

```

};

complesso::compleasso(double r, double i) : re(r){
    im=i;
}

complesso::compleasso(const compleasso &c) : re(c.re){
    im=c.im;
}

```

L'inizializzazione del membro costante avviene solo tramite una lista di inizializzazione.

```

class compleasso{
    double &re;
    double im;
public:
    //...
};

int main(){
    double my_re=5.5;
    compleasso c1(my_re,7.0);
    c1.scrivi(); //(5.5,7.0)
    my_re=-11.1;
    c1.scrivi(); //(-11.1,7.0)
}

```

Membro classe all'interno di classi

In una classe principale possono essere presenti membri di tipo classe diverso dalla classe principale.

```

class record{
    stringa nome, cognome;
public:
    //...
}

```

Quando viene dichiarato un oggetto della classe principale:

1. vengono chiamati i costruttori delle classi secondarie, nell'ordine in cui queste compaiono
2. viene eseguito il corpo del costruttore della classe principale

Quando un oggetto della classe principale viene distrutto:

1. viene eseguito il corpo del distruttore della classe principale
2. vengono richiamati i distruttori delle classi secondarie nell'ordine inverso in cui compaiono

```

class record{
    stringa nome, cognome;
public:
    record(const char n[], const char c[]);
    //...
};

record::record(const char n[], const char c[]) : nome(n), cognome(c){
    //...
}

int main(){
    record pers("Mario","Rossi");
}

```

Usi della parola **static**

Primo uso - regole di collegamento

Se dentro ad uno spazio dei nomi facciamo

```
static int a=5;
```

La variabile *a* avrà collegamento interno, quindi non sarà visibile fuori.

Secondo uso - variabili locali di una funzione per cambiare tempo di vita

La variabile static dichiarata dentro una funzione mantiene il suo valore ad ogni chiamata

Terzo uso - utilizzo nelle classi

Membri dato

```
class entity{  
    int dato;  
public:  
    entity(int n);  
    static int cont;  
};  
int entity::cont=0;  
entity::entity(int n){  
    cont++;  
    dato=n;  
}
```

Conta quante istanze di entity sono state chiamate

Membri funzione

```
class entity{  
    int dato;  
    static int cont;  
public:  
    entity(int n);  
    static int numero();  
};  
int entity::numero(){return cont;}
```

Possono accedere in scrittura e in lettura solo ai membri dato statici e non ai membri dato della specifica istanza. Non hanno il puntatore this, non possono accedere alla singola istanza.

Funzioni const

Funzioni che non possono modificare l'oggetto a cui sono applicate.

```
class complesso{  
    double re, im;
```

```

public:
    complesso(double r=0, double i=0);
    double reale();
    double immag();
    //...
};

int main(){
    const complesso c1(3.2,4);
    cout<<c1.reale()<<endl;
    //ERRORE, c1 costante
}

```

```

class complesso{
    double re, im;
public:
    complesso(double r=0, double i=0);
    double reale() const;
    double reale();
    double immag() const;
    //...
};

double complesso::reale() const {return re;}
double complesso::immag() const {return im;}
int main(){
    const complesso c1(3.2,4);
    complesso c2(7.8,5);
    cout<<c1.reale()<<endl; //viene chiamata reale() const
    cout<<c2.reale()<<endl; //viene chiamata reale()
}

```

Conversione mediante costruttori

Un costruttore che può essere chiamato con un solo argomento viene usato per convertire valori del tipo dell'argomenti nel tipo della classe a cui appartiene il costruttore.

```

class complesso{
    double re, im;
public:
    complesso(double r=0, double i=0);
    complesso operator+(const complesso &x);
};

int main(){
    complesso c1(3.2,4), c2;
    c2=c1+2.5;
    //costruttore trasforma il reale 2.5 nel complesso (2.5,0)
    c2.scrivi(); //(5.7,4)
}

```

Può essere applicato anche per concatenare stringhe.

Preprocessore

Parte del compilatore che elabora il testo del programma prima dell'analisi lessicale e sintattica. Espande i simboli definiti dall'utente secondo le loro definizioni. Include o esclude parti di codice dal testo che verrà compilato.

Queste operazioni sono controllate dalle direttive per il preprocessore (il primo carattere è #).

Grazie a queste direttive possiamo anche creare una certa porzione di codice che viene compilata se verificata una certa condizione.

Vengono usate anche per evitare di includere due volte la stessa libreria in un file.

Cenni sulla ricorsione

Funzione che invoca se stessa.

Ci deve essere un caso base in cui la funzione termina, altrimenti andrebbe avanti all'infinito.

```
int fatt(int n){  
    if(n==0) return 1;  
    return n*fatt(n-1);  
}
```

Unioni

Dichiarate e usate con la stessa sintassi delle strutture. Si usa la parola union al posto di struct.

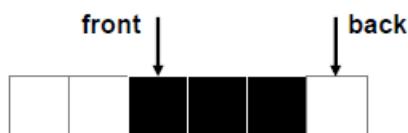
Rappresentano un'area di memoria che in tempi diversi può contenere dati di tipo differente.

```
union foi{  
    float f;  
    int i;  
}  
int main(){  
    foi v;  
    v.i=7;  
    cout<<v.i; //7  
    v.f=3.5;  
    cout<<v.f; //3.5  
    cout<<v.i; //NON STAMPA 7  
}  
//sono 4 byte usati o per float o per int, non insieme  
//minimizza l'occupazione di memoria
```

Coda

Tipo di dato astratto in cui vige la politica FIFO (First In First Out).

Il vettore viene usato in maniera circolare, gli indici non possono mai coincidere.



```
typedef int T;  
const int DIM=5;  
struct coda{  
    int front, back;  
    T queue[DIM];  
};  
void inic(coda &cc){  
    cc.front=cc.back=0;
```

```

}

bool empty(const coda &cc){
    if(cc.front==cc.back) return true;
    return false;
}
bool full(const coda &cc){
    if(cc.front==(cc.back+1)%DIM) return true;
    return false;
}
bool insqueue(coda &cc, T s){
    if(full(cc)) return false;
    cc.queue[cc.back]=s;
    cc.back=(cc.back+1)%DIM;
    return true;
}
bool esqueue(coda &cc, T &s){
    if(empty(cc)) return false;
    s=cc.queue[cc.front];
    cc.front=(cc.front+1)%DIM;
    return true;
}
void stampa(const coda &cc){
    for(int i=cc.front; i%DIM!=cc.back; i++)
        cout<<cc.queue[i%DIM]<<endl;
}

```