

FONDAMENTI DI PROGRAMMAZIONE

Informatica < { scienza che rappresenta l'elaborazione delle informazioni
studio sistematico degli algoritmi (→ sequenza precisa e finita di operazioni)

OPERAZIONI:

- SEQUENZIALI → eseguono un'operazione alla volta
- CONDIZIONALI → prima di eseguire un'operazione verifica una condizione
- ITERATIVE → ripetono un blocco di operazioni finché non si verifica una condizione

PROPRIETA' di un ALGORITMO

- Eseguibilità → ogni operazione deve poter essere eseguita dall'esecutore
- Non Ambiguità → ogni azione deve essere univoca
- Finitezza → il numero di operazioni deve essere finito
- Correttezza → deve arrivare alla soluzione senza produrre errori
- Efficienza → deve arrivare alla soluzione nel modo più veloce

Due algoritmi sono EQUIVALENTI quando hanno: - stesso dominio d'ingresso
- stesso dominio d'uscita
- stesso risultato per valori d'ingresso uguali

Tuttavia, possono avere diversa efficienza.

PROGRAMMAZIONE

PROGRAMMA = formula testuale di un algoritmo

Fasi della risoluzione di un problema

- individuare un procedimento risolutivo
- scomposizione in un insieme di azioni
- traduzione dell'algoritmo in un linguaggio comprensibile al calcolatore

CARATTERISTICHE: - SINTASSI: regole formali
- SEMANTICA: significati delle frasi

La GRAMMATICA BNF (Backus-Naur Form) specifica le regole di produzione:

1. Dato un simbolo non terminale X può essere descritto in una sequenza di simboli
2. X può essere sviluppato come uno qualunque dei simboli della sequenza (one of)

METALINGUAGGIO = è un linguaggio che si utilizza per scrivere la grammatica di un altro linguaggio

costrutti → opzionali
 → sequenziali

PROGRAMMA C++ SORGENTE = sequenza di caratteri HASH salvata sul disco
così strutturato: come un file di testo (nome.cpp)

```
int main() {
    istruz - seq;  → sequenza di istruzioni
                  oppure
                  sequenza singola (one of)
}
```

In ogni programma ci deve sempre essere questo pezzo:

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

libreria direttiva

return serve a terminare il programma

N.B. Scrivere un programma sintatticamente corretto, non implica che faccia quello per cui è stato scritto

Esistono due tipi di APPROCCIO:

1) COMPILATO

- EDITING (scrittura e salvataggio del testo)
 - COMPILAZIONE E LINKING (traduzione del programma sorgente in file eseguibile. Si compone di ANALISI e TRADUZIONE)
 - ↓
 - prog. sorgente generazione del codice
 - lessicale ottimizzaz. del codice
 - sintattica
 - ESECUZIONE
- (C, C++, Fortran, Rust, Go, ...)

cioè si occupa anche di collegare altri file

2) INTERPRETATO

- EDITING
 - INTERPRETAZIONE (ANALISI programma sorgente, lessicale, sintattica)
 - ESECUZIONE
- (Python, Matlab, Javascript, ...)

Programma C++ = sequenza di parole (TOKEN)
(che possono essere separate da spazi bianchi - white spaces)

ELEMENTI LESSICALI:

- IDENTIFICATORE della variabile (non possono avere lo stesso nome)
- PAROLE CHIAVE (tipi, cicli, ecc...)
- ESPRESSIONI LETTERALI (valori delle variabili)
- OPERATORI (+, -, *, /, %, ...)
- SEPARATORE (;)

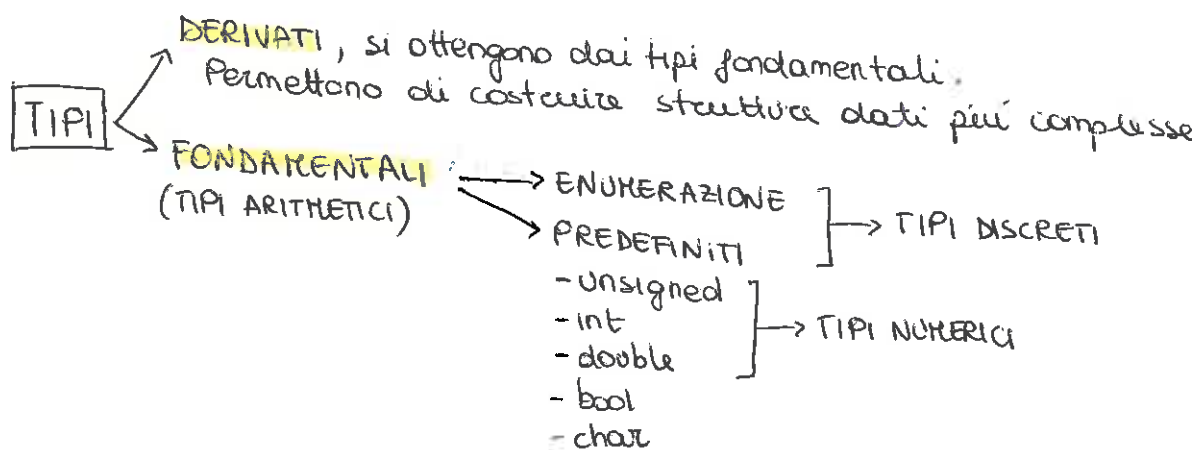
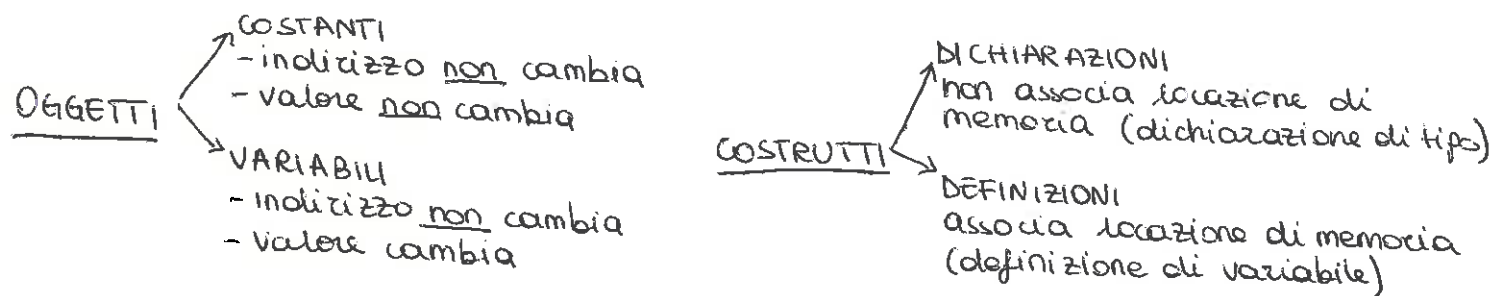
N.B. Il C++ è case sensitive. Distingue le maiuscole dalle minuscole.

PROPRIETA' DEGLI OPERATORI

1. **POSIZIONE**
 - PREFISSO (++x)
 - POSTFISSO (x++)
 - INFISSO (x+y)
2. **ARIETA'** (NUMERO DI ARGOMENTI)
3. **PRECEDENZA** (O PRIORITA') cioè vengono eseguiti quelli con priorità più alta
4. **ASSOCIATIVITA'** su operatori con stessa priorità:
 - da sx (+)
 - da dx (=)

OGGETTI = gruppo di celle consecutive

MEMORIA = insieme di celle di dimensione 1 byte ciascuna



TIPO INTERO

Una variabile inizializzata di tipo intero può assumere solo valori reali.

Una variabile ^{non}inizializzata prende un valore casuale anche molto grande.

Per assegnare un valore è necessario scrivere $i=2$ oppure $i(2)$.

OPERAZIONI SU INTERI: somma, sottrazione, moltip, div, resto (%)

→ Se mettiamo come prefisso lo 0 (zero) allora il numero seguente verrà interpretato in base 8. Esempio: `int ott = 011 //ott=9`

→ Se mettiamo come prefisso 0x il numero seguente verrà interpretato in base 16. Esempio: `int esa = 0xF //esa=15`

N.B. Se N è il numero di bit impiegati per rappresentare gli interi, i valori vanno da -2^{N-1} a $2^{N-1}-1$. Solitamente $N=32$.

TIPO UNSIGNED

Qualunque numero senza segno (NATURALE) seguito da U oppure u.

Occupi sempre 32 bit. Se short ne occupa 16 e se è long 64.

Se a questo tipo viene assegnato un valore numerico negativo viene inserita ugualmente la stringa di bit ma viene convertito in un positivo.

N.B. Se N è il numero di bit impiegati per rappresentare gli unsigned, i valori vanno da 0 a 2^N-1 .

OPERATORI BIT A BIT

- OPERAZIONI LOGICHE

OR |
AND &
OR ESCLUSIVO ^
COMPLEMENTO ~

- OPERATORI DI TRASLAZIONE

SINISTRA <<
DESTRA >>

essi lavorano bit a bit, non sul valore della variabile

TIPO BOOLEANO

I valori sono costanti predefinite FALSE o TRUE, codificati rispettivamente 0 e 1.

Operazioni su booleani:

- OR (||)
- AND (ampersand)
- NOT (!)

Viene memorizzato in 8 bit.

N.B. `i=10; 4<i<7; // TRUE`
prima viene fatto `4<i` che restituisce TRUE.
Poi `1(TRUE)<7` che è nuovamente TRUE.

OPERATORI DI CONFRONTO

5

$\begin{matrix} == & != \\ > & >= \\ < & <= \end{matrix} \left\{ \begin{array}{l} \text{Restituiscono} \\ \text{un booleano} \end{array} \right.$

Gli operatori di CONFRONTO hanno priorità più alta rispetto a quelli di UGUAGLIANZA.

STRUTTURA DI UN PROGRAMMA

Dopo `int main()` vengono scritte delle istruzioni:

- ISTRUZIONI STRUTTURATE

Consentono di specificare azioni complesse

- ISTRUZIONI COMPOSITE

Trasforma una qualunque sequenza di istruzioni in una singola istruzione grazie all'uso di parentesi graffe

- ISTRUZIONI CONDIZIONALI

Come IF e SWITCH

ISTRUZIONE IF

Si può trovare in 3 modi:

- IF (condizione)
- IF (cond)
ELSE
- IF (cond)
ELSE IF (cond) ...
ELSE

N.B. Se nell'IF mettiamo come ~~non~~ condizione una variabile restituisce sempre TRUE a meno che non sia uguale a 0.

CONDIZIONE = una qualunque espressione ~~data~~ grazie alla quale si possa stabilire un valore di verità

SWITCH e BREAK

switch (expression)

{
 case_alternative:
 istruzioni;
 break; —————> si può anche non mettere.
 In questo caso eseguirebbe
 a cascata fino al default.

default:

—————> Se il valore non si trova in nessuna alternativa si esegue il default. L'etichetta default è UNICA.

Per fare in modo che in due casi si eseguano le stesse istruzioni si scrive:
case 1: case 2:

RAPPRESENTAZIONE DELL'INFORMAZIONE

In un calcolatore, l'informazione viene rappresentata da una sequenza di bit (Binary digit). Un bit può assumere solo due valori: 0 o 1.
 - Una stessa sequenza può rappresentare informazioni diverse.

RAPPRESENTAZIONE DEL TESTO

Si utilizza la codifica ASCII (American Standard Code Information Interchange). Di solito la codifica è su 7 bit perché il primo è sempre messo a 0 (128 caratteri). Il C++ supporta la codifica estesa in cui ogni carattere richiede 8 bit (256 caratteri).
 N.B. I browser utilizzano la codifica unicode in cui un carattere = 16 bit.

RAPPRESENTAZIONE NUMERI NATURALI

(da 0 a 9)

La base utilizzata comunemente è la base 10, mentre nel linguaggio macchina viene utilizzata la base 2 (0, 1).

Base 10 esempio: $(123)_{10} = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1$

Base 2 esempio: $(11001)_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 + 0 + 1 \cdot 2^0$

Altre basi:

Base 5 $\rightarrow (0, 1, 2, 3, 4)$

Base 8 $\rightarrow (0, 1, 2, 3, 4, 5, 6, 7)$

Base 16 $\rightarrow (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)$

FORMULA DELLA SOMMATORIA

$$N = \sum_{i=0}^{p-1} a_i \beta^i = a_{p-1} \beta^{p-1} \dots a_1 \beta^1 + a_0$$

\downarrow numero \downarrow base

$$\left[\begin{array}{l} p = \text{posizione} \\ a_0 = \text{cifra meno significativa} \\ a_{p-1} = \text{cifra più significativa} \end{array} \right]$$

TEOREMA FONDAMENTALE DELLA RAPPRESENTAZIONE DEI NUMERI NATURALI

Per ogni numero naturale N esiste una sola sequenza $a_{p-1} \dots a_0$

CONVERSIONI

- da Base 2 a Base 10

$$(1101)_2 \rightarrow 1 \cdot 2^3 + 1 \cdot 2^2 + 0 + 1 \cdot 2^0 = 8 + 4 + 1 = (13)_{10}$$

- da Base 10 a Base 2. Si usa il procedimento DIV & MOD (che può essere usato per tutte le basi)

N=23

Q	R
23	1
11	1
5	1
2	0
1	1
0	

$$(23)_{10} = (10111)_2$$

PER I NATURALI, L'INTERVALLO DI VALORI È $[0, 2^{p-1}]$

Il calcolatore lavora con un numero finito di bit, questo comporta che alcune somme danno luogo a degli OVERFLOW (si risolve aggiungendo un bit). In C++ però, viene ignorato il bit di resto e questo provoca un errore.

N.B. Per convertire un numero da una base x a una base y si usa come "ponte" la base 10. Ma se x e y sono multipli di 2 si usa la base 2.

RAPPRESENTAZIONE MODULO E SEGNO

Useremo il primo bit per il segno e p-1 bit per il valore assoluto

$A = (\text{segno}_a, \text{ABS}_a) \rightarrow$ questo provoca il conteggio dello zero due volte (0 e -0)

Esempio su 4 bit

$$+3 \rightarrow \text{segno} = 0$$

$$\text{ABS}(3) = 011$$

$$\rightarrow (0, 011)_2$$

$$-3 \rightarrow \text{segno} = 1$$

$$\text{ABS}(3) = 011$$

$$\rightarrow (1, 011)_2$$

L'intervallo di valori è $[-(2^{p-1}-1), +(2^{p-1}-1)]$

RAPPRESENTAZIONE COMPLEMENTO A 2

Con $a \geq 0 \rightarrow \text{ABS}_a$

$a < 0 \rightarrow (\text{due}^p - \text{ABS}_a)$

Decodifica:

- se il primo bit è a zero il numero è positivo, negativo invece se è 1

Su p bit, l'intervallo è $[-2^{p-1}, +2^{p-1}-1]$

Tramite il complemento a 2 possiamo sommare tutti i numeri come se fossero naturali, però bisogna stare attenti all'OVERFLOW.

OPERATORI COSTANTI

Hanno come parola chiave "const". Non si può modificare il contenuto ed è richiesta l'inizializzazione.

OPERATORE sizeof

Restituisce il numero di celle che la variabile occupa in memoria.

- bool = 1 bit
- char = 1 byte
- short int = 2 byte $\rightarrow \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int})$
- int = 4 byte
- float = 4 byte
- long int = 8 byte $\rightarrow \text{sizeof}(\text{long}) \geq \text{sizeof}(\text{int})$
- double = 8 byte
- long double = 12 byte

ISTRUZIONI RIPETITIVE

- **WHILE** verifica una condizione e finché è vera esegue le istruzioni all'interno delle parentesi {}, quando la condizione diventa falsa si interrompe.
Di solito infatti viene usato un contatore.

TIPO ENUMERATO

Esso è un tipo che può definire l'utente ed è un insieme di valori costanti (di solito non numerici).

OPERAZIONI: stesse del tipo intero.

DICHIARAZIONE:

enum - Nome { cost 1, ... }; oppure enum { cost 1, ... } Nome;

I valori dell'enumerato sono convertiti in interi (primo valore = 0) ma un intero non può essere convertito in enumerato.

Un enumerato, essendo un intero, occupa 4 byte, quanto un int.

↓
si può scegliere anche un altro valore se si vuole

enum - Nome { costante = N, ... }

gli enumerati che seguono valgono N+1, ecc.

RAPPRESENTAZIONE CON BIAS

9

$$A = a + (2^{P-1} - 1)$$

↓
si suppone
che sia
positivo

→ Anche detta
RAPPRESENTAZIONE
CON POLARIZZAZIONE

su p bit, l'intervallo è $[-(2^{P-1}-1), +2^{P-1}]$

Viene utilizzata per rappresentare i numeri in virgola mobile.

Decodifica:

$$a = A - (2^{P-1} - 1)$$

Ogni volta che applico un algoritmo di conversione, mi devo assicurare che il numero da convertire si trovi nell'intervallo di rappresentabilità.

ISTRUZIONE DO

do { istruzioni } while (condizione)

→ restituisce
un booleano

Esecuzione:

- viene eseguita l'istruzione
- viene valutata la condizione
- se TRUE, l'istruzione viene ripetuta
- se FALSE, il ciclo termina

N.B. Il corpo dell'istruzione viene eseguito minimo una volta anche se la condizione è subito FALSE.

ISTRUZIONE FOR

for (inizializzazione; condizione; step)

←
qualunque
espressione o
dichiarazione

↓
restituisce
un booleano

→ qualunque espressione
anche se di solito aggiungiamo
la variabile di controllo.
È eseguito prima di ricontrollare
la condizione

- Sono tutte e tre OPZIONALI

Funziona quindi come:

```
inizializzazione  
while (condiz.) {  
    :  
    step;  
}
```

N.B. Come qualunque altro ciclo, può essere annidato. (i.e. possono essere messi più for uno dentro l'altro).

Inoltre si possono annidare anche due cicli diversi.

ISTRUZIONI DI SALTO

10

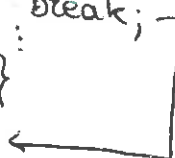
- **break**
- **continue**
- **goto** (sconsigliato e mai usato)
- **return** (serve a terminare il programma)

BREAK salta all'istruzione subito dopo al corpo del ciclo o dello switch che lo contiene

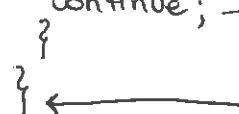
CONTINUE salta alla parte del ciclo che ricontrolla la condizione. Provoca quindi la terminazione di una iterazione del ciclo, ignorando le istruzioni successive.

Esempio:

```
while (cond)
{
    ...
    break;
}
```



```
while (...)
{
    switch (...) {
        ...
        continue;
    }
}
```



Se nel codice scrivi:

```
3+5;
```

compila:

Il compilatore crea una variabile temporanea, gli assegna il risultato e quando termina il programma la variabile scompare

```
cout << 3+5;
```

Scrivendo questo, il compilatore fa questo:

```
int temp;
temp = 3+5;
cout << temp;
```

OPERATORE VIRGOLA

È un operatore binario infisso, **associativo a sinistra**.

esp1, esp2

Viene prima valutata esp1 (sx), dopo viene valutata esp2 (dx).

Facendo così il risultato sarà uguale a quello di esp2, mentre quello di esp1 viene ignorato.

Esempio: $a = (b++, 5)$

→ $a = b++$ viene ignorato
 $a = 5$

→ Le parentesi sono importanti, altrimenti verrebbe visto come $(a = b++)$, 5

Viene utilizzata nel ciclo FOR se dobbiamo inizializzare o aggiungere più variabili:

```
for (int i=0, int j=0; ...)
```

TIPI DERIVATI

Sono derivati dai tipi fondamentali e possono essere composti tra loro così da formare tipi derivati complessi.

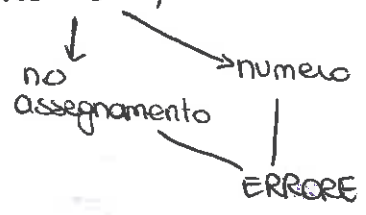
Sono:

- RIFERIMENTI
- PUNTATORI
- ARRAY
- STRUTTURE
- UNIONI
- CLASSI

RIFERIMENTI

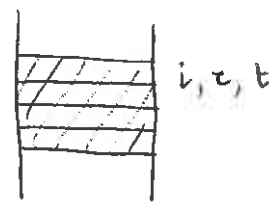
Si tratta di un identificatore di un oggetto.
Riferimento default: primo nome dell'oggetto.

SINTASSI: tipo &nome = variabile;



Crea un "nuovo nome"
per la variabile

Esempio: `int i=10;`
`int &r=i;`
`int &t=r;`
`r++;` //varranno 11 anche i e t



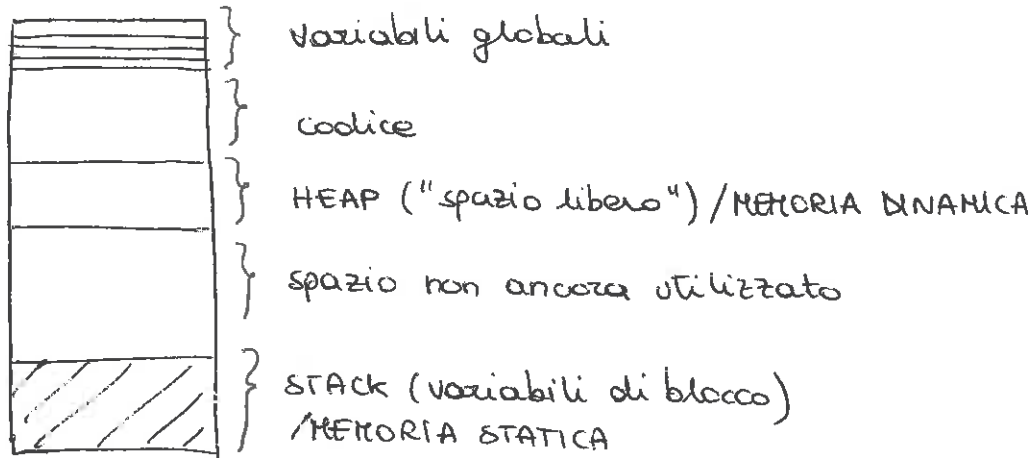
RIFERIMENTI COSTANTI

Un riferimento costante non si deve riferire per forza ad una variabile costante.

Esempio: `int i=10;` → i può modificare il valore, e invece no
`const int &r=i;` e accede al valore solo in lettura.

Se invece la variabile è costante, il riferimento DEVE essere costante.

La memoria RAM è divisa in 5 parti.



Dove vengono memorizzate le variabili?

Ogni volta che creo una variabile, essa viene impilata nello STACK, secondo la politica **LIFO** (Last In First Out).

Per questo, non posso deallocare una variabile che sta nel mezzo, posso solo deallocare quella che sta più in alto, cioè quella inserita per ultima.

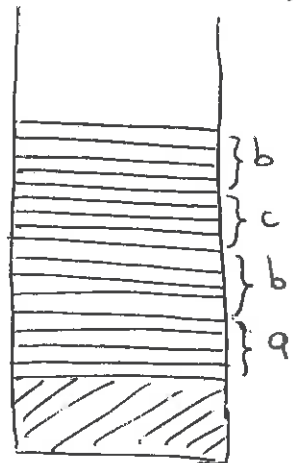
Esempio:

```
int main() {
    int a = 3;
    int b = 5;
    {
        int c = 7;
        int b = 11;
        cout << b; // 11, si riferisce alla
                  // variabile creata per ultima
    }
    cout << c; // ERRORE
}
```

Posso creare una variabile con nome già usato solo se appartengono a due blocchi diversi.

quando termina un blocco, le variabili create in esso vengono deallocate

STACK (PILA)



VARIABILI GLOBALI

- Vengono inizializzate fuori da tutti i blocchi.
- Vengono allocate nella parte più alta della RAM.
- Vengono definite come statiche.

CONCETTO DI FUNZIONE

Per creare una funzione si crea un blocco di istruzioni per poterlo chiamare in più punti del programma.

Per utilizzare una variabile all'interno di una funzione esistono due modi:

- dichiararla LOCALMENTE, in questo caso sarà visibile solo all'interno della funzione
- dichiararla GLOBALMENTE

RAPPRESENTAZIONE VIRGOLA FISSA

Si usa un numero fisso di bit per la parte intera ed un altro numero fisso di bit per la parte frazionaria.

[parte intera, parte frazionaria]

↓
si rappresenta con
metodi noti

↓
Algoritmo della
parte frazionaria

ALGORITMO DELLA P. FRAZIONARIA:

$I(r) \rightarrow$ parte intera

$F(r) \rightarrow$ parte frazionaria

$$F_1 = F(f_0 \cdot 2) \text{ e } a_{-1} = I(f_0 \cdot 2)$$

\vdots

$$F_n = F(f_{n+1} \cdot 2) \text{ e } a_{-n} = I(f_{n+1} \cdot 2)$$

L'algoritmo si ferma quando arriva a zero oppure alla precisione desiderata.

N.B. Se il numero della parte frazionaria non viene rappresentato in f bit si effettua un troncamento.

Esempio: $p=16$ $f=5$

$$(331,6875)_{10} = (101001011, 1011)_2$$

La parte intera è convertita con DIV & MOD ed è uguale a 101001011

La parte frazionaria:

$$F_1 = F(0,6875 \cdot 2) = 1 \text{ e } 0,375$$

$$F_2 = F(0,375 \cdot 2) = 0 \text{ e } 0,75$$

$$F_3 = F(0,75 \cdot 2) = 1 \text{ e } 0,5$$

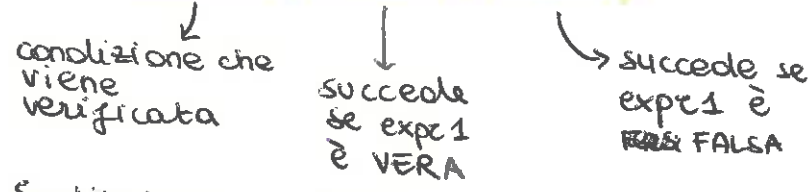
$$F_4 = F(0,5 \cdot 2) = 1 \text{ e } 0 \rightarrow \text{termina}$$

} diventa 1011

OPERATORE TERNARIO

È un operatore con 3 argomenti

expr1 ? expr2 : expr3;



Sostituisce IF (solo nel c++ perché negli altri linguaggi non esiste)
Si utilizza anche per assegnare un valore a una variabile:

```
int n; int DIM; //abbiamo bisogno di un numero positivo
cin >> n;
DIM = n > 0 ? n : 1;
```

PUNTATORI

Sono un tipo derivato e rappresenta l'indirizzo di un altro oggetto con lo stesso tipo.

Dichiarazione: 3 opzioni: 1) int* p;
2) int *p;
3) int * p;

Esempio:

```
int i = 1;
int *p1 = &i; // si riferisce all'indirizzo di i
cout << &i; // stampa l'indirizzo di i in esadecimale
int *p2; // viene creata nello stack con un valore // casuale solo per "tenere il posto"
```

112		p2
116	120	p1
120	1	i

N.B. Se avessimo scritto *p1 = i; avrebbe dato errore

p1 = 10; → chiamata di indicazione: l'operatore '' si aspetta che ci sia un operatore per prendere il suo indirizzo

questa operazione assegna 10 alla variabile che si trova all'indirizzo puntato da p1
*p1 = 10; è uguale a i = 10;

Ci sono diversi modi per stampare:

- cout << *p1; // stampa la variabile puntata
- cout << p1; // stampa l'indirizzo della variabile puntata
- cout << &p1; // stampa l'indirizzo di p1

N.B. Per vedere quanto occupa in memoria un puntatore si usa sempre sizeof (nome)

CASO PARTICOLARE

```
int *p1;  
cout << *p1; → verrà stampato un numero casuale molto grande  
               (che può essere sia positivo che negativo)  
*p1 = 5;  
p1 = 0; → non dà errore, però scrive in un posto non precisato  
         → lo facciamo puntare a niente, se dopo scrivessi  
           *p1 = 3; darebbe errore
```

Per evitare questo tipo di problemi si usa inizializzare il puntatore subito dopo la dichiarazione.

Nella libreria `<iostream>` esiste una direttiva, `#define NULL`, che crea una variabile che possiamo utilizzare.

Esempio:

```
int *p1 = NULL;
```

Si possono puntare anche variabili char:

```
char *p1 = nullptr;  
char c1 = 'z';  
p1 = &c1;  
*p1 = 'd';  
cout << *p1 << " " << c1; // d d
```

STRUTTURE DATI

Sono un tipo derivato. Si tratta di una n-upla ordinata di elementi (anche chiamati membri o campi). È specificata da un nome che andrà a creare un nuovo "tipo".

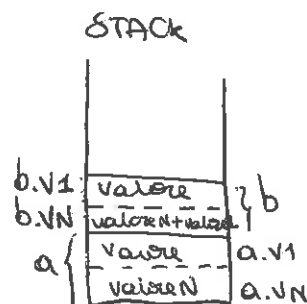
DEF: rappresenta una collezione di informazioni su un dato oggetto

Dichiarazione: `struct Nome {`

```
    tipo V1;  
    ...  
    tipo VN;  
};
```

// l'ordine è importante

```
int main() {  
    - Nome a, b;  
    a.V1 = valore; // stesso tipo  
    a.VN = valoreN;  
    b.V1 = a.V1;  
    b.VN = a.VN + valore1;  
}
```



- Nome $*p = &a;$

$p \rightarrow v_1$
 $p \rightarrow v_2$ } per riferirsi alle singole variabili della struttura ~~non~~ si usa " \rightarrow "

Si può anche inizializzare una struttura senza specificare le variabili, utilizzando $\{\}$.

Esempio:

- Nome t;

$t = \{\text{valore 1, valore N}\} \rightarrow$ l'ordine è importante, se fosse stato $\{\text{valore N, valore 1}\}$ avrebbe dato errore

N.B. struct _Nome {

...
 - Nome a;
 }, // è ERRORE

invece

struct _Nome {

...
 - Nome *a;
 }, // è OK

COME SI SCAMBIANO DUE VARIABILI?

Viene utilizzata una terza variabile:

```
int a, b, c;
a = 3;
b = 5;
cout << a << " " << b; // 3 5
c = a;
a = b;
b = c;
cout << a << " " << b; // 5 3
```

E SE VOGLIAMO UTILIZZARE UNA FUNZIONE?

```
void scambia (int &a, int &b) {
  int c;
  c = a;
  a = b;
  b = c;
  return;
}
```

PASSAGGIO PER RIFERIMENTO

Si utilizzano i riferimenti per poter modificare le variabili fuori e dentro la funzione.

Infatti, se negli argomenti non mettiamo il riferimento, all'interno della funzione le variabili cambiano, ma fuori no.

sizeof (tipo)

Operatore che calcola quanto occupa la variabile (tra parent.) in memoria

PUNTORI pt. 2

Quando scrivo

```
int *p1;  
p1 = 0;
```

p1 non sta puntando a nulla

Quando invece scrivo soltanto

```
int *p3;
```

viene allocata la variabile in memoria e viene inizializzata con la sequenza di bit che c'era prima.

Infatti se facessi

```
cout << *p3; // stampa il contenuto di quella cella dove è stato allocato  
// il puntatore interpretandolo come interi in compl. a due
```

Se scrivessi

```
*p3 = 5;  
cout << *p3; // stampa 5
```

Siamo andati a scrivere in una ~~posizione~~^{zona} in memoria, che può essere già allocata e si va di conseguenza a sovrascrivere il codice.

Per questo conviene scrivere

~~int *p4 = 0;~~

```
int *p4 = 8i; // se ho già la variabile da assegnare
```

```
int *p4 = 0; // se la var. da ass. viene dopo
```

```
*p4 = 5; // se provassi a scrivere qualcosa, non ci sarà mai errore di compilaz.
```

L'errore "SEGMENTATION FAULT" fa terminare il programma.

Se si incorre in questo errore è perché

Nella libreria Iostream esiste una direttiva:

```
#include <iostream>
```

```
#define NULL 0 // direttiva del preprocessore [programma che parte ancora  
// che fa passare il pre-prec. prima del processore]
```

```
// in base a tutte le istruzioni
```

```
// e ad ogni occorrenza di NULL ci sostituisce 0
```

```
int main(){
```

```
int *p5 = NULL;
```

```
int *p6 = nullptr;
```

```
} = 0
```

(versione più recente)

keybrd - Vale 0 e si tratta di un letterale tipizzato

FUNZIONI

```
void stampa3Ast (void) {
    cout << " *** ";
}
```

↑
oppure vuoto

Si è capito che utilizzando le variabili globali, viene meno la leggibilità del codice

Esercizio Funzione che calcola operaz. intere tra due interi e che restituisca 5 risultati

1° soluz.

```
void CalcoloOperazIntere (int a, int b, int &somma, int &sozz, int &prod, int &div,
                          int &mod) {
```

```
    somma = a + b;
    sozz = a - b;
    prod = a * b;
    div = a / b;
    mod = a % b;
```

```
}
```

```
int main() {
```

```
    int so, st, pc, dv, mo;
    CalcoloOperazIntere (5, 3, so, st, pc, dv, mo);
    cout << so;    // 8
```

```
}
```

2° soluz.

STRUTTURE DATI

N-upla ordinata di elementi, detti membri o campi. Ciascun membro ha un tipo e un nome specifico. È come un contenitore di informazioni

```
struct punto {
```

```
    double x;
    double y;
```

```
};
```

```
int main() {
```

```
    punto z, s;
```

```
    z.x = 3;
```

```
    z.y = 10.5;
```

```
    s.x = z.x;
```

```
    s.y = z.y + 10;
```

// prendiamo in considerazione il campo x della struttura dati z

```
punto *p = &z;
cout << 'z' << p->x << " " << p->y << ">\n"; // (*p).x (*p).y
punto t = { 1.0, 2.0 }; //inizializ.
```

"->" è un operatore

In una struttura non posso mettere un campo che ha come "tipo" la struttura stessa.

Posso tuttavia inserire un campo puntatore ad una struttura dati dello stesso tipo.

2ª soluz.

```
struct Risultati {
    int so;
    int st;
    int pr;
    int dv;
    int mo;
};
```

Risultati calcola Oper(int (int a, int b) {

```
    Risultati r;
    r.so = a+b;
    r.st = a-b;
    r.pr = a*b;
    r.dv = a/b;
    r.mo = a%b;
    return r;
}
```

int main() {

```
    Risultati s;
    s = calcolaOper(int (5,3); // si fa una copia membro a membro per mezzo
    cout << "Somma: " << s.so << end; // dall'operatore assegnamento
    //ecc.
}
```

Nelle funzioni che restituiscono un valore, al momento della chiamata viene creata, prima di tutte, una variabile in più in cui ~~contenuto del risultato~~ è contenuto il risultato. Quando nel main si fa l'assegnamento viene deallo-cata quella variabile risultato senza nome.

NOTAZIONE SCIENTIFICA

Standard IEEE 754-1985

2008

2019

$$x = \pm m \cdot \beta^e \quad m = \text{mantissa} \quad e = \text{esponente}$$

\uparrow
 determina l'intervallo di rappresentabilità determina l'accuratezza

$$x \leftrightarrow R = \langle s, E, F \rangle$$

La rappresentazione R è composta da 3 naturali:

- s codifica il segno (1 bit, 1 se negativo, 0 se positivo)
- E codifica dell'esponente su k bit
- F codifica della parte frazionaria della mantissa su 6 bit

$$x = (s=0)? [(1+f) \cdot \text{due}^e] : [-(1+f) \cdot \text{due}^e]$$

$$f = F/2^6 \quad e = E - (2^{k-1} - 1)$$

con BIAS

Esempi

HALF PRECISION

Numero su 16 bit, $k=5$ e $6=10$.

$$1) R = \{1, 10011, 1110100101\}$$

$$f = F/2^{10} = 0.1110100101$$

$$e = E - (2^{5-1} - 1) = 10011 - (+01111) = 100 \quad (\text{bias} = 15)$$

$$x = -1.1110100101 \cdot \text{due}^4 = -11110.100101$$

$$x = -30.578125 \quad \text{in base dieci}$$

$$2) R = \{0, 01111, 0000000001\}$$

$$F = 1/2^6 = 2^{-10} = 0.0009765625$$

$$e =$$

Esercizio

r: +5.125

$R \begin{cases} \text{quarter } (1, 4, 3) \quad \underline{8 \text{ bit}} \\ \text{half } (1, 5, 10) \quad 16 \text{ bit} \\ \text{float } (1, 8, 23) \quad 32 \text{ bit (detta anche single precision)} \\ \text{double } (1, 11, 52) \quad 64 \text{ bit} \end{cases}$

Parte intera di 5.125 è 5 $\rightarrow 101$ Parte fraz. è 0.125 $\rightarrow f_0 = 0.125$

$$f_{-1} = PF(2 * f_0) = 0.25$$

$$a_{-1} = I(0.25) = 0$$

$$f_{-2} = PF(2 * f_1) = 0.5$$

$$a_{-2} = I(0.5) = 0$$

$$f_{-3} = PF(2 * f_{-2}) = 0$$

$$a_{-3} = I(1.0) = 1$$

$$5.125 \rightarrow (101.001)_2$$

$$1.01001 \cdot 2^{-2}$$

$$0111 \text{ BIAS} = 7 \quad E = e + \text{BIAS}$$

$$2 + 7 = 9$$

$$5.125 = 0 \mid 1001 \mid 010$$

FUNZIONI CON PUNTATORI

PASSAGGIO PER VALORE CON ARGOMENTO PUNTATORE

```
int incrementa(int *p) {
```

```
    int aux = *p;
```

```
    (*p)++;
```

```
    return aux;
```

```
}
```

```
int main() {
```

```
    int a = 13
```

```
    int *q = &a;
```

```
    incrementa(q);
```

```
    cout << a << endl; // 14
```

```
    // int b = incrementa(q);
```

```
    // cout << b << endl; // 13
```

Effetto collaterale : modifica di una variabile non locale alla funzione
 NON RIENTRA l'effetto provocato dal ritorno della funzione perché quello è
 l'effetto principale della funzione

ARRAY / VETTORI

23

N-upla ordinata di elementi dello stesso tipo, memorizzati in memoria in maniera contigua. Ci si riferisce ad essi tramite un indice che ne indica la posizione.

 $v = 8[0];$

```
const int N=5;
int v[N]; //dimensione del vettore
for(int i=0; i<N; i++)
    cin >> v[i];

int j = v[1]; //j prende 1
v[1] = 7; //v[1] vale 7
```

104	0	v[0]
108	7	v[1]
112	2	v[2]
116	3	v[3]
120	4	v[4]
124	5	N

```
int *p = &v[1];
(*p)++; //indirettamente v[1]=8;
```

```
cout << v; //104, indirizzo della prima componente del vettore
```

Posso inizializzare un vettore specificandone il contenuto e non la dimensione

```
int a[] = {0, 1, 2, 3}; //array di 4 elementi
```

```
const int N=6;
int b[N] = {0, 1, 2, 3}; //avendo specificato solo 4 componenti di 6,
//le altre valgono 0 di default
```

//ERRORE CHE NON VIENE SEGNALATO

```
for(int i=0; i<N; i++)
    cout << a[i]; //verranno stampati i 4 elementi di a
//gli altri due saranno valori casuali
```

→ ACCESSO FUORI RANGE / OUT OF BOUND

OPERAZIONI SUGLI ARRAY

Non sono permesse operazioni aritmetiche, di confronto e di assegnamento.

ARITMETICA DEI PUNTATORI

Essendo p una variabile che indica l'indirizzo, $p+1$ punterà all'indirizzo della variabile successiva

```
q = 3; //q = 3 * sizeof(*q)
```

```
int v[5] = {0, 1, 2, 3, 4}
```

```
q = v;
```

```
*q = 3; // = v[0]
```

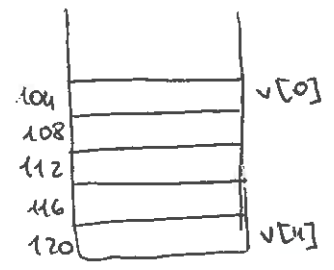
```
*(q+1) = 2;
```

```
cout << v[1] << endl; //2
```

int *q, *r;
cout << q - r; // differenza degli indirizzi / sizeof(int)

↓
sottrazione tra puntatori concessa

```
int v[5] = {1, 2, 3, 4, 5};
q = v;
r = &v[4];
cout << q - r; // 4, distanza in termini di indici
               // 10 fa direttamente il calcolatore
```



$$\frac{104 - 120}{4} = \frac{-16}{4}$$

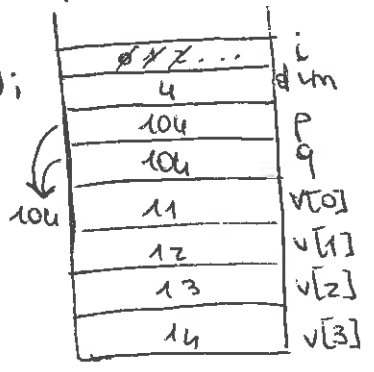
```
void StampaVettore (int *p, int dim);
```

```
int main() {
```

```
    int v[4] = {11, 12, 13, 14}; int *q = v;
```

```
    StampaVettore (q, 4); // oppure senza inizializzare q:
```

```
    return 0; // StampaVettore (&v, 4);
              // StampaVettore (&v[0], 4);
```



```
void StampaVettore (int *p, int dim) {
```

```
    for (int i = 0; i < dim; i++)
```

```
        cout << *(p+i) << " "; // anche p[i];
```

```
    cout << endl;
```

```
void Anz (int *p, int dim) {
```

```
    for (int i = 0; i < dim; i++)
```

```
        cout << *(p+i)++ << " ";
```

// questa funzione permette di
// modificare il contenuto del
// vettore dentro e fuori la funzione

```
    cout << endl;
```

Nei vettori esiste l'operatore **PARENTESI QUADRA**:

v[indice] ≡ *(v+indice)

TYPE DEF

25

È una parola chiave che consente di creare un sinonimo di un tipo già esistente

```
int main(){
    typedef int* pInt;
    pInt p1 = nullptr;
    int i = 5;
    p1 = &i;
    return 0;
}
```

PUNTATORI A COSTANTI

È una variabile puntatore che punta a oggetti costanti

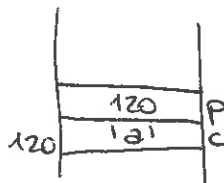
```
int main(){
    int i = 0;
    const int *p;
    p = &i; // i non è costante
    int j;
    j = *p;
    *p = 1; // NO. Limitazione: il puntatore, se costante, può accedere
           // alla variabile puntata in sola lettura
    const int k = 10;
    const int *q = &k;

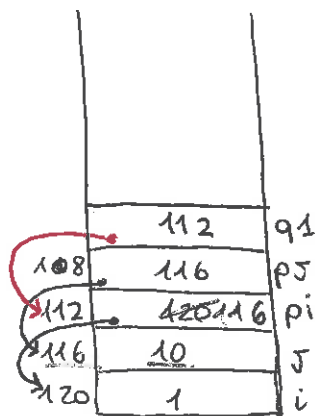
    int *qq;
    qq = &k; // ERRORE
    qq = q; // ERRORE
```

PUNTATORI COSTANTI

È una variabile puntatore costante che, in quanto tale, devo inizializzarla al momento della definizione e non posso più modificarla.

```
int main(){
    char c = 'a';
    const char *p = &c;
    cout << *p;
    // p pointer sempre a c
    *p = 'b'; // OK
    char char d;
    p = &d; // ERRORE
    char *p1;
    p1 = p; // OK
    p = p1; // ERRORE
    return 0;
}
```





```
int main(){
    int i=1, j=10;
    int *pi = &i, *pj = &j;
    int **q1 = &pi;
    cout << **q1; // 1
    *q1 = pj;
    cout << **q1; // 10
    return 0;
}
```

```
int ***t = &(int **)
```

```
int *t = &(int)
```

CONCETTO DI C-STRINGA

Una c-stringa è un qualunque vettore di caratteri che contenga al suo interno, almeno una volta, il carattere '\0'.

```
int main(){
    char vc[] = {'c', 'i', 'a', 'o'}; // vettore di caratteri
    char cstz[] = {'c', 'i', 'a', 'o', '\0'}; // stringa
```

// '\0' = MARCA DI FINE C-STRINGA

~~overloading~~

```
cout << cstz << endl; // ciao
cout << vc << endl; // ciaoazf89
```

→ Nel caso delle stringhe, l'operatore cout stampa tutti i caratteri fino a che incontra il carattere '\0'.

```
cout << (void *)vc;
```

```
// 0x123d43
```

```
cout << &vc[0];
```

```
// 0x123d43
```

CONVERSIONE DI
PUNTATORE A VOID

Questo perché intende il vettore di caratteri come una stringa, allora stampa il vettore più fa un accesso fuori dal range stampando anche tutte le variabili che si trovano dopo in memoria fino a che non incontra '\0' (se c'è).

C-stringa è vettore di char

(stringhe sono sottoinsieme proprio dei vettori di caratteri)

CONVERSIONE ESPLICITA

```
cout << int(3.56);  
cout << (int) 3.56; // funziona meglio sui puntatori
```

FUNZIONI SULLE STRINGHE

```
void stampaCstringa(char* p){  
    while (*p != '\0'){  
        cout << *p; // p = p + sizeof(char) * 1  
        p++;  
    }  
}
```

⇒ NEL MAIN:

```
stampaCstringa(cste);
```

Altro modo per stampare una stringa:

```
char cste3[] = "casa";  
cout << cste3 << endl; // casa
```

Ha:

- LUNGHEZZA FISICA = 5
perché occupa 5 celle di memoria (anche '\0')
- LUNGHEZZA LOGICA = 4
lung. fisica - 1 (SOLITAMENTE)

Funzione che restituisce la lunghezza logica di una c-stringa:

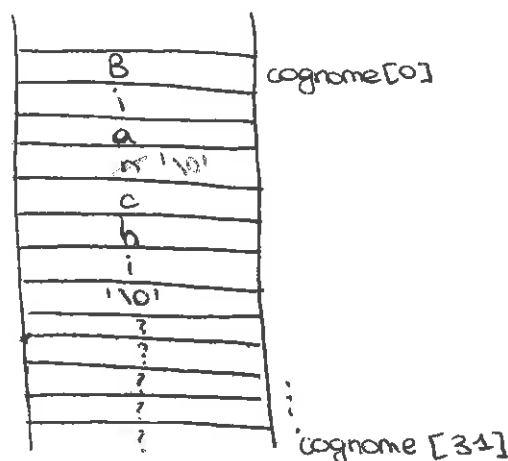
```
int my_strlen(char* p){  
    int i = 0;  
    while (*p != '\0') {  
        i++; p++;  
    }  
    return i;  
}
```

⇒ NEL MAIN:

```
strlen(cste3); // 4
```

Esiste una libreria sulle stringhe: `#include <cstring>`. Tra le altre funzioni, contiene anche `strlen()`.

```
char cognome[31];  
cin >> cognome; // salta gli eventuali spazi bianchi all'inizio, non li  
cout << cognome << endl; // considera  
cognome[3] = '\0';  
cout << cognome << endl; // Big
```



```
void LeggiCStringaDaTastiera(char *p, int max-len) {
```

```
    char ch;
```

```
    int i=0;
```

```
    while (i < max-len) {
```

```
        cin >> ch;
```

```
        if (ch == '\n' || ch == ' ')
```

```
            break;
```

```
        *p = ch;
```

```
        p++;
```

```
        i++;
```

```
    }
```

```
    *p = '\0';
```

```
}
```

==> NEL MAIN:

```
LeggiCStringaDaTastiera(nome, 10);
```

C-STRINGHE COSTANTI

```
const char cstch[] = "Sole";
```

```
cstch[3] = '\0'; //ERRORE
```

```
//char *x = cstch; // il puntatore deve essere anch'esso const
```

```
//x[4] = '\0'; //ERRORE
```

```
//cout << x;
```

```
const char *x = cstch;
```

```
cout << x[3]; //e
```

```
//x[3] = '\0'; //ERRORE
```

Se volessi passare ~~questa~~ questa stringa costante alla funzione `my_strlen()`; dovrei cambiare l'argomento formale della funzione in costante.

FUNZIONI

```
void my_strcpy(char *d, const char *s) {
```

```
    int i=0;
```

```
    while (s[i] != '\0') {
```

```
        d[i] = s[i];
```

```
        i++;
```

```
    }
```

```
    d[i] = '\0';
```

```
int main() {
```

```
    const sorg[] = { 'C', 'a', 's', 'a', '\0', 'm', 'i', 'a', '\0' };
```

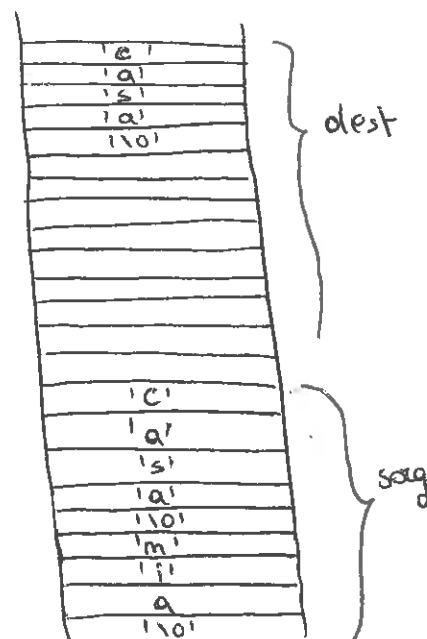
```
    char dest[100+1]; // char *dest = nullptr;
```

```
    my_strcpy(dest, sorg); cout << dest << " " << sorg << endl;
```

```
    return 0;
```

```
}
```

[strcpy è una funzione
presente nella libreria]



RIEPILOGO - RAPPRESENTAZIONE DELL'INFORMAZIONE

In un calcolatore i vari tipi di informazione si rappresentano per mezzo di sequenze di bit.

L'informazione corrisponde a tutte le possibili disposizioni di due oggetti (1 e 0) e in n case (quindi 2^n)

La codifica ASCII è standardizzata su 7 bit.

La codifica ASCII estesa è invece su 8 bit e contiene ulteriori 128 caratteri ($2^7 \cdot 2$)

RAPPRESENTAZIONE DEI NUMERI NATURALI

Di base, i numeri sono rappresentati in base 2 e base 10, attraverso una rappresentazione posizionale del tipo:

$$(123)_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

$$(11001)_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = (25)_{10}$$

Da questo procedimento si ricava la FORMULA DELLA SOMMATORIA:

$$N = \sum_{i=0}^{p-1} a_i \beta^i = a_{p-1} \beta^{p-1} + \dots + a_1 \beta^1 + a_0 \beta^0$$

p = posizione
↓
cifra meno significativa
↑
cifra + significativa
base
numero

SEQUENZE DI SIMBOLI in base β $\xrightarrow{\text{FORMULA DELLA SOMMATORIA}}$ SEQ. DI SIMBOLI in base 10

SEQ. DI SIMBOLI in base 10 $\xrightarrow{\text{DIV \& MOD}}$ SEQ. DI SIMBOLI in base β

PROCEDIMENTO DIV & MOD: in cui MOD è il RESTO e DIV è il quoziente

$N=23$ in $\beta=10$

QUOZ	R
23	1
11	1
5	1
2	0
1	1
0	

diventa 10111

In generale si pone $q_0 = N$ e poi

$$\begin{aligned} q_1 &= q_0 \text{ div } \beta & a_0 &= q_0 \text{ mod } \beta \\ q_2 &= q_1 \text{ div } \beta & a_1 &= q_1 \text{ mod } \beta \\ q_p &= q_{p-1} \text{ div } \beta & a_{p-1} &= q_{p-1} \text{ mod } \beta \end{aligned}$$

fino a quando q_p diventa uguale a 0.

Per una generica base β , l'intervallo di rappresentabilità con p cifre è $[0, \beta^p - 1]$.

p	Intervallo
8	[0, 255]
16	[0, 65535]
32	[0, 4294967295]

In generale, ci vogliono $p+1$ bit per sommare due numeri di p bit, perciò si dice che la somma ha dato luogo ad un OVERFLOW.

Es: $1+1=0$ $1+0=1$ $0+1=1$ $0+0=0$
 $R=1$

Esercizio CAMBIO da BASE X a BASE Y

$$(1023)_{\text{cinque}} = 5^3 \cdot 1 + 5^2 \cdot 2 + 5^1 \cdot 3 = (138)_{\text{dieci}}$$

Uso DIV & MOD: $q_0 = 138$

$$q_1 = 138 \text{ div } 9 \quad a_0 = 138 \text{ mod } 9$$

Q	R
138	-
15	3
1	6
0	1

$\rightarrow (163)_{\text{nove}}$

\rightarrow CASI PARTICOLARI: potenze di 2

RAPPRESENTAZIONE DEI NUMERI INTERI

① Rappresentazione in Modulo e Segno (trasformazione $a \Rightarrow A$)

Sia a il numero intero che si vuole rappresentare su p bit, la rappresentazione A di a è data da $A = a_{p-1}, \dots, a_0 = (\text{segno-}a, \text{ABS-}a)$

N.B.: Ad $a=0$ corrispondono due rappresentazioni:
+zero e -zero.
(0,00...0) (1,00...0)

è un unico bit che se vale 0 allora $a \geq 0$, 1 altrimenti
(BIT DEL SEGNO)

Rappresentazione del valore assoluto di a su $p-1$ bit
 a_{p-2}, \dots, a_0

Rappresentazione in Modulo e Segno (trasformazione $A \Rightarrow a$)

$$A = (a_{p-1}, a_{p-2}, \dots, a_0) \quad a = (a_{p-1} == 0)? +\text{ABS-}a : -\text{ABS-}a$$

Intervallo di rappresentabilità in H&S su p bit: $[-(2^{p-1}-1), +2^{p-1}-1]$

N.B.: Prima di applicare l'algoritmo $a \Rightarrow A$ occorre verificare che a sia rappresentabile su p bit.

OVERFLOW

Per gli interi in complemento a 2 l'overflow si genera quando si vogliono sommare due numeri definiti su p bit che hanno lo stesso segno, se i numeri sono troppo grandi infatti si rischia che il risultato non sarà rappresentabile su p bit ma su $p+1$, generando quindi overflow.

NUMERI INTERI

② Rappresentazione in COMPLEMENTO a 2 (trasformazione $a \Rightarrow A$)

Sia a il numero intero. La rappresentazione sarà:

$$A = a_{p-1} \dots a_0 = (a \geq 0) ? ABS_a : (2^p - ABS_a)$$

sono rappresentati in $\beta=2$ come i naturali su p bit

Esempio per $p=4$:

- $a=1 \Rightarrow ABS_a=1$ diventa 0001, quindi $A=0001$
- $A=-1 \Rightarrow ABS_a=1$, $2^4 - ABS_a = 16 - 1 = 15$, quindi $A=1111$

15	
7	1
3	1
1	1
0	1

Questa rappresentazione è anche detta in COMPLEMENTO ALLA BASE. Infatti, con lo stesso procedimento si possono rappresentare interi con $\beta \neq 2$.

Rappresentazione in COMPLEMENTO a 2 (trasformazione $A \Rightarrow a$)

$$A = (a_{p-1} \dots a_0) \quad a = (a_{p-1} = 0) ? +A : -(2^p - A)$$

↓ ↓
naturali su p bit naturali su p bit

Esempio per $p=4$:

- $A=0111 \Rightarrow a=+7$
- $A=1001 \Rightarrow 2^4 - 9 = 7 \Rightarrow a=-7$

In questo caso non viene sprecata nessuna rappresentazione: lo zero è rappresentato solo una volta.

Intervallo: $[-2^{p-1}, +2^{p-1}-1]$

p	Intervallo
4	$[-8, 7]$
8	$[-128, 127]$
16	$[-32768, 32767]$

③ Rappresentazione con BIAS (trasformazione $a \Rightarrow A$)

Sia a il numero intero.

$$A = a_{p-1} \dots a_0 = a + (2^{p-1} - 1) \text{ dove } BIAS = (2^{p-1} - 1), \text{ quindi } A = a + BIAS$$

Esempio per $p=4$:

- $a=1 \Rightarrow 1 + (8-1) = 8$ che in $\beta=2$ su $p=4$ è 1000, quindi $A=1000$
- $a=-1 \Rightarrow -1 + (8-1) = 6$ che in $\beta=2$ su $p=4$ è 0110, quindi $A=0110$

Questa rappresentazione è anche detta con PAARIZZAZIONE.

Rappresentazione con BIAS (trasformazione $A \Rightarrow a$)

Sia A un numero intero rappresentato con BIAS su p bit.

$$a = A - (2^{p-1} - 1) \text{ dove } A \text{ è visto come un naturale su } p \text{ bit, quindi } a = A - BIAS$$

Esempio per $p=4$:

- $A=1111 \Rightarrow 15 - (2^3 - 1) = 15 - 7 = 8 \Rightarrow a=8$
- $A=0000 \Rightarrow 0 - (2^3 - 1) = -7 \Rightarrow a=-7$

N.B.: lo zero, anche qui, è rappresentato una sola volta.

Intervallo: $[-2^{p-1}+1, 2^{p-1}]$, ossia $[-bias, +bias+1]$

I calcolatori rappresentano gli interi in complemento a 2 perché non occorre una circuiteria per sommare e sottrarre i numeri interi, ma viene utilizzata quella dei naturali.

NUMERI REALI - VIRGOLA FISSA

Si usa un numero fisso di bit per la parte intera e un altro numero fisso per la parte frazionaria. Sia r un numero reale da rappresentare:

$I(r)$ = parte intera $F(r)$ = parte frazionaria.

Siano p i bit per rappresentare r : f per la parte frazionaria e $p-f$ per la parte intera.

$R = a_{p-f-1} \dots a_0 a_{-1} a_{-f}$ $r \approx \sum_{i=-f}^{p-f-1} a_i \beta^i = a_{p-f-1} \beta^{p-f-1} + \dots + a_0 \beta^0 + \underbrace{a_{-1} \beta^{-1} + \dots + a_{-f} \beta^{-f}}_{\text{PARTE FRAZIONARIA}}$

$I(r)$ si rappresenta con le tecniche già note.

$F(r)$ si usa la procedura **PARTE FRAZIONARIA - PARTE INTERA**:

$f_0 = F(r)$ se $f_0 \neq 0$ si fa:

$f_{-1} = F(f_0 * 2)$ $a_{-1} = I(f_0 * 2)$

$f_{-2} = F(f_{-1} * 2)$ $a_{-2} = I(f_{-1} * 2)$... fino a che $f_{-j} = 0$ oppure si è raggiunta la precisione desiderata

Esempio: $p=16$ $f=5$ $r = +331,6875$

$331 = 101001011$ $0.6875 = 1011$

F	I
$f_{-1} = F(0.6875 * 2 = 1.375) = 0.375$	$a_{-1} = I(1.375) = 1$
$f_{-2} = F(0.375 * 2 = 0.75) = 0.75$	$a_{-2} = I(0.75) = 0$
$f_{-3} = F(0.75 * 2 = 1.5) = 0.5$	$a_{-3} = I(1.5) = 1$
$f_{-4} = F(0.5 * 2 = 1) = 0$	$a_{-4} = I(1) = 1$

$R = (+101001011, 1011)$

331	-
165	1
82	1
41	0
20	1
10	0
5	0
2	1
1	0
0	1

NUMERI REALI - VIRGOLA MOBILE

$$r = \pm m \cdot \beta^e \rightarrow \begin{array}{l} \text{intervallo di rappresentabilità} \\ \text{è fissato dal numero di cifre} \\ \text{dell'esponente} \end{array}$$

\downarrow
 MANTISSA
 (accuratezza)

parte intera costituito da un solo bit di valore 1

$$r \leftrightarrow R = \langle s, E, F \rangle$$

La rappresentazione R è composta da 3 naturali:

s = codifica del segno (1 bit)

F = codifica della parte frazionaria della mantissa su G bit

E = codifica dell'esponente su k bit

$$r = (s=0) ? [(1+f) \cdot \text{due}^e] : [-(1+f) \cdot \text{due}^e]$$

$f = F/2^G$ è la parte frazionaria della mantissa

$$e = E - (2^{k-1} - 1)$$

Lo zero NON è rappresentabile

Esempio TRASFORMAZIONE $R \Rightarrow r$

Half Precision: 16 bit, $k=5$, $G=10$

$$R = \{ \overset{s}{1}, \overset{E}{10011}, \overset{F}{1110100101} \}$$

$$f = F/2^G = F/2^{10} = 0.1110100101$$

$$e = E - (2^{k-1} - 1) = 10011 - 01111 = 100 \text{ che in bias} = 15$$

$$r = -(1+f) \cdot \text{due}^e = -1.1110100101 \cdot \text{due}^4 = -11110.100101$$

$\begin{array}{l} \downarrow \\ 19 \cdot (2^4 - 1) = 19 \cdot 15 = 285 \\ (E - e) \end{array}$

r diventa -30.578125 in base 10

Esempio TRASFORMAZIONE $r \Rightarrow R$

$r=2$ in half precision

$$f = F/2^G = 0$$

$$e = E - (2^{k-1} - 1) = 10000 - 01111 = 1$$

$$r = +1.0000000000 \cdot \text{due}^1 = 2 \text{ in base 10}$$

L'intervallo di rappresentabilità $(-\infty, +\infty)$ è approssimato da

$$[\cong -2^{\text{bias}+2}, \cong +2^{\text{bias}+2}]$$

Nella half precision:

$$+0 \Rightarrow 2^{-15} \cong 0.31 \cdot 10^{-4}$$

$$-0 \Rightarrow -2^{-15} \cong -0.31 \cdot 10^{-4}$$

ESERCIZI SULLA CONVERSIONE

$$(1252)_7 = (?)_4$$

$$(1252)_7 = 1 \cdot 7^3 + 2 \cdot 7^2 + 5 \cdot 7^1 + 2 \cdot 7^0 = 343 + 98 + 35 + 2 = (478)_{10}$$

478	-
119	2
29	3
7	1
1	3
0	1

 $\rightarrow (13132)_4$

base 16?

"SCORCIATOIA": si raggruppa ogni coppia di cifre in base 4 a partire da quelle meno significative e si trova la cifra equivalente in base 16.

$$(01|31|32)_4$$

$$(01)_4 = 1 = (1)_{16} \quad \rightarrow (13132)_4 = (1DE)_{16}$$

$$(31)_4 = 13 = (D)_{16}$$

$$(32)_4 = 14 = (E)_{16}$$

Questa scorciatoia è applicabile ogni volta che si vuole tradurre un numero e la base di uno è la potenza dell'altro.

ESERCIZIO 2

$$(703)_8 = (?)_{16} \quad \text{Ogni cifra in base 8 corrisponde a 3 bit}$$

$$(7)_8 = 7 = (111)_2$$

$$(0)_8 = 0 = (000)_2 \quad \rightarrow (703)_8 = (111\,000\,011)_2$$

$$(3)_8 = 3 = (011)_2$$

Ogni cifra in base 16 corrisponde a 4 bit

$$(0001|1100|0011)_2 = (1C3)_{16}$$

FUNZIONI SULLE STRINGHE

int my_strcmp(str1, str2) produce come risultato un intero che è:

- < 0, se str1 si trova nel dizionario prima di str2
- = 0, se sono uguali
- > 0, se str2 viene prima di str1

STANDARD della
funzione strcmp();
della libreria

```
int my_strcmp(const char *s1, const char *s2) {
    int i=0;
    while(s1[i]!='\0' && s2[i]!='\0'){
        if(s1[i] < s2[i])
            return -1; //valore minore di 0
        if(s1[i] > s2[i])
            return 1; //valore > 0
        return 0;
        return 0;
        i++; //esamino il carattere successivo
    }
    if((s1[i]!='\0') && (s2[i]=='\0'))
        return 1;
    if((s1[i]=='\0') && (s2[i]!='\0'))
        return -1;
    //arrivo a questo punto solo nel caso in cui sia uscito dal
    //while con entrambe le stringhe terminate
    //Quindi nel caso in cui le stringhe siano uguali
    return 0;
}
```

⇒ NEL MAIN:

```
cout << my_strcmp("casa", "casale") << endl; //questi due argomenti,
cout << my_strcmp("colore", "cobalto") << endl; //scritti direttamente,
cout << my_strcmp("casa", "casa") << endl; //sono costanti
//la stampa sarà: -1, 1, 0
```

Esiste la funzione strcmp standardizzata nella libreria <string>

ES X CASA

Scrivere una funzione my_strcat(char*dest, const char*sorg) con ritorno void che deve prendere in ingresso la c-stringa destinazione e sorgente e deve modificare dest aggiungendo in fondo sorg.
Vedi altre funzioni sulle slide. Implementare anche my-toupper, my-tolower.

strcpy(dest, sorg, MAX) → copia n bit dalla sorg alla dest
se nella sorg ci sono meno di n
bit si ferma a '\0'

ORDINAMENTO DI VETTORI

30

3, 7, -3, 7, 8, 21, 5

Vogliamo scrivere una funzione che, preso in ingresso un vettore, ne modifichi il contenuto ordinandolo.

1. PRIMA FUNZIONE (UTILITA')

Dato un vettore in entrata e un indice i e un indice j , scambia i due elementi all'interno del vettore che si trovano in posizione i e j .

```
void scambia (int *v, int i, int j){  
    //scambia v[i] con v[j]  
    int aux = v[i];  
    v[i] = v[j];  
    v[j] = aux;  
}
```

2. SECONDA FUNZIONE (UTILITA')

Dato un vettore, ci restituisce la posizione del minimo

```
int postMinimo (int *v, int n){  
    int vmin = v[0];  
    int postMin = 0;  
    for (int i = 1; i < n; i++){  
        if (v[i] < vmin){  
            vmin = v[i];  
            postMin = i;  
        }  
    }  
    return postMin;  
}
```

```
int minimo (int *v, int n){  
    int vmin = v[0];  
    for (int i = 1; i < n; i++){  
        if (v[i] < vmin)  
            vmin = v[i];  
    }  
    return vmin;  
}
```

Queste due funzioni sono di utilità perché ~~con~~ usate insieme servono ad ordinare un vettore.

SELECTION SORT (strategia divide et impera)

24

```
void SelectionSort (int *v, int n){
    for (int i=0; i<n; i++){
        int postMin = postMinimo (v, i, n);
        if (postMin != i)
            scambia (v, i, postMin);
    }
}
```

⇒ NEL MAIN:

```
int vett [] = { 2, 3, 2, 1, 4, 1 };
stampaVett (vett, 6);
cout << endl;
SelectionSort (vett, 6);
stampaVett (vett, 6);
cout << endl;
```

SELECTION SORT (SENZA FUNZIONI DI UTILITA')

```
void SelectionSort2 (int v[], int nElem){
```

```
    for (int i=0; i<nElem; i++)
        for (int j=i+1; j<nElem; j++)
```

// ha complessità n^2

```
            if (v[j] < v[i]) { → ordine decrescente
```

```
                int aux = v[i];
```

// $v[j] > v[i]$ ordine crescente

```
                v[i] = v[j];
```

```
                v[j] = aux;
```

```
    }
```

Funzione che ricerca un eventuale elemento dato all'interno di un sottovettore.

```
bool ricLin (const int *v, int infe, int supe, int k, int&pos){
```

```
    bool trovato = false;
```

```
    while (!trovato && infe <= supe){
```

```
        if (v[supe] == k){
```

```
            trovato = true;
```

```
            pos = supe;
```

```
        }
```

```
        supe--;
```

```
    }
```

```
    return trovato;
```

```
}
```

Questa soluzione, con variabile ausiliaria "trovato", permette di non uscire da un punto intermedio del while.

ALGORITMO DI RICERCA BINARIA:

32

Si chiama così perché divide il vettore in due sottovettori ogni volta che si esamina la condizione:

```
bool RicBin(const int *v, int infe, int supe, int k, int & pos) {
    while (infe <= supe) {
        int medio = (supe + infe) / 2;
        if (v[medio] == k) {
            pos = medio;
            return true;
        } else {
            if (v[medio] > k)
                supe = medio - 1;
            else
                infe = medio + 1;
        }
    }
    return false;
}
```

// complessità minore rispetto
// alla ricerca lineare

Un secondo algoritmo per l'ordinamento dei vettori è il BUBBLE SORT, cioè "ordinamento a bolle":

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

ordinato == false
// ha complessità quadratica
se completo questo ciclo e ordinato rimane true, non rientra nel ciclo più esterno (quello di i)
// se non ci troviamo nel worst case // il bubble sort è migliore

Se non ci fosse stata la variabile ordinato, l'algoritmo avrebbe funzionato ugualmente però avrebbe fatto tutte le passate. Infatti, il bubble sort ci consente di capire se la passata successiva sarà inutile se in quella precedente non ho effettuato uno scambio.

Come si crea una variabile in memoria dinamica:

```
int *p = nullptr;
```

`p = new int;` → Esiste un operatore, l'operatore `new`, con il quale posso creare una variabile senza nome di qualsiasi tipo in memoria dinamica. Esso restituisce l'indirizzo della prima cella dell'oggetto.

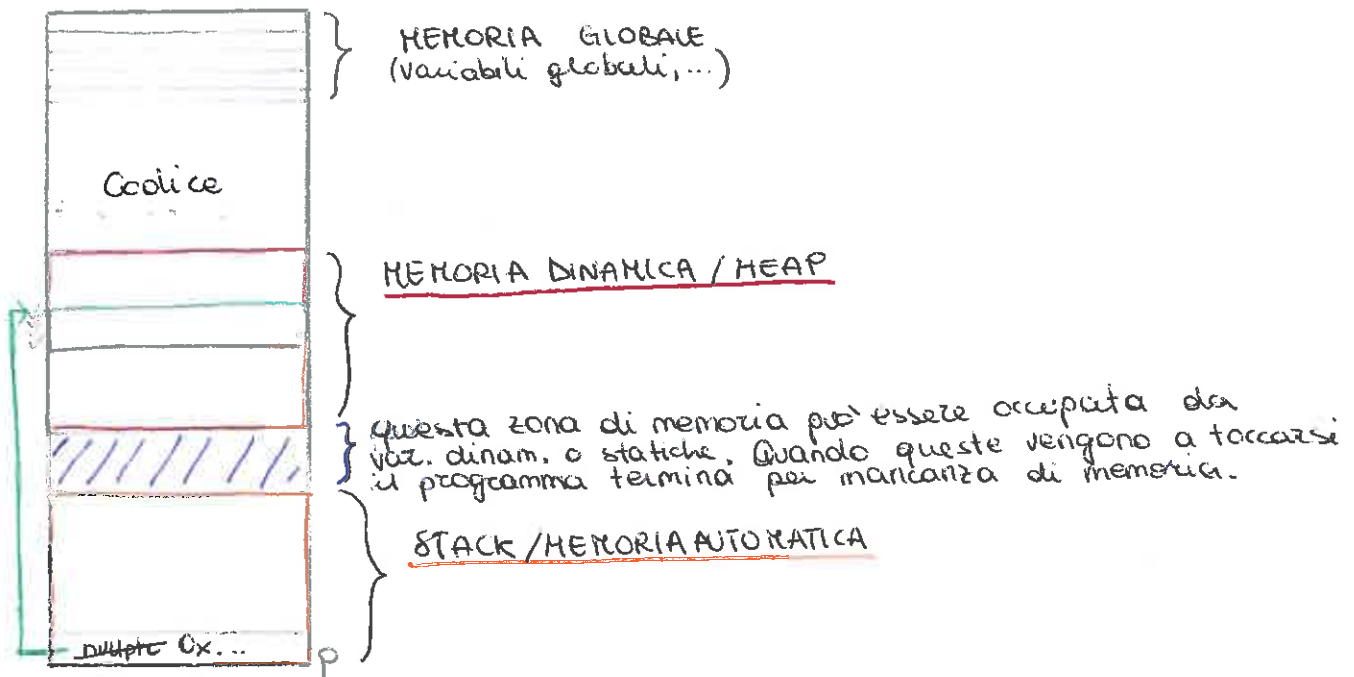
```
*p = 33;
```

`cout << *p << endl;` → Posso accedere in scrittura e lettura

```
delete p;
```

Posso decidere quando DEALLOCARE questa variabile, mediante l'operatore `delete`.

In memoria:



L'operatore `new` va a cercare tante celle quante il `sizeof` del tipo della variabile da allocare, e le va a cercare dopo la parte del codice. Appena trova quelle tot. celle consecutive le occupa, etichettandole. ^(FLAG) Crea quindi, in memoria dinamica la variabile SENZA NOME restituendone l'indirizzo. Per questo tramite puntatore, posso accedere alla variabile in lettura e scrittura.

La variabile dinamica poi viene distrutta non in automatico, quando si esce da un blocco, ma quando viene chiamato l'operatore `delete` su una variabile puntatore contenente l'indirizzo della variabile senza nome.

VARIABILI DINAMICHE E FUNZIONI

24

```
int* creaVariabileSulloHeap() {  
    int *q = new int;  
    return q; // restituisce l'indirizzo per valore  
}
```

```
void deallocaVariabileSulloHeap(int *k) {  
    delete k;  
}
```

==> NEL MAIN:

```
int *r;  
r = creaVariabileSulloHeap();  
*r = 44;  
// ...  
// delete r;  
// oppure scrivo la funzione  
deallocaVariabileSulloHeap(r);
```

DA NOTARE:

si è creata nella funzione una nuova variabile che poi viene passata al main.

```
int *t;  
t = new int[4];  
for (int i = 0; i < 4; i++)  
    t[i] = i;
```

// VETTORE DINAMICO

// t contiene l'indirizzo della prima cella del primo
// elemento del vettore senza nome

```
// ...  
delete [] t;
```

Grazie all'aritmetica dei puntatori, posso accedere a tutti gli elementi del vettore, usando la notazione $t[0]$, $t[1]$, ... ecc.

Differenze tra vettore sullo stack e dinamico:

VETTORE SULLO STACK/AUTOMATICO:

- ha un nome
- è allocato sullo stack
- non posso controllare la sua deallocazione
- non posso crearlo dentro una funzione

DINAMICO:

- non ha un nome
- è allocato in memoria dinamica
- posso decidere io quando distruggerlo
- posso scrivere una funzione che lo alloca

- Un vettore GLOBALE è detto anche vettore STATICO.
- I vettori globali hanno necessariamente dimensione statica (quindi dimensione nota a tempo di compilazione). Se io inizializzo un vettore globale, all'interno delle parentesi quadre posso metterci soltanto o un numero o una costante globale.

==> NEL MAIN:

```
int n;  
cin >> n;  
int vett[n]; // vettore automatico a dimensione dinamica  
// per questo sono detti anche VETTORI SEMI-DINAMICI
```

- Un vettore a dimensione automatica significa che la sua dimensione è nota solo a tempo di esecuzione

MATRICE

Una matrice è un array bidimensionale, caratterizzato da un numero di righe e colonne, e si crea così:

```
const int R=2;
const int C=3;
int m[R][C]; // questa è una matrice 2x3 allocata sullo stack
```

Gli array sono indicizzati da zero invece che da uno per consentire il funzionamento dell'aritmetica dei puntatori.

Per modificare gli elementi della matrice, uso gli indici

```
for(int i=0; i<R; i++)
    for(int j=0; j<C; j++)
        cin >> m[i][j]; ①
```

Con i puntatori:

```
int *p = &m[0][0];
for(int i=0; i<R; i++)
    cout << p[i] << " ";
// cout << *(p+i)
```

// 1 2 3 4 5 6
R₀ R₁

R ₀	1	m[0][0]
	2	m[0][1]
	3	m[0][2]
R ₁	4	m[1][0]
	5	m[1][1]
	6	m[1][2]

```
for(int i=0; i<R; i++) {
    for(int j=0; j<C; j++) {
        cout << p[i*C+j] << " "; ②
        // cout << *(p+i*C+j) << " "; ③
    }
    cout << endl;
}
```

// 1 2 3
// 4 5 6

> C=3

1, 2, 3 sono notazioni che stanno ad indicare la medesima cosa.

Tuttavia, seppur la prima è la più immediata, non sarà universalmente utilizzabile (si noterà quando si dovranno passare delle matrici a funzione).

Come passare una matrice a una funzione:

```
const int C=3; // il numero di colonne deve essere noto a tempo di compilazione
```

```
void inizializza(int m[][C], int r) {
```

```
    for(int i=0; i<r; i++) {
        for(int j=0; j<C; j++)
            m[i][j] = i+j;
    }
```

↓
// di più, dentro la funzione posso accedere all'elemento i, j con doppio indice senza problemi

Se dovessi:

```
stampavettore(int v[4]) { ... }
```

il compilatore ignora la dimensione,
quindi di fatto appare:

```
stampavettore(int v[]) { ... }
```

poi però lo traduce in questo modo:

```
stampavettore(int *v) { ... }
```

questo perché l'unica cosa che conosce / sa fare
è l'aritmetica dei puntatori

Grazie a ciò, è possibile scrivere una funzione che stampa vettori di qualsiasi lunghezza.

Problema con le matrici:

```
void stampaMatrici(int mat[2][3]) { ... }
```

il compilatore ignora la prima dimensione
quindi di fatto appare:

```
void stampaMatrici(int mat[][3]) { ... }
```

ci posso però passare anche il numero di righe

```
void stampaMatrici(int mat[][3], int R) {
```

```
    for(int i=0; i<R; i++)
```

```
        for(int j=0; j<3; j++)
```

```
            cout << mat[i][j] << ' ';
```

```
        cout << endl;
```

```
}
```

questa potrebbe essere
anche una possibile
notazione globale

PRO: questa tecnica permette di usare la notazione più intuitiva ($m[i][j]$)

CONTRO: questa tecnica permette di stampare solo matrici con 3 colonne

Per riuscire a stampare con un'unica funzione matrici di dimensione generica,
l'unica soluzione è:

```
void stampaMatriceGenerica(int *mat, int r, int c) {
```

```
    for(int i=0; i<r; i++) {
```

```
        for(int j=0; j<c; j++)
```

```
            cout << mat[i*c+j] << ' ';
```

```
        cout << endl;
```

```
    }
```

```
}
```

⇒ NEL MAIN:

```
int m[4][5];
```

```
stampaMatriceGenerica(m[0][0], 4, 5);
```

• DEVO SFRUTTARE L'ARITMETICA DEI PUNTATORI

PROBLEMA:

Scrivere una funzione che legga una sequenza di valori non negativi e termina la lettura non appena si incontra il primo valore negativo.

LISTA

È una concatenazione di elementi informativi dello stesso tipo.

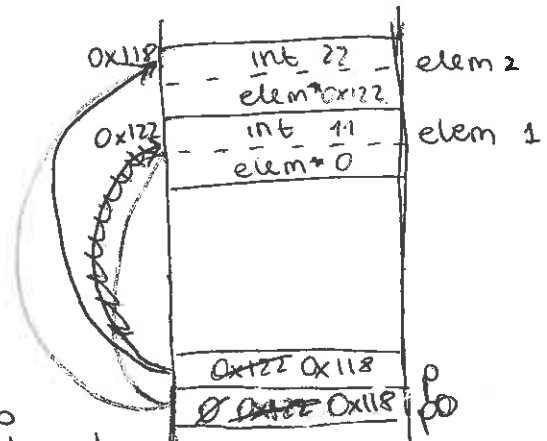
```
struct elem
{
    int inf;
    elem* pun;
};
```

L'idea è mettere nel campo puntatore del primo elemento l'indirizzo del secondo elemento. Nell'ultimo elemento, il campo puntatore prenderà nullptr. È fondamentale avere un puntatore che contenga l'indirizzo del primo elemento: è il punto di accesso alla lista.

CREAZIONE DI UNA LISTA

Si vuole creare una lista di n interi letti da tastiera.

```
elem* crealista (int n) { // RESTITUISCE UNA LISTA
    elem* p0 = 0; elem* p; // cioè l'indirizzo dove si trova il primo elemento
    for (int i = 0; i < n; i++)
    {
        p = new elem;
        cin >> p->inf; // = (*p).inf
        p->pun = p0; p0 = p;
    }
    return p0;
}
```



In p si ~~mette~~ mette l'indirizzo dell'elemento che si crea. $p0$ contiene l'indirizzo del primo elem.

Questo è la tecnica dell'inserimento in testa.
Questo tipo di struttura dati è estremamente flessibile.

```
void stampaLista (elem* p0) {
```

```
    elem* p = p0;
    while (p != 0) {
        cout << p->inf << ' ';
        p = p->pun; // aggiorno p con l'indirizzo del secondo elemento
    }
    // Lo scopo è scorrere la lista fino a che p non incontra
    // l'ultimo elemento della lista e punta a 0.
    Gli elementi vengono creati in tempo di esecuzione, perciò le liste possono
    avere tutti gli elementi che si vogliono.
```

⇒ NEL MAIN:

```
elem* L = nullptr;
L = crealista(10);
stampaLista(L);
```


FUNZIONI SUE LISTE

```
struct vagoncino {  
    int inf;  
    vagoncino * succ;  
};
```

```
vagoncino * leggiInteriNonNegativi() {  
    vagoncino * p0 = nullptr;  
    while (true) {  
        int aux;  
        cin >> aux;  
        if (aux < 0)  
            return p0;  
        vagoncino * p = new vagoncino;  
        p->inf = aux;  
        p->succ = p0;  
        p0 = p;  
    }  
}
```

```
// VARIANTE  
// void leggiInteriEcc(vagoncino *p0) {  
//     p0 = nullptr;
```

```
// break;
```

```
void stampaLista(vagoncino * p0) {  
    vagoncino * p = p0;  
    while (p != nullptr) {  
        cout << p->inf << " ";  
        p = p->succ;  
    }  
}
```

```
void deallocaLista(vagoncino *p0) {  
    vagoncino *q = p0; vagoncino *aux;  
    while (q != nullptr) {  
        aux = q->succ;  
        delete q;  
        q = aux;  
    }  
    p0 = nullptr; // la lista è vuota
```

=> NEL MAIN:

```
int main() {  
    vagoncino * L = nullptr;  
    L = leggiInteriNonNegativi(); // leggiInteriEcc(L);  
    stampaLista(L); deallocaLista(L);  
    // inserisco 11 22 33 44 -5  
    // l'output sarà 44 33 22 11
```

↑
in questo modo la
lista viene deallocata
sia dal punto di
vista logico che fisico

Se si perde l'indirizzo del primo elemento:

- Non si può avere più accesso alla lista
- Non si può deallocarla (occupa memoria)

FUNZIONI SUE LISTE

39

```
void intesta (elem * &p0, int a){
```

```
    elem *p = new elem;
```

```
    p->inf = a;
```

```
    p->pun = p0;
```

```
    p0 = p;
```

```
}
```

```
bool esttesta (vagoncino * &p0, int &a){
```

```
    if (p0 == nullptr)
```

// p0 è l'unico a contenere l'indirizzo

```
        return false;
```

// del primo elemento della lista

```
    vagoncino *p = p0->succ;
```

```
    a = p0->inf;
```

```
    delete p0;
```

```
    p0 = p;
```

```
    return true;
```

```
}
```

Se la funzione torna un risultato decente quando la lista è vuota o quando contiene un solo elemento (quindi nei casi particolari) allora è TOP.

==> NEL MAIN:

```
int val;
```

```
while (esttesta)
```

```
    cout << val << " ";
```

// stampa tutta la lista

// fino a che la funzione non ritorna false

// quindi lista vuota

Voglio aggiungere un nuovo elemento alla fine della lista. L'idea è quella di scorrere la lista con due puntatori q e r: q punta al primo elemento ed r al secondo. p0 punta anch'esso al primo elemento, e rimane lì. La funzione si può fare void perché l'operazione ha sempre successo.

```
void insfondo (vagoncino * &p0, int a){
```

```
    vagoncino *p = new vagoncino; p->inf = a; p->succ = nullptr;
```

```
    vagoncino *q = nullptr; // poi recupero in q l'indirizzo dell'ultimo elemento
```

```
    vagoncino *r; // q rimane un passo indietro rispetto a r
```

```
    for (r = p0; r != nullptr; r = r->succ)
```

// alla fine q conterrà l'ind. dell'ultimo elem

```
        q = r;
```

```
    if (p0 == nullptr)
```

```
        p0 = p;
```

```
    return;
```

```
} q->succ = p;
```

==> NEL MAIN:

```
insfondo (L, 555);
insfondo (L, 666);
insfondo (L, 777);
stampaLista (L); // 555 666 777
```

```
bool estfondo (vagoncino *&p0; int &a){
    vagoncino *p = nullptr; *q = nullptr;
    if (p0 == nullptr)
        return false;
    for (q = p0; q->succ != nullptr; q = q->succ)
        p = q;
    if (p != nullptr) { // caso lista con più elementi
        a = q->inf;
        delete q;
        p->succ = nullptr;
    } else { // caso lista con un solo elemento
        a = p0->inf;
        delete p0;
        p0 = nullptr;
    }
}
```

==> NEL MAIN:

```
int val2;
while (estfondo (L, val2))
    cout << "Valore estratto dal fondo " << val2 << endl;

// 777
// 666
// 555
```

Vedi altra versione sui lucidi

PROBLEMA

Supponiamo di avere una lista ordinata di elementi.
Voglio inserire un nuovo elemento in maniera ordinata.

void inserimentoOrdinato(vagoncino *&p0, int a){

// l'idea è: avere un puntatore r che punta all'elemento da inserire
// e avere anche un puntatore p che si ferma all'elemento che deve precedere
// quello da aggiungere e un puntatore q che punta all'elemento che
// segue quello da aggiungere

vagoncino *r = new vagoncino;

r->inf = a;

vagoncino *p = nullptr, *q = nullptr;

for (q = p0; q != nullptr && q->inf < a; q = q->succ)

p = q;

r->succ = q;

if (p != nullptr)

p->succ = r;

else {

if (p0 == nullptr) {

r->succ = nullptr;

p0 = r;

} else {

r->succ = p0;

p0 = r;

}

}

grazie alla REGOLA DELLA SCORCIATOIA
questa condizione non viene valutata
se q=0 e si deve inserire qualcosa
alla fine

// p è nullptr quando non eseguo neanche una volta il
// for e quindi q = nullptr e la lista è vuota oppure
// la seconda condizione è falsa e l'elemento da
// aggiungere va messo come primo elemento.

→ versione semplificata (più corta)
sui libri

REGOLA DEL CORTOCIRCUITO (DELLA SCORCIATOIA) - SHORTCUT RULE

Esiste solo per gli operatori logici binari AND e OR.

La regola dice: nel momento in cui ho var1 || var2 e var1 fosse vera,
mi viene ritornato true senza valutare var2 (per non perdere tempo).

Se var1 risulta falsa, allora viene valutata anche var2.

Per quanto riguarda l'AND: se abbiamo var1 && var2, se var1 è false

var2 non viene valutata.

Si vuole eliminare un elemento all'interno di una lista.

```
bool estrazione (elem * &p0, int a) {
    elem *p=0; elem *q;
    for (q=p0; q!=0 && q->inf != a; q=q->pun)
        p=q;
    if (q==0) // CASO LISTA VUOTA o ELEMENTO NON TROVATO
        return false;
    if (q==p0) // ESTRAZIONE IN TESTA
        p0 = q->pun;
    else
        p->pun = q->pun;
    delete q;
    return true;
}
```

Se la lista è ordinata, la funzione si può ottimizzare.

```
bool estrazioneOrdinata (elem * &p0, int a) {
    elem *p=0; elem *q;
    for (q=p0; q!=0 && q->inf < a; q=q->pun) {
        p=q;
        if ((q==0) || (q->inf > a))
            return false;
        if (q==p0)
            p0 = q->pun;
        else
            p->pun = q->pun;
        delete q;
        return true;
    }
}
```

Data una lista di caratteri, si vogliono eliminare tutte le occorrenze di un carattere.

int estrai tutte le Occorrenze (elem \neq p0, char a); DA IMPLEMENTARE

È possibile creare una struct con due puntatori di cui uno punta al primo elemento di una lista e l'altro punta all'ultimo elemento.

```
struct lista_n {
```

```
    elem *p0;
```

```
    elem *p1;
```

```
};
```

PILA

64

Insieme ordinato di elementi di tipo uguale, in cui è possibile effettuare operazioni di inserimento / estrazione secondo la regola di accesso LIFO. L'ultimo dato inserito è il primo ad essere estratto.

```
const int DIM = 5;
```

```
struct pila {
```

```
    int top;
```

```
    int stack[DIM];    // DIM è il numero max di elementi nella pila
```

```
}
```

top serve ad indicare il numero di elementi che ci sono nella pila. Se $top == -1$, la pila è vuota. Se $top == DIM - 1$ la pila è piena.

```
void inip(pila & pp) {
```

```
    pp.top = -1;
```

```
}
```

```
bool empty(const pila & pp) {
```

```
    if (pp.top == -1) return true;
```

```
    return false;
```

```
}
```

```
bool push(pila & pp, int s) {
```

```
    if (full(pp)) return false;
```

```
    pp.stack[++(pp.top)] = s;
```

```
    return true;
```

```
}
```

```
void stampa(const pila & pp) {
```

```
    cout << "Elementi contenuti nella pila:" << endl;
```

```
    for (int i = pp.top; i >= 0; i--)
```

```
        cout << '[' << i << "]" << pp.stack[i] << endl;
```

```
}
```

⇒ NEL MAIN:

```
pila st;
```

```
inip(st);
```

slide 318

[viene passata per riferimento perché altrimenti si dovrebbe fare la copia membro a membro (problematico se DIM non fosse 5 ma 1000). Quindi è poi anche const, per evitare la modifica.]

```
bool full(const pila & pp) {
```

```
    if (pp.top == DIM - 1) return true;
```

```
    return false;
```

```
}
```

```
bool pop(pila & pp, int & s) {
```

```
    if (empty(pp)) return false;
```

```
    s = pp.stack[(pp.top)--];
```

```
    return true;
```

```
}
```

VISIBILITA'

45

Per programmi complessi si creano diversi file sorgente.

La visibilità (scope) è il campo di visibilità di un identificatore (parte di programma in cui l'identificatore può essere usato).

Le regole di visibilità servono a controllare la condivisione delle informazioni fra i vari componenti di un programma: permettono a più parti del programma di riferirsi ad una stessa entità.

L'UNITA' DI COMPILAZIONE è costituita da un file sorgente e dai file inclusi mediante direttive `#include`. Se il file da includere non è di libreria, il suo nome va racchiuso tra virgolette.

OPERATORE :: UNARIO DI RISOLUZIONE DI VISIBILITA'

```
int i = 1;
int main() {
    cout << i << endl;    // 1
    {
        int i = 5;
        cout << ::i << '\t' << i << endl;    // 1 5
    }
    cout << ::i << endl;    // 1
    return 0;
}
```

SPAZIO DI NOMI

È un insieme di dichiarazioni e definizioni racchiuse tra parentesi graffe, ognuna delle quali introduce determinate entità dette membri. Può essere dichiarato solo a livello di unità di compilazione o all'interno di un altro spazio dei nomi.

Gli identificatori relativi ad uno spazio dei nomi sono visibili dal punto in cui sono dichiarati fino alla fine dello spazio dei nomi.

`using namespace std;` // SPAZIO DEI NOMI STANDARD DEFINITO IN LIBRERIA <IOSTREAM>

Lo spazio globale è uno spazio di nomi con nome vuoto.

La direttiva `using` serve ad informare il compilatore che da quel momento in poi si sta usando uno specifico spazio dei nomi, che poi esso andrà a cercare nello spazio dei nomi globale o in altri spazi di nomi.

Lo spazio dei nomi è aperto, chiunque può metterci altre entità.

COLLEGAMENTO

Un programma complesso è conveniente implementarlo usando più file sorgente, quindi più unità di compilazione.

Si distinguono:

- IDENTIFICATORI CON COLLEGAMENTO INTERNO sono utilizzati solo in quella specifica unità di compilazione in cui sono definiti.
- IDENTIFICATORI CON COLLEGAMENTO ESTERNO possono essere utilizzati anche in altre unità di compilazione (identificatori = oggetti e funzioni).

Regole di default:

- gli identificatori con visibilità locale hanno collegamento interno (cioè, tutte le variabili locali, o di blocco, hanno collegamento interno)
- gli identificatori con visibilità a livello di unità di compilazione hanno collegamento esterno (a meno che non siano dichiarati const), come ad esempio le variabili globali. Le costanti globali però non potranno essere utilizzate in altre unità di compilazione.

Anche se si fa uso di più unità di compilazione, se nei diversi file sorgente viene utilizzato un identificatore con lo stesso nome, inizialmente la compilazione avverrebbe senza dare alcun errore, ma, nel momento in cui viene chiamato il linker che ~~ha~~ ha il compito di unire i due file oggetto "fondendoli" in un unico file eseguibile (main.exe), si accorge che ci sono due entità globali con lo stesso nome che creano AMBIGUITÀ. L'eseguibile allora non viene generato, non compilerebbe. Perciò è possibile dichiarare una variabile senza definirla in questo modo:

`| extern int a;`

È quindi chiaro che, per quanto riguarda gli oggetti con lo stesso nome ~~non~~ definiti con collegamento esterno ce ne deve essere uno soltanto.

Una COSTANTE ha collegamento interno, quindi non potrà mai essere esportata in un'altra unità di compilazione. Se ci sono due costanti globali con lo stesso nome, esse non entreranno in conflitto in fase di linkaggio.

La parola chiave **STATIC** serve a dire che una variabile o una funzione ha collegamento interno.

Anche le strutture hanno collegamento interno.

HEADER FILE = FILE D'INTESTAZIONE

Ci sono delle situazioni in cui si ha la necessità di condividere la stessa costante avendo lo stesso valore o qualsiasi tipo di oggetto avendo la garanzia che sia identico.

Bisogna quindi dotarsi di un header file in cui verranno messe tutte le definizioni delle costanti, le dichiarazioni delle funzioni e le strutture dati per poi includerlo in entrambi i file sorgente.

(Le file che si intendono aggiungere nella finestra dei TARGET di CMake, sono quei file che si intende compilare. L'header file va incluso, non per forza compilato.)

```
#include "header.h" // va con le virgolette che perché il file è contenuto
                    // nella stessa directory del file.cpp su cui si
                    // sta scrivendo
```

- 1) il Pre-processor forma l'UNITÀ DI COMPILAZIONE dai file sorgente
- 2) Compilatore
- 3) Linker

Sauvere una funzione che tenga traccia di quante volte è stata chiamata.

- Abbiamo bisogno di:
- una variabile che come tempo di vita abbia lo stesso di quello di una variabile globale
 - una variabile che abbia visibilità uguale a quella di una variabile locale

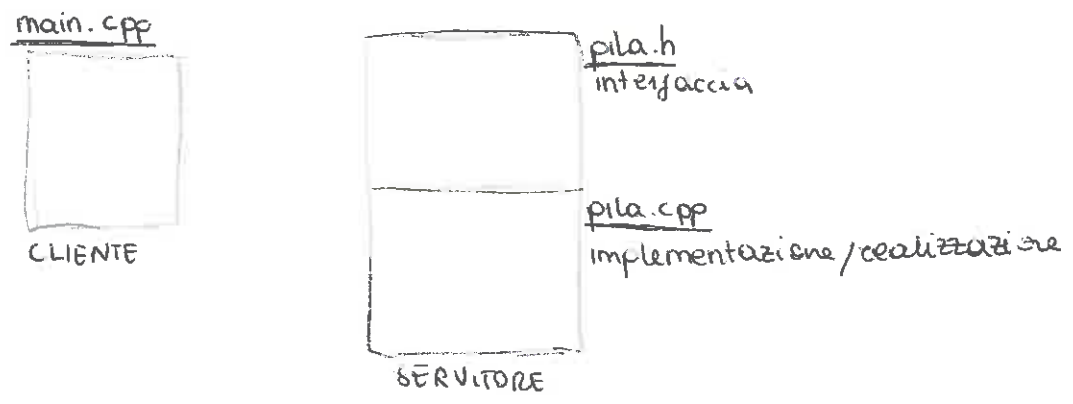
⇒ variabile STATIC

La parola chiave static:

1. messa davanti ad una variabile globale, ne cambia le regole di collegamento
2. messa davanti ad una variabile locale ad una funzione ne modifica la classe di memorizzazione ~~ad~~ da automatica a statica (quindi il tempo di vita) e la visibilità ha la stessa delle variabili di blocco, locali.
3. viene utilizzata nelle classi con un'altra semantica

PROGRAMMAZIONE A MODULI

Astrazione CLIENT - SERVER



MODULO = parte di programma che svolge una particolare funzionalità e che risiede su uno o più file (moduli servitori, moduli clienti)

La separazione tra INTERFACCIA e IMPLEMENTAZIONE ha per scopo

1. INFORMATION HIDING:
- semplifica le dipendenze tra i vari moduli (cioè è possibile migliorare l'implementazione del servizio, senza toccare l'interfaccia)

per compilare a mano da il comando:

```
g++ -c main.cpp -o main.o
      ↑
      chiama il compilatore
      lo dice main.cpp

      ↑
      chiama il linker
      lo dice main.o
```

per linkare:

```
g++ main.o pila.o -o main.exe
```

CLASSI

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

```
const int DIM=5;
```

```
class ClassePila {
```

```
private:
```

```
    int top;
```

```
    int stack[DIM]; // membri dato
```

```
public:
```

```
    bool full();
```

```
    bool empty();
```

```
    void inip();
```

```
// membri funzione
```

```
};
```

```
int main() {
```

```
    ClassePila s1;
```

```
    s1.inip(); // inip(s1);
```

COSTRUTTORE

È una funzione membro senza tipo di ritorno. È possibile definirne più di uno.

⇒ NEL MAIN:

```
ClassePila s1; // è come scrivere s1.ClassePila();
```

Il suo compito è quello di portare la struttura dati ad uno stato consistente.

Nella classe è scritta: `ClassePila() { top = -1; }`, con lo stesso nome della classe.

Fuori dalla classe, si definisce: `ClassePila::ClassePila() { top = -1; }`

La classe mette tutti i membri in uno spazio dei nomi che coincide col nome della classe.

Con le classi è possibile estendere il linguaggio dal punto di vista dei tipi, purché si riesca a specificare:

- 1) Quanto occupa il tipo classe (sizeof di ogni membro dato)
- 2) Quali sono i valori che può assumere un'istanza generica del tipo della nostra classe
- 3) Quali sono le operazioni possibili (funzioni, ...)

Membru della classe:

- tipo (enumerazione o struttura)
- campo dato (oggetto non inizializzato)
- funzione (dichiarazione o definizione)
- un'altra classe (diversa da quella a cui appartiene) (meno importante)

ESEMPIO:

```
class complesso {
private:
    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 << ")"; }
};
```

// l'utilizzatore non sa il nome di questi campi
 // se decidesse di cambiarli, l'utilizzatore non deve ricompilare
 implementare diversamente

FUNZIONI
 INLINE

⇒ NEL MAIN:

```
complesso c1, c2;
c1.iniz_compl(1.0, -1.0);
c1.scrivi(); // (1.0, -1.0)

complesso *cp = &c1;
cp->scrivi(); // (1.0, -1.0)
// (*cp).scrivi();
```

per i puntatori il tipo serve per stabilire la variabile a cui devono puntare dato che il puntatore di un tipo può puntare solamente a una variabile dello stesso tipo

Le funzioni membro che sono implementate all'interno della classe sono dette INLINE.

La chiamata di funzione è un meccanismo che parte a tinte. Se la funzione è INLINE, il compilatore sostituisce a ogni chiamata il corpo della funzione, aumentando il file eseguibile ma diminuendo il tempo di esecuzione.

Un oggetto appartenente alla classe si chiama oggetto classe o istanza di tipo classe.

⇒ NEL MAIN:

```
complesso c1;  
c1.iniz-comp1(1.0, -1.0);  
complesso c2 = c1;  
complesso c3(c1);
```

il compilatore:
- alloca sizeof(complesso) sullo stack
- aggiunge due linee di codice
cioè: `{ c2.re = c1.re;
c2.im = c1.im; }` ↓
"ricopiatura membro a membro"

```
complesso *pc1 = new complesso(c1); // sullo heap cerca sizeof(complesso)  
// e poi fa la ricopiatura membro  
// a membro  
// int *pi = new int(3);
```

Nelle parentesi tonde va una variabile dello stesso tipo per cui si sta allocando memoria. Questo se si vuole dichiarare e definire la variabile in una sola riga di codice.

ESERCIZIO

Lato cliente, aggiungere una funzione che calcoli la somma tra due complessi:

```
complesso somma(complesso a, complesso b){  
    complesso c;  
    c.iniz-comp1(a.reale() + b.reale(), a.immag() + b.immag());  
    return c;  
}  
// L'ho implementato senza applicare i meccanismi della classe  
// Lato cliente, appunto
```

MECCANISMO RUN-TIME

Al momento della chiamata della funzione scatta il meccanismo a run-time nel quale bisogna allocare sullo stack gli argomenti formali, allocare le variabili locali, poi assegnare alla variabile nel main il risultato (a meno che non è void) e poi deallocare tutte le variabili che si erano create.

PUNTATORE THIS

52

=> NEL MAIN:

```
// STRUTTURE DATI
```

```
pila p1;  
inip(p1);  
push(p1, 3);
```

```
// CLASSE
```

```
classe Pila p2;  
p2.inip(); // cambia il modo di invocare le funzioni  
p2.push(3); // tramite l'operatore selettore di membro  
// cosa c'è dietro a questo cambio di sintassi?
```

```
classe Pila p3;
```

```
p3.inip(); // ClassePila::inip(&p3); -> viene vista dal compilatore così
```

```
p3.push(3);
```

```
cout << &p3; // 0x333
```

IL COMPILATORE AGGIUNGE SEMPRE QUESTO ARGOMENTO:
(NON ESPLICITAMENTE)

=> FUNZIONE:

```
void ClassePila::inip(ClassePila *this) {
```

```
    // top = -1;
```

```
    // (*this).top = -1;
```

```
    this->top = -1;
```

```
    cout << this; // 0x333
```

```
}
```

SI TRATTA DI UN PUNTATORE
ALL'ISTANZA CLASSE SULLA
QUALE VOGLIAMO CHE LA
FUNZIONE OPERI IN QUEL
MOMENTO

THIS è un puntatore costante: garantisce il fatto che pointer sempre
a quell'istanza. Infatti non si può neanche modificare manualmente.

```
class complesso {
```

```
    /* ... */
```

```
    complesso scala(double s) {
```

```
        re * s;
```

```
        im * s;
```

```
        return *this;
```

```
    }
```

```
};
```

=> NEL MAIN:

```
complesso c1;
```

```
c1.iniz_compl(1.0, -1.0);
```

```
c1.scale(2); // (2, -2)
```


Potrebbe essere comodo concatenare un set di comandi:

```
c1.iniz-comp(1,-1);
c1.scala(2);
c1.scrivi(); // (2,-2)
c1.scala(2);
c1.scrivi(); // (4,-4)
```

1^a VERSIONE
SEQUENZIALE

```
c1.iniz-comp(1,-1).scala(2).scrivi().scala(2).scrivi(); ] 2a VERSIONE
// MANIERA CONCATENATA
```

La 2^a versione NON compila perché: l'operatore `selezione membro` è associativo da sinistra a destra e dopo aver inizializzato `c1`, restituisce un `void`, quindi poi andremmo a scalare qualcosa che non è un complesso. Supponiamo che l'oggetto sia già inizializzato:

```
complesso scala(double s) { // restituisce un valore
    this->re *= s;
    this->im *= s;
    return *this;
}
```

Scrivere `c1.scala(2).scrivi().scala(2).scrivi();`
è uguale a `(2,-2)`.

Questo perché la funzione `scala` "salva" in una variabile temporanea il valore scalato.

`complesso tmp = c1.scala(2);` → `c1` viene scalato, è il suo valore viene dato a `tmp`
 ⇒ `tmp.scala(2); // (4,-4)` → questo è lo step successivo, `tmp` scala
`c1.scala(2); // (2,-2)` → infatti `c1` rimane come prima

La funzione si riferiva ad operare su `tmp` e non più su `c1`.

È quindi necessario che questo oggetto non venga ritornato per valore (copia membro a membro) ma che venga ritornato per riferimento, quindi una variabile temporanea che sia un altro nome per `c1`.

```
complesso &tmp = c1.scala(2);
```

```
complesso & scala(double s) { // restituisce un riferimento
    re *= s;
    im *= s;
    return *this;
}
```


È possibile creare due funzioni per calcolare la radice quadrata con lo stesso nome e argomento diverso.

Questo perché il compilatore riesce a capire quali delle due usare in base al parametro che gli stiamo passando.

È possibile scrivere, in fase di dichiarazione di funzione, un valore da assegnare agli argomenti formali (valore di default), a patto che questi vengano definiti per ultimi.

Nell'esecuzione, il compilatore giunto alla chiamata se nota che manca qualche parametro, ~~lo~~ completa la chiamata utilizzando i valori di DEFAULT.

VISIBILITA' A LIVELLO DI CLASSE

Gli identificatori dichiarati all'interno di una classe sono visibili dal punto della loro dichiarazione fino alla fine della classe (membri dati).

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

All'esterno della classe possono essere resi visibili mediante l'operatore `::` applicato al nome della classe:

- funzioni membro (quando vengono definite)
- un tipo o enumeratore (se dichiarati in public)*
- membri statici

* \Rightarrow NELLA CLASSE:

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

\Rightarrow NEL .cpp :

```
grafica::colore punto;
```

Nel caso si dovessero creare due classi annidate, per ^(implementare) rendere visibili i membri public occorre utilizzare due operatori di visibilità. Inoltre, nessuna delle due classi può accedere alla parte privata dell'altra.

MODULARITÀ E RICOPILAZIONE NUOVE CLASSI

55

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

Quando si modifica la parte dati di una classe, si devono ricompilare gli altri moduli ma non il codice lato cliente se si tratta di correggere l'implementazione. Se invece si modifica il ^{eff} `sizeof` della classe va ricompilato anche il ⁱⁿ `main`.

CONSTRUTTORE

È una funzione membro il cui nome è il nome della classe. Se definita viene invocata automaticamente tutte le volte che viene creata un'istanza classe (subito dopo che è stata riservata memoria per i campi dati).

⇒ NUOVA CLASSE:

```
class complesso {  
    double re, im;  
public:  
    complesso(double r, double i);  
    /* ... */  
}
```

⇒ NEL MAIN:

```
complesso c1; // NON COMPILEREBBE perché  
              // il compilatore vede che questo pezzo di  
//ERRORE: codice così { allora memoria per la classe  
// non esiste complesso() { invoca la funzione costruttore su c1  
                          c1.complesso();
```

In questo caso però il compilatore chiama la funzione costruttore se ci sono specificati i due parametri. Allora occorre cambiare sintassi:

```
complesso c1(1.0, -1.0); { si alloca memoria  
                        { si invoca il costruttore correttamente
```

(il costruttore quindi sarà sempre inevitabile creare una ^{istanza della classe} ~~variabile~~ e inizializzarlo subito).

```
complesso c3(1); //ERRORE: non esiste complesso(int);
```

COSTRUTTORI DEFAULT (senza parametri)

Grazie al ¹meccanismo dell'overloading di funzioni è possibile definire più di un costruttore

=> NUOVA CLASSE:

```
class complesso {
    double re, im;
public:
    complesso();
    complesso(double r, double i);
    /* ... */
}
```

DUE COSTRUTTORI

=> FILE.cpp:

```
complesso::complesso() { re=0; im=0; } // COSTRUTTORE DEFAULT
complesso::complesso(double r, double i) { re=r; im=i; }
```

=> NEL MAIN:

```
complesso c1(1.0, -1.0);
complesso c2; // torna questa sintassi, viene chiamato il costruttore
               // di default
```

Non è corretta la chiamata al costruttore default `complesso c2();` perché ~~non~~ viene scambiato per una dichiarazione di funzione `c2`, senza argomenti con ritorno un oggetto complesso.

Soluzione migliore: ²meccanismo degli argomenti di default

=> NUOVA CLASSE:

```
class complesso {
    ....
public:
    complesso(double r=0, double i=0);
    /* ... */
}
```

Con questa sintassi si può scrivere

```
complesso c1(1.0, -1.0);    complesso c3(3.0);
complesso c2;               complesso c4 = 3.0;
senza implementare due funzioni.
```

Si vuole creare una classe stringa che occupi la memoria necessaria.

```
class Stringa {  
    char *str;  
public:  
    Stringa(const char s[]);  
};
```

⇒ NEL MAIN:

```
Stringa s1;
```

- si alloca sullo stack sizeof di stringa
- viene chiamato il costruttore che si ritrova a dover allocare ~~solo~~ un campo puntatore a una stringa sullo heap

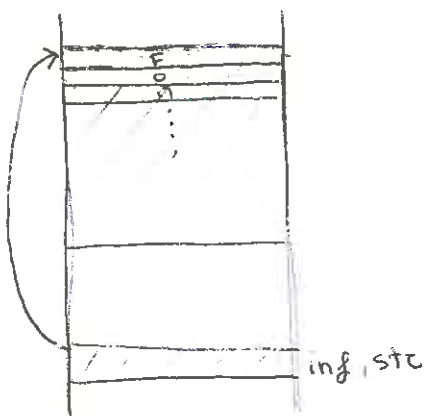
```
Stringa inf("Fondamenti di Progt."); // occupa lo spazio che serve a  
// contenere la stringa+1, più  
// sizeof(char*)
```

⇒ NEL FILE .cpp:

```
Stringa::Stringa(const char s[]) {  
    str = new char[strlen(s)+1];  
    strcpy(str, s); // copia s in str  
}
```

Il compito di un costruttore è quello di portare ad uno stato consistente la struttura dati.

Per le classi che hanno bisogno di utilizzare memoria dinamica senza compito del costruttore allocare anche la parte della classe che si trova in memoria dinamica.



// è la memoria necessaria

Altra soluzione (sbagliata):

```
class stringa {
    char str[n];    // con n da specificare dopo
public:
    /~ ~/
```

È sbagliata perché: nelle strutture dati non si può utilizzare (in un vettore) una costante non nota a tempo di compilazione.

- Il sizeof di una classe restituisce sempre la dimensione statica della classe.

CONSTRUTTORI PER OGGETTI DINAMICI

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

⇒ NEL MAIN:

```
complesso *pc1 = new complesso (3.0, 4.0);
```

DISTRUTTORE

È una funzione membro che consente di deallocare la memoria dinamica che si è allocata nel costruttore. Non ha argomenti.

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

⇒ NEL FILE.cpp:

```
stringa::~stringa() {
    delete [] str; // delete [] this->str;
}
```

⇒ NEL MAIN:

```
stringa *ps = new stringa("Fondamenti di Progc.");
/* ... */
```

```
delete ps;
```

Ci deve essere una doppia deallocazione: una che fa il distruzione nella memoria dinamica e un'altra che si fa nel main sul puntatore.

```
class B;
```

```
class A {
```

```
    friend class B; //oppure nel caso di funzione?
```

```
    // friend void B::f();
```

```
public:
```

```
    /* ... */
```

```
};
```

```
class B {
```

```
    A a;
```

```
public:
```

```
    void f();
```

```
    /* ... */
```

I costruttori vengono chiamati con le seguenti regole:

1. Per gli oggetti statici all'inizio del programma (variabili globali di tipo classe)
2. Per gli oggetti automatici, quando viene incontrata la definizione (all'interno del blocco)
3. Per gli oggetti dinamici, quando viene incontrato l'operatore new
4. Per gli oggetti membri di altri oggetti, quando questi ultimi vengono costruiti

I distruttori vengono chiamati:

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

Gli oggetti con lo stesso tempo di vita vengono distrutti nell'ordine inverso a quello in cui sono stati definiti.

CONSTRUTTORI DI COPIA

Esiste una versione predefinita che effettua la ricopiatura membro a membro degli oggetti.

Viene applicato:

1. Quando un oggetto classe viene inizializzato con un altro oggetto della stessa classe pre-esistente

ESEMPIO: `complesso c2(c1);` $\left. \begin{array}{l} \text{alloca memoria} \\ c2.\text{complesso}(c1); \end{array} \right\}$

2. Quando un oggetto classe viene passato ad una funzione come argomento valore
3. Quando una funzione restituisce come valore un oggetto classe (mediante l'istruzione return).

È un normale costruttore ma con un unico argomento ^{riferimento costante} dello stesso tipo della classe.

Il costruttore di copia deve essere ridefinito per quelle classi che utilizzano memoria libera.

Per impedire l'utilizzo del costruttore di copia predefinito occorre inserire la sua dichiarazione nella parte privata della classe stessa senza alcuna ridefinizione. Nel caso in cui il costruttore di copia venga nascosto non si possono avere funzioni che abbiano argomenti valore del tipo della classe o un risultato valore del tipo della classe (NASCEREMENTO DEL COSTRUTTORE DI COPIA).

```

class A {
public:
    A(); // costruttore senza parametri, predefinito
    A(int, int); // costruttore con due parametri
    A(const A&); // costruttore di copia
    ~A(); // distruttore
};

```

Se in una classe non dovessi mettere il costruttore, esiste comunque un costruttore predefinito ^{messo dal compilatore} vuoto, senza argomenti e senza corpo. Così come anche per il costruttore di copia.

FUNZIONI FRIEND

La parola chiave FRIEND è utilizzata solo nel contesto delle classi. Sta ad indicare che quella funzione può accedere a membri pubblici e privati della classe usando i selector di membro.

```

class complesso {
    double re, im;
public:
    double reale();
    double immag();
    friend complesso somma(const complesso &a, const complesso &b);
    /* ... */
};

```

⇒ NEL file MAIN.cpp

```

complesso somma(const complesso &a, const complesso &b) {
    complesso s(a.re + b.re, a.im + b.im); // si accede ai membri privati
    return s; // della classe
}
// è definita globalmente

```

Viola il principio dell'information hiding.

OVERLOADING DEGLI OPERATORI

62

=> NEL MAIN:

```
// c = a + b;
```

```
c = operator + (a, b);
```

Il compilatore vede la prima operazione come una chiamata di funzione.

Definita come:

```
int operator + (int, int) { }
```

ESEMPIO: => COME FUNZIONE MEMBRO

```
class complesso {
```

```
    double re, im;
```

```
public:
```

```
    complesso () { re = im = 0; }
```

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

```
    complesso operator + (double b) // voglio fare c = a + b  
        // this punta ad a
```

```
        complesso s (this->re + b.re, this->im + b.im)
```

```
    } return s;
```

=> COME FUNZIONE GLOBALE:

```
complesso complesso::operator + (const complesso &x, const complesso &y) {
```

```
    complesso z (x.re + y.re, x.im + y.im);
```

```
    } return z;
```

Nella classe questa funzione sarà dichiarata come FRIEND.

=> NEL MAIN

```
c3 = c1 + c2; // c3 = c1.operator + (c2);
```

Gli operatori (non tutti) possono essere ridefiniti come funzioni globali o funzioni membro della classe.

L'operatore di assegnamento può essere definito solo come funzione membro.

OPERATORE DI ASSEGNAMENTO

È una funzione membro predefinita che può essere ridefinita solo come funzione membro (in caso di allocazione dinamica della memoria).

⇒ NEL MAIN:

complesso c1, c2;

c1 = c2; // operatore di assegnamento

~~~~~

complesso c1;

complesso c2 = c1; // costruttore di copia

// perché siamo in fase di costruzione di c2

diverso da

Gli operatori che non si possono ridefinire sono:

- l'operatore risoluzione di visibilità (::)
- l'operatore selezione di membro (.)
- l'operatore selezione di membro attraverso un puntatore a membro (\*.)

Alcuni operatori possono essere ridefiniti solo come funzioni membro:

- assegnamento =
- indicizzazione []
- chiamata di funzione ()
- selezione di membro tramite puntatore →

NB: oltre all'operatore di assegnamento sono predefiniti anche quello di indiritto (&) e di sequenza (,).

## OPERATORI DI INCREMENTO

⇒ NELLA CLASSE:

complesso & operator++();

complesso operator++(int); // l'argomento serve a differenziare gli operatori e identifica quello postfix

- Sono due modi diversi di dichiarare e implementare una funzione. Il primo consente la concatenazione, il secondo no.

⇒ NEL .cpp:

complesso & complesso::operator++() {

→ migliore

re++; im++;  
} return \*this;

complesso complesso::operator++(int) {

→ non ha senso

complesso app = \*this;  
re++; im++;  
} return app;

Ancora per quanto riguarda l'OPERATORE DI ASSEGNAZIONE, esso deve:

- 1) deallocare 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.

ESEMPIO:

==> NELLA CLASSE:

```
class stringa {
    char *str;
public:
    :
    stringa & operator = (const stringa &);
    :
};
```

==> NEL .cpp

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

# CLASSI PER L'INGRESSO E L'USCITA

- **CLASSE PER L'USCITA NATI** (si trova nella libreria ostream)

```
namespace std {
    class ostream {
    public:
        // stato: configurazione di bit
        ostream(const ostream&); // COSTRUTTORE DI COPIA PRIVATO
        // non si possono creare delle copie di cout
        ostream& operator<<(int); // per impedire la data race
    };
    ostream cout, cerr; // istanze della classe ostream (UNICHE)
    FUNZIONE CHIAVE: ostream& put(char c);
```

per consentire la concatenazione /\*...\*/

viene ridefinito l'operatore di shift (associativo da sinistra)

ESEMPIO:

```
class complesso {
    double re, im;
public:
    complesso() { re = im = 0; }
    double reale() { return re; }
    double immag() { return im; }
    friend ostream& operator<<(ostream&, const complesso&); /*...*/
    ostream& operator<<(ostream& os, const complesso& c) { // FUNZIONE
        os<< '(' << c.reale() << ', ' << c.immag() << ')' // GLOBALE
        return os;
    }
};
```

non si può fare la copia

↳ per avere accesso ai membri privati

Non è definita come funzione membro di complesso perché andrebbe come funzione membro della classe ostream.

Per questo si definisce come funzione globale.

Si ritorna os, che è un riferimento all'istanza della classe (infatti è messo al posto di cout) e permette la concatenazione.

da slide 439

MANIPOLATORI slide 461-462

## • CLASSE PER L'INGRESSO DATI

```
class istream {
    // stato: configurazione di bit
    /* ... */

```

→ non si può usare una copia di cin  
anche qui il costruttore di copia è  
mascherato

public:

```
    istream() {}
```

```
    istream & operator >> (int &);
```

```
    /* ... */

```

```
} //
```

```
istream cin;
```

FUNZIONE CHIAVE: `ostream& get (char& c);`

↓

essa mette in c (in questo caso) il primo carattere disponibile nel buffer, se è vuoto attende che l'utente ci inserisca un carattere

- Si può leggere un complesso da tastiera?

Sì.

Tuttavia, mentre l'uscita non fallirà mai, per la lettura bisogna gestire tutti i casi di errore, grazie ai **BIT DI STATO**.

## COSTRUTTORI DI CONVERSIONE

Sono costruttori che hanno un solo argomento che è di tipo fondamentale (int, double, float, ...).

```
class stringa {
```

```
    char *str;
```

```
public:
```

```
    stringa(const char *s); //costruttore di conversione
```

```
    stringa(char a) {
```

```
        str = new char[2];
```

```
        str[0] = a;
```

```
        str[1] = '\0';
```

```
    },
```

```
    friend ostream& operator<<(ostream&, const stringa&); /*...*/
```

⇒ NEL MAIN:

```
    stringa s1("Fondamenti"); //converte una stringa in un oggetto stringa
```

```
    stringa s2('T');
```

Un'altra funzione potrebbe essere:

```
friend stringa operator+(const stringa s1, const stringa s2);
```

che potrei definire solo come funzione globale.

Questo perché voglio un risultato diverso da s1 e s2.

⇒ NEL MAIN:

```
    stringa s3 = s1 + s2;
```

```
    cout << s1 + s2; //cout << operator+(s1, s2);
```

```
    //FondamentiT
```

```
    cout << s1 + 'Z'; //OK
```

```
    //perché il compilatore vede s1 + stringa('Z')
```

I costruttori di conversione possono essere invocati in maniera implicita nel momento in cui il compilatore non trova il matching perfetto.

E se volessi convertire un tipo classe in un tipo fondamentale?

La conversione deve essere implementata in modo esplicito e si fa mediante gli operatori di conversione.

esempio:

=> NELLA CLASSE:

```
operator char() { // il tipo di ritorno è talmente evidente che
    return str[0]; // va omesso
}
```

=> NEL MAIN:

```
cout << char(s1); // F
char ch2 = s1; // char(s1)
```

TIPI DI ERRORI CHE POSSONO VERIFICARSI SUGLI STREAM DI INGRESSO:

- 1) Errore recuperabile (FAIL)
- 2) Errore non recuperabile (BAD)
- 3) Fine dello stream (EOF)

• Come rappresentarli?

```
bool fail
bool bad
bool eof
```

} bastano queste 3 variabili (booleane) per descrivere lo stato dello stream d'ingresso)

Tuttavia, questa soluzione spreca 3 byte (24 bit) quando sarebbero sufficienti 3 bit.

L'idea perciò è quella di usare un unsigned su 8 bit e usare i 3 bit meno significativi per rappresentarli.

ESERCIZIO

Crea un tipo enumerato che contenga questi bit che vogliamo descrivere.

```
namespace std {
    enum my_ios {
        my_failbit = 0b00000100, // quando definiamo un enumerato possiamo
        my_eofbit = 0b00000010, // specificare l'intero da associare
        my_badbit = 0b00000001 // a ciascun letterale enumerato
    };
}
```

=> NEL MAIN:

```
my_ios ss; // stream state
```

```
ss = my-failbit;
```

```
cout << my-EOFbit; // 1
```

```
cout << my-badbit; // 2 → stampa in base 10
```

```
cout << my-failbit; // 4
```

```
cout << (my-badbit | my-failbit); // 6 (OR bit a bit)
```

FUNZIONE good(): è una funzione che si trova dentro la classe istream che restituisce un booleano: true se tutti i bit di stato sono a zero, false altrimenti.

```
class istream {
```

```
    my-ios ss;
```

```
public:
```

```
    /* ... */
```

```
    bool fail(); // è una funzione che restituisce true se failbit o badbit sono a 1
```

```
    bool bad(); // essa controlla solo lo stato del badbit
```

```
    bool eof();
```

```
    bool good();
```

```
    operator bool() { return good(); } // uno stream è vero fin tanto // che tutti i bit di stato sono zero
```

```
    void clear(unsigned int s=0) { ss=s; }
```

⇒ NEL MAIN:

```
int a;
```

```
cin >> a; // a 45 → ho inserito per sbaglio a
```

```
if (!cin) {
```

```
    cin.clear(); // porta tutti i bit a zero
```

```
    char ch; // quindi cin è in stato di good
```

```
    cin >> ch; → ch contiene il carattere "a"
```

```
} cin >> a; → nuovo tentativo di leggere un intero ha successo
```

// se voglio porre lo stream in stato di errore recuperabile:

```
cin.clear(my-failbit); // porta solo il bit di fail a 1
```

// l'implementazione della funzione

// cambia in: { ss |= s; }

↓  
OR bit a bit



## MANIPOLAZIONE DEI FILE

70

Stream associati ai file visti dal sistema operativo. Sono gestiti da un'apposita libreria chiamata `<fstream>`, con la quale potremmo definire delle variabili di tipo classe `fstream`, come per esempio: `fstream ingr, usc` che vengono utilizzate per leggere il contenuto di un file Ascii salvato sul disco o per scrivere qualcosa sul disco, con lo scopo di salvare delle parti del programma importanti.

FUNZIONE open(): serve ad associare uno stream (la variabile) ad un file.

ESEMPIO: `ingr.open("nome_file");` → specificato come stringa

Quando si usa questa funzione bisogna specificare il tipo di modalità di lettura:

- **lettura** → `ios::in` (enumerato predefinito con un bit a 1)
- **scrittura** → `ios::out` ( " " )
- **append** (scrittura alla fine del file) → `ios::out | ios::app` (il file viene aperto in modalità di scrittura partendo dalla fine aggiungendo contenuto a quel file senza toccare la prima parte)

Per lo stream in lettura il risultato sarà:

```
ingr.open("file 1.in", ios::in);
```

Quando lo stream viene aperto in lettura il file deve essere già presente e il puntatore si sposta sulla prima casella

Per lo stream in scrittura il risultato sarà:

```
usc.open("file2.out", ios::out);
```

In questo caso, qual'ora il file non fosse presente, esso verrebbe creato, vuoto, ~~an~~ il cui puntatore punta all'unico elemento presente cioè alla marca di fine stringa.

Se il file esisteva già e conteneva delle informazioni, non appena vi accederai in scrittura il contenuto si cancellerebbe.

Stream aperto in append:

Il file se non presente viene creato. In questo caso il puntatore si sposta alla fine dello stream in corrispondenza della marca di fine stream ed eventuali dati presenti nel file non vengono perduti.

Alla chiamata della funzione `open` bisogna inserire il nome del file, esso viene cercato nella directory corrente e in sua assenza viene creato un file vuoto, sempre nella stessa directory. In alternativa, al posto del nome ci posso inserire il percorso del file.

Come faccio a scrivere "10" nel file2.out? (Per il quale ho effettuato una apertura in scrittura)

```
usc << 10; //ricorda la cout
```

Questo vuol dire che dentro la classe `fstream` ci sarà un overloading dell'operatore di shift a sinistra che prende quell'intero (in complemento a 2) e lo trasforma in un binario che attraverso la `put()` stampa sul file il binario stesso.

Dovessi accedere in lettura:

```
ingr >> x;
```

si prelevano dei caratteri consecutivi nel file e il puntatore si fermerà al primo spazio bianco.

FUNZIONE `close()`: consente di "rilasciare" il file: `ingr.close();`  
`usc.close();`

La classe è anche dotata di distruttore.

Cio' significa che, quando le variabili `ingr` e `usc` dovessero uscire dal loro scope verrebbero comunque rilasciate le risorse.

In quali casi l'operazione di scrittura potrebbe fallire?

- Hard disk pieno
- caso in cui non si hanno diritti di scrittura nella directory corrente

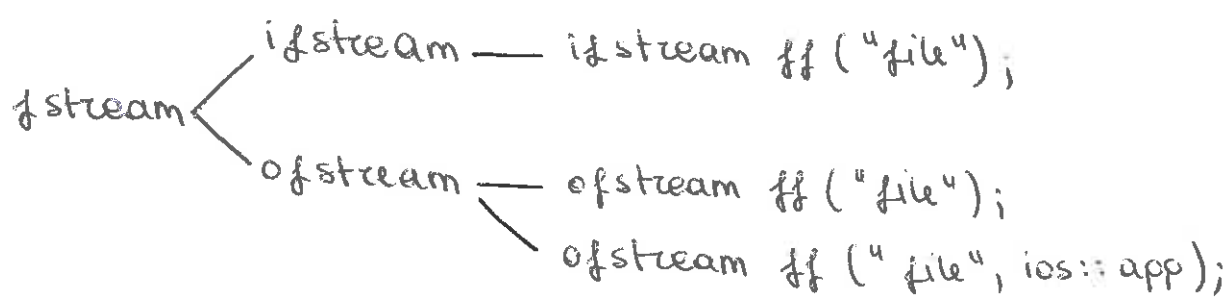
Lo stream si troverebbe in errore non recuperabile.

Il badbit verrebbe settato a 1 tramite la funzione `clear`.

Per questo, dopo aver fatto la `open()` occorre controllare se l'operazione ha avuto successo.

Inoltre, non è possibile effettuare operazioni di scrittura, lettura o append sullo stesso file contemporaneamente.

## VERSIONI ALTERNATIVE



## COSTANTI E RIFERIMENTI NELLE CLASSI

=> NELLA CLASSE:

```

class complesso {
  const double re;
  double im;
public:

```

// L'obiettivo è: istanze diverse potranno  
 // avere parti reali diverse ma ogni istanza  
 // nel corso della sua vita non può modificare  
 // la sua parte reale

// perciò viene impedito dal compilatore che la parte reale si  
 // trovi a sinistra di un assegnamento

// La versione giusta per assegnare un valore costante è:

```

complesso (double r=0, double i=0): re(r) {
  im=i;
}

```

↓  
 (oppure  
 re=r)

questa è la zona in  
 cui è possibile  
 assegnare una e una  
 sola volta un valore  
 a un membro dati  
 costante

Questa "tecnica" è detta LISTA DI INIZIALIZZAZIONE.

Si può usare per tutti i costruttori, anche quello di copia.

Caso riferimento:

=> NELLA CLASSE

```

class complesso {
  double &re;
  double im;
public:
  complesso (double &r, double i): re(r)
  { im=i; }

```

=> NEL MAIN:

```

double r1(1.0);
complesso c1(r1, 3.0);
c1.raddoppia();
cout << r1; //2.0

```

## MEMBRO CLASSE ALL'INTERNO DI CLASSI

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

```
class record {
```

stringa nome, cognome; → l'unico modo di inizializzarli è chiamare il costruttore della classe stringa

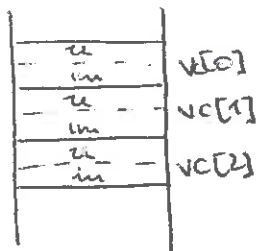
```
public:
```

```
record(const char n[], const char c[]): nome(n), cognome(c) {};
```

## ARRAY DI OGGETTI CLASSE

```
complesso vc[3];
```

Anche questa scrittura provoca la chiamata del costruttore



Per inizializzare il vettore la sintassi è:

```
complesso vc3[3] = { complesso(0.0), complesso(1.0, 1.0), complesso(2.0, 2.0) };
```

// stringa vs[5]; è un ERRORE perché non vi è un costruttore della classe stringa senza argomenti

```
stringa vs[3] = { stringa("ciao"), stringa("mondo"), stringa("...") };
```

```
stringa *ps = new stringa[3]; // NON COMPILA perché non vi è un costruttore senza argomenti
```

```
complesso *vc5 = new complesso[5]; // FUNZIONA
```

## MEMBRI STATICI

ESEMPIO:

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

public:

```
    static int quanti_complessi;
```



permette di aggiungere il membro dato intero e agganciarlo alla classe complesso, non alla singola istanza. Quindi tutte le istanze complesso vedranno questa variabile che sarà la stessa per tutte

=> NEL .cpp:

```
int complesso::quanti_complessi = 0;  
complesso::complesso() {  
    quanti_complessi++;  
}
```

=> NEL MAIN:

```
cout << complesso::quanti_complessi;
```

← può chiamare l'oggetto alla funzione

Si può accedere a quell'istanza solo attraverso l'operatore di visibilità. Questo se la inserisco nella parte public della classe. Qui però qualcuno potrebbe alterarne il conteggio, ed è possibile renderla privata.

=> NELLA CLASSE:

```
class complesso {  
    double re, im;  
    static int quanti_complessi;
```

public:

```
    complesso();
```

```
    static int numero(); // funzione membro statica
```



si inserisce la parola chiave static solo nel file.h.

La funzione può accedere solo ai membri dato statici (non usano il puntatore this).

=> NEL MAIN:

```
cout << c1.numero() << endl;
```

// si invoca con l'operatore selettore di membro

## FUNZIONI CONST

Si tratta di funzioni che non modificano un membro dati della classe

```
class complesso {
```

```
    double r, i, m;
```

```
public:
```

```
    complesso (double r=0, double i=0);
```

```
    double reale () const;
```

```
    double immag (i) const;
```

```
    complesso & scala (double);
```

```
    complesso & scrivi () const; /* ... */;
```

Il compilatore poi verifica che su un oggetto costante siano state chiamate solo funzioni con attributo const.

=> NEL MAIN:

```
const complesso c1 (3.2, 4);
```

```
cout << "Parte reale: " << c1.reale() << endl; //compila
```

```
cout << c1.scaled(2); //non compilerebbe
```

Inoltre il compilatore controlla che la dichiarazione della funzione const sia compatibile col contenuto della funzione.

L'attributo const va messo sia nella dichiarazione sia nell'implementazione, questo perché rientra nella definizione del tipo della funzione membro => il meccanismo di overloading distingue due versioni di una funzione che differiscono solo per l'attributo const.

ESEMPIO:

```
class Vettore {
```

```
    int *vett;
```

```
    int size;
```

```
public:
```

```
    int operator [] (int ind) const;
```

```
    int& operator [] (int ind);
```

```
    /* ... */;
```

=> NEL MAIN:

```
const Vettore v1 (10);
```

```
v1[0] = 4; //ERRORE
```

```
Vettore v2 (5);
```

```
v2[0] = 4; //ok, ritorna un riferimento alla i-esima componente
```

## PREPROCESSORE

È come un programmino eseguibile che il compito del preprocessore è il file sorgente e creare l'unità di compilazione (sequenza di caratteri ASCII che verrà compilata).

Tutte le operazioni che può fare sono controllate dalle DIRETTIVE PER IL PREPROCESSORE (il primo carattere è #).

Queste operazioni possono essere:

- includere nel testo altri file
- espandere i simboli definiti dall'utente secondo le loro definizioni
- includere o escludere parti di codice dal testo che verrà compilato

Per includere i file header si hanno due possibilità:

- #include <file.h>, il file verrebbe cercato nella cartella predefinita dei file di intestazione
- #include "file.h", il file verrebbe cercato nella directory corrente e, in cui verrà messo in esecuzione l'eseguibile oppure in una sotto-directory (#include "subdir/file.h") oppure in un altro punto ben preciso (#include "c:/subdir/file.h")

## MACRO

Simbolo che viene sostituito con una sequenza di elementi lessicali corrispondenti alla sua definizione

```
#define CONST 123
```

```
#define MAX(A,B)((A)>(B)?(A):(B))
```

```
int main(){
```

```
    X=CONST; // X=123
```

```
    Y=MAX(4,(z+2)); // Y=(4>(z+2)?(4):(z+2))
```

```
}
```

## COMPILAZIONE CONDIZIONALE

```
#if (cond) code;
```

```
#elif
```

```
#elif
```

```
#else
```

```
#endif (return 0;)
```

Sono seguite da un'espressione costante che se vera, verrà eseguito solo il pezzo di codice presente in quel ramo.

Una volta finito di controllare le condizioni

→ Serve per evitare che uno stesso file venga incluso più volte in una unità di compilazione



Vengono utilizzate per rappresentare un'area di memoria che in tempi diversi può contenere dati di tipo differente. I membri corrispondono a diverse "interpretazioni" di un'unica area di memoria.

Lo spazio che occupa una unione è determinata dal membro dati più grande: i membri dell'unione condividono la stessa area di memoria o parte di essa.

Sono utilizzate quando si programma a basso livello e si vuole evitare di sprecare memoria.

union {

char c;

int i;

double d;

// in fase di inizializzazione

} a = {'X'}, // è possibile inizializzare solo il primo campo

## FUNZIONI RICORSIVE

È una funzione che nel suo corpo invoca se stessa.

ESEMPIO:

```
int fatt (int n) {
```

```
    if (n == 0) return 1;
```

```
    return n * fatt(n-1);
```

```
}
```

ESEMPIO:

```
void div_and_mod (int q, int beta, int i=0) {
```

```
    if (q == 0)
```

```
        return;
```

```
    div_and_mod (q/beta, beta, i);
```

```
    cat << q%beta;
```

```
}
```

Nella formulazione di una funzione ricorsiva è necessario individuare uno o più casi base, nei quali termina la chiamata ricorsiva.

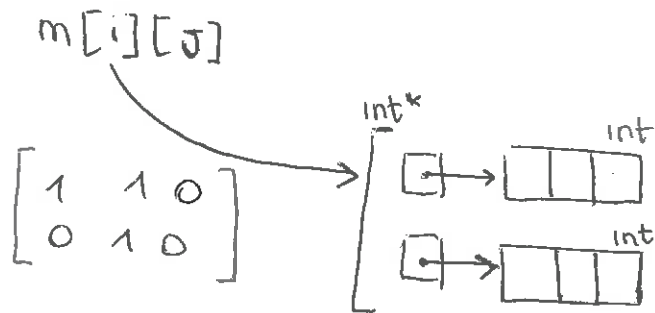
Ogni funzione ricorsiva può essere formulata in modo iterativo, e spesso quest'ultima può essere più conveniente in termini di tempo di esecuzione e occupazione di memoria.



## MATRICE DINAMICA

Devo creare una matrice le cui dimensioni vengono inserite dall'utente:  
matrice dinamica.

```
int righe, colonne;  
int **m = new int*[righe];  
for (int i=0; i<righe; i++)  
    m[i] = new int[colonne];
```



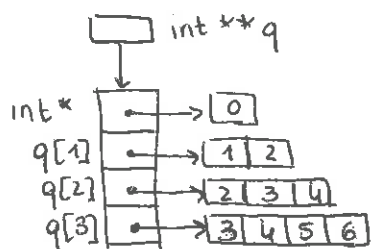
Per deallocare:

```
for (int i=0; i<righe; i++)  
    delete [] m[i];  
delete [] m;
```

## VEETTORE DI VETTORI

```
int **q = new int*[4];  
q[0] = new int[1];  
q[1] = new int[2];  
q[2] = new int[3];  
q[3] = new int[4]; // *q
```

```
for (int i=0; i<4; i++)  
    q[i] = new int[i+1];
```



Va gestito come una matrice:

```
for (int i=0; i<4; i++)  
    for (int j=0; j<i+1; j++)  
        q[i][j] = i+j;
```

Per deallocare:

```
for (int i=0; i<4; i++)  
    delete [] q[i];  
delete [] q;
```

## RIPASSO - OVERLOADING OPERATORI UNARI

Ci sono 3 modi per invocare un operatore unario:

### 1) SOLUZIONE CONE FUNZIONI GLOBALI

```
complesso operator~(complesso c1){  
    complesso ris(c1.reale(), -c1.immag());  
    return ris;  
}
```

in questo caso abbiamo creato  
un nuovo complesso, distinto  
da c1, coniugato.  
Non viene modificato c1.

```
void operator~(complesso &c1){  
    c1.im = -c1.im;  
}
```

NEL MAIN DIVENTEREBBE:  
`~c1;`  
`cout << c1;`  
non potrebbe fare le due cose insieme

```
complesso& operator~(complesso& c1){  
    c1.im = -c1.im;  
    return c1;  
}
```

questa versione modifica c1  
e consente la concatenazione  
con altre chiamate