UNIVERSITÀ DI PISA

Dipartimento di Ingegneria
Corso di Laurea Triennale in Ingegneria Informatica

Appunti

Fondamenti di Programmazione

Professori:
Prof. Cococcioni
Prof. Perazzo

Autore:
Enea Passardi

Anno Accademico 2023/2024

Contents

1 Rappresentazione dei numeri

1.1 Formula della sommatoria

Per poter rappresentare numeri maggiori o uguali alla base β posso usare la rappresentazione posizionale applicando la **formula della sommatoria**:

$$n = \sum_{i=0}^{p-1} \alpha_i \beta^i \tag{1}$$

In questa formula "p" indica il numero di cifre, α indica la cifra alla posizione specifica e β la base elevata alla posizione. Ad esempio, se volessi convertire il numero $(110)_2$ in un numero base 10 applicherei la formula nel seguente modo:

$$n = \sum_{i=0}^{3} \alpha_i 2^i = 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 = 6 \tag{2}$$

1.2 Metodo delle divisioni successive con resto

Il procedimento della divisione è un procedimento utilizzato per convertire da basi diverse da due verso base due, questo procedimento si basa sul fare una serie di divisioni in colonna, dividendo il numero per la base valutando poi il resto:

- Se il resto della divisione è diverso da 0 allora il valore binario sarà 1.
- Se il resto della divisione è uguale a 0 allora il valore binario sarà 0.

1.3 Somma tra numeri binari

Come in ogni base numerica è possibile svolgere operazioni di somma, divisione e prodotto anche nel sistema di numerazione **binario**, quando si somma di numeri binari vi sono quattro casi possibili:

- $(0 + 0)_2$ quest'operazione restituisce 0.
- $(1 + 0)_2$ o $(0 + 1)_2$ operazione che restituisce 1.
- $(1 + 1)_2$ operazione che restituisce zero generando un riporto che viene applicato al bit successivo.

Le operazioni in binario partono dal bit meno significativo verso quello più significativo.

1.3.1 Overflow

Un calcolatore lavora con un numero finito di bit, potrebbe però capitare che la somma di due numeri potrebbe essere maggiore della cifra massima di rappresentazione con n bit, in questo caso si parla di overflow. Per ovviare al problema si potrebbe dire che ci vogliono **n+1** bit per calcolare due numeri su n bit.

Riconoscere un Overflow Un overflow può essere riconosciuto quando sommando due numeri che hanno lo stesso segno, ottengo un risultato con segno diverso.

1.4 Rappresentazione in modulo e segno

Nella rappresentazione in modulo e segno utilizzo il primo bit **(MSB)** come bit di controllo per il segno, se il primo bit è a 0 allora il numero sarà positivo, se il primo bit è a 1 il numero sarà negativo:

$$a = \begin{cases} +(|a|) & a_{p-1} = 0 \\ -(|a|) & a_{p-1} = 1 \end{cases} \tag{3}$$

Questo tipo di rappresentazione può essere anche visto come: $a_{p-1}, a_{p-2}, \ldots, a_0$. Questo sistema presenta però una debolezza, lo zero può essere rappresentato in due modi diversi (1000, 0000). Il fatto di utilizzare l'**MSB** come bit di segno porta ad avere un intervallo di rappresentabilità di $[-(2^{p-1}-1), (2^{p-1}-1)]$. Ovviamente è importante considerare quanti BIT si intende utilizzare per il calcolo, in quanto si rischia di ottenere un risultato incoerente.

1.5 Rappresentazione in Complemento a 2

Nella rappresentazione in CA2 si cerca di ovviare al problema del bit di segno sottraendo al numero A, se negativo, il valore della base elevato al numero di bit su cui si cerca di rappresentare il numero:

$$A = \begin{cases} +A & A \geq 0 \\ (2^p - A) & A < 0 \end{cases} \tag{4}$$

Il complemento a 2 è noto anche con il nome di complemento alla base, infatti, questo stesso procedimento può essere applicato anche per basi diverse da due. L'intervallo di rappresentabilità del CA2 è $[-(2^{p-1}), (2^{p-1}-1)]$. Inoltre il CA2 prevede anche la possibilità di decodificare il valore ottenendo così il numero originale:

$$a = \begin{cases} +a & a \geq 0 \\ (2^p - a) & a < 0 \end{cases} \tag{5}$$

1.6 Rappresentazione in BIAS

Nella rappresentazione con **BIAS** l'idea è quella di prendere un qualsiasi numero reale sommandogli un **BIAS**, il principio di base è che la somma tra il numero **n** e il **BIAS** dovrebbe essere positiva, esso viene calcolato come:

$$n = a + (\beta^{p-1} - 1) \tag{6}$$

seguitamente alla somma si codifica il numero in binario tramite il metodo delle *divisioni successive con resto*. L'intervallo di rappresentabilità di questo meotodo è $[-(2^{p-1}-1), 2^{p-1}]$, questo intervallo specifica che in negativo vista la necessità di un bit a 0 per il segno non posso utilizzare il MSB per identificare il valore, mentre invece nella rappresentazione positiva non ho questo problema.
La decodifica del metodo avviene con il procedimento opposto, dopo aver decodificato il numero procedo a rimuovere il BIAS.

1.7 Rappresentazione in virgola fissa

Viene utilizzato un numero determinato di bit per la parte intera e un numero determinato di bit per la parte frazionaria, assumendo di avere p bit per rappresentare il numero, ne verranno usati f per rappresentare la parte frazionaria e $(p-f)$ per rappresentare la parte intera, la decodifica del numero avviene applicando il seguente metodo:

$$R = a_{p-f-1}, \ldots, a_0, a_{-1}, a_{-f} = \sum_{i=-f}^{p-f-1} a_i \beta^i \tag{7}$$

Quando l'obiettivo è codificare il numero, si rappresentata la parte intera con il metodo della divisione con resto, mentre la parte frazionaria viene calcolata moltiplicando per due il valore della parte *frazionaria*:

- se il prodotto restituisce un numero nella forma 1.n si aggiungerà un uno alla codifica binaria.
- se il prodotto restituisce un numero nella forma 0.n si aggiungerà uno zero alla codifica binaria.

può accadere anche che la parte frazionaria necessiti di un numero infinito di cifre per essere rappresentata (*periodica*), cadendo ogni n iterazioni in un periodo, per ovviare la problema si esegue un troncamento al numero di bit che ci eravamo prefissati di usare per la parte frazionaria.

```
0.87 * 2 = 1.74      Parte intera 1
0.74 * 2 = 1.48      Parte intera 1
0.48 * 2 = 0.96      Parte intera 0
0.96 * 2 = 1.92      Parte intera 1
0.92 * 2 = 1.84      Parte intera 1
0.84 * 2 = 1.68      Parte intera 1
```

Figure 1: Codifica parte frazionaria

1.8 Rappresentazione in virgola mobile

Questa forma di rappresentazione si ispira alla *notazione scientifica* e si basa su una struttura del tipo:

$$r = \pm m \cdot \beta^e \tag{8}$$

dove m identifica la mantissa (*il numero effettivo*), il simbolo β la base del sistema e e l'esponente. Nella rappresentazione binaria per codificare un numero in **virgola mobile** si deve prima codificarlo in virgola fissa, dopo di che portarlo ad una forma del tipo:

$$1.0101 \rightarrow 0.10\ldots \tag{9}$$

contanto di quanti spazi si è dovuta spostare la virgola (questa cifra serve per l'esponente), dopo di che si procede a codificare l'esponente con la rappresentazione in **bias** e infine si scrive tutto nella seguente forma:

$$\{1, 0101010101010, 0111111\} \tag{10}$$

dove la prima cifra rappresenta il segno, seguito dalla mantissa, a sua volta seguita dall'esponente.

In generale, si può definire le componenti della rappresentazione in un modo più formale :

- La parte frazionaria della mantissa viene definita come

$$f = \frac{F}{2^G} \tag{11}$$

  di conseguenza la mantissa m equivale a $m = 1 + f$.

- L'esponente viene rappresentato dal numero naturale E sottratto al bias

$$e = E - (2^{K-1} - 1) \tag{12}$$

L'intervallo di rappresentabilità è quindi $[-2^{bias+2}, 2^{bias+2}$, dove il bias è definito dal bias dell'esponente. Esistono tre principali varianti della rappresentazione in virgola mobile:

- **Half-precision**: la codifica avviene su **16bit**, di cui **1** per il **segno**, **5** per l'**esponente** e **10** per la **mantissa**.
- **Single-precision**: la codifica avviene su **16bit**, di cui **1** per il **segno**, **8** per l'**esponente** e **23** per la **mantissa**.
- **Full-precision**: la codifica avviene su **32bit**

1.9 Regola del cortocircuito

La regola del cortocircuito è una regola riguardante la valutazione di un espressione condizionale, in particolare nella valutazione di un espressione condizionale, se essa è divisa in due parti e la valutazione della prima condizione porta a far risultare falsa l'intera espressione, automaticamente non viene valutato il resto e viene restituito falso.

1.10 Costo di operazione con i bit

In generale quando si hanno operazioni tra due numeri in binario si necessita di al massimo:

- Nel caso di operazioni tra numeri binari semplici:
  - Per rappresentare la somma sono necessari al massimo $n + 1$ bit
  - Per rappresentare il prodotto sono necessari al massimo $2n$ bit
- Nel caso di numeri in CA2:
  - Per rappresentare la somma sono necessari al massimo $n + 1$ bit.
  - Per rappresentare il prodotto sono necessari al massimo $n + m$ bit.
- Nel caso di numeri rappresentati in BIAS:
  - Per rappresentare la somma sono necessari al massimo $n + 1$ bit
  - Per rappresentare il prodotto sono necessari al massimo $2n$ bit

2 Tipi primitivi

2.1 Oggetti

Un **oggetto** è un gruppo di **celle di locazione** *consecutive*, *numerate* e *organizzate* secondo uno schema verticale, utilizzate dal programmatore come se fosse un'unica *cella di locazione*. Una *cella* è costituita da **8 condensatori**, quindi un insieme di **8 bit** oppure **un byte**. Un **oggetto** è caratterizzato da:

1. **Indirizzo** della **prima cella di locazione**, tutte le altre celle che vengono usate avranno lo stesso indirizzo della prima cella e saranno quindi viste come un unico blocco.
2. **Contenuto delle celle**.

Ogni variabile o costante che dichiariamo occupa uno spazio di n celle, è importante considerare che, **non è possibile che due variabili siano contenute nella stessa cella di locazione**.
Durante tutta l'esecuzione del programma l'indirizzo di un oggetto non subisce modifiche, mentre il valore di un oggetto può subire delle modifiche:

- un oggetto prende il nome di variabile se il suo valore cambia durante l'esecuzione del programma.
- un oggetto prende il nome di costante se il suo valore non cambia durante l'esecuzione del programma.

Ogni oggetto viene definito da un tipo, il quale è a sua volta identificato da un insieme di valori che l'oggetto (*dominio*) può assumere e dalle operazioni definite su di esso. In C++ i tipi sono distinti in **tipi primitivi** e **tipi derivati**, i tipi fondamentali servono a rappresentare informazioni semplici, come i numeri interi o i caratteri, mentre i tipi derivati permettono di creare strutture dati complesse a partire da tipi fondamentali.

2.2 Tipo intero

Il tipo int è costituito dall'insieme dei numeri interi rappresentabili su 32 bit, quindi tutti quei numeri che rientrano nell'intervallo $[-231, 231 - 1]$. Le operazioni definibili su int sono quelle comuni che, anche a livello matematico vengono fatte sui numeri interi:

- **somma(+)**.
- **differenza(-)**.
- **prodotto(*)**.
- **quoziente(/)**.
- **divisione con resto** (%)

Oltre ai classici operatori aritmetici su int è anche possibile utilizzare operatori di confronto, il confronto tra due int non restituisce però un int, ma bensì un boolean. Inoltre è possibile distinguere diversi livelli di priorità tra i diversi operatori di confronto, infatti gli operatori di uguaglianza hanno una importanza più alta rispetto agli altri operatori di confronto.

2.2.1 Numeri naturali

Esiste inoltre un sottoinsieme dei numeri interi, i quali rappresentano i numeri naturali che, vengono identificati come unsigned int, il loro range di rappresentabilità varia da $[0, 2^n - 1]$. Questo tipo viene utilizzato principalmente per le operazioni a basso livello sfruttando gli operatori bit a bit:

- Operatore **OR**: |.
- Operatore **AND**: &.

- Operatore **XOR**:^.
- Operatore di **Complemento a 1**: ¬.
- Operatore di **Traslazione a sinistra**: <<.
- Operatore di **Traslazione a destra**: >>.

2.3 Tipo booleano

Il tipo booleano è un tipo di dato utilizzato per la memorizzazione di due singoli valori, **vero** e **falso**, a livello teorico è possibile rappresentare un booleano su un singolo bit, ma essendo la RAM organizzata in celle da 8bit sono costretto ad occuparne una intera. A livello di memoria RAM tutti i bit vengono posti a 0, escluso il **less significant bit**, il quale verrà usato per definire il valore del **bool**. Tendenzialmente quando si stampa il codice tramite il cout ci si trova restituito come risultato 0 e 1, se volessimo far stampare true o false dovremmo usare un comando diverso:

```
bool x = false;
cout << boolalpha  << x << endl;
//Stampa come valore false
```

2.4 Tipo Reale

Il tipo reale può essere definito mediante due dichiarazioni diverse:

- **Dichiarazione come float**: questa dichiarazione richiama l'ultilizzo della rappresentazione numerica in virgola mobile:

  ```
  float a = 4.0e-2
  ```

- **Dichiarazione come dobule**: questa dichiarazione richiama l'ultilizzo della rappresentazione numerica in virgola fissa:

  ```
  double a = 1.23e-4
  ```

la differenza principale sta nella precisione delle due dichiarazioni che, varia a seconda del contesto. Inoltre, è anche possibile definire tutte le operazioni disponibili sugli **interi**.

2.5 Tipo enumerazione

Un tipo di enumerazione è un insieme di valori costanti definiti dal programmatore, ognuna di esse viene identificata da un identificatore e assume il nome di enumeratore. La rappresentazione generica di un enumeratore è la seguente:
enum <identificativo> { <enumeratore>, <enumeratore>, ... , <enumeratore>};
In assenza di una specificazione precisa, il compilatore assegna automaticamente i valori costanti agli identificatori degli enumeratori, questi valori sono progressivi e iniziano da 0. Questi tipi vengono utilizzati per numeri finiti di valori che, generalmente sono non numeriche. Nonostante le enumerazioni coinvolgano solo operazioni di confronto è possibile anche eseguire le stesse operazioni eseguibili su numeri interi.

2.6 Esempio

```
#include <iostream>

using namespace std;
enum asse : char {x = 'x', y = 'y'};

int main () {
    // "frutta" è l'identificativo della enum
     enum frutta { mela, pera, banana, pesca, kiwi };
```

```
    // enum senza identificativo
    enum { mela, pera, banana, pesca, kiwi };
    // Assegnazione esplicita di valori ad ogni enumeratore.
    enum frutta { mela = 0, pera = 1, banana = 2, pesca = 3};
    cout << "Enumeratore x: " << (char) x << endl; // x = 'x'
    char x = 'c';
    cout << "Variabile x: " << x << endl; // x = 'c'
   cout << "Enumeratore asse::x: " << (char) asse::x << endl;// 'x'
   cout << x+1 << endl; //restituisce 'D'
}
```

2.7 Tipo Char

Il tipo *char* identifica un singolo carattere nella forma 'A', 'c', '3'. Esso viene rappresentato mediante una cella di un singolo byte e lla codifica del carattere avviene mediante un codice numerico definito dalla tabella ASCII.

3 Tipi derivati

3.1 Riferimenti

Sui riferimenti sono definite alcune operazioni aritmetiche:

- Sottrazione tra un **indirizzo** e un **intero**.
- Addizione tra un **indirizzo** e un **intero**.
- **Incremento** e **Decremento** postfissi.
- Sottrazione tra **indirizzi**: operazione che restituisce il numero di elementi compresi in un intervallo.

Inoltre è possibile che una variabile abbia un numero indefinito di riferimenti oltre al *riferimento di default*. Un tipo *riferimento* è costituito dai possibili identificatori di oggetti di un dato tipo, un indicatore si dichiara nel seguente modo:

- Un *identificatore* definito come *object-type-indicator* che comprende gli identificatore dei tipi fondamentali.
- Il *reference-initializer*, normalmente richiesto, costituto dal nome di un oggetto (*left-value*).

un riferimento una volta inizializzato non può essere il alcun modo modificato.
In C++ vi è la possibilità di definire riferimenti ad oggetti (presunti) non modificabili, in tal modo l'oggetto verrà considerato come costante:

```
int a = 5;
const int &val = a;
```

3.2 Puntatori

Si definisce come *oggetto puntatore* un oggetto il cui valore destro è **l'indirizzo di memoria di una variabile** il cui tipo primitivo deve essere uguale tipo del *puntatore*. Un tipo puntatore non *punta* necessariamente ad un variabile, esistono infatti **puntatori a funzioni**, **puntatori a oggetti**, ecc.... Un esempio di dichiarazione di puntatore è:

```
int* a;
```

Una volta dichiarato un puntatore è altrettando importante inizializzarlo correttamente, il puntatore infatti, a differenza di una variabile **non ha** come valore destro un valore intero, un carattere o un qualunque altro tipo di carattere, ma ha come **valore destro** un *indirizzo in esadecimale*, ad esempio:

```
int a = 3;
int* c = 0x61fea0;
```

L'operazione di assegnazione ad un **puntatore** richiede che venga utilizzato l'operatore di rifermento (&). Ad esempio:

```
int* a = &b;
```

Tramite gli operatori $*$ e &, i quali sono ambedue **unari** e **prefissi** è possibile compiere delle operazioni su i puntatori, ad esempio:

```
int i, j:
int* p, q;
p = &i; // Inizializzo il puntatore sull'indirizzo di i
cout << *p << endl; // stampa il valore della variabile i
cout << b << endl; // stampa il valore dell'indirizzo del puntatore
cout << b << endl; // stampa l'indirizzo della variabile a cui punta

j = *p // Assegno il valore di i al j
q = p // Assegno il valore dell'indirizzo di al puntatore q
q = &p; // Operazione errata.
```

L'ultima operazione è sbagliata in quanto q è un puntatore che punta ad un intero, assegnare per riferimento il valore di un puntatore (quindi un indirizzo) porta ad un errore, in quanto formalmente sono tipi diversi, un esempio è:

```
int* = int**
```

Per questo stesso motivo non è neanche possibile fare un puntatore che punta a se stesso. Un osservazione interessante da fare è la seguente: l'operatore "&" e l'operatore "*" si annullano a vicenda.

3.2.1 Puntatori costanti

Nella dichiarazione di un puntatore è anche possibile utilizzare l'operatore **const**, esistono tre casi fondamentali in cui si applicano:

- **Puntatore a costante**: è un puntatore di cui è possibile cambiare il riferimento (*variabie a cui punta*), ma di cui non è possibile modificare il valore referenziato.

```
const int* s;
```

- **Puntatore costante**: è un puntatore di cui non è possibile cambiare il riferimento, ma di cui è possibile modificare il valore associato ad esso.

```
int* const s;
```

- **Puntatore costante a costante**: è un puntatore di cui non è possibile cambiare il riferimento e di cui non è possibile cambiare il valore associato al riferimento.

```
const int* const s;
```

3.2.2 Puntatori a puntatori

Vi è anche la possibilità di definire dei puntatori a puntatori, di conseguenza dei puntatori il cui *left-value* è a sua volta il riferimento ad un puntatore (*un indirizzo*), esso viene definito come:

```
int val = 4;
int* a = &val;
int** p = &a; // Questo puntatore punta ad un altro puntatore
```

3.3 Array

Un oggetto *array* è identificato da una *n-tupla* di elemento dello stesso tipo, referenziati mediante un indice rappresentante la posizione degli elementi nell'array. La definizione di un array ha la seguente forma sintattica:

basic-array-object-definition
    object-type-indicator array-specifier
object-type-indicator
    type-indentifier
    pointer-type-indicator
array-specifier
    array-declarator aggregate-initializer
array-declarator
    identifier { *constant-expression* } *array-declarator* { *constant-expression* }

L'identificatore *identifier* individue l'array che viene definito, l'*object-type-indicator* indica il tipo degli elementi e l'espressione costante specifica il numero degli elementi stessi.
Esistono tre tipologie differenti di **array**:

- **vettori**: array *monodimensionali*.

- **matrici**: array *bidimensionali*, per convenzione identificati come $A_{i,j}$ dove i identifica la *i-esima* riga della matrice e j identifica la j-esima riga della matrice. L'accesso ad una matrice, essenod anchèessa alla fine un puntatore può essere fatto tramite l'artimetica dei puntatori, in particolare, assumendo di avere una matrice $r \cdot c$:

  - Per aumentare la colonna è sufficente aggiungere l'incremento j.
  - Per aumentare la riga è sufficente aggiungere l'incremento $i \cdot c$.

  Ad esempio:

  ```
  for(int i = 0; i < r; i++){
      for(int j = 0; j < c; j++){
          cin >> m[i*c + j]
          cin >> *(m + (i*c) + j)
      }
  }
  ```

  pre creare una matrice dinamica si crea una riga di puntatori a vettori nella quale ad ogni cella si fa corrispondere un'altro array dinamico:

  ```
  int** mat;
  mat = new int* [righe];
  for(int i = 0; i < righe; i++){
      mat[i] = new int[cols];
  }
  ```

- **tensori**: array *tridimensionali*.

Un **array** può anche essere definito costante e di conseguenza il suo valore non è più modificabile per tutta la durata dell'esecuzione.
Sugli array non sono definite ne operazioni artimentiche, ne tantomeno operazioni di confronto e inoltre, non possono essere adottati come tipo di ritorno delle funzioni.
Quando un array viene dichiarato esplicitamente non è necessario definire la dimensione:

```
char[] vettore = {'c','i','a','o',\0};
```

Tra le parentesi quadre di un **array** viene racchiusa un'espressione la quale permette di accedere ad un qualsiasi valore contenuto nell'array, a patto che questo valore sia compreso tra 0 e la dimensione - 1.
Nel caso in cui si provi ad acceredere ad una variabile fuori dal range di elementi del vettore, si rischia di accedere ad una cella di memoria di una variaibile casuale, la quale non è legata al vettore in sé, il motivo di ciò deriva dal fatto che per accedere alle varie celle del vettore viene utilizzato un puntatore, il quale punta inizialmente alla prima posizione del vettore:

```
int vett[5] = {1,2,3,4,5}
int* index = &vett[0];

cout << vett[4] << endl;
cout << *(index + 4) << endl;
```

Il nome di un array corrisponde quindi all'indirizzo del primo elemento, valgono quindi le seguenti equivalente:

- x equivale a &[x]0.
- *x equivale a x[0].

3.4 Oggetto struttura

Un ***oggetto struttura*** è una *n-tupla* ordinata di elementi definiti come ***membri*** o ***campi***, ognuno dei quali definito da un ***nome*** e un ***tipo***. Una struttura permette di rappresentare delle informazioni su un dato oggetto. Una struttura è definita da:

- Un indentificatore (*tag*): è il nome del tipo che viene dichiarato (il quale può essere anche anonimo).
- Membri della struttura (*structure-member-section*): una lista di membri eterogenei, ciascuno destinato a contenere un dato.

ad esempio, per definire una struttura è possibile usare il comando:

```
struct persona{
    char nome[20];
    char congome[20];
    int giorno_nascita, mese_nascita, anno_nascita;
};
```

vi è inoltre la possibilità di definire come mebro di una struttura un'altra struttura:

```
struct persona{
    char nome[20];
    char congome[20];
    struct nascita {
        int giorno_nascita, mese_nascita, anno_nascita;
    } n
};
```

In una struttura è possibile definire un mebro come puntatore ad una struttura non ancora definita, a patto che essa sia stata *dichiarata in maniera incompleta*:

```
struct T;
struct F{
    T* pun;
};
struct T{
    int a, b;
}
```

Una struttura può essere definita specificando il *right-value* dei singoli campi:

```
T oggetto1 = {1,3}
T oggetto2(oggetto1);
```

il secondo assegnamento viene definito come *assegnamento membro a membro*, ossia, ogni campo di *oggetto1* viene copiato nel rispettivo campo di *oggetto2*.

Strutture costanti è possibile definire una struttura costante, in questo caso essa va *inizializzata* al momento della dichiarazione e il suo valore non potrà essere modificato per tutta l'esecuzione. Per riferirsi ad una struttura referenziata tramite puntatore si utilizzano due modi:

```
(*pun).att
```

il motivo delle parentesi tonde deriva dalla priorità degli operatori, l'operatore "." ha una priorità maggiore rispetto all'operatore "*", di conseguenza si rende necessario l'ultilizzo delle parentesi tonde le quali elevano la priorità di "*pun" al massimo. Altrimenti è possibile usare anche la seguente notazione:

```
pun->att;
```

l'operatore "->" viene anche definito *selettore di membro* e comportando un *indirezione* richiede che il puntatore non sia nulla. La memoria occupata di una struttura è all'incirca il valore in byte del *sizeof* di ogni membro moltiplicato per il numero di membri, anche se per oggetti che occupano meno di un byte il compilatore riserva comunque 4 byte di spazio.

3.5 c-stringhe

Una premessa importante che si può fare a riguardo del C++ è che non esiste un tipo **stringa**.
Una cstringa consiste in una sequenza di caratteri avente lunghezza arbitraria, cio'e in un qualunque array di caratteri che contiene, ad un certo punto, il carattere di arresto \0. Ricordiamo che posso esprimere un letterale stringa racchiudendo una sequenza di char tra virgolette. Per esempio ho char c[5], che consiste in una cstringa che può contenere una parola avente al più 4 caratteri, tenendo conto della presenza del carattere di arresto. All'interno di una cstringa posso avere altri caratteri speciali come quello di ritorno a capo \n o di tabulazione \t.
Una *cstringa* può essere inizializzata in due modi:

- come un qualunque array: *char[] array1 = {'c', 'a', 'l', 'c', 'o', 'l', 'o', \0};*.
- ricorrendo a una stringa **letterale**: *char[] array2 = "calcolo"*;

Nella prima inizializzazione è necessario anche indicare il carattere di arresto, nella seconda esso viene incluso automaticamente.
Negli array non è definita l'operazione di assegnamento. L'uso di una stringa letterale è permesso soltanto solo nell'inizializzazione: non potremo porre infatti *array1 = array2*. Il **terminatore di stringa** ha codifica ASCII impostata a tutti 0 (0x00). Le operazioni di ingresso e di uscita accettano variabili stringa come argomento, ad esempio:

```
char[12] string;
cin >> string;
cout << string;
```

in particolare:

Operatore di ingresso per una stringa legge caratteri dallo stream di ingresso (*saltando eventuali spazi bianchi*) e si ferma dopo aver incontrato uno spazio bianco. Questo carattere, che non viene letto, porta al termine dell'operazione di lettura e la memorizzazione nella variabile stringa, con il carattere di terminazione posto sul fondo: tutto questo avviene, ovviamente, avendo una stringa di adeguate dimensioni.

Operatore di uscita l'operatore di uscita invece, scrive i caratteri della stringa sullo stream di uscita, non includendo il carattere di arresto.

3.6 Libreria *¡cstring¿*

Risulta necessario, non potendo utilizzare operazioni di assegnamento e/o operatori di confronto, introdurre delle funzioni aggiuntive per manipolare le nostre cstringhe. Le funzioni in questione si trovano nella libreria ¡ cstring ¿ e consistono nelle seguenti:

- **strlen**: Restituisce un intero, rappresentante il numero di caratteri della stringa (*escludendo il terminatore*).
- **strcpy**: Operazione con scopo analogo all'assegnamento, essa permette di copiare il valore di una *cstring source* in una *cstring destination*.
- **strcmp**: Operazione analoga al confronto, essa permette di confrontare due *cstringhe* restituendo un risultato che può assumere tre valori:
  - se viene restituito 0 le due *cstringhe* hanno lo stesso valore.
  - se viene restituito un numero positivo significa che il primo carattere diverso, da sinistra verso destra, ha un valore più grande nella prima cstringa rispetto alla seconda (Es: se *str1 = "Ciao"* ed *str2 = "Addio"* allora *strcmp(str1, str2) > 0*)
  - se viene restituito un numero negativo significa che il primo carattere diverso, da sinistra verso destra, ha un valore pi'u piccolo nella prima cstringa rispetto alla seconda (**Es**: se *str1 ="Addio"* ed *str2 ="Ciao"* allora *strcmp(str1, str2) < 0*).

se ho due *cstringhe* aventi lunghezza diversa, ma con corrispondenza tra tutti i caratteri, otterro -1 se il **terminatore di arresto** verrà incontrato nella prima stringa, mentre otterrò 1 nel caso opposto.

- **tolower** sfuttando operazioni di addizioni e sottrazioni trasforma ogni carattere in minuscolo.
- **toupper** tramite operazioni di addizione e sottrazione trasforma ogni carattere in maiuscolo.
- **strcat**: concatena due cstringhe. Concatenare, come gi'a evidenziato introducendo i letterali stringa, consiste unire due letterali stringa in un unico literal. Il valore di questo 'e l'unione dei valori dei due literal divisi. Mediante la funzione di libreria strlen individuo la lunghezza della cstringa. Ricordare che l'strlen non considera, nella somma, il carattere di arresto.

4 Memoria dinamica

4.1 Allocazione nell'heap

Talvolta in CPP potrebbe essere necessario allocare oggetti durante l'esecuzione del programma, in quanto non si conosce a priori il numero di essi, a questo scopo è stata introdotta la **memoria dinamica** la quale è definibile con un insieme di celle che posso essere allocate e deallocate.
In C++ per allocare un oggetto in questa memoria, chiamata generalmente **memoria heap** si utilizza l'operatore **new** associando un **puntatore** all'oggetto creato:

```
int* pun = new int;
```

a questo punto all'interno dell'heap viene controllata la prima posizione libera mediante un insieme di celle il cui unico scopo è memorizzare lo stato delle altre, una volta trovato viene modificato il valore di essa, la cella rispettiva viene occupata e viene restituito il valore della posizione della cella.
Un oggetto allocato dinamicamente può anche avere un valore iniziale racchiuso tra parentesi tonde:

```
int ex = 4*1;
int* pun = new int(ex);
```

Durante la compilazione viene anche calcolato il numero di byte necessari per l'instanziamento dell'oggetto calcolando il numero di oggetti che intendo stanziare per il loro **sizeof**, se lo spazio necessario è maggiore dello spazio disponibile, l'operatore **new** fallisce e viene restituito un errore.
Un errore comune che viene fatto è quello di rimuovere il riferimento o di instanziare un oggetto nell'heap senza che esso sia referenziato, in tal caso l'oggetto occuperà per tutta l'esecuzione del programma quelle celle senza poterlo **deallocare**.

4.2 Deallocazione dall'heap

Talvolta durante l'esecuzione di un programma potrebbe essere necessario dover deallocare un oggetto allocato nell'heap, ciò viene reso possibile mediante il comando **delete**:

```
int* vett = new int[10];
delete[] vett;
```

Quando avviene la deallocazione l'operatore delete entra nell'heap, nelle celle adibite a gestire quali celle della memoria dinamica sono disponibili o occupate, cerca le celle associate al vettore e le rende disponibili, ma in ogni caso **non cambia il loro valore**, semplicemente, appena avviene una nuova allocazione esse posso essere sovrascritte.

Dereferenziare il puntatore Quando avviene una deallocazione tramite operatore **delete** il vettore puntato viene lasciato inalterato, di conseguenza una *best-practice* che un programmatore potrebbe attuare è quella di deferenziare il puntatore:

```
int* pun = new int[10];
delete[] pun;
pun = nullptr;
```

4.3 Matrici dinamiche

Una matrice dinamica è una particolare matrice rappresentata come un vettore di vettori, in origine si crea un vettore di n dimensioni e ad ogni *n-esima* posizione si associa un ulteriore vettore dinamico di m dimensioni, per inizializzare una matrice di questo tipo si procede come:

```
int** mat = new int*[5];
for(int i = 0; i < 5; ++i){
    mat[i] = new int[5];
}
```

facendo in questo modo è anche possibile referenziare i singoli elementi della matrice con la seguente notazione:

```
mat[i][j]
```

andando a prelevare l'elemento alla *i-esima* riga e alla *j-esima* colonna.
Grazie a questo sistema di referenziazione è possibile anche prendere gli elementi in *input* nel seguente modo:

```
for(int i = 0; i < 5; i++){
    for (int j = 0; j < 5; ++j) {
        cin>>mat[i][j];
    }
}
```

4.4 Matrici linearizzate

Si definisce come matrice lienarizzata una matrice caratterizzata da una rappresentazione "simil-vettoriale", rappresentazione che garantisce alcuni vantaggi rispetto alla canonica forma di *vettore-di-vettori*, in primis una matrice linearizzata consente un accesso sequenziale, ciò rende più veloci le operazioni più semplici, in secondo luogo anche la rappresentazione rende più comodo l'utilizzo di algoritmi lineari e di ordinamento.

Le matrici linearizzate hanno una dimensione di n*m, per poter scorrere questa tipologia di matrice è necessario utilizzare due cicli for, cicli che permettono di scorrere nel seguente modo

```
const unsigned int N = 5;
...

int* a = new int[N*N];
for(int i = 0; i < N; ++i){
    for(int j = 0; j < N, ++j){
        a[(i*N) + j] = 1;
    }
}
```

difatti la scrittura *a[i*n + j]* è assolutamente equivalente alla scrittura che si avrebbe facendo *a[i][j]*. Uno dei vantaggi principale rappresentato dalle matrici linearizzate lo si ha in termini di spazio occupato:

- in una **matrice dinamica** si ha che lo spazio occupato totale equivale a *M * sizeof(vector) + M * N * sizeof(elemento della matrice).*
- in una **matrice linearizzata dinamica** si ha che lo spazio occupato totale equivale a *M*N*sizeof(elemento della matrice).*

Matrici bidimensionali e matrici linearizzate hanno rappresentazioni dei dati diverse, in particolare se una funzione richiede uno specifico dei due formati è necessario che il parametro attuale rispetti la rappresentazione richiesta per evitare di incorrere in errori causati da una mala lettura dei dati.

4.5 Liste

Una lista può essere vista come un'insieme di elemento omogenei dello stesso tipo, ogni membro della lista è rappresentato da una struct contenente al suo interno le informazioni sull'oggetto (con il *rispettivo tipo*) e un puntatore all'elemento successivo, segue l'esempio:

```
struct element{
    int inf;
    element* nextElement;
}
```

Il primo elemento della lista viene indirizzato dal puntatore della lista, mentre l'ultimo membro viene ha il puntatore nullo. Un esempio di come funzione una lista è il seguente:

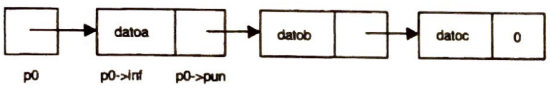

Figure 2: Organizzazione di una lista

L'inseriemento di n elementi in testa alla lista viene fatta nel seguente modo:

```
elem *p0 = 0;
elem *lista;
for(int i = 0; i < 10; i++){
    lista = new elem;
    cin >> lista->inf;
    lista->next = p0;
    p0 = lista;
}
```

Inizialmente ho un puntatore denominato lista, il quale identifica l'elemento che sto creando, mentre p0 è un puntatore che identifica l'elemento a cui devo far puntare l'elemento successivo che creo, in particolare, ad ogni iterazione creo un nuovo elemento, inserisco il valore che esso deve assumere e lo faccio puntare all'elemento che ho creato precedentemente, successivamente aggiorno la variabile. Se invece vi è la necessità di inserire degli elementi in coda si deve procede in un altro modo:

```
void inserisciSulFondo(elem* p0, int a){
    elem *p; elem* q;
    for(p = p0; q != 0; q = q->pun){
        p = q;
    }
    q = new elem;
    cin >> q->inf;
    q->next = 0;
    if(p0 == 0) p0 = q; else p->pun = q;
}
```

Per inserire un elemento in coda ad una lista è necessario prima di tutto scorrere all'ultimo elemento di essa utilizzando due puntatori, i quali puntano a due elementi consecutivi della lista, in partcolare, a fine scorrimento, q assume il valore 0 (in quanto ultimo elemento) e p punta all'ultimo elemento, se invece la lista è vuota essa viene fatta puntare al nuovo elemento.

4.5.1 Stampa della lista

Per stampare una lista è necessario tramite un vettore scorrere gli elementi fino a che il puntatore è diverso da 0:

```
for(q = p0; q->pun != 0; q = q->pun){
    cout << q->inf;
}
```

4.5.2 Estrazione da una lista

Estrazione dal fondo

```
bool estraiFondo(elem& p0, int& a){
    elem* p = 0;
    elem* q;
    if(p0 == 0){
```

```
        return false;
    }
    for(q = p0; q->pun != 0; q = q->pun)
        a = q->inf;
    if(q == p0) p0 = 0; else p->pun = 0;
    delete q;
    return true;
}
```

Estrazione dalla testa

```
bool estraiTesta(elem* p0, int& a){
    elem* p = p0;
    if (p0 == 0) return false;
    a = p0->inf;
    p0 = p0->pun;
    delete p;
    return true;
}
```

si procede andando a controllare se la lista è vuota tramite l'**if**, se essa non è vuota si procede a estrarre il valore assegnandolo al riferimento, dopodiché viene cambiato il puntatore della lista all'elemento successivo.

4.5.3 Deallocazione di una lista

Per deallocare una lista di procede nel seguente modo:

```
void dealloca(elem*&p0){
    elem *p = p0, *q;
    while (p != 0){
        q = p ->pun;
        delete p;
        p = q;
    }
}
```

Si procede passando a q il valore puntato da p, di conseguente l'elemento successivo a p, dopo di che si procede ad eleminare p e infine si assegna p a.

4.5.4 Ricerca di un elemento

5 Operatori, separatori e Espressioni

5.1 Operatori e separatori

La combinazione di alcuni caratteri speciali o le loro combinazioni possono essere utilizzate come fossero operatori, i quali permettono di definire determinate operazioni nel calcolo di espressioni. Un operatore è caratterizzato da:

- **Posizione**: dove si trova rispetto ai suoi operandi:
  1. ***Prefisso***: se si trova prima degli operandi
  2. ***Infisso***: se si trova nel mezzo ai due operandi.
  3. ***Postfisso***: se si trova dopo gli operandi.
- **Arietà**: la quale identifica il numero di operandi.
- **Precedenza**: la priorità dell'operazione nel flusso di esecuzione, la priorità di un operazione viene identificata da un numero intero.
- **Associatività**: l'ordine in cui vengono eseguiti gli operatori con la stessa priorità, ad esempio, gli operatori con associatività a sinistra verranno eseguiti da sinistra verso destra, mentre invece gli operatori con associatività da sinistra verso destra verranno invece eseguiti da destra verso sinistra.

Un separatore è invece un carattere il quale ha come scopo principale quello di dividere le operazioni all'interno di un'espressione, ad esempio, le parentesi tonde sono un separatore.

5.2 Istruzioni condizionali

L'istruzione condizionale **if** ha la seguente forma sintattica:

if(condition) *statement;* ***else*** *statement*;

Se la *condition* è vera viene eseguito il codice contenuto nella porzione dell'if (*parte then*), altrimenti, nel caso in cui fosse falso viene eseguita la porzione di codice contenuta nell'else (*parte else*).

5.3 Switch e break

Un'ulteriore istruzione condizionale è rappresentata dallo *switch*, il quale ha la seguente forma sintattica:

basic-switch-statement
switch **(condition) switch-body**
switch-body:
alternative-sequence

Ciascuna delle alternative (*alternative*) è costituita da una sequenza di istruzione preceduta da un etichetta, chiamata *case-label*:

switch **(condition) switch-body**
    case-label-seq statement-seq
case-label
    case *costant-expression*:
    default:

Le etichette *case-label* indicano le diverse alternative presenti nel corpo dell'istruzione switch (*switch-body*), inoltre, le *costant-expression* che compaiono in ogni *case* devono avere lo stesso

tipo della condizione e devono avere valori **distinti**.
L'esecuzione di uno switch si articola in alcune fasi:

1. viene valutata l'espressione che costituisce la condizione.
2. viene eseguita la porzione di codice (*statement-seq*) associata all'etichetta in cui compare il valore calcolato.

Se in nessuna etichetta compare tale valore, se e solo se presente, viene eseguta la porzione di codice sotto l'etichetta di **default**.

5.4 Ciclo iterativo While

Il ciclo while è sintatticamente costituito come:

basic-while-expression
    while *(condition) statement*

L'esecuzione del ciclo while avviene tramite la valutazione della *condition*, la quale dopo essere stata valutata restituisce un valore **booleano**:

- se la *condition* ritorna come valore **true** si procede con l'esecuzione dello statement.
- se la *condition* restituisce come valore **false** si ignora il ciclo.

Il while rientra in tutte quelle operazioni definite *ripetitive*, di conseguenza fintanto che l'istruzione non diventa falsa lo *statement* viene ciclicamente eseguito.

5.4.1 Istruzione ripetitiva do-while

Una variazione che può essere fatta al ciclo while riguarda l'introduzione dell'operatore *do*, la differenza principale sta nell'ordine di esecuzione del codice, in quanto:

- viene prima eseguita la porzione di *statement*.
- successivamente viene valutata la *condition*:
  - se **true** si procede a riesegure nuovamente lo *statement*.
  - se **false** si esce dal ciclo.

5.5 Ciclo iterativo For

Il for viene scritto con la seguente forma sintattica:

basic-for-statement
    for *(initialization condition-specification step)* statement
initialization
    definition-statement
    expression-statement **condition-specification**
    expression
step
    expression

L'inizializzazione (*initialization*) termina sempre con il ";" e viene generalmente usata per inizializzare una variabile di **controllo**. La specifica della condizione (*condition-specification*) serve invece a definire l'espressione di controllo, il cui risultato è un valore logico. Lo *step* infine è un'espressione che assegna un nuovo valore alla variabile di controllo (generalmente *incremento* o *decremento*).
Il for viene eseguito con un ordine specifico, per prima cosa viene inizializzata la **variabile di controllo**, quest'operazione è *non ripetibile* all'interno di uno stesso ciclo, successivamente:

1. viene verificata la *condition-specification*:
    - se il valore restituito è **true** viene eseguito lo *statement*.
    - se il valore restituito è falso viene terminato il ciclo.
2. assumendo che la *condition-specification* sia *true*, viene eseguito lo *statement* e infine viene eseguito lo *step*, il quale porta alla modifica della **variabile di controllo**.

5.6 Istruzioni di salto

Le istruzioni di salto permettono di controllare parti di programma e in base la verificarsi possono attuare due operazioni diverse:

- ripetizione di un istruzione strutturata.
- la scelta tra alcune alternative.

Le principali istruzioni di salto (*jump-statement*) sono:

- **break**.
- **continue**.
- **goto**.
- **return**.

5.6.1 istruzione break

L'istruzione *break* può essere utilizzata all'interno del corpo di un ciclo o di un istruzione *switch*, essa permette di eseguire un salto all'istruzione successiva al corpo stesso.

5.7 Operatori logici

Gli operatori *binari* presenti in CPP sono 3:

5.7.1 Operatore OR

L'operatore di or logico (||) è un operatore il quale restituisce vero se una delle due variabili o entrambe sono vere. L'OR è una operazione **binaria** e **infissa**

5.7.2 Operatore AND

L'operatore AND logico (&&) è un operatore (infisso) che restituisce vero solo se entrambe la variabili sono vere. L'AND è una operazione **binaria** e **infissa**

5.7.3 Operatore Implicazione

L'implicazione logica ($\implies$) è un'operatore che restituisce vero quando la **premessa è falsa** oppure quando **la premessa** è vera e la conseguenza è anch'essa vera.

5.8 Operatore sizeof

L'operatore **sizeof** è un operatore unario e prefisso, esso viene applicato ad un oggetto qualsiasi:

```
sizeof(oggetto);
```

Il compito di quest'operatore è quello di restituire la dimensione di un oggetto misurata in **byte**. L'operatore *sizeof* viene valutato a tempo di compilazione e restituisce un valore intero (*size_t*).

5.8.1 Esempio

```
int a;
cout << sizeof(a) << endl; //4 byte
```

5.9 Espressioni di assegnamento

Un'espressione che viene utilizzata comunemente nella programmazione C++ è quella di *assegnamento*, l'utilizzo di questa espressione è quella di calcolare il valore di un espressione e di sostituirla al *right-value* di una variabile. La sintassi di un'espressione di assegnamento è la seguente:

basic-assignment-expression
variable-name = expression
variable-name
identifier

A sinistra dell'espressione di assegnamento compare invece il *right-value* il quale identifica l'indirizzo dell'oggetto di cui voglio modificare il valore, talvolta però l'indirizzo di un oggetto può anche essere usato come un **right-value**. Inoltre un assegnamento essendo un espressione restiuisce come risultato l'*indirizzo della variabile a sinistra del simbolo* di assegnamento, un espressione può quindi contenere anche più di un assegnamento. Un esempio di espressione di assegnamento è:

```
int a = 4;
int b = a;
int c = 4;
int b = a = c + 1
```

5.9.1 Espressioni di incremento e decremento

Il C++ prevede alcune abbreviazioni per alcune espressioni di assegnamento, note come espressioni di *incremento* e *decremento*, in quanto invece di usare un'operatore di assegnamento è possibile utilizzare una simbologia più compatta:

```
i++;
++i;
i--;
--i;
```

Gli operatori di *incremento* e *decremento* sono unari, ma possono essere sia **prefissi** che **postfissi**, vi è però una differenza tra i due:

- **Incremento prefisso**: la variabile viene modificata prima che venga utilizzata da un'altro operatore.

  ```
  int a = 4;
  int b;
  b = ++a;
  ```

  a questo punto il valore di a sarà uguale a 5, come anche il valore di b. L'operazione restituisce un *left-value*.

- **Incremento postfisso**: la variabile a cui è applicato viene modificata dopo che il suo valore è stato eventualmente utilizzato da altri operatori:

  ```
  int a = 4;
  int b;
  b = a++;
  ```

  in questo caso il valore di a sarà 5, mentre il valore di b sarà 4. L'operazione restituisce un *right-value*.

5.10 Operazioni artimetiche e logiche

Quando si valuta un'espressione, sia che essa sia arimetica (*restituisce un risultato aritmetico*) o booleana (*restituisce un espressione booleana*) è fondamentale osservare le precedenze degli operatori:

1. per prima cosa vengono valutati i fattori derivati tramite le varie funzioni.
2. Vengono valutati gli operatori di decremento e incremento insieme a tutti gli operatori unari.
3. Vengono valutati i termini applicando gli operatori binari, secondo l'ordine:
   (a) Operatori **moltiplicativi**.
   (b) Operatori **additivi**.
   (c) Operatori di **traslazione**.
   (d) Operatori di **relazione**.
   (e) Operatori di **uguaglianza**.
   (f) Operatori **bit a bit**
   (g) Operatori **logici**.
   (h) Operatori di **assegnamento**.

   Utilizzando intorno ad un'espressione le parentesi tonde essa diventa automaticamente un fattore e viene quindi valutata con la *massima priorità*.

5.11 Conversione implicita

5.12 Conversione esplicita

6 Funzioni

6.1 Dichiarazione di funzioni

la dichiarazione di una funzione comprende divers parti, divisibili in due **macrocategorie**:

- **Intestazione** (*function-header*): essa specifica alcuni dati fondamentali sulla funzioni:
  - Tipi di ritorno della funzione.
  - Nome della funzione.
  - Argomenti su cui opera la funzione (*argomenti formali*)

  L'instestazione ha la seguente sintassi: *return-type identifier (argument-part)*
- **Corpo della funzione** (*function-body*): il quale specifica le operazioni da eseguire per ottenere il risultato della funzione.

è importante notare che la *argument-part* è **opzionale**. Esiste inoltre un caso di funzioni il cui *return-type* è definito come **void**, queste funzioni hanno la particolarità di non generare alcun risultato, inoltre, la stessa *keyword*, se inserita nell'*argument-part*, può indicare anche che la funzione non ha parametri. La differenza tra una funzione **void** e **non void** è la seguente:

- una funzione **non void** termina quando all'interno del *function-body*(*compound-statement*) viene ritornato il risultato della funzione tramite la *keyword* **return**, se essa non è presente la funzione ritornerà un errore.
- una funzione **void** termina quando si giunge al termine delle *function-body*, non ammettono risultato e di conseguenza non ammettono nenanche la *keyword* **return**.

6.1.1 Esempio di dichiarazione di funzione

```
int somma(int a, int b){
    return a+b;
}
```

Una dichiarazione di funzione è costuita da un'intestazione nella quale viene data una dichiarazione ad argomenti *formali*, argomenti generici che assumono valore quando la funzione viene *richiamata*. Inoltre vi è la possibilità di definire una funzione senza specificare il nome dei parametri:

```
int somma(int, int);
```

6.2 Chiamata ad una funzione

Una volta definita una funzione è possibile eseguirla, per farlo è necessario attuare una *chiamata di funzione*, ossia un espressione composta dal nome della funzione e la lista di *argomenti attuali*, la struttura è definita come:

basic-function-call-expression
identifier (*expression-list*)

Ciascuna *expression-list* costituisce un argomento attuale della funzione e rappresenta il dato su cui la funzione opera in quella chiamata, è importante seguire delle regole per mantenere una corretta corrispondenza tra argomenti attuali e formali:

- il numero degli argomenti *formali* deve essere uguale a quello degli argomenti *attuali*.
- il tipo degli argomenti *formali* deve essere uguale a quello degli argomenti *attuali*.
- ogni argomento *attuale* deve occupare la posizione occupata dall'argomento *formale* che si intende far corrispondere.

Una chiamata di funzione è un espressione il cui valore può essere un **fattore** di un'altra espressione.

6.3 Passaggio per valore e per riferimento

Quando si attua una *chiamata alla funzione* ogni **argomento attuale** identifica un'espressione che produce un valore dello stesso tipo dell'argomento *formale* rispettivo. Quando un argomento formale subisce una modifica per effetto di una qualunque espressione viene cambiato il valore del contenitore, senza però avere alcuna ripercussione sulla variabile associata a quell'argomento nella chiamata, ciò deriva dal tipo di passaggio, il quale viene detto **passaggio per valore**. In questo tipo di passaggio si viene a creare una variabile ***locale*** temporanea in memoria e di conseguenza ogni modifica sull'argomento viene attuata solo dentro quella cella, la quale viene poi *deallocata* al termine della funzione.

```
int somma(int a, int b);
```

Vi è anche un'ulteriore possibilità, quella di definire come parametro formale della funzione un *riferimento*, questo metodo permette di attuare delle modifiche anche sulla variabile che viene passata nella chiamata alla funzione, il motivo deriva dal fatto che in questo tipo di passaggio vi è il passaggio di un indirizzo di una *cella di memoria* contenente quella variabile.

```
int somma(int &a, int &b);
```

6.4 Prototipazione

La prototipazione è un meccanismo implementato in molti linguaggi che permette di dichiarare le funzioni prima del metodo main.

```
int somma(int a, int b);
int minEMax(int a, int b, int &min);
int main(){

}
int somma(int a, int b){
    ...
}
int minEMax(int a, int b, int &min){
    ...
}
```

6.5 Overloading di funzioni

In molti linguaggi di programmazione vi è la possibilità di inizializzare funzioni con lo stesso nome:

```
void function (...);
void function (...);
```

Affinché questa tecnica funzioni si deve differenziare in qualche modo le n funzioni. La differenza che si deve rilevare quanto si applica l'**overloading** è quella presente nella *signature* (firma) delle funzioni:

```
void function (int a, int b);
void function (float a, float b);
void function (float a, char b);
void function (char b, float a);
void function (const float vett[], float a);
void function (float vett[], float a);
```

6.6 Array come parametro di funzioni

Il passaggio di un array come parametro di una funzione è un operazione diversa rispetto al passaggio di una qualsiasi variabile, quando si passa un vettore non stiamo passando i valori, ma stiamo passando il riferimento a quel vettore, o nello specifico, il riferimento alla prima posizione del vettore:

```
int somma(int vett[], int dim);
int somma(int* vett, int dim);
int somma(int &vett[0], int dim);
```

Vi è anche la possibilità di passare come parametro un *vettore costante*, in questo caso il riferimento all'oggetto sarà **non modificabile** non ammettendo alcuna modifica del parametro formale. Inoltre, nel caso in cui passi un puntatore posso anche decidere di farlo passare come se fosse un riferimento, di conseguenza ogni modifica che verrà fatta sul mio puntatore formale verrà rispecchiata anche sul puntatore *attuale*:

```
int somma(int*& vett, int dim)
```

7 Algoritmi di ordinamento e ricerca

7.1 Introduzione

Il problema dell'ordinamento è uno dei problemi più classci dell'informatica, esso riguarda l'ordinamento di un vettore non ordinato:

```
int vett[5] = {1,3,5,2,6}
```

Utilizzando alcuni **algoritmi di ordinamento** è possibile portare il vettore ad essere ordinato in maniera crescente o decrescente, i due algoritmi più noti per l'ordinamento di un vettore sono due:

- **Selection sort**.
- **Bouble sort**.

Un'altro problema tipico della programmazione è quello della ricerca, il quale riguarda la ricerca degli elementi in un vettore dfati la posizione inferiore e quela supoeriore,

7.2 Selection Sort

Il **selection sort** è un *algoritmo di ordinamento* utilizzato per minimizzare il numero degli scambi. Assumo di avere un **vettore** iniziale di **n** elementi, lo scorro cercando l'elemento minimo, una volta trovato lo pongo alla prima posizione libera, successivamente "blocco quella posizione" e ripeto l'operazione fintanto che il vettore non è ordinato. Questo algoritmo esegue $n-1$ iterazioni. I passi sono i seguenti:

1. cerco il valore minimo nel vettore.
2. confronto il minimo con l'elemento successivo per tutti gli elementi el vettore;
   - se il *minimo* è minore del *valore confrontato*, semplicemente non faccio nulla.
   - se il *minimo* è minore del *valore confrontato*, scambio i due valori.
3. metto il valore minimo alla prima cella libera nel vettore e **blocco** quella cella.
4. ripeto l'operazione $n-1$ volte.

Il **selection sort** è un ottimo algoritmo per vettori di piccole dimensioni, in quanto esegue il minor numero di *scambi* e di *iterazioni*, le quali equivalgono a $n-1$ volte, d'altro canto però il numero di confronti equivale a:

$$\sum_{i=0}^{n-1}(n-i) = \frac{n(n-1)}{2} \tag{13}$$

Rendendo l'algoritmo inefficace per vettori di grandi dimensioni.

7.2.1 Implementazione C++

Il selection sort può essere facilmente implementato in linguaggio C++ come:

```
int main(){
    const unsigned int dim = 5;
    int vett[5] = {4,3,6,1,-1}
    for(int i = 0; i< dim-1; ++i){
        int min = i;
        for(int j = i; j < dim; ++j{
            if(min>vett[j]){
                int tmp = min;
                min = vett[j];
                vett[j] = tmp;
            }
        }
    }
}
```

7.3 Bubble sort

Il bubblesort è un algoritmo di ordinamento basato su un confronto progressivo di coppie, l'obiettivo di questo algoritmo è quello di portare ad ogni iterazione il numero minore nella prima posizione disponibile.
Ad ogni iterazione vengono controllate le coppie di elementi consecutivi:

- Se il numero a sinistra è maggiore di quello a destra si procede scambiando i due valori.
- Se il numero a sinistra è maggiore di quello a destra non si procede con alcuno scambio.

Alla fine di ogni iterazione avremo l'elemento minore messo alla prima posizione libera del vettore e dopo $n-1$ iterazioni generali abbiamo il vettore ordinato. Il **numero di scambi** dipende da come è configurato il vettore, generalmente però, la complessità computazionale è On^2, mentre, il numero di confronti è sempre definito come:

$$\sum_{i=0}^{n-1}(n-i) = \frac{n(n-1)}{2} \tag{14}$$

Assumiamo di avere un vettore non ordinato qualsiasi, applicando il **bubble sort** può anche verificarsi il caso in cui il vettore sia ordinato prima ancora del termine di tutte le iterazioni; Per ottimizzarlo è sufficente implementare un booleano che viene messo vero ad ogni iterazione e successivamente viene messo falso al primo scambio.

7.3.1 Implementazione

```
for(int i = 0; i < dim; ++i){
    for(int j = i; j < dim; ++j){
        if(vett[j] < vett[j+1]){
            int temp=vett[j];
            vett[j]=vett[i+1];
            vett[j+1]=temp;
        }
    }
}
```

7.4 Ricerca lineare

Ricerca in tutto il vettore. complessità lineare O(n) questa ricerca esegue $n-1$

7.5 Ricerca binaria

Nella ricerca binaria vi è la necessita che il vettore sia **ordinato in ordine crescente**, si confronta l'elemento cercato con quello nella posizione centrale:

- Se l'elemento cercato è minore dell'elemento in poisizione centrale, la ricerca prosegue nella prima metà del vettore.
- Se l'elemento cercato è maggiore dell'elemento in poisizione centrale, la ricerca prosegue nella seconda metà del vettore

La ricerca prosegue fintanto che non rimango con due elementi possibili, a questo punto confronto entrambi e se non trovo nulla l'elemento non è presente nel vettore, altrimenti ritorno la posizione a cui si trova. Il numero di confronti nella ricerca binaria non definibili, ma la complessità è O(log(n))

7.5.1 Implementazione iterativa

```
while (inf <= sup){
    int medio = (inf+sup)/2;
    if(k > ordVett[medio]){
        infe = medio+1
    } else{
        if(k>ordVett[medio]){
            sup = medio-1;
        } else {
            pos = medio;
            return true;
        }
    }
 }
```

7.5.2 Implementazione ricorsiva

8 Programmazione effettiva

Generalmente quando si sta coficiando un programma *semplice* ci si limita a usare poche funzioni, contenute nello stesso file che si scambiano informazioni attraverso metodi e funzioni, questo sitema però si rivela indatto per problemi più complessi.
Il c++ per ovviare a questo problema permette di suddividere il programma il blocchi, definendo regole di collamento tra i file e di visibilità, in modo che

8.1 Regole di visibilità

Si definisce come *campo di visibilità* o **scope** di un identificatore la parte di programma all'interno della quale l'identificatore è *visibile*. Ogni campo di visibilità viene definito da **regole di visibilità**, alcuni esempi di questi campi sono:

- Blocchi di codice,
- Spazi dei nomi.
- Unità di compilazione.
- Spazi dei nomi.
- Classi.

Un identificatore dichiarato all'interno di ha una *visibilità di blocco* (*visibilità **locale***)

8.2 Blocchi

un blocco di codice è costituito da una sequenza di istruzioni racchiusa tra parentesi graffe.
Un identificatore dichiarato in un blocco è visibile dal **punto in cui è stato dichiarato** fino alla **fine del rispettivo blocco**, includendo anche blocchi annidati.

```
int main(){
    // Blocco A
    int a;
    {
        // blocco b
        int j;
    }
    // j non visibile
}
```

Inoltre, un identificatore può essere dichiarato nel blocco più interno facendoli riferire a entità distinte, nonostante però sia una pratica non ottimale per la comprensbilità del codice. Inoltre, anche gli identificatori dichiarati all'interno del corpo di cicli iterativi, di istruzioni condizionali e di funzioni hanno una visibilità di blocco. Un'**etichetta** dichiarata all'interno di un blocco è visibile in tutto il corpo della funzione a cui appartiene.

8.3 Unità di compilazione

Una unità di compilazione è costituita da un file sorgente e dai file inclusi mediante le direttive:

```
#include...
```

Se il file da includere non è di libreria e si trova nella stessa cartella del file sorgente il suo nome va racchiuso tra virgolette.
Un identificatore dichiarato a livello di **file** è visibile dalla sua dichiarazione fino alla fine dell'unità di compilazione, inoltre, anche gli identificatori delle funzioni sono dichiarati a livello di file, in questo modo esse sono chiamabili da funzioni dichiarate antecedentemente.
Se a livello di blocco ho un identificatore uguale a un identificatore definito a livello di file, all'interno del blocco avrà priorità la dichiarazione locale, nel caso in cui voglia però usare la dichiarazione a livello di **file** si deve utilizzare l'operatore "::".

8.4 Spazio dei nomi

Uno *spazio dei nomi* o **namespace** è costitutio da un insieme di dichiarazioni e definizioni racchiuse tra parentesi graffe, ognuna delle quale specifica delle entità definite come **membri**, è possibile che un membro di uno spazio dei nomi sia a sua volta un'altro spazio dei nomi:

```
namespace nome{
    struct st {int a; double b;}
    int n;
    void ff(int a){}
}
```

Gli identificatore relativi ad uno spazio dei nomi sono visibili dall'inizio dalla loro dichiarazione all'interno dello spazio dei nomi fino alla fine di esso, ciò permette di utilizzare una stessa dichiarazione all'interno di diversi spazi di nomi.

```
namespace uno{
    struct st1 {int a; duouble b;};
}
namespace due{
    struct st2{int b; double c}
}
int main(){
    uno::st1 ss1;
}
```

Per specicare che venga usato uno specifico spazio dei nomi come *default* si può usare la direttiva **using namespace**:

```
namespace aaa{
    int n;
}
namespace bbb{
    int b;
}
using namespace aaa;
int main(){
    n = 10 // riferimento a n definito in aaa
    bbb: n = 100; // riferimento a n definito in bbb
}
```

Uno spazio dei nomi è *aperto*, ciò implica che sia possibile utilizzare più volte lo stesso identificatore in dichiarazioni successive. C++ mette a disposizione uno *spaziio dei nomi globale* del quale fanno parte le dichiarazioni e le definizioni fatte a livello di file.

8.5 Collegamento

Un programma può essere costituito da più unità di compilazione, mentre un file è costituto da una sola unità di compilazione, queste unità vengono sviluppate e *tradotte* separatamente per poi venir collegate per formare un file *eseguibile*.
Consideriamo ora la definizione di un oggetto o di una funzione, si dice che una unità di compilazione ha un indentificatore con *collegamento interno* se e solo se esso si riferisce ad un entità accessibile solo da quell'unità di compilazione, si dice invece che un identificatore ha collegamento esterno se si riferisce ad una entità accessibile anche da altre unità di compilazione (enità **globali**). Per default gli identificatori costanti hanno collegamento interno, pertanto vanno dichiarati in ogni unità di compilazione. Per fare in modo che un indentificatore abbia collegamento esterno è sufficente dichiararlo con la chiave **extern**, come nell'esempio:

```
extern int a;
```

d'altra parte, se si vuole che esso abbia collegamento interno si deve usare la parola chiave **static**

8.6 Classe di memorizzazione

La classe di *memorizzazione* di un oggetto è una proprietà che riguarda la durata della vita di un oggetto all'interno del sistema, per default gli oggetti dichiarati all'interno di un blocco hanno una *classe automatica* e di conseguenza vengono instanziati alla loro dichiarazione e distrutti alla fine del blocco, alcuni oggetti dotati di questa caratteristica sono:

- Argomenti formali delle funzioni.
- Variabili dichiarate in blocchi.

Un oggetto *statico* gode invece di proprietà opposte, esso viene creato all'inzio del programma e distrutto alla sua fine, il valore inziale dell'oggetto se diversamente specificato vale 0. Un oggetto statico dichiarato all'interno di un blocco nonostante abbia le proprietà di un oggetto statico ha comunque visibilità limitata al blocco in cui è stato definito.

8.7 Moduli

Un modulo identifica una parte di programma che svolge **una** determinata funzionalità, quando si decide di adottare una programmazione *modulare* si cerca di far interagire tra di loro i vari moduli, ogni interazione prevede due entità:

- Cliente(*client*): entità che richiede la funzionalità offerta da un particolare modulo.
- Server: entità che produce un risultato da restituire al client.

Generalemente un modulo è costituito da due file aventi lo stesso nome:

- Un file di intestazione (.h) che fornisce le dichiarazioni delle risorse di interfaccia.
- Un file di realizzazione (.cpp) contenente le implementazioni delle risorse di interfaccia

8.8 Compilazione di un programma

Dopo aver scritto un programma, generalmente, esso viene compilato ed eseguito, la fase di compilazione però non è un unica fase che passato il programma porta alla restituzione di un output, è un processo che passa da diverse fasi:

- **Preprocessing**: prima della conversione del programma in codice macchina avviene una fase nota come preprocessing, durante questa fase il preprocessore esegue delle istruzioni, le quali prendono il nome di: *direttive del preprocessore.*

  Queste istruzioni effettuano manipolazioni sul codice, includendo file esterni e sostituendo stringhe all'interno del testo del programma.

- **Compilazione**: il codice del programma viene convertito il linguaggio macchina, durante questa fase vi è il controllo di **sintassi** e di **semantica**, ma non viene in alcun modo controllato il collegamento tra le funzioni, quindi è possibile definire prototipi di funzioni senza realmente implementarle.

- **Linking**: il linking è un processo che ha come scopo principale quello di collegare il codice di funzioni mancanti (le quali potrebbero risiedere in file esterni) al codice del programma producendo così quello che viene chiamato *immagine eseguibile del programma.*

- **Caricamento**: prima di essere eseguito un programma deve essere caricato in memoria, a questo scopo vi è il **loader**, il quale prende l'immagine eseguibile e la trasporta in memoria.

- **Esecuzione**: sotto il controllo della CPU viene eseguito il programma un istruzione per volta.

8.9 Il preprocessore

Il preprocessore è la parte di programma che elabora il testo del programma prima che avvenga l'analisi lessicale e sintattica, esso può:

- Includere nel testo altri file.
- Espandere simboli definiti dall'utente secondo le loro definizioni.
- Includere o Escludere parti di codice dal testo che verrà compilato.

Tutte queste operazioni sono controllate dalle sopracitate direttive del preprocessore, il cui primo carattere è .

Uno dei compiti del preprocessore riguarda l'inclusione del file header, la quale può essere fatta in due modi diversi:

- Percorso a partire da cartelle standard: *include ¡file.h¿*
- Percorso a partire dalla cartella in cui avviene la compilazione: *include "file.h"*

Oltre a ciò il preprocessore si occupa anche della gestione delle costanti definite tramite la keyword *define*

```
#define dim 10
```

l'utilizzo di questa keyword è però generalmente sconsigliato, in quanto essa non ha alcuna tipizzazione, generalmente infatti si preferisce procedere nel seguente modo

```
const unsigned int dim = 10;
```

quando però si decide di utilizzare il define il preprocessore registra il *simbolo* all'interno di una tabella e sostituisce a tutti i rispettivi assegnamenti il valore della costante

```
#define dim 10

int main(){
    int a = dim;
}
```

quando la direttiva viene elaborata dal preprocessore si ottiene una struttura del tipo

```
int main(){
    int a = 10;
}
```

Il preprocessore permette anche di definire delle *macro*, definibili come delle espressioni costanti che restituiscono un risultato

```
#define MAX(A,B) ((A > B) ? (A) : (B))

int main(){
    int c = 9 + MAX(1, 2) // 11
}
```

quando si generano delle macro è fondamentale l'uso delle parentesi. Inoltre, il preprocessore può elaborare delle **compilazioni condizionali**

```
#if (expression)
// code
#elif (expression)
// code
#else (expression)
// code
#endif
```

I valori delle espressioni costanti vengono interpretati come valori logici ed in base a essi vengono compilati o no i frammenti testo (text) seguenti

9 Programmazione orientata agli oggetti

La programmazione a moduli permette quindi di creare una maggiore organizzazione all'interno di un programma. Un problema che però persiste anche utilizzanddo questa forma di programmazione il problema dell'astrazione, cercare quindi di rappresentare modelli estratti dalla realtà sotto forma di codice, a questo scopo sono state introdotte inizialmente le struct e successivamente anche le classi e consequenzialmente la *Programmazione Orientata agli Oggetti* (**OOP**).

9.1 Tipo classe

Un tipo classe è un tipo di dato derivato costituito da due parti fondamentali:

- Una **parte privata** dove vengono dichiarate le variabili e i metodi di struttura, metori ai quali l'utente non deve poter accede e che rimangono quindi accessibili soltanto dall'interno della classe stessa.
- Una **parte pubblica** dove vengono dichiarati i metodi di *interfaccia*, metodi con i quali gli utenti possono interagire.

Questa divisione del codice permette di rispettare il principio noto come **information hiding**, secondo il quale un utente che utilizza il sistema che stiamo progettando non debba poterlo vedere nella sua interezza, ma debba poter vedere solo le componenti destinate all'interfacciamento, attuando ciò si ottengono due vantaggi principali:

- Semplificare le dipendenze tra i moduli.
- Poter modificare un modulo senza influenzarne il funzionamento per il cliente.

la dichiarazione di una classe è, per alcuni aspetti, simile a quella di una struct, la differenza principale risiede nella possibilità di inserire i membri della classe all'interno di una sezione **privata** specificata dall'*access-indicator* **private**, o altresì, di inserire all'interno di una sezione pubblica specificata attraverso l'*access-indicator* **public**:

```
class Classe{
    private:
        // Parte privata
    public:
        // Parte pubblica
}
```

Una classe è caratterizzata dal suo nome e da i membri, i quali si dividono in:

- **Attributi**: il quali specificano le caratteristiche riguardo alla classe, essi posono essere:
  - Un tipo primitivo, come ad esempio un intero.
  - Un tipo derivato, come ad esempio una struct o un'altro oggetto.
  - Un puntatore.
- **Metodi**: i quali indentificano invece le funzionalità della quale la nostra classe è dotata.

Un oggetto appartenente ad una classe prende il nome di *oggetto classe* o di *istanza della classe*. Un oggetto classe può essere inizializzato con un altro oggetto della stessa classe attuando una ricopiatura membro a membro del campo di dati, inoltre, su un *oggetto classe* sono definite tutte le operazioni della **classe stessa** e le operazioni *predefinite*, in particolare, per invocare una funzione appartenente alla classe stessa si usa una sintassi composta da due parti:

- *class-object*: il quale rappresenta un oggetto classe.
- *class-object-pointer*: il quale rappresenta un puntatore a l'oggetto classe.

per poter attuare un metodo sulla classe stessa è necessario quindi avere un puntatore che punti all'indirizzo della classe stessa, a questo scopo è staoto introdotto il puntatore **this**, il quale consente quindi di riferisi all'oggetto stesso mediante il suo indirizzo in memoria.

9.2 Visibilità di classe

La visibilità di classe definisce che:

- Gli identificatori dichiarati all'interno di una classe sono visibili dal punto della loro dichiarazione fino alla fine della classe stessa.
- Nel corpo delle funzioni membro sono visibili tutti gli identificatori presenti nella classe (anche quelli non ancora dichiarati).
- Se nella classe viene riutilizzato 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 di risoluzione di visibilità :: applicato al nome della classe:
  - Le funzioni membro, quando vengono definite.
  - Un tipo o un enumeratore, se dichiarati nella parte pubblica della classe.
  - Membri statici.
- L'operatore di visibilità non può essere utilizzato per i campi dati non statici

9.3 Costruttori e Distruttori

Un *oggetto classe* può essere creato e inzializzato mediante una funzione, nota come *metodo costruttore*, definita come una funzione membro il cui nome coincide con il metodo della classe:

```
class Nome{
    private:
        // Parte privata
    public:
        Nome (char[] nome){
            // Metodo costruttore
        }
}
```

il costruttore viene chiamato automaticamente, dal compilatore, quando viene creato un oggetto della classe stessa. All'interno di una classe è possibile attuare un **overloading** del costruttore, in questo modo è possibile definire i tre differenti principali tipi di costruttori:

- Costruttore con **parametri**.
- Costruttore di **copia**: un costruttore di copia è un metodo che agisce tra due oggetti della stessa classe effettuando una **ricopiatura** membro a membro dei campi. Esso trova utilizzo in alcuni casi:
  - quando un oggetto viene inizializzato con un altro oggetto appartenente alla stessa classe.
  - quando un oggetto viene passato a una funzione per valore.
  - quando un oggetto restituisce come valore un oggetto *classe* della stessa classe.
  - nelle conversioni implicite tra tipo classe e tipo primitivo.
- Costruttore di **default**: Questo costruttore ha come compito quello di creare e di allocare oggetti senza specificare argomenti, andando a inizializzare l'oggetto inizializzando gli attributi con valori di **default**

Un costruttore richiede l'allocazione di **memoria libera** per l'allocazione dei membri della classe stessa.
I costruttori definiti per una classe possono anche essere chiamati implicitamente anche quando si utilizza l'operatore **new**, come nell'esempio:

```
int main(){
    Nome nome("Enea");
    Nome* nome = new Nome("Enea");
}
```

Distruttore Se si dichiara una variaibile automatica essa genererà nella memoria statica un puntatore con il nome dell'oggetto e nella memoria dinamica verranno allocati gli attributi della classe stessa, quando però l'esecuzione raggiunge la fine il puntatore viene eliminato dalla memoria statica, ma l'oggetto in memoria dinamica rimarrà anche se irraggiungibile, è conveniente quindi implementare in una classe un metodo in grado di deallocare l'oggetto anche dalla memoria libera, questo oggetto prende il nome di distruttore. Per definire un distruttore è necessario prendere il nome della classe e anteporgli l'operatore " ":

```
nome::~nome(){
    delete[] this;
}
```

9.3.1 Regole per distruttori e costruttori

Per i costruttori valgono le seguenti regole:

- Per gli oggetti statici, all'inizio del programma.
- Per oggetti automatici, quando viene incontrata la definzione.
- Per oggetti dinamici, quando viene incontrato l'operatore **new**.
- Per gli oggetti membri di altri oggetti, quando questi ultimi vengono costruiti.

Per i distruttori valgono le seguenti regole:

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

9.4 Funzioni friend

Una funzione definita come **friend** è una funzione esterna ad una classe alla quale viene concessa la facoltà di accedere a attributi pubblici e privati della classe di cui è amica, questa costruzione è di fondamnetale importanza nella ridefinizione di operatori.

```
class Nome{
    private:
        char[] nome;
    public:
        void stampaNome(){
            // Funzione stampa nome
        }
        friend Nome operator+ (Nome& nome2);
}
```

9.5 Membri statici di una classe

Talvolta capita che una classe debba contenere delle informazioni globali, definite come informazioni relative alla classe nel suo complesso e non riferite alla sua istanza singola, questi membri particolari prendono il nome di **membri statici**.
Un campo dati statico esiste come singola copia in memoria, rispetto ad una variabile un campo statico ha come vantaggio quello di avere un controllo di protezione determinato dal fatto di esse parte di una classe. Il fatto che un membro statico sia visibile o meno dall'interno di una classe è dettato dal tipo di operatore di visibilità a cui viene associato:

```
class Nome {
    private:
        // parte privata
    public:
        static void stampanome(Nome n1);
}
```

Una funzione definita come statica non ha alcun **riferimento implicito** ad un oggetto, di conseguenza non può accedere implicitamente a nessun membro statico e né usare il puntatore this.

9.6 Funzioni const

Una funzione membro può essere dichiarata in modo che non possa modificare l'oggetto a cui è applicata, ciò viene reso possibile mediante la parola chiave **const** inserita seguitamente alla lista dei parametri della funzione:

```
\begin{minted}{cpp}
class Nome {
    private:
        // parte privata
    public:
        static void stampanome(Nome n1) const;
}
```

Se un oggetto classe è costante è possibile applicare ad esso soltanto funzioni membro **costanti**. Il meccanismo di **overloading** oltre a distinguere le funzioni in base ai vari argomenti formali attua una distinzione in base alla proprietà della funzione di essere costante o meno

9.7 Espressioni letterali e conversione mediante costruttore

Per i tipi fondamentali del linguaggio sono definite le **espressioni letterali** le quali denotano costanti di questi tipi. Ad esempio, 3 è un'espressione letterale per un intero, 'a' è un'espressione letterale per un carattere. Non è però possibile definire delle vere e proprie espressioni letterali per i tipi definiti dall'utente.
Un costruttore che può essere chiamato con un solo argomento viene usato per convertire valori del tipo dell'argomento nel tipo della classe a cui appartiene il costruttore stesso, ad esempio, assumiamo di avere una classe Complesso, se io attuassi un operazione del tipo:

```
Complesso x, y(0.1, 0.2);
x = y + 2.5;
```

al verificarsi di un operazione di questo tipo il compilatore riconosce automaticamente il 2.5 come un numero complesso la cui parte reale equivale a 2.5 e la parte immaginaria 0.
Notare che questo genere di conversione avviene mediante una chiamata al costruttore, consequenzialmente, se questa operazione risulta costosta conviene definire versioni sovrapposte che trattano esplicitamente il caso specifico. L'utilizzo di un costruttore per effettuare operazioni di conversione tipo non consente di trasformare il tipo classe in un tipo diverso da una classe.

Operatori di conversione Per convertire il valore di un oggetto classe in un valore di un'altro tipo si può definire un *operatore di conversione* identificato dal nome del tipo, ad esempio:

```
class Complesso{
    double re, im;
    public:
    operator double(){return re;}
}
```

9.8 Allocazione e Deallocazione

Per una classe è possibile ridefinire gli operatori di **new** e **delete**: la ridefinizione si effettua quando la versione prefefinita non è abbastanza efficente e conviene quindi definirne un'altra. L'operatore **new** ridefinito dall'utente deve contenere alcune informazione:

- Dimensione di un **oggetto**, rappresentato da un tipo *intero*.

e deve restituire un **puntatore a void**. Gli operatori **new** e **delete** sono *implicitamente* membri statici e non riferiscono alcun oggetto
Quando l'operatore new viene usato per allocare un oggetto della classe in cui esso è definito viene passato come valore **sizeof(oggetto)**.
Vi è anche la possibilità di ridefinire anche altri argomenti, i quali generalmente specificano l'area di memoria nella qualle allocare gli oggetti.
Infine, per ridefinire l'operatore **delete** si deve avere un argomento di tipo puntatore a void il quale rappresenta l'oggetto da deallocare.

9.9 Costanti e riferimenti nelle classi

In una classe un campo dati può essere definito come *costante*, esso mantiene quindi lo stesso valore per tutti gli oggetti della classe stessa. Il campo dati deve essere quindi obbbligatoriamente **statico**.

Alternativamente, in una classe un campo dati può avere l'attributo **const:** in questo caso il campo deve essere inizializzato nel momento in cui viene definito un oggetto appartenente alla classe stessa, ottenendo una costante per ogni oggetto classe.

L'inizializzazione avviene mediante costruttore, ed è divisa in due parti:

- La prima parte è costituita dalld *lista di inizializzazione*, in cui vengono specificati, nell'ordine, i valori dei campi dati con l'attributo **const**.
- La seconda parte in cui vengono assegnati i valori iniziali ad altri campi.

si consideri il seguente esempio:

```
class Ora {
    const int hh, const int mm, const int ss;
    // ...
    public:
        ora(int h, int m, int s);
        //...
}
Ora::Ora(int h, int m, int s)
: hh(h), mm(m), ss(s) // lista di inizializzazione
{
    // Corpo costruttore
}
```

In generale, in un costruttore la lista di inizializzazione può mancare, ma deve essere presente se esistono campi costanti. Inoltre, il corpo può non comprendere istruzioni, ma restano comunque obbligatorie le due parentesi graffe. Le due fasi vengono eseguite il ordine, in modo che all'avvio del corpo del costruttore le costanti siano già state inizializzate.
La lista di inizializzazione può essere usata anche per riferimenti.

9.10 Membri classe all'interno di classi

In una *classe principale* è possibile che siano presenti membri di tipo classe diversa dalla classe principale

```
class record {
    stringa nome, cognome;
}
```

Quando viene dichiarato un oggetto appartenente alla classe principale:

1. Vengono richiamati i costruttori delle classi secondarie, se definiti, nell'ordine in cui queste compaiono nella dichiarazione dei membri della classe.

2. viene eseguito il corpo del costruttore della classe principale.

Quando, invece, viene distrutto un oggetto della classe principale:

1. Viene eseguito il corpo del distruttore della classe principale, se definito.

2. Vengono richiamati i distruttori delle classi secondarie, se definiti, nell'ordine inverso in cui queste compaiono nella dichiarazione dei membri della classe principale.

Se alcune classi secondarie possiedono costruttori con argomenti formali, anche per la classe principale deve essere definito un costruttore e questo deve prevedere una lista di inizializzazione.

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

}
```

se invece le classi secondarie prevedono dei costruttori di default, questi vengono richiamati. Se tutte le classi secondarie hanno costruttori di default, anche quello della classe principale può essere un costruttore di default o mancare.

9.11 Array di oggetti classe

Un array può avere come elementi oggetti classe: se nella classe sono definiti costruttor ee distruttore essi verranno richiamati per ogni elemento dell'array.

```
class complesso {
    double re, im;
    public:
        complesso(double r=0, double c=0);
}
complesso::complesso(double r=0, double c=0){re = r; im = c;}

int main(){
    complesso vc[3];
    for(int i = 0; i < 3; ++i){
        vc[i].scrivi(); cout << endl;
    }
    return 0;
}
```

mentre invece se non vi è la presenza di un costruttore di default bisogna procede con una inizializzazione esplicita degli attributi. Se la lista di membri del vettore non è completa, quindi vi sono elementi senza dichiarazione esplicita, nella classe deve essere mantenuto un costruttore di default, il quale viene richiamato per gli elementi non esplicitamente inizializzati. Inoltre, se l'array di oggetti è allocato in memoria dinamica, se nella classe è definito un costruttore, esso deve essere necessariamente di **default**, in quanto non sono possibili inizializzazioni esplicite.

9.12 Pila e coda

Si definiscono come Pile e code delle strutture dati, dotate di alcune caratteristiche:

- Tipi di dati omogenei.
- Dimensione definita a tempo di compilazione.

La differenza tra queste due strutture è la politica di gestione dei dati:

- Una pila uitlizza una politica **LIFO**(*Last In First Out*).
  Una pila è anche dotata di un indice chiamato top il quale indentifica la posizione del primo elemento della struttura, in particolare, se la lista è vuota l'indice top avrà valore -1, mentre invece in ogni altro caso questo indice identificherà l'ultima posizione in cui un elemento è stato inserito.

  ```
  class Pila{
      private:
          int stack[dim]
          int last = -1;
  }
  ```

  una lista inoltre possiede dei metodi, i quali servono per tre operazioni principali:

  – Inserimento, chiamata comunemente **push**.
  – Estrazione, chiamata comunemente pop.
  – Stampa.
  – Controllo del contenuto.

- Una coda invece segue una politica chiamata **FIFO** (*First In First Out*), inoltre, come anche nelle pile vi è l'uso di un indice che idenfica l'ultima posizione occupata.

9.12.1 Metodi di una Pila

```
    bool pop(Pila& pila, int& val) {
    if(isEmpty(pila)) return false;
    for(int i = 0; i != pila.top; ++i){
        if(pila.stack[i] == val){
            val = pila.stack[(pila.top)--];
            return true;
        }
    }
    return false;
}

bool push(Pila& pila, int val){
    if(isFulll(pila)) return false;
    pila.stack[(++pila.top)] = val;
    return true;
}
```

```
bool isEmpty(Pila& pila){
    if (pila.top == -1) return false;
    return true;
}


bool isFulll(Pila& pila){
    if (pila.top == dim) return false;
    return true;
}

void stampa(Pila& pila){
    for (int i = 0; i != pila.top ; ++i) cout << pila.stack[i] << endl;
}
```

10 Overloading degli operatori

Con l'implementazione della programmazione orientata agli oggetti si è creata la necessità di poter definire i propri operatori in modo da renderli maggiormente adatti in base alle necessità della classe che si sta implementando.

10.1 Overloading

A questo scopo nel linguaggio C++ è stata introdotta la possibilità di **sovrascrivere** un operatore modificandone il funzionamente e rendendolo quindi adatto al modello che si sta cercando di implementare. La ridefinizione di un operatore avviene mediante la chiave **operator** seguita dall'operatore che si sta cercando di ridefinire, a livello di codice ciò comporta che ogni chiamata all'operatore rispetto alla classe conta come un invocazione alla funzione.

```
operator+();
```

In particolare, se l'operatore è binario vi sono due possibilità implementative differenti:

- Definire un **metodo globale** (**static**) e definire esplicitamente come parametri le due istanze su cui si intende compiere l'operazione.
- Definire un **metodo sull'oggetto** e definire esplicitamente solo una delle istanze.

la possibilità di ridefinire gli operatori presenta comunque delle limitazioni:

- Si può ridefinire solo operatori già definiti di default nel linguaggio.
- Non è possibile ridefinire posizione, numero di operandi e sintattica dell'operatore.
- Una nuova definizione dell'operatore deve avere tra gli argomenti almeno un argomento di un tipo definito dall'utente
- Nel caso in cui un operatore sia riferito ad una particolare classe esso deve contenere come primo argomento un oggetto della classe stessa.

Quando si presenta un espressione da analizzare il compilatore tiene automaticamente conto delle possibili conversioni fra tipi fondamentali o delle conversioni implicite definite dal programmatore che coinvolgono i tipi classe.
Quando sono possibili più conversioni di tipo la scelta avviene in base alle regole di priorità definite dal linguaggio, nonostante ciò quando si lavora con tipi definiti dall'utente le equivalenze tra espressioni non valgono automaticamente.

Simmetria degli operatori Quando viene ridefinito un operatore binario il primo operando è necessariamente un oggetto appartenente alla classe stessa, d'altra parte, sul secondo operando non vi è alcuna limitazione specifica, a patto che sia stata definita l'operazione rispetto a quel tipo.

Operatore di cui non è possibile la ridefinizione In C++ esistono alcuni operatori di cui non è possibile effettuare una ridefinizione:

- Operatore di risoluzione di visibilità (::).
- L'operatore di selezione membro (.).
- L'operatore di selezione membro tramite valore di un puntatore(.*)

Gli operatori di indicizzazione in un vettore ([]), di selezione membro di un puntatore (-¿) e di assegnamento devono essere definiti come membri classe.

10.2 Operatore di assegnamento

L'operatore di assegnamento, se non viene in alcun modo ridefinito si occupa di attuare una ricopiatura membro a membro dei campi, quest'operazione risulta utile quando si sta operando con classi che non accedono, o non usano la memoria libera.

Quando invece ci si trova a dover operare tra classi che sfruttano la memoria libera si ha la necessità di ridefinire l'operatore per evitare appunto che avvenga la netta ricopiatura dei membri, operazione che potrebbe risultare in un segmentation fault.

10.3 Incremento pre/post-fisso

L'overloading degli operatori di incremento prefisso (++) e postfisso (++) in C++ consente agli sviluppatori di definire comportamenti personalizzati per queste operazioni quando vengono utilizzate con oggetti di una classe personalizzata: 1. **Incremento prefisso (`++variabile`):**

- La funzione membro per l'overloading ha la firma `Tipo& operator++()`.
- Restituisce una referenza al proprio oggetto dopo l'incremento.
- L'operatore è definito all'interno della classe.

Listing 1: Esempio di overloading dell'incremento prefisso

```
class MiaClasse {
public:
    MiaClasse& operator++() {
        // Logica di incremento personalizzata
        valore++;
        return *this;  // Restituisce una referenza a se stessa
    }

private:
    int valore;
};
```

2. **Incremento postfisso (`variabile++`):**

- La funzione membro per l'overloading ha la firma `Tipo operator++(int)`.
- Usa un parametro `int` fittizio per distinguere tra l'incremento prefisso e postfisso.
- Restituisce una copia del proprio oggetto prima dell'incremento.

Listing 2: Esempio di overloading dell'incremento postfisso

```
class MiaClasse {
public:
    MiaClasse operator++(int) {
        MiaClasse copia(*this);  // Crea una copia del proprio oggetto
        // Logica di incremento personalizzata
        valore++;
        return copia;  // Restituisce la copia prima dell'incremento
    }

private:
    int valore;
};
```

Esempio di utilizzo:

Listing 3: Utilizzo degli operatori di incremento

```
MiaClasse oggetto;
++oggetto;   // Chiamata all'operatore di incremento prefisso
oggetto++;   // Chiamata all'operatore di incremento postfisso
```

10.4 Operatore di lettura e di scrittura

Il meccanismo di **overloading** permette di ridefinire gli operatori di lettura e di scrittura per i tipi definiti dall'utente, in particolare, ciò è possibile creando una funzione friend la quale formatta le operazioni:

```
#include <istream.h>
#include <ostream.h>

class Complesso{...}
friend ostream& operator<<(ostream& ostream, Complesso& c1);
friend istream& operator>>(istream& istream, Complesso& c1);
```

per ridefinire gli operatori è necessario creare delle istanze della classe ostream, in questo modo esse verrano inserite all'interno dello stream di uscita.

11 Gestione dei dati

11.1 Lo stream

Un programma in C++ comunica con i dispositivi esterni tramite dei flussi o stream, i quali sono delle astrazioni, ogni operazione di input e output viene infatti gestita da istanze delle rispettive classi di ingresso e uscita dei dati. Ogni stream può essere utilizzato per compiere operazioni diverse, in C++ esse sono: operazioni di lettura (input) e operazioni di scrittura (output). Il C++ mette a disposizione tre stream predefiniti:

- **cin**: stream standard per l'**ingresso** dei dati.
- **cout**: stream standard per l'**uscita** dei dati.
- **cerr**: stream standard per la **gestione** degli errori.

Nel linguaggio C++ ogni stream corrisponde a un'istanza di una classe la cui parte pubblica contiene tutte le operazioni disponibili per leggere e scrivere dati, mentre la La "parte privata" della classe stream contiene dati e strutture necessari per implementare le operazioni di I/O.

Le operazioni più comuni che si effettuano su questi stream sono operazioni di lettura e scrittura, le quali a loro volta si dividono in:

- **Operazioni formattate**: Le operazioni formattate interpretano il contenuto delle caselle di uno stream come codifiche di caratteri:
  - ***Operazioni di lettura***: esse prelevano dallo stream sequenze dic caratteri e le convertono in rappresentazioni interne di dati.
  - ***Operazioni di scrittura***: esse convertono rappresentazioni interne di dati in codifiche di caratteri per poi trasferirle sullo stream.
- **Operazioni non-formattate**: le operazioni di lettura e scrittura non formattate non interpretano il contenuto delle caselle di uno stream e non effettuano alcuna trasformazione su di esse, l'unica cosa di cui si occupano è il trasferimento di sequenze di byte che transitano per la memoria di uno stream.

Uno stream viene gestito ad accesso sequenziale se una data operazione riguarda le caselle successive a quelle riguardanti l'operazione precedente. Uno stream viene gestito ad accesso casuale se può essere stabilita arbitrariamente la posizione della prima casella coinvolta.

11.2 Stream di input

Lo stream di ingresso **cin** è un'istanza predefinita della classe **std::istream**. Questo *stream* consente la lettura di dati dalla tastiera del terminale. Inizialmente, il puntatore di lettura (**get pointer**) è posizionato sulla prima casella disponibile. Quando viene effettuata l'operazione di lettura da **std::cin**, lo *stream* preleva una sequenza di caratteri dalla tastiera, interpretando la sequenza in base alla sintassi associata al tipo di dato della variabile specificata. Questa sequenza viene quindi convertita in un valore del tipo corrispondente e assegnata alla variabile. L'uso di manipolatori e specificatori di formato può influenzare il modo in cui l'input viene interpretato. Un esempio classico è quello dell'operazione:

```
cin >> x
```

dove, x è una variabile qualsiasi; Quando questa operazione viene eseguita lo *stream* preleva il valore della cella e lo assegna alla variabile spostando il puntatore alla cella successiva. Inoltre, essendo il cin un operazione formattata l'operatore è ottenuto mediante **overloading** dell'operatore di shif a sinistra (mantenendone associatività e precedenza).
In generale, gli operatori di lettura hanno come operando sinistro il riferimento ad uno stream e restituiscono tale elemento in modo da poter comporre le operazioni di lettura e di scrittura della classe stessa. Esistono anche delle altre funzioni di lettura oltre al cin:

- get(): funzione che preleva un carattere e lo restituisce converito in intero.

- get(c): preleva un carattere e lo assegna alla variaibile c.
- get(buf, dim, delim): legge caratteri dallo stram di ingresso e li trasferisce in but aggiungendo il carattere nullo finale fintanto che nel buffer non vi sono dim caratteri o non si incontra il carattere delim (che non viene letto).
- read(s,n): legge n byte e li memorizza a partire dall'indirizzo contenuto in s.

11.3 Controlli sugli stream

Ad ogni stream è associato uno stato, il quale viene rappresentato mediante un campo dati privato della classe **ios**, un *valore dello stato* è, tendenzialmente, costituito da una configurazione di bit di **stato**,in particolare, nel caso in cui la configurazione sia corretta (good) ogni bit equivale a 0. I bit di stato che sono invece associati a condizione di errore sono:

- il **bit fail**: bit che viene posto a 1 ogni volta che si verifica un errore.
- il **bit bad**: bit che **discrimina** se l'errore è recuperabile o meno, se l'errore è recuperabile il bit viene lasciato a 0, altrimenti viene posto a 1.
- il **bit eof**: bit che viene posto a 1 quando viene letta la marca di fine stream.

Un'altra differenziazione che si può fare è quella tra errori recuperabili e errori non recuperabili, gli errori recuperabili sono errori che portano al fallimento di una singola operazione, permettendo però di attuarne delle successive quando lo stato viene ripristinato a **good**, d'altra parte le operazioni non **recuperabili** portano ad avere un blocco completo delle operazioni di ingresso e uscita.
Lo stato di uno stream può essere manipolato mediante l'uso della funzione *clear()*, la quale ha per argomento una costante intera rappresentante il nuovo valore di stato. Queste costanti sono definite come enumerazioni della classe **ios**, i tre principali sono:

- eofbit.
- failbit.
- badbit.

di default il valore dell'argomento della funzione clear è zero, portando, se invocata ad un ripristino a **good** dello stato dello stream.

Sono inoltre definite a livello di codice delle funzioni che permettono l'analisi dello stream restituendo dei valori booleani:

- fail(): restituisce **true** se almeno uno dei due bit *fail* è a 1.
- bad(): restituisce **true** se il bit bad è a 1.
- eof(): restituisce **true** se il bit eof è a 1.
- good(): restituisce **true** se i bit di stato sono tutti a 0.

è possibile controllare lo stato dello stram anche attraverso una condizione, se fatto viene controlato il bit di **fail**

```
int x;
if (cin >> x){
    cout << "Operazione corretta";
} else {
    cout << "Operazione scorretta";
    cin.clear();
}
```

11.4 Controllo del formato

Il formato di uscita dei parametri dipende da una serie di impostazioni, come, ad esempio:

- base di numerazione (*ottale*, *esadecimale*, *decimale*).
- il numero di caratteri da usare per ciascun dato.
- Il numero di cifre significative (*se presenti*)

Il controllo del formato può avvenire in vari modi, il più semplice deriva dall'uso dei *manipolatori*, dichiarati nel file *iostream.h*. I manipolatori sono puntatori a funzione, e gli operatori di lettura e scrittura sono ridefiniti in modo da accettare tali puntatori, ad esempio, *endl* è un puntatore che inserisce il carattere /n nello stream di uscita e ne svuota il buffer. Altri manipolatori utili alla creazione di programmi sono:

- *hex*: formatta il testo convertendolo in **esadecimale**.
- **dec**: formatta il testo convertendolo in **decimale**.
- **oct**: formatta il testo convertenfolo in **ottale**.
- **boolalpha**: formatta i valori booleani convertendolo i rispettivi valori nelle stringhe *true* o *false*.

Questi manipolatori hanno effetto anche sulle operazioni di lettura e selezionano la base di numerazione in cui devono essere espressi i numeri interi. Molti manipolatori una volta impostati mantengono la loro attività fintanto che non ne viene utilizzato un'altro.

Alcuni manipolatori richiedono invece di aver passato un argomento, essi sono:

- setw(int): manipolatore il cui effetto si attiva sull'istruzione di scrittura immediatamente successiva, esso stabilisce l'ampiezza del campo di stampa.
- setprecision(int): stabilisce il numero di cifre significative.
- setfill(char): stabilisce quale carattere utilizzare per arrivare alla lunghezza desiderata, di default vi è lo spazio bianco.

11.5 Utilizzo dei file

Nelle librerie di ingresso e di uscita in C++, sono dichiarate anche due classi particolari, le quali permettono l'elaborazione di file:

- *ifstream*
- *ofstream*
- *fstream*

quando viene creato un oggetto appartenente ad una di queste due particolari classi esso può essere associato direttamente ad un file che viene passato sotto forma di stringa (contente la locazione del file) al costruttore della classe.

Sempre tramite il costruttore vengono anche definite le diverse modalità di apertura del file:

- *ios::in*: permette la lettura del file.
- *ios::out*: permette la scrittura del file.
- *ios::app*: permette la scrittura in fondo al fine, la sigla *app* significa *append*.
- *ios::nocreate*: non crea il file se esso non esiste.

per gli stream *ifstream* e *ofstream* le modalità, rispettivamente di *in* e *out* sono definite di default.

Esistono anche delle modalità "*composte*", che vengono ottenute come or bit a bit delle costanti definite nella classe ios.

Quando si crea uno stream non è necessario specificare immediatamente il file che si intende aprire, è sufficente infatti dichiararlo e susseguitamente, tramite la funzione *.open("")* specificare su quale file si intende lavorare; Se l'operazione fallise lo stream andrà in uno stato di errore e la funzione membri *bad()* restituirà valore 1.

Le classe di stream dei file fanno uso degli stessi operatori e delle stesse funzioni definiti negli stream di input e output. La casella dello stream a partire dalla quale avverrà la prossima operazione è definita da un contatore, il quale può essere modificato mediante due funzioni usate rispettivamente per le due modalità di apertura di un file:

- *seekg*: nel caso di modalità di lettura del file.
- *seekp*: nel caso di modalità di scrittura del file.

quando vengono richiamate queste due funzioni è possibile specificare da uno a due parametri, specificando un parametro solo si indica la posizione di partenza nello stream, due parametri invece permette di specificare *una distanza* rispetto ad un origine.

12 Domande orale con soluzione

12.1 Domanda 1

Cosa sono i puntatori?

I puntatori sono definibili come oggetti il cui scopo principale è quello di "puntare" delle specifiche aree di memoria.

A differenza degli oggetti normali, i puntatori hanno come *r-value* l'indirizzo della variabile a cui puntano, inoltre, posso anche compiere delle operazioni sulla suddetta variabile, ad esempio:

```
int d = 10;
int* c = &d

cout << *c << endl;
```

stamperà a video il valore 10, in quanto il comando *puntatore permette di accedere al valore della variabile puntata.

Un puntatore occupa lo stesso spazio occupato da un intero, quindi **4 byte** (32 bit), se invece si trova su un sistema a 64 bit verranno occupati 8 byte.

12.2 Domanda 2

Dato il seguente codice

```
unsigned int b = 1;
unsigned int a = 3;
unsigned int c = 5;
unsigned int x = (a|c)&b;
cout << x;
```

cosa si otterrebbe ? L'output restituito sarebbe 1. Se invece l'espressione fosse stata

```
unsigned int x = a|(c&b); // Uguale anche a (a|b&c)
```

l'output restituito sarebbe stato 3.

12.3 Domanda 3

Il selection sort è un algoritmo di ordinamento nato per vettori di piccole e medie dimensioni.

Il funzionamento si basa sul cercare per ogni iterazione il minimo, per poi sostuirlo alla prima posizione disponbile andando poi a bloccare quella posizione.

L'algoritmo esegue $(n-1)$ iterazioni e scambi, ma

$$\frac{n(n-1)}{2} \tag{15}$$

confronti.

12.4 Domanda 4

dato il seguente codice il codice compila correttamente ?

Il codice compila, ma la funzione non è corretta in quanto essendo il parametro passato per valore, una volta terminata la funzione esso verrà elimintato (oggetto automatico) e di conseguenza il riferimento, referenzierà un'area di memoria vuota.

12.5 Domanda 5

Il boublesort è un algoritmo di ordinamento utilizzato per vettori di grandi dimensioni, il suo funzionamento è basato su un confronto di coppie attuo a portare il valore minimo sul fondo del vettore.

Non si riesce a definire esattamente il numero di iterazioni e di scambi, in quanto dipendono dal compilatore, in generale si può dire che la sua complessità computazionale è $O(n^2)$, mentre il numero di confronti equivale a quello del selection sort.

12.6 Domanda 6

dato il seguente codice

```
int a = 1+2*-3-1%5(4/3);
cout << a;
```

cosa viene stampato a schermo.

L'output restituito sarà -5;

12.7 Domanda 7

Dato il seguente codice:

```
char str[] = "ciao mondo\n";
cout << strlen(str);
cout << sizeof(str);
cin >> str;
cin >> str;
```

spiegare singolarmente le istruzioni.

La prima istruzione inizializza il vettore in modo esplicito, andando automaticamente a inserire il terminatore di stringa alla fine del corpo della stringa.

Il secondo comando stampa a schermo il valore 11, viene escluso infatti il terminatore di stringa.

Il terzo comando invece stampa 12, in quanto ogni char occupa un solo byte, lo spazio occupato sarà quindi 1*n (dimensione del vettore contando anche il terminatore di stringa).

Il quarto comando prende in input la stringa, se viene inserita la stringa "*ciao mondo*" l'operatore cin andrà a interrompere la lettura allo spazio, di conseguenza il primo cin prenderà come valore "ciao ", mentre il secondo "mondo".

12.8 Domanda 8

Dato il seguente codice

```
Complesso max(Complesso a, Complesso b){
    if(a>b) return a;
    else return b;
}
```

Se l'operatore di confronto è stato ridefinito il costruttore di copia viene richiamato, in quanto i parametri sono passati per valore, quindi è necessario che ne venga creata una copia.

12.9 Domanda 9

Dato il seguente codice

```
int* somma(int a, int b){
    int ris = a + b;
    return &ris;
}
```

compila ? è corretto ?

Il codice è in grado di compilare, ma presenta problemi durante l'esecuzione, in quanto ris è un oggetto automatico con visibilità limitata al blocco della funzione, andare a assegnare il suo riferimento ad un puntatore porta a una situazione di segmentation fault.

Per sistemarlo si potrebbe pensare di allocare dinamicamente l'oggetto nella memoria heap.

12.10 Domanda 10

Dato il seguente codice con $n1 = 3$ e $n2 = 2$, che succede ?

n1 non risulta divisibile per n2.

12.11 Domanda 11

Dato il seguente codice

```
int a = 5, b = 5;
cout << (a=b) << endl;
cout << (a+=b+=1) << endl;
```

L'output restituito nel primo caso sarà 5, mentre nel secondo caso viene restituito 11.

12.12 Domanda 12

Quali sono i controlli che si fanno quando si va a ridefinire l'operator '=' nelle classi ?

Quando si procede a ridefinire l'operatore '=' si deve controllare innanzitutto il controllo sull'autoassegnazione, di conseguenza per evitare che un oggetto venga assegnato a se stesso. Inoltre, è necessario controllare che il tipo di ritorno sia il puntatore all'oggetto stesso, per poter concatenare i simboli di uguali. Infine è necessario anche deallocare identificatori precedenti prima di instanziarne di nuovi.

12.13 Domanda 13

Istruzione break e continue, cosa sono e cosa fanno ?

L'operatore break appartiene, così come anche la direttiva continue, alle istruzioni di salto. Il comando break permette quindi portare a termine un ciclo nel momento in cui si incontra il comando, il continue invece ha l'effetto opposto, facendo continuare l'esecuzione del ciclo quando viene incontrato.

12.14 Domanda 14

dato il seguente codice

```
for(int i = 0; i < 7; ++i){
    if(i==2) continue;
    if(i==4) break;
    cout << i;
}
```

dire cosa stampa il codice a schermo

Il codice stampa 0, 1, 3.

12.15 Esercizio 15

dato il seguente codice

```
int a,b,c;
cin >> a >> b >> c;
cout << (a+b+c)/3 << endl;
```

se si inserisce 1 1 2 cosa si ottiene ?

l'operazione restituisce 1, in quanto la divisione in questo caso è definita tra interi e di conseguenza essendo la media 1,333 viene ingnorata la parte con la virgola.

12.16 Esercizio 16

Dato il seguente codice

```
int a = 0, b = a++, c = ++a, d = a++; // a = 3, b = 0, c = 2, d = 2
cout << a << b << c << d;
```

cosa stampa il codice ?

il codice stampa 3022;

12.17 Domanda 17

Dato il seguente codice
cosa stampa il codice ?
L'output restituito sarà la dimensione in byte di un intero moltiplicato per il numero di elementi del vettore.

12.18 Domanda 18

Dato il seguente codice cosa viene stampato a schermo ?
L'output restituito in questo caso è la dimensione del solo puntatore, quindi o 8 o 4 byte.

12.19 Domanda 19

Cosa è l'overloading delle funzioni ?
L'overloading delle funzioni è un procedimento che permette la creazione di funzioni con nome e tipo di ritorno uguale ma con parametri diversi, differenziandole anche in base a qualificatori, come ad esempio const.

12.20 Domanda 20

La keywork "nullptr" significa "puntatore = 0". Perché si preferisce usare nullptr al posto di NULL?

12.21 Domanda 21

Dato il seguente codice

```
const int& a = b;
```

cosa indica ?
Il seguente codice indica un riferimento a *presunta costante*, ciò implica che il riferimento andrà a referenziare un oggetto che non può essere modificato, ma di cui si può solo leggere il valore.

12.22 Domanda 22

Perché in alcuni casi nelle intestazioni delle funzioni si mette la keyword "const"?
La parola chiave **const** inserita all'interno dell'intestazione delle funzioni rende la funzione incapace di modificare gli identificatori della classe, una funzione const è infatti una funzione membro in grado di accedere a tutti gli identificatori della classe senza però poterne modificare il valore.

12.23 Domanda 23

Dato il seguente codice

```
char str[10];
cin >> str;
```

cosa succede ?
lo strem di input preleverà la stringa carattere per carattere fermandosi fino a che non incontra uno spazio, successivamente procede a inserire il puntatore nelle varie celle della memoria stack, andando anche ad occupare delle caselle in cui non potrebbe accedere.

12.24 Domanda 24

dato il seguente codice cosa stampa ?
Il valore stampato sarà 4, in quanto il passaggio di un vettore passa un puntatore alla prima posizione del vettore, di conseguenza verrà restituita la dimensione del puntatore.

13 Osservazioni sul codice

13.1 Incremento/Incremento postfisso

quando viene eseguito un incremento postfisso senza parentesi tonde dentro a un cout, esso viene valutato dopo il cout, ad esempio

```
#include <iostream>

int incrementa(int &a){
    a = a + 1;
    return a;
    a = 8;
}

int main(){
    int v = 6;
    incrementa(v);
    std::cout<<v--;
    return 0;
}
```

il codice in questo caso restituisce sette visto che il decremento postfisso viene valutato dopo rispetto al cout.

13.2 Ambiguità nella chiamata

Quando si dichiara due funzioni cambiando solo l'ordine dei parametri senza cambiarne il tipo si ha una chiamata ambigua e il codice non compila correttamente.

13.3 Puntatori

Quando si dichiara due puntatori, andando a compiere delle operazioni tra di essi si potrebbe incorrere in due problematiche principali, assumiamo di avere due puntatori p e q

```
int* q;
int* p;
```

è possibile adesso compiere diverse operazioni

$$p = q \tag{16}$$

quest'operazione è legittima e indica l'assegnamento del riferimento di q a p, i due puntatori dopo l'assegnamento punteranno infatti alla stessa varaibile. Un'operazione che invece è errata dal punto di vista logico è

$$p = \&q \tag{17}$$

quest'operazione risulta sbagliata in quanto per il codice è un assegnamento tra tipi diversi, nello specifico è un assegnamento della forma

```
int* = int**
```

anche l'operazione risulta sbagliata per lo stesso motivo, soltanto che in questo caso i tipi che vengono assegnati sono

$$int* = int \tag{18}$$

13.4 Assegnamento

L'operatore di assegnamento restituisce un *left-value*, ad esempio l'operazione

```
int a = 100;
int b = 1;

(b = a)
```

restituisce come valore b, è infatti possibile compiere anche delle vere e proprie concatenazioni di queste operazioni

```
(b = a) = 400
```

quest'operazione prima assegna alla variabile b il valore di a, successivamente assengna al valore di b, il quale viene restituito dopo l'assegnamento con a, il valore 400, quindi alla fine di tutti gli assegnamenti si avrà $b = 400$ e $a = 100$.

Un caso interessante da analizzare è quando vengono fatte le operazioni di

```
++i = i;
++i = i++;
i = i++ // Se i è uguale a 1, l'operazione, stampando i darà come risultato 1
```

Nel primo caso il valore di i viene incrementato di 1 e viene successivamente comparato al valore di i non incrementato di 1, nel secondo caso avviene uguale, il valore di i viene incrementato di 1 e successivamente viene comparato al valore di i non incrementato di 1, l'incremento postfisso infatti ha priorità minore rispetto all'assegnamento.

Analogamente, l'assegnazione di un valore a i++ o ++i non è consentita in C, poiché entrambi sono valori e non locazioni di memoria modificabili (non sono lvalues). Tentare di assegnare un valore a i++ o ++i genererà un errore di compilazione.

Quindi, sia i = i++; che i++ = ++i; sono esempi di comportamenti non definiti in C, e la scrittura di tali espressioni può portare a risultati imprevedibili. È consigliabile scrivere codice chiaro e leggibile, evitando situazioni ambigue o comportamenti non definiti.

Tutte le operazioni aritmetiche, condizionali, comparative e bit a bit restituiscono dei right-value.

13.5 Stringhe

Copiare una stringa in un'altra con l'strcpy porta a una scrittura del buffer della stringa di destinazione, nel caso però in cui la stringa sorgente abbia grandezza maggiore della stringa di destinazione succede che avviene una scrittura consecutiva che va oltre alla dimensione del buffer destinazione, in altre parole, vengono prese delle celle che non appartengono alla stringa destinataria. Quando ciò avviene, se si stampa la stringa destinataria verrà comunque stampato tutto il messaggio, in quanto quando si stampa una stringa fintanto che non viene trovato un terminatore si continua a stampare, arrivando fino all'*end-of-stream*, ma se si procede ad operare sulla stringa destinataria si potrà solo operare per la grandezza della sua cella di locazione.

13.6 Unioni e vettori

Lo spazio occupato da un unione è lo spazio del tipo dell'unione con la grandezza maggiore, ad esempio, se ho un union con dentro un char e un double, lo spazio occupato dall'union sarà quello del double, quindi 64bit o 8byte. Quando si ha invece una struct normale lo spazio occupato è la somma degli spazi occupati dai vari campi della struttura.

13.7 Note su incremento e decremento

Incremento prefisso (`++i`):

- Incrementa la variabile i prima di utilizzare il suo valore nell'espressione.
- Restituisce il nuovo valore incrementato della variabile.

Incremento postfisso (`i++`):

- Restituisce il valore corrente di i nell'espressione.
- Dopo l'uso nell'espressione, incrementa il valore di i.

Espressioni come `i = i++` o `(++i)++`:

- Possono comportare comportamento indefinito a causa dell'ordine non definito di valutazione degli operandi.
- Il risultato può variare tra i compilatori e le ottimizzazioni.

13.8 Note

- è possibile compiere delle conversioni implicite tra **double** e **char**, come anche tra **unsigned int** e **char**.
- se utilizzo un operatore di negazione logica su una variabile intera essa verrà trattata come un bool tale che:
$$\begin{cases} 1 & x \neq 0 \\ 0 & x = 0 \end{cases} \tag{19}$$
di conseguenza un operazione del tipo
```
cout << boolapha << (!3) << endl;
```
stamperà come valore 0.
- è possibile compiere operazioni bit a bit tra:
  - **int** non **unsigned**.
  - **char** (carattere): viene letto come valore intero.
  - **enum**: viene letto il valore associato come intero
- La grandezza di una matrice bidimensionale nello stack è misurata come **n*n*sizeof(type)**.
- In alcuni casi le dimensioni di alcune strutture possono essere aumentate aggiungendo del padding per ottenere dei miglioramenti di prestazione.
- Lo spazio occupato da un puntatore occupa 4 byte su sistemi a 32 bit e 8 a sistemi su 64 bit.
- Una matrice **bidimensionale** nell'heap ha dimensione definita come n*n*sizeof(type) + n*sizeof(type*).
- Un riferimento ha sizeof pari al tipo.
- Scrivere **type long nomeVariabile** non ha alcun senso.
- è possibile fare l'overloading di funzioni differenziandole solo in base al qualificatore const nei parametri di una classe.
- Tra i vari operatori di confronti la priorità è ! > && > ||.
- Per via dell'incizzazione come unsigned int, sugli enum si può compiere ogni operazione.
- Non si può prendere in input un enum.
- Una matrice mat[n][m] se si chiede il sizeof(m[q]) restuisce il sizeof della colonna.
- Una matrice automatica, se utilizzata come argomento formale di una funzione, deve avere specificata la dimensione delle colonne.
- Non è possibile definire esattamente la dimensione di un tipo enumerazione.