Università degli Studi di Catania

FACOLTÀ DI MATEMATICA E INFORMATICA
Corso di Laurea in Informatica

Riassunti di:

Programmazione I

Candidato:
Salvo Polizzi Salamone
Matricola 1000030092

Relatore:
Prof. Giovanni Maria Farinella

Anno Accademico 2021-2022

Indice

Capitolo 1

PROBLEMI, ALGORITMI, DIAGRAMMI DI FLUSSO

1.1 Gli algoritmi

Un **algoritmo** è una sequenza di passi concepita per essere eseguita automaticamente da una macchina. Ovviamente alla base della creazione di un algoritmo vi è un problema che si dice **computabile** se è risolubile tramite algoritmi.

Esempio Un esempio di algoritmo potrebbe essere la ricetta di un risotto che si articola in diversi passi:

- Procurare gli ingredienti
- Seguire la ricetta
- Servire il piatto

Da qui capiamo come sia importante capire che la creazione della ricetta, ovvero la *risoluzione* dell'algoritmo possa avvenire solo tramite la mente umana; sarà poi la macchina ad *eseguire* l'algoritmo

1.1.1 Le caratteristiche di un algoritmo

Un algoritmo per funzionare deve avere determinate caratteristiche:

- Non può essere **ambiguo**: la parola "abbastanza" non è contemplata all'interno di un algoritmo o tutte le ambiguità derivate dalla lingua
- **Determinismo**: l'istruzione successiva alla precedente può essere solo una e ben determinata
- Deve avere un numero **finito** di passi. Ciò significa che:

  - ogni passo deve terminare in un tempo finito
  - ogni passo deve produrre un effetto visibile
  - ogni passo deve produrre sempre lo stesso effetto

- **Terminazione**: l'algoritmo deve finire, ovvero deve avere una terminazione e non può essere infinito[1].

1.1.2 Codifica di algoritmi

Un algoritmo viene *codificato* solitamente da linguaggi di alto livello, cioè molto distanti dal linguaggio macchina. Quindi il **programma** che viene creato è il risultato di tale codifica. E' importante osservare come bisogna attribuire la giusta **semantica**, ovvero il giusto significato a ogni passo dell'algoritmo, in modo tale che la macchina riesca a comprendere senza **ambiguità**, caratteristica fondamentale degli algoritmi, ogni passo in modo corretto

1.1.3 Progettazione di algoritmi

Prendiamo a esempio sempre la ricetta del risotto in cui nel primo passo indica "preparare il brodo vegetale": questa istruzione non può essere considerata finale poichè non sappiamo in partenza come si prepara un brodo, allora si vengono a creare dei sottoproblemi che ci specificano come fare a prepararlo; per cui si crea un ulteriore algoritmo che ci permette di risolvere questo problema[2]. Scomponiamo quindi il problema generale in sottoproblemi piu semplici. Tale approccio viene denominato **top-down** poichè risolviamo un problema generale (top) attraverso decomposizioni del problema (down), come se fosse un albero dove il problema è il tronco e i sottoproblemi i vari rami. A tale approccio si contrappone quello **bottom-up** dove si parte da parti specificate in dettaglio e si arriva attraverso le connessioni/composizioni del problema si arriva a risolvere il problema generale

1.2 Diagrammi di flusso

Gli algoritmi possono essere descritti in 2 modi:

- **Diagrammi di flusso**
- **Pseudo-codice**

I diagrammi di flusso rappresentano graficamente gli algoritmi e si sviluppano attraverso blocchi e connettori. I tipi di blocchi sono 3:

[1] **NB**: Un algoritmo con un numero finito di passi potrebbe non terminare mai quindi non soddisfare la caratteristica di terminazione di un algoritmo

[2] **NB**: è importante comprendere che creare dei sottoproblemi, quindi algoritmi, ci può aiutare a risolvere altri problemi poichè riusciamo a rendere **modulabile** la nostra soluzione e quindi magari se ci servisse la ricetta del brodo vegetale potremmo riutilizzare questo algoritmo in un altra ricetta dove è richiesta la preparazione di un brodo

- Blocchi di **inizio** e **fine**, che indicano rispettivamente dove inizia l'algoritmo e dove finisce. Si indicano con dei cerchi

- Blocchi di **input/output**, che indicano di leggere a input una variabile o di stamparla a video. Si indicano con dei parallelogrammi

- Blocchi di istruzioni **imperative**, dove si indicano delle istruzioni alle variabili o delle espressioni. Si indicano con dei rettangoli

1.2.1 Variabili e espressioni

Abbiamo accennato poco fa alle **variabili**, che non sono altro che delle celle di memoria, o "contenitori", nel quale conserviamo qualcosa dentro. Una variabile presenta delle caratteristiche:

- Un **nome**. Esempio: X, Y, PIPPO

- Un **valore**: quando dichiariamo la variabile stiamo utilizzando uno spazio di memoria, quindi dobbiamo porre un valore ad essa sennò sprecheremmo memoria. Per questo si dice che **inizializziamo** la variabile assegnandogli un valore; questo valore può essere anche un **espressione** ovvero una combinazione di operatori aritmetici che danno un risultato finale (Esempio: $X \leftarrow Y + 2$)

Esempio: scambio di due numeri Come possiamo fare data una X e una Y in input a scambiare il valore al loro interno? Ragioniamo: non possiamo cancellare entrambi e scriverle in modo da averle scambiate poichè la nostra memorie non è come quella di un computer che una volta cancellati i valori delle variabili non riesce a ripescarli e non possiamo nemmeno fare una cosa del tipo $X \leftarrow Y$ e $Y \leftarrow X$ poichè assegneremmo alla variabile X il valore di Y e alla variabile Y il valore $X = Y$. Abbiamo bisogno quindi di un ulteriore spazio di memoria **ausiliare** nel quale conservare il valore di una delle due variabili: abbiamo bisogno quindi di una variabile ausiliaria che conservi il valore di una cosi da assegnarlo all'altra e scambiare quindi effettivamente le due variabili.

1.2.2 Istruzioni condizionali e cicli

Il flusso può scegliere anche scegliere la direzione da prendere per cui se si verifica una **condizione** si fa un blocco di codice, **altrimenti** si fa un altro blocco di codice. In questo caso si tratta di un'**istruzione condizionale**, che differisce dal **ciclo** o **loop**, in quanto in quest'ultimo vi è sempre una condizione che però fa si che fino a quando essa si verifica, cioè fino a quando è vera, si continuano ad eseguire le istruzioni all'interno del ciclo; quando diventa falsa si esce dal ciclo e si passa all'istruzione successiva. La condizione nei diagrammi di flusso si indica attraverso un **rombo**

1.3 Notazione lineare-strutturata

I diagrammi di flusso sono sistemi che non sono molto adatti a descrivere algoritmi poichè molto confusionari e poco leggibili e per questo viene utilizzata la **notazione lineare-strutturata**, nel quale utilizziamo tre strutture per descrivere gli algoritmi

Ogni algoritmo può essere descritto tramite tre strutture, come dice il **teorema di Bohm-Jacopini**:

- **Sequenza**, nel quale indichiamo le istruzioni in maniera appunto sequenziale. Esempio:

```
INIZIO
ISTRUZIONE 1;
ISTRUZIONE 2;
ISTRUZIONE 3;
FINE
```

- **Selezione**, nel quale abbiamo due blocchi nel caso si verifichi o meno la condizione. Esempio:

```
INIZIO
IF(CONDIZIONE)THEN
    BLOCCO 1;
ELSE
    BLOCCO 2;
END IF
FINE
```

- **Iterazione**, nel quale possiamo ciclare un blocco. Esempio:

```
INIZIO
WHILE (CONDIZIONE) DO
    BLOCCO 1;
END WHILE
BLOCCO 2;
FINE
```

Nell'iterazione è molto importante comprendere che all'interno del blocco devo fare in modo che prima o poi la condizione diventi falsa in modo da non creare un loop infinito[3].

[3] **NB**: possono esserci diversi algoritmi che risolvono un problema e che quindi hanno la stessa soluzione ma alcuni possono essere piu efficienti di altri

Stampa i numeri da 1 a N

```
INIZIO
LEGGI N
M = 0
WHILE (M<N) DO
    M= M+1
    STAMPA M
END WHILE
FINE
```

Ragioniamo su come funziona questo ciclo while nel programma: innanzitutto da consegna dobbiamo fare un ciclo che stampi i primi N numeri quindi abbiamo proceduto nelle seguenti fasi:

- Viene allocata una zona di memoria per salvare il valore di N che ci viene dato in input
- Abbiamo **inizializzato** la variabile M assegnandogli il valore 0, questa variabile è fondamentale inizializzarla a 0 poichè vediamo che all'interno del ciclo si incrementa di 1, di conseguenza se avessimo inizializzato la variabile con 1 avrebbe stampato i numeri da 2 a N
- Abbiamo imposto come **condizione** del ciclo $M < N$ poichè ovviamente quando arriviamo a N, avrà già stampato N poichè la prima istruzione dentro al ciclo incrementa già di 1. Quindi se siamo arrivati in quel momento in cui M= N-1 subito M diventa M= N-1+1=N per cui stampa N
- All'interno del ciclo la prima istruzione M= M + 1 rappresenta il contatore, ovvero quella variabile grazie al quale riusciamo a "scorrere" i numeri da 1 a N. La seconda istruzione stampa il valore incrementato della variabile M

ATTENZIONE: se avessimo scambiato le istruzioni all'interno del ciclo:

```
STAMPA M
M = M+1
```

il programma non avrebbe svolto la consegna a noi richiesta poichè avendo inizializzato a 0 la variabile M avrebbe stampato i numeri iniziando da 0 e inoltre, visto che l'incrementatore è posto dopo la stampa, avrebbe stampato i numeri fino a n-1. Quindi ciò significa che se c'è un errore logico da parte dell'esecutore il compilatore della macchina potrebbe eseguire il codice nonostante la **semantica** sia sbagliata[4].

[4]Possiamo valutare se un programma è migliore di un altro in termini di **efficienza di tempo** e **spazio**. Quindi devo avere meno istruzioni o meno variabili possibili, nel caso appena studiato si potevano utilizzare altre soluzioni tutte corrette

Somma dei primi N numeri

```
LEGGI N
i = 0
S = 0
WHILE (i < N) DO
    i = i + 1
    S = S + i
END WHILE
STAMPA S
```

Ragioniamo su come funziona questo programma: innanzitutto ho bisogno, per sommare i primi N, di fare delle **somme parziali** e conservarle in una variabile e di una variabile che faccia da contatore che incrementi così da scorrere tutti i numeri fino a N:

- Abbiamo assegnato inizialmente al valore della variabile i, **contatore**, uguale a 0, così da scorrere da 1 a N
- Abbiamo assegnato il valore della variabile S che conterrà le somme parziali uguale a 0 poichè sappiamo che non altererà il risultato della somma
- Abbiamo imposto la condizione $(i < N)$ in modo tale che i scorri i numeri fino a N e poi esca dal ciclo
- Abbiamo dentro il ciclo i che incrementa di 1 fino all'ultima istruzione in cui i=N-1 e di conseguenza N-1+1=N e di conseguenza nell'istruzione successiva abbiamo conservato tutte le somme parziali in uno spazio di memoria da 1 a N
- Fuori dal ciclo stampiamo le somme parziali conservate in S

Capitolo 2

ARRAY

Fino ad oggi abbiamo creato dei programmi con le singole variabili, che contenevano un numero, un testo... Se però volessimo conservare piu elementi all'interno di un area di memoria possiamo utilizzare gli **array monodimensionali**, ovvero con una riga e ad esempio 10 spazi: $A[i], 0 \leq i \leq 9$.

2.1 Definizione di array monodimensionale

L'array monodimensionale è quindi una struttura dati **omogenea**, cioè i dati vengono scritti uno dopo l'altro nelle celle e gli elementi scritti all'interno di un array devono essere dello stesso tipo (numero, stringa...). Gli indici che indicano la posizione della cella all'interno dell'array vanno da 0 a n-1 dove n rappresenta la dimensione dell'array, ovvero quante celle ci sono nell'array.

2.1.1 Lettura e scrittura dell'array

Gli array si indicano con il nome dell'array seguito dalla parentesi quadra che indica l'indice della cella di quell'array (Esempio: a[2], valore presente nell'array a e nella cella che ha indice 2). Possiamo sia assegnare un valore a una determinata cella:

```
a[2] = 5
```

Possiamo inoltre assegnare il valore di un determinato array a una variabile:

```
Y = a[2]
```

2.2 Cicli con array monodimensionali

Adesso che conosciamo la notazione generale di un array monodimensionale vediamo come possiamo strutturare dei cicli utilizzando questi ultimi

2.2.1 Impongo a 0 tutti gli elementi di un array

Ragioniamo sui passi da compiere se volessimo porre tutti gli elementi di un array a 0. Innanzitutto dovremmo implementare un ciclo che scorra gli indici da 0 a n-1 e assegnare 0 a ogni elemento dell'array:

```
INIZIO
i = 0
WHILE (i < N)
    a[i] = 0
    i = i + 1
END WHILE
```

Ragionando sul programma abbiamo quindi **inizializzato** l'indice i dell'array a 0. In seguito abbiamo posto come condizione che $i < N$ cosicchè il ciclo si fermi dopo che $i = N - 1$; e dentro il ciclo abbiamo assegnato ad a[i] il valore 0 come richiesto da consegna e con il contatore $i = i + 1$ abbiamo fatto scorrere tutti gli elementi dell'array fintanto che i = N-1

2.2.2 Somma degli elementi di indice pari

Abbiamo un array V con N elementi e dobbiamo sommare gli elementi con idici pari:

```
LEGGI N
LEGGI V
S = 0
i = 0
WHILE (i < N) DO
    S = S + V[i]
    i = i + 2
END WHILE
STAMPA S
```

Ragioniamo sul programma: abbiamo detto che N è la dimensione dell'array (che avrà indici che vanno da 0 a N-1) e che V è la variabile contenente l'array. Le osservazioni importanti da fare sono che ovviamente nella condizione l'indice i deve essere minore di N, poichè arriva fino a N-1; nelle somme parziali sommo S con V[i] che vediamo che il contatore incrementa di 2, partendo da 0, poichè da consegna ci chiedeva di sommare i primi elementi con i pari. Infine tutte le somme parziali (S + V[0], S+ V[2], ..) vengono stampate fuori dal ciclo

2.3 Definizione di array bidimensionali

L'**array bidimensionale** o *matrice* ($a[i][j]$), quindi capiamo già del nome, è un array formato da righe e colonne. Indichiamo l'indice della **riga** nella prima parentesi quadra, e l'indice della **colonna** nella seconda parentesi quadra. Ad esempio: V[2][3] → valore presente in riga 2 e colonna 3. Possiamo assegnare il valore a una cella come facevamo negli array monodimensionali, e anche assegnare il valore di una cella a una variabile allo stesso modo. Esempio: V[2][3] = 5; Y = V[2][3]

2.4 Cicli con array bidimensionali

In questo caso, sapendo che gli array bidimensionali presentano righe e colonne, possono essere utili anche **doppi cicli** che ciclino sia le righe che le colonne.

2.4.1 Somma tutti gli elementi dell'array o matrice

Sommiamo tutti gli elementi dell'array bidimensionale V di $N \times M$ elementi:

```
Inizio
 LEGGI N;
 LEGGI M;
 LEGGI V;
 S ← 0 ;
 i ← 0 ;
 WHILE ( i<N) DO
 j ← 0 ;
 WHILE ( j<M) DO
 S ← S + V[i][j] ;
 j ← j + 1
 End While
 i ← i + 1
 end WHILE
 Stampa S ;
 Fine
```

Ragioniamo su come abbiamo impostato il programma:

- Abbiamo chiesto in input N, che rappresenta la variabile contenente le righe dell'array; M, che rappresenta la variabile contenente le colonne dell'array e V che è la variabile che contiene l'intero array
- Abbiamo **inizializzato** S, variabile contenente le somme parziali, e i, variabile che è il contatore degli indici delle righe.
- Abbiamo costruito due cicli:

  - Il ciclo piu **interno** che scorre gli elementi delle colonne fissato l'indice della riga, fino a quando si arriva alla colonna con indice M-1;
  - Il ciclo piu **esterno** dove è inizializzato il contatore degli indici delle colonne (j) e che incrementa di 1 la riga, fino a quando si arriva alla riga con indice N-1;

- Infine stampiamo le somme parziali dei vari elementi dell'array

In conclusione quindi noi fissando una riga e incrementandola a fine del ciclo esterno e scorrendo le colonne nel ciclo interno eseguendo le varie somme riusciamo a sommare tutti gli elementi della matrice.

2.5 Array k-dimensionali

Oltre agli array mono e bidimensionali esistono anche quelli **k-dimensionali**: $A[X][Y][Z]$. Un esempio può essere quello a tre dimensioni, cioè abbiamo tre "parametri" su cui lavorare e se fissiamo un indice otteremo un array (k-1) dimensionale: V[2][4][1] è un array k-dimensionale.

Figure 2.1: array tridimensionale

Capitolo 3

TRADUZIONE

I linguaggi di programmazione ad oggi conosciuti sono classificati in tre tipi:

- Linguaggio **macchina**: codice formato da soli 0 e 1, comprensibile quindi solo dalla macchina
- Linguaggio **assembly**: linguaggio a basso livello, è una versione leggibile dagli umani del linguaggio macchina
- Linguaggi ad **alto livello**: sono i linguaggi piu utilizzati e diffusi. I motivi sono essenzialmente 3:
  - **Astrazione**: ovvero non mi preoccupo di allocare un certo spazio di memoria per le variabili, di controllare i registri... quindi in questo senso i linguaggi ad alto livello sono astratti rispetto alla macchina
  - **Semplificazione**: è ovvio che grazie a questo tipo di astrazione, i linguaggi ad alto livello siano piu comprensibili dagli umani e ciò semplifica in qualche modo la scrittura di un programma da parte dell'umano
  - **Similarità con il linguaggio umano**

3.1 Paradigmi di programmazione

Per comprendere al meglio la *filosofia* di programmazione e in qualche modo la **metodologia** e la **computabilità** con cui vengono scritti esistono i cosiddetti **paradigmi di programmazione**. Ne esistono di diversi tipi:

- Programmazione **funzionale**: si tratta di un tipo di programmazione il cui flusso è una serie di **funzioni matematiche**, è ha le peculiarità di riscontrare pochi errori all'interno di un programma ma è molto lontana dal linguaggio comprensibile agli umani

- Programmazione **logica**: si tratta di un tipo di programmazione che segue la struttura logica con cui scriviamo un programma, quindi abbiamo variabili proposizionali, deduzioni logiche...

- Programmazione **imperativa**: si tratta di un tipo di programmazione caratterizzata da una sequenza di istruzioni che si susseguono. Esempio: "Leggi A", "Stampa B"...

- Programmazione **strutturata**: si tratta di un tipo di programmazione basata sul **teorema di Bohm-jacopini** che dice che ogni algoritmo puo essere scritto tramite i 3 costrutti di **sequenza**, **selezione** e **iterazione**

- Programmazione **procedurale**: si tratta di un tipo di programmazione basata su procedure o **funzione**, in cui ognuna di queste è formata da un blocco di istruzioni imperative e un blocco di programmazione strutturata

- Programmazione **modulare**: si tratta di un tipo di programmazione basata su **moduli**, che sono uno indipendente dall'altro e sono separati da delle **interfacce**. E' chiamata anche la tecnica di **design del software**

- Programmazione **orientata agli oggetti**: si tratta di un tipo di programmazione che contiene i paradigmi di programmazione imperativa, procedurale e modulare

3.2 Sintassi e semantica dei linguaggi di programmazione

Abbiamo compreso cosa si intenda per paradigma di programmazione, ma entrando nello specifico nei linguaggi di programmazione molto spesso si confonde la **sintassi** con la **semantica** di quest'ultimo. Vediamo in dettaglio quindi cosa si intende per sintassi e semantica:

- La **sintassi** di un linguaggio di programmazione identifica il modo con cui dobbiamo scrivere i "simboli" o dati o parole e combinarle tra loro per far si che l'istruzione possa essere **compilata** correttamente

- La **semantica** di un linguaggio di programmazione identifica il **significato** di un istruzione scritta con la giusta sintassi. Ovvero quello che poi effettivamente produrrà in outpout il calcolatore

In conclusione è molto piu importante comprendere la semantica delle istruzioni che scriviamo in un programma, poichè se anche avessimo una giusta sintassi non è detto che anche la semantica sia giusta. Avere una semantica sbagliata significa fare un qualcosa che non ti era stato richiesto e quindi sbagliato.

3.3 Traduzione

I linguaggi di programmazione nel corso del 900 fino ad oggi si sono evoluti moltissimo sviluppando l'espressività e la semplicità che caratterizza i linguaggi ad **alto livello** Un programma quando è scritto in un linguaggio ad alto livello deve essere tradotto in un programma in linguaggio macchina: per questo esistono i **compilatori** e gli **interpreti**:

- Gli interpreti sono quei traduttori che ogni volta che quel programma deve essere eseguito, deve avvenire una loro traduzione nonostante magari era stata già fatta in precedenza.
- I compilatori sono quei traduttori che, a differenza degli interpreti, non hanno bisogno di successive traduzioni, ma una volta che avranno compilato un determinato programma non dovranno rifare lo stesso procedimento.

3.3.1 Pro e contro di interpreti e compilatori

Come abbiamo potuto iniziare a capire quindi in termini di benefici il compilatore è quello che ne offre di più:

- **Performance**: codice sorgente eseguito preventivamente rispetto all'interprete che non ha questo tipo di esecuzione
- **Efficienza**: non occupa risorse durante l'esecuzione del programma, rispetto all'interprete che occupa molte risorse
- **Non invasivo**
- L'unico punto "negativo" dell'interprete è la **portabilità su architetture differenti**, infatti se scriviamo un codice per un architettura dovremo fare una *cross-compilazione* per trasportarlo su un altra architettura. Sull'interprete invece, non ci preoccupiamo dell'architettura nel quale scriviamo il codice.

Capitolo 4

INTRODUZIONE AL LINGUAGGIO C++

Il **linguaggio c++** nasce come estensione del linguaggio c ed è basato sul paradigma di **programmazione orientata agli oggetti**. Ha diversi pro:

- E' **semplice ed efficiente**
- Ha le caratteristiche dei **linguaggi orientati agli oggetti**
- Il codice è molto facile da x**leggere** e da **mantenere**
- **Prototipazione** veloce del software

4.1 Innovazione

Nel corso degli anni siamo quindi passati da un **linguaggio assembly**, che codificava simboli comprensibili dall'umano ma abbastanza complessi da scrivere, poichè molto vicini al linguaggio macchina; a un linguaggio **procedurale** dove vi è gia una complessità di dati attraverso diverse operazioni; e infine siamo arrivati all'**OOP** dove vi sono programmi con complessità avanzata

4.1.1 Modello di computazione

Quindi se nell'assembly dobbiamo conoscere l'**architettura** del calcolatore per scrivere programmi, nel linguaggio **procedurale** invochiamo funzioni e procedure attraverso uno specifico flusso di dati, nell'**OOP** vi sono oggetti che scambiano messaggi tra loro attraverso interfacce

4.2 C vs C++

Abbiamo compreso le varie funzionalità che c++ ci permette di svolgere. Inoltre presenta diversi pro in più rispetto a C:

- In c++ abbiamo la possibilità di progettare **interfacce grefiche**
- In c++ esiste l'**incapsulamento** che in c non c'è
- C++ favorisce la **modularità**
- C++ permette una migliore manutenzione

4.3 Concetti base OOP

Fondamentalmente la programmazione orientata agli oggetti si basa su **classi** che identificano delle categorie (esempio: classe persona).

4.3.1 Incapsulamento

I membri della classe (dati e metodi) possono essere **pubblici** o **privati**, ovvero possono essere richiamati dall'esterno (pubblici) o no (privati), quindi posso nascondere informazioni all'esterno.

4.3.2 Astrazione

In c++ possiamo definire classi astratte che insieme all'incapsulamento nascondono i dettagli implementativi all'esterno, esponendo solo le interfacce, permettendo così di fare **information hiding**

4.3.3 Ereditarietà

In c++ esistono le gerarchie di classi, ovvero classi derivate di una principale che aggiungono qualcosa e mantengono tutte le caratteristiche della principale. Esempio: motore(classe base) ← motore diesel, motore benzina (classe derivata)

4.3.4 Polimorfismo

Le classi possono assumere forme, aspetti, modi di essere diversi in base alla circostanza. Questa caratteristica viene chiamata **polimorfismo**

Capitolo 5

VARIABILI, TIPI E OPERAZIONI PRIMITIVE IN C++

5.1 Variabili

Una **variabile** è un contenitore di dati, che corrisponde a un preciso indirizzo di memoria, e associata da un **nome** e in c++ da un **tipo**. Esempio:

```
int numero = 10;
```

Analizzando tale variabile stiamo dichiarando:

- Il tipo della variabile: **int** (intero), la variabile conterrà un intero
- Il nome della variabile: **numero**, conviene sempre specificare il contenuto della variabile nel nome cosicchè se in futuro dovessimo riguardare il nostro codice capiremmo cosa c'è contenuto dentro la variabile
- In c++, come abbiamo visto in questo caso, possiamo anche inizializzare la variabile, assegnandogli il valore 10. Potremmo però anche non inizializzarla.[1]
- il carattere "**;**" è fondamentale e si deve mettere al termine di ogni istruzione

5.1.1 Regole delle variabili in c++

Vi sono diverse regole da rispettare per dichiarare le variabili in c++:

- Possiamo assegnare alla variabile la somma, prodotto o qualunque operazione aritmetico-logica tra 2 variabili

[1] conviene sempre inizializzare le variabili così da non commettere errori di assegnamento

- Non possiamo assegnare a una variabile intera una stringa o comunque bisogna fare molta attenzione al tipo della variabile. Esempio:

```
int codice = "AA10"
```

  In questo caso ad esempio abbiamo non solo commesso l'errore di mettere le virgolette che indicano la stringa in una variabile che può contenere solo interi ma abbiamo scritto anche dentro la stringa delle lettere e dei numeri

- Possiamo dichiarare e inizializzarepiù variabili dello stesso tipo contemporaneamente. Esempio:

```
int v1, v2=3, v3
```

- Il nome della variabile deve iniziare con una lettera oppure un underscore e non possono esserci spazi.

- C e C++ sono case-sensitive, ovvero "var" e "Var" sono due variabili diverse

5.1.2 Valori letterali e variabili

Quando inizializziamo una variabile si dice che le stiamo assegnando un **letterale**, che sarebbe il valore contenuto all'interno della variabile: true, 1.0, 40, "Acqua" sono ad esempio letterali di tipo rispettivamente booleano, double, intero e stringa.

5.1.3 Cambiare il valore di una variabile

Abbiamo visto che un primo modo per inserire un valore o letterale all'interno della variabile è inizializzarla :

```
a = 10;
a = a + 10;
```

In realtà vi sono anche altri due modi per far cambiare il valore della variabile nel corso del tempo:

- **Incremento/decremento**: può avvenire in maniera *prefissa*, ovvero prima incremento o decremento il valore e poi lo assegno alla variabile, oppure *postfissa*, in cui prima assegno il valore alla variabile e poi incremento o decremento questo valore. Esempio:

```
a++; a--
++a; --a
```

In questo caso nella prima riga era un incremento/decremento postfisso, nella seconda riga era incremento/decremento prefisso

- Istruzione di **input**: in questo caso chiediamo al utente di inserire il valore in input che poi verrà inserito in una variabile:

```
cin >> x;
```

5.1.4 Variabili costanti

Le variabili **costanti**, contengono appunto valori costanti, ovvero valori che **non cambiano** nel corso del programma e che devono essere inizializzati in fase di creazione. Queste vengono spesso usate per migliorare la **leggibilità del codice** così da evitare errori. Si indica con "const"

5.2 Sistemi di numerazione in base 2

Il nostro sistema di numerazione decimale è **posizionale**, ovvero ogni cifra ha una posizione specifica e significativa. Esistono anche altri sistemi di numerazione posizionali; tra questi vi è quello a **base 2** che è il più utilizzato nel mondo dell'informatica.

5.2.1 Conversione da base 2 a base 10

Vediamo convertire un intero e un numero con parte frazionaria da base 2 a base 10:

- **Intero**: Prendiamo come esempio il numero $(10101010)_2 = (1 \cdot 2)^7 + (0 \cdot 2)^6 + (1 \cdot 2)^5 + (0 \cdot 2)^4 + +(1 \cdot 2)^3 + (0 \cdot 2)^2 + (1 \cdot 2)^1 + (0 \cdot 2)^0 = (170)_{10}$
- **Numero con parte frazionaria**: Prendiamo come esempio il numero 101.0101: $(101)_2 = (1 \cdot 2)^2 + (0 \cdot 2)^1 + (1 \cdot 2)^0 = (5)_{10}$; $0101 = (0 \cdot 2)^{-1} + (1 \cdot 2)^2 + (0 \cdot 2)^3 + (1 \cdot 2)^4 = 0.3125_{10}$

Analizzando i procedimenti vediamo che con gli interi basta assegnare a ciascuna posizione la potenza che va da 0 a n-1, dove n è il numero di cifre del binario; e moltiplicare la potenza di due corrispondente a ciascuna posizione con il bit corrispondente (0 o 1).

5.2.2 Conversione da base 10 a base 2

Come per la conversione da base 2 a base 10, convertiamo prima la parte intera del numero a base 10 in base 2 e poi la parte decimale:

- **Parte intera**: per convertire la parte intera in decimale eseguiamo **divisioni successive** del numero per la base 2; i resti che otterremo ordinati da sotto verso sopra costituiranno la parte intera del numero con base 2

- **Parte decimale**: per convertire la parte decimale eseguiamo **moltiplicazioni successive** della parte decimale per la base 2 e prendiamo le cifre intere che otteniamo fino a che non otteniamo la parte decimale uguale a 0

5.3 Rappresentazione dei numeri reali con il calcolatore

Quando rappresentiamo dei numeri al calcolatore abbiamo a che fare con un certo numero di bit, più comunemente 32 o 64 bit. Per questo sono nate delle codifiche standard dei numeri reali in binario e quello che utilizzeremo come punto di riferimento è lo **standard IEEE 754**.

5.3.1 Rappresentazione di interi

Quando rappresentiamo un intero dobbiamo distinguere se abbia o meno il **segno**, poichè questo occupa un bit nella codifica e poi rappresentiamo il **valore assoluto** di tale numero. Quindi distinguiamo la rappresentazione di interi con o senza segno per quelli che sono effettivamente i valori rappresentabili:

- **Interi senza segno**: in questo caso i valori rappresentabili vanno da 0 a $2^n - 1$: se ad esempio abbiamo 32 bit a disposizione per codificare un intero senza segno; possiamo codificare i valori che vanno da 0 a $2^{32} - 1$

- **Interi con segno**: in questo caso dobbiamo shiftare i valori rappresentabili contando un bit di segno, quindi dobbiamo sottrarre anche questo valore. Di conseguenza la rappresentazione dei valori possibili è $\pm 2^{n-1} - 1$: se ad esempio abbiamo sempre 32 bit a disposizione i valori $\pm 2^{31} - 1$

5.3.2 Rappresentazione di numeri con virgola mobile

I numeri reali nel calcolatore vengono rappresentati attraverso il formato a **virgola mobile** o *floating point*, poichè anche in questo caso abbiamo uno spazio limitato di bit, solitamente o 32 o 64 bit. Utilizzando lo standard IEEE 754, abbiamo delle specifiche caratteristiche di codifica per ogni floating point:

- **Segno**: occupa un bit, sia che la codifica avvenga in uno spazio di 32 o 64 bit

- **Esponente**: occupa 8 bit in una codifica con **precisione** a 32 bit, e 11 bit in una codifica con **precisione** a 64 bit

- **Mantissa o significando**: si tratta delle cifre che codificano la parte frazionaria del numero e occupano 23 bit in una codifica con **precisione** a 32 bit, e 52 bit in una codifica con **precisione** a 64 bit

Mantissa ed esponente

Sappiamo che il segno viene rappresentato con 0(+) e con 1 (−). Abbiamo introdotto anche i concetti di **mantissa** e **esponente** senza però effettivamente spiegare di cosa si trattino:

- La **mantissa** non è altro che il numero binario codificato da un decimale che viene **normalizzato**: per essere normalizzata la mantissa dobbiamo spostare la virgola ottenendo 1 come unica cifra prima della virgola, che poi nel float finale non verrà contata; in seguito moltiplichiamo questo valore ottenuto con la potenza di 2 in base a quanti posti spostiamo la virgola. Esempio: $(113.25)_{10} = (1110001.01)_2$, spostiamo la virgola di 6 posti per normalizzare, di conseguenza moltiplichiamo il valore per 2^6: $1.\mathbf{11000101} \cdot 2^6$, la parte a seguire la virgola (ricordiamo che l'1 prima della virgola non si conta), sarà la mantissa

- L'**esponente** che otteniamo moltiplicando la mantissa deve essere trasformato però in binario, e per questo utilizziamo con lo standard IEEE 754 la codifica secondo il quale il valore dell'esponente **E** effettivamente rappresentato è: $E = e + k$ dove **e** rappresenta l'esponente trovato dal calcolo della mantissa; e **k** è un valore chiamato **bias** che nella codifica a 32 bit è $k = 127$, poichè avendo 8 bit a disposizione per la rappresentazione abbiamo $2^8 = 256$ combinazioni possibili e, dato che i valori 0 e 255 vengono utilizzati per usi speciali (che specificheremo dopo), shiftando i valori abbiamo che $-126 \leq e \leq 127$ e di conseguenza $1 \leq E \leq 254$. Riprendendo l'esempio di prima trovando 2^6 abbiamo che $E = 6 + 127 = (133)_{10} = (10100001)_2$.

Finendo l'esempio iniziato in cui volevamo codificare $(113.25)_10$:

- Sappiamo che il segno avrà bit **0**, poichè positivo

- Abbiamo codificato in valore assoluto di $(113.25)_10 = (1110001.01)_2$ e abbiamo trovato la mantissa normalizzando questo valore assoluto $1.\mathbf{11000101} \cdot 2^6$

- Abbiamo codificato l'esponente secondo lo standard IEEE 754: $E = e + k = 6 + 127 = (133)_{10} = (\mathbf{10100001})_\mathbf{2}$

- Il **numero finale** sarà quindi: 0|10100001|11000101000000000000000, in cui il primo bit è il segno, il secondo blocco l'esponente e il terzo la mantissa che continua fino a 23 bit con gli zeri

5.3.3 Approssimazione e precisione, overflow, underflow

Molto spesso capita di non poter rappresentare con un numero finito di cifre un numero oppure capita che il risultato finale di un operazione sia maggiore del valore massimo rappresentabile o minore. In questi casi si parla di **approssimazione, overflow e underflow**:

- L'**approssimazione** avviene quando non si può rappresentare un valore con un x**numero finito di cifre**, in questo caso il calcolatore troncherà i bit meno significativi a destra. In questo caso ci serve sapere la **precisione** p di un formato, ovvero la quantità massima di cifre significative rappresentabili dopo la virgola

- L'**overflow** si verifica quando il risultato di un operazione, che sia un intero positivo o negativo, è **maggiore** del massimo valore rappresentabile

- L'**underflow** si verifica quando il risultato di un operazione, che sia un intero positivo o negativo, è **minore** del minimo valore rappresentabile

5.3.4 Valori speciali

Abbiamo detto precedentemente che alcuni bit erano riservati per valori speciali:

- $\pm$ **inf**: $\pm\infty$;

- **NaN**: forme indeterminate $0/0$, ∞/∞...

- $\pm$**0**

5.4 Tipi in c++

C++ è un linguaggio fortemente tipizzato, ovvero quando inizializziamo una variabile dobbiamo sempre specificare il **tipo**. Questo ci aiuta in un duplice modo: infatti i vari tipi differiscono tra loro per il range di valori rappresentabili che è molto utile sia per **ottimizzare** lo spazio di memoria utilizzato nel programma e non sprecarne; sia per non incappare in problemi di overflow e underflow. Qui di seguito troviamo una tabella dei principali tipi utilizzati per codificare numeri reali con i loro range, precisione e numero di byte utilizzati:

Tipo	Range	Precisione	Dimensione
int	$\pm$2.147.483.647	...	4 bytes
unsigned	$[0, 4.294.967.295]$	...	4 bytes
long	$\pm(2^{63}1)$	...	8 bytes
unsigned long	$[0, 2^{64}1]$	...	8 bytes
short	$\pm$32768	...	2 bytes
unsigned short	$[0, 65535]$	...	2 bytes
double	$\pm10^{308}$	15 cifre	8 bytes
float	$\pm10^{38}$	6 cifre	4 bytes

Table 5.1: Tipi in c++

Abbiamo anche altri tipi solitamente utilizzati per rappresentare caratteri che utilizzano un numero di byte minore rispetto a quelli visti in precedenza:

Tipo	Range	Dimensione
char	$[-128, +127]$	1 byte
unsigned char	$[0, 255]$	1 byte
bool	$\{true, false\}$	1 byte
void	#	1 byte

Table 5.2: tipi di piccole dimensioni

5.4.1 Operatore sizeof() e librerie float.h, limits.h

In c++ abbiamo l'operatore **sizeof()** che restituisce il numero di byte utilizzati da un tipo e abbiamo anche le librerie **float.h** e **limits.h** che restituiscono rispettivamente il più piccolo e il più grande float rappresentabile e il più piccolo e il più grande intero rappresentabile.

5.5 Operatori aritmetici, funzioni matematiche e conversioni

5.5.1 Operatori aritmetici

Vediamo gli operatori aritmetici che possiamo utilizzare in c++ in questa tabella:

Operatore	**Num.argomenti**	**Significato**
+	2	Somma
-	2	Differenza
*	2	Prodotto
/	2	Divisione
%	2	Modulo (resto della divisione)
++	1	Incremento
–	1	Decremento

Table 5.3: Operatori aritmetici

5.5.2 Funzioni matematiche

Per usare diverse funzioni matematiche tra cui seno, coseno, radice quadrata, valore assoluto, logaritmo.. ci avvialiamo dell'uso della libreria <**math.h**>, oppure <**cmath**>, che dovremo includere nel nostro programma con:

```
#include <math.h>
```

5.5.3 Conversioni di tipo

Dobbiamo stare attenti quando convertiamo da un tipo a un altro, poichè in base alla dimensione dei tipi abbiamo bisogno di una **conversione implicita**

o **esplicita**:

- **Conversione implicita**: se passiamo da un tipo che contiene più informazioni a uno che ne contiene meno; il calcolatore dovrebbe attuare una conversione. Esempio:

```
// primo programma
double d = 2.8965;
int a = d; // perdiamo la parte decimale

// secondo programma
double d = 2.8965;
int a = d + 0.5; // arrotondiamo all'intero piu vicino
```

  Vediamo come in questi casi che, come nel primo caso, se assegniamo a un intero un double, che conteneva più informazioni, **perdiamo** in questo caso la parte decimale; se aggiungiamo 0.5 **arrotondiamo** all'intero più vicino . Il calcolatore ha dovuto fare una conversione implicita; dobbiamo stare attenti, inoltre, alle conversioni implicite anche nella divisione:

```
float result = 5/2; // =2, perdiamo la parte decimale
//////////
//////////
float result = 5.0/2; // =2.5, non perdiamo la parte decimale
```

  Notiamo che anche in questo caso l'operatore ha eseguito una conversione implicita, infatti nel primo caso se dividiamo due interi il calcolatore nonostante assegniamo questo risultato in una float ci darà un **intero**; mentre se noi specifichiamo esplicitamente la parte decimale il calcolatore eseguirà una conversione implicita restituendoci il risultato in **float**

- **Conversione esplicita** o **casting**: molto spesso abbiamo bisogno di convertire un risultato in un determinato tipo, e per questo utilizziamo un operatore che svolge questo ruolo, che è lo **static_cast**<type>(value); nel quale tra indichiamo il tipo tra gli indicatori maggiore e minore e il valore da convertire nelle parentesi tonde. Esempio:

```
// programma che arrotonda all'intero piu vicino

double d= 10.73;
double result = static_cast<int>(10 * d + 0.5);
```

  In questo caso nella prima istruzione abbiamo assegnato un valore frazionario al double e nell'istruzione dopo abbiamo fatto il **casting** a intero di un'espressione, che moltiplicava il valore per 10 e lo arrotondava all'intero più vicino (+0.5), ma assegnando questo intero al double il risultato che otterremo sarà un double con parte decimale 0.

Capitolo 6

Stringhe e IO di base in C++

Per utilizzare l'IO in C++ utilizziamo la libreria standard <**iostream**>, che proprio si basa sugli **stream** o flussi: questi flussi non sono altro che i canali di comunicazione con le periferiche di Input/Output, attraverso il quale quindi scambiamo dati, inviamo input e riceviamo output. Possiamo utilizzare stream standard in c++:

- **std::cout**: standard di **output**, generalmente associato al video
- **std::cin**: standard di **input**, generalmente associato alla tastiera
- **std::cerr**: standard di **output**, generalmente associato al video (segnale di error)

6.1 Standard output/input

6.1.1 Stampare un messaggio con lo standard output

Vediamo effettivamente con delle righe di codice come stampiamo un messaggio in output:

```
#include <iostream>
 using namespace std;
 cout << "Insert a number between 1 and 10:" << endl;
```

Analizziamo il codice:

- Abbiamo incluso la libreria <iostream>;
- Per evitare di specificare il namespace in cout con std::cout, utilizziamo per comodità la clausola "using namespace std", che definisce classi,funzioni e costanti al'interno di std;

- "<<", negli standard output possiamo utilizzare solo l'**operatore di inserimento** che inserisce la parte alla sua destra nello standard output
- **endl**, è un *manipolatore* che ci permette di andare a capo

6.1.2 Ricevere dati dallo standard input

Vediamo come funziona la ricezione di dati con lo standard input:

```
#include <iostream>
 using namespace std;
 int x;
 cin >> x;
```

In questo caso a differenza dello standard output, utilizziamo l'**operatore di estrazione** ">>" che appunto estrae il dato ricevuto in input e lo inserisce nella variabile alla sua destra, in questo caso x.

6.1.3 Formattare l'output

Per **formattare** l'output e renderlo il più leggibile possibile utilizziamo i cosiddetti **manipolatori** includendo la libreria <iomanip>. Esistono diversi manipolatori:

- **setprecision(n)**, questo manipolatore controlla il numero di cifre n totale da stampare con uno standard output
- **fixed**, questo manipolatore ci permette, se usato prima di setprecision(n), il numero di cifre decimale da stampare con lo standard output = n. Se non settato setprecision stamperà il numero totale di cifre n;
- **setw(n)**, questo manipolatore ci permette di controllare il numero totale dei caratteri n da stampare con lo standard output
- **scientific**, questo manipolatore ci permette di stampare i floating point in notazione scientifica

6.2 Stringhe e oggetto string

Una **stringa** è una **sequenza di caratteri** rappresentata coi doppi apici:

```
cout << "hello world"<< endl;
```

In C++ possiamo utilizzare inoltre anche la **classe string** se includiamo la libreria <string>, e ovviamente possiamo inzializzare una stringa come oggetto:

```
#include <string>
string name="Pippo";
string your name;
cout << "My name is " << name << endl;
cout << "Please write your name: " << endl;
cin >> your name;
cout << "Your name is" << your name << endl;
```

NB: Lo spazio " " è contato come carattere.

6.2.1 Concatenamento di stringhe

Abbiamo inoltre la possibilità di **concatenare** stringhe con l'operatore "+":

```
#include <string> // Header necessario!
string name="Pippo";
string your name="Marco";
string all names = name + "and" + your name; cout << "My name and your name is " << \
all names << endl;
```

NB: Non possiamo concatenare solo letterali (Esempio: "pippo" + "marco")

6.2.2 Lunghezza e indicizzazione di una stringa

Avendo la possibilità di utilizzare la stringa come oggetto possiamo utilizzare il metodo **.length()** per determinare la lunghezza in caratteri di una stringa:

```
 #include <string> // Header necessario!
  string name="Pippo";
cout << "The length of the my name is " << \
name.length() << " chars!";
```

Abbiamo inoltre la possibilità di indicizzare una stringa:

```
string name="Pippo";
int i = 3;
cout << "The char number " << (i+1) << "\ of my name is " << name[i];
//Il primo indice 'e zero!!
```

6.2.3 Sottostringhe

Esiste una funzione **substr(x,y)** che estrae una **sottostringa** restituendo una stringa: il parametro x indica l'**indice** del primo carattere della sottostringa(che viene contato partendo da 0) e il parametro y indica la **lunghezza** della sottostringa da estrarre. Se non viene specificato quest'ultimo parametro saranno considerati i rimanenti caratteri della stringa di partenza. Esempio:

```
string name="Pippo";
cout << "The name without the first letter: " \
<< name << " <<name.substr(1,name.length()-2);
```

6.3 stringstream

Qualora volessimo immagazzinare dati (stringhe o numeri) in **buffer**, ovvero conservarli per riutilizzarli, possiamo utilizzare la classe **stringstream**, che dobbiamo includere con la libreria <sstream>. Questa classe può essere utilizzata solamente con l'operatore di inserimento (<<), oppure con l'operatore di estrazione (>>). Esempio:

```
#include <sstream>
using namespace std ;
stringstream ss ;
ss<<     HelloWorld     <<endl;
//oppure
ss.str(  Hello    World!    );
///////
//inoltre ..
cout << ss.str (); // estrazione del contenuto
```

In questo piccolo programma abbiamo visto come usare stringstream per stampare un letterale e usarla come classe ss.str("Hello world!").

Questa classe è molto utile per la **conversione di un numero nella sua sequenza di caratteri** e per la conversione di **stringhe in numeri**:

- **Conversione di un numero nella sua sequenza di caratteri**:

  ```
  #include <sstream>
  using namespace std ;
  stringstream ss ;
  double g = 12345.6789;
  ss << g << endl ; //inserimento dei dati nel buffer
  string my_number;
  my_number = ss.str (); // stringa equivalente
  ```

  In questo caso abbiamo inserito con "<<", una var.double nel buffer ss; poi abbiamo inizializzato una var.string al quale abbiamo assegnato il valore presente all'interno del buffer; e infine questo valore per essere assegnato a una var.string verrà convertito in una sequenza di caratteri

- **Conversione di stringhe in numeri**:

  ```
  #include <sstream>
  using namespace std ;
  stringstream ss ;
  double y;
  ss <<   123456  .893    ; //inserimento in ss
  ss >> y; //estrazione in y
  cout <<  y =    << y << endl;
  ```

  In questo piccolo programma abbiamo inserito con l'operatore "<<" nel buffer la stringa "123456.893"; lo abbiamo estratto con ">>" nella variabile double y; e infine abbiamo stampato il valore numerico presente in y

Capitolo 7

CONTROLLI CONDIZIONALI IN C++

Nel capitolo riguardante gli algoritmi e i diagrammi di flusso abbiamo discusso del fatto che esistano delle strutture che ci permettono di controllare il flusso di esecuzione di un programma, cioè i **controlli condizionali**

7.1 Il costrutto *if/else* in C++

In C++ per condizionare il flusso di esecuzione del codice utilizziamo la struttura **if/else** con il quale, assunto un **predicato** che può restituire un valore di verità **true** o **false**, il flusso prenderà una strada diversa a seconda del valore restituito dal predicato. Nel caso in cui il valore di verità restituito sia true, ovvero la condizione imposta si verifica vera, allora si eseguirà il blocco di istruzioni presente nell'**if**; se il valore di verità restituito è false, ovvero la condizione imposta si verifica falsa, allora l'esecuzione passerà direttamente al blocco di istruzioni dell'**else**.

Esempio Vediamo un pezzo di codice estrapolato dove è implementato il controllo condizionale if/else:

```
int x;
cout <<    Inserire    un numero < 10 oppure     << endl;
cin >> x;
if (x>=10)
cout <<   Numero   inserito non valido !   ;
else
cout <<   Il   numero inserito  e       << x;
```

In pratica diciamo al compilatore che se il valore dato in input è maggiore o uguale a 10, stampa la stringa "Numero inserito non valido !"; altrimenti, se il numero inserito è minore di 10, stampa "Il numero inserito e' x"

In questo caso, visto che abbiamo scritto una sola istruzione, non abbiamo messo le parentesi graffe; ma sarebbe buona norma metterle

7.1.1 Operatori relazionali

In C++ abbiamo diversi operatori che ci permettono di effettuare dei **confronti** e sono quelli rappresentati nella seguente tabella:

C++	Descrizione
>	Maggiore di
>=	Maggiore o uguale di
<	Minore di
<=	Maggiore o uguale di
==	Uguale
!=	Diverso

Table 7.1: OPERATORI RELAZIONALI

NB Non si utilizza il simbolo "=" poichè in C++ rappresenta l'istruzione di assegnamento, mentre, "==" rappresenta il confronto sull'uguaglianza tra due valori

7.1.2 *if* annidati

Ovviamente il costrutto dell'if/else si configura come una struttura ad *albero*; cioè noi possiamo avere un costrutto più complesso con degli **if annidati**. Esempio:

```
int x;
cout <<    Inserire    un numero positivo < 10, ma che \
non sia 5!    << endl;
cin >> x;
if (x>=10)
cout <<   Numero   inserito maggiore di 10!   ;
else if(x==5)
cout <<   Hai   inserito proprio il 5!    << x;
else if(x<=0)
cout <<   Il   numero e   negativo!    << x;
else //eseguito se prec. condizioni non verificate
cout<< Il   numero inserito e :   <<x<<endl;
```

In questo pezzo di codice possiamo considerare i vari if annidati come dei rami del blocco principale. Infatti partendo dall'if principale: se la condizione ($x >= 10$) è **vera** esegui il blocco if, altrimenti (else, condizione $x >= 10$ falsa), se ($x == 5$) esegui il nuovo il blocco if; altrimenti (else, condizione $x >= 10 \wedge x == 5$ falsa) se $x <= 0$ esegui il nuovo blocco if; altrimenti (condizione $x >= 10 \wedge x == 5 \wedge x <= 0$ falsa) e quindi nessuna delle condizioni precedenti si è verificata esegui l'ultimo blocco else. Per comprendere al meglio l'esecuzione dei singoli blocchi conviene sempre **indentare** e usare le **graffe** per non incorrere in errori **sintattici** o peggio, **semantici**

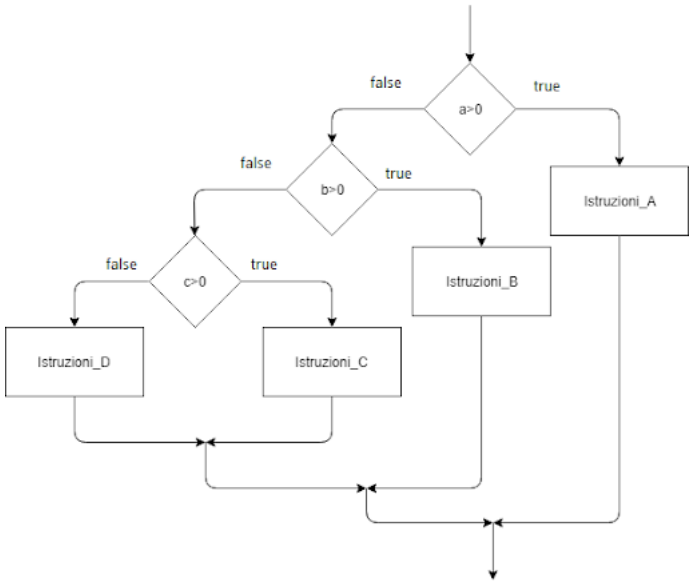

Figure 7.1: Struttura di if annidati

7.1.3 Errori sintattici/semantici

E' proprio in questi casi che è importante comprendere ciò che si può intendere per sintatticamente o semanticamente sbagliato; poichè ciò che è **sintatticamente sbagliato** viene evidenziato dal **compilatore** (per questo è buona norma leggere gli errori descritti dal compilatore), mentre se facciamo un errore di **semantica** non rispettando le specifiche che dovrebbe restituire il programma, allora non ci aiuta il compilatore, poichè potrebbe compilare lo stesso, ma dobbiamo capire noi dove sta l'errore. Esempio:

```
if (x>=10){
result = (alpha      x) / 10;
cout <<   Numero   inserito maggiore di 10!    ;
}
else; {
result = alpha      x;
cout <<    Inserito    numero valido!    ;
}
```

Questo è proprio il caso di un codice **semanticamente scorretto**; infatti se noi facciamo compilare questo codice al compilatore, questo non ci darà errore (sintatticamente corretto), ma succede che eseguirà il blocco else incodizionatamente, ovvero anche quando esegue il blocco if poichè abbiamo posto il ";" dopo l'else.

7.2 Operatore condizionale

Esiste un'**operatore condizionale** che ci permette di eseguire la stessa funzione di un blocco if/else:

- **Sintassi**: (**COND1** ? **EXPR1** : **EXPR2**)

- **Semantica**: **Se** il valore di verità di COND1 è true;
  allora valuta EXPR1;
  Altrimenti valuta EXPR2

Esempio:

```
cout <<      Max (x,y) =      << ( x > y ? x : y ) << endl;
```

equivalente a..

```
 if (x>y)
cout <<   Max (x,y)=     << x < endl;
else
cout <<   Max (x,y)=     << y < endl;
```

7.3 Confronti lessicografici

Qualora, con gli operatori relazionali, volessimo confrontare stringhe o caratteri, faremmo un **confronto lessicografico**. Esempio:

```
 char a =     a    ;
 char b =     b    ;
cout << (a<b ?   a   <   b    :   b   >   a   ) << endl;
```

7.3.1 Come avviene il confronto tra due stringhe o caratteri

Il confronto tra due stringhe o caratteri avviene prendendo entrambe le stringhe o caratteri e si confronta ogni carattere delle due stringhe con **indice uguale** (S1[1] vs S2[1], S1: stringa 1, S2: stringa 2). Se il carattere contenuto è uguale nelle due stringhe si procede con il carattere successivo; altrimenti si vede qual'è il minore o maggiore tra i due.

Caratteri in codifica ASCII Abbiamo detto che nel confronto fra stringhe e caratteri, ogni carattere di una stringa viene confrontato con un altro di indice uguale dell'altra stringa per vedere se questo sia maggiore o minore: questo confronto avviene in base al **valore con il quale il carattere viene codificato** nella codifica ASCII; se il valore che codifica un carattere a è minore del valore che codifica un carattere b, allora $a < b$; altrimenti $a > b$. Esempio: $'\mathbf{A}' <' \mathbf{a}'$? Nella codifica ASCII 'A': 65; 'a':97 quindi il confronto è vero ($'A' <' a'$)

7.4 Costrutto switch

Molto spesso ci capita di dover annidare diversi if e sarebbe molto poco leggibile e molto tedioso per un programmatore scrivere vari *else if*; per questo esiste il costrutto **switch** che ci permette di annidare diversi if in maniera molto più leggibile e veloce. L'unica cosa importante da comprendere è che esso vale solo per le espressioni dove si verifica l'**uguaglianza tra costanti**, quindi **non** possiamo inserire **etichette, variabili o intervalli** a differenza dell'if.

Esempio Vediamo un'esempio di come si costruisce ed utilizza il costrutto switch:

```
 switch ( digit ) {
case 1:
digit name =     one    ; break ;
case 2:
digit name =     two    ; break ;
```

```
// altri case ...
default :
digit name = NO DIGIT ; break ;
}
```

La **sintassi** del costrutto switch è molto semplice:

- **switch**(variabile da controllare);
- **case** 1 (variabile == 1?): ...; **break**;, **case** 2 (variabile == 2?): ....; **break**;, ...;
- **default**: ....

La **semantica** è molto semplice:

- Abbiamo una **variabile da controllare** nelle parentesi dello **switch**;
- Questa variabile viene confrontata con un letterale (valore, stringa, carattere..) nei vari **case**; quando si verifica la condizione di un case si esegue quel blocco e in caso di **break** si esce dallo switch, se il break vengono eseguite le istruzioni dei vari case fino a quando è presente un break;
- Se non si verifica nessuna condizione dei case si può utilizzare **default** che esegue un blocco di istruzioni nel caso in cui appunto non sia stato eseguito nessun blocco case

7.5 Hand tracing

L'*hand tracing* è una tecnica che ci permette di simulare manualmente l'esecuzione di un programma. Si tratta di una **tabella di traccia** costruita nel seguente modo:

- Nella prima riga si scrivono i **nomi delle variabili** interessate
- Nelle righe successive si tiene traccia (**tracing**) dei valori di queste variabili
- Si crea una riga ogni volta che una variabile **cambia valore**

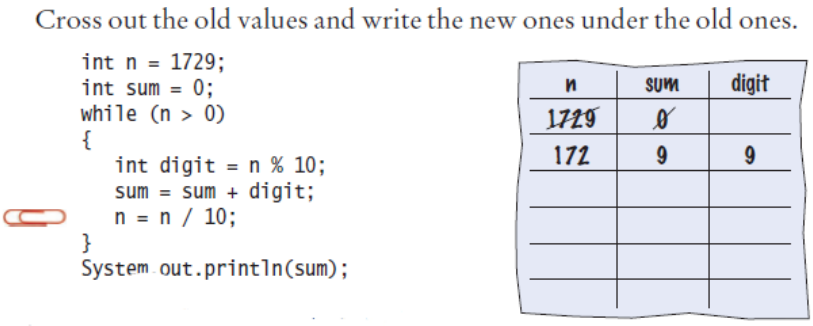

Figure 7.2: Hand tracing

7.6 Operatori logici

Gli **operatori logici** permettono di combinare espressioni con risultato **booleano**, ovvero espressioni che possono assumere valore **true** o **false**. Nell'algebra booleana gli operatori logici sono:

- **AND**: in C++ è denotato dal simbolo"&&". Esso restituisce valore **true** solo nel caso in cui entrambe le variabili iniziali restituiscono valore true; altrimenti restituirà false;

- **OR**: in C++ è denotato dal simbolo "||". Esso restituisce valore **false** solo nel caso in cui entrambe le variabili iniziali restituiscono valore false; altrimenti restituirà true;
- **NOT**: in C++ è denotato dal simbolo "!". Esso restituisce il **valore di verità opposto** alla variabile in entrata. In questo caso si tratta di un **operatore unario**, la variabile iniziale è solo una

A	B	A && B	A || B
true	true	true	true
false	true	false	true
true	false	false	true
false	false	false	false

A	! A
false	true
true	false

Figure 7.3: Tabelle di verità AND, OR, NOT

Esempio di utilizzo di operatori logici Vediamo ad esempio in un controllo if/else, l'utilizzo di AND:

```
 alpha =1.0;
 if (a!=2){ // a non deve essere ne 2 ne 3!!
    if (a!=3)
            alpha = 0.5;
}
else
            alpha =1.0;

// forma migliore

if (a!=2 && a!=3) // equivalente
        alpha = 0.5;
else
        alpha =1.0;
```

7.6.1 La valutazione a corto circuito

La **valutazione a corto circuito** dice che se ho un **insieme di AND** e il primo valore restituisce un valore false; la valutazione dell'espressione termina poichè il risultato è già ben determinato (false). Lo stesso meccanismo vale per un **insieme di OR**: infatti se il primo valore in un OR restituisce true, la valutazione dell'espressione termina poichè il risultato è già ben determinato (true).

Capitolo 8

COSTRUTTI DI CICLO IN C++

Affrontiamo in questo capitolo come possiamo scrivere dei cicli, ovvero blocchi di istruzioni che si ripetono in base ad una condizione, in C++.

8.1 Il costrutto while in C++

La **sintassi** del while in C++ è molto semplice:

```
while(condition)
    statement
```

Abbiamo quindi una condizione, cioè un'espressione che può restituire solo valori booleani **true** o **false**, e uno statement, ovvero il blocco di istruzioni del while. In questo caso il compilatore valuterà il valore di verità vero o falso, e, se la condizione restituisce true, eseguirà il blocco di istruzioni; in seguito per i vari cicli rivaluterà il valore di verità della condizione, e, se risulta sempre vera entreremo in un **loop infinito**. E' fondamentale quindi avere all'interno del blocco di istruzioni del ciclo un istruzione che faccia si che la condizione vari, fino a farla diventare falsa, così da avere una **terminazione del ciclo**.

8.1.1 Esempi di cicli while in c++

Esempio 1 Vediamo il primo esempio di un ciclo while in cui il **numero di iterazioni** non è conosciuto a priori:

```
const double TARGET = 1800.0;
const double TASSO INTERESSE = 0 . 1 ;
double capitale =1000.0;
int anno=0;
          while(capitale<TARGET){ // condizione
             capitale+=capitale*TASSO INTERESSE;
             anno++;
    }
```

In questo esempio vediamo un ciclo in cui **non sappiamo il numero di iterazioni** a priori, poichè non sappiamo quante volte dovremo eseguire il ciclo affinche la condizione restituisca valore false. Di fatto questo è l'uso più tipico del while: lo scopo quindi di questo ciclo è quello di calcolare il tasso di interesse annuale sul capitale; l'istruzione che fa variare

la valutazione della condizione, in questo caso, è l'aggiornamento della variabile "capitale" ad ogni ciclo. A un certo punto dell'esecuzione del ciclo avremo quindi $capitale >= target$ e usciremo dal ciclo

Esempio 2 Vediamo il secondo esempio di un ciclo while in cui il **numero di iterazioni** è conosciuto a priori:

```
const double TASSO INTERESSE = 0 . 1 ;
const int N=5;
double capitale =1000.0;
int anno=0; // inizializzazione
while(anno<N){ // condizione
            capitale+=capitale TASSO INTERESSE;
            anno++; // incremento/aggiornamento
```

In tale caso sappiamo che il numero di iterazioni è N, ovvero 5, poichè avendo inizializzato anno=0, sappiamo che l'ultimo blocco del ciclo verrà eseguito quando N sarà 4, cioè 5 volte. Lo scopo del ciclo è come quello del programma precedente però, in questo caso, non abbiamo la costante TARGET che confrontiamo con la variabile capitale, ma l'aggiornamento della valutazione della condizione avviene con l'incremento di anno "anno++". Di conseguenza usciremo dal ciclo quando $anno = N$.

Esempio 3 Possiamo inserire l'incrementatore nella condizione; l'importante e capire che con il post-incremento prima valutiamo la variabile e poi la incrementiamo; con il pre-incremento prima incrementiamo la variabile poi viene valutata la condizione:

```
while (anno++ < N) //valutiamo la condizione e poi incrementiamo

while (++anno < N) // incrementiamo la variabile e poi valutiamo la condizione
```

8.2 Costrutto for

Il **costrutto for** è un costrutto di ciclo che si utilizza solitamente quando sappiamo **il numero di iterazioni a priori**, mentre con il while non sempre abbiamo bisogno di sapere il numero di iterazioni. Vediamo quindi la sintassi:

```
for (inizialization; condition; update )
        statement
```

Vediamo quindi tutte le componenti del for:

- **inizialization**: viene inizializzata una variabile una sola volta all'inizio del ciclo e viene utilizzata solamente per il ciclo, quindi non occupa locazioni di memoria nel programma;
- **condition**: come per il ciclo while viene valutata una condizione ad ogni iterazione; se questa restituisce valore true allora si fa un'iterazione, quando restituisce false si esce dal ciclo;
- **update**: viene aggiornata una variabile alla fine di ogni iterazione

Se quindi volessimo fare 5 iterazioni, utilizzando il ciclo for avremmo:

```
for (i=0; i<5; i++)
```

8.2.1 for vs while

Vediamo alcuni esempi in cui può essere più conveniente utilizzare il ciclo for o while:

Esempio 1 A volte utilizzare il for è più conveniente anche per una questione di leggibilità del codice:

```
int anno=0;
while(anno<N){
    capitale+=capitale*TASSO_INTERESSE;
    anno++;
}

// codice pi  leggibile
 for(int anno=0; anno<N; anno++)
        capitale+=capitale*TASSO_INTERESSE;
```

Esempio 2 La variabile che inizializziamo in un ciclo for può essere anche utilizzata in altri cicli for con parametri diversi senza dover preoccuparci che sia stata già dichiarata in precedenza:

```
for(int anno=0; anno<N; anno++)
        capitale+=capitale*TASSO_INTERESSE;

//il tasso di interesse e cambiato!
for(int anno=0; anno<N; anno++)
        capitale+=capitale*TASSO_INTERESSE_2;
```

Esempio 3 Come abbiamo detto in precedenza il for non è adatto a cicli in cui non sappiamo il numero di iterazioni, come vediamo in questo esempio:

```
double capitale =1000.0;
while(capitale<TARGET){
        capitale+=capitale*TASSO_INTERESSE;
}

// codice poco leggibile
 double capitale =1000.0;
 for (; capitale<TARGET;){
        capitale+=capitale*TASSO_INTERESSE;
}
```

Esempio 4 Nel for possiamo avere anche espressioni non correlate fra loro:

```
double capitale =1000.0;
int anno=0;
while ( capitale<TARGET ) {
      capitale+=capitale*TASSO_INTERESSE;
      anno++;
}

// espressioni non correlate col for
for(anno=0; capitale<TARGET; anno++)
        capitale+=capitale*TASSO_INTERESSE;
```

8.3 Costrutto do-while

Il costrutto do-while è identico al funzionamento del while, ma la caratteristica fondamentale del do-while è che la **prima istruzione** viene **eseguita incondizionatamente**, ovvero questo ciclo fa almeno un iterazione; in seguito viene valutata la condizione che in base al valore di verità farà altre iterazioni o meno. La sintassi del do-while è la seguente:

```
do{
    statement
}while (condition)
```

8.3.1 Uso del do-while

Vediamo adesso un esempio di uso tipico del do-while:

```
int input;
do{
cout <<    Inserire    un numero minore di 100 o maggiore o uguale a 50:     << endl ;
cin >> input;
}while(input >= 100 || input <50);
```

Quindi, ad esempio, qui sappiamo che la prima istruzione eseguita incodizionatamente, quindi chiediamo all'utente di darci in input un numero minore di 100 o maggiore o uguale a 50; in seguito dopo aver valutato la condizione, che ci indica che l'utente ha scritto un numero non valido, se l'utente ci invia in input un numero non valido allora con il do-while gli richiederemo di scrivere il numero

8.4 Contare le iterazioni di un ciclo

Per sapere quante iterazioni farà il nostro ciclo, soprattutto quando utilizziamo il for, dati a e b numeri interi con $a < b$, abbiamo:

- **b − a iterazioni** quando il for è costruito in tale modo:

  ```
  for (i=a; i<b; i++)
  ```

- **b − a + 1 iterazioni** quando il for è costruito in tale modo:

  ```
  for (i=a; i<= b ; i++)
  ```

8.5 *break* e *continue*

Abbiamo già visto l'utilizzo del **break** all'interno dello switch. In realtà è possibile utilizzarlo anche nei cicli while, for e do-while e indica di **interrompere il flusso di esecuzione del ciclo** nel punto esatto in cui è posta l'istruzione. Non è consigliato però utilizzarla spesso poichè potrebbe peggiorare la comprensibilità del codice e inoltre esiste sempre una forma di controllo iterativo senza break. Vediamo un esempio:

```
double capitale =1000.0;
int anno=0;
while ( true ){
        capitale+=capitale TASSO INTERESSE; anno++;
if ( capitale >= TARGET )
        break ;
}
```

L'istruzione **continue** invece salta le istruzioni restanti dell'iterazione passando alla prossima iterazione. Anche in questo caso non è consigliato utilizzarla spesso poichè potrebbe peggiorare la comprensibilità del codice e inoltre esiste sempre una forma di controllo iterativo senza continue. Vediamo un esempio:

```
const int N=20;

//stampa solo i numeri dispari
for(int i=0; i<N; i++){
        if ( i%2==0)
            continue ;
        cout << i << endl;
}
```

Capitolo 9

GESTIONE ERRORI DI IO IN C++

Molto spesso se chiediamo qualcosa in input all'utente, non sempre questo restituirà ciò che noi chiedevamo commettendo un errore nella sequenza dei caratteri. Per questo esistono alcuni metodi che ci permettono di **notificare all'utente un errore**. Vediamo un esempio di errore di IO:

```
float x; //floating point to collect the user input
std :: cout <<    Insert    any number:      << endl;
std :: cin >> x ;
```

Se ad esempio l'utente dovesse inserire in input "pippo", nella variabile x non sarà copiato alcun valore poichè noi avevamo dichiarato quella variabile come una float. Di conseguenza avremo un errore di IO. Vediamo adesso i metodi solitamente utilizzati per verificare un errore di IO

9.1 Metodo fail() e operatore !

Abbiamo due modi essenzialmente per notificare un errore:

- Il primo metodo che utilizziamo è il **fail()**:

  ```
  float x;
  std :: cin >> x;
  if(cin.fail()){ //IO error
        cerr <<    Inserito    input non valido!    << endl;
        return   1  ;
  }
  ```

  In questo caso creiamo un if nel quale se la condizione, ovvero "cin.fail()" restituisce il valore vero, allora l'input non è valido: notifichiamo all'utente che non è valido questo input con un cerr(standard output di errore). Con la return, che essenzialmente codifica ciò che è successo all'interno del programma, tornare -1 possiamo codificarlo come standard di errore; una volta ritornato un intero con la return il programma **terminera**.

- Il secondo metodo che possiamo utilizzare è con l'**operatore "!"**:

```
float x;
if(!(std::cin >> x)){ //IO error
        std::cerr <<    Inserito    input non valido!    << endl;
        return   1  ;
}
```

In questo caso se la condizione restituisce valore vero, come con il metodo fail(), l'input non è valido

Ma se volessimo utilizzare il metodo fail(); quando resituisce true? Dipende da alcuni "*flags*" che caratterizzano lo stato dello stream, ovvero individuano eventuali errori di IO:

- **eofbit** (End Of File)
- **failbit** (Errore di IO: formattazione o estrazione)
- **badbit** (Altri errori)
- **goodbit** (nessun errore)

In particolare per restituire true; il metodo fail(), ha bisogno della condizione:

```
(failbit==true || badbit==true)
```

Quindi restituirà vero quando almeno uno dei due flag badbit o failbit restituisce vero

9.2 Metodi clear() e ignore()

Il metodo **clear()** è fondamentale per **resettare** a 0 tutti i flag di errore dello stream, così da poter fare ulteriori operazioni di IO. Ad esempio:

```
if(cin.fail()){ //IO error!
 // . . .
        cin .clear(); //RESET IO flags
 // . . .
}
```

Il metodo **ignore()**, invece, permette di **scartare i caratteri rimasti nello stream**:

- A seguito di un **errore**, ad esempio utente ha inserito "pippo" anziche un numero;
- perchè l'utente ha inserito **più stringhe** separate da spazi.

Capitolo 10

FILE IO IN C++

In C++ esistono alcune librerie che ci permettono di effettuare operazioni di **lettura**, **scrittura** e **lettura e scrittura** contemporaneamente su un **file**. La libreria da includere è *fstream* e le classi che possiamo utilizzare sono le seguenti:

- **std::ofstream**: scrittura;
- **std::ifstream**: lettura;
- **std::fstream**: lettura e scrittura.

10.1 Apertura di un file

Vediamo adesso i vari metodi utilizzati per **aprire un file** in lettura, scrittura o lettura/scrittura.

10.1.1 Aprire un file in lettura con *ifstream*

Abbiamo due modi di aprire un file in lettura con *ifstream*:

1. Metodo che utilizza il **costruttore**:

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

ifstream myfile( test .txt );
if (! myfile .is_open())
    cerr << Errore apertura file ! << endl ;
```

   In questo caso abbiamo inizializzato la classe ifstream con una variabile "myfile" che conteneva tra le parentesi tonde il nome del file. Per controllare se ci fosse un errore di apertura del file, abbiamo utilizzato il metodo $is_open()che, serestituisceunvalorefalso, inquestocasonegatodalsimbolo"!"(quindiqu$

2. Metodo che utilizza *open()*:

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

ifstream myfile (); //NB: costr . default
myfile.open( test . txt ); //NB: metodo open
```

```
if (! myfile . is_open ())
        cerr <<  Errore  apertura file !  << endl ;
```

In tal caso, come visto nell'esempio precedente, abbiamo inizializzato in ifstream una variabile "myfile" che però non abbiamo utilizzato con il costruttore: infatti abbiamo aperto il file con il metodo open(), e, tra parentesi tonde abbiamo specificato il nome del file. In caso di errore di apertura del file l'utente verrà notificato attraverso il metodo is_open.

10.1.2 Aprire un file in scrittura con *ofstream*

Aprire un file in **scrittura** è molto semplice e si assomiglia molto con la struttura di un programma per aprire un file in lettura. L'unica differenza sostanziale è che dobbiamo utilizzare la classe *ofstream*:

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

ofstream myfile (); //NB: costr . default
myfile.open( test . txt ); //NB: metodo open
if (! myfile . is_open ())
        cerr <<  Errore  apertura file !  << endl ;
```

Anche in questo caso possiamo utilizzare allo stesso modo sia il metodo del **costruttore**, oppure il metodo *open()*.

10.1.3 Aprire un file in lettura e/o scrittura con *fstream*

Aprire un file in lettura e/o scrittura con la classe **fstream** si differisce dall'apertura di un file con *ifstream* o **ofstream**, dal fatto che bisogna specificare se stiamo aprendo il file **solo in lettura**, **solo in scrittura** o **sia in lettura che in scrittura**:

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

fstream myfile (); //NB: costr . default
myfile.open( test . txt , fstream::out); //NB: metodo open
if (! myfile . is_open ())
        cerr <<  Errore  apertura file !  << endl ;
```

Vediamo ad esempio in questo programma che quando andiamo ad utilizzare il metodo open() non solo specifichiamo il nome del file, ma specifichiamo anche la **modalità di apertura del file** (**fstream::out**), cioè in **scrittura**. Se volessimo combinare **lettura e scrittura**, allora utilizzeremo l'**operatore bitwise** "—":

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

fstream myfile (); //NB: costr . default
myfile.open( test . txt , fstream::out|fstream::in); //NB: metodo open
if (! myfile . is_open ())
        cerr <<  Errore  apertura file !  << endl ;
```

10.2 Lettura e scrittura di caratteri da/su un file

Vediamo adesso come poter **leggere o scrivere un dato** su un file e controllare **eventuali errori di lettura o scrittura**

10.2.1 Scrittura di un file

La **scrittura di un file** e il **controllo dell'eventuale errore** può avvenire con metodi che noi conosciamo:

- Scriveremo con l'**operatore di inserimento** "<<";
- Controlleremo l'eventuale errore con il metodo *cin.fail()*, oppure con l'operatore "!".

Vediamo un esempio:

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

fstream myfile ( test . txt , fstream::out);
myfile << "pippo test";
if ( myfile .fail())
        cerr << Errore di scrittura nel file ! << endl ;

\\ oppure

fstream myfile ( test . txt , fstream::out);

if ( !(myfile << "pippo test"))
        cerr << Errore di scrittura nel file ! << endl ;
```

10.2.2 Lettura di un file

In questo caso, **lettura di un file** e il **controllo dell'eventuale errore** può avvenire sempre con metodi che noi conosciamo:

- Scriveremo con l'**operatore di estrazione** ">>";
- Controlleremo l'eventuale errore con il metodo *cin.fail()*, oppure con l'operatore "!".

Vediamo un esempio:

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

fstream myfile ( test . txt , fstream::in);
string s;
myfile >> s;
if ( myfile .fail())
        cerr << Errore di lettura nel file ! << endl ;

\\ oppure

fstream myfile ( test . txt , fstream::in);
string s;
if ( !(myfile >> s))
```

```
        cerr << Errore di lettura nel file ! << endl ;
```

10.3 Chiusura di un file

E' sempre buona norma **chiudere un file** dopo averlo utilizzato in operazioni di lettura/scrittura in modo tale da non creare conflitti nella compilazione del programma. Il metodo che utilizziamo è *close()*:

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

fstream myfile( test .txt , fstream::out);
if (!( myfile << test ))
        cerr << Error writing on file .. << endl;
myfile .close(); // metodo close()
```

Capitolo 11

GENERAZIONE DI NUMERI PSEUDO-CASUALI IN C++

Molto spesso quando si deve **simulare** un'attività del mondo reale bisogna creare algoritmi e programmi che facciano si che i dati in output siano più conformi possibili alla realtà. E per far sì che questo accada dobbiamo creare un **generatore di numeri pseudo-casuali** che dati dei **parametri in input** possa generare numeri pseudo-casuali che abbiano una certa sequenza (**seme**), oppure che abbiano una sequenza **randomica** che siano magari generati in un **intervallo definito**

11.1 Caratteristiche di un generatore di numeri pseudo-casuali

Un **generatore di numeri pseudo-casuali** dovrebbe rispettare le seguenti **caratteristiche**:

- **Random**: ogni generatore deve poter generare **sequenze randomiche di numeri** per scopi ben specifici;
- **Controllabile**: ogni generatore, allo stesso tempo, deve poter generare **sequenze controllate**, ovvero che hanno una certa logica (seme);
- **Portabile**: Ogni generatore dovrebbe essere portabile su architetture differenti;
- **Efficiente**: ogni generatore dovrebbe essere efficiente in termini di risorse per il calcolo

11.2 Generatori in C++

Il linguaggio C++ ci fornisce 2 librerie standard per la generazione di numeri casuali, ovvero **cstdlib** e **ctime**: la prima contiene le **funzioni srand() e rand()**, la seconda contiene la **funzione time()**. Vediamo un esempio di come vengono applicate tali funzioni così da capire come funzionano e la loro semantica:

```
#include <cstdlib > // per rand () ed srand ()
#include <ctime> // per la funzione time()
srand (111222333); // seme
//oppure
srand(time (0));

for(int i=0; i<1000; i++)
        cout << rand () << endl ;
```

Questo programma stamperà una **sequenza di 1000 numeri casuali**. Analizzando il codice vediamo come agiscono le varie funzioni:

- La funzione **srand() fissa la sequenza**: nel primo caso noi stiamo impostando tra le tonde una sequenza fissa per il quale ogni volta che rieseguiamo il programma avremo la stessa sequenza di valori casuali. In questo caso la sequenza passata tra parametri nella funzione srand() viene denominata come **seme**, o *seed*. Nel secondo caso, invece, inserendo tra le tonde la funzione **time()**, non avremo come prima una sequenza fissa di generazione, ma i numeri generati avranno una **sequenza random**.
- La funzione **rand()** invece **estrae un numero casuale compreso tra** $0 < x < RAND\_MAX$; dove "$RAND\_MAX$" indica il numero più alto che la funzione può estrarre. Quindi in questo caso avendo impostato un ciclo for che fa 1000 iterazioni; stamperemo 1000 numeri casuali o con una sequenza fissa (srand(11122233)) oppure con una sequenza randomica (srand(time(0)))

11.2.1 Generare numeri pseudo-casuali in un range

Oltre che generare numeri casuali compresi tra 0 e "$RAND\_MAX$"; può essere molto utile generare numeri compresi in un **range** $[a, b]$: sappiamo che il numero di elementi di un range, dati $a, b \in \mathbb{N} \wedge a < b$, è $(\mathbf{b - a + 1})$. Se quindi eseguiamo un'estrazione del tipo $rand()\%P$, abbiamo un numero r, ovvero il resto, che sarà $0 \leq r < P$. Adesso per trovare un $\mathbf{a \leq r \leq b}$, dovremmo **shiftare di a** $(b - a + 1)$:

```
unsigned int r = rand()%(b - a + 1) + a;
```

Ciò che otteniamo saranno numeri casuali assegnati in r, **compresi tra a e b**. Nella espressione precedente infatti:

- $rand()\%(b - a + 1)$ permette di generare numeri compresi tra 0 e b-a (estremi inclusi);
- Il termine $+a$ fa si che i numeri si trovino tra a e b (estremi inclusi).

Generare numeri pseudo-casuali in virgola-mobile in un range

Sappiamo che la funzione rand() estrae solo numeri interi; se invece volessimo generare numeri pseudo-casuali in **virgola mobile** dobbiamo **forzare una divisione in virgola mobile**:

```
double r = rand() / (RAND_MAX * 1.0);
//oppure
double r= rand()/ (double)RAND_MAX
```

In tal caso ciò che faremo è **generare numeri in virgola mobile compresi tra 0 e 1**

Capitolo 12

ARRAY IN C++

Come avevamo visto nel capitolo 4, gli **array** sono locazioni di memoria consecutive dello **stesso tipo**. In C++ la lettura/scrittura di un array avviene con nome dell'array e indice tra le parentesi quadre "$V[3]$". Inoltre il primo indice di un array è sempre 0, e l'ultimo $DIM-1$, dove "DIM" rappresenta la dimensione dell'array.

12.1 Dichiarazione, inizializzazione e assegnamento di un array in C++

Vediamo adesso come si dichara e inizializza un array in C++.

12.1.1 Dichiarazione di un array

Vediamo con un esempio i possibili modi di **dichiarazione di un array**:

```
#define DIM 10
const int dim = 10;
short mydim = 10;
int V1[10]; //10 elementi interi
float V2[DIM]; //10 elementi interi
double V3[dim]; //10 elementi
double long V4[mydim]; //10 elementi long


V1[7] = 4; //assegnamento di un valore a una locazione dell'array
V2[0] = 6.7;
```

Come possiamo vedere in questo esempio, possiamo dichiarare gli array con diversi tipi e utilizzando le costanti e indichiamo in ordine il **tipo dell'array**, il **nome dell'array** e tra parentesi quadre la **dimensione dell'array**. Per quanto riguarda l'assegnamento ad esempio quando diciamo $V1[7] = 4$ significa che stiamo allocando il valore 4 nell'array V1 nella locazione con indice 7, ovvero l'ottava partendo da 0.

12.1.2 Dichiarazione vs inizializzazione di un array

Ovviamente se volessimo richiamare un valore all'interno di un array senza averlo mai inizializzato ci sarà un errore poichè è come se stessimo richiamando una locazione di memoria vuota. Ad esempio:

```
#define DIM 10
int V1[DIM]; //10 elementi interi

//Quanto vale V1[5] ?? ?
cout <<    Elemento    di V1 con indice 5:      << V[5] << endl ;
```

Per evitare questi errori dobbiamo **inizializzare** in qualche modo l'array ed esistono diversi modi per farlo:

- **Inizializzamneto di tutti gli elementi**: in questo caso tra parentesi graffe inizializziamo tutti gli elementi dell'array in base alla dimensione di quest'ultimo;
- **Inizializzamento parziale**: in questo caso inizializziamo solo parte degli elementi dell'array. La restante parte degli elementi, in base alla dimensione dell'array, verranno inizializzati tutti a 0;
- **Dimensione implicita**: in questo caso inizializziamo tutti gli elementi dell'array, ma tra parentesi quadre non inseriamo la dimensione che implicitamente verrà ricavata dal numero di elementi che inizializziamo
- **Inizializzazione con ciclo for**: in questo caso sapendo quanti elementi inizializzare, possiamo "costruire" l'array con il ciclo for inizializzando l'indice dell'array a 0 e nella condizione ovviamente dobbiamo far sì che il ciclo termini quando l'indice arriva a essere uguale a $DIM - 1$.

Vediamo degli esempi di queste inizializzazioni:

```
//tutti gli elementi inizializzati
int V[10] = {1,2,3,4,5,6,7,8,9,10};

//inizializzazione parziale
int W[10] = {1,2,3,4,5};


//Dimensione array definita implicitamente
int Z[] = {1,2,3,4,5};

int V[1000] = {0}; // tutti a zero!
int W[1000] = {}; //tutti a zero!
int Z[1000]; //non inizializzati !

// inizializzazione con ciclo for (solo numeri pari nell'array)
for(int j = 0; j<1000; j++)
        Z[j] =  j 2 ;
```

12.2 Array multidimensionali

Le regole sintattiche di lettura, dichiarazione e inizializzazione di un **array multidimensionale** sono le stesse che valgono per gli array monodimensionali:

```
#define N 3
#define M 4

//matrice dimensioni N x M
int V[N][M] = {0}; // tutti a zero
float W[N][M] = {}; // tutti a zero

//inizializzazione delle righe
int Z[N][M] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

In questo caso la matrice $N \times M$, verrà costruita in ordine seguendo lo shape della matrice, ovvero la precisione di righe e colonne. La cosa che differisce dagli array monodimensionali è l'inizializzazione parziale. Infatti se noi non dovessimo inizializzare almeno un parametro o di riga o di colonna si avrebbe un errore di compilazione:

```
#define N 3
#define M 4

//Compilation error !
int Z[][] = {1,2,3,4,5,6,7,8,9,10,11,12};

//////////////

#define M 4

 //OK!
int Z[][M] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

Nel caso di una matrice bidimensionale, ad esempio, l'inizializzazione non basta per determinare la lunghezza di righe e colonne, ma bisogna passare almeno un parametro di riga o colonna in modo tale che il compilatore determini lo **shape della matrice** con precisione, individuando l'altro parametro in base al **numero di inizializzatori**. In generale possiamo dire che passato un parametro di riga o colonna, il compilatore ricaverà il **numero di righe o colonne minimo** per fare entrare tutte le costanti nella matrice:

```
#define M 4

 //OK!
int Z[][M] = {1,2,3,4,5,6,7,8,9,10}; //quante righe? 3!
```

In questo esempio vediamo che la matrice è inizializzata con 4 colonne e 10 elementi: di conseguenza il numero di righe minimo per fare entrare tutte le costanti è 3. Con 2 righe, ad esempio, sarebbero rimasti esclusi 2 elementi.

Capitolo 13

PUNTATORI IN C++

13.1 Definizione di variabile puntatore

Per definizione una **variabile puntatore** è una variabile che **contiene un indirizzo di memoria**; ovvero contiene l'indirizzo a cui punterà la variabile per prelevare un valore.

13.1.1 Dichiarazione di variabili puntatore

Vediamo adesso come vengono dichiarate in C++ queste varibili puntatore:

```
int num = 12000;
int *p; //dichiara p come puntatore a int
p = &num; //assegna a p indirizzo di num
*p = 10; //modifica il dato puntato da p
```

Quindi analizzando questo piccolo programma, possiamo fare alcune considerazioni:

- La variabile puntatore deve essere dichiarata con un **tipo**, in questo caso intera, in modo tale da sapere quanti byte allocare per la variabile;
- Il carattere "" è chiamato carattere di **deferenziazione** e serve appunto per **dichiarare una variabile puntatore** o a **modificare il dato puntato dalla variabile** (ultima riga);
- il carattere "&" è chiamato carattere di **referenziazione** e serve per **estrarre l'indirizzo di una variabile** (terza riga);

13.1.2 Dichiarazione di un puntatore e contestuale inizializzazione

Vediamo adesso con un esempio come possiamo dichiarare e contestualmente inizializzare un puntatore:

```
int num=25;
int *p=&num;
cout << p << endl;
cout << * p << endl;
```

Vediamo di analizzare questo pezzo di codice e capire ancora meglio come funzionano i puntatori:

- In questo programma abbiamo prima dichiarato e inizializzato una variabile "num", assegnandogli 25;

- In seguito abbiamo prima dichiarato e inizializzato una variabile puntatore "$*p$", assegnandogli l'indirizzo della variabile "num";
- Di conseguenza con le stampe di "p" e "$*p$" possiamo vedere come nel primo caso, stampando il contenuto di "p", stamperemo l'**indirizzo puntato dal puntatore p**. Nel secondo caso, quando stampiamo il contenuto di "$*p$", stampiamo il **dato puntato da p**, ovvero "25"

Ovviamente la variabile puntatore non è una costante quindi può essere modificata riassegnando altri indirizzi di memoria:

```
int num=25;
int k = 20;
int*p=&num;
*p=34;
p=&k;
```

13.2 Array vs puntatori

Adesso che abbiamo introdotto i puntatori, possiamo dire che il **nome di un array** è il **puntatore alla prima cella di un array**:

```
double v[] = {1.2, 10.7, 9.8};
cout << v ; //stampa un indirizzo , ES: 0x11223344
cout << *v ; //stampa 1.2 (puntatore alla prima cella = cella con indice 0)
cout << v[0] ; //stampa 1.2 (puntatore alla prima cella = cella con indice 0)

 double w[] = {3.4, 6.7, 9.8};
 v = w; //Errore di compilazione!
```

Quindi possiamo dire analizzando questo codice, che array e puntatori sono strettamente correlati proprio per la motivazione descritta a inizio sezione; l'unica cosa che differisce tra array e puntatori è che **non possiamo modificare l'indirizzo di un array** come facevamo coi puntatori. Infatti vediamo che nelle ultime 2 righe ci da un **errore di compilazione** proprio per questo motivo.

13.2.1 Puntatori come indici di un array

Possiamo vedere inoltre che possiamo utilizzare i **puntatori come indici di un array**:

```
double v[] = {1.2, 10.7, 9.8};
double *ptr = v; // assegnamo l'indirizzo della prima cella di v al puntatore
cout << ptr [1]; //stampa 10.7
cout << ptr [2]; //stampa 9.8
```

13.3 Aritmetica dei puntatori

Abbiamo compreso quindi che la variabile puntatore punta alla prima cella di un array; la cosa interessante è che esiste un'**aritmetica dei puntatori** per il quale sommando il puntatore con un valore x, esso si sposterà di **x locazioni di memoria**, e ciò equivale a puntare a una cella dell'array spostata di x posizioni rispetto all'indice 0:

```
double v[] = {1.2, 10.7, 9.8};
double *ptr = v;
cout << *(ptr + 1); //stampa 10.7
cout << *(ptr + 2); //stampa 9.8
cout << *(v + 2); //stampa 9.8
```

Quindi in base a ciò che abbiamo detto e analizzando questo codice possiamo dire che il puntatore, dopo essere stato assegnato con l'indirizzo della prima cella dell'array v, si sposterà, ad esempio nella terza riga, di una cella dell'array o di una locazione di memoria contenente un double, puntando alla cella dell'array con indice 1. Di conseguenza tale **incremento dipende dai byte di spazio occupati dal tipo del puntatore**, in questo caso double, avendo dichiarato il puntatore come double.

13.3.1 Incremento del puntatore e dimensione di un tipo

Come conseguenza delle considerazioni appena fatte, ad esempio, supponendo che il tipo int abbia come **dimensione 4 byte** (*sizeof(int)*= 4), vediamo con diverse stampe come cambia l'indirizzo stampato:

```
int a[10];
cout << a; // 0x23aaff40
cout << (a+1); //0xaa23ff44
cout<< &a[1]; //0xaa23ff44
```

Quindi vediamo che sommando 1 al puntatore, esso si sposterà di 4 byte, puntando all'indirizzo contenente il prossimo intero, che anch'esso occuperà 4 byte; infatti **0xaa23ff44 = 0x23aaff40 + 4**.

13.3.2 Accesso ai valori di un array

Adesso che conosciamo i puntatori quindi sappiamo accedere diversi modi ad un array, utilizzando sia **indici** che **aritmetica dei puntatori**:

```
int v[] = {1,2,3};
int   ptr   = v;
```

Seguendo tale codice vediamo la varie modalità di accesso a un array:

Metodo di accesso	Esempio
Nome array e []	$v[2]$
Nome puntatore e []	$ptr[2]$
Nome array e aritmetica dei puntatori	$*(v+2)$
Nome puntatore e aritmetica dei puntatori	$*(ptr+2)$

Table 13.1: Metodi di accesso a un array

13.3.3 Operatori per l'aritmetica dei puntatori

Vediamo adesso gli operatori consentiti per i puntatori:

- **Operatori unari di incremento/decremento** $++/--$ applicati a una variabile puntatore;
- **Operatori binari di addizione e sottrazione** $+/-$ e di **assegnamento** $+=/-=$ nel quale un membro è un intero e l'altro un puntatore;
- **Sottrazione tra 2 puntatori**, che sottrae al valore di un puntatore, il valore dell'altro.

Esempi Vediamo con dei pezzi di codice come vengono utilizzati tali operatori:

- **Operatori di incremento/decremento**:

```
int v[] = {1,2,3,4,5};
int*ptr=v;
cout<<*(++ptr); //stampa 2
```

```
cout<<*(    ptr    );//stampa 1
cout <<*(ptr++); //stampa 1
cout<<*(ptr);//stampa 2
```

In questo esempio possiamo vedere come varia la variabile puntatore in base al fatto che ci sia un incremento/decremento **prefisso** o **postfisso**; nel primo caso, ovvero con incremento/decremento prefisso rispettivamente eseguirà l'incremento/decremento e poi valuta l'espressione; mentre con un incremento/decremento postfisso prima valuterà l'espressione e poi incrementerà/decrementerà l'espressione

- **Operatori di addizione/sottrazione e assegnamento**:

```
int v[] = {1,2,3,4,5};
int *ptr1 = v; //punta al dato       1
int *ptr2 = &v[4]; //punta al dato       5
cout << *(ptr1+1); //stampa il dato       2
cout << *( p t r 2 1 ); //stampa il dato       4
ptr2    =2;
cout << *ptr2; //stampa il dato       3
ptr1+=1;
cout << *ptr1; //stampa il dato       2
cout <<  p t r 2 ptr1 ; // stampa 1 (non  e   un dato..)
```

13.4 Inizializzazione di puntatori

13.4.1 Puntatore NULL

Non sempre capita che un puntatore è inizializzato a una variabile; per testare la **validità del puntatore**, cioè che esista qualcosa all'interno dell'indirizzo puntato dal puntatore si utilizza la macro "NULL":

```
int *ptr = NULL; //macro C/C++
float f;
int *ptr3 = &f ; // Errore di compilazione !

if(!ptr){ // test if ptr is valid

// . . do something
}
```

In questo codice vediamo ad esempio che quando proviamo a inizializzare la variabile puntatore "ptr3", con l'indirizzo di f, non avendolo inizializzato ci darà un errore.

13.4.2 Valore vs indirizzo

Quindi, abbiamo compreso che con i puntatori volendo possiamo confrontare sia l'indirizzo a cui puntano, sia il valore dell'indirizzo a cui puntano:

```
int num;
int *p1 = &num;
int *p2 = &num;

....
//confronta indirizzi
if (p1 == p2)
......
```

```
//confronta valori
if (*p1 == *p2)
```

13.5 Puntatori costanti e puntatori a costanti

13.5.1 Puntatori a costanti

Un **puntatore a costante** è un puntatore che punta all'indirizzo di memoria di una costante di qualsiasi tipo. In tal caso **è possibile cambiare l'indirizzo a cui punta il puntatore**, ma **non è possibile modificare il valore contenuto all'interno dell'indirizzo**. Ad esempio:

```
double d1= 10.8;
double d2 = 7.4;

const double *ptr1 = &d1;
*ptr1= 56.9; // errore di complilazione
ptr1 = &d2; // OK
```

13.5.2 Puntatori costanti

Un **puntatore costante** è un puntatore di tipo costante, nel quale **non può variare l'indirizzo di memoria contenuto al suo interno**, ma nel quale **possiamo cambiare il valore puntato**:

```
double d1= 10.8;
double d2 = 7.4;

double * const ptr1 = &d2;
*ptr1= 56.9; // OK
ptr1 = &d1; // errore di compilazione
```

Capitolo 14

FUNZIONI IN C++

Abbiamo visto a inizio corso quando abbiamo parlato di **programmazione strutturata/-modulare** che potevamo creare programmi, secondo un **approccio top-down**, che prevedeva una struttura ad albero nel quale dal programma principale, ramo principale, si diramavano diversi rami, che erano *sottoprogrammi* che rendevano **modulare** il nostro programma poichè erano programmi riutilizzabili. Questi sottoprogrammi in C++ vengono chiamate **funzioni** o *metodi*, che non sono altro che **blocchi di codice identificati da un nome**, che eseguono attività correlate tra loro.

14.1 Definizione e struttura di una funzione in C++

In C++ una funzione viene costruita seguendo questi criteri:

- **Nome** della funzione;
- **Tipo di ritorno** della funzione, che non sempre ha bisogno di essere definito, poichè non sempre abbiamo bisogno che ritorni qualcosa (void).
- Lista di argomenti o **parametri formali**, specificando il **tipo** di questi ultimi. Anche in tal caso non sempre dobbiamo passare dei parametri alla funzione;
- **Corpo** della funzione;

Vediamo un esempio:

```
char func(string s, int i){
     return s[i];
}


char ret = func("pippo", 3);
cout << ret
```

Analizzando tale programma, possiamo dire che:

- La funzione ha **tipo di ritorno char**, **nome func** e **parametri s e i**, che sono rispettivamente di **tipo stringa e intero**; il **ritorno della funzione** sarà quindi un carattere "char" della stringa s con indice i. Quindi "*returns*[*i*]" è il **corpo della funzione**;
- Le ultime due righe di codice invece sono chiamate **invocazione alla funzione**, cioè stiamo richiamando la funzione nel programma principale (main). Ciò che viene fatto è salvare in una variabile "ret", il ritorno della funzione, quindi un char, passando i

parametri s="pippo" e $i = 3$. Di conseguenza stampando ret, stamperemo il ritorno della funzione che in questo caso è il carattere 'p' della stringa pippo in posizione 3

14.1.1 Definizione vs prototipo di funzione

Abbiamo visto fino ad ora, come **definire una funzione**, in realtà possiamo **dichiarare il prototipo di una funzione** nel quale dichiariamo:

- **Tipo di ritorno**;
- **Segnatura** o *signature*: la segnatura è la **dichiarazione dei tipi dei parametri**, con la possibilità di omettere il nome dei parametri formali e il **nome della funzione**

Vediamo un esempio di prototipo di funzione e lo confrontiamo alla definizione della stessa funzione:

```
//dichiarazione del prototipo
double sum(double , double );


//definizione della funzione
double sum(double p, double q){
      double result = p+q;
      return result;
}
```

Conoscendo il **prototipo** e la **definizione** di una funzione, è buona pratica:

- raccogliere le **dichiarazioni dei prototipi** in un file **header** (ES:"modulo.h"). Tale header, oltre ai prototipi delle funzioni, può contenere:
  - **direttive** *#include* o *#define*;
  - **dichiarazioni di classi**;
  - ecc..
- raccogliere le **definizioni delle funzioni** in un file sorgente (ES: "modulo1.cpp");
- includere la direttiva **#include "modulo.h"** nei file cpp nei quali si fa uso di tale funzione;

14.2 Invocazione di funzioni

Adesso passiamo alla parte fondamentale delle funzioni, vedendo come fare a passare parametri come valore oppure come array e inoltre vedremo i vari modi con cui allocare in memoria.

14.2.1 Parametri formali vs parametri attuali

Abbiamo visto nella introduzione alle funzioni che tra parentesi tonde noi passiamo **parametri formali**; i valori che invece passiamo nell'**invocazione alla funzione** vengono chiamati **parametri attuali**, vedremo a breve che sarà molto conveniente, per dati grandi come array o matrici, utilizzare **puntatori come parametri attuali**. Inoltre, come abbiamo già detto nell'introduzione, tutta la programmazione strutturata/procedurale è una **sequenza di invocazione di funzioni** e, inoltre, per ottenere un **programma eseguibile** è necessario **invocare la funzione main**.

14.2.2 Area di memoria stack e record di attivazione

Quando invochiamo una funzione, viene riservata **una porzione di memoria per lo stoccaggio di dati utili per eseguire il corpo della funzione** che viene chiamata **stack**(pila). Lo stack è una struttura di tipo *LIFO*, cioè nel quale l'ultimo dato ad entrare è il primo ad uscire, che possiamo immaginare come una pila di piatti. Infatti lo stack è caratterizzato da **operazioni PUSH**, nel quale aggiungiamo un dato nel TOP dello stack; e **operazioni POP**, nel quale estraiamo un dato dal TOP dello stack. Consideriamo tale programma per capire come avviene l'**allocazione di memoria nello stack**:

```
void foo(int x){
     double c,d;
/////
}

/////
int main(){
     int a,b;

     ////

     foo(a);

     ////
}
```

Quindi vediamo in questo programma una funzione che ha nome "foo", definita all'inizio del blocco di codice e una funzione main, vediamo cosa succede nello stack:

- Innanzitutto viene allocato un "**record di attivazione**", chiamato anche *area di attivazione*, che con delle operazioni push deposita i **parametri attuali** (a) della funzione main, in questo caso chiamato x come **parametro formale**.
- Vengono allocate le **variabili locali** "c,d";
- Una volta terminate le istruzioni della funzione oppure con un istruzione *return*, il programma torna ad essere eseguito dall'istruzione successiva alla chiamata a foo, eliminando la parte di memoria utilizzata per l'attivazione della funzione

14.2.3 Passaggio di parametri mediante valore e indirizzo

Avendo compreso come funziona l'allocazione sullo stack dei parametri, abbiamo la possibilità di **passare parametri per valore o indirizzo**:

- Quando passiamo un parametro per "valore", ciò che stiamo facendo è passare alla funzione una **copia della variabile** che verrà impilata sullo stack. Da ciò ne consegue che, la funzione, lavorando su una copia, non potrà modificare il valore all'esterno, ma una volta utilizzato nella funzione la variabile stessa verrà cancellata. Facciamo un esempio:

```
void foo(int x){ //x parametro formale
            // . . .
             x = 90;
}

int main(){ //...
int a = 10;
foo(a); // parametro attuale  e   il VALORE contenuto in a
cout << a; //stampa 10!
}
```

- Quando passiamo un parametro per indirizzo, utilizzando l'operatore "&" di **deferenziazione**, possiamo agire sul valore contenuto nell'indirizzo con un **puntatore**. Ad esempio:

```
void spam(int *x){
        // . . .
        *x = 90;
}

int main(){ //...
int a = 10;
spam(&a); // parametro attuale e l'INDIRIZZO di a
cout << a; //stampa 90!
}
```

  In tal caso utilizzando un puntatore, dovremo per forza passare come parametro attuale un indirizzo che ci permetterà di effettuare modifiche sul valore puntato

14.2.4 Passaggio di parametri array

Il **passaggio di parametri array ad una funzione avviene sempre per indirizzo**. Ricordiamo infatti che il nome di un array è un puntatore costante al primo elemento di un array. Vediamo infatti un esempio:

```
void init(int*v, int n){ // primo modo
 // . . .
    for(int j=0; j<n; j++){
    v[j]=0; 5}
}

int main(){ //...
    int x[10];
    init(x, 10);
}

//////////////

void init(int v[], int n){ // secondo modo equivalente a int *v
 // . . .
    for(int j=0; j<n; j++){
    v[j]=0; 5}
}

int main(){ //...
    int x[10];
    init(x, 10);
}
```

Notiamo che quando nel main passiamo l'array, nonostante sia un *puntatore alla prima cella dell'array*, non possiamo passare l'indirizzo con l'operatore di referenziazione, ma passiamo solo il nome dell'array in questione.

Passaggio di array multidimensionali Per il passaggio di array multidimensionali allocati sul segmento DATA o sullo STACK, nel prototipo della funzione che riceve il dato, **vanno specificate tutte le dimensioni, dalla seconda in poi**. Vediamo un esempio:

```
}
#define N 5
```

```
#define M 10
void init(int v[][M]); // OK
void init(int v[N][M]); // OK
void init(int v[][], int n); // Err. di compilazione!
void foo(){
int w[N][M];
init(w, N);
}
```

Passaggio di array multidimensionali tramite puntatore Dobbiamo fare molta attenzione quando passiamo un **puntatore a una matrice**, o una **matrice di puntatori**:

```
void init(int *v[M][L], int n);
int x[10][M][L];
init(x, 10); // Errore di compilazione!

////////////

void init(int (*v)[M][L], int n); // OK
int x[10][M][L];
init(x, 10); // OK
```

Vediamo che nel primo programma stiamo passando una **matrice di puntatori**, per cui se passiamo nel main ad esempio un'array tridimensionale avremo un errore di compilazione. Se, invece, passiamo un **puntatore ad una matrice**, ovvero un array tridimensionale la scrittura di passaggio dei parametri è valida. Ovviamente qui abbiamo fatto l'esempio con le matrici, ma ovviamente vale per qualsiasi dimensione di un array.

14.2.5 Categorie di allocazione delle variabili

Ci siamo limitati fino ad ora a descrivere un tipo di allocazione che utilizzava come area di memoria lo stack. Vediamo adesso i vari tipi di **allocazioni di variabili**:

- **Allocazione automatica**: Avviene quando dichiariamo e inizializziamo delle variabili **all'interno del corpo della funzione**. Vengono chiamate **variabili locali** proprio perchè sono limitate all'utilizzo all'interno della funzione e vengono "distrutte" o deallocate una volta che il blocco di esecuzione della funzione termina. Inoltre occupano l'area di memoria dello **STACK**;
- **Allocazione dinamica**: In questo caso le variabili possono essere dichiarate e inizializzate **in qualsiasi punto del programma** con la direttiva **new** e vengono "distrutte" o deallocate con la direttiva **delete** sul puntatore. Inoltre l'area di memoria utilizzata è chiamata **HEAP**;
- **Allocazione statica**: In questo caso le variabili vengono dichiarate e inizializzate **al di fuori di qualsiasi blocco**. Il ciclo di vita inizia e termina con il programma stesso; e sono allocate nell'area di memoria chiamata **DAT**

Capitolo 15

Allocazione dinamica di memoria in C++

Abbiamo accennato già ai vari tipi di **allocazione di memoria**, e, parlando di **allocazione dinamica**, possiamo dire che il segmento di memoria utilizzato è chiamato **heap**, e, a differenza dello *stack* in cui una volta che ritornavamo dalla funzione al programma chiamante le **variabili locali** venivano distrutte, la memoria allocata dinamicamente non viene deallocata fino a quando noi la deallochiamo manualmente.

15.1 Allocazione e deallocazione in memoria dinamica

Vediamo adesso come utilizzare la memoria dinamica, cioè allocare e deallocare variabili e puntatori in C++

15.1.1 Allocazione: operatore *new* in C++

L'**operatore new** è quello che ci permette di allocare dinamicamente le variabili. In tal caso avremo che nella sintassi dovremo indicare la **variabile puntatore al tipo da allocare**, e assegnargli, eventualmente, il **contenuto** che può essere una singola variabile, un'array o un oggetto. Vediamo ad esempio questo codice:

```
/*
allochiamo in memoria dinamica un'intero puntato dalla variabile
puntatore a intero 'a' e gli assegniamo 10
*/
int *a = new int(10);

 /*
allochiamo dinamicamente un'array di tipo double con dimensione 10
*/
double *arr = new double [10];
```

15.1.2 Deallocazione: operatore *delete*

Come abbiamo detto nell'introduzione, a differenza dello stack, la memoria in questo caso va **deallocata manualmente**:

```
int *a = new int;

//////

delete a;
```

Se dobbiamo **deallocare un blocco di memoria** avremo:

```
double *arr = new double [10];

/////

delete [] arr;
```

Quando deallochiamo la memoria, comunque il puntatore punterà sempre allo stesso indirizzo di memoria. Cioè stiamo deallocando solo il contenuto della memoria dinamica, e, il puntatore si trova sempre nello stack.

15.2 Memory leak e double deletion

Può capitare di commettere "errori" sull'allocazione/deallocazione dinamica, come ad esempio **memory leak** o **double deletion**

Memory leak Vediamo un esempio:

```
double *arr = new double [10];
// . .
double *v = new double[15];
// . .
v=arr;
```

Questo è un esempio di memory leak, infatti **copiando l'indirizzo differente** rispetto a v, proprio nella variabile v, si perde il riferimento al blocco di memoria contenente l'array di double di 15 elementi. Questo provoca un effetto **aliasing** per cui non sarà più possibile deallocare la variabile v dalla memoria dinamica.

Double deletion A volte può capitare, operando con un programma molto lungo, di **deallocare una stessa variabile più volte**. Ciò provoca un effetto disastroso e indefinito; per cui è buona norma ad ogni tentativo di deallocazione di una variabile una variabile:

- Controllare il valore del puntatore prima di deallocarlo;
- "azzerare" il puntatore una volta che è stato deallocato.

In pratica facciamo questo:

```
double *arr = new double [10];
//...
if(arr){ // arr punta a qualcosa?
    delete [] arr;
    *arr=nullptr; //azzeriamo il puntatore
}
```

15.3 Funzioni che restituiscono un puntatore

Abbiamo questo pezzo di codice:

```
int *func(int k){
    int arr[k];
    for(int i=0; i<k; i++)
      arr[k] = 2 k ;
    return arr;
    }

    // ....
int *array = func (10);
```

Notiamo che questa funnzione **restituisce una variabile puntatore a intero**, e vediamo che all'interno della funzione stiamo inizializzando un array e, quindi, un puntatore costante alla prima cella dell'array di dimensione *k*. Modificando l'array in tale funzione, ma avendo **allocato l'array nello stack**, quando nel main chiameremo la funzione, tutte le variabili locali che erano presenti nell'area di attivazione dello stack vengono cancellate. Per cui questa funzione è semanticamente scorretta poichè è come se lavorasse su una copia dell'array che poi viene distrutta. La versione corretta, infatti, prevede un **allocazione dinamica**, in modo tale, che le modifiche che facciamo all'interno della funzione sulle variabili e i puntatori ci restituiranno il valore modificato:

```
int *func(int k){
    int *arr= new int [k];
    for(int i=0; i<k; i++)
      arr[k] = 2 k ;
    return arr;
    }

    // ....
int *array = func (10);
```

Capitolo 16

Array di caratteri e oggetti string in C++

Abbiamo già parlato di stringhe in generale. Adesso che abbiamo trattato i puntatori e gli array, dobbiamo sapere che in C++ esistono due modi di utilizzare le stringhe:

- La stringa può essere utilizzata come **array di caratteri**, e utilizzare varie funzioni dedicate attraverso la libreria $< cstring >$. In linguaggio C questo è l'unico modo con cui poterle utilizzare;
- Possiamo utilizzare la stringa come **oggetto**, con tutti i suoi pro, includendo la libreria $< string >$.

16.1 Stringhe come array di caratteri

Vi sono molti modi per scrivere la stringa come array di caratteri come possiamo vedere in questo pezzo di codice in cui inizializziamo un array di caratteri in diversi modi:

```
char s[] = "stringa";
char s[] = {'s','t','r','i','n','g','a',0};
char s[] = {'s','t','r','i','n','g','a','\0'};

////

char s[] = {'s','t','r','i','n','g','a'}; //errore di compilazione
```

Notiamo che quando inizializziamo la stringa per caratteri, se utilizziamo i singoli apici, indicando quindi un carattere; a fine stringa si dovrà porre il **carattere di terminazione** che può essere " 0 *o* 0".

16.1.1 Funzioni di libreria per array di caratteri

C++ fornisce diverse funzioni per lavorare con le stringhe in stile C, ovvero quelle definite come array di caratteri. Queste sono dichiarate nell'header cstring. Le funzioni definite in cstring accettano come parametri formali sia puntatori a carattere (array di caratteri) che puntatori costanti a caratteri (letterali stringa).

Copia La funzione *strcpy* permette di copiare una stringa da una sorgente a una destinazione:

```
char name[15];
strcpy(name, "pippo");
cout << name;
```

Attenzione al fatto che non verrà fatto nessun controllo sull'area di memoria puntata da name. Se name non contiene abbastanza spazio, potremmo avere un comportamento indesiderato.

Confronto lessicografico Possiamo controllare quale tra due stringhe "viene prima" dell'altra se ordinate lessicograficamente mediante *strcmp*:

```
//restituisce un valore >0 perch  pippo>paperino
cout << strcmp("pippo","paperino") << endl;
//restituisce un valore <0 perch  pippo<paperino
cout << strcmp("paperino","pippo") << endl;
//restituisce 0 perch  pippo=pippo
cout << strcmp("pippo","pippo");
```

Ricerca di sottostringhe E' possibile cercare una sottostringa all'interno di una stringa di caratteri mediante il metodo strstr che restituisce un puntatore al primo carattere della prima stringa che indica la prima occorrenza della seconda stringa. Ad esempio:

```
char s1[] = "ddabcdd";
char s2[] = "abc";

char *s3 = strstr(s1,s2);
cout << s3; //abcdd
```

Conversioni numeriche E' possibile convertire una stringa in un numero mediante le seguenti funzioni:

- *atoi*: da array di caratteri a int;
- *atol*: da array di caratteri a long;
- *atoll*: da array di caratteri a long long;
- *atof*: da array di caratteri a double.

Esempi:

```
cout << atoi("22") << endl; // 22
cout << atol("35") << endl; // 35
cout << atoll("18") << endl; // 18
cout << atof("2.2") << endl; // 2.2
```

sscanf Un metodo più sofisticato di conversione è dato da **sscanf**, che permette di specificare il formato del numero o dei numeri da leggere. I formati principali sono i seguenti:

- %: int;
- %: float;
- %: double (long float);

sccanf vuole in input la stringa, il formato e un riferimento alla variabile in cui mettere il risultato. Esempi:

```
int i;
float f;
double d;

sscanf("15", "%d", &i);
sscanf("2.2", "%f", &f);
sscanf("3.333", "%lf", &d);

cout << i << endl << f << endl << d; //15, 2.2, 3.333
```

E' possibile anche leggere più valori in una volta da una stringa formattata:

```
sscanf("L'intero : 376, il float : 2.98, il double : 92.22", "%d %f %lf", &i, &f, &d);
cout << i << endl << f << endl << d;
```

I formati specificati verranno associati ai riferimenti alle variabili nell'ordine specificato. E' possibile anche scegliere quante cifre leggere:

```
int x;
sscanf("123456789", "%06d", &x); //legge solo le prime 6 cifre
cout << x;
```

16.2 Oggetti string

Gli oggetti string costituiscono un altro modo per rappresentare le stringhe. Questo approccio è basato sulla programmazione a oggetti e quindi più in linea con la filosofia di C++. Per usare gli oggetti string dobbiamo includere l'header string:

```
#include<iostream>
#include<string>
using namespace std;
/////
/////
string s = "ciao";
cout << s;
```

Gli oggetti string permettono di gestire le stringhe più ad alto livello. Ad esempio, è possibile copiare una stringa in fase di dichiarazione:

```
string s1 = "stringa";
string s2(s1); //copia s1 in s2
cout << s2;
```

16.2.1 Overloading degli operatori

Gli operatori degli oggetti string sono sovraccaricati in modo da poter confrontare le stringhe in maniera naturale. In particolare:

- L'operatore di assegnamento = permette di copiare la stringa a destra dell'operatore nella stringa alla sinistra dell'operatore;
- L'operatore + permette di concatenare stringhe;
- Gli operatori di confronto $>, <, >=, <=, ==, != $ permettono di verificare se due stringhe sono uguali, diverse o di capire qual è il loro ordinamento lessicografico.

Esempi:

```
string s1 = "pippo";
string s2 = "paperino";
```

```
cout << (s1==s2) << endl;// 0
cout << (s1>s2) << endl;// 1
cout << (s1<s2) << endl;// 0
cout << (s1!=s2) << endl;// 1
cout << (s1+s2) << endl;// pippopaperino
```

16.2.2 Accesso ai caratteri

E' possibile accedere ai singoli caratteri di una stringa mediante la notazione *stringa*[*i*], come per gli array:

```
string s1 = "pippo";
cout << s1[0] << s1[1]; //pi
```

Tuttavia, questa notazione non fa controlli sulla lunghezza della stringa. Ad esempio, se proviamo ad accedere al carattere 100 della stringa "pippo", non otteniamo un errore, ma un carattere vuoto:

```
string s1 = "pippo";
cout << s1[0] << s1[100]; //p
```

Un modo diverso per accedere ai caratteri, consiste nell'usare il metodo at, che ci notifica l'errore in caso che la lunghezza della stringa sia superata:

```
string s1 = "pippo";
cout << s1.at(0) << s1.at(1); // pi
////
////
string s1 = "pippo";
cout << s1.at(100); //errore
```

16.2.3 Altri metodi di accesso alle stringhe

Vediamo adesso altri metodi che permettono di manipolare gli oggetti string:

```
string s1 = "pippo";
string s2 = "paperino";

s1.append(s2); //concatena s2 a s1. Equivalente a s1+=s2
cout << s1; //pippopaperino

//////////

string s1 = "pippo";
string s2 = "paperino";

s1.push_back('c'); //concatena un singolo carattere
cout << s1;//pippoc

////////

string s1 = "pippo";
string s2 = "paperino";

s1.assign(s2); //assegna s2 a s1
cout << s1; // paperino
```

```
///////

string s1 = "pippo";
string s2 = "paperino";

s1.clear(); //re-inizializza la stringa con una stringa vuota
cout << s1;

//////

string s1 = "ddabcd";
string s2 = "abc";
//restituisce l'indice della prima occorrenza di s2 in s1
cout << s1.find(s2) << endl; //2

/////

string s1 = "ddabcd";
//elimina due caratteri a partire da quello di indice 2
cout << s1.erase(2,2) << endl; //ddcd
```

Capitolo 17

INTRODUZIONE ALLA PROGRAMMAZIONE A OGGETTI

Il linguaggio C++ ci permette di utilizzare gli **oggetti**, che sono strumenti molto utili per i programmatori. Vedremo infatti in tale capitolo come favoriscano la *modularità* e la capacità di riuso del codice.

17.1 Oggetti

Gli oggetti sono ***tipi*** *non primitivi*: ovvero non sono definiti dal linguaggio stesso, ma dall'utente (*user-defined*) e si tratta di **entità** che simulano oggetti del mondo reale che hanno la capacità di interagire tra loro tramite **scambio di messaggi**.

17.1.1 Caratteristiche di un oggetto

Un oggetto si compone di:

- **Stato**: rappresenta l'insieme di **proprietà** e **attributi** dell'oggetto;
- **Comportamento**: rappresenta l'insieme di **operazioni** che è capace di eseguire l'oggetto, tra le quali:
  - **Cambiare lo stato iniziale**;
  - **Inviare messaggi** ad altri oggetti.

Esempio Facciamo un esempio cercando di rappresentare un ***automobile*** *tramite oggetti*: ovviamente da quanto abbiamo capito ci servirà definire *stato* e *comportamento*:

- **Stato**:
  - targa;
  - colore;
  - accesa;
  - velocità;
  - livello di benzina;

  - ...

- **Comportamento**:
  - Accensione;
  - Leggere la targa;
  - Accelera/decelera;
  - Leggere i giri del motore;
  - ...

Possiamo quindi ridefinire i concetti di **stato** e **comportamento** in tal modo:

- Lo **stato** di un oggetto, che è costituito da una serie di *attributi* che definiscono le proprietà in modo astratto, è **mappato su variabili**;
- Il **comportamento** di un oggetto è descritto tramite *metodi* che vengono detti **funzioni membro**.

17.1.2 Tipo vs oggetto

Quindi, prendendo come riferimento lo *stato* di un oggetto, vediamo come viene definito in maniera **formale** il **tipo automobile**, e come vengono date delle specifiche in un **oggetto di tipo automobile**:

- **Tipo automobile**:

```
Targa: string
colore: int
MotoreInMoto: bool
velocita: int
livellobenzina: short
```

- **Oggetto di tipo automobile**:

```
targa = JK 1234
colore = 112233
motoreInMoto = false
velocita = 0 km/h
livelloDiBenzina = 28 lt.
```

17.1.3 Principio di identità e di conservazione dello stato

In base a quanto detto nel precedente paragrafo, esistono due **principi fondamentali** alla base degli oggetti, che regolano poi lo stato e il comportamento di ogni oggetto:

- **Principio di identità**: ogni oggetto di un determinato tipo è **distinguibile** da altri oggetti dello stesso tipo;
- **Principio di conservazione dello stato**: ogni oggetto, durante l'esecuzione del programma, *mantiene il proprio stato* per un **tempo indefinito**.

17.1.4 Ciclo di vita di un oggetto

Nel corso dell'elaborazione di un programma, un oggetto presenta un suo **ciclo di vita** che è regolato dai seguenti passi:

- **Creazione** dell'oggetto, che è lo *stato iniziale* in cui si presenta l'oggetto;
- **Utilizzo** dell'oggetto, cioè i *cambiamenti di stato* o *messaggi* a cui è sottoposto;
- **Distruzione** dell'oggetto, nel quale *liberiamo la memoria*.

17.2 Classi vs oggetti

Abbiamo precedentemente fatto un confronto fra una *tipo e oggetto*; vediamo adesso di approfondire tale concetto, che sta alla base del concetto di **classe**. Appunto, parlando di classi, si tratta di una *descrizione formale del* ***tipo*** che descrive: *come deve essere* ***costruito*** *un* ***oggetto istanza*** *di essa* e la ***struttura*** *dell'oggetto istanza di essa.* Possiamo immaginare quindi la classe e gli oggetti istanza della classe, ovvero come vengono chiamati come nella seguente immagine, che descrive come può essere rappresentata la *classe automobile*, e gli oggetti *m1* e *m2* di *tipo automobile*:

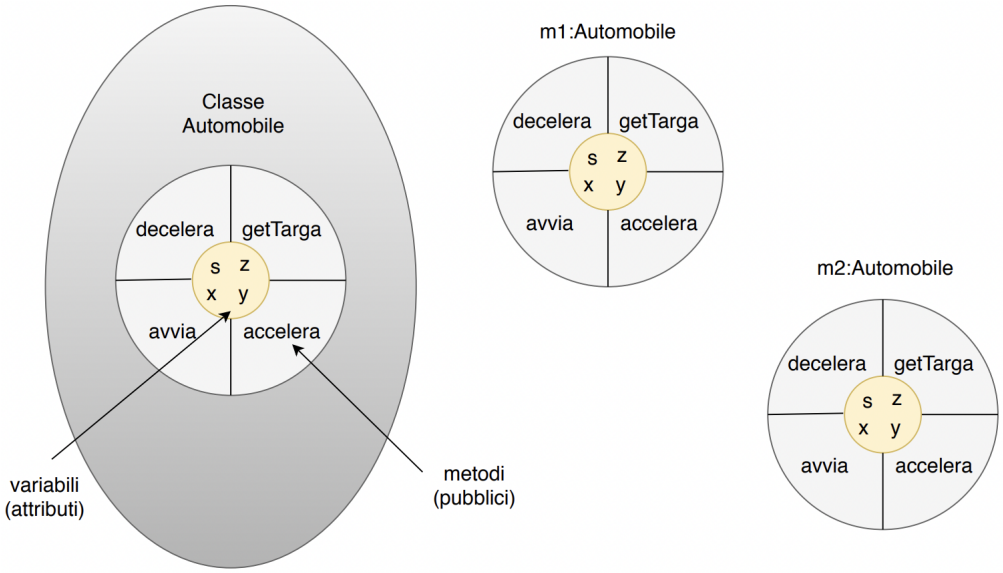

Figure 17.1: Classe automobile e oggetti istanza della classe automobile

Possiamo fare alcune considerazioni sull'immagine 17.1, che in parte abbiamo anche già trattato:

- Parlando di **classe**, quindi, la possiamo vedere come una **rappresentazione delle caratteristiche salienti** di un'*entità* che è proprio l'**oggetto istanza della classe**;
- Tale oggetto si dice **istanza** della classe poichè il suo **stato** e il suo **comportamento** sono conformi alla descrizione della classe. Tuttavia per il principio di *identità* sappiamo che ogni oggetto ha un proprio valore per le sue **variabili** o **attributi**. Infatti ricordiamo che gli oggetti, come possiamo vedere nell'immagine 17.1, sono essenzialente *contenitori* di:
  - **Variabili** o **attributi**, che descrivono lo *stato dell'oggetto*;
  - **Metodi**, detti **funzioni membro**, che possono avere più scopi, tra i quali:
    - **Elaborare le variabili**;
    - **Cambiare lo stato dell'oggetto**;
    - **Interagire con altri oggetti**

17.2.1 Progettazione della classe

In C++ la parola riservata a descrivere una classe è **class**, e il programmatore, procede nei seguenti passi per la *progettazione* di essa:

- Inizialmente vi è una fase di ***analisi***, che serve al progettista a capire i **requisiti** per descrivere la classe stessa, e il **contesto** nel quale si sta andando a descrivere una determinata classe;
- Vi è una fase di ***design*** nel quale il progettista *modella* ***stato*** *e* ***comportamento***;
- Vi è, infine, un ultima fase che è quella di ***implementazione*** nel quale il progettista:
  - deve **mappare** il *modello* dello **stato** in un insieme di **variabili o attributi**;
  - deve **mappare** il *modello* del **comportamento** in un insieme di **metodi**.

17.3 Messaggi

Gli oggetti possono interagire tra loro tramite **scambio di messaggi**, che avviene tramite **invocazione dei metodi**. Un *metodo*, sappiamo che è una **funzione associata** ad una ben determinata *classe*. Ogni metodo facente parte di una determinata classe sappiamo che fa parte dell'*implementazione del* ***comportamento*** di un oggetto. Ora, all'invocazione di un metodo, possiamo avere tre tipi di comportamento diverso:

- Possiamo **prelevare un'informazione**, che si traduce nel prelevamento de parte dello stato dell'oggetto stesso;
- Possiamo **causare un cambiamento di stato** dell'oggetto;
- Possiamo **avviare un'attività** per il quale l'oggetto è stato progettato

Vedremo tra poco che tali tipi di **invocazioni di metodi**, si tradurranno nei **tipi di messaggi** che si potranno inviare tra oggetti.

17.3.1 Sintassi dell'invocazione di un metodo

Per quanto riguarda la **sintassi** dell'invocazione di un metodo in C++ è la seguente:

```
obj.metodo(p1, p2,...);
```

In tale sintassi:

- **obj** rappresenta l'oggetto *destinatario del messaggio*;
- **metodo** denota il nome del metodo;
- **p1, p2** sono gli eventuali parametri che passiamo per l'invocazione del metodo.

Esempio Vediamo di fare un esempio per capire la finalità dell'invocazione dei vari metodi, e come viene scritto in codice:

```
Automobile m1; // oggetto di tipo automobile, con nome m1
m1.setMotore("elettrico"); // cambiamento di stato
m1.avviatergicristallo(); // avvia un attivit
m1.getVelocit (); //invio di informazioni al chiamante
```

In tal caso possiamo vedere, all'invocazione di 3 metodi diversi, 3 diverse finalità per ognuno:

- *setMotore()* provoca un **cambiamento di stato** poichè stiamo cambiano una **variabile dell'oggetto**;
- *avviatergicristallo()* **avvia un'attività**;
- *getVelocità* **invia un informazione al chiamante** m1

17.3.2 Flusso di invocazione dei metodi

Quando invochiamo un metodo succede che:

- Il flusso di controllo procede con l' **esecuzione del metodo**;
- Alla **fine dell'esecuzione del metodo**, ovvero o in corrispondenza di una **return**, oppure dopo l'**ultima istruzione del metodo**, il flusso di esecuzione procede con l'**esecuzione della riga di codice successiva alla chiamata**.

Se facessimo quindi un confronto fra programmazione **strutturata/procedurale** e quella **orientata agli oggetti**, possiamo dire che la differenza sta nel fatto che nella prima si procede su una **sequenza di invocazione di funzioni**; nella seconda, invece, si procede per un **flusso di messaggi tra oggetti**, ovvero una **sequenza di invocazioni di metodi**, come possiamo vedere nella seguente immagine:

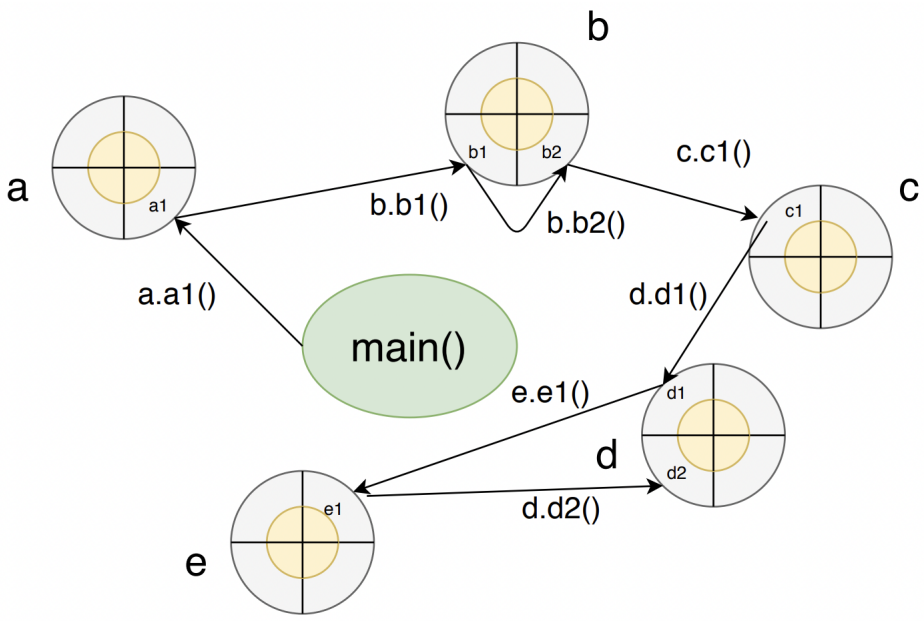

Figure 17.2: Flusso di invocazione di metodi

17.3.3 Tipi di messaggio

In base a quanto visto nella sezione precedente, abbiamo che le **tipologie di messaggio** grazie ai quali gli oggetti interagiscono tra loro a seguito di *invocazione di metodi*, sono 3:

1. **Informativo**: in tal caso informiamo l'oggetto chiamante che qualcosa è cambiato e che deve **aggiornare il suo stato**. Ad esempio con il metodo *setTarga()* possiamo dire all'oggetto chiamante di *cambiare il suo stato* aggiornando il valore di targa;
2. **Interrogativo**: in tal caso chiediamo all'oggetto chiamante **informazioni sullo stato dell'oggetto**. Ad esempio con il metodo *getTarga()* possiamo chiedere all'oggetto chiamante di restituirci il numero di targa;
3. **Imperativo**: in tal caso diciamo all'oggetto chiamante di **avviare una o più attività**. Ad esempio con il metodo *avviamotore()* possiamo chiedere al chiamante di avviare delle funzioni per il quale risulta che il motore sia stato avviato.

Esempio Vediamo alcuni esempi delle varie tipologie di messaggi:

```
Dado d;
d.effettuaLancio(): void; //avvia un'attivita
d.getUltimoLancio(); //interrogativo
```

Possiamo vedere in tal caso che:

- il metodo *effettuaLancio()* simula il lancio di un dado e non restituisce nulla (void), quindi è un **messaggio imperativo**;
- il metodo *getUltimoLancio()* è un **messaggio interrogativo** poichè interroga l'oggetto sul suo stato (faccia superiore del dado)

17.4 Metodo costruttore

Il **metodo costruttore** serve ad **inizializzare lo stato dell'oggetto** ed è una **chiamata automatica**: ovvero il compilatore *forza* la chiamata in modo tale che creazione dell'oggetto e inizializzazione dello stato avvengano *contestualmente*. Inoltre il **nome del costruttore** è dato dal nome stesso della classe. Ad esempio prendendo la classe *Dado*, abbiamo il metodo costruttore *Dado()*.

Capitolo 18

Aggregati e collezioni di oggetti

18.1 Relazioni "part-of"

Molto spesso capita di dover modellare ad oggetti *entità complesse*. Per questo si suddivide l'entità in un insieme di costituenti dell'entità stessa, che nella programmazione a oggetti si traducono:

- In fase di ***codifica*** nell'inserire **diverse classi** contenute nella **classe principale**;
- In fase di ***esecuzione del codice*** nell'avere **diversi oggetti** contenuti in un **oggetto principale**.

18.1.1 Esempio classe "automobile"

Se prendiamo l'esempio dell'entità ***automobile***, ad esempio, possiamo notare che essa possiede diverse componenti, che sono altrettante classi come:

- *Telaio*;
- *Motore*;
- *Ruote*;
- ecc.

Inoltre tali componenti, facenti parte di un unica classe automobile, scambieranno **messaggi** tra loro, come:

- il sistema di ***avviamento*** dell'automobile comanda, tra le altre cose, l'accensione del motore (messaggio di tipo *imperativo*);
- il galleggiante del **serbatoio** invia *messaggi informativi* all'indicatore di livello nel cruscotto;
- oppure l'indicatore di livello nel cruscotto invia messaggi *interrogativi* al galleggiante nel serbatoio del carburante per conoscerne lo stato;

Tradotto in codice, nella dichiarazione della classe automobile, avremp quindi che:

```
class Automobile{
    //...
    Serbatoio serbatoioCarburante;
    Motore   motoreABenzina;
}
```

Considerando questo codice ci possiamo porre le seguenti domande:

- Quando vengono creati gli oggetti *serbatoioCarburante* e *motoreABenzina*? La risposta è: **contestualmente alla creazione dell'oggetto istanza di Automobile**.
- Quando vengono distrutti gli oggetti *serbatoioCarburante* e *motoreABenzina*? La risposta è: **contestualmente alla distruzione dell'oggetto istanza di Automobile**.

Su queste due domande si basa la *tipologia di relazione* che può sussistere tra oggetto *contenitore* e oggetto contenuto

18.1.2 Relazione stretta o di composizione

Consideriamo adesso, avendo fatto l'esempio della classe automobile, consideriamo la relazione automobile,motore e la relazione automobile, serbatoio: queste relazioni vengono chiamate **relazioni strette o di composizione** poichè l'oggetto *contenuto* (serbatoio, motore) è creato e distrutto insieme all'oggetto *contenitore*

Diagramma UML della relazione di composizione Se volessimo rappresentare *graficamente* le relazioni descritte precedentemente utilizziamo i **diagrammi UML** (*unified modeling language*). Ad esempio per la classe automobile abbiamo:

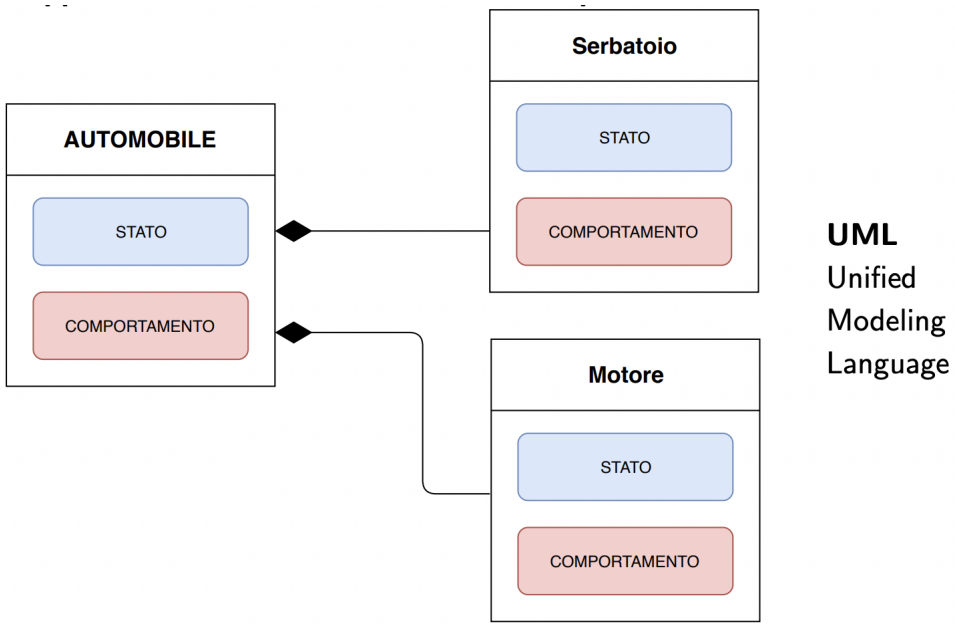

Figure 18.1: Diagramma UML con relazione di composizione

Quindi abbiamo che nella *relazione di* ***composizione*** della classe automobile l'oggetto *contenitore* è "AUTOMOBILE" che al suo interno contiene gli oggetti "Serbatoio" e "Motore". Ciò che ci fa capire che questi oggetti sono contenuti in automobile è il *rombo pieno*

18.1.3 Relazione di aggregazione

Abbiamo fatto un esempio di *relazione di composizione* prendendo in considerazione relazioni come automobile,motore o automobile,serbatoio. Prendiamo adesso in considerazione, come

esempio, la relazione automobile, persona. Prendendo in maniera specifica la *classe* ***persona***, abbiamo che ci potrebbe essere un *booleano* tra gli attributi che indica se si sta guidando l'auto o meno. Ma se mettessimo un oggetto *allaGuida* di tipo *persona* avrebbe comunque una sua **vita propria**: tale relazione che intercorre tra la classe persona e la classe automobile è detta **relazione di aggregazione** poichè l'oggetto contenuto di tipo persona ha **vita propria** rispetto alla classe automobile. La rappresentazione in UML è la seguente:

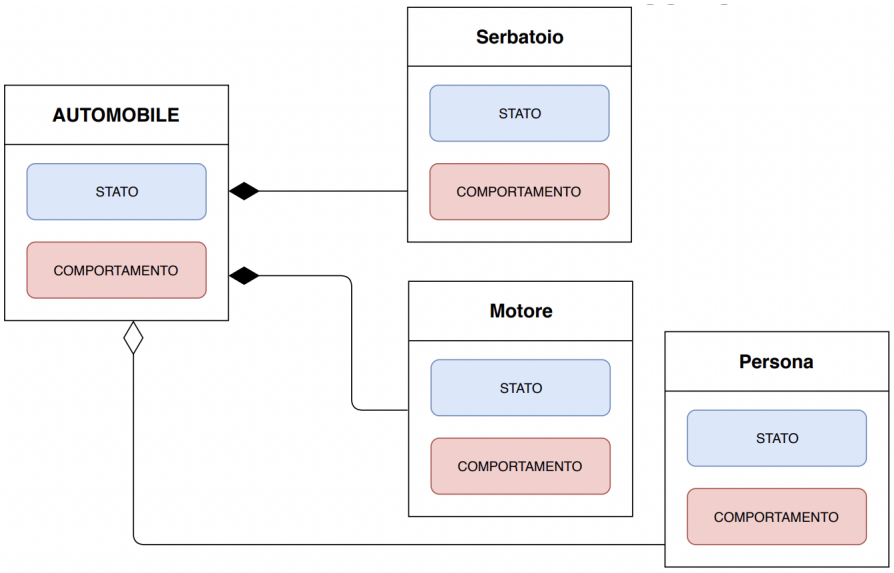

Figure 18.2: Aggregazione e composizione

In tal caso l'**aggregazione** è indicata, nel diagramma, dal *rombo vuoto*.

18.1.4 Composizione vs aggregazione

Andiamo quindi a riassumere le differenze principali tra i due tipi di relazione:

- **Relazione di composizione**:
  - L'oggetto *contenuto* **non ha vita propria**;
  - Si realizza con un degli oggetti contenuti in un **oggetto contenitore**;
  - L'oggetto *contenitore* è **responsabile della creazione e distruzione degli oggetti contenuti**
- **Relazione di aggregazione**:
  - L'oggetto *contenuto* ha **vita propria**;
  - Anche in questo caso, si realizza con degli oggetti contenuti in un **oggetto contenitore**;
  - Il contenitore **non è responsabile della creazione e distruzione del contenuto**

Capitolo 19

Implementazione di classi in C++

Nei capitoli precedenti, accennando alle classi, abbiamo capito che si tratta di una sorta di ***progetto*** che specifica la **struttura di un istanza della classe stessa**, che chiamiamo ***oggetto***.

19.1 Dichiarazione di una classe

Un esempio di ***dichiarazione di classe*** può essere la seguente:

```
class ClassName {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
}
```

Vediamo di analizzare il corpo di questa classe nelle sue varie parti:

- ***ClassName*** indica il **nome della classe** che stiamo andando a dichiarare;
- *access_specifier* indica il **modificatore di accesso della classe**, che può essere *private, public o protected*;
- ***member*** indica un **membro della classe**, che può essere una *variabile o attributo* o un *metodo*. Dai precedenti capitoli abbiamo capito che gli attributi memorizzano dei dati mentre i metodi scambiano messaggi tra gli oggetti ovvero permettono di comunicare con gli oggetti attraverso *messaggi informativi, interrogativi o imperativi*.

19.1.1 Esempio di classe (senza costruttore)

Vediamo un semplice esempio di classe:

```
#include<iostream>
using namespace std;
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
```

```
    int area ();
};
```

Il codice qui sopra definisce una classe di nome *Rectangle* con 2 variabili (width, height) entrambe **private** poichè non è stato specificato il modificatore d'accesso; e abbiamo 2 metodi pubblici di cui è stato dichiarato soltanto il **prototipo**. Come per le funzioni, infatti, l'implementazione dei metodi può essere scritta anche in un apposito file utilizzando l'***operatore di scope*** (::), come possiamo vedere nel seguente esempio:

```
void Rectangle::set_values(int x, int y) {
    height = x;
    width = y;
}
//////
//////
int Rectangle::area() {
    return height*width;
}
```

In tal caso quindi abbiamo che:

- **set_values()** inizializza le variabili height e width, che, pur essendo *private*, possiamo utilizzare poichè stiamo operando all'interno della classe Rectangle;
- **area()** restituisce il prodotto $height * width$, che sarebbe l'ipotetica ***area del rettangolo***.

Una volta che abbiamo definito la classe e implementato i metodi di quest'ultima, possiamo dichiarare un oggetto di tipo Rectangle e interagire con tale oggetto utilizzando i suoi metodi con la notazione *oggetto.metodo(p1,p2..)*:

```
Rectangle r;
r.set_values(2,3);
cout << r.area();
```

Se vogliamo possiamo anche unire la dichiarazione e l'implementazione dei metodi all'interno della dichiarazione della classe. Tale maniera di procedere è consigliabile solamente nei casi in cui il metodo sia molto breve e di facile comprensione.

19.2 Costruttori

Nell'esempio precedente, nella dichiarazione della classe Rectangle, abbiamo dovuto chiamare *set_values()* prima di *area()* per avere un esempio sensato. Resta comunque il fatto che se non avessimo invocato il primo metodo, chiamando direttamente *area()* avremmo ottenuto un ***valore indeterminato***. Ciò che facciamo in tali casi è quello di utilizzare il **costruttore**, che è una *funzione speciale* che viene **chiamata automaticamente quando viene *costruito* un oggetto** istanza di una determinata classe. Il costruttore deve essere pubblico e avrà lo ***stesso nome della classe***:

```
#include<iostream>
using namespace std;
class Rectangle {
    int width, height;
  public:
    Rectangle(int a, int b) {
        width = a;
        height = b;
    }
    int area() {return (width*height);}
};
```

In tal caso eliminiamo il metodo *set_values()* e inseriamo il costruttore in modo tale che ogni volta che costruiremo un oggetto di tipo Rectangle, esso non abbia valori indeterminati. Ovviamente in base a quanti argomenti vuole il costruttore come parametri, noi dobbiamo dare (in questo caso 2), poichè altrimenti avremmo un errore:

```
//Rectangle r; //darebbe errore: non stiamo specificando i due valori
Rectangle a(2,3);
Rectangle b(3,7);
cout << "Area di a: " << a.area() << endl;
cout << "Area di b: " << b.area() << endl;
```

19.2.1 Overloading dei costruttori

Può essere molto utile **definire più di un costruttore** per un oggetto. Utilizzando come esempio sempre la classe *Rectangle*, vogliamo ad esempio che il rettangolo venga costruito in questi modi:

- **Rettangolo vuoto**, quindi che ha ***lato uguale a 0***;
- **Rettangolo di lato** l, quindi che ha ***base e altezza uguali***, cioè un ***quadrato***;
- **Rettangolo di lati** h, w, cioè un normale rettangolo.

Possiamo costruire questi diversi oggetti utilizzando **prototipi diversi**. In fase di costruzione dell'oggetto verrà scelto, in base ai parametri inseriti e al tipo, quale costruttore verrà usato. Vediamo l'implementazione in codice:

```
#include<iostream>
using namespace std;
class Rectangle {
    int width, height;
  public:
    Rectangle() { //rettangolo "vuoto"
        width = 0;
        height = 0;
    }
    Rectangle(int l) { //quadrato di lato l
        width = l;
        height = l;
    }
    Rectangle(int w, int h) { //quadrato standard
        width = w;
        height = h;
    }
    int area() {return (width*height);}
};
```

19.2.2 Inizializzazione dei membri nei costruttori

In C++ è possibile inizializzare gli attributi di una classe tramite **inizializzazione diretta** utilizzando l'operatore (:) prima del corpo del costruttore. Vediamo degli esempi utilizzando sempre la classe Rectangle:

```
Rectangle::Rectangle(int w, int h) { // inizializzazione non diretta
    width = w;
    height = h;
}
//////////////
Rectangle::Rectangle(int w, int h) : width(h), height(h) {}
```

In questo caso l'effetto delle due implementazioni sarà il medesimo, ma ciò che accade effettivamente no, infatti:

- Nel primo caso gli attributi all'interno del costruttore verranno prima dichiarati e poi inizializzati. Ci sarà quindi un lasso di tempo in cui tali variabili avranno un valore indeterminato;
- Nel secondo caso invece la dichiarazione e inizializzazione degli attributi avviene **contestualmente alla costruzione dell'oggetto**, ovvero equivale a dire "int width = w; int height = h"

Tale inizializzazione è l'unica possibile nelle ***relazioni di composizione*** poichè sappiamo che l'oggetto contenuto all'interno del contenitore viene creato e distrutto insieme all'oggetto contenitore.

19.3 Metodi di accesso (getter) e metodi di cambiamento stato (setter)

Nell'esempio della classe *Rectangle*, gli attributi **width** ed **height** erano *privati*. Se avessimo scritto quindi le seguenti righe di codice avremmo ottenuto un **errore**:

```
Rectangle r;
cout << r.width << endl; //errore
cout << r.height << endl; //errore
```

L'errore sta nel fatto che non possiamo accedere dall'esterno a tali variabili poichè le abbiamo definite come private. Potremmo pensare di mettere tali variabili come ***public*** ma ciò non è una buona pratica per la stesura del codice. Per questo utilizziamo dei **metodi getter** per il quale **accediamo solo in lettura** a tali attributi:

```
#include<iostream>
using namespace std;
class Rectangle{
    int width, height;
  public:
    Rectangle(int w, int h) : width(w), height(h){}
    int getWidth() {return width;} // getter di width
    int getHeight() {return height;} // getter di height
    int area() {return (width*height);}
};
```

Per convenzione i metodi getter usano il *camel case* (getNomeAttributo()) e non richiedono ***nessun parametro*** in input. Se invece volessimo **accedere in scrittura** a tali attributi potremmo pensare di utilizzare dei **metodi setter**, che ci permettono di *sovrascrivere* i valori contenuti nelle variabili width ed height:

```
#include<iostream>
using namespace std;
class Rectangle {
    int width, height;
  public:
    Rectangle(int w, int h) : width(w), height(h){}
    int getWidth() {return width;}
    int getHeight() {return height;}
    void setWidth(int w) {width=w;} // setter width
    void setHeight(int h) {height=h;} // setter height
    int area() {return (width*height);}
};
```

Per convenzione, anche i metodi setter usano *camel case* (setWidth(int)) e hanno **void** come tipo di ritorno. Vediamo un esempio d'uso di getter e setter:

```
Rectangle r(2,3);
cout << r.area() << endl; // stampa "6"
r.setWidth(8);
cout << r.area() << endl; // stampa "24"
```

19.4 Puntatori a classi e array di oggetti

Come per i tipi primitivi è possibile utilizzare dei **puntatori ad oggetti**, come ad esempio:

```
RectangleV5 r(2,3);
RectangleV5 *r_ptr;
r_ptr = &r;
```

Inoltre è possibile definire un oggetto in **allocazione dinamica** nel seguente modo:

```
RectangleV2 *r1 = new RectangleV2;
RectangleV2 *r2 = new RectangleV2(); //analogo al precedente
RectangleV2 *r3 = new RectangleV2(2);
RectangleV2 *r4 = new RectangleV2(2,3);
```

Quando richiamiamo i membri di un oggetto mediante il puntatore, dobbiamo ricordarci di *de-referenziare*:

```
cout << (*r1).area() << endl;
cout << (*r2).area() << endl;
cout << (*r3).area() << endl;
cout << (*r4).area() << endl;
```

Alternativamente, possiamo utilizzare l'operatore $(->)$ direttamente sul puntatore anzichè deferenziare:

```
cout << r1->area() << endl;
cout << r2->area() << endl;
cout << r3->area() << endl;
cout << r4->area() << endl;
```

19.4.1 Array di oggetti

In certi casi può essere utile definire **collezioni di oggetti dello stesso tipo**, ovvero definire un **array di oggetti**. Prendiamo ad esempio la definizione della classe Punto2D:

```
#include<cmath>
#include<iostream>
using namespace std;
class Punto2D {
    double x, y;
    public:
        Punto2D(double _x, double _y): x(_x), y(_y) {}
        double distanzaDalCentro() {return sqrt(x*x+y*y);}
        void stampa() {cout << "("<<x<<","<<y<<")"<<endl;}
};
```

Se volessimo definire un ***array di oggetti di tipo Punto2D*** possiamo farlo nei seguenti modi:

```
Punto2D punti[3] = {Punto2D(2,3),Punto2D(4,5),Punto2D(7,7)};
for (int i=0; i<3; i++)
    punti[i].stampa();
    /////// oppure....
```

```
Punto2D punti[3] = {{2,3},{4,5},{7,7}};
for (int i=0; i<3; i++)
    punti[i].stampa();
```

Notiamo che insieme alla dichiarazione dell'array di punti abbiamo effettuato anche la costruzione degli oggetti mediante l'**inizializzazione unificata**. Questo è necessario in quanto l'oggetto Punto2D non ha un costruttore di default con zero parametri. Infatti, le seguenti righe di codice darebbero **errore**:

```
Punto2D punti[3]; //nessuna inizializzazioni
Punto2D punti[3] = {{2,3},{4,5}}; //troppo pochi inizializzatori
Punto2D punti[3] = {{2,3}}; //troppo pochi inizializzatori
```

Se voglio poter ***definire l'array senza una inizializzazione esplicita***, devo definire un costruttore di default nel seguente modo:

```
#include<cmath>
#include<iostream>
using namespace std;
class Punto2D {
    double x, y;
    public:
        Punto2D(): x(0), y(0) {} //costruttore con 0 parametri
        Punto2D(double _x, double _y): x(_x), y(_y) {}
        double distanzaDalCentro() {return sqrt(x*x+y*y);}
        void stampa() {cout << "("<<x<<","<<y<<")"<<endl;}
};

////////////////////

Punto2D punti[3]; // lecito
for (int i=0; i<3; i++)
    punti[i].stampa();
```

In alternativa, posso definire un ***array di oggetti allocati dinamicamente***:

```
Punto2D *punti[3]; //dichiaro un puntatore a un array di 3 oggetti di tipo punto2D
for(int i=0; i<3; i++)
    punti[i] = new Punto2D(i,i+1); // costruisco l'array dinamicamente
for(int i=0; i<3; i++)
    punti[i]->stampa(); //utilizziamo le istanze costruite
```

19.5 Definizione del corpo dei metodi

Abbiamo già visto che con l'operatore (::) è possibile definire un metodo di una determinata classe. Sappiamo che all'interno di una funzione è possibile inoltre **referenziare variabili di istanza della classe**, che di norma sono private. Vediamo un esempio:

```
class Moneta{
    private:
        char ValroreUltimoLancio;
        //....
}

bool Moneta::testa(){
return ValoreUltimoLancio;
}
```

E'importante quindi non confondersi nel **passaggio di parametri** tra il riferimento alle variabili locali o alle variabili di istanza. Vediamo che in questo caso abbiamo utilizzato una variabile privata nella definizione del metodo testa() come ritorno. Ora analizziamo il seguente codice:

```
class A{
 private:
    int a, b, c;
 public:
    void foo ( int a, int x ){
        a = 2 * x - a;
    }
}
```

Ci chiediamo in questo caso *a quale variabile si riferisce nel passaggio di parametri del metodo foo "a"?* In questo caso ci stiamo riferendo alla ***variabile locale***. Se volessimo riferirci alla variabile di istanza, dovremmo utilizzare l'operatore (− >) con la parola chiave **this** in questo modo:

```
class A{
 private:
    int a, b, c;
 public:
    void foo ( int a, int x ){
       this -> a = 2 * x - a;
    }
}
```

Ovviamente come nelle funzioni normali, utilizziamo la **return** per indicare la fine del corpo del metodo, che ovviamente dovrà essere congruente al ***tipo*** definito come uscita del metodo.

19.5.1 Passaggio di parametri

Per quanto riguarda il **passaggio di parametri** anche qui avviene in maniera molto simile alle *funzioni*, infatti può essere:

- **Passaggio per valore**: in tal caso verrà inserita una *copia della variabile sullo stack*, che verrà distrutta una volta terminata l'esecuzione del metodo;
- **Passaggio per riferimento**: in tal caso verrà passato un *riferimento* tramite l'operatore (&), ed è molto utile quando vogliamo passare un riferimento a un oggetto nel costruttore (relazione di composizione), oppure quando vogliamo passare array di oggetti molto grandi e ovviamente quando dobbiamo ***modificare in generale l'argomento passato***.
- **Passaggio per puntatore**: anche in questo caso possiamo modificare il valore del puntatore *deferenziando* lo stesso.

19.6 Argomenti funzione main

I programmi da riga di comando possono accettare in input uno o più argomenti. Questi vengono specificati dall'utente quando viene richiamato il programma da console. Quando gli argomenti sono specificati dall'utente, questi vengono resi disponibili come argomenti della funzione main. Vediamo un esempio:

```
#include<iostream>

using namespace std;

int main(int argc, char *argv[]) {
```

```
    //argc sara uguale al numero di argomenti specificati
    //argv[] e un vettore di stringhe (intese come char*)
    //contenenti i valori dei parametri

    //questo ciclo scorre e stampa gli argomenti
    for(int i=0; i<argc; i++)
        cout << "Argomento #"<<i<<": "<<argv[i]<<endl;
}
```

Capitolo 20

Reference in C++

L'uso dei puntatori permette una maggiore efficienza nell'utilizzo di strutture dati molto pesanti in termini di memoria. Se infatti pensiamo di avere una *funzione* che prenda come parametro un oggetto o di tipo *Oggetto* che occupi 3 GB, e lo passiamo per valore avremo che tale oggetto verrà copiato sullo stack, e contestualmente alla fine della funzione verrà deallocato:

```
bool fun(Oggetto o) {
    //operazioni su o
    return o.check();
}
```

Tale operazione è molto poco efficiente; sarebbe più efficiente un passaggio del puntatore all'oggetto:

```
bool fun(Oggetto *o) {
    //operazioni su o
    return o->check();
}
```

20.1 Limiti dei puntatori

Abbiamo visto nell'introduzione che ci sono molti vantaggi molto spesso nell'utilizzo dei puntatori; il problema si pone poi, però, sul fatto che lavorare con i puntatori può risultare molto tedioso per la notazione e si può verificare il **nullptr**:

```
Oggetto o;
Oggetto *o2;

o.metodo();
(*o2).metodo();
o->metodo();

o2 = nullptr;
o2->metodo(); //questo genera un errore a runtime!
```

20.2 Riferimenti

Uno strumento che mette a disposizione il C++, che ha i *pro* dei puntatori e non i *contro* (nullptr o notazione verbosa). Il concetto che sta alla base è che il riferimento rappresenta

un **alias** dell'oggetto: ovvero possiamo utilizzarlo al posto dell'oggetto stesso con la sintassi standard:

```
class Classe {
    int x;
    public:
        Classe() : x(0) {}
        Classe(int _x): x(_x) {}
        void setX(int _x) {x=_x;}
        int getX() {return x;}
};


int main(){
    Classe o;
    Classe &ref_o = o;
    ref_o.setX(12); //alias dell'oggetto
    cout << ref_o.getX();
}
```

Come possiamo vedere dal codice sopra, nel caso della reference, l'***dichiarazione*** *avviene* ***contestualmente all'inizializzazione***, quindi non può essere inizializzato a nullptr come i puntatori, e la sintassi è: $< tipo >$ & $nome\_reference$. Un altra *differenza* con i puntatori sta nel fatto che non può essere riassegnata la refernce a un nuovo oggetto dello stesso tipo poichè potrebbe portare dei risultati indesiderati nell'esecuzione:

```
ref_o = altro_oggetto; // risultato indesiderato
```

20.2.1 Riferimenti di sola lettura

I riferimenti visti fino ad adesso sono di tipo **lvalue**, ovvero possono essere utilizzati nel *lato sinistro* di un assegnamento e quindi possono modificare lo stato dell'oggetto. Come per i puntatori, può essere utile definire un riferimento "**di sola lettura**", che permetta di accedere in lettura al valore puntato (o allo stato dell'oggetto puntato), ma che non ne permetta l'accesso in scrittura. Per definire un riferimento di sola lettura, dobbiamo usare la keyword const:

```
int x = 8;
const int &r = x;
cout << r << endl;

r=2; //errore!
```

Lo stesso discorso vale per gli oggetti. In particolare i riferimenti di sola lettura possono essere utilizzati per quei metodi che **non cambiano lo stato dell'oggetto** in modo tale da non risultare fuorviante. Infatti prendiamo ad esempio un metodo ***getter*** che in teoria non dovrebbe modificare lo stato dell'oggetto, ma se facessimo la seguente operazione sarebbe comunque legittima:

```
void Classe::getX() {
    int z = x;
    x--;
    return z;
}
```

Per evitare tali problemi e per "comunicare" al compilatore che un dato metodo non modificherà lo stato dell'oggetto quando invocato, dobbiamo usare la keyword const (indica che il metodo è "di sola lettura") nel prototipo del metodo. Esempio:

```
class Classe {
    int x;
    public:
        Classe() : x(0) {}
        Classe(int _x): x(_x) {}
        void setX(int _x) {x=_x;}
        int getX() const {return x;} //inserisco "const" per indicare che il metodo non modifica lo
};
```

A questo punto, possiamo dichiarare una reference a un oggetto di classe Classe e usarla per invocare il metodo getX. L'invocazione di setX restituirebbe invece un errore:

```
int main(){
    Classe o(-7);
    const Classe &r=o;
    cout << r.getX();
    r.setX(-2); //errore!
}
```

20.2.2 Riferimenti come parametri formali

I riferimenti possono essere usati come parametri formali di funzioni e metodi. E' ciò che abbiamo chiamato in passato "passaggio per riferimento":

```
void fun(Classe &o) {
    o.setX(0);
}

int main(){
    Classe o(10);
    cout << o.getX() << endl;
    fun(o);
    cout << o.getX() << endl;
}
```

Quanto scritto sopra, è analogo al più esplicito:

```
void fun(Classe *o) {
    o->setX(0);
}

int main(){
    Classe o(10);
    cout << o.getX() << endl;
    fun(&o);
    cout << o.getX() << endl;
}
```

Capitolo 21

Modificatori *static* e *friend*

Vediamo in questo capitolo l'utilizzo dei modificatori **static** e **friend**, che sono molto utili soprattutto per gli oggetti

21.1 Modificatore static

La parola **static** può essere utilizzata per le cosiddette **variabili di classe**. Ricordiamo infatti che le variabili viste fino ad adesso sono **variabili di istanza**, ovvero che vengono allocate in una memoria "privata" del singolo oggetto e ognuno di questi oggetti possiede una **copia**. Le *variabili di classe* invece sono **condivise da tutte le istanze della data classe**. Vediamo un esempio:

```
class Classe {
    static int contatore; //variabile di istanza, condivisa tra tutti gli oggetti
    int x; //variabile di istanza: ogni oggetto avr  la sua copia
    public:
        Classe() : x(0) {contatore++;}
        int getX() const {return x;}
        void setX(int _x) {x=_x;}
        int getCount() {return contatore;}
};
```

Per quanto riguarda l'**inizializzazione della variabile statica**, essa deve avvenire **al di fuori della classe** ed è obbligatoria:

```
class Classe {
    static int contatore; //variabile di istanza, condivisa tra tutti gli oggetti
    int x; //variabile di istanza: ogni oggetto avr  la sua copia
    public:
        Classe() : x(0) {contatore++;}
        int getX() const {return x;}
        void setX(int _x) {x=_x;}
        int getCount() {return contatore;}
};
int Classe::contatore=0; //inizializzazione variabile statica
```

21.1.1 Metodi static

I **metodi static** sono dei metodi il cui funzionamento non dipende dallo stato dell'istanza. Ad esempio, il metodo *getCount* definito prima non dipende dallo stato, ma solo da una variabile

static, quindi può essere definito come static:

```
class Classe {
    static int contatore;
    int x;
    public:
        Classe() : x(0) {contatore++;}
        int getX() const {return x;}
        void setX(int _x) {x=_x;}
        static int getCount() {return contatore;} //metodo static
};
int Classe::contatore =0;
```

21.2 Modificatore friend

Abbiamo visto che tutti metodi e le proprietà dichiarate come "private" all'interno di un oggetto **non possono essere accedute dall'esterno.**. Consideriamo ad esempio la classe:

```
class Point2D {
    private:
        double x, y;
    public:
        Point2D() : x(0), y(0) {}
        Point2D(double _x, double _y) : x(_x), y(_y) {}
        double getX() const {return x;}
        double getY() const {return y;}
};
```

Per modificare le variabili privati della classe punto, non avendo metodi *setter*, avrei un errore. Infatti non posso fare i seguenti tentativi:

```
p1.x=0; //errore
p1.y=0; //errore
```

Supponiamo ad esempio di voler creare un metodo che ***normalizzi*** un vettore, ovvero la *radice quadrata del prodotto scalare di un vettore con se stesso*, ovvero creare la seguente funzione:

```
void normalize(Point2D &p) {
    double mod = sqrt(p.x*p.x + p.y*p.y);
    p.x/=mod;
    p.y/=mod;
}
```

Per far si che sia possibile l'accesso ai membri privati della classe *Point2D*, possiamo dichiarare la funzione come **friend**, ovvero "amica" in fase di definizione della classe in questo modo:

```
class Point2D {
    private:
        double x, y;
    public:
        friend void normalize(Point2D &p);
        Point2D() : x(0), y(0) {}
        Point2D(double _x, double _y) : x(_x), y(_y) {}
        double getX() const {return x;}
        double getY() const {return y;}
};
```

Una volta dichiarata **friend**, possiamo definire la funzione fuori dalla classe.

Capitolo 22

Namespace e overloading dei metodi

22.1 Namespace

I **namespace** in C++ permettono di definire dei "contenitori" per i simboli (nomi di variabili, funzioni, classi). Lo scopo dei namespace è quello di evitare il *name clash*, ovvero evitare che diverse librerie definiscano funzioni con nomi uguali. Finora abbiamo visto l'utilizzo del namespace std, ma altre librerie ne mettono a disposizione altri, ed è possibile definire il proprio namespace. Vediamo come:

```
namespace geometric {
    class Rectangle {
        double base;
        double height;
        public:
            Rectangle(double b, double h) :base(b), height(h) {}
            double area() {return base*height;}
    };

    class Square {
        double side;
        public:
            Square(double s) : side(s) {}
            double area() {return side*side;}
    };

    class Circle {
        double r;
        public:
            Circle(double _r): r(_r) {}
            double area() {return 3.14*r*r;}
    };
}
```

Una volta definite le classi nel namespace, posso accedervi in diversi modi. Usando il **selettore di scope**:

```
geometric::Square s(2);
cout << s.area();
```

Possiamo importare il simbolo **Square**:

```
using geometric::Square;
Square s(2);
cout << s.area();
```

Possiamo importare l'intero namespace:

```
using namespace geometric;
Square s(2);
cout << s.area();
```

22.2 Overloading di metodi

C++ permette di effettuare l'**overloading di funzioni e metodi**, ovvero di definire *diverse "versioni" di una funzione/metodo* che hanno lo **stesso nome** e lo **stesso tipo di ritorno**, ma **prendono in input numeri diversi di parametri con tipi diversi**. Lo scopo è quello di semplificare l'invocazione dei metodi. Consideriamo ad esempio la seguente funzione che concatena due stringhe:

```
string concat(string s1, string s2) {
    return s1+s2;
}
```

Supponiamo adesso di voler concatenare una stringa a un numero. Non posso usare *concat* poichè nella *signature* della funzione non è previsto che io possa passare una stringa e un numero. Posso però, attraverso l'overloading dei metodi, definire un'altra funzione *concat*, oltre a quella vista in precedenza, che preveda come parametri in input una *string* e un *int*, ad esempio in questo modo:

```
string concat(string s1, int x) {
    stringstream ss;
    ss << s1 << x;
    return ss.str();
}
```

Potrei anche definire una funzione con un **numero diverso di parametri**:

```
string concat(string s1, string s2, string s3) {
    return s1 + s2 + s3;
}
```

Capitolo 23

Copia e distruzione di un oggetto

Vediamo in questo capitolo di affrontare quello che è il **ciclo di vita di un oggetto**, che può essere suddiviso in 3 fasi:

- **Creazione** dell'oggetto;
- **Copia** dell'oggetto;
- **Distruzione** dell'oggetto

La fase di creazione dell'oggetto è stata già affrontata quando abbiamo parlato del metodo **costruttore** (sezione 19.2), che è appunto quello che viene invocato sempre alla creazione di un oggetto. Vediamo adesso le fasi di copia e distruzione.

23.1 Copia di un oggetto

Quando un oggetto viene assegnato a un altro oggetto, C++ opera una copia "member-wise" dello stato. In pratica **il valore di ciascun attributo viene copiato nel nuovo oggetto**. Vediamo un esempio:

```
class Point2D {
    double x, y;
    public:
        Point2D(int a, int b) : x(a), y(b) {}
        double getX() const {return x;}
        double getY() const {return y;}
};

Point2D p1(3,2);
Point2D p2(4,5);

cout << p1.getX() << ", "<<p1.getY() << endl; // 3,2
cout << p2.getX() << ", "<<p2.getY() << endl; // 4,5

p2=p1; //copia automatica membro a membro di C++
cout << p2.getX() << ", "<<p2.getY() << endl; // 3,2
```

23.1.1 Costruttore di copia

C++ permette inoltre di specificare un **costruttore di copia**, nel quale si specifica esattamente cosa fare quando la copia è richiesta in **fase di inizializzazione dell'oggetto**. Tale costruttore si definirà con codice *custom* (personalizzato per la copia) e dovremo passare un **riferimento all'oggetto dello stesso tipo**. Vediamo un esempio:

```
class Point2D {
    double x, y;
    public:
        Point2D(Point2D &p) :x(p.x), y(p.y) { //costruttore di copia
            //codice custom
            cout << "costruttore di copia"<<endl;
        }
        Point2D(int a, int b) : x(a), y(b) {}
        double getX() const {return x;}
        double getY() const {return y;}
};
```

Vediamo alcuni esempi di utilizzo:

```
Point2D p1(2,3);
Point2D p2 = p1; //richiama il costruttore di copia
Point2D p3(p1); //richiama il costruttore di copia

//non richiama il costruttore di copia, ma fa una copia membro a membro
p3 = p2;
```

23.2 Distruzione di un oggetto

Gli oggetti allocati dinamicamente hanno un loro naturale ciclo di vita, così come visto per i tipi primitivi. Ad esempio, un oggetto costruito mediante una dichiarazione del tipo Classe o; verrà in automatico "**distrutto**" quando termina l'esecuzione del blocco di codice o della funzione che lo definisce. Se invece l'oggetto è instanziato dinamicamente, ad esempio mediante Classe *o = new Classe();, sarà necessario utilizzare l'**istruzione delete** o; per distruggerlo. Quando un oggetto viene distrutto, in automatico vengono *deallocati tutti gli attributi dichiarati staticamente.* Se però abbiamo allocato della memoria dinamicamente durante la costruzione o l'interazione con l'oggetto, la memoria allocata va liberata manualmente specificando uno speciale metodo chiamato **distruttore** che ha lo stesso nome della classe preceduto dalla tilde . Vediamo un esempio:

```
class Classe {
    int *x;
    int n;
    public:
        Classe(int _n) : n(_n) {
            x = new int[n];
            for(int i=0; i<n; i++)
                x[i] = i+1;
        }
        int getNum(int i) {return x[i];}
};

int main(){
    Classe *o = new Classe(3);
    delete o; //errore
}
```

Nell'esempio sopra, il costruttore *alloca dinamicamente della memoria* per un array di n interi e fa sì che x punti a tale memoria. Alla fine del ciclo vita dell'oggetto, perderemo il puntatore x, ma ***la memoria da esso puntato non sarà liberata***. Il codice del main di questo tipo, quindi, genererà dunque un **memory leak**. Per questo in C++ si definisce un metodo chiamato **distruttore** che viene richiamato automaticamente quando l'oggetto viene distrutto con la delete, se allocato dinamicamente, oppure automaticamente, se allocato nello stack. Il distruttore va dichiarato come un costruttore senza parametri, ma anteponendo al nome la tilde ( ):

```
class Classe {
    int *x;
    int n;
    public:
        Classe(int _n) : n(_n) {
            x = new int[n];
            for(int i=0; i<n; i++)
                x[i] = i+1;
        }
        int getNum(int i) {return x[i];}
        ~Classe() { //distruttore
            delete[] x;
            cout << "Eseguito il distruttore";
        }
};

int main(){
    Classe *o = new Classe(3);
    delete o; //corretto
}
```

Adesso possiamo notare che la delete del main sarà effettuata correttamente.

Capitolo 24

Overloading di operatori

L'**overloading di operatori** permette di abilitare l'uso degli operatori disponibili in C++ per tipi diversi rispetto a quelli di default. Conosciamo ad esempio l'operatore "+" per le stringhe che fa un *operazione di concatenamento tra stringhe*:

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

string s1 = "hello";
string s2 = "world";
string s3 = s1 + " " + s2;
cout << s3; // hello world
```

Così come nel caso dell'overloading di metodi e funzioni, questa pratica permette di definire l'uso dell'operatore quando cambiano i tipi degli operandi. Ad esempio, esiste un'altro operatore overloaded per concatenare una stringa a un carattere:

```
string s4 = "hello ";
string s5 = s4 + 'w';
cout << s5; // hello w
```

Un operatore può essere definito come una ***funzione membro*** oppure come ***funzione non membro***

24.1 Overloading con funzioni non membro

Vediamo inizialmente come fare **overloading con funzioni non membro**. Consideriamo dunque la seguente classe Point2D:

```
class Point2D {
    double x, y;
    public:
        Point2D(double _x, double _y) : x(_x), y(_y) {}
        Point2D(): x(0), y(0) {}
        double getX() {return x;}
        double getY() {return y;}
};
```

Potremmo pensare di fare una somma tra due punti costruendo un metodo *sum*, come nel seguente modo:

```
class Point2D {
    double x, y;
    public:
        Point2D(double _x, double _y) : x(_x), y(_y) {}
        Point2D(): x(0), y(0) {}
        double getX() {return x;}
        double getY() {return y;}
        Point2D sum(Point2D p) { // metodo sum
            return Point2D(x+p.x, y+p.y);
        }
};
```

Per semplificare la sintassi però facciamo overloading dell'operatore come segue:

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

class Point2D {
    double x, y;
    public:
        Point2D(double _x, double _y) : x(_x), y(_y) {}
        Point2D(): x(0), y(0) {}
        double getX() {return x;}
        double getY() {return y;}
};

Point2D operator+(Point2D a, Point2D b){ // overloading operatore +
    return Point2D(a.getX()+b.getX(), a.getY()+b.getY());
}

int main() {
    Point2D p1(2,3);
    Point2D p2(3,3);

    Point2D p3 = p1+p2;
    cout << "("<<p3.getX() << "," << p3.getY() << ")" << endl;
}
```

Notiamo che viene utilizzata la parola chiave **operator**, che appunto indica che stiamo facendo overloading di tale operatore indicato. Se vogliamo ancora semplificare la sintassi, possiamo dichiarare la funzione che fa overloading dell'operatore come "**friend**". In questo modo, la funzione può accedere allo stato interno degli oggetti di tipo Point2D:

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

class Point2D {
    double x, y;
    public:
        Point2D(double _x, double _y) : x(_x), y(_y) {}
        Point2D(): x(0), y(0) {}
        double getX() {return x;}
        double getY() {return y;}
        friend Point2D operator+(Point2D, Point2D); // utilizzo il modificatore friend
};

Point2D operator+(Point2D a, Point2D b){
    return Point2D(a.x+b.x, a.y+b.y); // poich  la funzione    friend accedo allo stato
}
```

```
int main() {
    Point2D p1(2,3);
    Point2D p2(3,3);

    Point2D p3 = p1+p2;
    cout << "("<<p3.getX() << "," << p3.getY() << ")" << endl;
}
```

Possiamo anche vedere un implementazione di un **operatore unario**, ovvero che prende un solo parametro in input, come ad esempio l'operatore "-":

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

class Point2D {
    double x, y;
    public:
        Point2D(double _x, double _y) : x(_x), y(_y) {}
        Point2D(): x(0), y(0) {}
        double getX() {return x;}
        double getY() {return y;}
        friend Point2D operator+(Point2D, Point2D);
        friend Point2D operator-(Point2D); // operatore unario -
};

Point2D operator+(Point2D a, Point2D b){
    return Point2D(a.x+b.x, a.y+b.y);
}

Point2D operator-(Point2D a) {
    return Point2D(-a.x,-a.y);
}

int main() {
    Point2D p1(2,3);
    Point2D p2(3,3);

    Point2D p3 = -(p1+p2);
    cout << "("<<p3.getX() << "," << p3.getY() << ")" << endl;
}
```

Possiamo anche effettuare una **promozione di uno scalare a un tipo della classe**, in modo tale da poter fare anche operazioni tra un tipo Point2D e uno scalare ad esempio, con l'overloading del costruttore:

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

class Point2D {
    double x, y;
    public:
        Point2D(double _x, double _y) : x(_x), y(_y) {}
        Point2D(): x(0), y(0) {} // costruttore apposito per promuovere uno scalare a Point2D
        Point2D(double _x) : x(_x), y(_x) {}
        double getX() {return x;}
        double getY() {return y;}
        friend Point2D operator+(Point2D, Point2D);
};
```

```
Point2D operator+(Point2D a, Point2D b){
    return Point2D(a.x+b.x, a.y+b.y);
}

int main() {
    Point2D p1(2,3);
    Point2D p2 = p1+4; //equivalente a Point2D p2 = p1 + Point2D(4);
    cout << "("<<p2.getX() << "," << p2.getY() << ")" << endl;
}
```

24.2 Overloading con funzioni membro

In alternativa, è possibile effettuare l'**overloading degli operatori mediante funzioni membro**. In questo caso, dovremo passare in input solo l'operando di sinistra nel caso di operatori binari o nessun operando nel caso di operatori unari. Ad esempio:

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

class Point2D {
    double x, y;
    public:
        Point2D(double _x, double _y) : x(_x), y(_y) {}
        Point2D(): x(0), y(0) {}
        Point2D(double _x) : x(_x), y(_x) {}
        double getX() {return x;}
        double getY() {return y;}
        Point2D operator+(Point2D a){ // un solo parametro in input
            return Point2D(a.x+x, a.y+y);
        }
        Point2D operator-() { // 0 parametri in input
            return Point2D(-x,-y);
        }
};

int main() {
    Point2D p1(2,3);
    Point2D p2(1,2);
    Point2D p3 = -(p1 + p2);
    cout << "("<<p3.getX() << "," << p3.getY() << ")" << endl;
}
```

In questo caso però abbiamo una limitazione in più. Vediamo ad esempio le seguenti righe di codice:

```
Point2D p3 = p1 +2; //promozione di 2 a Point2D permessa
Point2D p3 = 2 + p1; //promozione non permessa! l'operatore non    membro di "2"
```

Infatti ricordiamo di aver detto che dovevamo passare, nel caso di operatore binario, l'operatore di sinistra, lo scalare "2" non è definito come membro quindi non può essere fatta la promozione

24.3 Operatori speciali

Vediamo adesso una serie di **operatori speciali** con i quali è possibile fare overloading

24.3.1 Operatore <<

L'overloading dell'operatore ¡¡ permette di "stampare" facilmente oggetti a schermo. Ricordiamo che la stampa avviene così: cout ¡¡ obj ¡¡ obj ¡¡obj ¡¡ ... Dove obj rappresenta un oggetto (es. una stringa). L'operatore ¡¡ è dunque un **operatore binario** che ha due operandi:

- Un **ostream** (ad esempio cout);
- Un **oggetto** di un determinato tipo

Per mantenere l'***associatività***, l'operatore restituisce un riferimento a ostream. Infatti, la catena sopra può essere letta come: (cout << obj) << obj ... Dove (cout << obj) restituisce un **riferimento a cout**. Se vogliamo abilitare l'operatore ¡¡ nel nostro oggetto, dobbiamo implementare l'*operatore come funzione* ***friend*** (non possiamo implementarlo come membro perché il primo operando non è del tipo di obj). Vediamo un esempio:

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

class Point2D {
    double x, y;
    public:
        Point2D(double _x, double _y) : x(_x), y(_y) {}
        double getX() {return x;}
        double getY() {return y;}
        friend ostream& operator<<(ostream &, Point2D);
};

ostream& operator<<(ostream &s, Point2D p) {
    // inserisco una rapresentazione di p nel riferimento a ostream
    s << "(" << p.x << "," << p.y << ")";
    return s; //restituisco il riferimento
}

int main() {
    Point2D p(2,3);
    cout << p;
}
```