

# *Appunti di Informatica (Un po' meno bozza)*

Prof. Bortolai Gabriele  
gabriele.bortolai@iovallescrivio.edu.it

10 dicembre 2021



*Omnibus discipulis*



# Indice

<b>1</b>	<b>C++</b>	<b>8</b>
1.1	Introduzione . . . . .	8
1.2	Variabili . . . . .	17
1.3	Operatori logici . . . . .	18
1.4	Interazione macchina utente . . . . .	19
1.5	Controllo di flusso . . . . .	20
1.6	Cicli . . . . .	21
1.7	Array . . . . .	24
1.7.1	Array multidimensionali . . . . .	26
1.8	Puntatori . . . . .	27
1.9	Stream da file . . . . .	30
1.10	Allocazione dinamica . . . . .	32
1.11	Funzioni . . . . .	37
1.12	Librerie . . . . .	39
1.13	ROOT . . . . .	39
1.14	Programmi . . . . .	40
1.14.1	Primo Programma . . . . .	40
1.14.2	Calcolo dell'area . . . . .	41
1.14.3	Equazione di secondo grado . . . . .	42
1.14.4	Somma . . . . .	45
1.14.5	Array costante . . . . .	47
1.14.6	Array a dimensione non definita inizialmente . . . . .	48
1.14.7	Array multidimensionali . . . . .	49
1.14.8	Cramer . . . . .	51
1.14.9	Scrittura su file . . . . .	53
1.14.10	Allocazione dinamica . . . . .	54
1.14.11	Funzioni . . . . .	54
1.14.12	Librerie . . . . .	54
1.15	Esercizi . . . . .	55
1.15.1	Esercizio . . . . .	55
1.15.2	Esercizio . . . . .	55
1.15.3	Esercizio . . . . .	55
1.15.4	Esercizio . . . . .	55
1.15.5	Esercizio . . . . .	55
1.15.6	Esercizio . . . . .	55
1.15.7	Esercizio . . . . .	55
1.16	Soluzioni . . . . .	57
1.16.1	Esercizio 1 . . . . .	57
1.16.2	Esercizio 2 . . . . .	57
1.16.3	Esercizio 3 . . . . .	57
1.16.4	Esercizio 4 . . . . .	57
1.16.5	Esercizio 5 . . . . .	57

<b>2</b>	<b>Python</b>	<b>57</b>
2.1	Introduzione . . . . .	57
2.2	Variabili . . . . .	57
2.3	Operatori logici . . . . .	57
2.4	Controlli di flusso . . . . .	57
2.5	Cicli . . . . .	57
2.6	Array . . . . .	57
2.6.1	Array multidimensionali . . . . .	57
2.7	Funzioni . . . . .	57
2.8	Librerie forse . . . . .	57
2.9	Programmi . . . . .	57
2.10	Qiskit . . . . .	57
2.10.1	Introduzione . . . . .	57
2.10.2	Quantum Lab and Quantum Composer . . . . .	57
2.10.3	Implementazione circuito quantistico . . . . .	57
2.10.4	Simulator . . . . .	57
2.10.5	Measure circuit . . . . .	57
2.10.6	Real Simulation . . . . .	57
2.10.7	Schrödinger's cat . . . . .	57
<b>3</b>	<b>Quantum Computing</b>	<b>57</b>
3.1	Introduzione . . . . .	57
3.1.1	Qubits . . . . .	57
3.2	Quantum information . . . . .	57
3.2.1	Quantum Gates . . . . .	57
3.2.2	Quantum parallelism . . . . .	57
3.2.3	Deutsch-Jozsa Algorithm . . . . .	57
3.3	Theorem of no cloning . . . . .	57
3.3.1	Quantum security . . . . .	57
3.4	Quantum transmission of information . . . . .	57
3.4.1	Super dense coding . . . . .	57
3.4.2	Quantum teleportation . . . . .	57
3.5	Quantum Errors . . . . .	57
3.5.1	Bit flip . . . . .	57
3.5.2	Phase flip . . . . .	57
3.5.3	Little errors . . . . .	57
3.5.4	Shore code correction . . . . .	57
3.6	Quantum games? . . . . .	57
<b>4</b>	<b>Integrali Indefiniti</b>	<b>57</b>
4.1	Introduzione . . . . .	57
4.2	Proprietà degli integrali . . . . .	57
4.3	Regole di integrazione . . . . .	57
4.3.1	Polinomi . . . . .	57
4.3.2	Integrazione per parti . . . . .	57
4.3.3	Integrazione per sostituzione . . . . .	57

4.3.4	Integrazione delle funzioni razionali . . . . .	57
4.3.5	Integrazione di alcune funzioni trascendenti . . . . .	57
4.4	Teorema di Liouville . . . . .	57
4.5	Esercizi . . . . .	57
<b>5</b>	<b>Integrali Definiti</b>	<b>57</b>
5.1	Integrale definito di una funzione a scala . . . . .	57
5.2	Teorema della media integrale . . . . .	57
5.3	Teorema fondamentale del calcolo integrale . . . . .	57
5.3.1	Corollario . . . . .	57
5.4	interpretazione geometrica . . . . .	57
5.5	Esercizi . . . . .	57
<b>6</b>	<b>Integrali Impropri</b>	<b>57</b>
<b>7</b>	<b>Bibliografia</b>	<b>58</b>

# 1 C++

## 1.1 Introduzione

Il C++ é un linguaggio di programmazione compilato, ovvero per rendere eseguibile il programma dal calcolatore serve un compilatore che si occupi di tradurre il codice da alto livello a basso livello.

I programmi compilati sono in genere più veloci ed ottimizzati ma vanno ricompilati su ogni nuova macchina per essere utilizzati.

In particolare il C++ é molto formale ed estremamente stabile e viene spesso usato in ambito scientifico insieme al C.

Esistono molti modi per programmare la più semplice in assoluto é l'interfaccia IDE ma rende l'utente dipendente dall'ambiente di lavoro ( MacOS, Windows o Linux ), invece l'utilizzo della shell rende l'utente capace di lavorare su ogni macchina indipendentemente dalla sua architettura.

Per prima cosa bisogna creare un file, detto sorgente, con l'estensione .cpp, supponiamo che il programma si chiami prova.cpp.

Una volta creato il file .cpp bisogna aprirlo con un editor per poter scrivere il codice, supponendo di lavorare su MacOS e di aver già in stallato Emacs come editor, dal terminale lanciare il seguente comando:

```
open -a Emacs prova.cpp
```

Con il precedente comando si aprirà il file sorgente con Emacs e a quel punto sarà possibile cominciare a scrivere il codice.

Un file sorgente é diviso in due parti, la prima il preambolo nel quale per prima cosa bisogna includere le librerie che verranno usate e i prototipi delle funzioni, la seconda parte invece é il *main()* dove invece si scriverà il codice, segue un esempio:

```
#include <name_lib>
#include <name_lib.h>

using namespace std;

int main(){

    ...

}
```

Una volta scritto il codice bisogna compilare il file sorgente, per fare ciò basta lanciare il seguente comando nella stessa cartella in cui é presente il file prova.cpp:

```
g++ -o name_exe prova.cpp
```



Dove *name\_exe* é il nome del file eseguibile, é consigliato assegnarli lo stesso nome del file sorgente senza l'estensione, se la compilazione é andata a buon fine si può avviare l'eseguibile digitando il seguente comando:

*./name\_exe*

Il terminale può anche essere usato per navigare all'interno del nostro computer ed interagire con esso, supponiamo di aver aperto il terminale per prima cosa bisogna capire dove é stato aperto per farlo basta lanciare il seguente comando:

`pwd`

Il comando restituirà la posizione dell'utente, per esempio su Ubuntu l'output sarà:

`/home/name_utente`

Mentre su MacOS:

`/Users/name_utente`

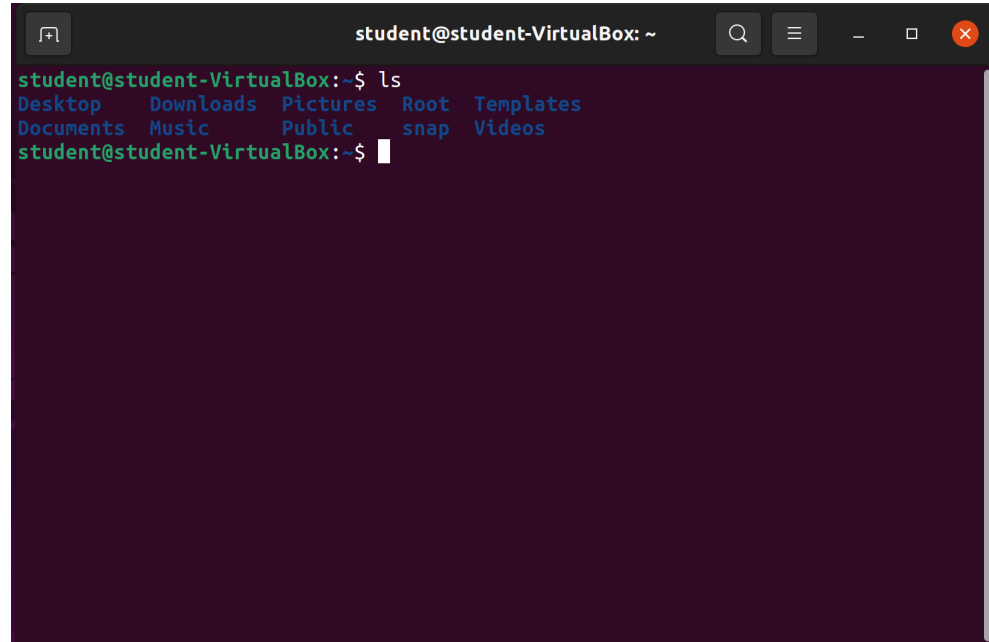
Nota la posizione in cui ci si trova saremo interessanti a veder il contenuto della cartella, il comando é il seguente:

`ls`

Se invece si vogliono maggiori informazioni e il contenuto venga elencato in verticale il comando é il seguente:

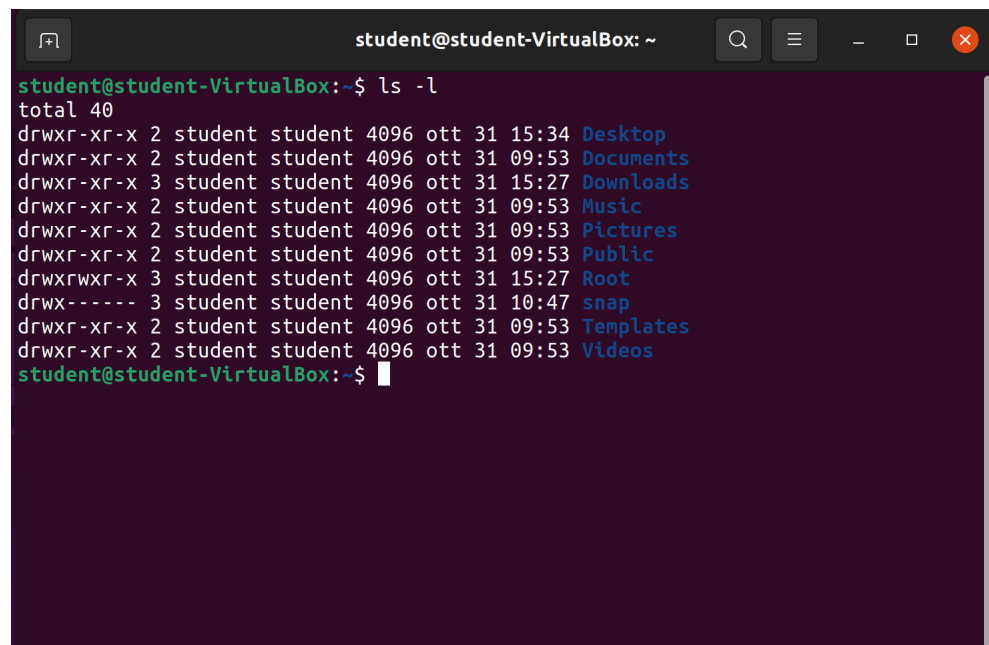
`ls -l`

Supponendo di lanciare il comando su Ubuntu l'output sarà:

A terminal window titled 'student@student-VirtualBox: ~' with a search icon, a menu icon, and window control buttons. The prompt is 'student@student-VirtualBox:~\$'. The command 'ls' has been entered, and the output is displayed in two rows: 'Desktop Downloads Pictures Root Templates' and 'Documents Music Public snap Videos'. The prompt 'student@student-VirtualBox:~\$' is followed by a cursor.

```
student@student-VirtualBox:~$ ls
Desktop  Downloads  Pictures  Root  Templates
Documents Music      Public   snap  Videos
student@student-VirtualBox:~$
```

Mentre lanciando il comando con l'opzione -l l'uscita sarà:

A terminal window titled 'student@student-VirtualBox: ~' with a search icon, a menu icon, and window control buttons. The prompt is 'student@student-VirtualBox:~\$'. The command 'ls -l' has been entered, and the output is displayed in a long list format. It starts with 'total 40' and then lists ten directories with their permissions, owner, group, size, date, and name. The prompt 'student@student-VirtualBox:~\$' is followed by a cursor.

```
student@student-VirtualBox:~$ ls -l
total 40
drwxr-xr-x 2 student student 4096 ott 31 15:34 Desktop
drwxr-xr-x 2 student student 4096 ott 31 09:53 Documents
drwxr-xr-x 3 student student 4096 ott 31 15:27 Downloads
drwxr-xr-x 2 student student 4096 ott 31 09:53 Music
drwxr-xr-x 2 student student 4096 ott 31 09:53 Pictures
drwxr-xr-x 2 student student 4096 ott 31 09:53 Public
drwxrwxr-x 3 student student 4096 ott 31 15:27 Root
drwx----- 3 student student 4096 ott 31 10:47 snap
drwxr-xr-x 2 student student 4096 ott 31 09:53 Templates
drwxr-xr-x 2 student student 4096 ott 31 09:53 Videos
student@student-VirtualBox:~$
```

Ripetendo i comandi precedenti su MacOS si ottiene:

```
gabrielebortolai@Air-di-Gabriele ~ % ls
Applications  Downloads  Music      Public      238 KB  PDF Document
Desktop        Library    Pictures   VirtualBox  277 KB  PDF Document
Documents      Movies     Prova      VMs         207 KB  PDF Document
               hello-world
```

Aggiungendo l'opzione -l si ottiene:

```
gabrielebortolai@Air-di-Gabriele ~ % ls -l
total 0
drwx-----@  5 gabrielebortolai  staff  160 Jun  2  2020 Applications
drwx-----@ 15 gabrielebortolai  staff  480 Nov  7 11:29 Desktop
drwx-----@ 19 gabrielebortolai  staff  608 Nov  2 18:06 Documents
drwx-----@ 100 gabrielebortolai staff 3200 Nov  5 14:45 Downloads
drwx-----@ 90 gabrielebortolai  staff 2880 Oct 30 20:10 Library
drwx-----+  7 gabrielebortolai  staff  224 Aug 24  2020 Movies
drwx-----+  6 gabrielebortolai  staff  192 Jul 16  2020 Music
drwx-----+ 30 gabrielebortolai  staff  960 Jul  5 09:39 Pictures
drwxr-xr-x  2 gabrielebortolai  staff   64 Oct 27 10:00 Prova
drwxrwxr-x+  5 gabrielebortolai  staff  160 Nov  2 18:06 Public
drwxr-xr-x  4 gabrielebortolai  staff  128 Oct 31 17:30 VirtualBox VMs
drwxr-xr-x  6 gabrielebortolai  staff  192 Jul 24 11:59 hello-world
```

Per entrare in una cartella il comando é:

cd name\_dir

Supponendo di usare Ubuntu, si vuole entrare nella cartella Documenti il comando da lanciare é il seguente:

cd Documents

Lanciando nuovamente il comando di locazione, l'uscita sarà:

```
/home/name_utente/Documents
```

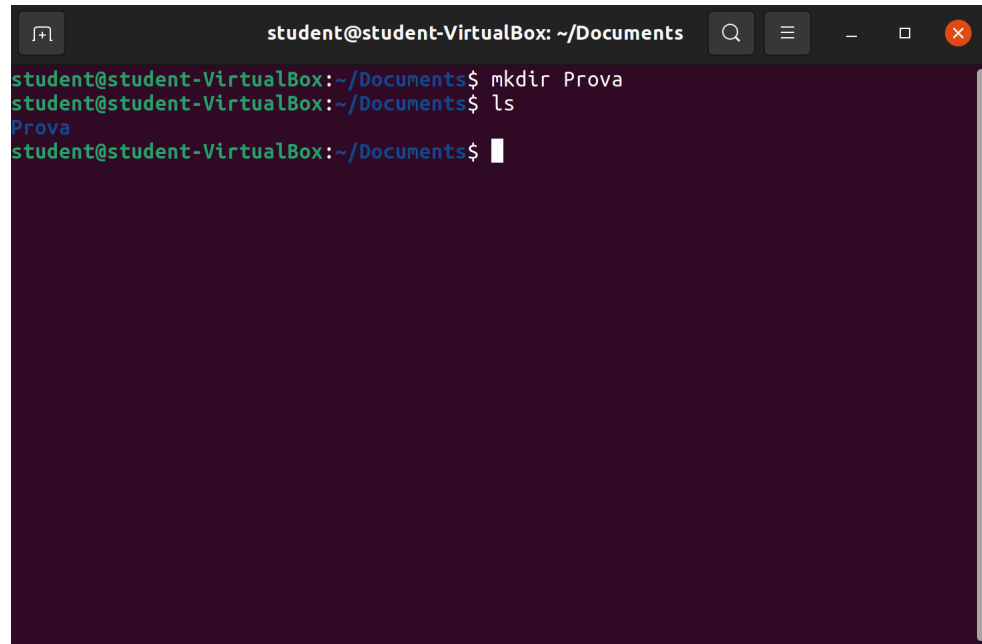
Per tornare nella cartella precedente il comando da lanciare é il seguente:

```
cd ..
```

Per creare una nuova cartella basta lanciare il seguente comando:

```
mkdir name_dir
```

Per esempio su Ubuntu nella cartella Documents é stata creata una nuova cartella chiamata Prova:

A screenshot of a terminal window titled 'student@student-VirtualBox: ~/Documents'. The terminal shows the following commands and output: 'mkdir Prova' is executed, followed by 'ls' which outputs 'Prova'. The prompt is then ready for the next command.

```
student@student-VirtualBox: ~/Documents
student@student-VirtualBox:~/Documents$ mkdir Prova
student@student-VirtualBox:~/Documents$ ls
Prova
student@student-VirtualBox:~/Documents$
```

Per cancellare, senza passare dal Bin, una cartella il comando é il seguente:

```
rm -r name_dir
```

Invece per creare un file il comando é il seguente:

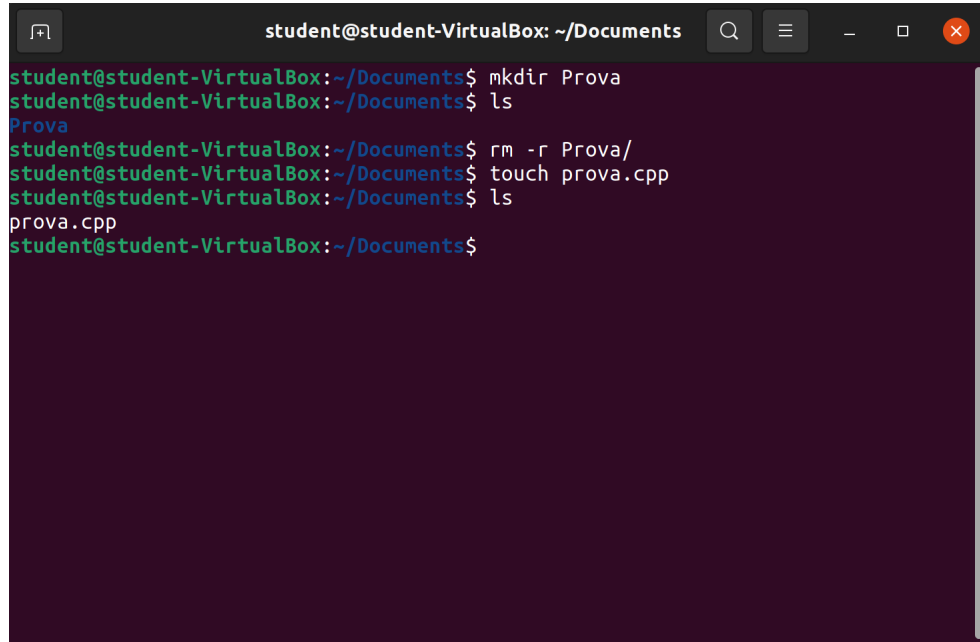
```
touch name_file
```

Supponiamo di voler creare il file per programmare in C++, il file lo chiameremo prova e dovrà avere l'estensione .cpp.

Il comando che dovremo lanciare sarà:

```
touch prova.cpp
```

Sempre da Ubuntu il risultato sarà il seguente:

A terminal window titled 'student@student-VirtualBox: ~/Documents'. The window has a dark background with light-colored text. The terminal shows the following commands and output: 'mkdir Prova' (no output), 'ls' (output: 'Prova'), 'rm -r Prova/' (no output), 'touch prova.cpp' (no output), and 'ls' (output: 'prova.cpp'). The prompt 'student@student-VirtualBox:~/Documents\$' is visible at the end of each line.

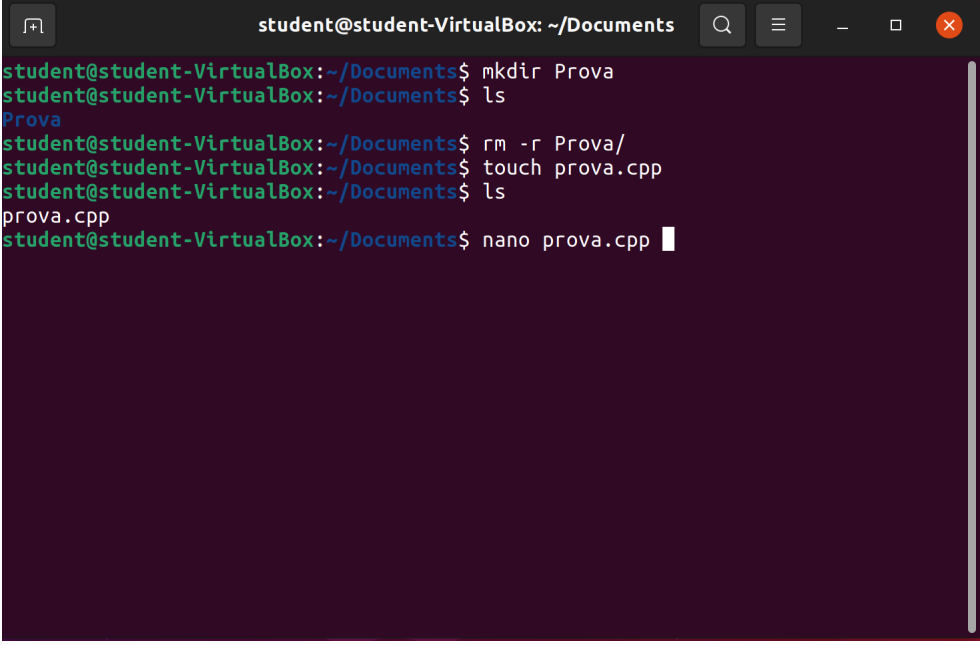
```
student@student-VirtualBox:~/Documents$ mkdir Prova
student@student-VirtualBox:~/Documents$ ls
Prova
student@student-VirtualBox:~/Documents$ rm -r Prova/
student@student-VirtualBox:~/Documents$ touch prova.cpp
student@student-VirtualBox:~/Documents$ ls
prova.cpp
student@student-VirtualBox:~/Documents$
```

Per scrivere all'interno del file lo si può fare in due modi diversi, il primo modo é utilizzare l'editor tramite il terminale, il secondo é utilizzare l'editor mediante l'interfaccia grafica.

Il primo metodo é comodo da utilizzare se bisogna scrivere poche righe di codice oppure apportare piccole modifiche, per fare ciò ci serviremo dell'editor *nano*, per aprire un file con *nano* bisogna utilizzare il seguente comando:

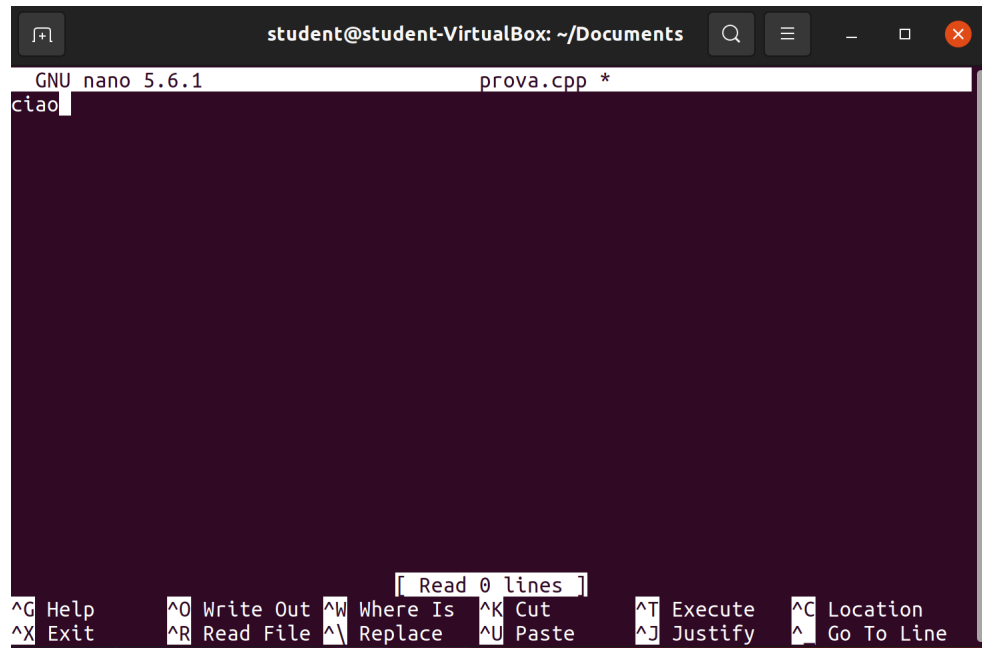
nano name\_file

Applicando quanto detto in precedenza al file *prova.cpp*, utilizzando Ubuntu:



```
student@student-VirtualBox: ~/Documents
student@student-VirtualBox:~/Documents$ mkdir Prova
student@student-VirtualBox:~/Documents$ ls
Prova
student@student-VirtualBox:~/Documents$ rm -r Prova/
student@student-VirtualBox:~/Documents$ touch prova.cpp
student@student-VirtualBox:~/Documents$ ls
prova.cpp
student@student-VirtualBox:~/Documents$ nano prova.cpp
```

Una volta dato il comando si aprirà la seguente finestra in cui sarà possibile scrivere all'interno del file, nell'esempio che segue è stata scritta la parola ciao.



```
student@student-VirtualBox: ~/Documents
GNU nano 5.6.1      prova.cpp *
ciao
```

Read 0 lines

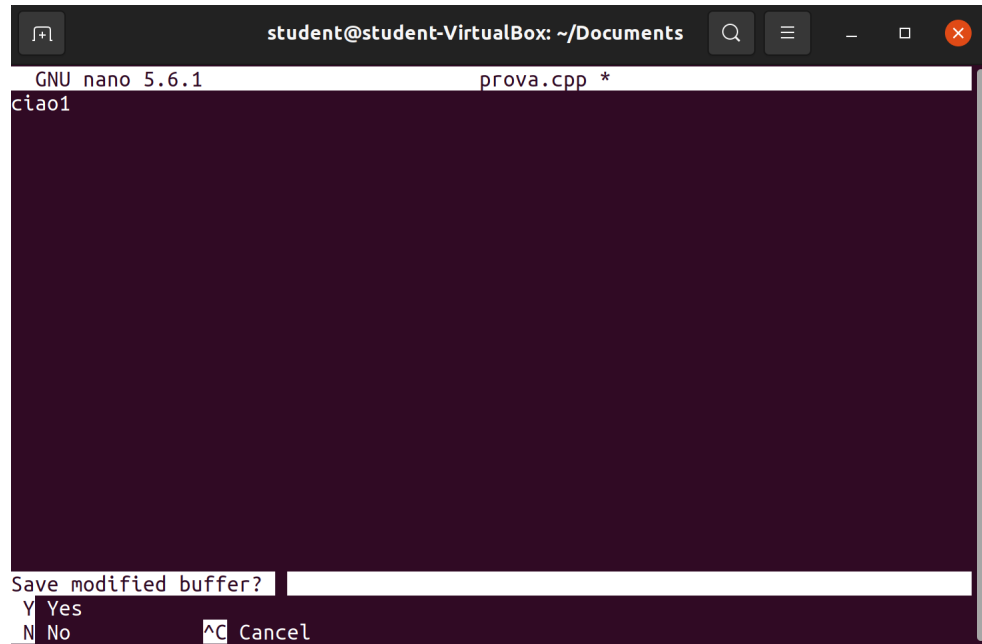
<b>^G</b> Help	<b>^O</b> Write Out	<b>^W</b> Where Is	<b>^K</b> Cut	<b>^T</b> Execute	<b>^C</b> Location
<b>^X</b> Exit	<b>^R</b> Read File	<b>^I</b> Replace	<b>^U</b> Paste	<b>^J</b> Justify	<b>^_</b> Go To Line

Infine utilizzando la seguente combinazione di tasti:

ctrl+x

E successivamente rispondendo positivamente alla domanda se si vuole salvare utilizzando la lettera *y* che sta ad indicare "yes", si sarà modificato il file.

Segue l'immagine di quest'ultima operazione:



La seconda strada per modificare il file sfrutta l'interfaccia grafica, come editor verrà utilizzato Emacs.

Per aprire il file con Emacs basterà lanciare il seguente comando da MacOS:

```
open -a Emacs name_file
```

Invece con Ubuntu basta lanciare il seguente comando:

```
emacs name_file &
```

E verrà aperto in entrambi i casi il file tramite l'interfaccia GUI di Emacs.

Nel caso in cui su Ubuntu non fosse già installato Emacs basterà lanciare il seguente comando e seguire le istruzioni:

```
sudo apt-get install emacs
```

Infine una volta finite tutte le operazioni é possibile spegnere il computer lanciando il seguente comando:

```
shutdown -h now
```



## 1.2 Variabili

Una variabile é un nome simbolico che identifica il contenuto di un'allocazione della memoria, la dimensione della memoria allocata dipende dal tipo di variabile.

Qualsiasi variabile per essere usata va dichiarata e risulta visibile solo nel blocco di graffe in cui essa é definita, segue un esempio:

```
#include <iostream>

using namespace std;

void res(){

int b;

}

int main(){

//Qui b non é più visibile

return 0;
}
```

Molto importante le variabili non possono coincidere con istruzioni, non possono contenere caratteri speciali, non possono iniziare con una cifra.

Inoltre il C++ é case-sensitive ovvero le variabili definite con lettere maiuscole e minuscole sono differenti.

Le variabili fondamentali sono:

- i. char 1 byte
- ii. short int 2 byte
- iii. **int** 4 byte
- iv. long long int 8 byte
- v. **float** 4 byte
- vi. **double** 8 byte
- vii. long double 12 byte

I tipi in grassetto sono i più importanti ed utilizzati le variabili *int* rappresentano numeri interi, i *float* rappresentano numeri reali ma con pochi decimali e i *double* rappresentano numeri reali con molte cifre decimali.

Alle variabili é possibile aggiungere l'aggettivo *const* così facendo le variabili diventano costanti é sarà impossibile aggiornare o riscrivere il loro valore.

### 1.3 Operatori logici

Gli operatori sono simboli o una collezione di simboli che agiscono sulle variabili e possono essere unari, binari e ternari i fondamentali sono:

- i. Assegnazione ( `=` ): tale operatore assegna un valore alla variabile su cui agisce, ad esempio scrivendo `a=1` si intende che alla variabile viene assegnato il valore 1 che é diverso dall'affermare che `a` é uguale ad 1, l'operatore uguaglianza ha un'altra definizione.
- ii. Operazioni elementari ( `+`, `-`, `*`, `/` e `%` ): rappresentano le operazioni aritmetiche fondamentali nonché somma, differenza, moltiplicazione e divisione mentre `%` identifica l'operatore resto di una divisione.
- iii. Operatori un po' meno elementari ( `++` e `--` ): il primo se fatto agire su una variabile incrementa quest'ultima di uno, mentre il secondo é l'inverso del primo.
- iv. Operatori di confronto ( `<`, `<=`, `>`, `>=`, `==` e `!=` ): tali operatori mettono in relazione le variabili la prima identifica il minore, la seconda il minore uguale, la terza maggiore, la quarta maggiore uguale, la quinta uguale in senso matematico e l'ultimo é il diverso in senso matematico.
- v. Operatori logici ( `||`, `&&` e `!` ): tali operatori equivalgono alle porte logiche or, and e not.

## 1.4 Interazione macchina utente

Di vitale importanza é l'interazione tra l 'utente e la macchina o viceversa, in C++ ciò avviene tramite il terminale.

Nel caso si voglia stampare un risultato, una variabile o un avviso il comando per fare ciò é:

*cout*

Nel caso in cui si voglia stampare un avviso di testo la sintassi sarà la seguente:

*cout*<<"Text"<<endl;

Invece per stampare una variabile o un risultato la sintassi é la seguente:

*cout*<<var/res<<endl;

Dove *endl* sta ad indicare che la linea é finita e il prossimo avviso verrà stampato a capo.

Invece se si volesse inserire il valore di una variabile o un testo il comando per fare ciò é:

*cin*>>var/"Text";

## 1.5 Controllo di flusso

I controlli di flusso sono dei comandi che permettono di iterare più volte un'azione o creare dei condizionali, se aggiunti al programma creano un flusso.

Il flusso di un qualunque programma lo si può scrivere in tre sole strutture di controllo, questa proprietà va sotto il nome di Teorema di Bohm-Jacopini.

Le strutture di controllo sono:

- i. Struttura sequenziale: il programma si sviluppa eseguendo le istruzioni nell'ordine in cui sono state scritte.
- ii. Struttura di selezione: la successiva istruzione da eseguire è determinata dall'esito della valutazione di un'espressione logica.
- iii. Struttura di iterazione: esecuzione ciclica di una serie di istruzioni fintanto che non si ottenga un'espressione logica vera.

Un programma sviluppato con tale struttura si dice strutturato e non presenta salti tra un'istruzione e l'altra.

Il primo controllo di flusso è il comando *if* che consente di inserire una domanda e se la risposta è positiva allora verrà svolto un determinato comando altrimenti il programma non svolgerà nessuna azione, la sintassi è la seguente:

$$if( test ) \{ \text{if test is true do this} \}$$

Nel caso in cui si voglia aggiungere una sola alternativa si può usare il comando *else*, la cui sintassi è:

$$if( test ) \{ \text{if test is true do this} \} \\ else \{ \text{if test is false do this} \}$$

Per aggiungere più alternative si può usare il comando *else if*, segue la sintassi:

$$else\ if( test ) \{ \text{if test is true do this} \}$$

La condizione di test può essere una sovrapposizione di richieste, ciò può essere fatto utilizzando gli operatori logici, seguono due esempi:

$$if( test_1 \ \&\& \ test_2 ) \{ \text{do this} \} \\ if( test_3 \ || \ test_4 ) \{ \text{do this} \}$$

Dove nel primo *if* verrà svolta l'operazione tra graffe se entrambe i test sono verificati, mentre nel secondo caso verrà svolta l'operazione se uno dei due è verificato oppure entrambi i test sono veri.

Se necessario di possono annidare i controlli di flusso, segue un esempio:

```
if( test ){
if( test ){
...
}
}
```

## 1.6 Cicli

Le strutture iterative o cicli vengono utilizzati per ripetere un gruppo di istruzioni, esistono diversi modo per creare un ciclo tra questi spiccano d'importanza:

- i. Il ciclo *while*;
- ii. Il ciclo *for*;
- iii. Il ciclo *do while*;

Il ciclo *while* ripete un determinato gruppo di istruzioni se non é verificata una condizione di test, la sintassi é la seguente:

```
while( test ){
instructions
}
```

Segue un esempio:

```
int n=0;
while(n<2){
cout<<"Ciao"<<endl;
n++;
}
```

Nel precedente ciclo la condizione di test é il valore di  $n$  che non deve essere maggiore di 2, ad ogni ciclo tramite l'istruzione  $n++$  il suo valore cresce di uno e raggiunto il valore due il ciclo si interrompe.

Il ciclo *for* é il ciclo più usato e semplice, esso permette di ripetere un gruppo di istruzioni data una condizione iniziale, finale e l'incremento.

Segue la sintassi:

```
for(conditioninit; conditionend; increase){
instruction
}
```

Segue un esempio:

```
for( int i=0; i<2; i++){  
    cout<<"Ciao"<<endl;  
}
```

Nell'esempio precedente il ciclo *for* stamperà sull'terminale la scritta ciao per due volte.

Infine il ciclo *do while* permette di ripetere un gruppo di istruzioni fin quando non é più verifica una condizione di test, segue la sintassi:

```
do{  
    istruzione;  
    incremento;  
}  
while( test );
```

Segue un esempio:

```
int n=0;  
do{  
    cout<<"Ciao"<<endl;  
    n++;  
}  
while( n<3 );
```

Il programma stamperà sul terminale la scritta Ciao per tre volte.

All'interno dei cicli é possibile usare due comandi che interrompono il flusso creando dei salti, le istruzioni sono:

- i. **break**: Interrompe il flusso del programma;
- ii. **continue**: Manda il flusso del programma all'inizio del blocco d'interazione ignorando le istruzioni successive;

Segue un esempio:

```
for( int i=0, i<6; i++){  
    if(i==3) continue;  
    if( i==5) break;  
    cout<<i<<endl;  
}  
return 0;  
}
```

Il programma stamperà sul terminale i numeri: 0,1,2,4.

Infine nel caso in cui si voglia ripetere un set d'istruzioni fin quando l'utente non interrompe l'esecuzione del programma o il ciclo abbia durata infinita si può usare il ciclo *for* nella seguente configurazione:

```
for( ; ; ){  
instructions  
}
```

## 1.7 Array

Gli array sono un'allocazione statica di una porzione della memoria, vengono utilizzati per rappresentare vettori, matrici e stringhe.

Gli elementi della memoria allocata dipendono dalla definizione dell'array.

Un array prima di essere usato va definito specificando per prima cosa la dimensione che deve essere intera o intera costante se la dimensione é nota a priori e successivamente il tipo, segue la sintassi:

```
const int dim;  
type name[dim];
```

L'aggettivo *const* impedisce la ridefinizione o riscrittura della variabile, se così non fosse si rischierebbe di compromettere il vettore.

Nel caso in cui la dimensione non fosse nota basta definirla intera, segue la sintassi:

```
int dim;  
type name[dim];
```

Una volta definito sarà possibile riempirlo, segue la sintassi:

$$\text{name}[\text{dim}] = \{ \text{element}_1, \text{element}_2, \dots, \text{element}_n \}$$

Il riempimento di un array può avvenire anche dinamicamente tramite la chiamata della componente, questo perchè quando si alloca un vettore ad ogni componente viene assegnato in indirizzo ovvero un codice alfa numerico tipo:

$$0\text{xFFF000n}$$

Dove  $n$  può essere un numero o una lettera, nota la coordinata della memoria la si può puntare e riempire.

Per chiamare la componente del vettore basta scrivere il nome del vettore e tra le quadre indicare il numero della componente desiderata. Si consideri il seguente vettore:

```
const int n=3;  
double v[n]={1,23,4};
```

La componente 0 equivale ad 1, la componente 2 equivale a 23 e infine la componente 2 equivale a 4, segue la sintassi per chiamare le componenti del vettore:

```
cout<<v[0]<<endl;
```



Dove  $v[0]$  é la chiamata alla prima componente del vettore e vale 1.  
Noto questo fatto é possibile riempire dinamicamente il vettore utilizzando un ciclo d'interazione, riempiendo volta per volta le componenti, segue la sintassi:

```
int dim;  
double v[dim];  
for(int i=0, i<n, i++){  
    cin>>v[i];  
}
```

### 1.7.1 Array multidimensionali

Gli array multidimensionali sono array con più indici, nonché matrici, la sintassi per definirli é la medesima solo che bisogna aggiungere un indice in più, segue la sintassi:

```
int dim1,dim2;  
type name[dim1][dim2];
```

Segue un esempio:

```
int x=2,y=2;  
double M[x][y] = {{1, 2}  
                  {3, 4}  
                  };
```

Dove  $dim_1$  rappresenta il numero di righe,  $dim_2$  il numero di colonne. Nell'esempio precedente é stata definita una matrice di taglia 2x2 le cui entrate sono:

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Anche per le matrici esiste un modo per riempirli dinamicamente, per prima cosa si definisce la taglia e tramite in ciclo si inseriscono l'entrate.

Segue la sintassi:

```
int dim1,dim2;  
type name[dim1][dim2];  
for (int i=0; i<dim1; i++){  
  for(int j=0; j<dim2; j++){  
    cin>>M[i][j];  
  }  
}
```

Nella sintassi precedente il primo ciclo *for* scorre sulle righe e il secondo sulle colonne.

Per prima cosa si fissa la riga che si vuole riempire e dopo scorrendo sulle colonne si inseriscono i valori, iterano questo procedimento su tutte le righe e su tutte le colonne si crea la matrice.

POCO CHIARO

MANCA LA CHIAMATA ALLE COMPONENTI

## 1.8 Puntatori

I puntatori sono uno strumento tanto ponte quanto difficili da comprendere ed utilizzare, una mal definizione del puntatore può portare dei bug molto difficili da correggere e al crash del programma.

I puntatori sono un ottimo strumento nel caso si voglia:

- i. Velocizzare e facilitare la gestione dei dati allocati in memoria;
- ii. Sviluppare algoritmi più veloci ed efficaci;
- iii. **Allocare dinamicamente della memoria;**
- iv. Passare parametri alle funzioni;

Come visto in precedenza una variabile è l'etichetta di una locazione della memoria e la sua dimensione dipende dal tipo di definizione del contenuto, in maniera analoga una variabile puntatore è un'etichetta di una locazione di memoria delle dimensioni necessarie ad ospitare un indirizzo di memoria.

I puntatori vengono gestiti e definiti tramite i seguenti operatori:

$\&^1$   $*^2$

Per utilizzare i puntatori per prima cosa vanno definiti e successivamente vanno fatti puntare verso qualcosa, segue la sintassi per la definizione:

`type* name`

Dove *type* è il tipo di variabile ( `int`, `double` e `float` ), una volta definita va puntata, supponendo di aver precedentemente definito una variabile *a*, il puntamento può essere fatto usando la seguente sintassi:

`*name= &a`

Supponendo di aver chiamato il puntato *p* a di avergli dato come referenza *a*, diremo che:

*p* punta ad *a*

---

<sup>1</sup>Tale operatore viene chiamato operatore di referenza.

<sup>2</sup>Tale operatore viene chiamato operatore di dereferenziazione.

Per comprendere al meglio quanto detto consideriamo la seguente figura:

Indirizzo	Contenuto	Variabile
7FF7B4506954	3	int a
7FF7B1E63948	7FF7B4506954	int* p

Dalla figura si deduce che la variabile  $a$  vale 3 ed è stata assegnato il seguente indirizzo 0x7FF7B4506954, ovvero il luogo del computer in cui è stato salvato il contenuto.

Invece al puntatore  $p$  è stato assegnato come contenuto l'indirizzo della variabile ovvero 0x7FF7B4506954 ed è stato assegnato il seguente indirizzo 0x7FF7B1E63948. Tramite il puntatore è possibile accedere a diverse informazioni:

- i. Se si chiama il puntatore tramite l'operatore di referenza otterremmo l'indirizzo del puntatore:

$\&p$

- ii. Se si chiama il puntatore tramite l'operatore di dereferenziazione otterremmo il valore della variabile puntata:

$*p$

- iii. Se si chiama il puntatore senza nessun operatore di ottiene il valore del suo indirizzo.

Noi in precedenza abbiamo già visto dei puntatori sotto mentite spoglie, ovvero gli array.

Quando si inizializza un array si definisce il puntatore alla locazione di memoria contenente l'indirizzo dell'array.

Supponiamo di aver definito in array di dimensione 3 di valori interi e di averlo chiamato  $v$  se effettuiamo la chiamata alla componente  $i$ -esima del puntatore tramite l'operatore di dereferenziazione otterremo il valore  $i$ -esimo della componente dell'array.

Invece se chiamiamo il valore dell'indirizzo del pointer e dell'array scopriremo che sono identici questo perché il puntatore dell'array punta a se stesso. Questo legame tra i pointer e gli array ci permette di modificare il valore dell'array senza agire direttamente su di esso, oppure di crearne uno senza sapere preventivamente la quantità di memoria necessaria per definirlo.

## 1.9 Stream da file

Con stream da file si intende la possibilità di accedere al contenuto di un determinato file o scrivere su esso.

Per accedere ai comandi necessari per svolgere tale operazione bisogna utilizzare la libreria *fstream*, il comando per accedere al file è<sup>3</sup>:

*ifstream*

Mentre per scrivere su un file il comando è<sup>4</sup>:

*ofstream*

Nel caso in cui si voglia accedere al contenuto del file, per prima cosa bisogna assicurarsi che il file sia nella stessa cartella del programma, utilizzando il metodo di *open* è possibile aprire il file.

Segue la sintassi:

*ifile.open*("name\_file.dat");

Dove *ifile* è il nome con cui è stata importata la classe *ifstream*.

Una volta che sono state svolte tutte le operazioni bisogna chiudere il file, ciò può essere fatto utilizzando il seguente metodo:

*ifile.close*()

Invece per correggere eventuali errori nell'apertura è possibile utilizzare il seguente metodo:

*ifile.clear*()

Per tornare all'inizio del file si può usare il seguente metodo:

*ifile.seekg*(ios::begin)

Infine dopo aver importato il contenuto del file per scriverlo riga per riga all'interno di un puntatore basta utilizzare la seguente sintassi:

*ifile*>>p[i];

---

<sup>3</sup> *ifstream* sta ad indicare che è stato chiamato lo *stream* in modalità *input*.

<sup>4</sup> *ofstream* sta ad indicare che la chiamata allo *stream* è stata fatta in modalità *output*.

Invece nel caso in cui si voglia scrivere sul file si può usare il metodo che crea, nel caso in cui il file non esistesse, e apre il file, segue la sintassi:

```
ofile.open("name_file.dat")
```

Dove *ofile* é il nome con cui é stata importata la classe di *ofstream*, dopo aver scritto sul file bisogna chiuderlo e per fare ciò basta usare il seguente metodo:

```
ofile.close()
```

Infine per scrivere all'interno del file basta utilizzare la seguente sintassi:

```
ofile<<things<<endl;
```

Dove *things* sta ad indicare una generica cosa che si vuole scrivere sul file, ad esempio le componenti di un vettore, un risultato, del testo e ecc..

## 1.10 Allocazione dinamica

L'allocazione di memoria é il processo in cui viene assegnata una parte di memoria della macchina per un determinato processo, per esempio quando si definisce un array é necessario inizializzare quanto é grande il vettore cosí facendo si riserva una porzione di memoria per il vettore.

Però per svolgere tale operazione bisogna sapere a priori quanta memoria serve per definirlo, purtroppo la maggior parte delle volte la quantità di memoria necessaria é nota solo quando si esegue il programma ciò implica che tale variabile non é controllabile dall'utente.

Per ovviare a questo problema si può operare in due modi:

- i. Allocare una grande quantità di memoria;
- ii. Allocare la memoria dinamicamente;

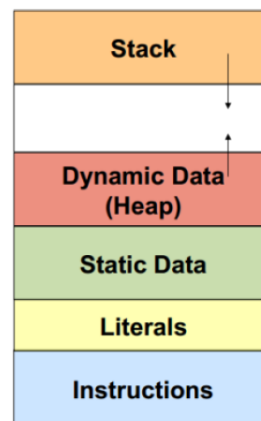
La prima scelta risulta essere quella più dispendiosa di risorse e pericolosa, supponiamo di dover scrivere dei valori in un vettore se allocassimo troppa poca memoria il sistema andrà in *overflow* causando il crash del programma.

Invece allocando più memoria del necessario useremo troppe risorse inutilmente. Possiamo concludere che la scelta migliore é allocare la memoria dinamicamente, ovvero definire la memoria basilare al momento della compilazione o dell'esecuzione.

Prima di procedere é doveroso fare un approfondimento sui tipi di memorie e dove tale processo avverrà.

Quando un processo viene caricato in memoria il sistema operativo gli assegna uno spazio composto da cinque regioni più una:

- i. Stack: zona in cui é contenuto il codice;
- ii. Free: regione libera di memoria ed allocabile al bisogno;
- iii. Static Data: zona dedicata alle variabili globali o statiche;
- iv. Stack: regione dedicata alle variabili locali;
- v. Dynamic Data o Heap: regione dedicata all'allocazione dinamica;



Durante il processo di allocazione dinamico l'Heap forza lo Stack ad espandersi o ritirarsi dalla zona Free, così facendo si riesce a modificare le quantità di risorse per un determinato processo, per fare ciò serve un operatore che costruisca un oggetto nel Heap e punti ad una variabile nello Stack tale operatore é:

*new*



Tale operatore alloca memoria nel Dynamic Data e restituisce un puntatore con quel indirizzo di memoria, segue la sintassi per allocare una spazio di memoria di dimensione  $n$ , intero:

```
type *name_pointer = new type[n]
```

Infine per deallocare la memoria si può usare il comando *delete*.

Arrivati a questo punto si hanno tutti gli strumenti necessari per svolgere una delle operazioni più importanti ovvero accedere al contenuto del file, salvarlo in memoria e poterlo usare successivamente.

Supponiamo di voler accedere al contenuto di un file chiamato *dati.dat*, la cui dimensione è ignota, il file è strutturato nel modo seguente:

$x_1$	$y_1$
$x_2$	$y_2$
$x_3$	$y_3$
...	...

I passaggi per svolgere tale operazione sono:

- i. Aprire il file;
- ii. Contare gli elementi;
- iii. Allocare la memoria;
- iv. Tornare in cima al file;
- v. Correggere eventuali errori;
- vi. Scrivere il contenuto in un pointer;

Per aprire un file si può usare lo stream in configurazione di input, segue la sintassi:

```
#include<iostream>
#include <fstream>

using namespace std;

int main(){
    ifstream ifile;
    ifile.open("dati.dat");
```

Una volta aperto il file bisogna contare gli elementi sono al suo interno, il modo più semplice é far scorrere una variabile per ogni colonna su ogni elemento del file e incrementare un contatore fin quando non si é raggiunto la fine del file. Per fare ciò serve un ciclo che ripeta una determinata operazione fin quando non é più verificata la condizione di test, ciò può essere fatto dal ciclo *while*:

```
#include<iostream>
#include <fstream>

using namespace std;

int main(){
    double b,g;
    int n=0;
    ifstream ifile;
    ifile.open("dati.dat");
    while(ifile>>b>>g){
        n++;
    }
```

A questo punto é nota la dimensione di memoria da allocare, quindi si può procedere con l'allocazione dinamica di un pointer per ogni colonna:

```
#include<iostream>
#include <fstream>

using namespace std;

int main(){

    double b,g;
    int n=0;
    ifstream ifile;
    ifile.open("dati.dat");
    while(ifile>>b>>g){

        n++;

    }

    double* x = new double[n];
    double* y= new double[n];
```

Una volta che definiti bisogna riportare *ifile* in cima al file e pulire eventuali errori, segue la sintassi:

```
#include<iostream>
#include <fstream>

using namespace std;

int main(){

    double b,g;
    int n=0;
    ifstream ifile;
    ifile.open("dati.dat");
    while(ifile>>b>>g){

        n++;

    }

    double* x = new double[n];
    double* y= new double[n];

    ifile.seekg(ios::begin);
    ifile.clear();
```

A questo punto si possono riempire ed usare successivamente, per farlo si usa *ifile* iterando l'operazione per ogni elemento del file, segue la sintassi:

```
#include<iostream>
#include <fstream>

using namespace std;

int main(){

    double b,g;
    int n=0;
    ifstream ifile;
    ifile.open("dati.dat");
    while(ifile>>b>>g){

        n++;

    }

    double* x = new double[n];
    double* y= new double[n];

    ifile.seekg(ios::begin);
    ifile.clear();

    for(int i=0; i<n; i++){

        ifile>>x[i]>>y[i];

    }

}
```

## 1.11 Funzioni

Le funzioni sono uno strumento che consentono di decomporre un problema in moduli logicamente indipendenti, permettendo di raggruppare un determinato numero d'istruzioni che codificano un compito in risposta al suo input.

Supponendo che nel programma siano presenti due funzioni A e B diremo che A chiama B quando la funzione A trasferisce il controllo alla funzione B trasmettendogli un input, a suo volta la funzione B finite le operazioni restituisce alla funzione A dei risultati.

Per definire una funzione bisogna:

- i. Definire il prototipo della funzione nel preambolo;
- ii. Costruire il corpo della funzione prima o dopo il *main()*, ovvero i compiti che deve svolgere;

Per definire il prototipo se procede nel modo seguente:

*typedef* name(*type1* p1, *type2* p2, ..., *typen* pn);

Dove:

- i. *typedef* indica il tipo di valore che ritorna la funzione che può essere double, float, int o void se non si ha nessun output;
- ii. *type1*, *type2*, *typen* sono i tipi di valori in ingresso e posso essere double, float, int, string oppure nulla se non serve nessun valore in input;
- iii. p1, p2, pn sono gli oggetti in ingresso e possono essere variabili, array e puntatori;

Dopo aver definito il prototipo si può costruire la funzione, la sintassi é la seguente:

```
typedef name(type1 p1, type2 p2, ..., typen pn){  
    instructions;  
    return val_undef;  
}
```

Supponiamo di voler scrivere una funzione che calcoli la potenza tra, iniziamo definendo il prototipo:

*double* pot( *int*, *int*);

Tale funzione si chiama *med* e restituisce un valore double, in input invece si hanno due interi che saranno la base e l'esponente.

Definito il prototipo si costruisce il corpo della funzione come segue:

```
double pot( int, int);  
int main(){  
    ...  
}  
double pot(int a, int b){  
    double res=pow(a,b);  
    return res;  
}
```

Infine se all interno del *main()* si passano due valori alla funzione lei restituirà il valore della potenza:

```
double pot( int, int);  
int main(){  
    cout<<pot(2,2)<<endl;  
}  
double pot(int a, int b){  
    double res=pow(a,b);  
    return res;  
}
```

Sul terminale verrà stampato 4 ovvero il valore di due alla seconda.  
MANCANO GLI ATRI CASI VETTORI, MATRICI E RITORNO VETTORI.

### **1.12 Librerie**

TESTARE PRIMA SUI PC DELLA SCUOLA

### **1.13 ROOT**

PROVA AD INSTALLARE ROOT SE FUNZIONA USARE LE MACRO.

## 1.14 Programmi

### 1.14.1 Primo Programma

Il seguente programma non ha nessuna utilità pratica ma solamente didattica, consente di vedere la struttura base di un programma e il primo comando per l'interazione uomo macchina.

```
#include <iostream>

using namespace std;

int main(){

    cout<<"Hello world"<<endl;

    return 0;
}
```

Dopo aver compilato ed eseguito il programma esso stamperà sul terminale la scritta Hello world.



### 1.14.2 Calcolo dell'area

Lo scopo del programma che segue è quello di calcolare l'area di un rettangolo di lato  $a$ ,  $b$  i cui valori vengono inseriti dall'utente e stampare il risultato dell'area sul terminale.

```
#include <iostream>

using namespace std;

int main(){

    double a,b,area;

    cout<<"Inserire il valore del lato a:"<<endl;
    cin>>a;

    cout<<"Inserire il valore del lato b:"<<endl;
    cin>>b;
    area=a*b;

    cout<<"L'area vale:"<<area<<endl;

    return 0;

}
```

### 1.14.3 Equazione di secondo grado

Lo scopo del seguente programma é risolvere le equazioni di secondo grado, discriminando se esistono soluzioni distinte, coincidenti o non esistono soluzioni reali e il tipo di equazione.

Una generica equazione di secondo grado può essere scritta nel modo seguente:

$$ax^2 + bx + c = 0$$

Dove i coefficienti possono essere numeri reali qualsiasi, le soluzioni delle equazioni di secondo grado sono:

$$x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}$$

Dove:

$$\Delta = b^2 - 4ac$$

É detto discriminante e decreta l'esistenza delle soluzioni, in particolare:

- i. Se  $\Delta > 0$  le soluzioni sono reali e distinte;
- ii. Se  $\Delta = 0$  le soluzioni sono reali e coincidenti;
- iii. Se  $\Delta < 0$  non esistono soluzioni reali

Infine diremo che un'equazione di secondo grado é:

- i. Spuria se  $b \neq 0$  e  $c=0$ ;
- ii. Pura se  $b=0$  e  $c \neq 0$ ;
- iii. Completa se nessuno dei coefficienti é nullo;

```

Segue il programma:
#include<iostream>
#include<cmath>

using namespace std;

int main(){

double a,b,c,delta,x1,x2,x;

cout<<"Inserire i valori dei coefficienti"<<endl;
cin>>a;
cin>>b;
cin>>c;

delta=pow(b,2)-4*a*c;
if( b!=0 && c=0){

cout<<"L'equazione é spuria"<<endl;

}

if(delta>0){

x1=(b+sqrt(delta))/(2*a);
x2=(b-sqrt(delta))/(2*a);

cout<<"La soluzione uno vale"<<x1<<"La soluzione due vale"<<x2<<endl;

}

if(delta==0){

x=-b/(2*a);

cout<<"La soluzione é unica e vale"<<x<<endl;

}

if(delta<0){

cout<<"Non esistono soluzioni reali"<<endl;

}

```

Nel recedente programma si é utilizzato il metodo *pow* della libreria *cmath*, il metodo *pow* restituisce la potenza data la base e l'esponente. Segue la sintassi:

$$pow(a, b) = a^b$$

#### 1.14.4 Somma

Lo scopo del seguente programma é quello di calcolare la somma delle variabili inserite dall'utente:

```
#include<iostream>

using namespace std;

int main(){

    int n;
    double a,sum;

    sum=0;

    cout<<"Inserire quanti numeri si vogliono sommare"<<endl;
    cin>>n;

    for(int i=0;i<n;i++){

        cout<<"Inserire il valore"<<endl;

        cin>>a;

        sum=sum+a;

    }

    return 0;

}
```

La somma delle variabili viene svolta all'interno del ciclo *for*.  
La variabile *sum*, che identifica il valore della somma, viene inizializzata a zero e ad ogni ciclo viene aggiornata aggiungendo il nuovo valore al precedente.  
Per comprendere il funzionamento del programma supponiamo di dover sommare 3 numero quindi  $n=3$  e i valori in ingresso sono 1,2,3.  
Inizialmente  $sum=0$  e alla prima interazione, ovvero  $i=0$ ,  $a=1$  quindi avremo che il nuovo valore di *sum* sarà:

$$sum_0 = sum + a = 0 + 1 = 1$$

Dove  $sum_0$  indica il valore della somma quando  $i=0$ , all'interazione successiva quando  $i=1$  il nuovo valore sarà:

$$sum_1 = sum_0 + a = 1 + 2 = 3$$

Infine al passo  $i=2$  si otterrà il valore finale:

$$sum_2 = sum_1 + a = 3 + 3 = 6$$

#### 1.14.5 Array costante

Lo scopo di questi due programmi é definire un array, il primo definisce un array di dimensione e contenuto definito mentre il secondo la memoria dell'array e le sue componenti vengo riempite dall'utente.

Primo programma:

```
#include<iostream>

using namespace std;

int main(){

    const int n=3;
    double v[n]={0,1.5,2.7};

    return 0;

}
```

Nel precedente programma é stato definito un array di dimensione costante pari a 3 a componenti double, nel seguente programma la dimensione del vettore rimane costante pari a 3 ma le componenti verranno inserite dall'utente:

```
#include<iostream>

using namespace std;

int main(){

    const int n=3;
    double v[n];

    for(int i=0;i<n;i++){
        cin>>v[i];
    }

    return 0;

}
```

Il ciclo for consente di assegnare un valore alle componenti del vettore partendo dalla componente 0.

#### 1.14.6 Array a dimensione non definita inizialmente

Lo scopo del programma é quello di definire un array la cui dimensione viene inserita dall'utente e successivamente vengono inserite le componenti dell'array:

```
#include<iostream>

using namespace std;

int main(){

    int n;
    double v[n];

    cout<<"Inserire la dimensione del vettore"<<endl;
    cin>>n;

    for(int i=0;i<n;i++){
        cin>>v[i];
    }

    return 0;

}
```



### 1.14.7 Array multidimensionali

Lo scopo dei seguenti programmi é quello di definire e riempire un array multidimensionale, si procede come per gli array unidimensionali, prima si definisce la dimensione ( che deve essere un intero ) e successivamente il vettore.

Nel primo programma verrà definito un vettore multidimensionale statico, cioè dimensione e componenti non saranno più modificabili:

```
#include <iostream>

using namespace std;

int main(){
    const int x=2,y=2;

    double M[x][y] = {{1,2}
                       {4,6}}

    for( int i=0; i<x; i++){
        for( int j=0; j<y; j++){
            cout<<M[i][j]<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

L'output del programma sarà:

$$\begin{pmatrix} 1 & 2 \\ 4 & 6 \end{pmatrix}$$

Nonché una matrice 2x2, la stampa del risultato avviene in due fasi per prima cosa si fissa la riga da stampare e ciò viene fatto dal primo ciclo *for*, successivamente si procede nel stampare i valori delle colonne. Una volta che tutti i valori sono stati stampati si procede alla riga successiva e si stampano i valori delle colonne.

In maniera del tutto analoga si può scrivere un programma in modo tale che sia l'utente ad inserire i valori della matrice, segue un esempio:

```
#include <iostream>

using namespace std;

int main(){

    const int x=2,y=2;

    double m[x][y];

    cout<<"Inserire i valori della matrice:"<<endl;

    for(int i=0; i<x; i++){
    for(int j=0; j<y; j++){

        cin>>m[i][j];

    }
    }

    return 0;
}
```

### 1.14.8 Cramer

Lo scopo del seguente programma é quello di risolvere i sistemi di equazioni con il Metodo di Cramer, si rammenta che un generico sistema a due incognite può essere scritto del modo seguente:

$$\begin{cases} ax + by = c \\ a'x + b'y = c' \end{cases}$$

Dove  $a, a', b, b'$  sono i coefficienti delle incognite e  $c, c'$  sono i termini noti. Per risolvere il sistema con il Metodo di Cramer bisogna:

- i. Scrivere la matrice del sistema associata;
- ii. Calcolare la matrice delle incognite ed il suo determinante;
- iii. Calcolare le matrici delle soluzioni ed i loro determinanti;
- iv. Infine calcolare i valori delle incognite;

Nel nostro esempio la matrice associata al sistema é la seguente:

$$S = \begin{pmatrix} a & b & c \\ a' & b' & c' \end{pmatrix}$$

La matrice delle incognite é:

$$M = \begin{pmatrix} a & b \\ a' & b' \end{pmatrix}$$

Le matrici delle soluzioni si ottengono dalla matrice  $M$  sostituendo la colonna dei valori noti con quella delle soluzioni che si sta cercando, per le  $x$  si avrà:

$$M_x = \begin{pmatrix} c & b \\ c' & b' \end{pmatrix}$$

Invece per le  $y$  si ottiene:

$$M_y = \begin{pmatrix} a & c \\ a' & c' \end{pmatrix}$$

Ricordando che il determinante di una matrice  $2 \times 2$  si calcola facendo il prodotto degli elementi della diagonale principale meno gli elementi della diagonale secondaria, applicato alla matrice  $M$  si ha:

$$\det M = ab' - a'b \tag{1}$$

Infine utilizzando la (1) per le altre matrici e ricordando che:

$$\begin{aligned} x &= \frac{\det M_x}{\det M} \\ y &= \frac{\det M_y}{\det M} \end{aligned}$$

Si giunge alla soluzione del problema.

```

Segue l'implementazione del programma:
#include <iostream>

using namespace std;

int main(){

    int x=2,y=2;
    double detM,detMx,detMy;

    double S[x][y+1];
    double Mx[x][y];
    double My[x][y];

    cout<<"Inserire i valori dei coefficienti"<<endl;

    for(int i=0; i<x; i++){
    for( int j=0; j<y+1; j++){

        cin>>S[i][j];

    }

    }

    for(int i=0; i<x; i++){
    for( int j=0; j<y+1; j++){

        if( j==0 ){

            Mx[i][0]=S[i][2];
            My[i][0]=S[i][0];

        }

        else if( j==1 ){

            Mx[i][1]=S[i][1];
            My[i][1]=S[i][2];

        }

        detM=S[0][0]*S[1][1]-S[0][1]*S[1][0];
        detMx=Mx[0][0]*Mx[1][1]-Mx[0][1]*Mx[1][0];
        detMy=My[0][0]*My[1][1]-My[0][1]*My[1][0];

```

```
    cout<<"x="<<detMx/detM<<endl;  
    cout<<"y="<<detMy/detM<<endl;  
  
    return 0;  
  
}
```

#### **1.14.9 Scrittura su file**

**1.14.10    Allocazione dinamica**

**1.14.11    Funzioni**

**1.14.12    Librerie**

## 1.15 Esercizi

### 1.15.1 Esercizio

Utilizzando il terminale recarsi nella cartella dei documenti e creare una cartella di nome Prova e creare interno ad essa un file di nome prova.cpp, successivamente cancellare la cartella precedentemente creata.

Scrivere i comandi utilizzati su un file e caricarlo nell'apposita sezione per la consegna.

### 1.15.2 Esercizio

Scrivere un programma che inserendo il valore del diametro di un cerchio calcoli:

- 1) Il raggio;
  - 2) La circonferenza del cerchio;
  - 3) L'area del cerchio;
- E stampi i risultati sul terminale.

### 1.15.3 Esercizio

Scrivere un programma che risolva il moto del proiettile, specificando il valore della gittata in base all'angolo.

### 1.15.4 Esercizio

Scrivere un programma che dati in ingresso 5 numeri restituisca all'utente se il numero é pari o dispari.

### 1.15.5 Esercizio

Scrivere un programma che calcoli la media dei voti di Informatica e stampi il risultato sul terminale, se la media é minore di 5 stampare sul terminale che lo studente verrà bocciato e invece che verrà promosso se la media é maggiore di 6.

### 1.15.6 Esercizio

Scrivere un programma che calcoli la somma delle componenti di un vettore di dimensione 3 e stampi il risultato sul terminale.

### 1.15.7 Esercizio

Scrivere un programma che utilizzando il metodo di Cramer risolva un sistema 3x3, sapendo che il determinante di una matrice M:

$$M = \begin{pmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & M_{22} \end{pmatrix}$$

Si calcola utilizzando la seguente formula:

$$\begin{aligned} \det(M) = & M_{00}[M_{11}M_{22} - M_{12}M_{21}] + \\ & -M_{01}[M_{10}M_{22} - M_{12}M_{20}] + \\ & +M_{02}[M_{10}M_{21} - M_{11}M_{20}] \end{aligned}$$



## 1.16 Soluzioni

### 1.16.1 Esercizio 1

### 1.16.2 Esercizio 2

### 1.16.3 Esercizio 3

### 1.16.4 Esercizio 4

### 1.16.5 Esercizio 5

## 2 Python

### 2.1 Introduzione

### 2.2 Variabili

### 2.3 Operatori logici

### 2.4 Controlli di flusso

### 2.5 Cicli

### 2.6 Array

#### 2.6.1 Array multidimensionali

### 2.7 Funzioni

### 2.8 Librerie forse

### 2.9 Programmi

### 2.10 Qiskit

#### 2.10.1 Introduzione

#### 2.10.2 Quantum Lab and Quantum Composer

#### 2.10.3 Implementazione circuito quantistico

#### 2.10.4 Simulator

#### 2.10.5 Measure circuit

#### 2.10.6 Real Simulation

#### 2.10.7 Schrödinger's cat

## 3 Quantum Computing

### 3.1 Introduzione

#### 3.1.1 Qubits

### 3.2 Quantum information

#### 3.2.1 Quantum Gates

#### 3.2.2 Quantum parallelism

#### 3.2.3 Deutsch-Jozsa Algorithm

### 3.3 Theorem of no cloning

#### 3.3.1 Quantum security

## 7 Bibliografia

- c++
- A.Bacciotti, F.Ricci, *Analisi matematica 1*, Liguori Editore;
- Michael A. Nielsen, Isaac L.Chuang, *Quantum Computing and Quantum Information*, Cambridge University Press;
- IBM-Q, *Learn Quantum Computing using Qiskit*, <https://qiskit.org/textbook/preface.html>;
- C.E.R.N., *ROOT Reference Documentation*, :<https://root.cern/doc/master/>;