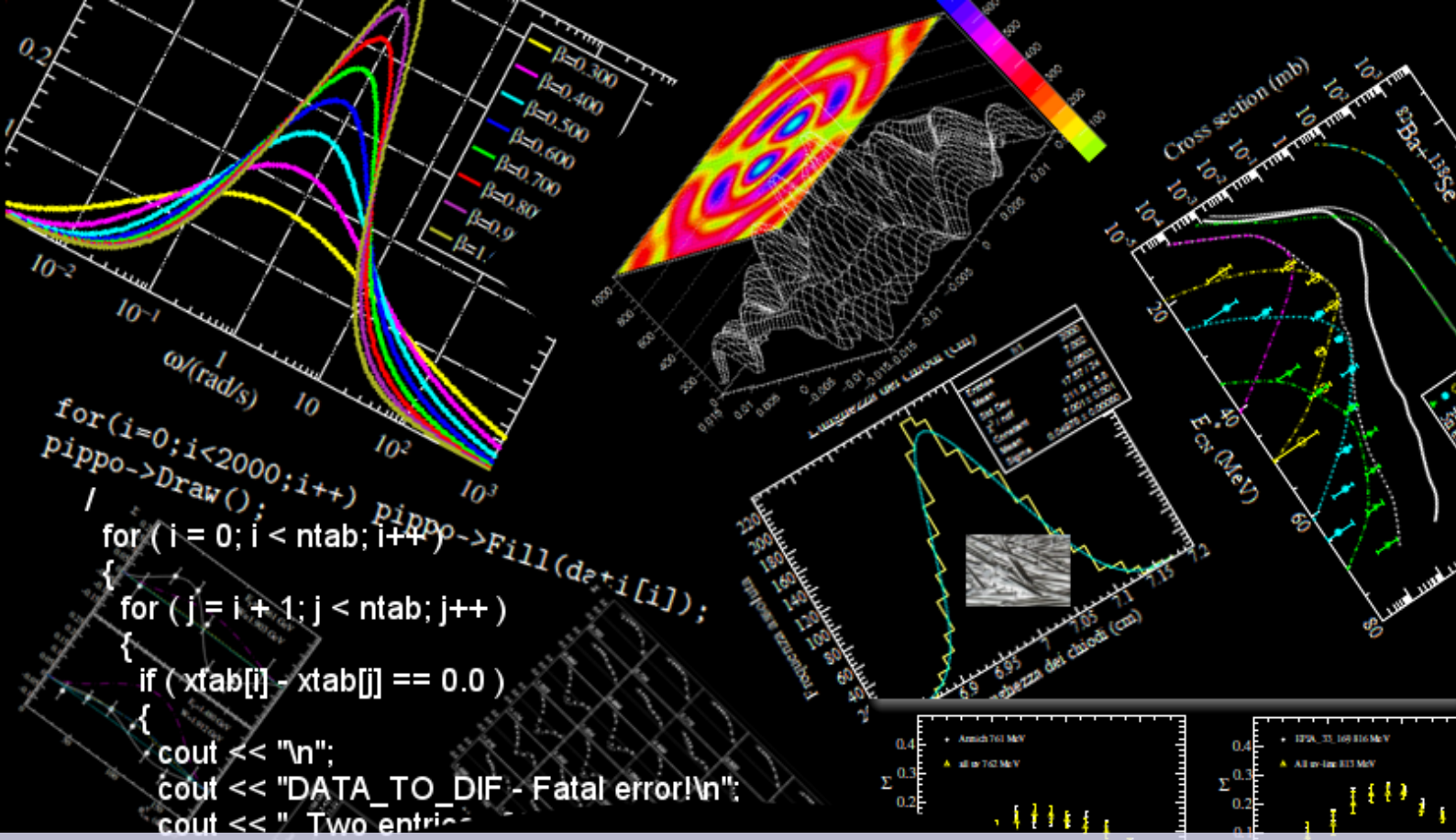
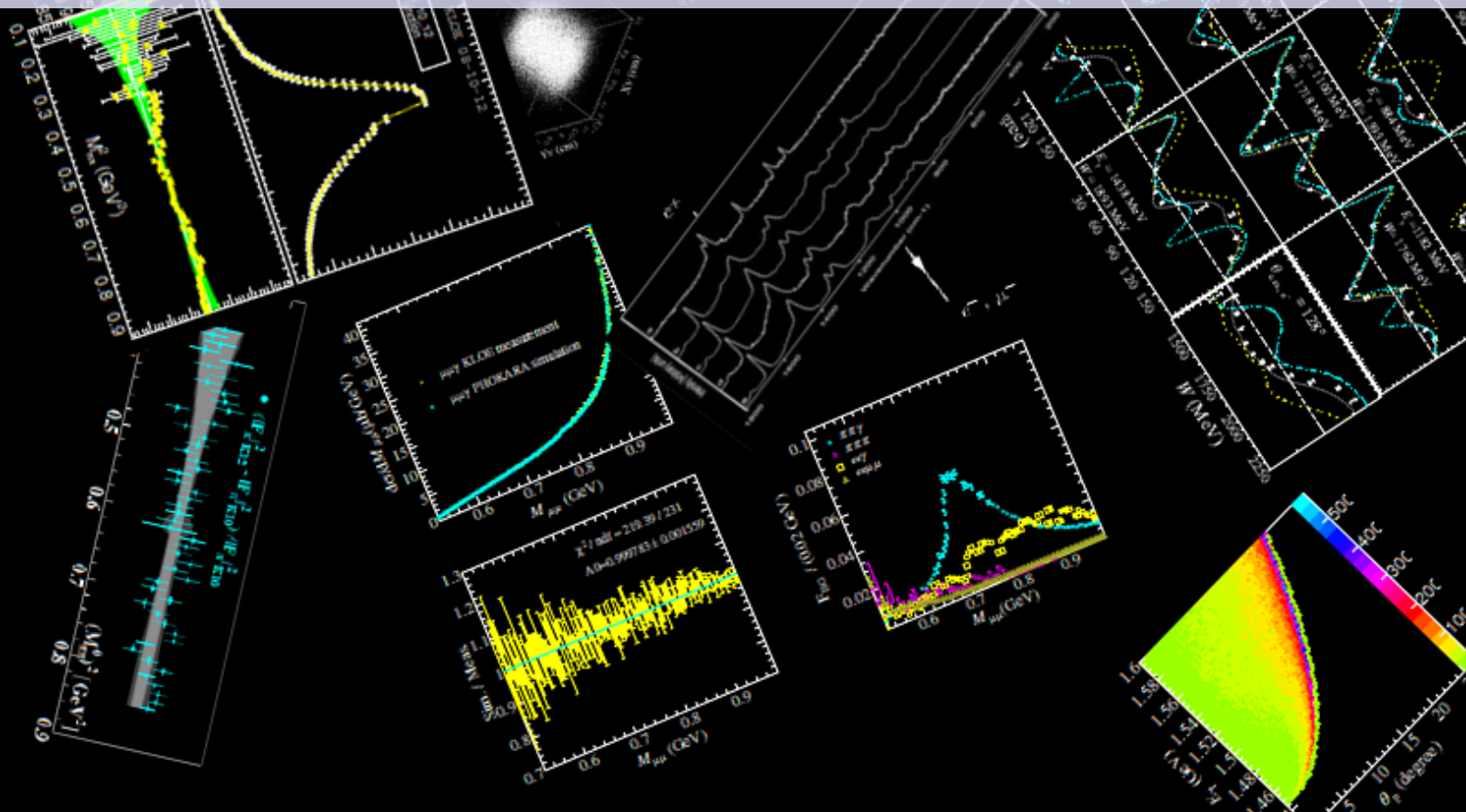


Laboratorio Informatico

Appunti al corso anno accademico 2016/17

Giuseppe Mandaglio

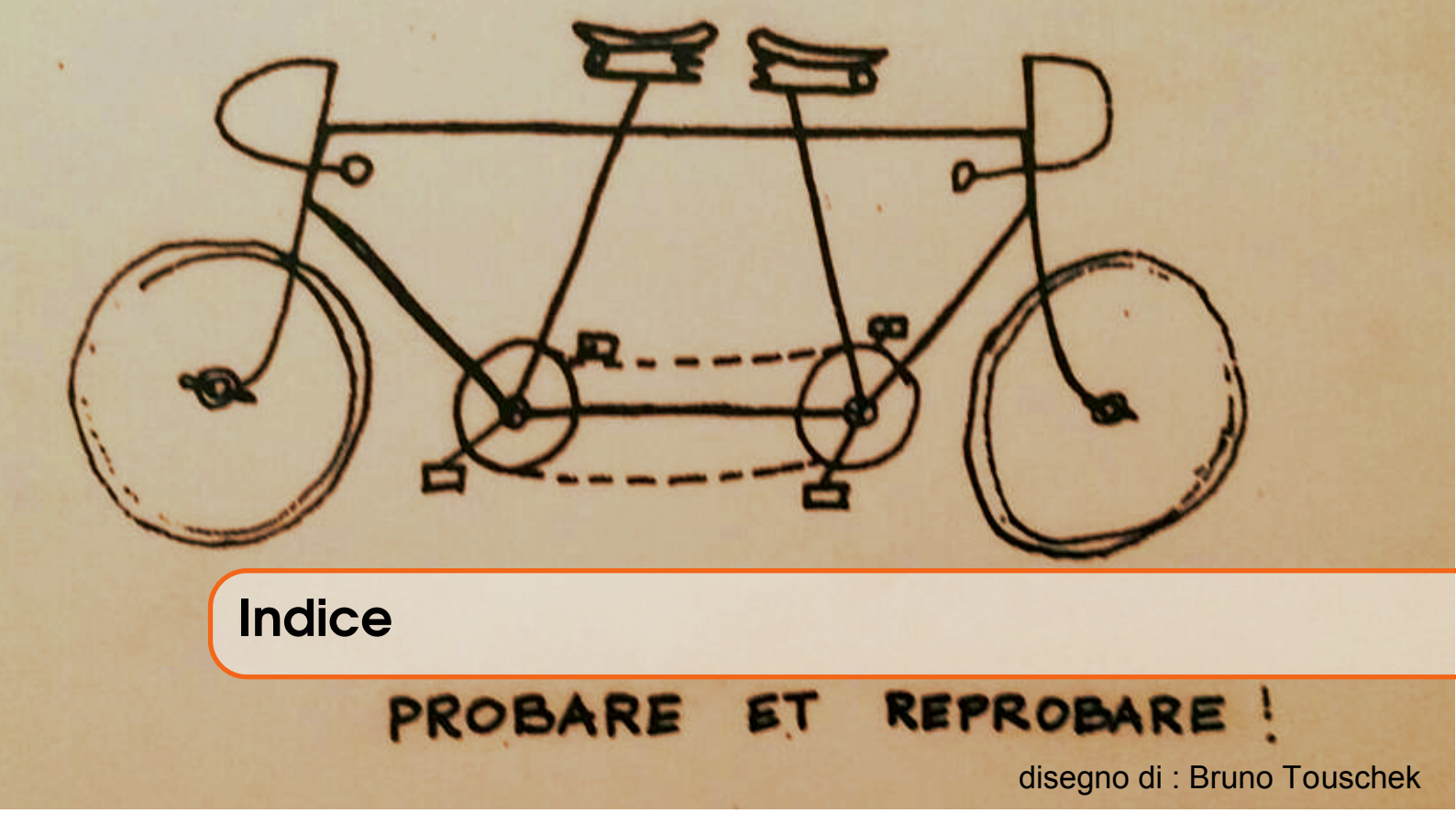


BOOK-WEBSITE.COM

First printing, ??

Prefazione

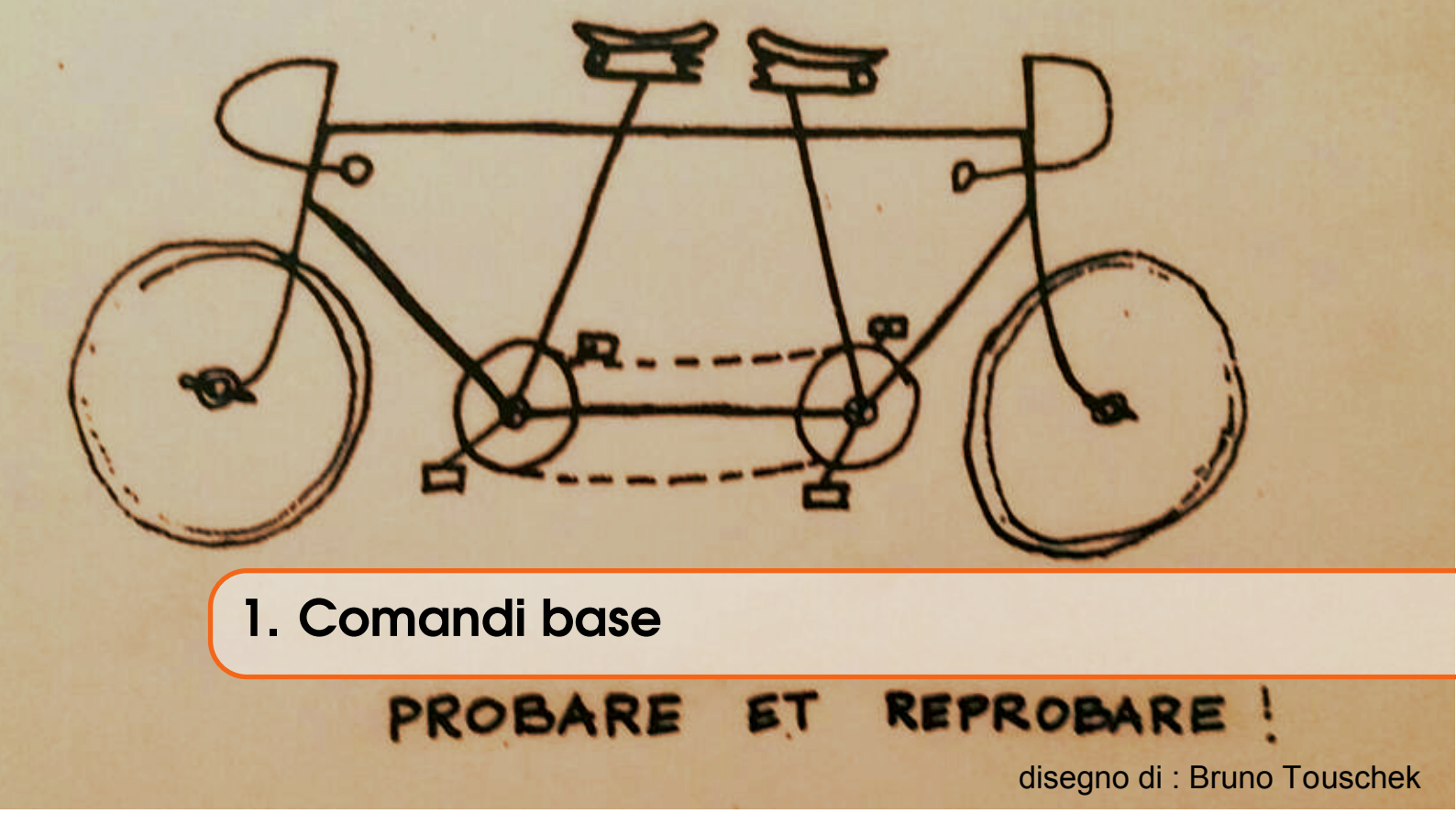
Il corso di Laboratorio Informatico per il corso di Laurea Triennale in Fisica è stato ideato dal Consiglio di Corso di Laurea come un insegnamento che fornisse i primi rudimenti alla programmazione e all'analisi dei dati. Il corso di Laurea in Fisica è in assoluto il percorso di studi che contempla il maggior numero di insegnamenti di Laboratorio tra tutti i corsi di laurea nei vari atenei, e questo è dovuto alla natura sperimentale di questa disciplina. Quindi, questo corso deve necessariamente fornire agli studenti gli strumenti informatici di base per analizzare i dati che vengono raccolti negli esperimenti condotti nei vari corsi di laboratorio.
Il corso inoltre deve rendere gli studenti appena iscritti al corso di laurea e provenienti da diversi scuole capaci di tradurre i metodi di risoluzione dei problemi in algoritmi interpretabili ed eseguibili da un calcolatore. Queste abilità sono fondamentali per affrontare al meglio gli altri corsi e propedeutiche ai corsi di Fisica Computazione e di Analisi Dati che vengono tenuti nel corso di Laurea Magistrale in Fisica dell'Università di Messina.
Il linguaggio di programmazione che sarà utilizzato nel corso è il C++ standard. I motivi sono molteplici, questo linguaggio di programmazione può essere utilizzato sia in programmi ad alto-livello che a basso-livello (alto e basso si riferiscono a quale livello il linguaggio riesce a interagire con la macchina).
Verrà inoltre presentato un parallelo tra la programmazione procedurale in C++, che equivale alla programmazione in C, e quella in Fortran in modo da consentire agli studenti di comprendere un codice scritto in questo linguaggio, modificarlo o re-implementarlo. Il Fortran è un linguaggio di programmazione molto potente per il calcolo scientifico ed ancora oggi esiste una varietà infinita di programmi scritti in questo linguaggio principalmente da Fisici.
Il corso è rivolto a futuri Fisici, quindi i metodi numerici e la formulazione di algoritmi originali saranno privilegiati rispetto alle spiegazioni meticolose sulle infinite possibilità offerte dal dato linguaggio di programmazione. In altre parole, verrà dato maggiore riguardo alla programmazione che al linguaggio, e di quest'ultimo sarà trasmesso giusto l'essenziale necessario per implementare i propri algoritmi.
Un terzo delle ore di questo corso sarà riservato all'apprendimento dei primi rudimenti del framework di analisi dati ROOT (software libero rilasciato dal CERN interamente governabile con il C++) e di un programma leggero e potente come gnuplot, che forniranno agli studenti strumenti potenti per l'analisi e la rappresentazione grafica dei dati che raccoglieranno nel corso di Laurea e nella loro futura vita di professionisti della scienza. Root è una delle piattaforme software più potenti e utilizzate nel campo della Fisica e non solo, mentre gnuplot è un programma leggero e potente per la grafica e l'analisi elementare dei dati capace di funzionare su qualunque dispositivo.
Il sistema operativo di riferimento del corso è Linux. Nel presente corso sarà utilizzato solo software libero, essenzialmente codici sorgente, librerie e compilatori. Linux è l'ambiente ideale dove trovare tutto quello che serve al riparo da virus e dalla dipendenza da programmi commerciali i cui codici sorgenti sono inaccessibili.
In questa raccolta di appunti, ci sono numerosi esempi e i codici implementati durante il corso. Nei codici ci sono numerosi commenti che devono essere considerati parte integrante e fondamentale di questo testo.



disegno di : Bruno Touschek

Indice

I Linux



disegno di : Bruno Touschek

1. Comandi base

Linux è un sistema operativo libero, potente e competitivo con quelli commerciali. Esistono diverse famiglie e distribuzioni, per il corso non si consiglia una versione in particolare, l'unica avvertenza è che la versione installata sia stabile e gradita all'hardware del computer che si utilizza. Nel Laboratorio di Informatica del Corso di Laurea in Fisica la distribuzione in uso è Ubuntu, le macchine sono stabili e ben manutenute.

In questo capitolo verranno descritti i comandi base per iniziare a lavorare sul terminale. Si, assolutamente si, avete letto bene lavoreremo sul terminale e il mouse e l'interfaccia grafica li utilizzeremo pochissimo. In questo momento vi state chiedendo se abbiamo intenzione di trasportarvi nel passato, ma vi assicuro che è tutto l'opposto. L'intento del corso è di portarvi nel futuro. Il passato è fatto di interfacce grafiche accattivanti che vi nascondono ciò che succede realmente nella macchina e che vi condannano all'ignoranza e alla dipendenza. Quindi un po' di fiducia e il divertimento non tarderà ad arrivare.

1.1 Shell-iniziare a muoversi sulla riga di comando

In figura 1.1 è presente un dettaglio del terminale dei comandi offerto da un sistema operativo di tipo Linux, in questo caso specifico si tratta di Opensuse. Dal terminale anche detto "shell" è possibile attraverso la chiamata diretta di una serie di comandi (programmi offerti dal sistema

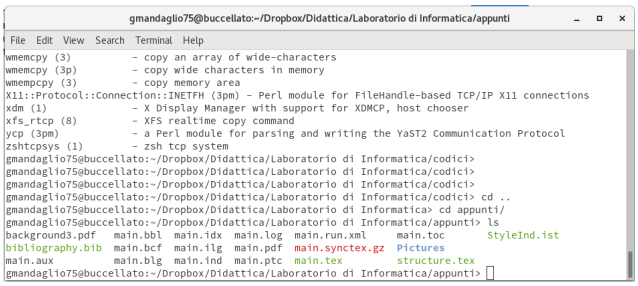


Figura 1.1: Dettaglio del terminale.

operativo - molti ereditati da UNIX) effettuare diverse operazioni: copiare, creare cartelle, muoversi tra le cartelle, leggere documenti, cancellare documenti o cartelle etc.

Ogni comando può avere diverse opzioni, per accedere a queste opzioni bisogna scrivere di seguito il comando spazio-bianco opzione(in genere una lettera) spazio-bianco argomento-su-cui-agisce-il-comando.

Per interrogare il "terminale" circa le opzioni del comando è sufficiente scrivere **man** spazio-bianco nome-del-comando.

Di seguito ne elenchiamo i più comuni e frequentemente usati:

ls stampa a schermo la lista dei file e delle directory contenute nella directory in cui il comando viene invocato. le opzioni -a vi fa vedere anche i file nascosti (su Linux il loro nome inizia con un punto) -l fornisce una lista dei file con più informazioni.

cd acronimo di *change directory* serve appunto a cambiare la directory corrente, a spostarsi dentro la memoria del pc. Per entrare in cartelle e sottocartelle basta separare in fase di comando i nomi delle cartelle con uno slash /. Per scendere di una cartella cd ../.

cp serve per copiare i file. Per copiare le cartelle bisogna utilizzare l'opzione -r (copia ricorsiva). All'interno della stessa cartella bisogna dire al comando il nome del file da copiare e il nome della copia (che può essere differente dal nome del file da copiare) nel caso in cui si copia un file in un altro posto (secondo parametro del comando è il percorso di dove si vuole copiare il documento) il comando automaticamente creerà una copia con lo stesso nome del file di partenza.

mkdir creare una nuova cartella (directory).

rmdir rimuovere una cartella.

rm rimuovere i file. Con l'opzione -r consente di cancellare cartelle e tutto ciò che contengono. Il comando rm -rf è molto potente e pericoloso, consente di cancellare file e cartelle in modo ricorsivo attraverso l'opzione r, mentre non chiede il permesso di cancellare attraverso l'opzione f (force).

pwd vi dirà la posizione della directory corrente.

chmod consente di cambiare i permessi dei file (lettura=r, scrittura=w esecuzione=x). Grazie a questo comando potete rendere eseguibile un documento di testo contente una serie di comandi interpretabili dal terminale che "lanciato" (./nomedelfile invio) eseguirà in sequenza i comandi contenuti nel file (bash script eseguibile). chmod seguito dal meno per rimuovere il permesso, seguito dal + per conferirlo.

mv utile per muovere i documenti da una cartella ad un'altra, oppure per cambiare il nome di un file.

tar consente di creare file di archivio utili per conservare i propri dati o per condividerli.

gzip consente di comprimere i file.

gunzip consente di decomprimere i file compressi.

df vi fornisce informazioni sullo stato della memoria del vostro pc.

Questa lista si potrebbe prolungare per diverse pagine, senza però dare mai una descrizione completa di tutte le funzionalità dei vari comandi. Il consiglio migliore per chi comincia a lavorare su un terminale unix-like tipo linux è quello di interrogare il sistema circa le potenzialità del

comando che si intende usare visualizzando a schermo il manuale del comando utilizzando la parola chiave *man* nel seguente modo man-spazio-ilnomedelcomando-invio; e inoltre quello di cercare di imparare più comandi possibili ma non con la smania di un collezionista ma con la voglia di migliorare le proprie possibilità. L'unico modo per memorizzare comandi e opzioni è usarli, usarli e usarli. L'unico modo per apprenderne altri è non usare sempre le stesse cose che si conoscono ma cercare se esistono modi alternativi per fare in modo più efficiente e veloce quello che normalmente facciamo (lo so quando si ha fretta è più comodo usare ciò che si conosce, ma prima o poi ci sarà l'occasione in cui non avremo fretta: bene in quelle circostanze bisogna cercare di essere un po' più curiosi e un po' meno pigri).

1.2 Compilatori

I compilatori il più delle volte risultano già installati su sistemi tipo Linux oppure è possibile scaricarli gratuitamente. Sono disponibili praticamente tutti i compilatori dei linguaggi di programmazione: fortran, pascal, C, C++, perl, python, java etc. Il linguaggio di riferimento di questo corso è il C++ e il suo compilatore si invoca attraverso il comando g++ (su mac il comando è cc e la sintassi è la stessa). Verrà inoltre utilizzato il linguaggio fortran e il compilatore di su linux è gfortran. La lettera g che precede questi compilatori sta per gnu. La sintassi e l'utilizzo dei due compilatori è esattamente la stessa.

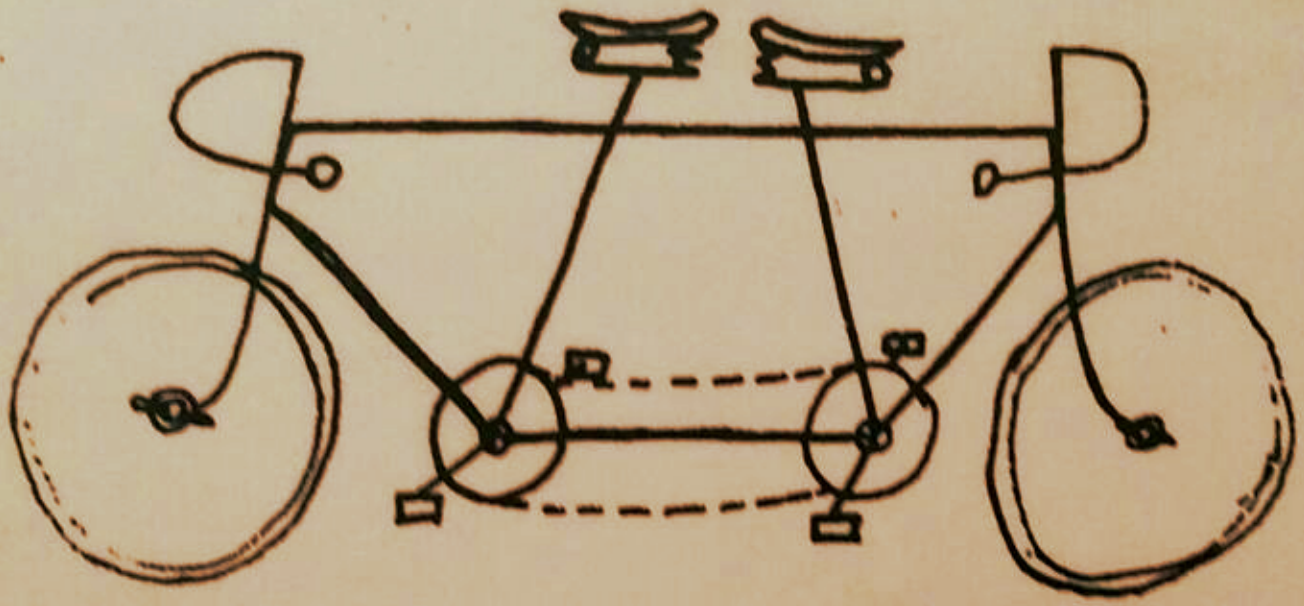
Il compilatore ha il compito di tradurre il sorgente del programma (un documento di testo contenente l'implementazione del programma) scritto e comprensibile all'uomo in un file comprensibile dalla macchina che chiameremo file eseguibile o eseguibile.

Il compilatore quando viene invocato ad analizzare un codice sorgente esegue sempre un controllo degli errori e nel caso in cui ne dovesse trovare qualcuno non completerà la compilazione, non produrrà il file eseguibile e vi mostrerà a schermo una serie di messaggi (numero di riga del codice ed errore) che vi aiuteranno ad individuare l'errore/gli errori che avete commesso (vedi fig. 1.2).

```
gmandaglio75@buccellato:~/Dropbox/Didattica/Laboratorio di Informatica/codici> g++ maxmin.cpp
maxmin.cpp: In function ‘int main()’:
maxmin.cpp:28:1: error: expected ‘;’ before ‘}’ token
 }
 ^
maxmin.cpp:39:31: error: expected ‘;’ before ‘)’ token
 for (int i=0; i<contatore, i++){
                               ^
gmandaglio75@buccellato:~/Dropbox/Didattica/Laboratorio di Informatica/codici>
```

Figura 1.2: Esempio di errori evidenziati dal compilatore. .

II Programmazione in C++



disegno di : Bruno Touschek

2. Introduzione

La programmazione è la capacità di far eseguire a una macchina una serie di istruzioni nel modo più efficiente possibile. La sequenza delle istruzioni la chiameremo **algoritmo**. La correttezza degli algoritmi, l'assenza di errori logici (bug), l'efficienza e la leggibilità del codice rappresentano l'abilità principale del programmatore. Normalmente i programmatori informatici sono formati per implementare codici per soddisfare le esigenze di un committente (un cliente), mentre i programmatori fisici implementano codici per loro stessi o per il gruppo di ricerca con cui collaborano. Sebbene i fisici siano molto apprezzati come programmatori e risolutori di problemi complessi, tuttavia hanno anche la pecca di scrivere codici comprensibili solo a loro e spesso privi di qualsiasi documentazione. Quindi prima di prendere questa brutta abitudine impariamo che i codici devono essere leggibili e ben documentati in modo che anche altri li possano usare.

Riprendendo il sermone precedente notiamo che ci sono alcune parole che ci suggeriscono che per poter programmare abbiamo bisogno di alcune cose....

Alcune cose:

Algoritmo. Questa parola che sembra complicata, in realtà è una delle cose più comuni della nostra vita. Implementiamo algoritmi in continuazione da sempre: alle elementari alle prese con i primi problemini di algebra pieni di contadini, uova, mercatini etc; quando cuciniamo una frittata; quando guidiamo la macchina; quando scriviamo un messaggio sul cellulare etc. Creiamo algoritmi in continuazione creiamo una sequenza di azioni atte a risolvere un problema o a fare una azione e in questi casi il pc normalmente siamo noi stessi.

Purtroppo il pc non possiede un cervello come il nostro e non può risolvere i problemi al posto nostro. Si, purtroppo è così!!! I problemi li dobbiamo risolvere noi. Il pc però è veloce, straordinariamente veloce.

Il pc parla una sua lingua e noi ne parliamo un'altra; come possiamo fare a comunicare le nostre idee geniali in modo che il pc possa aiutarci a risolvere il dato problema??

Ci serve un interprete che traduca i nostri codici in un linguaggio comprensibile alla macchina, un linguaggio con delle regole e degli strumenti che ci consentano di implementare i nostri algoritmi.

I compilatori o gli interpreti sono dei programmi che hanno il compito di tradurre i nostri codici (codice sorgente) in linguaggio macchina. I compilatori traducono interamente il codice sorgente

in codice macchina comprensibile ed eseguibile dal pc, mentre gli interpreti traducono e fanno eseguire al pc una istruzione alla volta. I codici sorgente eseguiti da un interpretate sono molto più lenti, in genere sono di piccole dimensioni e facilmente manutenibili e forniscono completo accesso al codice. Quelli compilati sono invece molto veloci perché eseguibili per intero senza ulteriori traduzioni, non hanno bisogno dei codici sorgenti ne dell'interprete (si può nascondere il sorgente), tuttavia per rimediare ad un errore all'interno del codice è necessario ricompilare il programma e in progetti di grosse dimensioni può richiedere una certa quantità di tempo.

I compilatori hanno capacità di diagnostica sulla correttezza sintattica del codice, vi informano degli errori e non procedono alla creazione un programma eseguibile finché il codice sorgente non viene corretto da tutti gli errori di sintassi. Il compilatore non ci rende immuni dagli errori di logica, quindi scrivere codici corretti è condizione necessaria ma non sufficiente alla risoluzione di un problema con un codice. L'errore più grave che può commettere un programmatore inesperto è subordinare la logica del programma all'esigenza di portare a buon fine un processo di compilazione: sentire pronunciare espressioni del tipo "ho spostato questo frammento di codice perché non compilava" nuoce gravemente la salute di chi ascolta (o nei casi meno gravi può causare una certa acidità di stomaco) quindi dire cose del genere è moralmente deplorevole.

Nel presente corso il linguaggio eletto allo scopo è il C++, la scelta ha svariate ragioni: è un linguaggio a basso e alto livello, consente di strutturare i programmi in modo che siano orientati agli oggetti, è il linguaggio con il quale sono implementati sistemi operativi, linguaggi ad alto livello come ad esempio python. Si presta bene per imparare a programmare in modo procedurale ed ovviamente orientato agli oggetti.

Alcune lezioni saranno spese per imparare a tradurre i codici procedurali implementati in C++/C in linguaggio Fortran. Il Fortran è un linguaggio molto potente e rivolto al calcolo scientifico. Conoscere la sua sintassi è di grande utilità per poter accedere al patrimonio immenso di codici scritti in questo linguaggio dalla comunità dei fisici e ancora diffusamente utilizzati.

Alla fine del corso, una introduzione alle tecniche fondamentali per l'analisi dei dati verrà affrontata utilizzando il framework root, e il programma gnuplot.

Il corso suggerisce caldamente l'utilizzo di un pc con sistema operativo Linux (qualunque distribuzione va bene), o comunque un sistema operativo tipo Unix. Su linux il compilatore (libero) che verrà utilizzato è g++.

2.1 Stuttura di un programma

Di seguito il listato di un semplicissimo programma in C++ che una volta compilato ed eseguito attraverso la sua versione eseguibile dal pc stampa a monitor la stringa di caratteri "Ciaooooooo!".

```
//con il doppio slash inseriamo commenti nel programma
//i commenti sono ignorati dal compilatore
//i commenti sono utili per rendere più leggibile il codice
// Il primo programma in C++
#include <iostream>// include la libreria iostream

int main()
{
  std::cout << "Ciaooooooo!";
  return 0;
}
```

Per compilare questo codice bisogna che esso sia trascritto in un file con estensione cpp (utile perché ci dice che abbiamo a che fare con un codice sorgente di tipo C++ e inoltre informa i

programmi di editor del tipo di file in questione e l'editor colora in modo molto utile le righe del codice aiutandoci nella programmazione). Poi su un Linux da riga di comando si digita :
g++ nomedelsorgente.cpp
il compilatore g++ per default crea il file a.out che contiene il nostro eseguibile. Per eseguire il programma bisogna digitare sul terminale il comando

```
./a.out
```

e poi fare invio.

Ogni volta che eseguiamo una compilazione in questo modo, g++ sovrascriverà il file a.out con il nuovo eseguibile e quindi il vecchio sarà perso. Se invece si vuole conservare una copia dell'eseguibile può essere utile utilizzare l'opzione di g++ che consente di decidere il nome dell'eseguibile.
g++ -o nome_eseguibile sorgente.cpp (la lettera o sta per output della compilazione)

All'interno del codice sorgente è possibile scrivere dei commenti che sono riservati al programmatore e che non vengono interpretati dal compilatore. I commenti sono preceduti da un doppio slash. I commenti sono molto utili a rendere il codice più leggibile e forniscono una breve documentazione al codice.

Procediamo a commentare il programma riga per riga: prima di ogni cosa troviamo una istruzione che inizia con carattere cancelletto e che segue con il comando include. Questa istruzione serve a dire al preprocessore del compilatore che deve appunto includere nel programma la libreria **iostream**. Senza l'inclusione della libreria **iostream** che contiene le funzionalità di input-output non avremmo potuto utilizzare il comando std::cout « "Ciaooooooo!"; e stampare a schermo la stringa "Ciaooooooo!".

Inclusa la libreria, il programma definisce la funzione principale. In C++/C qualunque programma non può prescindere dalla funzione principale, in questo linguaggio tutte le procedure sono funzioni e la funzione che ha il nome main è una funzione speciale e come già detto la più importante. La funzione main ha il compito di coordinare il funzionamento del programma. Usiamo la funzione **main** per introdurre la struttura di una funzione.

Ogni istruzione deve essere chiusa con **un punto e virgola** ad eccezione dell'inclusione delle librerie o quando raggruppiamo istruzioni utilizzando le parentisi graffe.

```
// int è il tipo di valore che la funzione restituisce
// se la funzione non restituisce nulla si può mettere void al posto di int
// all'interno delle parentesi ci sono le variabili che riceve
//possono essere di diverso tipo come nell'esempio
//oppure non esserci nulla, la funzione non riceve nulla
int nomefunzione (int a, float b, double c, ...etc)
{
//all'interno delle graffe ci sono le istruzioni
istruzione 1;
// tutte le istruzioni chiudono con un punto e virgola
// eccetto l'istruzione #include
istruzione 2;
.....etc;
//alla fine dell'esecuzione, o al suo interno
// l'operazione return restituisce un valore
// del tipo dichiarato ed esce dalla funzione
// se la funzione in questione è la funzione
// principale il programma termina
// se è una funzione qualunque restituisce il dato valore
```

```
// alla funzione principale e il programma continua a "girare".
return variabile_che_restituisce;
}
```

Ritorniamo al programma, l'istruzione *std* :: *cout* << "*Ciaooooooo*!"; utilizza la classe **cout** della libreria **iostream** per stampare a schermo la frase "*Ciaooooooo*!". L'operatore minore-minore << dice che la stringa "*Ciaooooooo*!", definita attraverso le virgolette (senza virgolette Ciaooooooo sarebbe una variabile) dal programmatore viene data in input al programma **cout** e il programma **cout** lo stampa sullo schermo.

cout è un oggetto che ci consente di scrivere sullo schermo, possiamo scrivere parole o frasi scrivendole tra virgolette, oppure variabili e in questo caso il valore della variabile sarà stampata a schermo. es.

```
std::cout << "il valore del nostro calcolo e' = "<< valore<<"\n";
```

Nell'esempio, **cout** stamperà a monitor se valore fosse uguale a 1.23
il valore del nostro calcolo è = 1.23
e poi sarebbe andato a capo perché abbiamo messo il comando tra virgolette slash n (al posto di questo comando si sarebbe ottenuto lo stesso effetto utilizzando l'espressione **endl**). Se non chiediamo a **cout** di andare a capo, **cout** non lo farà automaticamente e scritture multiple appariranno a schermo sulla stessa riga.

Includendo tra le inclusioni delle varie librerie *#include* e la funzione principale il comando:
using manespace std;
si evita il bisogno di scrivere davanti ai membri della libreria il prefisso *std* ::.

I programmi vengono eseguiti in modo sequenziale dalla prima all'ultima istruzione, quindi leviamoci dalla testa che il seguente pezzo di codice possa restituire il valore 5:

```
int a,b,c;
c=a+b;
a=2;
b=3;
cout<<"non restituisce 5 manco se ti metti a piangere :P  ="<< c <<endl;
```

il codice corretto sarebbe:

```
int a,b,c;
a=2;
b=3;
c=a+b;
cout<<"grazie così va bene, la somma di a e b è  ="<< c <<endl;
```

2.2 Operazioni di input-output I

Il programma seguente lo utilizziamo per introdurre l'uso delle variabili. Nell'esempio le variabili si chiamano a,b, e results e sono tutte variabili di tipo intero. Il programma assegna i valori alle due variabili (5 e 2), poi somma la quantità 1 alla variabile a e poi fa la differenza tra a e b e assegna questo valore alla variabile result, infine stampa a monitor il risultato della differenza attraverso la variabile result.

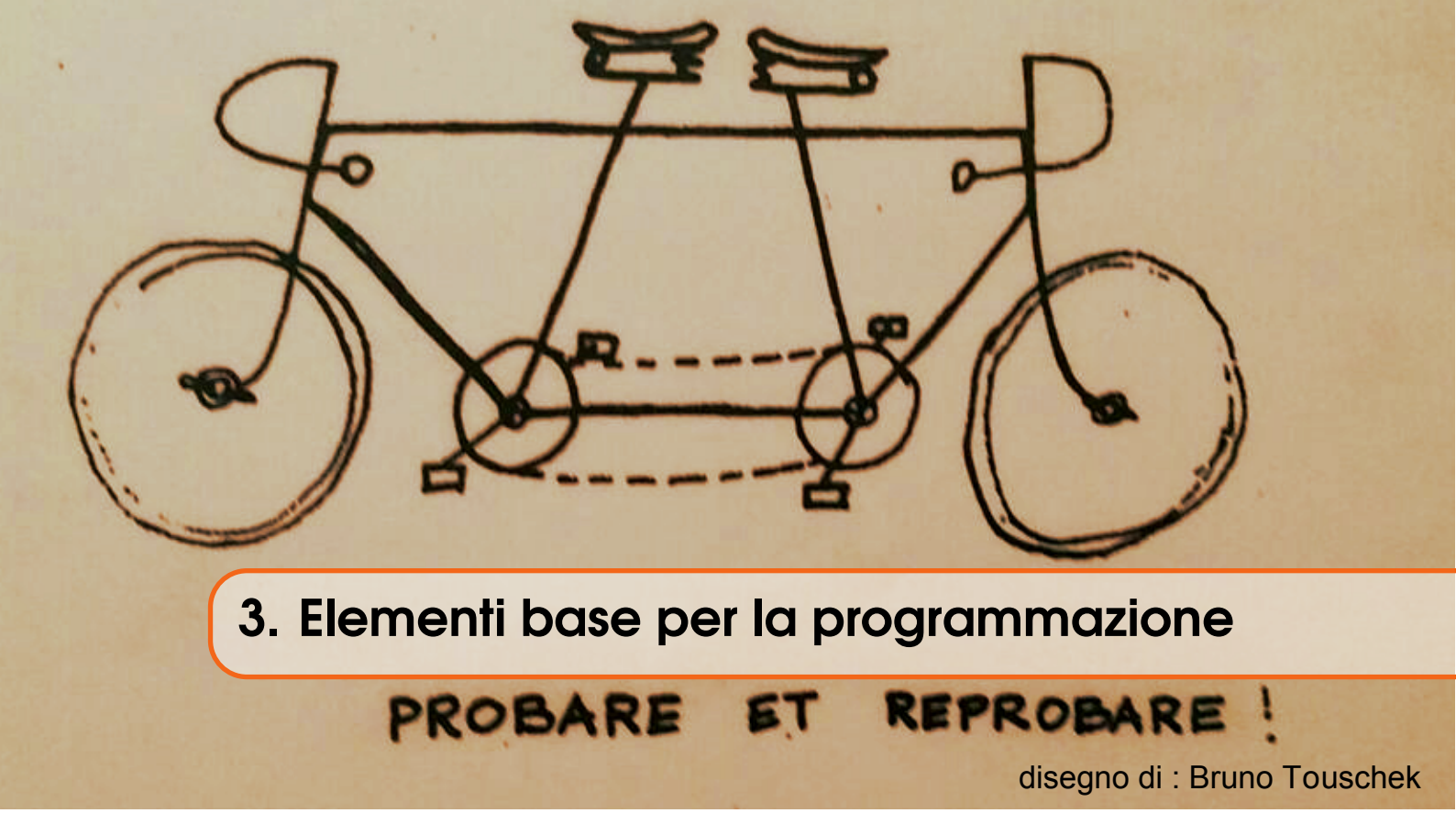
```
// operating with variables
```

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

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // print out the result:
  cout << result;

  // terminate the program:
  return 0;
}
```

Ora immaginiamo di voler confezionare un programma capace di chiedere ad un utente di fornirgli da riga di comando due numeri, e lui in cambio li somma e restituisce a schermo il risultato. Per fare questo ci serve un qualche cosa che operi al contrario di **cout**. Il metodo che fa al nostro scopo è **cin**. **cin** funziona sintatticamente in modo equivalente a **cout** solo che i minore-minore si trasformano in maggiore-maggiore ad indicare che **cin** prende qualche cosa dal terminale e lo assegna ad una variabile nel programma. In pratica **cin** e **cout** si comportano come due messaggeri che fanno parlare il mondo del programma con quello esterno del terminale. Più avanti presenteremo anche i loro fratelli omologhi del C printf e scanf, utilizzabili nel C++ che assorbe completamente C.
es. **cin»variabile;** //assegna ad variabile il valore digitato al terminale dopo l'invio
es. **cin»variabile1»variabile2;** // in questo caso due valori. Per passarli a **cin**, abbiamo due strade, digitiamo un valore poi invio e così per il secondo oppure digitiamo entrambi i valori separati da uno spazio bianco e poi invio.

```
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  cout << "potresti darmi un numero?"<<endl;
  //endl è un modo alternativo per andare a capo
```

```
  cin  >> a;
  cout << "potresti darmi un'altro numero?"<<endl;
  cin >> b;
  cout << "Grazie!"<<endl;

  result = a + b;

  // print out the result:
  cout << "ecco il risultato della somma =" <<result<<endl;
  cout << "ecco il prodotto occhio al codice ="<< a*b<<endl;

  // terminate the program:
  return 0;
}
```



disegno di : Bruno Touschek

3. Elementi base per la programmazione

3.1 Variabili I

Le variabili in qualunque linguaggio di programmazione possono essere di diverso tipo: intere, reali, complesse, caratteri, stringhe (insiemi di caratteri) etc. Le variabili vanno sempre dichiarate, questa operazione è fondamentale perché bisogna prenotare le locazioni di memoria capaci di ospitare il dato. Se le variabili non vengono dichiarate il compilatore segnalerà la cosa come un errore: il C++/C non sono dotati di un sistema automatico di assegnazione delle variabili (python ad esempio è dotato di un sistema di assegnamento automatico, ma phython è un linguaggio al alto livello e tutto quello che ci risparmia è dovuto a implementazioni che fanno il lavoro al posto nostro. python è implementato in c++).

Il codice seguente:

```
#include <iostream>// include la libreria iostream

int main()
{
  pi_greco=3.14;
  return 0;
}
```

produrrebbe il seguente errore in fase di compilazione

```
gmandaglio75@buccellato:~> g++ prova.cpp
prova.cpp: In function ‘int main()’:
prova.cpp:6:3: error: ‘pi_greco’ was not declared in this scope
   pi_greco=3.14;
   ^~~~~~~
```

Come si può vedere in tabella 3.1 ci sono diversi tipi di variabile che possono essere utilizzate dentro i nostri codici. É importante notare che la grandezza delle variabili è considerevole nelle

Gruppi	Tipi di carattere	note
Caratteri	char	1 byte, almeno 8 bits
	char16_t	non più piccolo di un Char, almeno 16 bits.
	char32_t	non più piccolo di un char16_t, almeno 32 bits.
	wchar_t	il più grande tipo di carattere supportato.
Tipo intero con segno	signed int	da -2147483648 a 2147483648 (32 bit)
	signed long int	da -2147483648 a 2147483648
	signed long long int	da -9223372036854775808 a 9223372036854775808
Tipo reale	float	da 1.17549×10^{-38} a $3.40282\times10^{+38}$
	double	da 2.22507×10^{-308}a $1.79769\times10^{+308}$
	long double	da 3.3621×10^{-4932} a $1.18973\times10^{+4932}$
Tipo Booleana	bool	assume i valori 1 - 0 o “true” - “false”
Tipo vuoto	void	nessuno spazio di memoria occupato

Tabella 3.1: Lista di variabili disponibili in C++, nome e dimensioni.

variabili di tipo *long*, ma non infinita. Il concetto di infinito è per gli esseri umani non per le macchine.

3.2 Operatori

3.2.1 Assegnazione =

L’operatore di assegnazione è =. Lo abbiamo già utilizzato negli esempi precedenti. É importante sottolineare che questo operatore esegue solo l’operazione di assegnazione ovunque esso venga invocato, non comparazione, non risponde a nessuna domanda! Scrivere che a=b; o a =25; significa che la variabile a assumerà lo stesso valore della variabile b oppure il valore 25 nel secondo caso.

3.2.2 Operatori aritmetici (+, -, *, /,%)

Gli operatori somma, sottrazione, prodotto(simbolo asterisco *), divisione (simbolo slash /), e modulo (simbolo %) ci consentono di lavorare su numeri e variabili. Bisogna ricordarsi che gli operatori obbediscono ad un ordine gerarchico che dá a prodotto/divisione/modulo priorità di esecuzione sulle operazioni di somma e sottrazione. Quindi la variabile pippo nel seguente frammento di codice assumerà il valore 11 piuttosto che 21.

```
int pippo = 5 + 2 * 3;
```

L’operatore modulo % restituisce il resto della divisione tra due numeri. Questa operazione è molto utile se vogliamo stabilire se un numero intero memorizzato in una variabile sia pari o dispari, per fare questo basta verificare se il valore restituito dall’operazione variabili%2 sia pari a zero (pari) o a 1 (dispari).

3.2.3 Operatori di assegnazione composti (+=, -=,*=,/=)

In C++ è possibile comporre gli operatori di assegnazione con altri creando delle scritture abbreviate come nella tabella seguente:

3.2.4 Operatori di incremento e decremento

Gli operatori ++ e −− sono detti operatori di incremento e di decremento, nel senso che aumentano o diminuiscono di una unità la variabile alla quale vengono applicate.

```
x++; // equivale a x = x +1;
x--; // equivale a x = x -1;
```

espressione	equivalente a...
y += x;	y = y + x;
x -= 5;	x = x - 5;
x /= y;	x = x / y;
x *= y;	x = x * y;

Tabella 3.2: Equivalenza tra assegnazioni composte.

é importante notare il verso dell'applicazione dell'operatore nel caso in esso venga utilizzato nelle espressioni:

```
x=20; //
y = x++; // in questo caso y è uguale 20 e x a 21
//mentre!!!!!!!!!!
y = ++x;  //in questo caso y e x sono uguali a 21
```

Nel primo caso prima viene effettuata l'operazione di uguaglianza e poi quella di incremento producendo una differenza tra le due quantità alla fine dell'operazione, nel secondo caso invece prima si incrementa e poi si eguaglia.

3.2.5 Operatori di comparazione

Gli operatori di comparazione ci aiutano a stabilire la veridicità di una comparazione tra variabili e grandezze. Se la relazione scritta è soddisfatta restituirà il valore *vero* altrimenti il valore *falso*. Queste espressioni saranno di fondamentale importanza per le strutture di controllo e per il ciclo *while* che vederemo in seguito.

operatore	descrizione
==	Uguale
!=	Diverso
<	Minore di
>	Maggiore di
<=	Minore uguale a
>=	Maggiore uguale a

Tabella 3.3: Operatori di comparazione.

3.2.6 Operatori di logici

Gli operatori logici ci consentono di combinare delle condizioni, e a seconda della loro veridicità restituiranno un valore vero o falso. L'operatore OR è realizzato legando due condizioni con un *doppio pipe* mentre la condizione AND con una *doppia e commerciale*, mentre la negazione è ottenuta con il punto esclamativo.

operatore	descrizione
&&	AND
||	OR
!	NOR

Tabella 3.4: Operatori di logici principali.

L'operatore **AND** restituisce un valore vero se e soltanto se entrambe le condizioni sono vere, mentre l'**OR** restituisce vero se almeno una delle due condizioni è vera.

3.3 Strutture di controllo

Le strutture di controllo ci consentono di inserire nei nostri programmi la possibilità di scegliere se eseguire alcune istruzioni oppure altre a seconda della veridicità di una condizione. La condizione può essere una variabile logica booleana, o una espressione composta con gli operatori di comparazione, oppure il valore (0 o 1) restituito da qualcuno (va bene, va bene, non vi seccate: questo qualcuno potrebbe essere una classe o una funzione ma lo vedremo in seguito). Le parole chiave che ci consentono di attivare una tale struttura sono **if** ed **else**, la prima significa **se** mentre la seconda **altrimenti**. L'opzione **else** non è obbligatoria, e se non si usa l'istruzione **if** porrà condizione alla istruzione o all'insieme di istruzioni racchiuse tra graffe che la seguono. Se dopo l'**if** o dopo l'**else** se si vuole far seguire una sola istruzione allora in questo caso le parentesi graffe possono essere omesse. La condizione controllata dall'**if** deve essere racchiusa tra parentesi tonde, mentre l'**else** eseguendo ciò che la segue se la condizione dell'**if** non è "vera" non ha bisogno di una sua condizione tra parentesi tonde.

```
\\struttura del costrutto if ... else

if(condizione) {
istruzione 1;
istruzione 2;
....

istruzione n;
}
else{
Istruzione 1;
Istruzione 2;
....

Istruzione n;
}
```

Un primo esempio semplice, che richiama l'uso dell'operatore modulo % è quello di determinare se un numero dentro un codice sia pari o dispari.

esempio:

```
int pippo;
.....
......
....... (a un certo punto pippo assumerà un valore)

if(pippo%2==0)
cout<<"pippo è pari";
else
cout<<"pippo è dispari";
```

Un secondo esempio, familiare a masticatori di numeri come i fisici è quello in cui si vuole indicare l'interno di un intervallo oppure il suo esterno. Un caso molto semplice che possiamo considerare è quello di un intervallo fissato da due valori su un asse di un sistema di riferimento cartesiano. Grazie a questo esempio possiamo esercitarci con la struttura di controllo e nel definire la condizione della struttura con gli operatori di confronto e gli operatori logici. L'esempio può

essere formulato in modo equivalente utilizzando l'operatore **AND** (che individua l'interno) oppure l'**OR** (che individua l'esterno).
esempio **AND**:

```
// immaginiamo su un ascissa un intervallo tra i valori 2 e 7
float x;
....
.....
....
if((x>=2) && (x<=7))
cout<< "x è interno all'intervallo"<<endl;
else
cout<< "x è esterno all'intervallo"<<endl;
```

esempio **OR**:

```
// immaginiamo su un ascissa un intervallo tra i valori 2 e 7
float x;
....
.....
....
if((x<2) || (x>7))
cout<< "x è esterno all'intervallo"<<endl;
else
cout<< "x è interno all'intervallo"<<endl;
```

3.4 Cicli: while, do ...while,for

Nella programmazione una operazione molto utile e ricorrente è quella di ripetere una istruzione o un gruppo di istruzioni un certo numero di volte.

Un esempio molto semplice che ci può aiutare a comprendere l'utilità dei cicli è l'operazione di somma. É vero che se dobbiamo sommare due numeri non è un grande problema, e neanche se fossero 10 se invece fossero 20 sarebbe orribilmente noioso ma ce la potremmo fare ma se le quantità che dobbiamo sommare fossero 100, o 1000 o 1000000 etc. In questo caso il modo più semplice per non scrivere una istruzione lunga un chilometro, e cavarsela con poche righe di codice è ricorrere a un ciclo. L'istruzione potrebbe consistere di una operazione che somma un addendo alla volta e lo accumuli in una variabile contenitore inizializzata a zero (elemento neutro dell'operazione di somma) prima dell'inizio del ciclo.

Come esempio implementeremo un programma che ci chiede il voto che abbiamo conseguito in tutti i corsi che abbiamo superato e una volta che abbiamo finito di darglieli ci restituirà la media:

```
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables:
  float somma, voto;
  int n_materie_superate = 0;
```

```
   somma = 0;

  cout << "questo programma calcola la media"<<endl;
  cout << "dei voti che hai conseguito"<<endl;
  voto=1; //per innescare il ciclo while
  while (voto > 0){
  cout << "dimmi il voto o scrivi 0 per chiudere"<<endl;
  cin  >> voto;
  if( voto > 0)
  {
   somma = somma + voto ; // avremmo potuto scrivere somma + = voto;
   n_materie_superate++; // incrementiamo di una unità
  }
  }
  // print out the result:
  cout << "La tua media e' =" <<somma /n_materie_superate <<endl;
  return 0;
}
```

Nell'esempio precedente abbiamo implementato un ciclo utilizzando il costrutto del **while**, questa parola significa *finchè*e consente di ripetere le istruzioni che lo seguono finché la condizione contenuta tra le parentesi tonde che lo seguono è soddisfatta (o in altre parole è vera). Il ciclo **while** consente di costruire cicli che non si conosce a-priori la durata, e come abbiamo visto nell'esempio precedente abbiamo avuto bisogno di un "innesco" che consentisse al ciclo di partire. Una volta partito, le sorti del ciclo dipendono dalle istruzioni che lo seguono. Questo costrutto è molto potente, e come dice lo zio dell'Uomo Ragno "da grandi poteri discendono grandi responsabilità" e quindi è anche molto pericoloso. A parte gli scherzi, se il **while** non è implementato in modo corretto può trasformarsi in un ciclo infinito dal quale non è possibile uscire.

Se volessimo ovviare al problema dell'innesco potremmo usare il costrutto **do-while** il quale in ogni caso almeno una volta eseguirà le istruzioni che il costrutto contiene:

```
do{
istruzione 1;
istruzione 2;
....
....
....
}while(condizione)
```

Se invece vogliamo implementare un ciclo e conosciamo a-priori il numero di volte che lo vogliamo ripetere, allora in questo caso possiamo utilizzare il costrutto del **for**. Il costrutto del ciclo è il seguente:

```
for(int indice; indice< fine; indice++){
istruzione 1;
istruzione 2;
....

....
....
}
```

Anche per il **for** se l'istruzione che si deve ripetere è una sola si possono omettere le parentesi graffe, tuttavia per ragioni legate alla leggibilità e alla manutenibilità del codice è **buona educazione** mettere sempre le parentesi graffe anche nei casi in cui si possono omettere. La disinvoltura nel rimuoverle quando è possibile farlo crea codici meno leggibili e che in caso di modifica spesse volte presentano errori di logica. Nel presente testo si cercherà di non violare mai questa buona regola di scrittura del codice, nel caso accadesse prego a chiunque lo noti di segnalarmelo.

Le implementazioni dei cicli sono equivalenti tra di loro. Nell'esempio che segue il codice stampa a schermo i numeri da 0 a 9 utilizzando in modo equivalente i due costrutti.

```
for(int i =0 ; i<10; i++) cout<<i<<endl;
//oppure
int i =0;
while(i<10){cout<<i<<endl; i++;}
```

Un esempio di utilizzo del ciclo **for** utile quando si vuole generare una sequenza di numeri è dato dal seguente codice che stampa a schermo 20 valori a passo costante da 0 a 3.14.

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

main (){
float pi=3.14;
float n=0;
cout<<"Sequenza dei numeri da 0 a pi-greco"<<endl;
for(int i=0; i<20; i++){
n=n+(pi/20);
cout<<n<<endl;
}
return 0;
}
```

3.5 Esercitazioni - Algoritmi

3.5.1 Integrale di una funzione con il metodo dei rettangoli

In questo paragrafo utilizzeremo un esercizio svolto in classe per fare un po' di riassunto delle cose finora apprese. L'esercizio consiste nel calcolare l'integrale della funzione $f(x) = x^2$ nell'intervallo tra 0 e 5 con il metodo numerico dei rettangoli. Questo metodo numerico è probabilmente il più semplice per stimare numericamente un integrale definito. Esso consiste nell'approssimare l'area sottesa alla curva con la somma dei rettangoli costruiti in modo che la loro base sia pari all'intervallo di integrazione diviso per il numero di divisioni che vogliamo utilizzare per il calcolo e la loro altezza sia pari al valore assunto dalla funzione nel valore medio del sotto intervallo in considerazione. In Figura 3.1 sono rappresentati la curva in questione e i rettangoli costruiti per 10, 20, 30, 40 divisioni dell'intervallo di integrazione. Dalla figura intuiamo che all'aumentare del numero di divisioni aumenta la precisione dell'integrale. Come funziona questo metodo? L'idea di base consiste nel fatto che l'area del rettangolo rappresenta una buona stima di quella ottenuta sostituendo alla base superiore del rettangolo la curva descritta dalla funzione. Se poniamo attenzione all'intersezione tra curva e rettangolo notiamo la formazione di due figure simili a

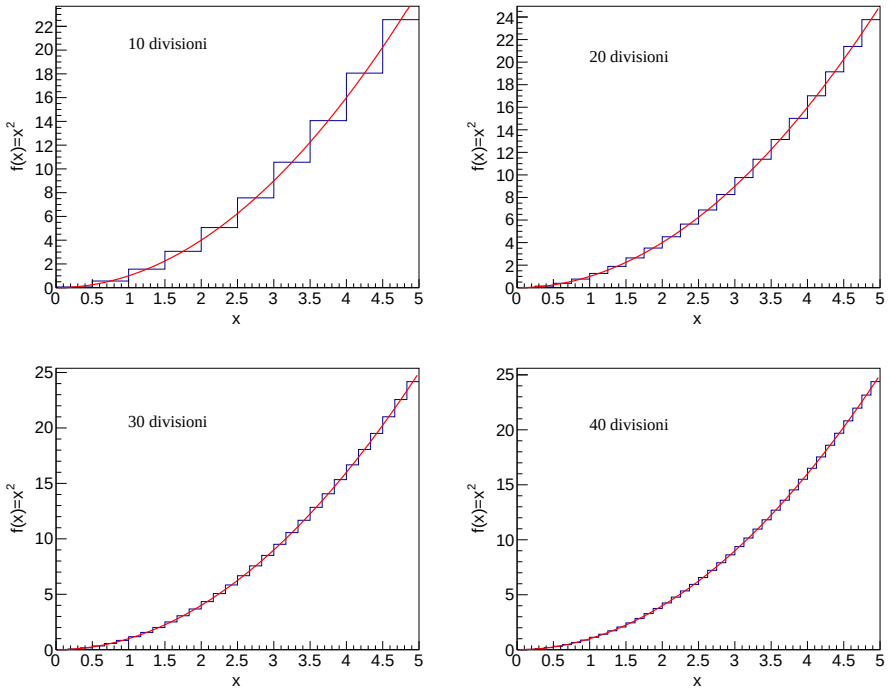


Figura 3.1: Rappresentazione del metodo dei rettangoli per l'integrale della funzione $f(x) = x^2$ tra 0 e 5, utilizzando 10, 20, 30 e 40 divisioni.

"triangoli", uno superiore alla curva che introduce una sovrastima all'area e uno inferiore che la sottostima. L'algoritmo sfrutta il fatto che queste due aree si compensano.

Sfruttiamo questo esempio per introdurre l'utilizzo di una nuova libreria. Quando abbiamo bisogno di utilizzare funzioni matematiche come seno, coseno, tangente etc o operazioni tipo l'elevamento a potenza abbiamo bisogno di caricare la libreria che ci abilita a poter utilizzare queste funzionalità del linguaggio di programmazione. La libreria matematica standard si chiama *cmath*, e nel nostro esempio ci avvaleremo della funzione *pow* (diminutivo di power) per elevare alla seconda potenza una variabile. Il costrutto di *pow* è il seguente:

```
y = pow(base, potenza);
```

Notare che la potenza può essere anche un numero razionale, ciò ci consente di utilizzare questa funzione per definire le operazioni di radice a qualunque ordine: potenza =1/2 - radice quadrata; potenza = 1/3 radice cubica; potenza = 7/5 - radice quinta di potenza 7 etc.

NOTA: Le funzioni trigonometriche sin(var); cos(var); tan(var) etc vogliono come informazione l'angolo espresso in radianti, quindi se avete dati espressi in angolo bisogna prima convertirli in radianti dividendo l'angolo per 180° e moltiplicando per π-greco.

Di seguito i codici implementati durante il corso da due allieve, uno con una implementazione standard e uno con una scrittura sintetica del ciclo **for**:

```
//integrale di x^2
#include <iostream>
using namespace std;
main (){
float a=0, b=5;
float x; //misura dell'intervallo
```

```
float m; //punto medio di ogni intervallo
float Area=0;
x=(b-a)/100;
m=a+x/2;
for(int i=0; i<100; i++){
Area=Area+(x*(m*m));
m=m+x;
}
cout<<Area<<endl;
return 0;}
```

Nel codice seguente è molto bella la implementazione del ciclo **for** proposta da una studentessa del corso, probabilmente un esperto di programmazione potrebbe obbiettare che un codice compatto e furbo potrebbe essere poco leggibile, ma nonostante l'obbiezione legittima ho voluto mostrare come esempio questa implementazione perché anche la creatività ha un suo valore e deve essere incoraggiata, anche se poi un collega che non riesce a capire cosa abbiamo scritto ci dedicherà una litania di benedizioni.

```
#include <iostream>
#include <cmath>
using namespace std;
float somma=0;
float x=0;
float a=0;
float b=5;
float m;
int main() {
m=(b-a)/7527;
x=m/2;
for(x;x<=5;x=x+m){
  somma= somma+m*pow(x,2);
}
cout<<"L'integrale è  "<<somma<<endl;
return 0;
}
```

Ricerca dinamica del numero di divisioni

L'esercizio del calcolo dell'integrale di una funzione semplice e regolare come una parabola ci fornisce la possibilità di implementare un nuovo codice, molto divertente, che ha bisogno dell'implementazione di un ciclo *while* per rispettare le specifiche del problema.

Il nuovo esercizio consiste nello scrivere un programma capace di stimare il numero minimo di divisioni dell'intervallo di integrazione per raggiungere un certo grado di precisione. La funzione in oggetto è molto semplice, ma anche fosse stata più complicata a patto di essere capaci di integrarla analiticamente, possiamo facilmente integrarla in modo esatto e a questo punto spostare l'oggetto del nostro desiderio dall'integrazione alla precisione dell'integrazione.

```
#include <iostream>
#include <cmath>
using namespace std;
int main (){
```

```
float a=0, b=5;
float j=1;
float x;
float m;
float Area=0;
while((fabs(125./3.-Area))>0.001){
  j++;
  x=(b-a)/j;
  m=a+(x/2.);
  Area=0;
  for(int i=0; i<j; i++){
    Area=Area+(x*m*m);
    m=m+x;
  }
}
cout<<"L'area e': "<<Area<<endl;
cout<<"Il numero di intervalli minimo e': "<<j<<endl;
return 0;
}
```

3.6 ifstream e ofstream: lettura e scrittura su file

Per leggere e scrivere su file il c++ fornisce due strumenti appartenenti alla librerie fstream[1] che si chiamano **ifstream** e **ofstream**. É facile memorizzare che **ifstream** serve per leggere, infatti la i sta per input e la f per file, mentre **ofstream** serve per scrivere su file, la o ovviamente sta per output.

Mostriamo di seguito il costrutto dei due operatori, prima iniziamo dall'operatore di lettura:

```
ifstream leggimi;
leggimi.open("nomedelfiledilettura.dat");
float dato;
leggimi>>dato;
cout<<"ecco cosa abbiamo letto: "<<dato<<endl;
leggimi.close();
```

Poi mostriamo quello dell'operatore di scrittura:

```
ofstream scrivimi;
scrivimi.open("nomedelfilediscrittura.dat");
for(int i=0;i<100;i++){
   scrivimi<< i <<endl;
}
scrivimi.close();
```

Osservando i due costrutti possiamo notare che i due operatori differiscono per il nome e la loro funzione, ma il loro utilizzo è molto molto simile. Infatti la sintassi di apertura e chiusura del file[2] è la stessa mentre l'operatività degli operatori è identica a quella delle operazioni di input-output da/su terminale **cin** e **cout** attraverso gli operatori $>>$ e $<<$.

[1] NOTA BENE: ci dobbiamo ricordare di mettere in testa la scrittura #include <fstream> altrimenti non possiamo usare questi strumenti.

[2] Nota bene: Chiudere il file è importantissimo. Lasciare un file aperto significa dare accesso al sistema operativo alle locazioni di memoria riservate al documento con la possibilità che questo possa essere danneggiato.

Ogni operazione di lettura o di scrittura muove il punto di lettura al successivo valore che deve essere letto o scritto: in altre parole il computer legge o scrive in maniera progressiva nel file. Per "riavvolgere il nastro" ovvero per spostare il punto di lettura/scrittura ci sono apposite funzionalità (**seekg**, **seekp**).

Le classi **ifstream** e **ofstream** hanno diverse funzionalità alle quali si accede attraverso l'operatore . (si proprio così l'operatore punto). Abbiamo visto in azione già l'operazione di apertura e chiusura del file, e negli esempi che seguono vedremo le operazioni **eof()** che ci dice se si è arrivati alla fine del file o meno, le funzioni **clear()**, **seekg** e **seekp** che servono per spostare il punto di lettura o scrittura del file (li useremo per riportare il punto di lettura all'inizio del file dopo aver effettuato una prima lettura).

3.6.1 Lettura di un file di un numero incognito di quantità

Una operazione molto utile è quella di essere capaci di leggere un file di dati il cui numero ci è incognito. A questo punto potremmo fare l'obiezione che è difficile immaginare una simile evenienza. In realtà non è così. Immaginiamo di avere il compito di analizzare dati che provengono da una stazione di monitoraggio di quantità fisiche, per esempio una stazione che monitora la temperatura, o la pressione o la radioattività o etc e che salva queste letture su un file di testo e che effettua una lettura ogni certa quantità di tempo. In un certo momento della giornata scarichiamo il file contenete questi dati e iniziamo ad analizzarli.

Il numero di dati contenuti non ci è noto, e noi non abbiamo il tempo e il modo di aprire il documento e di contare il numero di dati che il file contiene. La soluzione è scrivere un programma capace di farlo in maniera automatica, di fornirci il numero di dati che abbiamo letto, e quindi procedere con delle semplici analisi.

lettura del X-file

Per risolvere questo problema faremo uso di un ciclo **while** che ci consente di implementare un ciclo con numero di ripetizioni variabili e dipendenti dalla condizione che lo governa e la funzione **eof()**[3] dell'operazione **ifstream** che ci dice se ci troviamo alla fine del file oppure no. L'operazione **eof()** risponde in modo diretto alla domanda, nel senso che restituisce 1 (vero) se è arrivato alla fine e 0 (zero) in caso contrario. Per usare questo comportamento come condizione per governare il ciclo **while** dobbiamo usare l'operatore punto esclamativo (!) per negare la risposta della funzione **eof()** in maniera tale che quando lui ci dice che non è alla fine noi diciamo al **while** di effettuare un'altra lettura e quando invece ci dice che è arrivato alla fine noi diciamo al **while** di uscire dal ciclo.

Riportiamo di seguito il frammento di codice per la lettura del file e il conteggio del numero di letture:

```
ifstream leggimi;
leggimi.open("Xfile.dat");
double temperatura;
int NumerodiConteggi=0;
while (!leggimi.eof()){
  leggimi>>temperatura;
  if (!leggimi.eof()) {
     cout<< temperatura << endl;
     NumerodiConteggi++;
  }
}
cout<< il numero di conteggi è <<NumerodiConteggi<<endl;
```

[3] eof sta per end of file.

Commentiamo brevemente il codice che abbiamo scritto: prima di tutto abbiamo creato l'operatore di lettura, aperto il file e creato le variabili su cui copiare la singola lettura e su cui effettuare un conteggio (inizializzata a 0); dopo di che il ciclo **while** verifica se la sua condizione che la governa è vera oppure no. Se è vera c'è almeno un dato dentro il file. A questo punto entriamo dentro il ciclo e procediamo a leggerlo procedura per procedura: leggiamo un valore e lo copiamo nella variabile temporanea "temperatura", controlliamo sull'**if** se è vero che abbiamo letto un valore, lo stampiamo a monitor e poi incrementiamo il contatore, ripassiamo la palla al **while** e così via. Quando processo legge l'ultimo dato valido del file continuerà a soddisfare la condizione del **while** entrerà nel ciclo ed effettuerà una lettura che fallirà perché non c'è un altro dato da leggere, la condizione **if** non consentirà di stampare nulla a schermo ne di incrementare il contatore, dopo di che si esce definitivamente dal **while** e si stampa a monitor il valore esatto del numero dei lettura.

Il processo di lettura non interpreta gli spazi vuoti o gli "a capo" e non fa differenza tra file in cui i dati sono scritti in modo ordinato o disordinato, il processo legge elemento ad elemento. Se un file è ordinato e lo vogliamo leggere noi sfruttando questa informazione è nostra responsabilità adottare una strategia capace di utilizzarla al meglio.

Esempio se i dati sono ordinati su due colonne dove sulla prima colonna mettiamo una quantità fisiche indipendente e sulla seconda una quantità dipendente dalla prima basta scrivere un codice in questo modo:

```
ifstream leggimi;
leggimi.open("Xfile.dat");
double temperatura, Energia;
while (!leggimi.eof()){
  leggimi>>temperatura>>Energia;
  if (!leggimi.eof()) {
     cout<< temperatura << "  "<< Energia<< endl;
  }
}
```

Visto che siamo capaci di accedere ai dati e di stamparli a monitor, possiamo avventurarci nello studio degli stessi e provare a calcolare la media e lo scarto quadratico medio utilizzando le seguenti formule

$$< T >= \sum_{i=0}^{N} \frac{t_i}{N}$$

$$\sigma = \sqrt{\sum_{i=0}^{N} \frac{(< T > -t_i)^2}{N}}$$

e con il seguente codice

```
//calcola la somma e la media

ifstream leggimi(nome);   //legge dal file dati.dat
double a;   //variabile di copia provvisoria
double somma =0;  //accumulatore
int NumerodiLetture = 0; //contatore
while(!leggimi.eof()){
  leggimi>>a;   //attribuisce i valori del file alla variabile a,
                //finchè non siamo arrivati alla fine del file
```

```
  somma=somma+a;
  if(!leggimi.eof()){
    cout<<a<<endl;      //stampa i valori del file
    NumerodiLetture++;
  }
}
media=somma/NumerodiLetture;   //calcola la media
cout<<endl;
cout<<"Hai stampato "<<NumerodiLetture<<" numeri."<<endl;
cout<<"La somma dei numeri stampati e' "<<somma<<".\n";
cout<<"La media dei numeri stampati e' "<<media<<" \n";
leggimi.close();  //chiusura del file
```

3.6.2 Caricare dati in un vettore e ricerca dei max e min assoluti

Nella sessione precedente abbiamo imparato a leggere un file con un numero di dati incogniti e a farci dire il numero di letture. Ora scriveremo un codice capace di caricare i dati in un vettore di dimensione pari al numero di letture e di trovare il massimo assoluto e il minimo assoluto tra i valori letti.

Per creare un vettore delle dimensioni giuste, capace di contenere i nostri dati, prima di tutto dobbiamo conoscere quanto è questa dimensione e per avere questa informazione ci sono diverse opzioni: la conosciamo (evviva, fine); apriamo il file e li contiamo (che noia!!!!!); le contiamo con il nostro codice (che forti che siamo!!!). La conoscenza di questo numero è importante perché se imponiamo una dimensione inferiore a quella necessaria il programma di lettura cercherà di scrivere i dati in allocazioni di memoria non riservate al vettore e otterremo un errore nella esecuzione del programma (NOTA BENE - non in fase di compilazione ma in fase di esecuzione) se invece mettiamo prudentemente un numero enorme probabilmente potremmo essere al sicuro da questo problema (ma non è detto) ma violeremmo il principio di richiesta minima di risorse (in questo caso memoria) che è sempre importante tenere presente, anche nell'epoca dei Giga- e dei Tera-Byte.

Quindi prima contiamo e creiamo il vettore e poi...??? A questo punto dobbiamo riavvolgere il nastro. Si esatto l'oggetto che abbiamo usato per leggere è arrivato alla fine del file e quindi non può più accedere ai dati. Bisogna dirgli di ricominciare da capo. Questo lo possiamo fare o con un trucco, cioè chiudiamo e riapriamo il file

```
leggimi.close();    //chiude il file
leggimi.open("dati.dat"); // la riapertura del file mette il "cursore"
                          // dell'ifstream all'inizio del file
```

oppure possiamo usare i metodi offerti dalla classe ifstream come segue

```
leggimi.clear(); //resetta le etichette di errore,
                 //è necessaria prima di una operazione di rewind
leggimi.seekg(0); //rimettiamo il riferimento del sistema
                  //di lettura al primo elemento, realizza il rewind
```

Ora siamo finalmente pronti per caricare i nostri dati in un vettore delle dimensioni giuste.

```
float misure[NumerodiLetture];//crea il vettore che fa per noi!
for(int i=0; i<NumerodiLetture; i++){
   leggimi>>misure[i];
}
leggimi.close(); chiudiamo definitivamente!
```

I dati sono caricati in una variabile ed è a nostra completa disposizione, e quindi possiamo per esempio trovare quale sia il massimo e il minimo assoluto tra i dati che abbiamo letto:

```
//restituisce il valore massimo e il valore minimo

float max, min;
max=misure[0]; //inizializziamo al primo valore
min=misure[0];
for(i=0; i<NumerodiLetture; i++){
  if(misure[i]>max){
    max=misure[i];}
  else if(misure[i]<min){
  min=misure[i];}
}
cout<<"Il valore minimo e': "<<min<<endl;
cout<<"Il valore massimo e': "<<max<<endl;
```

Questo esercizio oltre a contribuire a renderci più familiare la scrittura di un codice, fornisce un pezzo di programma fondamentale per la costruzione di un istogramma.

3.6.3 Costruzione di un istogramma

Prima di iniziare a scrivere un programma per la risoluzione di un problema bisogna conoscere e capire bene il problema. Non avere nessun dubbio su ciò che ci serve e su cosa bisogna fare.

Quindi prima di tutto proviamo a capire che cosa sia un istogramma. Un istogramma è un modo "ordinato" di rappresentare i dati/misure ripetute di una stessa quantità (istogramma monodimensionale) o di più quantità in relazione tra di loro (istogramma multidimensionale) che ci consente di leggere in modo veloce le informazioni statistiche contenute nei dati.

Un modo di costruire un istogramma potrebbe essere quello di individuare il massimo e il minimo assoluto dell'insieme di dati che si vogliono usare per costruire l'istogramma; usare questi due valori per determinare l'intervallo di variabilità dell'istogramma; poi si divide l'intervallo di variabilità in un numero arbitrario di sotto-intervalli (di uguale ampiezza nel nostro esempio, ma esistono anche rappresentazioni con intervalli di ampiezza variabile), i sotto intervalli vengono chiamati in gergo "bin"[4]; infine bisogna contare per ogni sotto intervallo quante misure sono comprese nei rispettivi limiti del sotto intervallo stesso. Se in ogni sotto intervallo registriamo il numero di volte che una misura cade nel sotto intervallo, l'istogramma si dice delle frequenze assolute, se invece questo numero è diviso per il numero del campione allora si dice che l'istogramma è delle frequenze relative.

Di seguito viene riportato il codice capace di assolvere questo compito. Il codice stampa a schermo su una colonna il valore medio di ogni sotto-intervallo e su una seconda colonna la frequenza assoluta delle misure nel sotto-intervallo.

```
//creare istogramma
int rebinning=0;//entra comunque la prima volta
do{ //per ripetere il ciclo, e cambiare binnaggio
 int nint;  //nint=numero intervalli
 cout<<"\nInserisci il numero di intervalli dell'istogramma: \n";
 cin>>nint;
```

[4] bin in inglese vuol dire secchio, parola azzeccata e figurativa in quanto i sotto intervalli di un istogramma si comportano esattamente come un contenitore nel quale uno accumula i dati come quando si mettono oggetti in un contenitore.

```
 float l=(max-min)/(float)nint;//larghezza intervalli
 cout<<"larghezza intervalli: "<<l<<endl;
 int bin[nint]; //crea un vettore di dimensione pari a nint
 for(i=0;i<nint;i++){
bin[i]=0; //inizializza a zero i valori di ogni intervallo
}
 for(i=0;i<nint;i++){
   for(int v=0; v<j; v++){
    if((misure[v]>=min+i*l)&&(misure[v]<=min+(i+1)*l)){
//per ogni intervallo conta quali valori cadono entro l'intervallo stesso
    bin[i]++;
    }
   }
  }

//incrementa il numero di valori per l'intervallo preso in considerazione
//cout << scientific;
 cout << fixed; //risolve il problema dell'allineamento
 cout<<"\nIstogramma:"<<endl;
 for(int n=0; n<nint; n++){
  cout<<min+l/2+n*l<<"\t"<<bin[n]<<endl;
 }
//stampa istogramma e valori medi di ogni intervallino
 cout<<"\n Numero diverso di intervalli? (Se si, premere 1):\n";
 cin>>rebinning;
 }while(rebinning==1);
```

Analizzando un file creato appositamente con un codice MC, una distribuzione gaussiana, con questo codice otteniamo:

```
Inserisci il numero di intervalli dell'istogramma:
15
Larghezza intervalli: 1.522647

Istogramma:
-4.049477 21
-2.526830 154
-1.004183 1038
0.518463 4618
2.041110 13894
3.563756 27787
5.086403 36796
6.609050 31854
8.131697 18555
9.654344 7240
11.176990 1821
12.699636 329
14.222283 40
15.744930 3
17.267576 0
```

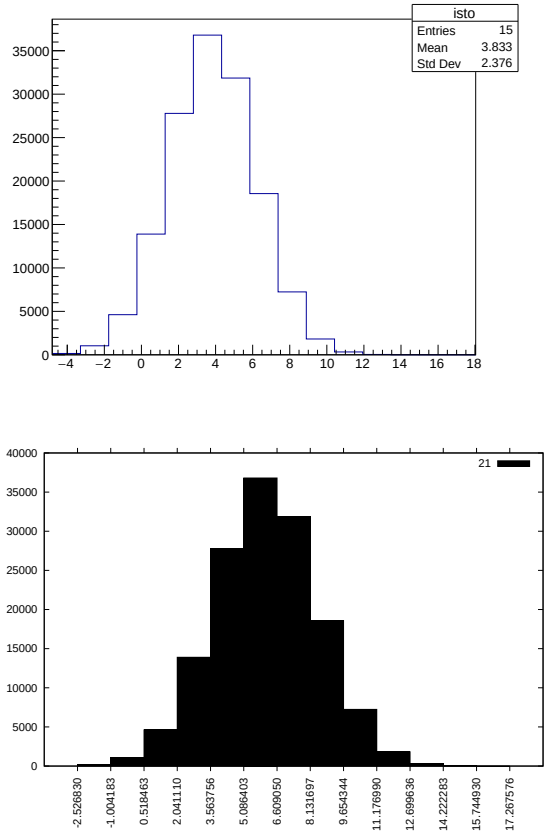


Figura 3.2: Rappresentazioni dell'istogramma computato per 15 divisioni con root in alto e gnuplot in basso.

```
Numero diverso di intervalli? (Se si, premere 1):
0
gmandaglio75@buccellato:>
```

Questa che vediamo è una rappresentazione numerica, non grafica, ma come dicevamo prima la tabella riassume delle informazioni statiche, infatti anche a colpo d'occhio notiamo che la massima frequenza è tra 5.086403 e 6.609050, quindi il valore medio della distribuzione sarà un numero compreso tra questi due valori. Infatti nel calcolo diretto del valore medio effettuato negli esempi precedenti è risultato essere pari a 5.35344.

In figura 3.2 è rappresentato l'istogramma computato con il nostro codice per 15 bin con Root e gnuplot. Più avanti vedremo che le librerie di Root forniscono delle classi che ci consentono di costruire l'istogramma partendo dai dati grezzi. Il programma di grafica gnuplot ha bisogno che il lavoro di calcolo dell'istogramma sia fatto da noi. ROOT non è un programma di grafica ma un tool completo per l'analisi dati che possiede anche la capacità di fare grafica ad alto livello.

3.7 Vettori e Matrici

Sfruttiamo ciò che abbiamo imparato nella sezione precedente per fare un po' di pratica con i vettori e poi successivamente con le matrici. Vettori e Matrici sono un comodo espediente per maneggiare con una sola variabile dati omogenei anche di grosse dimensioni.

3.7.1 Vettore-semplice codice di ordinamento

Sfruttiamo i codici che abbiamo implementato nella sezione precedente per leggere un file incognito, capire quante letture abbiamo fatto e quindi la dimensione da assegnare a un vettore, caricare i dati nel vettore, e poi implementare un semplicissimo algoritmo di ordinamento.

In questo semplice esercizio faremo pratica anche con la scrittura dei dati su file.

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

int main (){
  float pippo;

  ifstream leggimi("ordinami.dat");
  int numerodiletture=0;//contatore
  while(!leggimi.eof()){//gli dico di leggermi il file
        leggimi>>pippo;
        if(!leggimi.eof()){
          numerodiletture++;
        }
  }
  leggimi.close();//lo chiudiamo

  leggimi.open("trovami.dat");
  double vettore[numerodiletture];
  for(int i=0; i<numerodiletture; i++)leggimi>>vettore[i];

  leggimi.close();

  ofstream scrivimi("ordinato.dat");

//*********BLOCCO CODICE CHE ORDINA*****************//
  double min=10000000; //DEVE ESSERE UN REALE
  int indicedelminimo =0;
  double bicchierevuoto;
  for(int partenza=0; partenza<(numerodiletture/3-1); partenza++){
    min=10000000;  //è importantissimo mettere questa condizione
    for(int i=partenza; i<numerodiletture/3; i++){
      if(min>vettore[i]){min=vettore[i]; indicedelminimo=i;}
    }

    bicchierevuoto=vettore[indicedelminimo];
    vettore[indicedelminimo]=vettore[partenza];
    vettore[partenza]=bicchierevuoto;

    scrivimi<<vettore[partenza]<<endl;
  }
```

```
  scrivimi.close();
  return 0;
}
```

L'algoritmo di ordinamento cerca l'indice del vettore corrispondente al minimo assoluto dei valori contenuti nel vettore, scambia questo valore con quello del primo indice, il dato ordinato e ripete la procedura con i dati rimanenti ricorsivamente finché tutti i dati non sono ordinati.

Un esercizio giocoso da proporre a questo punto è chiedere quale sia il numero minimo di digitazioni per modificare il codice in modo che l'ordinamento non avvenga in ordine crescente ma in quello decrescente?
La risposta è 3. Ora però quali sono i tasti non ve lo dico :P .

3.7.2 Matrice - ordinamento

Ora immaginiamo di avere un file che contiene dati ordinati e distribuiti su 3 colonne e una certa quantità di righe. E che questi dati siano stati ordinati con delle proprietà che conosciamo, ad esempio le colonne contengono dati dello stesso tipo solo se appartengono alla stessa colonna mentre se guardiamo i dati rispetto alle righe questi sono tutti di natura differente ma sono accomunati dal fatto che sono stati misurati-creati nello stesso istante. Un esempio potrebbe essere dato da un sistema di rivelazione composto da tre sensori che misurano quantità fisica di diversa natura e un sistema di acquisizione che scrive su un documento le loro misure contemporanee al variare del tempo.

Come possiamo caricare queste informazioni in variabili che possono essere utilizzate da un codice?

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

int main (){
float trovami;
ifstream leggimi("trovami.dat");
int numerodiletture=0;//quanto è lunga la colonna
while(!leggimi.eof()){//gli dico di leggermi il file (trovami)
        leggimi>>trovami;
        if(!leggimi.eof()){
                numerodiletture++;
        }
}
leggimi.close();//lo chiudiamo
leggimi.open("trovami.dat");
//creiamo la matrice [indice di riga][indice di colonna]
double tabellina [numerodiletture/3][3];
for(int i=0; i<numerodiletture/3; i++){//numerodiletture=righe
  for(int j=0; j<3; j++){ //3 colonne
    leggimi>>tabellina[i][j];
  }
}
leggimi.close();
double min=10000000; //DEVE ESSERE UN REALE
```

```
int indicedelminimo =0;

double bicchierevuoto;

for(int partenza=0; partenza<(numerodiletture/3-1); partenza++){
    min=10000000;  //è importantissimo mettere questa condizione
  for(int i=partenza; i<numerodiletture/3; i++){
  if(min>tabellina[i][0]){min=tabellina[i][0]; indicedelminimo=i;}
}

for(int k=0; k<3;k++){
bicchierevuoto=tabellina[indicedelminimo][k];
tabellina[indicedelminimo][k]=tabellina[partenza][k];
tabellina[partenza][k]=bicchierevuoto;
}

cout<<tabellina[partenza][0]<<"  ";
cout<<tabellina[partenza][1]<<"  ";
cout<<tabellina[partenza][2]<<endl;
}
return 0;
}
```

Il codice che abbiamo appena scritto legge i dati da un file in cui ci sono tre colonne di dati e ordina i dati rispetto alla prima colonna con ordinamento crescente. Se volessimo ordinare rispetto alla seconda colonna basterebbe cambiare il secondo indice della matrice nella parte di codice di ricerca degli indici rispetto ai quali fare lo scambio:

```
for(int partenza=0; partenza<(numerodiletture/3-1); partenza++){
    min=10000000;  //è importantissimo mettere questa condizione
  for(int i=partenza; i<numerodiletture/3; i++){
  //cambiamo colonna
  if(min>tabellina[i][1]){min=tabellina[i][1]; indicedelminimo=i;}
}
```

così come abbiamo appena fatto.

Allocazione dinamica della memoria - malloc e new

Sfruttiamo l'esempio precedente per spiegare l'allocazione dinamica della memoria. Un variabile array non è altro che una variabile di tipo puntatore, ovvero una variabile capace di memorizzare un indirizzo di memoria relativo ad uno spazio di memoria consecutivo capace di memorizzare un numero di dati, tutti dello stesso tipo, pari alle dimensioni dell'array. Un array può essere utilizzato con la comoda notazione con le parentesi quadre che racchiudono gli indici oppure con l'algebra dei puntatori,

```
//le due scritture sono equivalenti
cout<<vettor[3]<<endl;
cout<<*(vettore+2)<<endl;//+2 perchè si conta da zero
```

E' bene ricordare che se si definisce un array non attraverso l'uso diretto dei puntatori, gli indirizzi che vengono memorizzati nelle variabili puntatori sono statiche, fisse e non possono essere modificate.

L'accesso alla memoria è una possibile sorgente di errori ma è anche una possibilità molto potente. Proviamo a dimostrarlo con un semplice esempio utilizzando il codice che abbiamo appena implementato per l'ordinamento.

Il seguente frammento di codice, una volta individuati gli indici per lo scambio, esegue ordinatamente lo scambio colonna per colonna:

```
for(int k=0; k<3;k++){
bicchierevuoto=tabellina[indicedelminimo][k];
tabellina[indicedelminimo][k]=tabellina[partenza][k];
tabellina[partenza][k]=bicchierevuoto;
}

cout<<tabellina[partenza][0]<<"  ";
cout<<tabellina[partenza][1]<<"  ";
cout<<tabellina[partenza][2]<<endl;
}
```

e se le colonne del nostro documento fossero 10 al posto di 3, o 100, o 1000 etc? Mi direte, che problema c'è, cambiamo la condizione di stop del ciclo **for** e il gioco è fatto :)...e io mi seccherò come una biscia! In realtà la risposta è corretta, ma costringeremo il pc a eseguire 3 operazione per enne numero delle colonne. Grazie all'uso dei puntatori e della costruzione di una tabella (matrice) con l'allocazione dinamica della memoria potremo agilmente eseguire l'ordinamento effettuando solo le classiche tre operazioni di scambio. Lo scambio avverrà tra gli indirizzi di memoria che puntano alle locazioni di memoria in cui sono iscritti i dati di una intera riga (lunga quanto gli pare..).

Un matrice è costruibile con i puntatori utilizzando un puntatore doppio ($double**puntatoredoppio$;). Un puntatore doppio è una variabile che contiene un indirizzo di memoria che si riferisce ad una zona di memoria riservata alla memorizzazione di indirizzi. Se in questa zona di memoria si memorizzano indirizzi che puntano a zone di memoria dove si possono memorizzare numeri, il gioco è fatto abbiamo creato una variabile capace di gestire una matrice.

Ora vederemo le implementazioni di una matrice con l'allocazione dinamica della memoria fatta con la funzione **malloc** di C o con l'operatore new (l'algebra dell'operatore new è mostrata nell'esempio che segue) di C++, ovviamente ciò che vale in C vale anche in C++ il contrario non è generalmente vero.

Implemantazione C-Style:

```
//libreria necessaria per usare malloc
#include <stdlib.h>
...
....

//ha bisogno della libreria stdlib.h compatibile con C
double** tabellina=  (double**)malloc((numerodiletture/3)*sizeof(double*));
for(int i = 0; i < numerodiletture/3; ++i)
    tabellina[i] = (double*)malloc(3*sizeof(double));
```

Implemantazione C++-Style:

```
//libreria necessaria per usare malloc
#include <stdlib.h>
```

```
...
....
double** tabellina = new double*[numerodiletture/3];
for(int i = 0; i < numerodiletture/3; ++i)
    tabellina[i] = new double[3];
//scrittura equivalente, forse meno comoda
//  *(tabellina+i)= new double[3];
```

A questo punto abbiamo la nostra matrice dinamica, e abbiamo accesso alle variabili indirizzo che la governano. Per effettuare l'ordinamento basterà scrivere:

```
//lavoriamo direttamente con gli indirizzi!!!
bicchierevuoto= tabellina[indicedelminimo];
tabellina[indicedelminimo]=tabellina[partenza];
tabellina[partenza]=bicchierevuoto;
```

In un certo senso abbiamo reso una struttura bidimensionale in una monodimensionale.

Ricerca binaria

Una conseguenza dell'ordinamento di una sequenza di dati è la possibilità di fare delle ricerche degli indici della matrice in cui dati valori sono memorizzati. L'algoritmo più veloce è l'algoritmo di ricerca binaria o dicotomica.

Nell'esempio che segue presentiamo il frammento di codice che consente di trovare l'indice in cui è memorizzato il valore più prossimo a quello desiderato:

```
double valore=2.35;
int indice_alto=numerodiletture/3 -1; //indice più alto
int indice_basso=0;          //indice più basso
int cont =0; //contatore: contiamo il numero di prove
int indicemedio=(indice_alto-indice_basso)/2;
//ciclo che realizza la ricerca binaria
while(fabs(tabellina[indicemedio][0]-valore)>0.5){
  cont++;
  if (tabellina[indicemedio][0] > valore){
    indice_alto=indicemedio;
  }
  else {
    indice_basso=indicemedio;
  }
indicemedio=(indice_alto-indice_basso)/2;
}
```

3.8 Funzioni

In c/c++ qualunque programma e sotto-programma sono funzioni. Anche il programma principale è una funzione e si chiama **main** (in inglese vuole dire principale). Tutte le funzioni, compresa la principale, hanno lo stesso paradigma: cosa restituisce - nome della funzione - tra parentesi tonde ciò che riceve in input - l'implementazione della funzione segue racchiusa tra graffe.

Il codice che segue è la semplice implementazione di una funzione che restituisce la somma di due numeri interi forniti alla funzione.

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

Ora però vediamo come le funzioni vengono implementati all'interno di un programma, come vengono chiamate e che regole devono essere eseguite. Una regola che non ammette eccezioni è che ogni funzione, come del resto le variabili, deve essere dichiarata prima dell'esecuzione della funzione main. La dichiarazione della funzione si chiama prototipo. Il prototipo ripeto deve essere sempre dichiarato, la dichiarazione può coincidere con l'implementazione oppure no. Le variabili che vengono usate per l'implementazione delle funzioni sono puramente simboliche, e in generale nella funzione principale non siamo tenuti a usare le stesse variabili. Le variabili dichiarate all'interno di una funzione sono variabili locali.

Esempio prototipo e implementazione prima della funzione principale:

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

int somma(int a, int b){
return a+b;
}
int main(){
int pino,pina;
pino =10;
pina=20;
cout<<somma(pino,pina)<<endl;
return 1;
}
```

Esempio prototipo prima della funzione principale e implementazione dopo la funzione principale:

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

int somma(int a, int b);

int main(){
int pino,pina;
pino =10;
pina=20;
cout<<somma(pino,pina)<<endl;
return 1;
}
int somma(int a, int b){
return a+b;
}
```

Esempio in cui la funzione accessoria somma è stata scritta in un file "header" somma.h:

```
#include <iostream>
#include "somma.h"
using namespace std;
```

```
int main(){
int pino,pina;
pino =10;
pina=20;
cout<<somma(pino,pina)<<endl;
return 1;
}
```

L'header file somma.h che in questo esempio vive nella stessa cartella che contiene il programma che lo carica attraverso la chiamata #include contiene prototipo e implementazione della funzione:

```
// file somma.h
int somma(int a, int b){
return a+b;
}
```

3.8.1 Passaggio di valori a funzioni via value o reference

Esercizi

- usare una funzione che in input vuole una vettore di caratteri o una stringa che contenga il nome/o percorso+ nome di un file e in uscita fornisca il numero di valori scritti nel file.
- Scrivere una funzione che in input vuole una vettore di caratteri o una stringa che contenga il nome/o percorso+ nome di un file e in uscita contiene un vettore che contiene tutti i valori letti nel file.
- Scrivere una funzione che in input prenda un vettore di reali e che restituisca il massimo e il minimo assoluti contenuti dentro il vettore. Come possiamo avere due uscite invece che una?
- Scrivere una funzione che metta in evidenza l'utilità di utilizzare il passaggio di parametri per valore.

3.8.2 Ricorsività delle funzioni

Le funzioni hanno la proprietà di poter chiamare se stesse, questa proprietà viene detta ricorsività. Questa funzione è molto utile e può essere utilizzata per esempio quando si implementano dei codici di ricerca dicotomica (detta anche binaria), o per esempio se volessimo implementare l'operazione di fattoriale.

```
// calcolo del fattoriale
#include <iostream>
using namespace std;

long fattoriale (long valore)
{
  if (valore > 1){
   return (numero * fattoriale (valore-1));
  }
  else{
   return 1;
  }
}

int main ()
{
```

```
  long Numero = 9;
  cout << Numero << "! = " << fattoriale (Numero);
  return 0;
}
```

3.8.3 Template di funzione

Header file della funzione template del minimo:

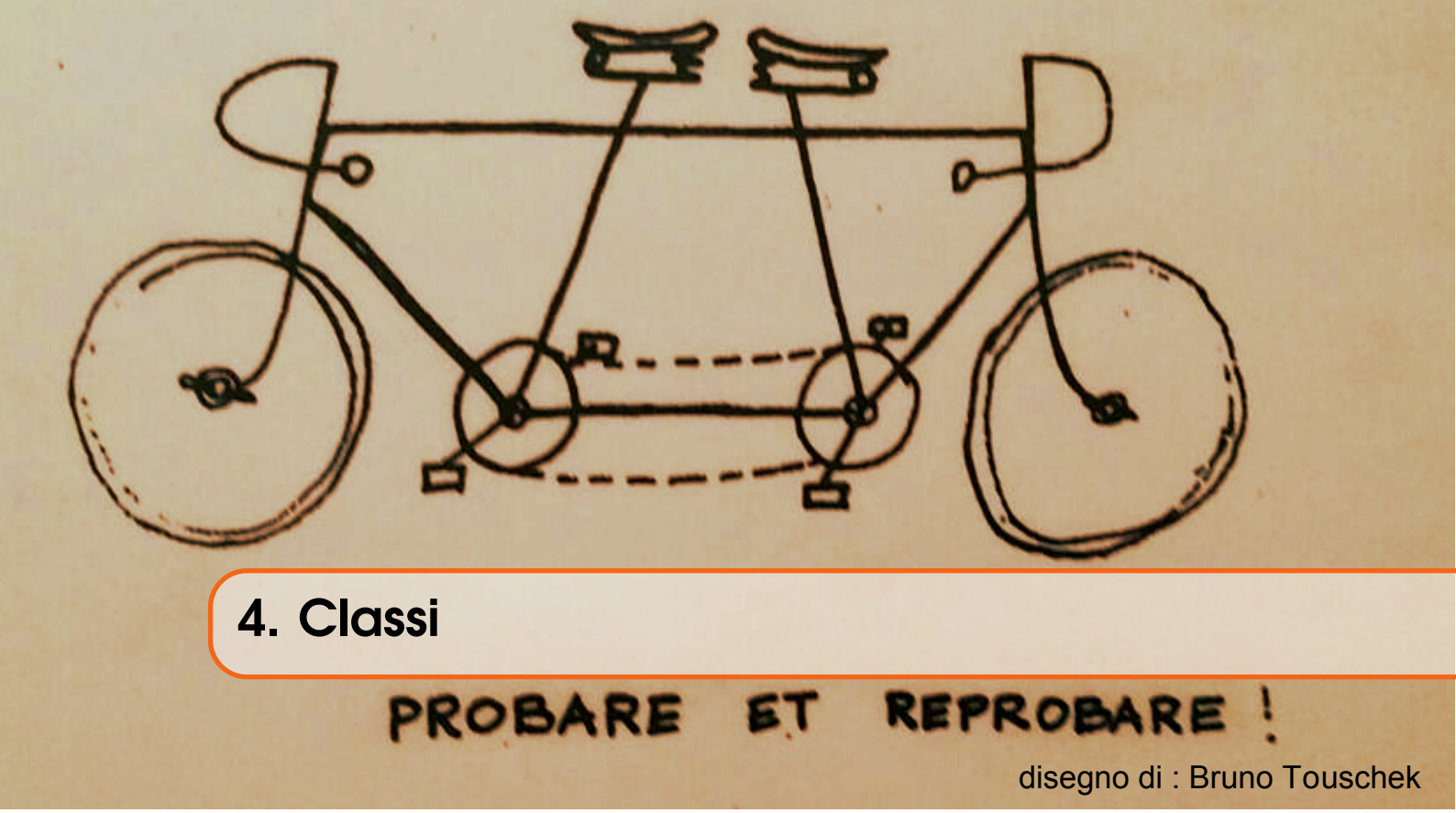
```
//file header min.h
template <class Type>

Type minimo(Type a, Type b){
return (a<b)?a:b;
}
```

Header file della funzione template del massimo:

```
//file header max.h
template <class Type>

Type massimo(Type a, Type b){
return (a<b)?a:b;
}
```

Codice che fa uso delle due funzioni template:

```
#include <iostream>
#include "max.h"
#include "min.h"

using namespace std;

int main (){

double a=1.2; double b=3.0;

double Max, Min;

Max=massimo(a,b);
Min=minimo(a,b);

cout<<"Max= "<<Max<<endl;
cout<<"Min= "<<Min<<endl;

return 0;}
```



disegno di : Bruno Touschek

4. Classi

Le classi rappresentano ciò che rende speciale e differente dal C il C++, ovvero costruire programmi con architettura orientata agli oggetti. La classe la possiamo vedere come una sorta di nuovo dato, speciale, definito dall'utente, il quale è capace di incamerare informazioni, fornire informazioni, elaborare le informazioni. La classe contiene quindi dati di vario tipo e funzioni (in generale vengono detti metodi della classe). La programmazione orientata agli oggetti è ideale per progetti di grosse dimensioni, in cui una nuova classe viene implementata per risolvere nel modo più generale possibile una serie di problematiche e viene messa a disposizione del progetto. Dal momento che la nuova classe è caricata in una libreria e viene messa a disposizione, entra a far parte delle utilities che possono essere utilizzate dall'utente del programma generale. Sulla programmazione orientata agli oggetti, sulle classi, sull'architettura del software e sulla sua ottimizzazione in progetti di grosse dimensioni esiste una letteratura infinita. Lo scopo del corso è quello di apprendere i rudimenti dell'implementazione e dell'uso delle classi.

4.1 Le classi e i tipi di dati

Prima di tutto scriviamo la struttura che ha una classe quando viene implementata:

```
//Dichiarazione della classe
class nomedellaclasse{
//Specificatore del tipo di dato
Public:
istruzione..;
istruzione...;
istruzione...;
Private:
istruzione ...;
istruzione....;
Protected:
istruzione..;
```

```
istruzione...;
};
```

La dichiarazione di una classe inizia con la parola chiave *class* seguita dal nome della classe e dalla sua implementazione racchiusa tra graffe. La classe può contenere abbiamo detto dati e funzioni, e questi possono avere differente autorizzazioni di accesso. Per default i dati di una classe se non specificato sono di tipo **Private**. I dati di tipo **Private** sono accessibili solo a membri della classe o a loro amiche (friend), è una forma di protezione dei dati, perchè siano accessibili anche non da membri della classe bisogna dichiararli di tipo **public**. Il modo di dichiarare il tipo di dato si esegue attraverso le parole chiave private, public, protected seguito dai due punti, questi operatori sono detti modificatori del tipo di accesso.

4.2 La mia prima Classe

La scrittura di una classe dovrebbe essere guidata dall'idea che questa sia utile a risolvere in modo quanto più generale possibile, in maniera da evitare il proliferare di classi che svolgono funzioni simili e ripetizioni inutili di codici. Anche nella scrittura dei codici bisogna avere uno spirito rivolto al risparmio delle risorse e all'efficienza.

Magari non sarà l'esempio migliore, ma proviamo a scrivere una classe che richiedendo due informazioni sia capace di restituire un certo numero di informazioni. L'esempio che vogliamo scrivere è una classe che chiameremo PoligonoRegolare che richiede come dati la base e l'altezza e che sia capace di restituire nel caso in cui fosse interrogato il perimetro e l'area di un triangolo, un quadrato, o un rettangolo, o un pentagono, intendendo il classico significato di base e altezza per il triangolo e il rettangolo, il lato del quadrato pari alla base, l'apotema del pentagono pari alla variabile altezza.

```
class PoligonoRegolare {
  double base;  //per default base e altezza così sono private
  double altezza;
  public:
  //prototipi di funzioni
  void dammi_la_base_e_altezza(double,double);
  double areatriangolo ();
  //(implementazione in fase di dichiarazione)
  double perimetrotriangolo(){
        return 3*base;
  }
  double arearettangolo(){
        return base*altezza;
  }
  double perimetrorettangolo(){
        return 2*(base+altezza);
  }
  double areaquadrato ();
  double perimetroquadrato ();
  double areapentagono ();
  double perimetropentagono();
};

//implementazione esterna di alcuni metodi della classe
```

```
//PoligonoRegolare
void PoligonoRegolare::DammiBaseeAltezza(double a, double b){
  base=a;
  altezza=b;
  return;
  }
double PoligonoRegolare::areatriangolo(){
        return base*altezza/2;
}
double PoligonoRegolare::perimetroquadrato(){
        return base*4;
}
double PoligonoRegolare::areaquadrato(){
        return base*base;
}
double PoligonoRegolare::perimetropentagono(){
        return base*5;
}
double PoligonoRegolare::areapentagono(){
        return (base*altezza/2)*5;
}
```

Nella classe appena mostrata noterete che vengono mostrati i metodi per implementare i metodi della classe sia internamente alla definizione della stessa, che esternamente collegando il nome del metodo al nome della classe con l'operatore due punti doppi. Questo non è per invitarvi all'anarchia, solo per farvi vedere che è consentito farlo nei due modi, generalmente quando si scrivono i programmi si tende a mettere la definizione della classe in un file header con estensione .h e l'implementazione dei metodi dentro un altro file con estensione .cpp.

Un'altra cosa degna di nota è che i dati base e altezza non mostrano nessuna dichiarazione di modo di accesso quindi per default sono di tipo private. Importante come per le parentesi quadre, è bene conoscere questa regola ma non abusarne e dichiarare sempre il tipo di accesso riservato alle variabili e ai metodi. Le parole chiavi per definire il tipo di accesso seguite dai due punti caratterizzano tutto ciò che li segue fino a che non interviene un'altra parola chiave che definisce un nuovo modo di accesso. Nell'esempio che segue le variabili A,B,D sono pubbliche la variabile C è privata:

```
class pippo{
public:
double A,B;
private:
int C;
public:
char E;
};
```

4.3 Come usare una classe

In questo paragrafo ci riproponiamo di far vedere come si utilizza la classe "poligono" che abbiamo appena implementato:

```
 int main (){
 //pippo è un oggetto della classe poligono
poligono pippo;

double a, b;
cout<<"Inserire base e altezza: "<<endl;
cin>>a>>b;
pippo.DammiBaseeAltezza(a,b);
//dato che è private non posso stamparela
//a meno che nella classe non la dichiariamo pubblica
//cout<<pippo.altezza<<endl;
//ci stampa area del triangolo
cout<<"Area del triangolo: "<<pippo.areatriangolo()<<endl;
//il perimetro
cout<<"Perimetro del triangolo: "<<pippo.perimetrotriangolo()<<endl;
//etc
cout<<"Area del rettangolo: "<<pippo.arearettangolo()<<endl;
cout<<"Perimetro del rettangolo: "<<pippo.perimetrorettangolo()<<endl;

//se pippo fosse un puntatore poligono
//pippo come variabile puntatore
//poligono maria;
//poligono *pippo;
//pippo = &maria;
//l'unica cosa che cambia nell'utilizzo della classe è
//che l'operatore punto viene sostituito dall'operatore freccetta (->))

return 0;
}
```

Implementata una classe, per utilizzarla basta fare una semplice dichiarazione formalmente identica a quando dichiariamo una variabile qualunque del nostro codice, con la differenza che la variabile che dichiariamo è del tipo della classe. Questa operazione, inconsapevoli, l'abbiamo già fatta durante il corso quando abbiamo implementato i codici che richiedevano la lettura o la scrittura da e su file utilizzando le classi **ifstream** e **ofstream**. La variabile leggimi o scrivimi di turno era a tutti gli effetti una variabile di tipo **ifstream** o **ofstream**. Queste variabili vengono detti oggetti.

Un oggetto di una classe funziona come un programma completo che è stato pensato e implementato per risolvere un problema nel modo più generale possibile. Un oggetto, come qualunque programma, può avere bisogno di informazioni, necessita di strumenti per ricevere le informazioni, elaborare informazioni quando gli viene chiesto, strumenti per fornire informazioni. Le funzioni che operano quanto descritto vengono chiamati metodi. Ai metodi si accede attraverso l'operatore punto che collega il nome dell'oggetto e il nome del metodo, oppure l'operatore freccetta (si compone del simbolo meno + il simbolo maggiore $->$) nel caso in cui ci riferiamo all'oggetto attraverso una variabile puntatore.

4.4 Costruttori e distruttori

Nell'esempio precedente abbiamo implementato una classe poligono che per funzionare ha bisogno di due informazioni di tipo reale che vengono fornite attraverso un metodo. Ma cosa succede se

non utilizziamo il metodo di trasmissione delle informazioni richieste e proviamo a richiedere lo stesso informazioni all'oggetto? Quello che succede è che l'oggetto non ci fornirà nessuna informazione non gliene abbiamo fornito nessuna. Un sistema per fare in modo che l'oggetto diventi immediatamente operativo all'atto della sua creazione è quello di implementare un costruttore. Il costruttore è una funzione particolare della classe. Questa funzione ha lo stesso nome della classe può avere argomenti in ingresso ma non restituisce nessun valore. Nell'esempio che segue vediamo suoi possibili prototipi nell'esempio della classe PoligonoRegolare:

```
class PoligonoRegolare {
  double base;  //per default base e altezza così sono private
  double altezza;
  public:
   PoligonoRegolare();
   PoligonoRegolare(double,double);

  .....
  .....
  }; //notare il punto e virgola a chiusura della def.
```

L'implementazione del costruttore può avvenire inline oppure esternamente come visto per gli altri metodi:
Modo inline

```
class PoligonoRegolare {
  double base;  //per default base e altezza così sono private
  double altezza;
  public:
   PoligonoRegolare(){};
   PoligonoRegolare(double a,double b){
    base = a;
    altezza = b;
   };
  .....
  .....
  };
```

Modo esterno:

```
class PoligonoRegolare {
  double base;  //per default base e altezza così sono private
  double altezza;
  public:
   PoligonoRegolare();
   PoligonoRegolare(double,double);
  .....
  .....
  };

    PoligonoRegolare::PoligonoRegolare(){
```

```
 base=0.;
 altezza=0.;};
 PoligonoRegolare::PoligonoRegolare(double a,double b){
 base = a;
 altezza = b;
};
```

Ovviamente nella implementazione esterna bisogna dire al compilatore a chi appartiene il metodo e quindi va specificata la classe e poi a seguire i doppi due punti e infine il metodo. Un'altra cosa importante da notare è che nel nostro esempio abbiamo implementato una doppia versione del costruttore. La cosa non è vietata, anzi avremmo potuto implementarne molte di più. Questa caratteristica l'abbiamo già vista quando abbiamo studiato le funzioni, e abbiamo detto che si chiama overload delle variabili di utilizzo delle funzioni. L'overload, definizione multipla del costruttore si estende anche agli altri metodi. Se i metodi si differenziano nei dati di input che ricevono, il compilatore è capace di distinguere quale versione del metodo deve utilizzare.

Mostriamo ora come viene invocato il costruttore nell'esempio:

```
PoligonoRegolare pippo;
```

in questo caso stiamo usando il costruttore senza parametri e quindi i dati della classe sono inizializzati a delle costanti scelte da chi ha concepito la classe;

```
PoligonoRegolare pippo(3.2,1.5);
//oppure attraverso variabili
//PoligonoRegolare pippo(bas,alte);
```

in questo caso invece l'oggetto pippo viene dichiarato passandogli dei parametri, questa scrittura dice al compilatore che deve utilizzare il secondo costruttore e in questo caso i dati passati all'oggetto sono decisi in fase di dichiarazione dall'utilizzatore della classe (user).

Distruttori

Il distruttore è una funzione membro speciale della classe, la sua dichiarazione è fatta dal nome della classe preceduto dal simbolo tilde $\sim$. Esso viene invocato implicitamente quando l'oggetto di una classe viene distrutto e ciò accade quando l'esecuzione del programma oltrepassa l'ambito di visibilità della classe. Il distruttore non restituisce immediatamente la memoria occupata dall'oggetto al sistema, esso agisce sulla memoria in modo tale che tale lo spazio occupato possa essere riutilizzato per memorizzare nuovi oggetti.

Il distruttore, ripeto, si definisce all'interno della classe come un metodo che ha lo stesso nome della classe, come il costruttore, ma preceduto da una tilde. Il distruttore se non definito, viene implicitamente creato. Il distruttore a differenza del costruttore può essere definito una sola volta, non ha argomenti e non restituisce nulla, neanche void.

```
class PoligonoRegolare {
  double base;  //per default base e altezza così sono private
  double altezza;
  public:
   PoligonoRegolare();
   PoligonoRegolare(double,double);
   ~PoligonoRegolare();
  .....
  .....
```

```
};
//implementazione del distruttore
PoligonoRegolare::~ PoligonoRegolare(){
cout<<"Salve Carissimo!!!!"<<endl;
cout<<"ti ho appena distrutto un oggetto di tipo PoligonoRegolare"<<endl;
cout<<"caramente, il Distruttore"<<endl;
}
```

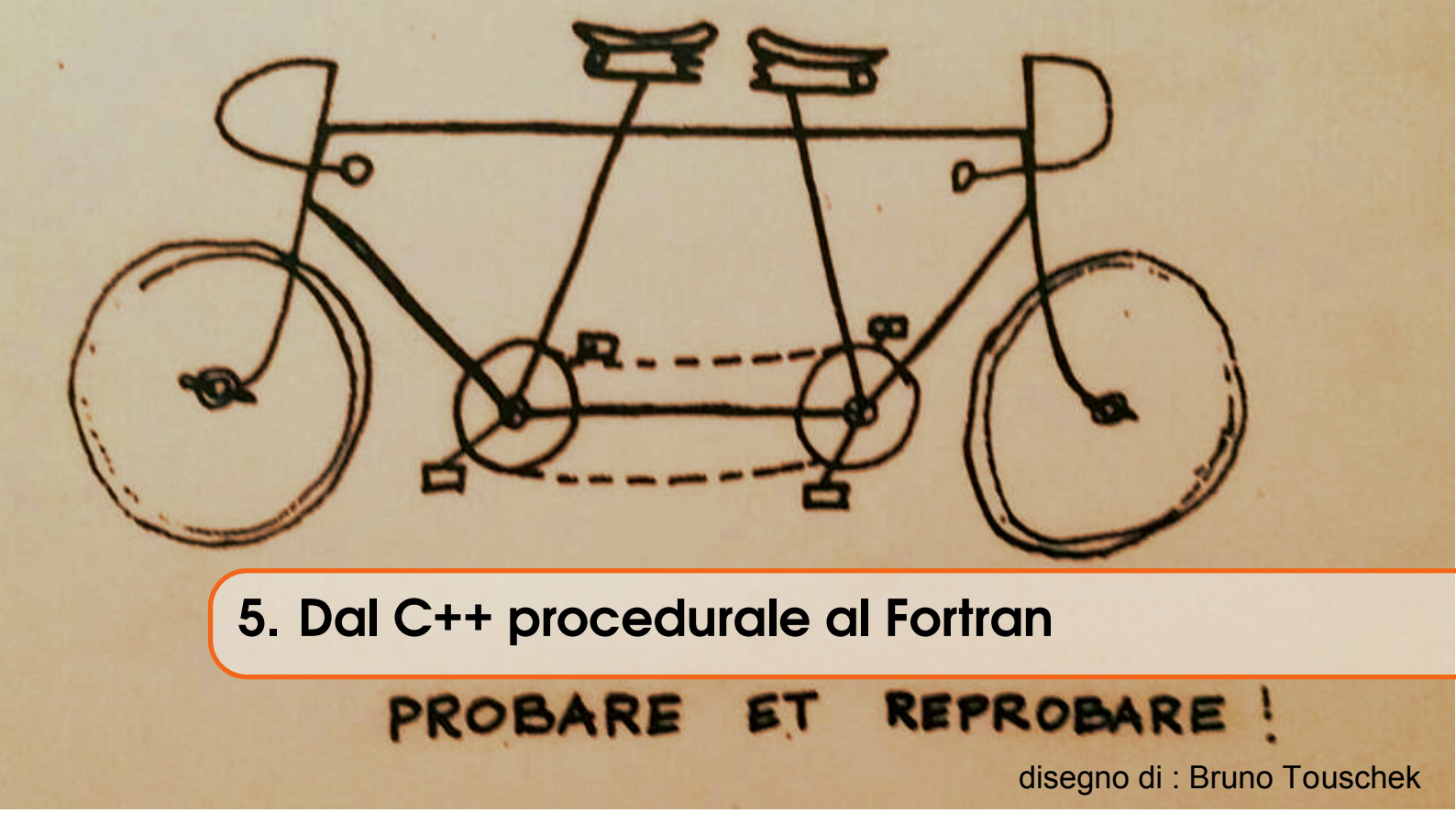
Per invocarlo all'interno di un codice basta scrivere nomeoggetto.∼nomedellaclasse() se nomeoggetto è una variabile del tipo nomedellaclasse, nomeoggetto− >∼nomedellaclasse() se nomeoggetto è un puntatore alla classe nomedellaclasse.

Il distruttore viene invocato automaticamente ogni volta che il programma termina, o la funzione che utilizza una data classe termina la sua vita.

4.5 Ereditarietà

```
#include <iostream>

using namespace std;

class forma{
protected:
double base, altezza;
public:
forma();
forma(double, double); //costruttore
//~forma(); //distruttore
void dammi_base_e_altezza(double a, double b){
base=a; altezza=b;
return;}
double dammialtezza(){
return altezza;}

};

//implementazione costruttore
forma::forma(){
cout<<"ho appena creato un oggetto di tipo forma"<<endl;
}
forma::forma(double p, double m){
base=p;
altezza=m;
}

//implementazione distruttore
forma::~forma(){
cout<<"\ndistrutto"<<endl;
}
```

```
//creazione classe derivata
class rettangolo:public forma{
public:
rettangolo(double,double);   //costruttore se si vuole
~rettangolo();       //distruttore se si vuole
double area(){
return base*altezza;}
};

//implementazione costruttore
rettangolo::rettangolo(double a, double b){
base=a;
altezza=b;
};

//implementazione distruttore
rettangolo::~rettangolo(){
cout<<"distrutto"<<endl;
}

int main(){
double a, b;
cout<<"Inserire base e altezza: "<<endl;
cin>>a>>b;
forma pippo(a,b);
cout<<"Altezza per prova: "<<pippo.dammialtezza()<<endl;
rettangolo pina(a,b);
cout<<"Area: "<<pina.area()<<endl;
return 0;
}
```



disegno di : Bruno Touschek

5. Dal C++ procedurale al Fortran

Il linguaggio di programmazione Fortran, acronimo di Formula Translation, è stato creato per la soluzione di problemi computazionali e grazie alla sua somiglianza tra la sua implementazione e la scrittura formale del linguaggio aritmetico semplifica molto la preparazione dei problemi da sottoporre alla macchina. Qualunque problema implementabile in fortran può essere trascritto molto facilmente in C++, mentre il contrario non è sempre vero. A questo punto ci potremmo chiedere per quale motivo vale la pena imparare i rudimenti di questo linguaggio. Il primo motivo che mi viene in mente è che richiede pochissimo sforzo, dopo avere fatto un robusto training di programmazione procedurale in C++ (o C), il secondo motivo è che la conoscenza di questo linguaggio vi consentirà di avere accesso al patrimonio infinito di codici scritti in fortran dalla comunità dei Fisici teorici, fenomenologici e anche sperimentali negli ultimi 70 anni circa.

Lo stesso potentissimo framework per l'analisi dati, ROOT, discende dai preziosissimi codici in fortran in cui erano scritte le cernlib e PAW.

5.1 Struttura di un programma in Fortran

Un documento di testo su un computer è come un foglio di carta a quadretti dove ogni quadretto può ospitare un carattere. Con questa banalità in mente, possiamo immaginare il foglio come una griglia composta da colonne e righe. Il compilatore fortran fa uso solamente delle prime 72 colonne: le colonne 1-5 sono riservate alle etichette (label, il valore massimo di una etichetta è ovviamente 99999), la colonna 6 per i simboli di accapo (questo consente di avere più spazio per scrivere una singola istruzione), dalla 7 all 72 per scrivere una istruzione. Scrivendo il carattere * o C alla colonna 1 si può adoperare l'intera riga per inserire dei commenti. Questa struttura rigida della griglia su cui scrivere il proprio programma deriva dalle schede forate su cui originariamente venivano implementati questi programmi e che poi venivano lette - compilate e eseguite dal calcolatore.

Di seguito è riportato un semplice esempio di programma scritto in fortran che svolge l'operazione di somma tra due numeri:

```
colonne
c23456789......
```

```
CCC FILE Esempio.f
      PROGRAM ESEMPIO
      Real a,b
      print*,'dammi i primo numero'
      read(*,*)a
      Print*,'dammi il secondo numero'
      read(*,*)b
      write(*,100) 'il risultato della somma a + b = ',a+b
100   Format(A302,2x,F6.2)
      Return
      END
```

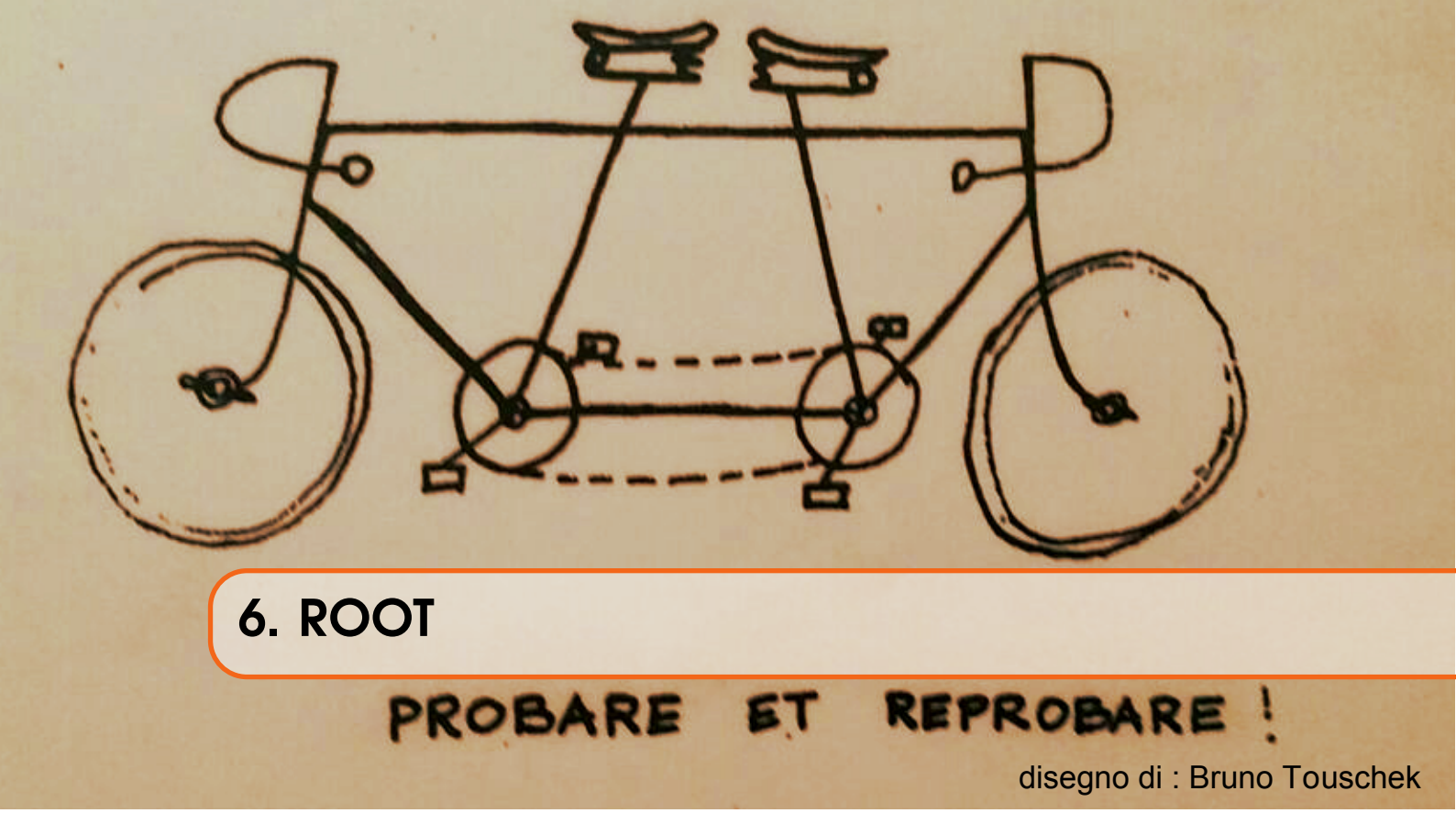
Ritroviamo nel codice quanto affermato precedentemente: le istruzioni vivono dalla colonna 6 in poi, le righe che iniziano per c sono commenti ignorati dal compilatore, nella zona di colonne 1-5 troviamo una etichetta prima dell'istruzione format. Il codice è scritto in maniera alquanto disordinata, c'è il comando **print** scritto una volta con l'iniziale maiuscola e una volta minuscola, la maleducazione informatica serve solo a dirvi che il fortran a differenza del C++ è "case-insensitive" cioè non distingue tra maiuscole e minuscole e non ovviamente a incoraggiarvi a scritture disordinate e poco leggibili.

5.2 Codice "istogrammatore" in Fortran

```
CCCCC file = istogrammatore.f
      PROGRAM ISTOGRAMMATORE
      integer i,j,num_dati,num_intervalli,bin(1000000)
      real a,media,somma,somm,sqm,misure(1000000)
      real massimo,minimo,larghezza
      character*80 nome
      integer continuo
      print*,'mi dici il nome del file dei dati'
      read(*,*) nome
      OPEN(10,FILE= nome,STATUS='old')
      print*,'mi dici il numero di dati che devo leggere'
      read(*,*)num_dati
c      real misure(num_dati)
      do  i=1,num_dati
        read(10,*)misure(i)
c        print*,misure(i)
      enddo
      somma = 0.
      do i=1,num_dati
        somma = somma + misure(i)
      enddo
      media = somma/real(num_dati)
      sqm=0.
      do i=1,num_dati
        sqm = sqm + (media - misure(i))**2.
      enddo
      sqm= sqm/real(num_dati)
      write(*,*)'media = ',media
```

```
      write(*,*)'deviazione standard =',sqm
c ricerca del massimo e del minimo assoluto
      minimo  = misure(1)
      massimo = misure(1)
      do i=1,num_dati
        if(misure(i).gt.massimo)then
            massimo=misure(i)
        endif
        if(misure(i).lt.minimo)then
            minimo=misure(i)
        endif
      enddo
      write(*,*)'minimo  = ',minimo
      write(*,*)'massimo = ',massimo
C codice istogrammatore
      continuo=1
      do while(continuo.eq.1)
        print*,'mi dici il numero di intervalli?'
        read(*,*)num_intervalli
        do i=1,num_intervalli
            bin(i)=0
        enddo
        larghezza = (massimo-minimo)/real(num_intervalli)
        do i=1,num_intervalli
            do j=1,num_dati
                if((misure(j).ge.(minimo+real(i)*larghezza)).and.
     &              (misure(j).le.(minimo+real(i+1)*larghezza)))then
                    bin(i)=bin(i)+1
                endif
            enddo
        enddo
        write(*,*)'centro del bin     frequenza'
        do i=1,num_intervalli
        print*, minimo+larghezza*(0.5+real(i-1)),'  ',bin(i)
        enddo
        print*,'vuoi fare un altro giro digita 1'
        read(*,*)continuo
      enddo
      return
      end
```

III Strumenti di Analisi Dati



disegno di : Bruno Touschek

6. ROOT

Root è un framework per l'analisi dati, ideato, sviluppato e distribuito dal CERN. Esso mette a disposizione una immensa quantità di librerie (aperte, accesso completo ai sorgenti) scritte in C++ utili per analizzare dati, scrivere programmi di simulazione, generare dati, memorizzare e gestire grosse moli da dati, affiancare sistemi di acquisizione per l'analisi online etc. etc. etc.

Dopo aver finito di scrivere una pagina di etc, è bene evidenziare che un'altro dei vantaggi nell'utilizzare questo framework di lavoro consiste nel fatto che utilizza una linguaggio pubblico (quello maggiormente insegnato nelle facoltà scientifiche delle Università in tutto il mondo), potente, utilizzato largamente per scrivere i sistemi operativi o linguaggi a più alto livello e potentissimi quali il python. Si si, mi riferivo al C++, quello originale, niente varianti, dialetti o personalizzazioni, C++ e basta. Ci sono molti altri software liberi per l'analisi dati molto potenti e che consiglio vivamente di studiare e provare a utilizzare quali scilab, octave, etc, ma questi hanno un linguaggio dedicato che bisogna apprendere. Ricordiamoci che non è la conoscenza del linguaggio a farvi diventare dei programmatori, ma sicuramente una volta che programmatori lo siete già diventati, inesperti alle prime armi non ha importanza, conoscere il C++ e avere accesso immediato ad una risorsa potente come Root è un vantaggio da non sottovalutare.

Conoscere il C++ concede pieno e completo accesso a Root. Root non è una macchinetta graziosa e con i bottoncini colorati sui quali cliccare quello che ci serve. Root mette a disposizione una quantità infinita di strumenti, ma il lavoro lo dobbiamo fare noi. Bisogna scrivere codici! Semplici per operazioni semplici e via via più complicati a seconda della complessità del problema.

Root può essere utilizzato in diversi modi: fornisce un terminale di comando interattivo (come perl, python, gnuplot, etc), consente di interpretare degli script o di caricare ed eseguire delle funzioni (codici scritti in un file con estensione .C), possono essere utilizzate le sue librerie dentro programmi C++ da compilare, consente anche di compilare le macro prima della esecuzione in modalità interattiva prima della esecuzione per esecuzioni ad alta velocità.

Per paragrafi che seguono verrà data illustrazione degli elementi essenziali di Root che torneranno utili per analizzare i dati che saranno raccolti nei numerosi corsi di laboratorio del corso di Laurea in Fisica.

6.1 Installazione di Root su Linux

Installare Root su piattaforma Linux è semplicissimo. Prima di tutto bisogna installare i "Prerequisites" ovvero le librerie accessorie, i compilatori e i tool di compilazioni necessari a root. Per fare questo, basta digitare su un motore di ricerca "cern root prerequisites" per trovare il seguente indirizzo:
https://root.cern.ch/build-prerequisites
In questa pagine web troverete le istruzione per installare i pacchetti software necessari e quelli opzionali.

A questo punto ci si muove nella pagina dove è possibile scaricare i sorgenti dell'ultima versione stabile di root:
https://root.cern.ch/downloading-root
Si scarica l'archivio tar.gz contenete i sorgenti e si seguono le istruzioni contenuti nel file README per compilare root. A seconda della potenza del computer la procedure di compilazione potrebbe richiedere un po' di tempo. Quando la procedura di compilazione è terminata, bisogna entrare nel file nascosto contenuto nella propria home directory che si chiama .bashrc, e aggiungere alla fine di questo file di testo la scrittura:
source /percorsodellacartellachecontieneroot/root6-xxquellocheèxx/bin/thisroot.sh

Questa modifica del file .bashrc farà sì che ogni volta che apriremo un terminale, sarà eseguito automaticamente l'istruzione di comando che abbiamo aggiunto e questa istruzione eseguirà lo script thisroot.sh che definirà nella sessione di lavoro sul terminale tutte le variabili d'ambiente necessarie a far funzionare root.

Terminata l'installazione e l'impostazione automatica del file thisroot.sh, se digiteremo la parola **root** sul terminale, a prescindere in quale cartella ci troviamo, root aprirà la sua console di istruzioni da terminale:

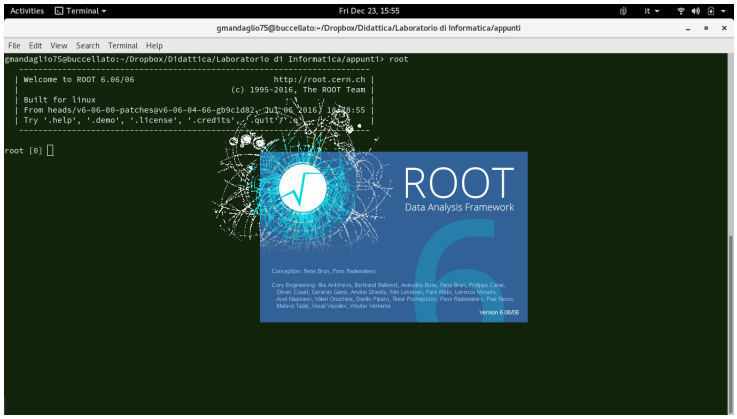


Figura 6.1: screenshot dello start di root da riga di comando.

Dalla console interattiva di root possiamo parlargli in C++, invocando una qualunque delle sue classi. Nella prossima figura, si può vedere il risultato di una breve chat con root:

Nella chat mostrata in figura 6.2 si nota che sono state definite due variabili di tipo double, sono stati assegnati due valori, è stata stampato a schermo il risultato delle somma di dette variabili; è stato creato un oggetto relativo alla classe TF1 di root il cui indirizzo è memorizzato nel puntatore pippo, la creazione dell'oggetto è stato effettuato attraverso il costruttore che passa alla classe il nome dell'oggetto, la parametrizzazione della formula di una funzione, l'intervallo in cui è definita la funzione; poi è stato utilizzato il metodo Draw della classe per far apparire il grafico della funzione; stampiamo a schermo il valore della funzione stimata per $x = 0.3$ attraverso il metodo Eval di TF1.

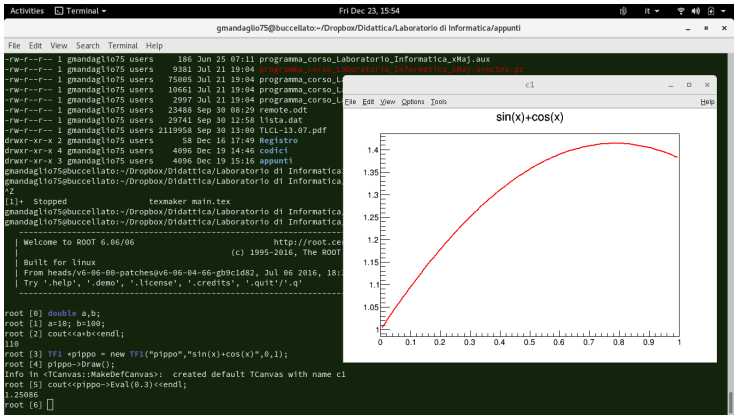


Figura 6.2: screenshot di una breve chat con root e delle sue reazioni.

Ovviamente l'utilizzo di root come fosse una calcolatrice che parla in C++ da riga di comando è comodo e rapido ma non è il solo modo di utilizzarlo. E' possibile scrivere dentro un file una serie di comandi racchiusi tra due graffe (script) o implementare delle funzioni, caricarle e farle eseguire a root. Di questo parleremo nel prossimo paragrafo.

6.2 Script e functions

Proviamo a ricopiare il carteggio di cui parlavamo nel paragrafo precedente in un file con estensione .C nel seguente modo:

```
//script carteggio.C  double a,b; a=10; b=100; cout«a+b«endl; TF1 *pippo = new
    TF1("pippo","sin(x)+cos(x)",0,1); pippo->Draw(); cout«pippo->Eval(0.3)«endl;
```

a questo punto per far eseguire lo script a root basta scrivere su terminale nella cartella dove si trova il file script **root carteggio.C** e poi invio e lui produrrà ciò che si vede in figura 6.3

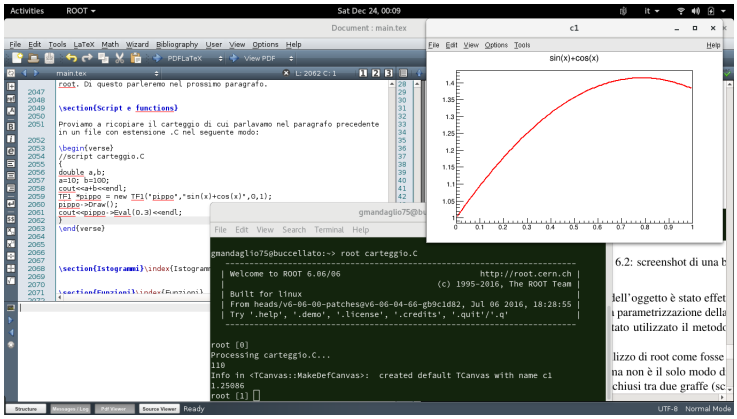


Figura 6.3: screenshot dell'operazione **root carteggio.C**.

É possibile ottenere lo stesso risultato lanciando **root** e poi scrivendo sul terminale di root **.x carteggio.C** e poi invio.

root parla in C++, lo ripeto per l'ennesima volta e come esercizio facciamo ora vedere come un codice scritto in questo corso per la computazione di un istogramma puo' essere facilmente essere convertito in uno script interpretabile da root.

```
//script prova.C
{
int i=0;
double a;
int num_dati=0;
float media;
double somma=0;
double somm=0;
double sqm;
char nome[20];

cout<<"Mi dici il nome del file dei dati?"<<endl;
cin>>nome;

//calcola la somma e la media
ifstream leggimi(nome);
while(!leggimi.eof()){
leggimi>>a;
somma=somma+a;
if(!leggimi.eof()){
num_dati++;
}
}
media = somma /num_dati;
cout<<endl;
cout<<"Hai stampato "<<num_dati<<" numeri."<<endl;
cout<<"La somma dei numeri stampati e' "<<somma<<".\n";
cout<<"La media dei numeri stampati e' "<<media<<" \n";

leggimi.clear();
leggimi.seekg(0);
float *misure = new float[num_dati];
for(i=0; i<num_dati; i++){
leggimi>>misure[i];
somm=somm+pow((misure[i]-media),2);
}
sqm=pow((somm/num_dati), 0.5);
cout<<"Lo scarto quadratico medio e' "<<sqm<<".\n";

//restituisce il valore massimo e il valore minimo

float maxi, mini;
maxi=misure[0];
mini=misure[0];
for(i=0; i<num_dati; i++){
if(misure[i]>maxi){
maxi=misure[i];
}
else if(misure[i]<mini){
```

```
mini=misure[i];
}
}

cout<<"Il valore minimo e': "<<mini<<endl;
cout<<"Il valore massimo e': "<<maxi<<endl;

//creare istogramma
int r;
cout << fixed;
do{
int nint;
cout<<"\nInserisci il numero di intervalli dell'istogramma: \n";
cin>>nint;
float l=(maxi-mini)/(float)nint;
cout<<"Misura intervalli: "<<l<<endl;
int bin[nint];
for(i=0;i<nint;i++){
bin[i]=0;
}
for(i=0;i<nint;i++){
  for(int v=0; v<num_dati; v++){
if((misure[v]>=mini+i*l)&&(misure[v]<=mini+(i+1)*l)){
bin[i]++;
}
}
}
    cout<<"\nIstogramma:"<<endl;
for(int n=0; n<nint; n++){
cout<<mini+l/2+n*l<<"\t"<<bin[n]<<endl;
}
cout<<"\n Istogramma con nuovo binning? (Si? Premi 1):\n";
cin>>r;
}while(r==1);
}
```

Come si può vedere abbiamo tagliato solo la testa del programma con le chiamate agli header files e la definizione della funzione principale, abbiamo lasciato solo il corpo della funzione principale racchiusa tra due parentesi graffe. In figura 6.4 si può vedere lo script in azione attraverso l'interprete fornito da root:

Lo stesso risultato lo avremmo potuto realizzare attribuendo il corpo del precedente script a una funzione, nel seguente modo:

```
//script prova.C
istogrammami()
{
int i=0;
double a;
int num_dati=0;
float media;
```

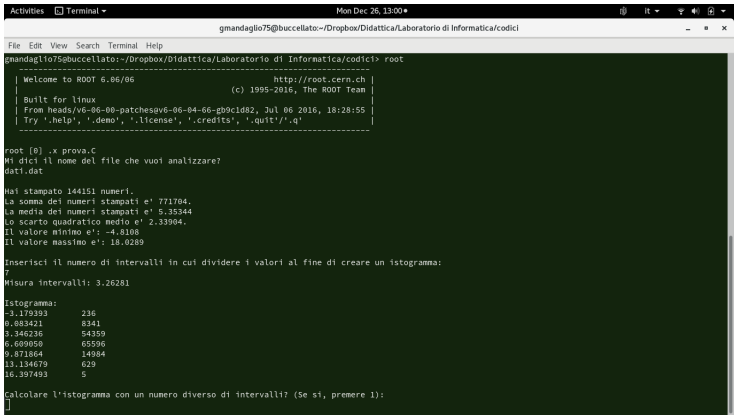


Figura 6.4: screenshot dell'operazione **root carteggio.C**.

```
.....;
..tutto paro paro come sopra ;)
.....;
.....; //etc
cout<<"\n Istogramma con nuovo binning? (Si? Premi 1):\n";
cin>>r;
}while(r==1);
return;
}
```

Abbiamo scritto una funzione di nome istogrammami che non restituisce nessun valore e che ha come corpo lo script precedente.

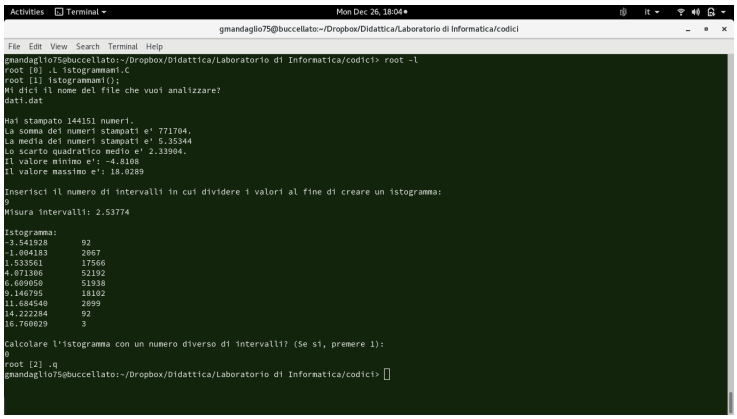


Figura 6.5: Screenshot del caricamento ed esecuzione di una funzione con root.

Per lanciare una funzione da root, prima lo apriamo, carichiamo la funzione con il comando **.L istogrammami.C** (la L sta per link), poi si fa girare la funzione invocandola come faremmo in un normale programma C++ **istogrammami();**. Anche nel caso delle funzioni come negli script non è necessario caricare gli header file come faremmo in un programma C++ compilato. root apparecchia di tutto il necessario, il nostro codice e il suo interprete fanno il resto.

6.3 Istogrammi

Nel paragrafo 3.6.3 abbiamo speso un po' di parole su cosa siano gli istogrammi e quale importanza abbiano nell'analisi statistica dei dati. Abbiamo inoltre imparato a costruire un istogramma partendo dai dati grezzi implementando un codice in C++.

In questo paragrafo vedremo come root fornisca una serie di classi per la costruzione, la visualizzazioni e le operazioni su istogrammi mono- bi- e tri- dimensionali. Le classi[1] preposte allo scopo sono:

- Istogrammi 1D:
  - TH1C : Istogrammi con un *byte* per canale. Massimo valore per *bin*= 127
  - TH1S : Istogrammi con un *short* per canale. Massimo valore per *bin*= 32767
  - TH1I : Istogrammi con un *int* per canale. Massimo valore per *bin*= 2147483647
  - TH1F : Istogrammi con un *float* per canale. Precisione massima 7 *digits*
  - TH1D : Istogrammi con un *double* per canale. Precisione massima 14 *digits*
- Istogrammi 2D:
  - TH2C : Istogrammi con un *byte* per canale. Massimo valore per *bin*= 127
  - TH2S : Istogrammi con un *short* per canale. Massimo valore per *bin*= 32767
  - TH2I : Istogrammi con un *int* per canale. Massimo valore per *bin*= 2147483647
  - TH2F : Istogrammi con un *float* per canale. Precisione massima 7 *digits*
  - TH2D : Istogrammi con un *double* per canale. Precisione massima 14 *digits*
- Istogrammi 3D:
  - TH3C : Istogrammi con un *byte* per canale. Massimo valore per *bin*= 127
  - TH3S : Istogrammi con un *short* per canale. Massimo valore per *bin*= 32767
  - TH3I : Istogrammi con un *int* per canale. Massimo valore per *bin*= 2147483647
  - TH3F : Istogrammi con un *float* per canale. Precisione massima 7 *digits*
  - TH3D : Istogrammi con un *double* per canale. Precisione massima 14 *digits*

Uno dei tanti costruttori per inizializzare un istogramma 1D ha questa forma:

```
TH1D (const char *name, const char *title, Int_t nbinsx, Double_t xlow, Double_t xup)
```

dove le espressioni **Int_t**, **Double_t** si intende i tradizionali interi e reali a doppia precisione del C++, la scrittura differente è introdotta da root in modo da rendere il codice machine-independent, ovvero adatta il tipo di dato alla piattaforma hardware sulla quale dovrà essere eseguito il codice. Il paradigma del costruttore appena mostrato richiede: il nome dell'oggetto che sarà memorizzato nella stringa *name*, il titolo dell'istogramma nella stringa *title*, il numero di bin ovvero il numero di intervalli dell'istogramma *nbinsx*, gli estremi dell'istogramma *xlow* e *xup*.

La classe dispone di una grande varietà di metodi: per caricare i dati *Fill*, per produrre un grafico *Draw*, per gestire la rappresentazione dell'istogramma, effettuare fit etc.

Di seguito quattro righe di codice per rappresentare con un istogramma i dati raccolti in un esperimento nel corso di Laboratorio di Fisica 1 A; l'esperimento consisteva nella misura del diametro di biglie di plastica effettuate con un calibro. I dati sono stati trascritti in un file (Misure.dat).

```
{
   TH1F *pippo = new TH1F("pippo"," ",10,3.6,4);
   ifstream leggimi("Misure.dat");
   double a;
   while(!leggimi.eof()){
   leggimi>>a;
   if(!leggimi.eof()){
```

[1] https://root.cern.ch/root/html516/TH1D.html

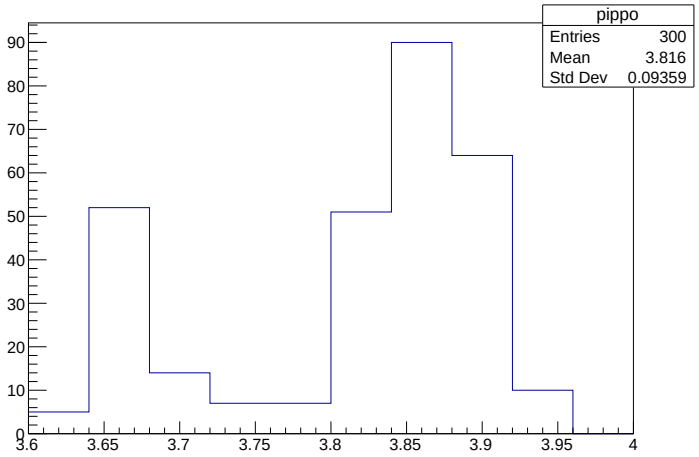


Figura 6.6: Istogramma computato dall'oggetto pippo della classe TH1D.

```
     pippo->Fill(a);
    }
    }
    pippo->Draw();
    //consente di salvare la figura in formato pdf
    c1->SaveAs("grezzo.pdf");
}
```

Il risultato che viene prodotto dall'interpretazione delle quattro righe di codice appena scritte è mostrato in figura 6.6. Una prima cosa che possiamo notare è che il codice che abbiamo scritto per costruire un istogramma nel paragrafo 3.6.3 può essere sostituito da una classe **TH1D** che risolve il problema in un modo molto generale e fornendo molte più funzionalità come quella grafica come possiamo vedere in questo esempio attraverso il metodo *Draw*.

Di seguito mostreremo uno script che nella sostanza fa le stesse cose dello script appena scritto e descritto, ma che contiene un numero di istruzioni maggiori, che servono semplicemente a rendere la presentazione dei nostri dati più ordinata e decorosa, pronta ad essere inclusa in una relazione del lavoro svolto.

```
{
 int mymarkerstyle=20;
 float mymarkersize=1.;
 float mytextsize=0.065;
 int mytextfont=132;

 TCanvas *Canvas_1 = new TCanvas("Canvas_1", "Canvas_1",600,500);
 gStyle->SetOptStat("");
 Canvas_1->SetFillColor(0);
 Canvas_1->SetBorderMode(0);
 Canvas_1->SetBorderSize(2);
 Canvas_1->SetLeftMargin(0.18);
 Canvas_1->SetRightMargin(0.12);
 Canvas_1->SetTopMargin(0.03);
 Canvas_1->SetBottomMargin(0.16);
```

```
 Canvas_1->SetTickx(1);
 Canvas_1->SetTicky(1);

////ZOOM SCATOLA////////////////////////////////////
   TH2F *hpx = new TH2F("hpx"," ",10,3.6,4,10,0.,100);
   hpx->SetStats(0);
   hpx->SetNdivisions(905);
   hpx->GetXaxis()->SetTitleSize(mytextsize);
   hpx->GetYaxis()->SetTitleSize(mytextsize);
   hpx->GetXaxis()->SetLabelSize(mytextsize);
   hpx->GetYaxis()->SetLabelSize(mytextsize);
   hpx->GetXaxis()->SetTitleFont(mytextfont);
   hpx->GetYaxis()->SetTitleFont(mytextfont);
   hpx->GetXaxis()->SetLabelFont(mytextfont);
   hpx->GetYaxis()->SetLabelFont(mytextfont);
   hpx->GetYaxis()->CenterTitle(1);
   hpx->GetXaxis()->CenterTitle(1);
   hpx->GetYaxis()->SetTitleOffset(1.2);
   hpx->GetXaxis()->SetTitleOffset(1.10);
   hpx->SetTitle("");
   hpx->GetYaxis()->SetTitle("Conteggi");
   hpx->GetXaxis()->SetTitle("Diametro Palline (mm)");
   hpx->Draw();
//////////////////////////////////////////////////////////////////////

   TH1F *pippo = new TH1F("pippo"," ",10,3.6,4);
   pippo->SetLineWidth(2);
   pippo->SetLineColor(1);
   ifstream leggimi("Misure.dat");
   double a;
   while(!leggimi.eof()){
   leggimi>>a;
   if(!leggimi.eof()){
     pippo->Fill(a);
    }
    }
    pippo->Draw("same");

   Canvas_1->SaveAs("palline.pdf");
}
```

Descriviamo brevemente lo script un po' meno scarno del primo: la classe **TCanvas** gestisce la scatola in cui vivrà la nostra figura, l'istogramma 2D *hpx* serve per creare una figura finta sulla quale sovrapporre l'istogramma con i nostri dati, infine usiamo la classe **TH1D** per caricare i dati e modificarne l'aspetto attraverso i metodi *SetLineWidth* e *SetLineColor* che cambiano lo spessore e il colore della linea. Il risultato di questo script è mostrato nella figura 6.7.

Ripetiamo l'esperimento misurando la lunghezza di 2000 chiodi da 7 cm. L'esperimento è identico al precedente, cambia solamente il soggetto della misura e la sua numerosità. A parte il divertimento e le ore necessarie nella misura di ciascun chiodo, l'analisi dei dati risulta immediata

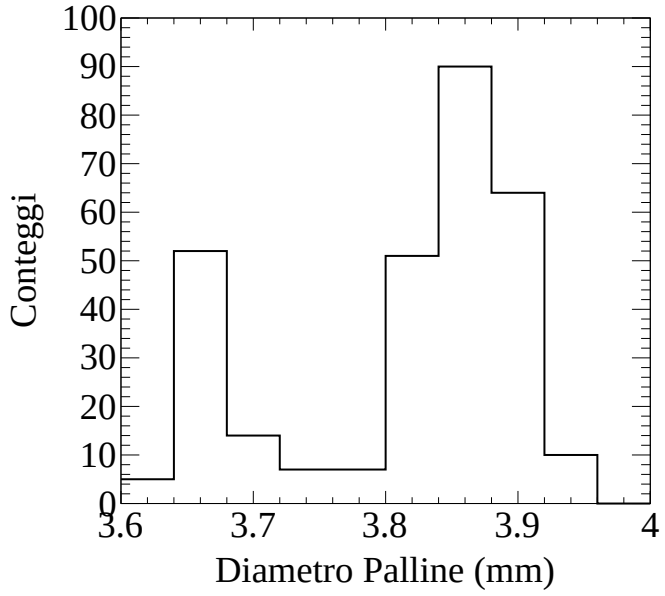


Figura 6.7: Istogramma computato dallo script un po meno scarno.

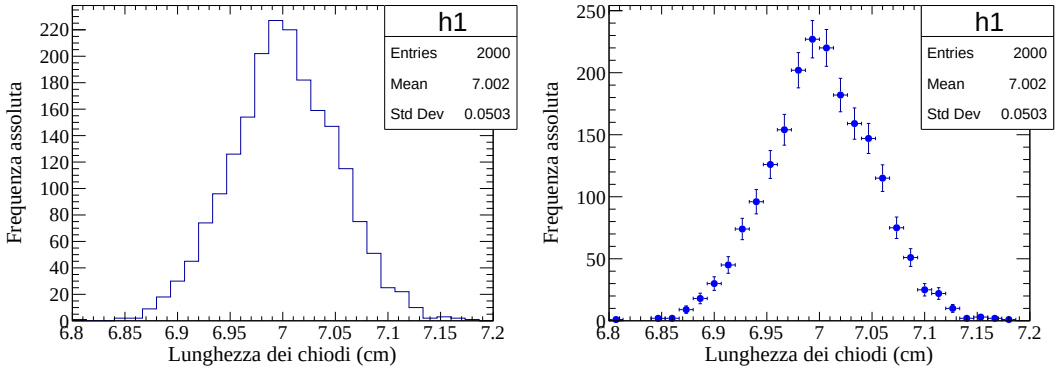


Figura 6.8: Istogrammi esperimento dei chiodi. Sinistra rappresentazione equivalente dell'istogramma, ogni punto fornisce la frequenza assoluta, l'errore sulla lunghezza è legato alla larghezza del bin, l'errore sull'altezza è l'errore statistico (la radice quadrata del numero di conteggi dell'istogramma).

reciclando il codice scritto per l'esperimento delle palline. É sufficiente modificare gli estremi dell'istogramma, il titolo degli assi e il risultato può essere osservato in figura 6.8.

Abbiamo detto che è possibile computare anche istogrammi bi-dimensionali. Prima di far vedere i modi in cui possono essere visualizzati questi istogrammi, descriviamo cosa sono attraverso un semplice esempio: immaginiamo di ripetere l'esperimento della misura della lunghezza dei chiodi e di misurare questa volta anche un'altra quantità come ad esempio il peso del chiodo o il diametro a metà lunghezza del chiodo. In questo secondo esperimento per ogni campione abbiamo due misure che lo caratterizzano. A questo punto potremmo costruire un istogramma utilizzando due quantità, l'altezza e il peso. Identifichiamo l'asse x con l'altezza e l'asse y con il peso, gli intervalli in cui dividiamo i due assi individueranno sul piano lunghezza peso delle superfici rettangolari, a questo punto possiamo usare l'asse z per contare il numero di volte che un chiodo è stato classificato appartenere ad un dato rettangolo. Questo tipo di rappresentazione può farci capire se esistono correlazioni tra le diverse misure che caratterizzano l'oggetto: il chiodo più pesante è più lungo, forse è più corto, oppure non ha legame con la lunghezza; chi può dirlo? Lo

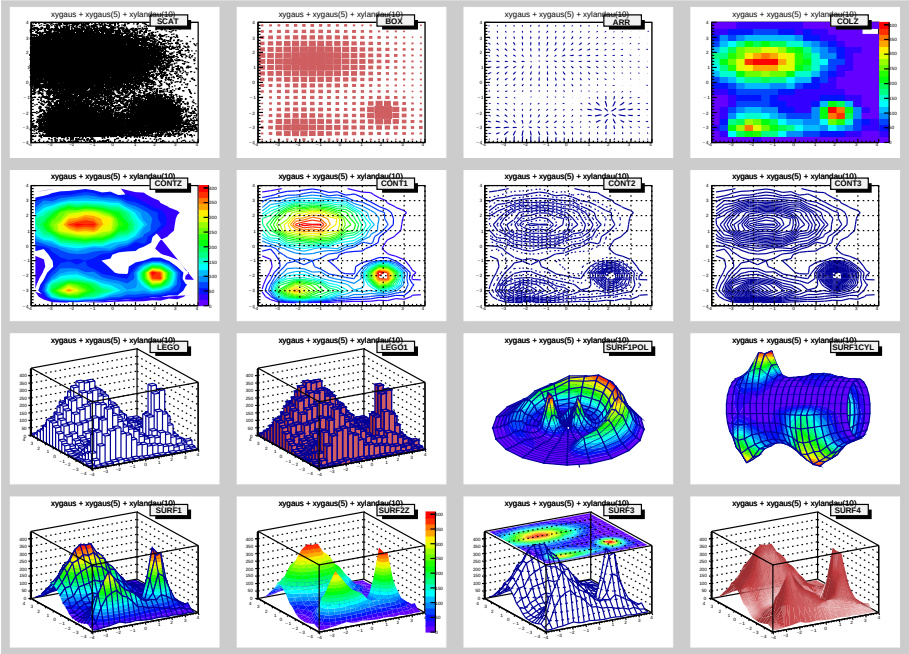


Figura 6.9: Modi possibili per rappresentare un istogramma 2D con root.

può dire l'analisi con l'istogramma 2D.

In figura 6.9 sono riportate alcuni modi di rappresentare un istogramma bidimensionale con root.

6.4 Funzioni matematiche

Le funzioni matematiche possono essere rappresentate attraverso le classi **TF1**, **TF2** e **TF3**, ciascuna consente di modellizzare funzioni di una, due o tre variabili. Le funzioni matematiche possono essere utilizzate come modello di riferimento per effettuare dei fit oppure per generare distribuzioni di numeri con un MonteCarlo. Di seguito mostreremo alcuni esempi, ma ormai avrete capito che non avete bisogno di ulteriori aiuti, spiegazioni o suggerimenti. Considerando che root parla il C++(e son 10 volte che lo ripeto), voi ne conoscete i rudimenti, il CERN fornisce attraverso il suo sito web una larga documentazione delle sue classi e manuali sfogliabili sul web sul sito o scaricabili in formato pdf (https://root.cern/doc/master/index.html); non avete più scuse usatelo usatelo usatelo, e se scoprite cose nuove condividetele con i colleghi, e se trovate errori (bug) comunicateli ai colleghi del CERN perchè li possano correggere.

Immaginiamo di aver sentito parlare a lezione di Fisica generale 1 del moto oscillatorio smorzato, e di esserne rimasti particolarmente colpiti essendo tutti quanti patiti di altalene, biglie, tappetini elastici, chitarre etc. A questo punto vogliamo visualizzare le funzioni che descrivono questo tipo di fenomeno, l'esponenziale che definisce lo smorzamento e la funzione sinusoidale per le oscillazioni:

```
//script funzioni.C
{
TF1 *expy = new TF1("expy","10*exp(-x/3)",0,15);
TF1 *expyneg = new TF1("expyneg","-10*exp(-x/3)",0,15);
TF1 *expsin = new TF1("expsin","[0]*exp(-x/[1])*sin([2]*x)",0,15);
expsin->SetParameters(10,3,4);
expy->SetLineColor(1);
```

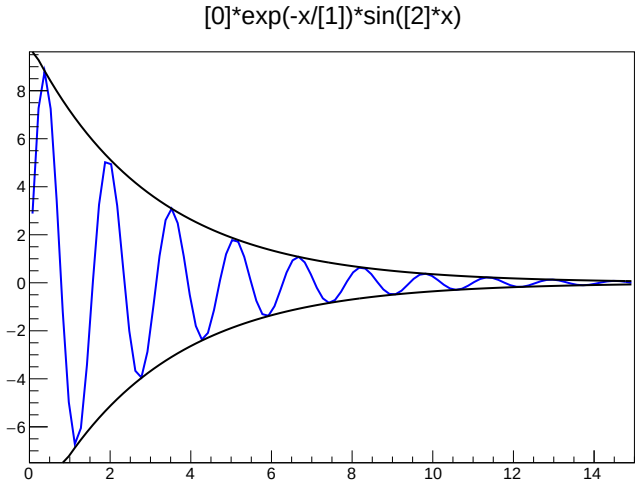


Figura 6.10: Figura prodatta dallo script funzione.C.

```
expy->SetLineWidth(2);
expyneg->SetLineColor(1);
expyneg->SetLineWidth(2);
expsin->SetLineColor(4);

expsin->Draw();
expy->Draw("same");
expyneg->Draw("same");
}
```

In questo semplice script vediamo come scrivere un funzione in modo completo dentro il costruttore nell'oggetto *expy* e come scriverla in dipendenza di parametri ($[0], [1] etc$) e come passare i valori alla funzione *expsin* attraverso il metodo *SetParameters*.

Proviamo a scrivere una funzione un po' più complicata, immaginiamo un fenomeno che obbedisca ad una legge della forma:

$$R_p = A \cdot \frac{\sqrt{1-(\frac{sin(\theta)}{n_1})^2} - n_1 \cdot cos(\theta)}{\sqrt{1-(\frac{sin(\theta)}{n_1})^2} + n_1 \cdot cos(\theta)} \tag{6.1}$$

scriviamo lo script capace di mettere a grafico la funzione:

```
{  //script funzione_complicata.C
  TF1 *ff=new TF1("ff","[0]*pow((sqrt(1-pow(sin(x)/[1],2))
                        -[1]*cos(x))/  (sqrt(1-pow(sin(x)/[1],2))
                        +[1]*cos(x)),2)",25*3.14/180,75*3.14/180);
  ff->SetParameter(0,2320);
  ff->SetParameter(1,1.6);
  ff->Draw();
  }
```

Notiamo come passiamo i valori ai due parametri attraverso il metodo *SetParameter* e la figura 6.11 prodotta dallo script.

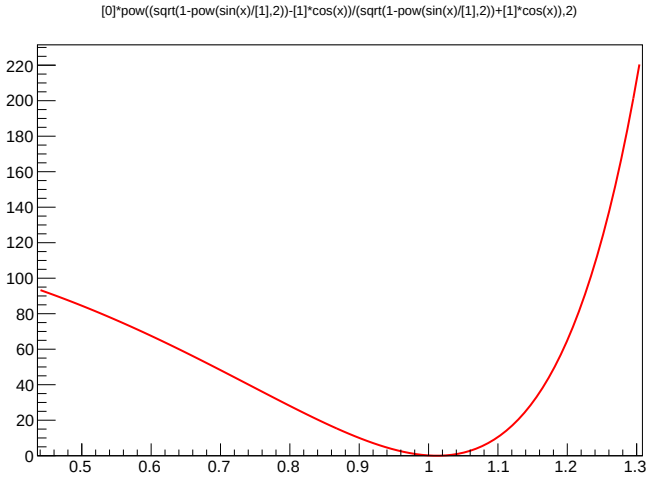


Figura 6.11: Figura prodatta dallo script funzione.C.

Non bisogna dimenticare che la programmazione è uno strumento per risolvere problemi e root fornisce un arsenale di possibilità per analizzare dati, ma tutto ciò che scriviamo deve obbedire alle regole della matematica e possibilmente avere anche senso dal punto di vista della fisica. Se ad esempio assegnassimo al parametro n_2 dell'equazione 6.1 il valore 0.5 cosa succederebbe? L'analisi ci dice che la funzione seno è limitata tra ± 1 quindi se $n_2 < 1$ può accadere che per certi valori di θ la funzione non ammetta valori reali. Proviamo a "girare" lo script assegnando il valore 0.5 ($ff->SetParameter(1,0.5);$), il risultato che vedremo è il seguente:

```
root [0]
Processing funz2.C...
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
Warning in <TCanvas::ResizePad>: Inf/NaN propagated to the pad.
Check drawn objects.
Warning in <TCanvas::ResizePad>: c1 height changed from 0 to 10

root [1] Warning in <TCanvas::ResizePad>: Inf/NaN propagated to the pad.
Check drawn objects.
Warning in <TCanvas::ResizePad>: c1 height changed from 0 to 10

Warning in <TCanvas::ResizePad>: Inf/NaN propagated to the pad.
Check drawn objects.
Warning in <TCanvas::ResizePad>: c1 height changed from 0 to 10
```

root produce dei messaggi di errore e li stampa a schermo, ma l'errore lo abbiamo commesso noi, non vuol dire che root non funziona anzi vuol dire il contrario.

6.5 Fit

Il fit è una procedura matematica-numerica che consente di determinare i parametri e i relativi errori di una data legge (funzione) che noi assumiamo possa descrivere l'andamento di dati correlati. Se mettiamo un filo metallico in trazione con un peso e ne misuriamo la lunghezza, poi aggiungiamo un ulteriore peso e riusciamo ad apprezzare un aumento della lunghezza, ripetiamo l'operazione e ci annotiamo queste due grandezze, ad esempio potremmo supporre che trazione e allungamento siano

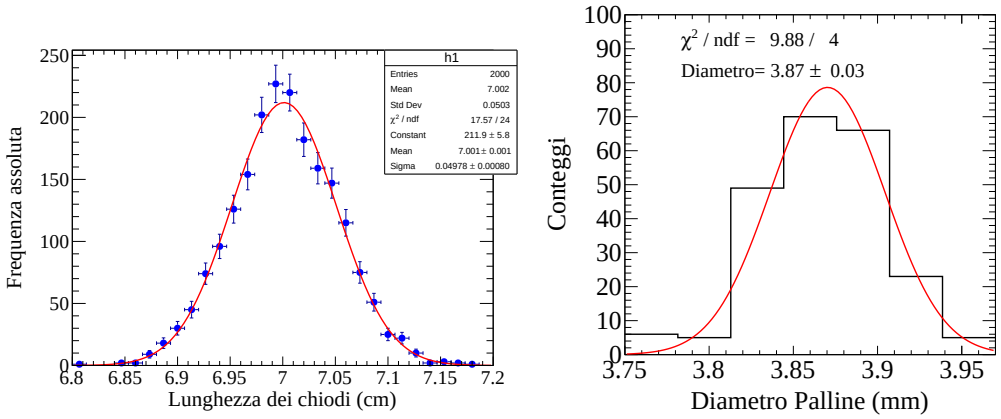


Figura 6.12: .

legate da una legge di tipo lineare; lo stesso se sottoponiamo a riscaldamento una barra metallica; o che le oscillazioni di un pendolo obbediscano ad una legge come quella messa in grafico in figura 6.5 etc. Finora abbiamo parlato di leggi di tipo fisico, ma la legge che descrive l'andamento dei dati potrebbe essere di tipo statistico. La distribuzione delle lunghezze dei chiodi fabbricati per lavori di carpenteria, le palline di plastica di cui abbiamo parlato nei paragrafi precedenti producono istogrammi con distribuzione di tipo gaussiano, quindi l'istogramma potrebbe essere fittato con una gaussiana. Quindi non solo leggi fisiche ma anche andamenti statistici.

root mette a disposizione una serie di funzione già implementate, di largo uso, per effettuare i fit oppure consente all'utente di scrivere da se la legge con la quale su vuole fare il fit attraverso una classe di tipo **TF1** e usarla in un calcolo completamente personalizzato.

Le funzioni preconfezionate di root sono la Gaussiana (**gaus**), la Landau (**landau**), i polinomi (**polN**).

Se volessimo fittare la distribuzione delle palline e dei chiodi presentate nelle figure 6.7 e 6.8 basterebbe usare il metodo **Fit** della classe **TH1F** passandogli l'opzione **gaus**.

```
pippo->Fit("gaus");
```

e otterremmo il risultato riportato in figura 6.12.

La distribuzione è chiare negli esempi appena esposti, e invocando direttamente il metodo di fit abbiamo buone possibilità di ottenere il risultato desiderato. In generale quando si effettuano delle operazioni di fit al calcolatore, gli algoritmi di ricerca dei parametri hanno bisogno di un suggerimento, hanno bisogno che gli diamo dei valori vicini a quelli attesi per poi delegare loro alla ricerca dei migliori valori, migliori nel senso quelli che producono la curva che più si avvicina ai dati. I dati suggeriti al fit vengono chiamati parametri di innesco del fit. Uno dei metodi per stabilire la bontà di un fit è quello del χ^2. La variabile del χ^2 si costruisce come la sommatoria degli scarti quadratici tra i valori sperimentali e quelli del fit normalizzati al quadrato della somma dei rispettivi errori (errore sperimentale ed indeterminazione della funzione di fit). Ricercare i valori dei parametri della funzione che rendono minima questa variabile può rappresentare un criterio per la ricerca del fit migliore (il best fit).

Supponiamo ora di aver effettuato un esperimento che metter in relazione un angolo rispetto al quale un foto-rivelatore misura un segnale espresso in milli Volt. I dati in questione sono scritti nel file *datuzzi.dat* e in Tabella 6.2.

Proviamo ad utilizzare la legge definita dalla funzione 6.1 ed a implementare un fit. Lo script che segue è completo di tutti i ghirigori necessari per ottenere una figura decorosa in realtà le uniche istruzioni necessarie per portare a casa il fit sono segnate nel codice con tre asterischi.

Tabella 6.1: Tabella dei dati scritti nel file datuzzi.dat.

θ_i (rad.)	R_p (mV)
0.441868	79.10
0.616312	53.20
0.790757	18.70
0.860534	9.16
0.965201	2.00
1.03498	7.60
1.06987	13.70
1.13965	43.45
1.20942	101.50
1.31409	249.20

```
//codice con fit su dati senza attribuire l'errore
// Script fitnoerr
{
  int mymarkerstyle=20;
  float mymarkersize=1.;
  float mytextsize=0.065;
  int mytextfont=132;
  TCanvas *Canvas_1 = new TCanvas("Canvas_1", "Canvas_1",600,500);
  Canvas_1->SetFillColor(0);
  Canvas_1->SetBorderMode(0);
  Canvas_1->SetBorderSize(2);
  Canvas_1->SetLeftMargin(0.15);
  Canvas_1->SetRightMargin(0.07);
  Canvas_1->SetTopMargin(0.03);
  Canvas_1->SetBottomMargin(0.16);
 //  Canvas_1->SetGridx();
 //  Canvas_1->SetGridy();
  Canvas_1->SetTickx(1);
  gStyle->SetOptStat(1111);
  TF1 *ff=new TF1("ff","[0]*pow((sqrt(1-pow(sin(x)/[1],2))
                       -[1]*cos(x))/  (sqrt(1-pow(sin(x)/[1],2))
                       +[1]*cos(x)),2)",25*3.14/180,75*3.14/180);//***
  //parametri di innesco
  ff->SetParameter(0,2320);//***
  ff->SetParameter(1,1.6);//***
  TGraph *pippo= new TGraph("datuzzi.dat");//**
  pippo->SetTitle("");
  pippo->GetXaxis()->SetTitleSize(mytextsize);
  pippo->GetYaxis()->SetTitleSize(mytextsize);
  pippo->GetXaxis()->SetLabelSize(mytextsize);
  pippo->GetYaxis()->SetLabelSize(mytextsize);
  pippo->GetXaxis()->SetTitleFont(mytextfont);
  pippo->GetYaxis()->SetTitleFont(mytextfont);
  pippo->GetYaxis()->CenterTitle(1);
  pippo->GetXaxis()->CenterTitle(1);
```

```
  pippo->GetYaxis()->SetTitleOffset(1.1);
  pippo->GetXaxis()->SetTitleOffset(1.10);
  pippo->GetYaxis()->SetTitle("R_{p} (mV)");
  pippo->GetXaxis()->SetTitle("#theta_{i} (rad.)");
  pippo->SetMarkerStyle(20);
  pippo->Draw("ap");
  pippo->Fit("ff");//***
  float media,sigma,media2,sigma2,chi,ndf;
  TLegend *leg = new TLegend(0.2,0.7,0.55,0.85,NULL,"brNDC");
  leg->SetBorderSize(0);
  leg->SetLineColor(1);
  leg->SetLineStyle(1);
  leg->SetLineWidth(1);
  leg->SetFillColor(0);
  leg->SetTextSize(0.05);
  leg->SetTextFont(132);
  leg->SetFillStyle(1001);
  media=ff->GetParameter(0);
  sigma=ff->GetParError(0);
  media2=ff->GetParameter(1);
  sigma2=ff->GetParError(1);
  chi=ff->GetChisquare();
  ndf=ff->GetNDF();
  leg->AddEntry(pippo,Form("#chi^{2} / ndf = %6.2f / %3.0f",chi,ndf),"");
  leg->AddEntry(pippo,Form("A=%5.2f #pm %5.2f",media,sigma),"");
  leg->AddEntry(pippo,Form("n2=%5.2f #pm %5.2f",media2,sigma2),"");
  leg->Draw();
  Canvas_1->SaveAs("fit.pdf");
}
```

Il fit riportato in figura 6.14 appare ottimo, la curva è molto vicina ai punti sperimentali e i parametri ragionevoli per la fisica che regola questo fenomeno, tuttavia la variabile del χ^2 è grande e questo in generale vuol dire che il fit non è buono. Come interpretare questo risultato?

In realtà la nostra percezione è corretta, il fit è buono, ma è anche vero che il χ^2 è stato calcolato correttamente. Il motivo dell'apparente discordanza è legato fatto che non abbiamo associato ai valori sperimentali i relativi errori. In tabella **??** riportiamo gli stessi dati in questione, ma questa volta riportiamo anche i relativi errori.

Rifacciamo il fit e modifichiamo lo script utilizzato in precedenza facendo una modifica piccolissima: passiamo dalla classe **TGraph** alla classe **TGraphErrors** che consente di tenere in conto anche degli errori sulle variabili x e y.

```
// script fitsierr.C
//tutto come nella macro precedente
//cambia solo la classe che si preoccupa
//di caricare e rappresentare i dati
//passiamo da TGraph a TGraphErrors
.....
.....
TGraphErrors *pippo= new TGraphErrors("datuzzierr.dat");
.....
```

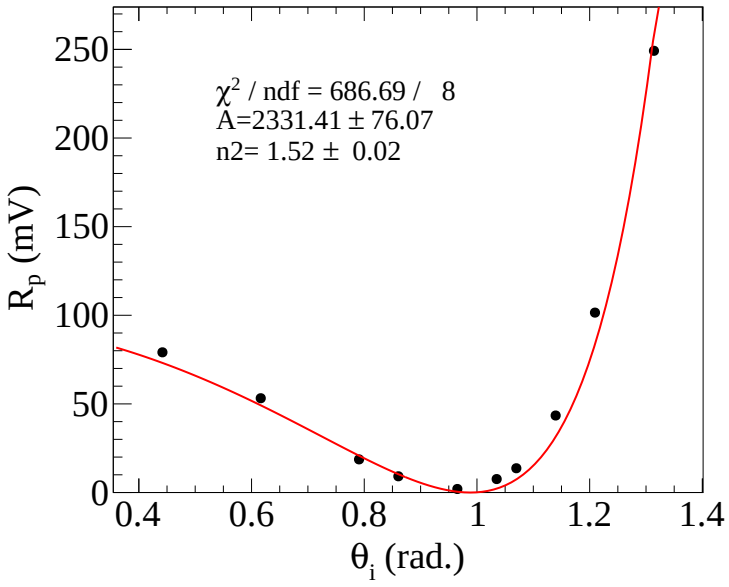


Figura 6.13: Risultato dello script Script fitnoerr.

Tabella 6.2: Tabella dei dati scritti nel file datuzzierr.dat.

θ_i (rad.)	R_p (mV)	err. θ_i (rad.)	err. R_p (mV)
0.441868	79.10	0.0349067	5
0.616312	53.20	0.0349067	5
0.790757	18.70	0.0349067	5
0.860534	9.16	0.0349067	5
0.965201	2.00	0.0349067	5
1.03498	7.60	0.0349067	5
1.06987	13.70	0.0349067	5
1.13965	43.45	0.0349067	5
1.20942	101.50	0.0349067	5
1.31409	249.20	0.0349067	5

```
Canvas_1->SaveAs("fiterr.pdf");
```

Il risultato di questo secondo fit è leggermente differente dal precedente, ma in perfetto accordo nei limiti degli errori sul parametri, mentre il χ^2 è decisamente piccolo e in accordo con la nostra percezione della bontà del fit. In questo secondo fit, il calcolatore ha computato anche l'errore sperimentale, mentre nel primo fit non avendo questa informazione ha normalizzato gli scarti al solo errore commesso nel fit.

Spieghiamo meglio questa questione, il fit (la curva rossa nelle figure di questo paragrafo) ha un errore associato, non è priva da errore. Se notate in figura i parametri A e n_2 hanno una indeterminazione. Ebbene se propaghiamo gli errori associati ai parametri nella formula descritta dalla funzione 6.1 otteniamo l'errore associato al fit.

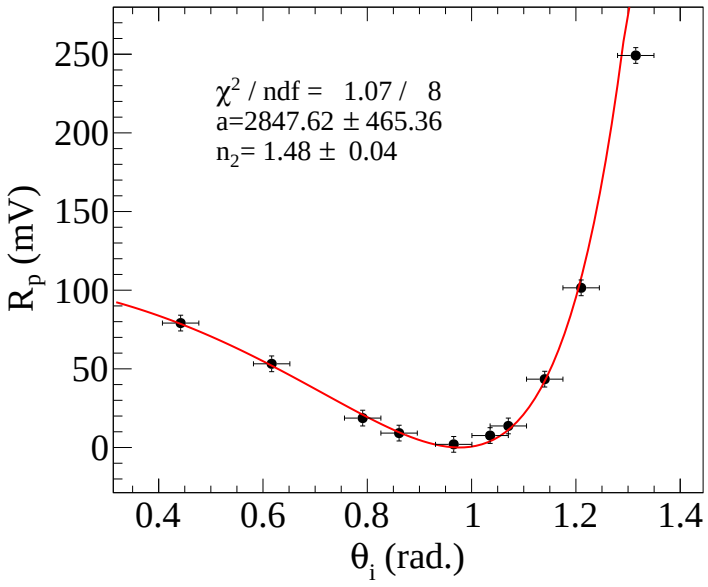
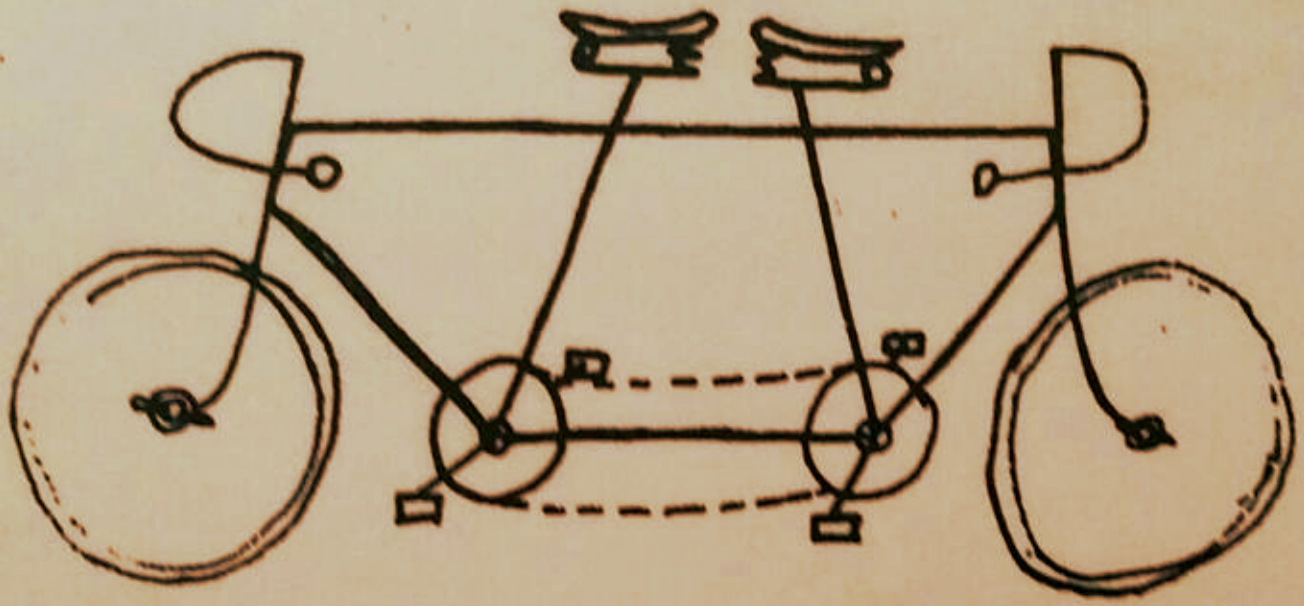


Figura 6.14: Risultato dello script Script fitsierr.



Bibliography

PROBARE ET REPROBARE !

disegno di : Bruno Touschek