Informatica 1

Lorenzo Uboldi

Versione aggiornata al 12 maggio 2017,
Lorenzo Uboldi (lorenzo.uboldi@studenti.unimi.it), Milano.

Indice

Guida alla lettura vii

Leggimi please! ix

1 Introduzione alla programmazione 1
- 1.1 Com'è fatto un computer? . . . 1
- 1.2 Cosa vuol dire programmare? . . . 2
  - 1.2.1 Linguaggio macchina . . . 2
  - 1.2.2 Linguaggi di programmazione . . . 2
  - 1.2.3 Da C++ a linguaggio macchina . . . 3
- 1.3 Il tuo primo programma . . . 3
  - 1.3.1 Il codice . . . 3
  - 1.3.2 Spazio di tab e leggibilità del codice . . . 5
- 1.4 Il file sorgente, il terminal e il compilatore . . . 6
  - 1.4.1 Quale sistema operativo? . . . 6
  - 1.4.2 I file sorgente . . . 7
  - 1.4.3 Il terminale e il compilatore . . . 7
  - 1.4.4 La catena di compilazione: preprocessore, compilatore, assembler e linker. 8

2 Tipi di dato standard e variabili 11
- 2.1 I dati in informatica . . . 12
- 2.2 Tipi di dato specifici . . . 14
  - 2.2.1 Int . . . 14
  - 2.2.2 Long . . . 17
  - 2.2.3 Short . . . 18
  - 2.2.4 Float . . . 18
  - 2.2.5 Double . . . 19
  - 2.2.6 Char . . . 20
  - 2.2.7 Bool . . . 20
  - 2.2.8 Riassumiamo... . . . 20

2.3 Operazioni tra tipi di dato diversi . . . 22
2.3.1 Conversione tra tipi di dato diversi . . . 22
2.3.2 Operazioni numeriche tra interi e decimali . . . 23
2.4 Il discorso della precisione: l'importanza per un Fisico . . . 26
2.5 Visibilità delle variabili . . . 29
2.6 Macro e costanti . . . 30
2.7 *cout* e *cin* . . . 31

3 Rappresentazione in memoria **33**
3.1 La struttura della memoria . . . 34
3.1.1 Indirizzi di memoria . . . 34
3.2 Stack e Heap . . . 35
3.3 I dati e la memoria . . . 36
3.4 Array: elementi di memoria contigui . . . 37
COMPLEMENTI . . . 39
3.A Architetture a 32 e 64 bit . . . 39

4 Strutture di controllo e cicli **41**
4.1 Strutture di controllo . . . 41
4.1.1 *if... else* . . . 42
4.1.2 Operatori binari e logici . . . 46
4.2 Cicli . . . 49
4.2.1 *while* . . . 49
4.2.2 *do... while* . . . 51
4.2.3 *for* . . . 52
4.2.4 Gli enunciati *break* e *continue* . . . 53
4.2.5 I cicli infiniti . . . 54
4.2.6 *switch... case* . . . 56

5 Puntatori **59**
5.1 L'operatore "&" . . . 60
5.2 Dereferenziare i puntatori: l'operatore $*$ . . . 61

6 Array e Matrici **65**
6.1 Array statici . . . 66
6.1.1 Dichiarare array . . . 66
6.1.2 Indici e... out of range! . . . 68
6.1.3 Puntatori e Array . . . 68
6.2 Matrici: array di array . . . 70
6.2.1 Rappresentazione della matrice in memoria . . . 72
6.2.2 Un esempio concreto sui pericoli dell'out of range . . . 72
6.3 Operazioni su array: qualche algoritmo . . . 75
6.3.1 Elemento massimo e minimo . . . 75
6.3.2 Ordinamento . . . 76

7 In/Out: i file **79**
7.1 Canali standard . . . 79
7.2 Comunicazione con i file . . . 80
7.2.1 Scrivere su file . . . 80
7.2.2 Leggere da file . . . 82
7.2.3 Opzionale: una versione generalizzata . . . 84

8 Allocazione Dinamica **85**
8.1 Gli operatori *new* e *delete* . . . 85
8.2 Intermezzo: di più su Stack e Heap . . . 87
8.3 Array dinamici . . . 88
8.3.1 Ingrandire e rimpicciolire vettori . . . 89
8.3.2 Alcuni errori . . . 90
8.4 Matrici dinamiche . . . 92

9 Struct **95**
9.1 Definire un nuovo tipo di dato . . . 95
9.1.1 L'operatore punto (".") . . . 96
9.2 Struct e array . . . 98
9.3 Struct, puntatori, allocazione dinamica e memoria . . . 98
9.3.1 L'operatore "->" . . . 99
9.3.2 L'operatore *sizeof* . . . 100
9.3.3 Struct e indirizzi di memoria . . . 103
9.4 Il C++ e gli oggetti . . . 103

10 Funzioni **105**
10.1 Dichiarare, definire e usare una funzione . . . 106
10.1.1 *return variabile;* . . . 109
10.1.2 *void funcion* . . . 110
10.1.3 Funzioni e array . . . 111
10.2 La chiamata a funzione . . . 113
10.3 Passaggio di argomenti: *by value, by pointer e by reference* . . . 114
10.3.1 Allocazione dinamica dentro a funzioni . . . 116
10.3.2 Stream e funzioni . . . 118
10.4 Overload di funzioni . . . 119
10.4.1 Default arguments . . . 122
10.5 Alcune funzioni utili . . . 123
10.5.1 Lettura da file: dimensione non nota . . . 123
10.5.2 Random numbers . . . 124
10.5.3 Calcolo del tempo di esecuzione . . . 127
10.5.4 Funzioni di ordinamento e statistica . . . 127
COMPLEMENTI . . . 128
10.A Argomenti del *main*: *argc* e *argv* . . . 128
10.B La Ricorsione . . . 129
10.C Variabili *static* . . . 130

11 Librerie **133**
11.1 Definire una libreria: *header* e *.cpp* . . . 133
11.1.1 Compilazione parziale: preprocessore, linker e *#ifndef* . . . 135
11.2 Il Makefile . . . 137

12 Containers: strutture di dati **139**
12.1 La Queue . . . 139
12.2 Il Vettore . . . 139

13 ROOT CERN **141**
13.1 Librerie da includere, flag di compilazione, TApplication e TCanvas . . . 142
13.2 Plot e Istogrammi . . . 143
13.2.1 TH1F . . . 143
13.2.2 TGraph . . . 146
13.2.3 Nomi degli assi e funzioni . . . 147

A Rapida guida all'uso di Linux **151**
A.1 Il filesystem . . . 152
A.1.1 Navigare nel filesystem . . . 152
A.1.2 Modificare il filesystem . . . 153
A.2 Secure Shell: Linux in remoto . . . 154

B La reference online **157**

C Installazione di ROOT **159**
C.1 Installare su Linux . . . 159
C.1.1 La tua distribuzione c'è! . . . 159
C.1.2 Sei sfortunato... . . . 160
C.2 Installare su Mac . . . 160

D Un esame di laboratorio risolto **161**
D.1 21 Luglio 2015 . . . 163

Guida alla lettura

Questo scritto è "a più livelli". Come noterai sono presenti diverse note a piè di pagina, appendici, ecc... Tutto ciò è un "di più", non strettamente necessario a passare l'esame, ma se da subito ti accorgi che l'Informatica ha un suo fascino, potrebbe stimolarti ad "andare oltre": qualche piccola curiosità quà e là, qualche approfondimento, qualche chiarimento...

Ovviamente, puoi saltare tutto e andare solo al dunque.

In appendice, sono presenti una rapida guida per familiarizzare con Linux, una guida per l'installazione di ROOT e qualche tema d'esame risolto.

Ci tengo a sottolineare una cosa: queste note non sono appunti del corso o, tanto meno, una "*dispensa del corso*" ufficiale. Piuttosto, dopo essermi appassionato all'Informatica e dopo aver approfondito questa materia, ho deciso, anni dopo il corso, di provare a "buttare giù" qualche riga. Ti dico tutto ciò per metterti in guardia: ho seguito una mia linea espositiva cercando di coprire tutti gli argomenti del corso, ma lungi da me l'intento di sostituirmi alle lezioni, preziosissime e da frequentare! Confrontati sempre con loro!

Infine, così come il corso, queste note sono pensate per essere "introduttive" al `C++` e alla programmazione; per cui, a volte, l'esposizione è semplicistica per favorire la comprensione. Sappi, però, che il `C++` è estremamente più vasto, complesso e completo di quanto da queste note possa apparire, citando l'autore del libro che più mi ha istruito[1]: "*C* -e, per eredità, il `C++-` *has never been a language to bar entry where others fear to tread*".

Detto ciò, buona lettura... Ah, se trovi imprecisioni, errori o chissà quale altra castroneria, segnalamela (`lorenzo.uboldi@studenti.unimi.it`), te ne sarò grato!

[1] Thinking in C++, Bruce Eckel

Leggimi please!

L'attuale versione di queste note è un *work in progress*: in primis non è ancora completa, in secondo luogo è in corso di revisione. Purtroppo, vi sarà sicuramente qualche errore.

Essendo uno dei primi lettori, ti chiedo gentilmente di comunicarmi tutto per poter correggere (lorenzo.uboldi@studenti.unimi.it), grazie!

(Mi auguro di far sparire al più presto questa stupida pagina...)

CAPITOLO 1

Introduzione alla programmazione

Tutti i giorni, quando accendiamo il computer o usiamo il nostro smartphone, abbiamo a che fare con programmi. Ti sei mai chiesto cosa c'è sotto? Come è nato e come è stato costruito quel programma?

Se anche queste domande non ti avessero mai neanche lontanamente sfiorato, nel corso di Informatica 1 ti toccherà imparare le basi del retroscena dei programmi che usi con nonchalance tutti i giorni. Vedremo giusto le fondamenta della programmazione, niente di più! Chi lo sa, magari troverai perfino interessante questo mondo...

Bando alle ciance, iniziamo!

1.1 Com'è fatto un computer?

Il computer (calcolatore in inglese) è una macchina in grado di eseguire istruzioni e di fare calcoli: per come lo schematizzeremo noi, è composto dal processore, dalla memoria RAM e dai dispositivi di input e output.

Il processore è la parte che fa i conti, che ragiona: è il cervello del computer, è la parte pensante.

La memoria di cui parleremo, trattando di programmazione, è sempre la RAM (Random Access Memory); quando un programma "gira", viene caricato sulla RAM, non sull'hard disk. Sistema operativo, drivers, programmi "vivono" tutti nella RAM: questa è la memoria con cui ci interfacciamo quando programmiamo.

I dispositivi di input sono (al nostro livello) solo due: la tastiera e l'hard disk. I files, infatti, a differenza dei programmi, sono archiviati sull'hard disk e un programma può accedervi facilmente, sia in lettura che in scrittura. L'hard disk è, quindi, sia un dispositivo di output che di input.
L'altro dispositivo di output sarà, per noi, lo schermo.

1.2 Cosa vuol dire programmare?

Il computer è una macchina che ragiona in linguaggio binario: la lingua che "parla" è fatta di 0 e 1. Ovviamente i computer non nascono "imparati", loro sanno fare solo conti: bisogna istruirli su che conti fare.

Un programma non è altro che un'istruzione per il computer, un'insieme di parole (del computer, non le nostre parole!) che, una dopo l'altra, con la loro logica, rappresentano dei calcoli e delle operazioni.

1.2.1 Linguaggio macchina

Ecco quindi introdotto il concetto di linguaggio macchina: è una serie di 0 e 1 che rappresentano istruzioni. Il computer, oltre a saper elaborare questi 0 o 1, li sa immagazzinare: l'unità logica che può contenere questo dato è il *bit* (un bit o vale zero o vale uno). La memoria del computer è, in realtà, organizzata in *byte*: un byte è composto da 8 bit. Il byte è l'unità elementare della memoria, nei termini che interessano a noi: i dati che noi trattiamo sono sempre multipli di questa unità.

Ma torniamo al linguaggio macchina. Se volessimo scrivere un programma (anche il più semplice, un banale calcolo) sarebbe quasi impossibile, o comunque di una complicazione estrema.

Con la nascita dell'informatica, per rendere più agevole la scrittura di programmi da parte degli informatici, sono stati sviluppati i primi linguaggi di programmazione, decisamente più comprensibili rispetto ad una serie di 0 e 1.

1.2.2 Linguaggi di programmazione

I linguaggi di programmazione si dividono in due grandi categorie: di basso livello e di alto livello. Cosa vuol dire? Non ci si riferisce certo alla qualità, bensì a quanto il linguaggio di programmazione è vicino al linguaggio macchina.

Un linguaggio di basso livello è composto da istruzioni elementari, del tipo: prendi il dato, spostalo in tale cella di memoria, prendi l'altro dato, spostalo nell'altra cella, ecc. . . .

Un linguaggio di basso livello è estremamente complicato da comprendere, e scrivere programmi complessi richiede capacità veramente notevoli, oltre a tempi ingenti. Un esempio di linguaggio di basso livello è l'*assembly*.

I linguaggi ad alto livello sono nati per semplificare ulteriormente la vita ai programmatori: sebbene l'assembly sia molto più comprensibile dei fatidici 0 e 1, scrivere un programma in assembly è estremamente complesso e lungo. Nei linguaggi ad alto livello, molte delle istruzioni necessarie in assembly sono rese implicite. È tutto molto più "chiaro": spesso, leggendoli, sembrano una serie di istruzioni in simil-inglese. Potremmo dire che, nei linguaggi ad alto livello, al programmatore è affidata la parte logica del programma, ma non la gestione della macchina (computer), che è invece resa implicita. Linguaggi ad alto livello sono `C`, `C++`, `Java`, `Python` e molti altri.

I linguaggi ad alto livello si dividono in "procedurali" ed in "orientati agli oggetti". In realtà non credo sia fondamentale, per questo corso, capire la differenza. Ti basta sapere che `C` è un linguaggio procedurale, mentre `C++` (che è la sua evoluzione) è un linguaggio orientato

agli oggetti. Noi studieremo quest'ultimo ma, sfortunatamente, per come viene affrontato nel corso di Informatica 1, ci limiteremo alla parte procedurale: impareremo poco di più di quanto si potrebbe studiando `C`. Solo nei corsi futuri (o nelle lezioni facoltative di questo corso) affronterai la programmazione ad oggetti.

1.2.3 Da C++ a linguaggio macchina

Ma come avviene il passaggio da `C++` (d'ora in poi parleremo solo di lui, non degli altri linguaggi), che come ho accennato ha elementi di inglese e tra poco ne vedrai un esempio, a linguaggio macchina?

Essenzialmente bisogna invocare un programma, detto compilatore, che si occupa di tradurre il `C++` in linguaggio comprensibile alla CPU. Nella sezione 1.4.4 approfondiremo meglio il discorso. Per ora ti basta sapere che il compilatore esiste e fa il lavoro sporco per noi.

Ma basta parlar di teoria, è arrivato il momento di vedere un po' di codice!

1.3 Il tuo primo programma

Ti riporto un esempio di programma e poi (promesso!) ti spiegherò tutto: cosa vogliono dire quelle strane parole, ma anche come compilare tutto ciò ed eseguirlo!

1.3.1 Il codice

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

int main(){//Qui inizia il nostro programma
        cout << "Hello World!" << endl;
        return 0; //e qui finisce
}
```

Esempio 1.1

Analizziamo riga per riga:

- **#include<iostream>** Questa è una direttiva per il preprocessore (così come lo è tutto ciò che inizia con #). Stiamo dicendo al preprocessore di includere nel nostro codice l'*header* (dall'inglese: intestazione) di una libreria, *iostream*. L'*header* è essenzialmente un file dentro cui è contenuto del codice dove sono dichiarati degli "strumenti"; includendolo nel nostro codice possiamo usare gli strumenti della libreria[1].

  Includere l'*header* di una libreria vuol dire che il preprocessore va a cercare il file dove è contenuto, copia tutto il codice e lo incolla al posto della riga "#include ...": ci risparmia un noiosissimo lavoro e ci evita di lavorare con file composti da migliaia di righe di codice, magari neanche scritte da noi.

[1] Nel capitolo 11, vedremo che per usare le librerie scritte da noi dovremo fare qualche passo in più.

Per le librerie standard del C++, includere l'*header* è tutto ciò che dobbiamo fare per poter usare la libreria per cui, d'ora in poi, con un abuso di linguaggio, dirò "includere la libreria" per rendere più chiaro il concetto.

Iostream vuol dire "in-out stream", ovvero canale di ingresso e uscita. Lì dentro sono definite alcune funzioni per comunicare con l'utente, come quelle per stampare a video parole e numeri o quelle per leggere da tastiera degli input.

Anche se ancora non abbiamo parlato di in e out, includi sempre nei tuoi programmi *iostream*.

- **using namespace std;** Te lo dico subito: questo codice ti rimarrà oscuro fino alla fine del corso. Spiegarti cosa significa vorrebbe dire immergersi in un argomento, il *namespace*, che non userai mai in questo corso. Quindi evitiamo. Solo una cosa: ricordati di metterlo in ogni file dopo le inclusioni delle librerie. È importante!

- **int main(){ }** Eccoci qua: questo è il programma vero e proprio! Main() è una funzione (nei prossimi capitoli impareremo di più sulle funzioni), è l'"entità" che viene lanciata dal sistema operativo (d'altronde, "main", significa "principale"), è qui che inizia il programma vero e proprio!

  Il programma è tutto ciò che è racchiuso tra le due parentesi graffe, può essere scritto su file differenti, può svilupparsi in migliaia di righe di codice ma, sul piano logico, tutto è racchiuso tra queste due parentesi.

  In poche parole, in ogni programma che scrivi deve esistere il *main*: tutto si svolge tra quelle parentesi graffe. Senza *main* non esiste il programma (e se per caso provi a compilare un file sorgente senza *main*, il compilatore ti avvisa della sua mancanza e non compila).

- // Il doppio slash è il simbolo in C++ per inserire un commento. Cos'è un commento? È qualcosa che viene completamente ignorato dal compilatore, può quindi essere inserito nel codice per commentare quello che abbiamo scritto senza intaccare la funzionalità del programma. Ad esempio, può trattarsi di frasi per spiegare i vari passaggi del nostro codice ed aumentarne la comprensibilità.

  I commenti in C++ sono tutto ciò che segue il doppio slash (rimanendo sulla stessa riga). Oppure tutto ciò che è contenuto tra "/*" e "*/": il primo simbolo dà inizio al commento, il secondo lo conclude; possono essere anche molte righe e non semplicemente una!

- **cout. . . endl;** Cout è l'abbreviazione di "console output": è il codice per utilizzare il dispositivo di output primario, lo schermo (tramite il terminale, che tra poco introdurremo). Quando vogliamo stampare a video qualcosa, dobbiamo chiamare la funzione *cout*. Questa parola va seguita da "<<" che, simbolicamente, rappresenta un flusso in uscita. Dopo possiamo mettere ciò che vogliamo stampare: una stringa di lettere racchiusa tra virgolette, come il nostro "Hello World!", o una variabile (vedi il prossimo capitolo).

  Alla fine dobbiamo "mandare a capo", e per farlo inseriamo di nuovo il simbolo "«" seguito dalla parola *endl*, che significa "end line": appunto, "vai a capo!".

- **return 0;** In tutte le chiacchiere che andremo a fare non ci occuperemo mai di sistemi operativi, di cosa fanno, di perché esistono e a cosa servono (puoi, però, trovare una brevissima introduzione all'uso di Linux in appendice A).

  In ogni caso, in questa riga stiamo comunicando con il sistema operativo. Essenzialmente quando il programma arriva a questo punto si conclude. Nel finire, però, restituisce qualcosa al sistema operativo (un numero): "0" sta per "va tutto bene". Se tutto non va bene, puoi restituire al sistema operativo un altro numero, ma è una finezza...

- **Il punto e virgola!** Puoi vedere il codice di un programma `C++` come una serie di istruzioni e operazioni. Ogni istruzione va conclusa, bisogna dire al compilatore: "l'istruzione finisce qua, quello che segue è un'altra cosa!". Mandare a capo non ha questo significato (mandare a capo rende più leggibile il codice per il programmatore, e se tu non mandassi a capo compilerebbe lo stesso, ma per carità, non lo fare!): è il punto e virgola che conclude un'istruzione! Alla fine di ogni unità logica è necessario un punto e virgola.

  Ci sono solo due situazioni del codice dove non ci vuole il punto e virgola, anche se viene conclusa un'unità logica: alla fine di una direttiva per il preprocessore e dopo la chiusura di una parentesi graffa (esistono delle eccezioni a quest'ultimo caso, le vedremo più avanti).

  Nel mio codice, come puoi vedere, dopo "`#include <iostream>`" non c'è un punto e virgola. Non c'è nemmeno dopo la chiusura della graffa del *main*.

Tutto chiaro?

1.3.2 Spazio di tab e leggibilità del codice

Mandare a capo non è l'unica cosa che rende il codice più leggibile: ce n'è una seconda che, purtroppo, all'inizio gli studenti sono davvero restii a fare, ma è importante!

Ciò che è racchiuso tra due parentesi graffe è detto *scope* (vedi il prossimo capitolo per una definizione migliore): ogni volta che apriamo una parentesi graffa stiamo scrivendo un piccolo "capitolo" del nostro codice (e alla fine di questo capitolo la parentesi va chiusa). È elegante, e rende il codice molto più agevole da leggere, dare uno spazio di tab in più (tasto della tastiera) per gli elementi all'interno delle graffe, uno spazio in più rispetto al "livello" delle graffe.

Ti faccio un esempio di codice, ma non cercare di capire ciò che succede, semplicemente mi interessa che ti chiarisca le questioni grafiche.

```
#include <iostream>
using namespace std;
int main(){ //apro graffa, tutto cio' che segue va preceduto da uno spazio
    di tab
        int j=10;
        for(int i=0; i<10; i++){//nuovo scope, ci vuole un ulteriore spazio
            di tab
                cout << i << endl;
                if(i==5){//nuovo scope, nuovo spazio!
                        cout << "Sono a meta'!" << endl;
                } //chiudo al ''livello'' dell'apertura
        }
        return 0;
```

```
}
```

Esempio 1.2

Questa tecnica aiuta a capire cosa è dentro cosa, con una sorta di matriosca di graffe. Aiuta a distinguere graficamente i vari "capitoli" del nostro codice. Dà un impatto grafico agli scopes[2].

Ti assicuro che rende il tuo codice molto più leggibile, per te e per gli altri. Inoltre, mancare di questa accortezza ti fa togliere punti all'esame.

La leggibilità del codice non è da sottovalutare, e un'altra cosa che aiuta a renderlo più leggibile sono i commenti. Commentare tanto (spiegando quello che fai) rende tutto più chiaro, non solo a chi deve leggere quello che hai scritto, ma anche a te stesso! Se dopo mesi riprendi in mano un tuo programma e lo devi modificare, ti posso assicurare che non ti ricorderai niente di quello che hai scritto: avere tanti bei commenti ti risparmierà ore a cercare di capire cosa ti era passato per la testa mesi prima. Se poi stai scrivendo codice con altri, commentare è davvero importantissimo: non tutti sono nella tua testa! (Neanche il prof all'esame...)

1.4 Il file sorgente, il terminal e il compilatore

Ho parlato di codice e te ne ho introdotto uno molto semplice, ma credo che ancora non ti sia chiaro dove va fisicamente scritto, come va compilato e, una volta fatto ciò, come si usa il programma.

1.4.1 Quale sistema operativo?

Ho promesso che non mi sarei dilungato a parlare di sistemi operativi, ma un piccolo accenno, puramente pratico, conviene farlo.

L'ambiente di programmazione ideale è il sistema operativo Linux. Linux è completamente free (open source) ed è altamente performante, tanto che si adatta perfettamente ai centri di calcolo, server di Google, supercomputer, ecc...

Se non disponi di Linux, te lo puoi procurare installandolo su un tuo computer, in una partizione dedicata o in virtual machine. Consiglio, tra le tante distribuzioni, Ubuntu o una sua derivata (Kubuntu o, se hai un computer non molto prestante, le più leggere Xubuntu o Lubuntu): in università molti hanno questa distribuzione e puoi trovare colleghi disposti ad aiutarti, sia nell'istallazione che nell'uso. Ubuntu è user-friendly, comodo e bello da usare, ma ovviamente non è l'unica distro possibile: ce ne sono tante (altre molto valide: Fedora, Manjaro, ...), scegli tu quella che più ti aggrada!

Non sei obbligato ad installarti Linux, puoi lavorare usando le macchine del laboratorio in remoto anche da Windows (ovviamente con una lentezza esasperante): se sei interessato consulta la sezione "materiale didattico" del sito del laboratorio, dove è spiegato tutto: http://www.l30-informatica.di.unimi.it/.

[2] In alcuni linguaggi, come il Python, non esistono le parentesi graffe che delimitano gli scopes, ma sono proprio gli spazi di tab a definirli. Prendere l'abitudine ad usare gli spazi di tab ti faciliterà la vita se vorrai imparare un linguaggio di quel tipo (e a Fisica, in alcuni corsi, il Python ti sarà richiesto).

Se sei un Mac user, tienti stretto il tuo computer: va benissimo per programmare, sempre su quel sito ci sono istruzioni su come puoi fare.

Io uso Linux e in queste note mi baserò, nei rarissimi casi dove è necessario differenziare, su questo sistema operativo. La programmazione in `C++` è uguale (al nostro livello) dappertutto: a seconda del sistema operativo cambia solo il programma con cui scrivere, come compilare e come far girare il programma stesso.

Nell'appendice A ho raccolto qualche nota sull'uso di Linux, i comandi base che ti servono per lavorare a casa o in remoto, collegato al laboratorio dell'università.

1.4.2 I file sorgente

Dove si scrive il codice? Sono dei semplici file, molto simili a dei file di testo. I file di codice `C++`, però, hanno un'estensione riservata: ".cpp" (sono estensioni del `C++` anche ".cxx" e ".C", scegli tu quale preferisci...). Per modificare il file sorgente hai bisogno di un editor di testo qualsiasi. I più user friendly su Linux sono Gedit e Kate.

Apri l'editor e salva il file con l'estensione richiesta, ad esempio "helloworld.cpp"; ora puoi scrivere il tuo codice!

Se hai installato Lubuntu, l'editor di testo è il più sfortunato Leafpad. Non è un gran che e ti consiglio di rimuoverlo e installare Gedit.
Per farlo apri il terminale (ctrl+alt+T, oppure cercalo tra i programmi) e scrivi:

```
sudo apt-get remove leafpad
```

Ti verrà chiesta la tua password e poi di confermare.

Una volta completata la disinstallazione, installa Gedit e, sempre nel terminale, scrivi:

```
sudo apt-get install gedit
```

Conferma tutto e installa. Bene, ora sei pronto!

Una volta scritto il tuo file sorgente e salvatolo con l'estensione giusta (prima salvalo, poi scrivi: così facendo Gedit capisce che è codice `C++` e ti colora le parole del linguaggio), è arrivato il momento di compilare!

Due parole per i Mac users: usate Xcode. Lì scrivete il vostro codice e salvate (sempre con estensione ".cpp"); non usate Xcode per compilare, ma il terminal: questo è per essere in un ambiente più simile a quello che userete all'esame!

1.4.3 Il terminale e il compilatore

Abbiamo detto che, una volta scritto il codice sorgente, bisogna compilarlo; il programma che effettua questa operazione è il compilatore. Per lanciare quest'ultimo abbiamo bisogno del terminal. Il terminale è una versione elementare di un computer, è come i computer degli anni '80: un cursore lampeggiante, uno sfondo inerte e la possibilità di interagire solo tramite la tastiera.

Inoltre, su Linux, il terminale è forse lo strumento più potente che dà accesso al controllo totale del nostro sistema operativo.

Per poter compilare il file dobbiamo "spostarci" con il terminal nella cartella del codice sorgente. Per capire di più sull'uso del terminale, prova a leggere l'appendice A.

Una volta posizionatici nella cartella del file sorgente, possiamo avviare il compilatore. Il codice da scrivere nel terminale è il seguente:

```
g++ nomefilesorgente.cpp -o nomeprogramma
```

Analizziamolo.

`g++` è il nome del compilatore, e scrivendo così lo stiamo "chiamando". Il compilatore vuole degli argomenti, per sapere cosa fare. Il primo è il nome del file sorgente (che deve terminare, come detto, con le estensioni citate); poi abbiamo il "-o", un'opzione che essenzialmente dice al compilatore di creare un file eseguibile con il nome che segue; infine il nome del file eseguibile: puoi mettere un'estensione (ad esempio in Informatica 1 usano spesso ".x"), o puoi evitarla.

Il compilatore viene avviato e inizia a controllare, prima di tutto, che il codice abbia senso. Se trova errori, non prova neanche a compilare, in compenso ti avvisa degli errori che ha trovato. Lentamente imparerai a capire gli strani messaggi che ti invia anche se, probabilmente, all'inizio lo odierai terribilmente.

Scherzi a parte, un piccolo consiglio su come interpretare i suoi messaggi: inizia sempre dal primo messaggio d'errore, mai dall'ultimo o dal centro. Perché? Magari il primo messaggio è l'unico vero errore, i successivi potrebbero esserci solo perché, a causa del pezzo di codice sbagliato, tutto il resto perde di senso. Ad esempio: se ti dimentichi un punto e virgola in alcuni luoghi del codice, il compilatore ti avvisa, ma siccome anche il codice che segue perde di senso, ti dà altri errori che in realtà non ci sono.

Per cui: parti sempre dal primo errore, correggi quello che hai scritto e poi prova a compilare di nuovo. Se ci sono altri errori, continua cosi: correggi un errore alla volta (il primo) e ricompila.

Se il compilatore ritiene che il codice sia sensato, lo compila e produce il file eseguibile. Come lo lanciamo? Se ci fai "doppio click" sopra non succede niente: lo devi avviare sempre dal terminal.

Per lanciare un file eseguibile devi usare questo codice:

```
./nomefilesegubile
```

Niente di complicato, quindi!

Che dire? Prova a scrivere il programma "Hello World!", compilalo e fallo partire; così vedi se ti è tutto chiaro.

1.4.4 La catena di compilazione: preprocessore, compilatore, assembler e linker.

Iniziamo con il primo "fuori programma": ciò che segue non è essenziale per superare l'esame ma, nella tua carriera da programmatore, è sicuramente utile conoscerlo.

Quando scriviamo "`g++ filesorgente.cpp -o programma`", quello che succede è una conversione tra codice `C++` a linguaggio macchina. Questo processo non è eseguito solo dal compilatore e, soprattutto, non avviene "tutto di un botto".

Supponiamo che "`hello.cpp`" contenga il nostro programma "Hello World!"; quando scriviamo "`g++ hello.cpp -o hello`", vengono chiamati in sequenza quattro personaggi: il preprocessore, il compilatore, l'assembler e il linker.

Il preprocessore è il signore più "stupido" della nostra catena. È in grado, semplicemente, di sostituire delle parole (dette "macro", che studieremo più avanti) con dei pezzi di codice. Non sa né leggere né scrivere: semplicemente sostituisce parole, ma non è in grado di valutare se queste sono corrette (e, infatti, abusare del preprocessore, è pericoloso[3]).

Il compilatore, invece, è il protagonista del processo; se non altro è colui con cui ti interfaccerai di più: conosce perfettamente il `C++` e, ogni volta che fai un errore di sintassi, ti avviserà con messaggi adeguati a caso. Le funzioni del compilatore sono due: controllare il codice alla ricerca di errori (e di comunicarti quali sono) e, se non ne trova, di tradurre il codice `C++` in *assembly* (il linguaggio a basso livello per antonomasia)[4].

Ora entra in gioco l'assembler: è colui che traduce il codice assembly nel vero e proprio linguaggio macchina (00100011100...). C'è poco altro da dire: l'assembler, per qualsiasi programmatore che non scriva in assembly, è un fantasma silenzioso ed efficiente che lavora senza farsi vedere.

Il codice prodotto dall'assembler non è "completo". Il nostro programma si articola in diverse parti, ognuna delle quali viene compilata e assemblata individualmente (per esempio le chiamate al sistema operativo non sono dentro il nostro codice). In queste piccole porzioni individuali di linguaggio macchina, vi sono delle "referenze" a pezzi esterni: vere e proprie parole da sostituire con codice eseguibile. Il linker si occupa di prendere i diversi mattoncini, metterli insieme e costruire il programma finale da essere eseguito.

Il linker agisce sempre ma, in molti casi (ad esempio quando abbiamo un solo file sorgente), risulta invisibile (però agisce: il nostro eseguibile viene sempre linkato con eseguibili del sistema operativo). Invece, lo incontreremo come parte attiva nel capitolo sulle librerie.

[3] Il C++, infatti, ha introdotto diverse nuove funzionalità che sostituiscono strutture del C precedentemente gestite solo dal preprocessore.

[4] Se sei curioso di vedere il codice assembly prodotto dal tuo programma puoi dire a g++ di fermarsi prima di chiamare l'assembler scrivendo "g++ file.cpp -S".

CAPITOLO 2

Tipi di dato standard e variabili

Cos'è un dato?

In sostanza, quando pensiamo ad un dato immaginiamo qualcosa che può essere "immagazzinato" da qualche parte, e a cui poi possiamo successivamente accedere.

In realtà, se spogliamo al massimo il concetto, un dato è un'informazione e, ovviamente, l'informazione può essere immagazzinata.

In matematica, ad esempio, un dato può essere un numero. È interessante il parallelo con la matematica, perché ci permette di aggiungere alcune caratteristiche importanti al concetto di dato che stiamo cercando di costruire.

Pensiamo ad un numero: la prima cosa che ci viene da fare è creare delle categorie, degli "insiemi". Ma perché lo facciamo? Essenzialmente gli elementi di questi insieme hanno delle caratteristiche, o anche delle proprietà, in comune.

Ad esempio: i numeri naturali, $\mathbb{N}$, sono i numeri interi che vanno da 0 ad infinito; inoltre su di essi sono definite alcune operazioni, come la somma e il prodotto, con le relative proprietà.

Sempre nel parallelo con la matematica, potremmo dire che "17" è un *dato* che appartiene ad $\mathbb{N}$; i numeri interi, invece, sono un *tipo di dato*.

Sono sicuro che avrai già capito la generalizzazione: un dato è un singolo elemento, mentre un tipo di dato è un insieme di elementi con proprietà comuni su cui, spesso, sono anche definite delle operazioni.

Altri notevoli tipi di dato, in matematica, sono i numeri relativi, razionali e reali. Vedremo che, in informatica, questi insiemi numerici (o meglio, la loro rappresentazione in macchina) sono i principali tipi di dato standard.

L'ultimo concetto da chiarire, prima di parlare dell'informatica, è quello di *variabile*.
Quando in matematica scriviamo $\forall x \in \mathbb{R}$ intendiamo che x è una *variabile* di tipo "numero reale".

Ecco il concetto di variabile: un tipo di dato è un insieme, una variabile è, invece, un elemento di questo insieme che può assumere valori diversi. Il valore che assume la variabile è il dato.

Abbiamo introdotto, appoggiandoci ad un mondo che ci è più amico (la matematica), tutto ciò di cui, tra poco, andremo a parlare.

2.1 I dati in informatica

In informatica i tipi di dato principali sono i seguenti: **int**, **long**, **float**, **double**, **char**, **bool**.

Non preoccuparti! Tra poco ti verranno spiegati tutti; prima, però, approfondiamo il concetto di variabile e di dato.

Abbiamo già visto la struttura elementare di un programma dove, però, non compariva alcuna variabile. È arrivato il momento di trovare un posto alle nostre variabili all'interno del programma. Ecco un esempio:

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

int main(){
        int x; //dichiarazione della variabile
        x=7; //operazione di assegnazione
        cout << "Valore della variabile x: " << x << endl;

        return 0;
}
```

Esempio 2.1

Dichiarazione di variabili Dunque, cosa significa **int** `x`? Stiamo dichiarando la variabile `x`: istruiamo il computer a riservare della memoria per la nostra `x`. Senza dichiarare le variabili è impossibile usarle! Nella riga successiva uguagliamo `x` a 7; se non avessimo dichiarato la variabile, per il computer questa riga non avrebbe senso: per lui `x` non esisterebbe!

Dovrebbe essere chiaro cosa significa dichiarare una variabile: vuol dire informare il computer della sua esistenza. Il che, tradotto, ha come risultato il far riservare della memoria per la nostra variabile per poterla successivamente usare nel programma.

Quando hai tante variabili dello stesso tipo da dichiarare, non devi per forza specificare il tipo di dato per ognuna: basta farlo una volta e separare i nomi delle variabili con una virgola. Ecco un esempio:

```
int main(){
        int a; //dichiaro le tre variabili una alla volta in righe diverse
        int b;
        int c;

        int d, e, f; //Questa scrittura e' piu' sintetica, ma comunque
            valida!
        return 0;
}
```

Esempio 2.2

Assegnazione Torniamo all'esempio precedente, non penso ci sia molto da dire sul significato di `x=7`... ma, per amor di precisione: stiamo assegnando un valore alla variabile. Lo possiamo fare sia esplicitamente, assegnando un numero, sia uguagliando una variabile ad un'altra (generalmente dello stesso tipo, o comunque compatibile, ne parleremo più avanti...). Esempio:

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

int main(){
        int a;
        int b;
        a=15;
        b=a;
        a=13;
        cout << a << endl << b << endl;
        return 0;
}
```

Esempio 2.3

Cosa succede in queste righe di codice?
Ho dichiarato le variabili `a` e `b`, poi ho assegnato ad `a` il valore 15, successivamente ho uguagliato `b` ad `a` (ora valgono entrambe 15), infine ho posto `a` uguale a 13.

Nota una cosa fondamentale: quando si uguagliano due variabili *si uguaglia il loro contenuto*, solo quello! Nel nostro caso, `b` non diventa `a`, semplicemente `b` viene riempito con lo stesso valore contenuto in `a`! Questo vuol dire che il successivo cambiamento di `a` non ha alcuna influenza su `b`.

Il precedente codice dovrebbe rendere chiaro che, una volta dichiarata una variabile, posso successivamente modificarla quante volte voglio.

Giusto per chiarezza, lo ripeto: il comando `cout << nome della variabile` va a stampare a video il *contenuto* della variabile.

Inizializzazione Prima di parlare dei tipi di dato specifici, voglio spiegarti una cosa importante che, sin da subito, devi imparare a fare sempre.

Cosa succede se noi dichiariamo una variabile, tipo "**int** a;" e poi la andiamo ad usare? Questa variabile non è stata inizializzata (non le abbiamo assegnato un valore) e non è detto che contenga 0: molto probabilmente è riempita con un valore casuale! Non inizializzare le variabili è estremamente pericoloso: se poi ci dimentichiamo di assegnare un valore, ci ritroveremo variabili contenenti dati assolutamente non voluti (ed usandole, potremmo incappare in risultati molto fastidiosi).

Quindi, ogni volta che dichiari una variabile, inizializzala subito, ovvero assegnale un valore, anche 0 se per ora non sai quanto deve valere. Puoi farlo sia nella dichiarazione che subito dopo, entrambe le seguenti scritture sono valide:

```
int main(){
        int a=0;
        int b;
        b=0;
        return 0;
```

```
}
```

Esempio 2.4

A volte ti sembrerà inutile, o magari ridondante, ma fallo sempre! Inizializzare le variabili può salvarti da brutte situazioni e, comunque, è indice di un buono stile di programmazione (per dirla in soldoni: è anche apprezzato all'esame!).

NOTA: Sebbene le variabili possano essere dichiarate in qualsiasi punto del codice, nell'esame di Informatica 1 si richiede che vengano dichiarate all'inizio di uno *scope* (o comunque all'inizio del *main*, più avanti vedremo in dettaglio cos'è uno scope). Probabilmente questo è per aiutarti a scrivere un codice più organizzato, obbligandoti a pianificare quello che ti serve ed evitando di farti "pasticciare" con il codice. Quando diventerai un programmatore più esperto, inizierai a trovare più comodo e intelligente dichiarare le variabili dove servono, ma, per ora, per imparare, attieniti alle richieste dei tuoi professori. Dichiara (e inizializza) tutte le variabili all'inizio del main!

2.2 Tipi di dato specifici

Ora, viste le caratteristiche comuni a tutte le variabili, possiamo passare ad analizzare i singoli tipi di dato. Rivediamo quali sono: **int**, **long**, **float**, **double**, **char**, **bool**. Li riporto in grassetto non a caso: se provi a scrivere queste parole su un file di codice `C++`, a seconda dell'editor, ti verranno evidenziate in grassetto e di un colore specifico. Questo perché sono *parole riservate*, ovvero possono essere utilizzate *solo* per indicare il relativo tipo di dato. Nel tuo codice non puoi usare la parola **int** per nessun altro scopo se non quello di indicare il tipo di dato **int**. Le parole riservate in `C++` sono tante. Tra queste rientrano tutte quelle che vedi scritte in grassetto negli esempi di codice finora presentati e quelle che vedrai in esempi futuri.

2.2.1 Int

Il nome di questo dato lascia poco spazio all'immaginazione: le variabili di tipo **int** possono contenere i numeri interi, positivi e negativi.

Abbiamo accennato al fatto che i dati possono essere immagazzinati: ogni variabile di uno specifico tipo occupa una certa quantità di memoria. Le variabili **int** occupano 4 byte.

Sorge spontanea un'osservazione: i numeri relativi vanno da meno infinito a più infinito, ma 4 byte sono tutt'altro che infiniti! Da questo discende che un **int** può contenere un numero relativo di grandezza finita.

Il più grande intero rappresentabile è 2147483647; il più piccolo è -2147483648. [1]

In informatica il "segno" di un numero occupa spazio, esattamente un bit (ricorda: un bit rappresenta uno 0 o un 1; in sistema binario, avere un bit in più a disposizione vuol dire poter immagazzinare un numero due volte più grande); possiamo liberare questo bit decidendo di

[1] Nei dati con *segno*, il primo bit è zero se il numero è positivo, uno se è negativo. Un int è composto da 4 byte, ovvero 32 bit. Se un bit è usato per il segno, ne rimangono 31: il numero più grande rappresentabile in base due è 2^{31}. I numeri positivi partono da 0, quindi arrivano a $2^{31} - 1$, quelli negativi da -1, quindi il massimo è proprio -2^{31}. Se fai i conti, sono esattamente i numeri riportati sopra.

trattare numeri solamente positivi. Come facciamo? Anteponiamo ad **int** la parola riservata "**unsigned**".

Il valore massimo rappresentabile da un **unsigned int** è: 4294967295.[2]

Bene, abbiamo parlato di cosa può immagazzinare un **int** e di quanta memoria occupa, rimane da parlare delle operazioni che si possono fare tra dati di tipo **int**.

Propongo un esempio di codice che poi analizzeremo:

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

int main(){
        int a, b, c;
        a=5;
        b=2;

        //SOMMA
        c=a+b;
        cout << "Somma: " << c << endl;

        //DIFFERENZA
        c=a-b;
        cout << "Differenza: " << c << endl;

        //MOLTIPLICAZIONE
        c=a*b;
        cout << "Moltiplicazione: " << c << endl;

        //DIVISIONE
        c=a/b;
        cout << "Divisione: " << c << endl;

        return 0;
}
```

Esempio 2.5

Per quanto riguarda le prime tre operazioni non c'è alcuna sorpresa, l'ultima è l'unica un po' delicata (in realtà se ci si pensa bene il risultato non dovrebbe affatto stupire); ecco l'output del programma:

```
Somma: 7
Differenza: 3
Moltiplicazione: 10
Divisione: 2
```

La divisione è la divisione intera (insomma, alla fin dei conti stiamo lavorando con numeri interi...)!

[2] Se aggiungiamo il bit del segno abbiamo 32 bit, dovendo partire da zero, il massimo numero rappresentabile in base due sarà $2^{32} - 1$, esattamente quello riportato nel testo.

5 diviso 2 dà come risultato 2 con il resto di 1; il semplice operatore "slash" istruisce il computer a fare la divisione e restituisce il risultato senza resto. Noi, però, potremmo essere interessati anche a quest'ultimo e il simbolo necessario per richiederlo è "%". Ecco un esempio di codice:

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

int main(){
        int a, b;
        a=5;
        b=2;
        cout << "5 diviso 2 = " << a/b << " con il resto di " << a%b << endl
            ;

        return 0;
}
```

Esempio 2.6

L'output sarà il seguente:

```
5 diviso 2 = 2 con il resto di 1
```

Dobbiamo stare attenti quando facciamo la divisione tra interi: a meno di forzare il meccanismo, non dobbiamo mai aspettarci il risultato decimale. Per quanto riguarda il "forzare il meccanismo" ne riparleremo nella prossima sezione.

Incremento di variabili Un'operazione utile è incrementare una variabile intera, ovvero sommarle 1 (quando parleremo dei cicli vedrai quanto spesso si userà!). Se `a` è la nostra variabile da incrementare, al posto di scrivere `a=a+1` possiamo usare la scrittura più compatta `a++`[3] oppure `++a`.

Cosa cambia? Vediamo il seguente codice:

```
#include <iostream>
using namespace std;
int main(){
        int a=1;
        int b=1;
        cout <<"a all'inizio: " << a++ << endl;
        cout <<"b all'inizio: "<< ++b << endl;

        cout << "ora a=b? " << a << "=" << b << "? " << endl;

        return 0;
}
```

Esempio 2.7

[3]Una curiosità: ti sei chiesto cosa significa il nome C++? La risposta sta proprio nel concetto di incremento: C++ vuol dire un passo avanti al C.

Ecco l'output:

```
a all'inizio: 1
b all'inizio: 2
ora a=b? 2=2?
```

Cosa succede? Cosa cambia nelle due scritture?
Scrivendo `a++` prima usiamo `a` e poi la incrementiamo; scrivendo `++b` per prima cosa viene incrementata, poi viene usata.

Ti propongo un codice: per esercizio prova a capire cosa succede senza compilarlo!

```
int main(){
        int a=13;
        int b=1;
        int c=0;

        c++;
        c=c+(b++);
        b=b-(++a);
        c=c+(c++);

        return 0;
}
```

Esempio 2.8

Quanto valgono a, b e c alla fine del codice?

Nota che quanto detto vale sia per il `++` che per il `--` (quindi per ridurre la variabile).

Un'ultima cosa che potresti trovare in alcuni codici e che ti spiego per non farti trovare impreparato. Scrivere `a=a+b` (quindi dire che `a` è uguale a se stesso a cui si aggiunge `b`) può essere abbreviato con la scrittura `a+=b` (lo stesso vale per divisione e moltiplicazione: `a/=b` e `a*=b`).

2.2.2 Long

Il tipo di dato **long** è essenzialmente un **int** più grande. Ha le stesse identiche caratteristiche di un **int** (quindi le operazioni che possiamo fare sono le stesse e possiamo immagazzinare sempre numeri interi), solo che può contenere numeri molto più grandi.

Una variabile di tipo **long** occupa 8 byte, il doppio di una variabile **int**.

Il più grande numero rappresentabile da una variabile **long** è 9223372036854775807; il più piccolo è -9223372036854775808.

Anche in questo caso possiamo utilizzare il bit riservato al segno anteponendo la parola **unsigned** (potendo, però, immagazzinare solo numeri positivi).
Il più grande intero rappresentabile da una variabile di tipo **unsigned long** è:
18446744073709551615.

Come vedi sono numeri decisamente più grandi rispetto agli **int**, ma purtroppo c'è un prezzo da pagare: viene usato il doppio della memoria.

Nella prossima sezione parleremo delle relazioni reciproche tra **int** e **long**: quando una variabile di un tipo può essere assegnata all'altra e viceversa.

2.2.3 Short

Il dato **short** è un **int** più corto. È composto da solo 2 byte. Il numero più grande rappresentabile è: 32767. Il più piccolo: -32768. Essendo fondamentalmente un **int**, possiamo anche qui dichiararlo senza segno potendo così rappresentare un numero due volte più grande.

L'unica utilità dei dati **short** è di risparmiare memoria quando sappiamo in anticipo che i numeri che immagazzineremo sono piccoli. È una vera e propria finezza che, se sei un programmatore inesperto e intenzionato semplicemente a passare l'esame, ti puoi anche dimenticare...

2.2.4 Float

Abbiamo parlato finora di numeri interi. Bene, ora è naturale introdurre i numeri decimali. "Float" significa "floating point", ovvero virgola mobile, e tra poco capiremo cosa vuol dire.

Dunque, se in teoria gli **int** dovrebbero rappresentare $\mathbb{Z}$ con i relativi limiti, i **float** dovrebbero rappresentare $\mathbb{R}$ con i relativi limiti. I quali limiti, purtroppo, non sono pochi.

In macchina i numeri **float** sono composti da due parti: l'esponente e la mantissa. La mantissa contiene le cifre vere e proprie del numero; l'esponente, invece, indica dove è posizionata la virgola (da qui, virgola mobile). L'esponente può essere molto grande: i **float**, oltre a rappresentare numeri decimali possono rappresentare numeri alla potenza di dieci notevole.

I limiti dei **float** sono principalmente due. Vediamo il primo.

Dal momento che occupano solo 4 byte, le cifre che il dato **float** può contenere sono limitate, così come è limitato l'esponente. In particolare, un **float** ha una precisione di 6 cifre decimali (può contenere fino a 6 cifre), mentre il numero può avere un esponente minimo pari a 10^{-37} e massimo pari a 10^{38}.

Ora, il fatto di contenere al massimo sei cifre ci porta subito ad introdurre l'errore di troncamento. Pensiamo a 1234567.89. Le cifre di questo numero sono ben più di sei, sono nove. In macchina questo numero viene rappresentato così: $1.23456 \cdot 10^6$. Cosa succede? Perdiamo le cifre successive alla sesta, semplicemente non ci stanno!

Un altro esempio notevole (dove questo errore può dare molto fastidio) è il seguente: 1.000000356. In macchina viene troncato: le ultime cifre eccedono le sei possibili. Se facciamo dei conti, ad esempio sottraendo a quel numero un altro simile, avremo risultati sbagliati!
Esempio:

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

int main(){
        float a= 1.000000356;
        float b= 1.000000321;
        float c=a-b;
        cout << c << endl;
```

```
        return 0;
}
```

Esempio 2.9

L'output di questo programma sarà un tristissimo "0".

Parliamo del secondo problema. Abbiamo accennato al fatto che in macchina i numeri sono rappresentati in base due. La conversione tra sistema decimale a sistema binario porta con sé dei problemi delicati.

Mi spiego meglio: essenzialmente alcuni numeri, che in sistema decimale sono finiti, in sistema binario non lo sono. Quando questi numeri vengono rappresentati in macchina, dal momento che può essere contenuto un numero finito di cifre binarie, vengono troncati. Questo avviene non a livello delle cifre decimali (che magari sono finite), ma delle cifre binarie! Quindi il numero originario, in macchina, non è il numero voluto, ma qualcosa che si avvicina. Di tutto ciò, a meno di conoscere il sistema binario, non ce ne accorgiamo facilmente! Facendo conti con questi numeri, all'apparenza innocui, possono succedere cose spiacevoli. Un esempio di numero che in base due non è finito è 0.1. Ti mostro un codice e il relativo output:

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

int main(){
        float a=0.3;
        float b=0.2;
        float c=0.1;
        float d=a-b-c;
        cout << d << endl;
        return 0;
}
```

Esempio 2.10

A video viene stampato il seguente numero: $7.45058e-09$, che è il modo del computer per dire $7.45058 \cdot 10^{-09}$, in ogni caso non zero!

Un piccolo appunto su come vanno scritti i **float**. Prima di tutto avrai notato che nei numeri decimali ho sempre usato il punto e mai la virgola, questo perché in informatica va utilizzato il punto e non la virgola (ricordatene quando assegni ad una variabile decimale un numero!). In secondo luogo possiamo assegnare alle variabili numeri esponenziali. Per farlo si scrive così: ad esempio $1.34 \cdot 10^7$ va assegnato come $1.34E07$, oppure $3.14 \cdot 10^{-12}$ diviene $3.14E-12$.

2.2.5 Double

Così come i **long** sono essenzialmente degli **int** più grandi, i **double** sono dei **float** più grandi e di maggiore precisione.

Un **double** occupa 8 byte e può rappresentare dati in virgola mobile. Ha una precisione di 15 cifre (ben maggiore delle 6 dei **float**!); il massimo esponente rappresentabile è 10^{308}, il minimo 10^{-307}.

Nel laboratorio di Informatica 1, se non ricordo male, l'uso dei **double** viene sconsigliato in favore dei **float** (stesso vale per **long** e **int**): insomma, se usi un **double** devi avere una buona ragione per farlo. Più avanti, invece, per la loro maggiore precisione, userai praticamente solo loro.

2.2.6 Char

Char significa "carattere". Un dato **char** può contenere un carattere della tastiera (in particolare sono utili le lettere dell'alfabeto). I caratteri che può contenere sono quelli ASCII: puoi cercare in internet quali sono. Per assegnare una lettera ad una variabile **char** ci sono due modi:

- Assegnando il codice del carattere ASCII, es:

```
char esempio=97;
```

Esempio 2.11

Dove 97 è il codice per la lettera "a".

- Assegnando il carattere che ci interessa, es:

```
char esempio='a';
```

Esempio 2.12

Nota che, con questo secondo metodo, assegniamo la lettera racchiusa tra apici (apici, non virgolette!): senza gli apici il compilatore darà errore.

2.2.7 Bool

Il dato **bool** può contenere una variabile booleana, una variabile logica: 1 o 0, vero o falso. Occupa solo un byte (e questo è il vantaggio rispetto ad usare un **int** a cui assegnare 0 o 1!) e ti accorgerai, più avanti, che è un dato estremamente utile, non tanto per immagazzinare dati esterni, quanto come strumento di controllo all'interno del tuo codice. Ne riparleremo nel capitolo 4!

Un dato **bool** può essere uguagliato ad 1, 0, **true** e **false** (nota: sono parole riservate). Ovviamente 1 equivale a **true** e 0 a **false**, ma le parole rendono il codice molto più chiaro e leggibile!

2.2.8 Riassumiamo...

Nota che le parole **unsigned**, **long**, **short** sono degli "specificatori". Essenzialmente, posti prima di un tipo di dato, ne modificano alcune caratteristiche. Un **long** è in realtà un **long int** (e, infatti, se li metti entrambi nel codice è corretto, ma per brevità **int** si omette). Stesso vale per un dato **short**. Perché ti dico ciò? Perché una di queste parole può essere usata anche

con i **double**: esiste il dato **long double**. Detto ciò, sappi che questo è assolutamente inutile saperlo, per l'esame, ma magari ho stimolato il tuo interesse...

Ti dico un'altra cosa non fondamentale per l'esame, ma che potrebbe esserti utile sapere. I **long** e i **double** occupano 8 byte, ovvero 64 bit. Non tutti i computer sono in grado di gestirli: i processori da 32 bit sanno gestire pacchetti di dati grandi, al massimo, 32 bit. I **long** e i **double** troppo grandi per processori di questo tipo! Su una macchina a 32 bit un **long** diventa un normale **int** e un **double** un **float**. Va detto, però, che oggi giorno quasi tutti i processori sono a 64 bit e questo problema ormai non si pone più..., ma meglio conoscere i limiti dei computer più datati, se mai dovrai programmare su uno di quelli! Per ulteriori informazioni sulla differenza tra CPU a 32 e 64 bit, puoi leggere i complementi al terzo capitolo.

Ti riporto, per riassumere, un output di un programma con tutte le caratteristiche dei dati numerici visti finora. Sappi che, in realtà, nell'esame di Informatica 1 ti ritroverai ad usare praticamente solo **int** e **float**; più avanti, però, ti saranno utili anche gli altri.

```
INT dim (byte): 4
Minimo: -2147483648
Massimo: 2147483647
Unsigned: 4294967295

SHORT dim (byte): 2
Minimo: -32768
Massimo: 32767
Unsigned: 65535

LONG dim (byte): 8
Minimo: -9223372036854775808
Massimo: 9223372036854775807
Unsigned: 18446744073709551615

FLOAT dim (byte): 4
Cifre di precisione: 6
Esponente minimo: -37
Esponente massimo: 38

DOUBLE dim (byte): 8
Cifre di precisione: 15
Esponente minimo: -307
Esponente massimo: 308

LONG DOUBLE dim (byte): 16
Cifre di precisione: 18
Esponente minimo: -4931
```

```
Esponente massimo: 4932
```

È implicito che, per l'esame, non devi assolutamente sapere a memoria i limiti di ogni tipo di dato. Però, è importante sapere che i limiti esistono ed avere un'idea dei relativi ordini di grandezza. Più avanti, nella scrittura di programmi di calcolo più complessi, la conoscenza dei limiti diverrà molto importante.

2.3 Operazioni tra tipi di dato diversi

Cosa succede se nel nostro programma ci troviamo a lavorare con variabili di tipo diverso? Se abbiamo **int**, **long**,**float** e **double**, come "interagiscono" tra di loro? Possiamo uguagliare variabili di natura differente?
Essenzialmente sì: sono tutti numeri e il compilatore effettua una conversione implicita. Il problema è che non sono numeri uguali e la conversione porta con sé delle modifiche spesso indesiderate.

2.3.1 Conversione tra tipi di dato diversi

Vediamo in breve cosa succede nei vari casi.

int ↔ long Se una variabile **int** viene assegnata ad una di tipo **long** non succede assolutamente nulla di male: il **long** è un **int** più grande, quindi lo può contenere senza alcun problema.

L'operazione inversa può presentare problemi. Se assegniamo ad un **int** un **long**, le cui dimensioni (in valore assoluto) sono inferiori alle dimensioni massime rappresentabile da un **int**, allora non abbiamo nessun problema: la conversione implicita va a buon fine. Se invece il numero in questione è più grande (sempre considerando il valore assoluto) dei limiti degli **int**, allora incorriamo nel problema dell'*overflow*: non riuscendo ad immagazzinare quel numero, si esce dai limiti della variabile e, all'interno della stessa, viene immagazzinato un numero più o meno casuale (spesso anche di segno opposto), insomma ci ritroviamo qualcosa che non c'entra nulla con quello che volevamo!

float ↔ double Valgono, con le dovute precisazioni, le stesse identiche considerazioni fatte nel paragrafo precedente (con la differenza che abbiamo a che fare con numeri con la virgola). Non credo ci sia molto altro da aggiungere.

int ↔ float Ben più interessante e complicata è la relazione tra interi e numeri in virgola mobile. Sono due i "paletti" da rispettare: i limiti di grandezza dei numeri rappresentabili (problema di overflow) e il fatto che uno ha la virgola e l'altro no.

Se assegniamo un intero ad un numero in virgola mobile, l'unico problema possibile è gli **int** e i **long** possono avere più cifre delle sei disponibili in un **float** (ricorda: il massimo numero **int** è 2147483647, ad esempio in questo caso verrà approssimato a 2.13748E09, perdendo alcune cifre finali). Per il resto, se uguagliamo numeri piccoli non incappiamo in alcun problema, la conversione è fattibile e innocua.

Il contrario lo è un po meno: se assegniamo ad un **int** un **float**, la parte decimale viene completamente troncata! Nota: non approssimata, troncata! Ad esempio 11.89 diviene 11; 0.782 diviene 0; 1.1 diviene 1 e così via.

Non è l'unico problema possibile. Il secondo è dovuto al fatto che le cifre più l'esponente potrebbero essere, in valore assoluto, decisamente maggiori del più grande **int**: se proviamo a convertire in **int** 10^{13} andremo sicuramente in *oveflow*.

Tutte queste considerazioni, a meno delle diverse grandezze, valgono per le relazioni tra **long**, **double**, ecc...

2.3.2 Operazioni numeriche tra interi e decimali

Cosa succede quando facciamo operazioni matematiche usando sia **float** che **int**? Quale tipo di operazione viene effettuata?

Questo problema, oltre ad essere di notevole importanza nella scrittura di programmi di calcolo, è uno dei punti fondamentali dell'esame scritto di informatica, quindi presta attenzione!

Premetto che il caso più importante è quello della divisione, in quanto, come abbiamo visto, la divisione tra **int** è la divisione intera, un'operazione decisamente diversa dalla divisione tra numeri reali.

Per qualsiasi operazione tu vada a fare nel tuo codice, starai eguagliando una variabile (posta a sinistra dell'uguale) a qualche operazione tra variabili (poste a destra dell'uguale). Voglio che ti entri bene in testa un concetto: il computer per prima cosa calcola quello che si trova a destra dell'uguale e, mentre lavora, non gli interessa assolutamente nulla di cosa ci sia a sinistra dell'uguale (quindi il tipo di variabile posta a sinistra). Lui fa i suoi conti, poi, una volta finiti, assegna alla variabile di sinistra il risultato. Ma ricorda: ciò avviene solo dopo aver fatto i conti; la variabile di sinistra non influenza quello che accade a destra!

Ci sono due problemi: i conti da fare e il fatto che il risultato deve essere convertito alla variabile di sinistra, con tutte le problematiche analizzate prima.

Somma, sottrazione e moltiplicazione Per quanto riguarda queste tre operazioni, il concetto è molto semplice: a destra dell'uguale avviene esattamente quello che ci aspettiamo. Se facciamo un'operazione tra interi avremo come risultato un intero; se facciamo operazioni tra interi e decimali avremo come risultato un decimale; un'operazione tra decimali avrà come risultato un decimale.

Bene, l'unico problema, allora, nasce da quello che vi è a sinistra dell'uguale: se abbiamo un **int** e il risultato sarà decimale avremo i problemi analizzati precedentemente, così come negli altri casi.

Insomma, niente di particolarmente complicato da capire.

Divisione La divisione presenta una difficoltà in più: quello che succede a destra dell'uguale non è così banale.

Ricordati che se la divisione è tra **int** (o **long** o **short**) viene effettuata la divisione intera, e se questo non è quello che desideri, puoi forzare il computer a fare la divisione decimale.

Se, invece, anche solo il numeratore o solo il denominatore sono decimali, allora viene fatta la divisione decimale.

Vediamo degli esempi:

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

int main(){
        int a=3, b=5, c=0;
        float d=7., e=9., h=0.;

        h=b/a; //a destra interi: divisione intera, poi convertito ad un
            float (ma non influenza il tipo di divisione)
        h=(float)b/a; //forziamo a fare la divisione decimale
        h=b/(float)a; //stesso di prima
        h=7/2; //a destra costanti intere, divisione intera
        h=7./2; //7. e' una costante float, divisione decimale
        h=7/2.; // stesso di prima
        c=7./2; //viene fatta la divisione decimale e poi il risultato
            troncato nella conversione ad int
        c=d/e; //divisione decimale, poi troncato (risultato diviene zero)
        h=e/d; //tutti float, nessun problema!

        return 0;
}
```

Esempio 2.13

Se non sei convinto scrivi un programma che stampi i risultati! Magari prova anche a fare un po' di esperimenti per capire meglio come funziona!

Hai visto come forzare la divisione ad essere decimale: mettere "(**float**)" converte la variabile a **float**; puoi porre anche "(**double**)" davanti, l'idea è la stessa (ma usi 8 byte al posto di 4, guadagnando in precisione).

Per quanto riguarda le costanti numeriche, se non sono presenti punti o parti decimali, il computer li interpreta come interi e si comporta di conseguenza. Inserire un punto alla fine li trasforma in **float**.

Un paio di esami risolti... Come ho accennato, il problema della divisione viene spesso affrontato negli scritti. Proviamo a vedere un paio di esercizi d'esame insieme.

- Il testo del primo esercizio è:

```
Sia int matr un intero che contiene il vostro numero di matricola ed inoltre
    float matr1, matr2, matr3;
se
    matr1 =(int)(matr/(matr+1.));
    matr2 = (matr+1)/matr;
    matr3=((matr+1)/100)/(matr/100);
scrivere il valore di:
```

```
matr1_____________matr2_____________matr3_____________
```

Dunque, poniamo che il mio numero di matricola sia 543987 (completamente inventato, ma di lunghezza giusta).
Vediamo cosa succede a `matr1`. Non dobbiamo farci confondere da quell'**int**, che è posto prima della parentesi tonda: vuol dire che converte in intero il risultato che esce dalle parentesi. Nella parentesi abbiamo a numeratore un intero, mentre a denominatore un numero decimale (somma tra un **int** e un **float**, che abbiamo visto dare come risultato un **float**). Verrà fatta la divisione decimale e il risultato (che non ci interessa con precisione) sarà comunque minore di 1, uno 0.qualcosa che, convertito in **int**, diviene 0.

Passiamo a `matr2`. A numeratore e denominatore abbiamo interi, quindi avremo una divisione intera. È evidente che il risultato sarà 1; a sinistra dell'uguale abbiamo un **float** ma, comunque, 1, convertito rimarrà immutato.

Sotto con `matr3`! Abbiamo tutti interi, tutte divisioni intere (e di nuovo, anche se a sinistra dell'uguale c'è un **float** per ora ci importa davvero poco!). Senza analizzare tutti i passaggi possiamo concludere che il risultato sarà comunque 1, che convertito a **float** rimane 1.

- Il testo del secondo esercizio è:

```
Sia int matr un intero che contiene il vostro numero di matricola
ed inoltre:
    int a,b;
    float x,y;
dopo:
    a=matr/1000;
    b=matr/0.1E4;
    x=matr/1000;
    y=matr/0.1E4;
avremo:
    a=_____ b=_____ x=_____ y=_____
```

Di nuovo, dobbiamo curarci solo di quello che è alla destra dell'uguale: prima viene effettuata l'operazione e solo dopo il risultato viene convertito al tipo di dato posto a sinistra! Ipotizzando che il nostro numero di matricola sia ancora 543987, la situazione tra **a** e **b**, e tra **x** e **y** è pefettamente simmetrica: nel primo caso (`matr/1000`) il risultato è 543 (le ultime tre cifre vengono troncate: divisione tra interi); nel secondo caso (`matr/0.1E4`) il risultato è 543.987 (divisione tra un intero e un float). A questo punto avviene la conversione al tipo di dato a sinistra dell'uguale, e avremo quindi i seguenti risultati: `a=543; b=543; x=543; y=543.987`.

- Un esercizio un po' più fine è il seguente:

Sia `expr` un'espressione che convolge `int` e `float`. Fare dei semplici esempi che giustifichino le seguenti affermazioni:
a) non vale la proprietà associativa;
b) non vale la proprietà commutativa.

Quello che il tema d'esame ci sta chiedendo è di trovare degli esempi in cui non valgono, effettuando operazioni tra *int* e *float*, la proprietà associativa e la proprietà commutativa [4]

Per quanto riguarda la proprietà associativa è abbastanza semplice: basta trovare casi patologici in cui, se effetuiamo prima un'operazione o l'altra, andiamo o non andiamo in *overflow*. Ad esempio, nel caso della moltiplicazione, possiamo prendere due numeri molto grandi, `a` e `b`, e uno molto piccolo, chiamiamolo `x`. Se calcoliamo $(x \cdot a) \cdot b$, normalmente non avremo problemi; $x \cdot (a \cdot b)$, invece, potrebbe andare in *overflow* e i due risultati sarebbero diversi.

Se consideriamo l'addizione, possiamo sostituire al numero molto piccolo un numero molto grande ma negativo, chiamiamolo `c`: $a + (b + c)$ non darà problemi, mentre $(a + b) + c$ rischia l'*overflow* nella parentesi.

Per quanto riguarda la proprietà commutativa, non ho trovato esempi con solo due variabili per cui essa non vale. Se invece ne consideriamo tre, il discorso è simile a prima: se `a` e `b` sono molto grandi e `x` molto piccolo, $a \cdot x \cdot b$ non dovrebbe dare problemi (le operazioni vengono calcolate da sinistra a destra), mentre $a \cdot b \cdot x$ rischia di essere patologico, e tanti saluti alla proprietà commutativa.

Per l'addizione il discorso è analogo.

Questi temi d'esame dovrebbero farti riflettere non solo sul fatto che il computer non è perfetto, ma –in particolare l'ultimo– anche sul fatto che il programmatore può evitare situazioni patologiche se conosce in anticipo, ad esempio, gli ordini di grandezza delle variabili (o il risultato che gli interessa quando ci sono in gioco operazioni tra float ed int). A proposito di ciò, ti invito a considerare la sezione 2.4 dove ho voluto presentare due situazioni più applicative.

2.4 Il discorso della precisione: l'importanza per un Fisico

Dovresti aver capito che i computer non sono così "precisi" come ci si aspetta. La rappresentazione in macchina è finita, e questo causa numerosi problemi. I casi più patologici sono dati dai numeri in virgola mobile.

Hai già visto che sommare e sottrarre **float** può dare risultati inattesi. La moltiplicazione e la divisione sono ancora più problematici. Banalmente, se dividiamo due numeri il cui risultato è inferiore alla precisione della macchina il risultato sarà errato.

[4] Associativa: $(a + b) + c = a + (b + c)$ oppure, per la moltiplicazione, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. Commutativa: $a + b = b + a$ per l'addizione, oppure, $a \cdot b = b \cdot a$ per la moltiplicazione. Ah! non ho dubbi che tu sappia benissimo cosa sono, volevo solo fare il punto della situazione.

Ma perché farsi questi problemi? A questa domanda è facile rispondere tenendo presente *perché* un Fisico dovrebbe utilizzare la programmazione.

Sia uno sperimentale che un teorico si ritrovano ad affrontare situazioni per cui un computer è fondamentale. Per i primi, quasi tutti gli esperimenti producono una mole di dati ingestibile con carta e penna, così come da programmi come Excel. Una singola presa dati del CERN occupa diversi terabyte: un esperimento di rivelazione della radiazione nucleare deve registrare milioni di eventi, e così via. I dati che escono dall'esperimento, di per sé, sono privi di significato: bisogna elaborarli. Per esempio, un rivelatore di radiazione registra l'energia della radiazione incidente tramite dei canali discreti: dovremo convertire questi canali, con dei calcoli, all'energia della radiazione. Inoltre, per dar senso alla nostra misura, dovremo rimuovere la radiazione di fondo ambientale, effettuare della statistica per ricavare grandezze veramente significative ed utili, ecc... Sapere che i calcoli che facciamo potrebbero essere "patologici" e che il risultato potrebbe essere sbagliato, è importante (siamo Fisici: ci interessa il dato finale, che non deve essere errato). Dobbiamo sapere cosa possiamo fare, cosa potrebbe essere fonte di errore e dove andare a correggere. Una cifra diversa, anche di poco, in un esperimento può portare a conclusioni fisiche sbagliate...

Un teorico, viceversa, è spesso interessato a simulare situazioni che non possono essere analizzate con carta e penna: il comportamento di migliaia di particelle e la reciproca interazione, il comportamento degli ioni di un cristallo, ecc... è palese che un computer può risultare molto comodo.

A qualsiasi Fisico può capitare di affrontare problemi matematici la cui soluzione analitica è troppo lunga e complessa o banalmente non esiste. Risolvere un sistema lineare di diecimila equazioni è fattibile a priori, ma auguri! Forse non hai ancora affrontato integrali ed equazioni differenziali, ma nel corso di Analisi 2 vedrai che questi oggetti matematici non sempre hanno soluzione analitica, anzi: nella maggior parte dei casi non ce l'hanno. Esistono però tecniche numeriche per risolverli[5], e i computer si prestano in maniera eccelsa. Di nuovo, però, è importante tenere conto della precisione limitata del computer.

Vediamo un paio di esempi:

Calcolo delle derivate Dai dati del nostro esperimento, vogliamo estrapolare una funzione (leggi: ho i punti su un grafico, voglio trovare una funzione che ci passa) la cui derivata è la grandezza fisica che voglio studiare (esempio banale: il mio rivelatore è in grado di trovare la posizione di una particella, ma a me interessa la funzione velocità). Misurati i punti del mio esperimento, con tecniche numeriche estrapolo la funzione che meglio li approssima e ne calcolo la derivata in diversi punti con il computer.

Con ingenuità, so dal corso di Analisi che la derivata di una funzione in un punto è definita come:

$$f'(x_0) = \lim_{h\to 0} \frac{f(x_0+h) - f(x_0)}{h}$$

Ma pensiamoci bene: sopra ho qualcosa di piccolo, e sotto qualcosa che tende a zero. "Piccolo" e "tende a zero" sono quanto di più patologico esista nell'aritmetica di un computer. È implicito

[5] Se sai cos'è l'integrale di Riemann, ti sarà noto che rappresenta l'area sotto una funzione. Se l'integrale non può essere calcolato esplicitamente, una tecnica numerica può essere quella di lanciare casualmente punti in una figura geometrica di area nota che racchiude la funzione, quindi calcolare il rapporto tra i punti caduti sotto la funzione e quelli sopra.

che, se prendo un "h" molto piccolo, il risultato che troveremo sarà sicuramente sbagliato, per "colpa" del computer; se invece è troppo "grande", sarà sbagliato per "colpa" della matematica[6].

Un prototipo di simulazione Vogliamo studiare come varia l'energia totale e lo stato di eccitazione delle molecole di un cristallo quando quest'ultimo viene colpito da una radiazione ionizzante, il tutto al variare di parametri quali "energia della radiazione" e "punto del cristallo che viene colpito". Se consideriamo che il nostro cristallo, per quanto piccolo, avrà milioni, se non miliardi, di molecole, è chiaro che il problema non può essere risolto con carta e penna.

Iniziamo a considerare un problema monodimensionale: il cristallo è rappresentato da una retta e le molecole sono particelle equidistanziate tra loro in grado di scambiarsi energia quando eccitate.

Nel nostro primo test, ipotizziamo che ci siano centomila molecole su una retta lunga 10000 e distanti 0.1 (sì, numeri adimensionali). Per riempire la retta usiamo il seguente codice:

```
....
int a=0;
while(a<=10000){ //"while" serve per eseguire il codice tra le graffe finche
    ' a e' minore di 10000
        a+=0.1;
        //riempi punto della retta
}
....
```

Esempio 2.14

Dove "a" rappresenta la posizione della retta. Il *ciclo* **while** esegue il codice tra le graffe finché la condizione tra le tonde è verificata (ovvero finché a è minore di 10000), nel capitolo 4 affronteremo il discorso in maniera sistematica. Sicuri di noi, lanciamo la simulazione e troviamo un risultato che non ci aspettavamo.

Dopo aver perso tempo a capire che è successo, scopriamo che nel riempimento qualcosa è andato storto. Prova ad eseguire il seguente codice:

```
#include <iostream>
int main(){
        float a=0; //posizione
        int n=0; //numero di particelle
        while(a<=10000){
                a+=0.1;
                n++;
                std::cout << "Particella numero: " << n << "\t Posizione: "
                    << a << std::endl;
        }
        std::cout <<"Numero totale di particelle: " <<n<< std::endl;
}
```

Esempio 2.15

[6]Le derivate, tendenzialmente, è meglio calcolarle a mano e non con computer. Piuttosto, conviene estrapolare (tramite il calcolatore) una funzione che meglio approssima i dati e, quindi, derivarla con carta e penna.

Ti accorgerai che il numero totale di particelle è 100015 e non le centomila che ci aspettavamo. Il tutto perché 0.1, come hai già visto, non ha una rappresentazione finita in macchina: sommare ripetutamente 0.1 non porta al risultato desiderato, perché in effetti non stiamo sommando *esattamente* 0.1 (già con a<=100 il risultato non è quello atteso...).

Ti vengono in mente possibili soluzioni? Ad esempio, avremmo potuto usare, per riempire la retta, un ciclo **for** contando esattamente il numero di particelle: in questo modo, però, avremmo riempito un segmento un po' più corto. Avere sia la lunghezza esatta che il numero di particelle desiderate, con questi numeri, è impossibile.

Un'altra soluzione potrebbe essere di scalare il problema su grandezze meno patologiche: ad esempio, moltiplicando la dimensione del reticolo per 10 e così anche lo step (che diventerebbe, quindi, 1). Una volta distribuiti i punti, scalare "indietro" tutto. Questa soluzione presenta due problemi: la prima è che dilatando le dimensioni potremmo finire in *overflow*; la seconda è che quando rimpiccioliamo la retta riempita, potremmo andare incontro a nuovi errori di precisione.

Ovviamente, questi erano casi banali. Ma, per quanto semplici, sono esempi di situazioni patologiche che possono falsare i risultati. Stai attento, quindi, quando fai conti in `C++`: cerca sempre di capire dove possono nascere errori, ed evitali quando puoi; il discorso della precisione non serve solo a passare l'esame!

2.5 Visibilità delle variabili

Abbiamo già introdotto il concetto di *scope* ma, per chiarezza, lo approfondisco. Uno *scope* è essenzialmente l'area in cui "vive" una variabile. Se una variabile è locale (dopo spiego) allora uno scope è ciò che è racchiuso tra due parentesi graffe. Per ora hai presente solo le parentesi graffe del *main* ma, in un programma, possono esservene molte altre: quelle dei cicli, delle strutture di selezione, di funzioni.

Una variabile "nasce" dove viene dichiarata, e "muore" alla fine dello *scope*. Che implicazioni ha tutto ciò? Una variabile può essere vista ed usata solo all'interno dello *scope* in cui è stata dichiarata; finito lo *scope* la variabile "muore" e non può più essere usata né vista. Se provi ad usare una variabile fuori dal suo *scope*, il compilatore ti avviserà dell'errore.

Gli *scope* funzionano un po' come matriosche: possono esserci *scope* uno dentro l'altro. Una variabile dichiarata in uno *scope* più esterno viene vista in tutti gli *scope* contenuti in esso. Il viceversa, invece, non è vero.

Una variabile dichiarata all'interno di parentesi graffe si dice locale, e il motivo credo sia chiaro: viene vista solo localmente. Vi è un secondo tipo di variabile: quella globale. Una variabile globale viene dichiarata, per esempio, fuori dal *main*, nell'intestazione del file. Questo tipo di variabile viene vista dappertutto, non solo in quel file, ma anche in tutti quelli che lo includono (capirai quando parleremo di librerie). Proprio per il fatto che sono visibili e modificabili in qualsiasi parte del codice, le variabili globali sono molto pericolose e, personalmente, ne sconsiglio fortemente l'uso. Se tu dichiari un "**int** a" globale, questo è visibile in ogni file che include il file dove è dichiarato. Se altrove dichiari un altro "**int** a", fai, come si suol dire, un bel casino. Il `C++` ha sviluppato un potentissimo strumento per proteggere i nomi delle variabili: il *namespace*. Purtroppo questo argomento non rientra in quelli richiesti

in Informatica 1, per cui non approfondiremo.

Ora ti riporterò un esempio di codice che dovrebbe chiarire l'ambito di visibilità delle variabili, ma non preoccuparti di capire cosa succede: userò costrutti che non abbiamo ancora analizzato, l'unica cosa che mi interessa è che tu segua il ragionamento delle variabili.

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

int global; //variabile globale, puo' essere vista ovunque!

int main(){
        int i=0; //puo' essere vista in tutto cio' che e' contenuto nello
            scope del main
        global=15; //viene vista dappertutto, posso modificarla
        while(i<global){ //inizio scope
                int local=global; //variabile locale
                i++; //siamo in uno scope contenuto in quello del main, i
                    viene vista!
                local--;
        } //fine scope, fuori da qui local non esiste piu'
        cout << local << endl; //sono fuori dallo scope di "local", l non
            esiste piu', non posso farlo!

        return 0;
}
```

Esempio 2.16

2.6 Macro e costanti

Una macro è una qualsiasi riga di codice che inizia col carattere "#" (asterisco). Ad esempio il "#include <iostream>" è una macro.

Le macro non vengono analizzate dal compilatore, bensì dal preprocessore. Prima che inizi la compilazione, il preprocessore si occupa di sostituire le macro con ciò che è opportuno. Per esempio, dove trova "#include <iostream>" il preprocessore prende la libreria iostream e fisicamente la incolla al posto di quella riga di codice. In poche parole: questo meccanismo è ideato per evitare di avere migliaia di righe di codice quando quelle scritte da noi magari sono poco più di una decina. Il preprocessore, poi, fà il lavoro sporco per noi, senza che ce ne accorgiamo. Una volta incollato tutto il codice, il file formatosi viene compilato dal compilatore.

C'è un altro tipo di macro, tipica del `C`, ovvero quella usata per definire delle costanti. Esempio:

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

#define PI 3.14159
int main(){
        float p=PI;
        cout << p << endl;
```

```
        return 0;
}
```

Esempio 2.17

Abbiamo definito la costante PI: per farlo si scrive, come puoi vedere, "#define" seguito dal nome della costante e, dopo uno spazio e nessun "=", il valore dalla costante, senza punto e virgola alla fine (ti ricordi il paragrafo sul punto e virgola? Questa è una direttiva preprocessore!).

Se scrivi il nome di quella macro nel codice, il preprocessore andrà a sostituirla con il valore che le hai assegnato.

In realtà, per come la vedo io (e molti altri), è meglio evitare l'uso delle macro (ad esclusione degli "#include"): il preprocessore non effettua tutti i controlli che il compilatore, invece, assicura. Lui semplicemente sostituisce la parola col valore, ma non controlla che le variabili siano compatibili o che quello che stiamo scrivendo abbia senso. È uno strumento del C, infatti, nel C++ è stato introdotto un nuovo tipo di costante gestita, invece, dal compilatore.

Se vuoi avere dei dati costanti nel tuo codice, dichiara una variabile e precedi il tipo di dato con la parola riservata **const**.

L'esempio di prima diventa:

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

const float PI=3.14159;
int main(){
        float p=PI;
        cout << p << endl;
        return 0;
}
```

Esempio 2.18

Ora PI è una variabile globale gestita dal compilatore (e quindi a minor rischio di errore), ma è costante. Se provi a modificarla nel codice, il compilatore ti dà errore. Può sembrare una cosa inutile ma, se nel tuo programma inserisci costanti fisiche o matematiche, renderle costanti anche nel computer è una sicurezza in più.

La differenza tra macro e variabili costanti è, di nuovo, una finezza e, se non ricordo male, probabilmente nel corso di Informatica 1 nemmeno si sottolinea. Il concetto è che sei liberissimo di usare le macro (nello stesso corso credo si usino spesso), ma sappi che l'idea del C++ è di evitare di usarle; se la cosa ti interessa approfondisci tu l'argomento!

2.7 *cout* e *cin*

Cout e *cin* fanno parte della classe degli stream di dati, che sono canali di input ed output: ti ricordi la schematizzazione del computer? Abbiamo detto che esistono RAM, CPU e dispositivi di input e output; ecco: *cin* e *cout* fanno parte degli strumenti del C++ per far comunicare CPU e RAM con i dispositivi di input ed output. Questi strumenti saranno analizzati diffusamente nel capitolo 7, però ci saranno utile nel corso degli esempi di codice dei capitoli precedenti. Quindi, ti accenno una rapida spiegazione molto pragmatica.

Cout sta per "console output", in cui la console è il terminal. Insomma, output su terminale: serve per stampare a video stringhe e variabili. Prende una stringa o una variabile e stampa a video il loro contenuto.

Cin, viceversa, sta per "console input", quindi l'input da console (che avviene tramite tastiera). *Cin* aspetta che l'utente inserisca da tastiera un valore, premendo invio lo passiamo a cin, il quale lo passa ad una variabile.

Per poter usare sia *cin* che *cout* è necessario includere la libreria "iostream". Al posto di grandi discorsi, ecco un esempio concreto:

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

int main(){
        int a=0;
        cout << "Benvenuto in questo programma!" << endl; //sto stampando
            una stringa: lettere racchiuse tra virgolette. Endl manda a capo
        cout << "Inserisci un numero intero: "; //non vado a capo
        cin >> a; //cin aspetta che tu inserisca un numero da tastiera. Non
            andando a capo aspetta sulla stessa riga. Premendo invio andra' a
             capo

        cout << "Hai inserito: " << a << endl;

        cout << "Alcuni caratteri speciali:\tbackslash t e' uno spazio di
            tab\nbackslash n manda a capo" << endl;
        cout << "Per stampare piu' variabili e stringhe separali con <<:" <<
             endl;
        cout << 1 << "\t" << 2 << "\t" << 3 << "\n";
        return 0;
}
```

Esempio 2.19

Ecco l'output:

```
Benvenuto in questo programma!
Inserisci un numero intero: 17
Hai inserito: 17
Alcuni caratteri speciali:      backslash t è uno spazio di tab
backslash n manda a capo
Per stampare più variabili e stringhe separali con <<:
1       2       3
```

CAPITOLO 3

Rappresentazione in memoria

Premessa: questo capitolo ti sembrerà estremamente astratto e fine a se stesso. Non è così, o meglio: è sì astratto, ma è molto utile per poter capire due argomenti decisamente pratici che affronteremo più avanti: gli *array* e i *puntatori* (oltre al fatto che alcune nozioni di questo capitolo sono oggetto dello scritto). In ogni caso, tranquillo: sarà un capitolo breve.

Qualsiasi programma, per funzionare, deve essere caricato nella memoria. Ricordo che per *memoria* intendiamo la RAM, non il disco fisso. Quando avviamo un programma, il sistema operativo va a leggere il codice eseguibile che è immagazzinato nel disco fisso, pensa quest'ultimo come un deposito dove le informazioni possono essere solo stanziate, ma non "usate" in maniera attiva; da lì prende il pacchetto di dati che rappresenta il programma e lo carica sulla RAM, e così il programma diventa "vivo", cioè può iniziare a funzionare.

Piccolo inciso (giusto per curiosità, non sono informazioni necessarie per l'esame): la differenza a livello pratico tra hard disk e RAM è che il primo è molto più lento del secondo. Gli hard disk classici sono composti da dischi di materiale magnetico che girano con una testina che legge i dati. La RAM, invece, è una memoria composta da circuiti a stato solido: è estremamente veloce e, mentre la testina dell'hard disk ha difficoltà a muoversi casualmente da un punto all'altro del disco, i dati sulla RAM possono essere letti in maniera sparsa in modo estremamente rapido: non a caso RAM, significa "random access memory", memoria ad accesso casuale. I nuovi dischi a stato solido, gli SSD, sono più veloci rispetto a quelli "classici", ma comunque lentissimi a confronto con la RAM.

Questa non è l'unica differenza. L'altra sta nel fatto che l'hard disk, magnetico o a stato solido, non "perde la memoria": i dati hanno vita virtualmente infinita. Anche quando viene a mancare la corrente di alimentazione, i dati "sopravvivono". Lo stesso non vale per la RAM: quando viene meno la corrente (si spegne il computer), si perdono tutti i dati contenuti. Infine, gli hard disk possono essere centinaia di volte più capienti della RAM.

Insomma, tutto ciò per dire che il processore ha accesso diretto alla RAM e non all'hard disk. Possiamo pensare il computer come un cervello: il processore è la parte pensante; la

RAM è la memoria a breve termine dove vengono caricati i pensieri e i ragionamenti che la parte pensante elabora; l'hard disk è la memoria a lungo termine a cui la parte pensante fa più fatica ad accedere e su cui non può "ragionare", e per farlo deve spostare i pensieri sulla memoria a breve termine. Paragone temerario, ma spero un minimo efficace...

3.1 La struttura della memoria

Dovrebbe esserti chiaro, ormai, che in informatica nulla è lasciato al caso. Come fa il processore, una volta caricato un programma, a sapere dov'è? Quando usiamo delle variabili, come fa a sapere dove si trovano nella RAM? Insomma: le moderne RAM spesso superano i 4 gigabyte, e un modesto **int** da 4 byte come può non perdersi in quel mare di memoria?

È semplice! Non devi pensare la memoria come un "continuo": visto che siamo Fisici, potremmo dire che la memoria è "quantizzata". Esistono pacchetti minimi di memoria che il sistema operativo può gestire: i *byte*. La memoria è organizzata in byte: è divisa in tante cellette, una adiacente all'altra, ognuna grande un byte.
Ogni celletta non è persa nel nulla, ma ha un *indirizzo*.

Voglio che ti sia chiaro un concetto: le cellette di memoria non sono sparse in modo generico, e non sono neanche poste in maniera "bidimensionale": sono come lungo una retta, una dopo l'altra. Ogni celletta di memoria confina solo con altre due celle. Immaginati la RAM come una lunghissima striscia suddivisa in miliardi di caselle (sì, miliardi!: un gigabyte sono più di un miliardo di byte) contigue e ordinate (ordinate? ovvero ognuna con un proprio indirizzo!).

3.1.1 Indirizzi di memoria

Ogni cella ha un proprio indirizzo, bene, ma come è fatto?

Un indirizzo di memoria non è altro che un numero intero che la rappresenta: ad esempio, la prima celletta potrebbe essere 00001, la seconda subito adiacente 00002, la terza 00003 e così via fino ad arrivare 78354 o addirittura 99999. Insomma: finché c'è spazio la numerazione continua. Io ho usato interi a cinque cifre, ma se la memoria fosse molto più estesa, perché non spingersi fino a 9, 10, ecc... cifre? Nessuno lo vieta, se non la dimensione dalla RAM e l'architettura della CPU; se sei interessato all'argomento prova a leggere i complementi di questo capitolo (3.A).

Però ho commesso, volontariamente, un "errore" per aiutarti a capire. I byte non sono indicizzati in sistema decimale (che io ho usato), ma esadecimale (base 16).

Quando chiediamo al computer di stamparci l'indirizzo di uno specifico byte, ci viene riportato il numero in base 16: dobbiamo velocemente imparare a contare in sistema esadecimale per risolvere gli esercizi di esame (e capire cosa vogliono dire quegli strani simboli che il computer ci riporta quando lo interroghiamo su un indirizzo).

Ho parlato di simboli perché la base 16 ha bisogno di sedici cifre, ma le cifre decimali sono dieci. Le restanti sei dove le troviamo? Le lettere! Ecco: le "cifre" della base sedici sono dieci cifre numeriche e sei lettere. Contiamo da 1 a 16: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f$.

Se vogliamo scrivere il numero 17 in base esadecimale sarà: 10; 18 diventerà 11 e così via.

Insomma non ti stupire se trovi numeri di questo tipo: $1a34f2d$.

In "informatica" i numeri costanti esadecimali, per contrassegnarli dalle normali stringhe di lettere (e dai numeri in base dieci), iniziano in un modo particolare: $0x\ldots$
Un generico indirizzo di memoria potrebbe essere: $0x7ffd97889f98$ (generico ma sensato: l'ho preso dal mio computer!).

Abbiamo parlato di questa immaginaria striscia di cellette di memoria da un byte l'una e abbiamo detto che ognuna di loro ha un indirizzo, ecco un'immagine di come puoi pensarla:

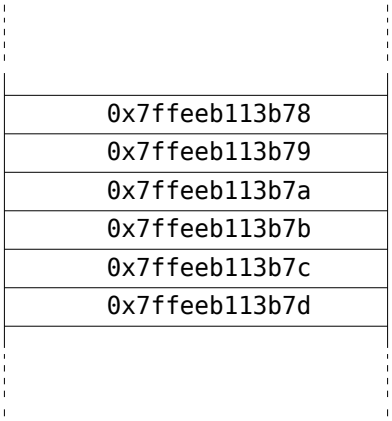

Figura 3.1: Rappresentazione di un pezzo di memoria: ogni cella, con il relativo indirizzo, rappresenta un byte[1].

3.2 Stack e Heap

La RAM è, sul piano logico, suddivisa in due: la *stack* e la *heap*. Si potrebbe dire molto sulle reciproche differenze, ma cercherò di limitarmi alle informazioni che ci saranno utili.

La *stack* è completamente gestita dal compilatore, e perché ciò avvenga è necessario che la memoria del programma sia già definita a *compile time*, cioè al momento della compilazione. La *stack* è la cosidetta *memoria statica*.

Quando noi dichiariamo una serie di variabili, per esempio nel *main*, la memoria necessaria è completamente definita al momento della compilazione: se abbiamo dichiarato un **char** e tre **int** il compilatore sa che deve riservare un byte per il primo e dodici byte per i secondi. Viene detta *memoria statica* perché non può essere modificata dall'utente a *run time*, durante l'esecuzione del programma.

Il compilatore, nel momento in cui definiamo una variabile, le riserva lo spazio necessario nella *stack*; quando la variabile esce dallo *scope* essa, come abbiamo visto, "muore": il compilatore, allora, libera quella memoria. Insomma, fa tutto lui.

Essendo gestita dal compilatore, la *stack* riesce ad essere molto ordinata: è completamente deframmentata. Se dichiariamo tre variabili nel *main*, queste vengono poste in memoria una dopo l'altra, contigue, senza buchi in mezzo.

La *heap*, invece, è gestita dal programmatore. Il vantaggio di questa memoria è che può modificarsi a *run time* (per esempio su richiesta dell'utente durante l'uso del programma),

[1] Ringrazio Stefano Balzan per la realizzazione dell'immagine.

si dice infatti che è *dinamica*. Vedremo solo successivamente il problema dell'allocazione dinamica della memoria, ma quel che volevo introdurti è che questa memoria è molto più "delicata". Perché? Perché siamo noi a doverla gestire. Quando riserviamo spazio nella *heap* ad una variabile, dovremo essere noi a liberarla, non sarà il compilatore a farlo per noi.

Per queste caratteristiche, la *heap* è molto più "disordinata": il compilatore non può ottimizzarla, non conoscendo come si modificherà durante l'esecuzione del programma. La *heap*, infatti, è frammentata: se riserviamo memoria per tre variabili *distinte* non è detto che le relative celle di memoria siano contigue (sottolineo: variabili distinte; questo non vale per gli array, che vedremo tra poco).

3.3 I dati e la memoria

Ci manca da capire come, esattamente, i dati vengono immagazzinati in memoria. A questo punto il passo da fare è estremamente semplice. Sappiamo che la memoria è organizzata in cellette successive da un byte l'una, inoltre conosciamo la dimensione di ogni tipo di dato. L'idea è: una variabile **int** occupa quattro byte, quindi quattro cellette di memoria. Possiamo pensare una variabile di un determinato tipo come una cella diversa da quella elementare, una cella composta dalle cellette elementari di memoria. Quindi, quando pensiamo ad una variabile di un tipo di dato, immaginiamola come una casella non spezzettata e della dimensione necessaria. Ma che indirizzo avrà questa casella? Quello del primo byte che occupa! Il compilatore sa che se usiamo, per esempio, un **int**, l'indirizzo di questo dato inizierà nel primo byte e occuperà quattro cellette (**int** = 4 byte), non una singola.

In `C++` esiste un operatore per richiedere l'indirizzo: "`&`"; posto prima di una variabile, ne restituisce l'indirizzo in memoria. Vi faccio un esempio:

```
#include <iostream>
using namespace std;
int main(){
        int a, b;
        double c, d;
        cout << "A: " << &a << endl << "B: " << &b << endl;
        cout << "C: " << &c << endl << "D: " << &d << endl;
        return 0;
}
```

Esempio 3.1

Sul mio computer l'output di questo programma è:

```
A: 0x7fff0d9ba298
B: 0x7fff0d9ba29c
C: 0x7fff0d9ba2a0
D: 0x7fff0d9ba2a8
```

Sappiamo che gli **int** occupano 4 byte, mentre i **double** 8 byte. Inoltre, essendo nella *stack* (per ora sarà tutto sulla *stack*, la *heap* la vedremo solo nel capitolo sull'allocazione dinamica), i dati sono contigui. Tutto ciò è visibile negli indirizzi di memoria che il mio computer ha stampato.
Guarda l'ultima cifra, 8: siccome `a` è un **int** occupa 4 byte. Per arrivare al dato successivo dobbiamo spostarci di quattro cellette di memoria: 8, 9, a, b. Il dato successivo inizierà nella celletta con indirizzo che termina per "c".
Abbiamo un altro **int**, stesso ragionamento di prima: c, d, e, f.
Il dato che segue inizierà nella posizione successiva, dopo la f abbiamo terminato le cifre della nostra base esadecimale, quindi (come faremmo in base 10), scala la cifra prima e la nostra torna a zero: la cifra precedente era 9, diventa "a", la nostra cifra torna a 0.
Ci troviamo tra le mani un **double**, 8 byte, partiamo da 0 e abbiamo: 0,1,2,3,4,5,6,7. Il dato che segue inizierà, quindi, in posizione "8".

Tutto chiaro?

3.4 Array: elementi di memoria contigui

In `C++` esiste una struttura di dati estremamente utile: l'*array*[2]. Cos'è?

L'*array* rappresenta una struttura matematica molto utilizzata: il vettore. Un vettore è un insieme di elementi generalmente dotati di caratteristiche comuni, come potrebbero essere le coordinate di un punto. È naturale raggruppare gli elementi di un vettore in una struttura unica.

Ti faccio un rapidissimo esempio: pensiamo alle coordinate di un punto nello spazio. Possiamo immagazzinarle in tre variabili distinte:

```
        float x, y, z;
```

Esempio 3.2

Ma perché non raggrupparle in un vettore? In `C++` esiste l'*array*, dichiariamo un *array* di tre elementi così:

```
        float coordinate[3];
```

Esempio 3.3

Per accedere agli elementi dell'*array*, è necessario scrivere il nome della variabile con in parentesi quadre il numero dell'elemento che richiediamo. Attenzione: si parte a contare da 0, non da 1!

Esempio:

```
int main(){
        float coordinate[3];
        coordinate[0]=1.;
        coordinate[1]=12.2;
        coordinate[2]=-17.17;
```

[2]NOTA: segue una brevissima introduzione dell'argomento. L'anticipo qui giusto per analizzare alcune caratteristiche della memoria; una spiegazione più esaustiva verrà presentata nel capitolo dedicato.

```
        return 0;
}
```

Esempio 3.4

Per quanto riguarda l'uso e le proprietà degli *array*, dai un'occhiata al relativo capitolo; ora occupiamoci, invece, di come questa struttura di dati è rappresentata in memoria. Ti è utile per tre motivi: per capire come trattare gli *array*; per capire come costruire un *array* dinamico; perché la maggior parte dei temi d'esame sugli indirizzi di memoria riguarda gli *array*.

La caratteristica fondamentale di un *array* è che, in macchina, si tratta di un insieme di blocchi di memoria *contigui*, sia che ci si trovi nella *stack* che nella *heap*. Quando definiamo un *array* il compilatore riserva sempre un blocco di memoria contiguo, non spezzettato, sempre!

Come è fatto? Nell'esempio precedente, se proviamo il seguente codice (immaginatelo prima del return), cosa succede?

```
        cout << coordinate << endl;
```

Esempio 3.5

Accade una cosa curiosa: viene stampato un indirizzo di memoria, non, come potremmo aspettarci, il primo elemento dell'*array*.

Infatti, `coordinate` è una variabile che più avanti chiameremo *puntatore*, che contiene un indirizzo di memoria, precisamente l'indirizzo del primo elemento dell'*array*.

Se dichiariamo un *array* di **float**, il compilatore sa che ogni elemento occupa 4 byte: sa che è una serie di blocchi di memoria contigui da 4 byte ciascuno. Per potersi ricordare dove si trovano, si salva l'indirizzo del primo, ma come fa a trovare i successivi? Facile! Sa la posizione del primo: per trovare il secondo somma quattro cellette (da un byte l'una), nella successiva inizierà l'elemento seguente e così via. Puoi immaginare un *array* in memoria come nella figura 3.2.

Una parte di un esercizio di un tema d'esame (25 febbraio 2014) chiedeva:

```
Sia
    int w[5];
se l'indirizzo di w[0] termina con 1dfd come terminano gli indirizzi
di w[2] e w[4]?
```

Per trovare l'indirizzo di `w[2]`, dobbiamo sommare 8 byte all'indirizzo di `w[0]`, e quindi troviamo $1e16$. Per trovare l'indirizzo di `w[4]`, sommiamo altri 8 byte e troviamo $1e1a$.

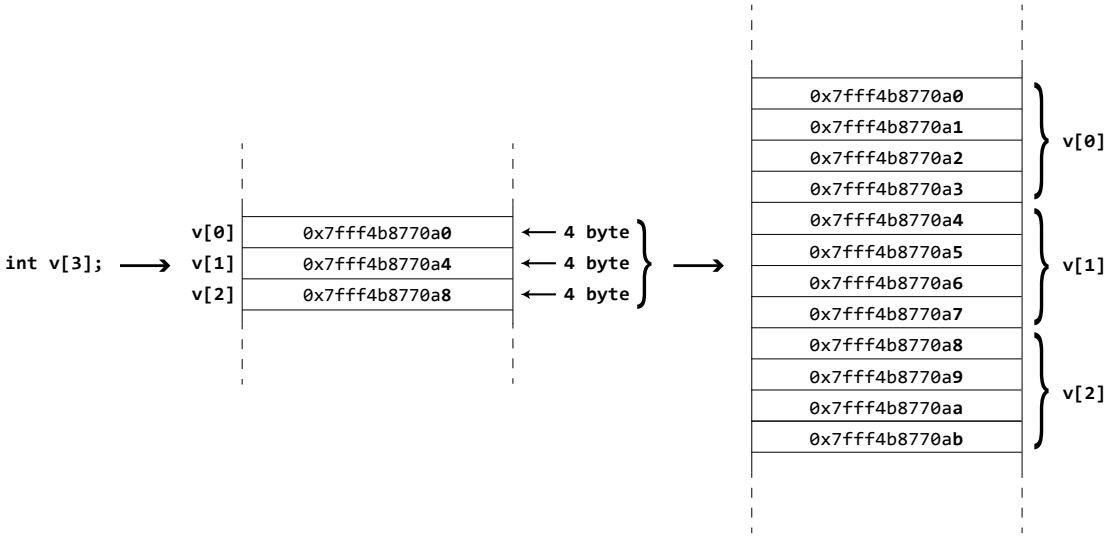

Figura 3.2: Rappresentazione di un *array* in memoria. La prima schematizzazione rappresenta gli elementi (**int**) dell'*array* con i relativi indirizzi. La seconda schematizzazione rappresenta tutti i singoli byte dell'*array*: notare che l'indirizzo di ogni elemento è l'indirizzo del suo primo byte.[3]

COMPLEMENTI

3.A Architetture a 32 e 64 bit

Questa sezione, per programmare in `C++`, non ha alcuna utilità pratica, ma può essere interessante a livello "culturale".

Hai mai sentito parlare di computer a 32 e a 64 bit?

Il processore comunica con la memoria tramite dei *bus*, che sono veri e propri canali fisici della scheda madre: vi è il *bus* degli indirizzi, il *bus* dei dati e il *bus* di controllo; quest'ultimo, a noi, non interessa. Nel *bus* degli indirizzi viaggia l'indirizzo fisico a cui andare a leggere o scrivere, mentre nel *bus* dei dati passa il dato letto o da scrivere. Ecco, la dicitura "32 bit" indica esattamente la dimensione di questi *bus*, oltre alla dimensione di alcune piccolissime memorie interne alla CPU, detti *registri* (la RAM, per la CPU, è lenta: i *registri* sono piccolissime memorie ad accesso praticamente istantaneo utilizzate per scrivere i risultati intermedi di calcoli vari). L'insieme di queste caratteristiche costituisce l'*architettura* della CPU.

I primi computer avevano *bus* da 8 bit, quindi da 16 (ad esempio Windows 95 era scritto per girare su computer di questo tipo). Poi si è passati a 32 e, più o meno recentemente, sono nate CPU a 64 bit, che oggi predominano il panorama.

Ma cosa cambia tra queste diverse architetture?

La prima grande restrizione è nel *bus* degli indirizzi. Torniamo alla fine degli anni '90, quando l'architettura maggiormente in voga era il 32 bit. Quanto è grande un indirizzo da 32 bit in base decimale? Una variabile di questa dimensione può immagazzinare numeri che

[3] Ringrazio Stefano Balzan per la realizzazione dell'immagine.

vanno da 0 a 4.294.967.296. Se ogni numero rappresenta un indirizzo di memoria (e, ricorda, ogni indirizzo è un byte), posso indicizzare circa $4 \cdot 10^9$ byte, ovvero 4 gigabyte. Ecco trovato il primo limite dei 32 bit: un processore di questo tipo non può gestire una RAM più grande di 4 gigabyte[4].

L'altro limite riguarda il *bus* di dati: anche le variabili sono limitate a 32 bit e una CPU di questo tipo non sa gestire in maniera naturale i **long** e i **double**.

Ben presto i 32 bit divennero eccessivamente limitanti e, nei primissimi anni duemila, nacque l'architettura a 64 bit: il primo utilizzatore (e per un po' l'unico) fu Linux. Con i 64 bit la dimensione fisica di entrambi i *bus* fu raddoppiata. Consideriamo gli indirizzi: 2^{64} è circa $1.8 \cdot 10^{19}$, ovvero circa 18 milioni di terabyte ($1.8 \cdot 10^{10}$ gigabyte, qualcosa di spropositato). Le prime versioni dell'architettura a 64 bit limitarono il *bus* virtuale degli indirizzi a 48 bit: in questo modo sono indicizzabili 256 terabyte di RAM, assolutamente più che sufficienti oggi e, probabilmente, per molti altri anni[5].

I diversi sistemi operativi gestiscono questi indirizzi in maniera differente. Linux usa per i processi 47 bit di indirizzi rendendo utilizzabili "solo" 128TB (mentre il kernel li può usare tutti e 256).

Windows, fino alla versione 7, limitava l'uso di solo 8TB di indirizzi virtuali. Dalla versione 8.1 è stato esteso l'uso a 128TB di indirizzi, ma purtroppo non esistono compilatori open-source a 64 bit (il compilatore proprietario Microsoft è a pagamento). Inoltre, a meno di non dare al compilatore l'opzione "*LARGEADDRESSAWARE*", tutti i programmi sono limitati a 2 gigabyte (quindi anche meno dei 4GB delle vecchie CPU da 32 bit...).

Per quanto riguarda MacOS, ammetto la mia completa ignoranza.

Tornando ai *bus*, rimane da analizzare quello dei dati: a differenza del *bus* degli indirizzi, viene utilizzato interamente, permettendo di gestire in modo naturale dati da 8 byte (64 bit), cioè i nostri **long** e **double**. Ma non è l'unico vantaggio: l'esistenza delle piccole memorie interne alla CPU, anch'esse da 64 bit, permette di lavorare contemporaneamente su due dati da 4 byte (**float** e **int**) dimezzando il tempo di calcolo (se il codice macchina istruisce la CPU a farlo), ma questo interessa a chi scrive i compilatori....

[4] In realtà ben presto nacque la tecnologia PAE ("Phisical Address Extention") che aggiunge 4 bit agli indirizzi fisici: il sistema operativo riesce così a gestire fino a 64GB di memoria, ma i singoli programmi rimangono limitati a 4GB.

[5] Alla data di scrittura, la maggior parte dei computer portatili sono dotati di 4-16 gigabyte di memoria, i server più potenti arrivano a 128.

CAPITOLO 4

Strutture di controllo e cicli

Abbiamo visto buona parte di ciò che concerne i dati e la memoria; siamo capaci di usare le variabili e abbiamo capito come vengono trattate nella RAM. Ma, con gli strumenti che abbiamo, il nostro codice non ha molte potenzialità operative, non è particolarmente versatile.

In questo capitolo studieremo due importantissime strutture che rendono il codice potente e dinamico: le strutture di controllo e i cicli. Dal momento che le prime sono spesso utili nelle seconde, inizieremo con queste.

4.1 Strutture di controllo

Ci sono situazioni in cui l'operazione che desideriamo eseguire cambia in base alle condizioni iniziali. Mi spiego meglio: se all'inizio abbiamo una situazione, potremmo voler fare una determinata operazione; se invece la situazione è un'altra, potremmo desiderare una diversa operazione. Come facciamo? Il nostro codice viene eseguito in maniera consequenziale: un'operazione dopo l'altra (una riga di codice dopo l'altra). Come facciamo a dire: "No, non eseguire quest'operazione, esegui quella!"?

Per capire meglio ti propongo un esempio pratico: vogliamo scrivere un programma che, preso in input un numero, ne trovi il valore assoluto.

Se viene inserito un numero negativo, ci basta farne il modulo moltiplicando per -1, ma se viene inserito un numero positivo? La moltiplicazione per -1 non deve assolutamente avvenire!

Cerchiamo di estrapolare i concetti generali. Per prima cosa dobbiamo controllare la situazione iniziale, ovvero se si verifica una condizione specifica: nel nostro caso il segno del numero (ad esempio, per condizione prendiamo la negatività del numero). A seconda che la condizione si verifichi o no, dobbiamo eseguire un'istruzione o l'altra.

Tornando al nostro esempio: se il numero è negativo (condizione verificata) allora dobbiamo trovarne il valore assoluto, ma se il numero è positivo (condizione non verificata) allora

non dobbiamo eseguire nessuna operazione.

Nota una cosa: il controllo di una condizione è un'operazione binaria. La condizione è verificata o non è verificata, non esiste una terza opzione. Una variabile booleana è perfetta per rappresentare questa situazione: può essere settata su **true** se la condizione si verifica, **false** negli altri casi.

4.1.1 *if... else*

Passiamo ad analizzare il codice che rappresenta questa situazione.

if(...){...} Esiste un costrutto che, a partire da una situazione iniziale, controlla se una condizione viene verificata. Quindi, se ciò avviene, entra in un nuovo scope ed esegue ciò che è contenuto.

Questo costrutto è il seguente:

```
if(condizione della situazione iniziale){//nuovo scope
        //istruzioni
}
```

Esempio 4.1

Nota che **if** è una parola riservata. "If" viene seguito da parentesi tonde: al loro interno vi è un codice (generalmente breve) che controlla la situazione iniziale, questo codice deve produrre un risultato tassativamente booleano. Infatti, se nelle parentesi tonde, a controllo concluso, risulta un valore **true** allora il programma entra nelle parentesi graffe che seguono; se, invece, si trova un valore **false** ciò non avviene.

Per capire meglio l'uso di questo enunciato all'interno di un programma, vediamo un esempio pratico: vogliamo scrivere un programma che decida se è pari un numero inserito da tastiera.

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

int main(){
        int numero=0;
        cout << "Inserisci un numero: " << endl;
        cin >> numero; //cin legge da tastiera un valore e lo inserisce in
            numero
        if(numero%2==0){
                cout << "Il numero e' pari!" << endl;
        }
        return 0;
}
```

Esempio 4.2

Cosa succede? Fino a *cin* dovrebbe esserti tutto chiaro. *Cin*, così come *cout*, è un operatore di input/output (o meglio, *cin* di input, *cout* di output). *Cin* è un'abbreviazione di "console

input" e legge da tastiera un valore, viene seguito da "»" e da una variabile: scrive il valore inserito da tastiera nella variabile. Per una spiegazione più approfondita vedi il capitolo in/out.

A questo punto interviene **if**: trova il resto della divisione e lo paragona con zero, e se è uguale a zero (il numero è pari) il confronto restituisce **true**. Potremmo dire che, in questo caso, **if** dà l'ok per entrare nelle parentesi graffe: il codice in esse contenuto viene eseguito. Se il resto della divisione non è zero, il confronto con zero dà come risultato **false**: **if** non permette di entrare nelle parentesi graffe e il relativo codice non viene eseguito.

Perché ho scritto "==" e non un semplice "="? Questi due simboli, in realtà, sono profondamente diversi. Il primo è un operatore binario[1], ovvero confronta ciò che è a destra con ciò che è a sinistra: se sono uguali restituisce **true**, altrimenti **false**. Il semplice "=", invece, è un operatore di assegnazione: il valore contenuto nella variabile a destra viene assegnato a quella di sinistra.

Nella prossima sottosezione approfondiremo gli operatori binari e il loro funzionamento. Prima, però, voglio introdurre il completamento naturale della struttura **if**.

else{...} Nell'esempio precedente controlliamo se una variabile è pari e stampiamo a video questa considerazione. E se volessimo stampare a video qualcosa anche quando la variabile è dispari? Potremmo pensare di fare una cosa così (ti avviso, è sbagliato!):

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

int main(){
        int numero=0;
        cout << "Inserisci un numero: " << endl;
        cin >> numero; //cin legge da tastiera un valore e lo inserisce in
            numero
        if(numero%2==0){
                cout << "Il numero e' pari!" << endl;
        }
        cout << "Il numero e' dispari!" << endl;
        return 0;
}
```

Esempio 4.3

Se il numero è dispari il programma non entra nello scope di **if** e tutto va bene: viene stampata la cosa giusta. Ma se il numero è pari? Abbiamo un problema: non solo il programma entra nello scope di **if**, ma viene stampato anche ciò che segue. Il risultato ottenuto non è quello desiderato! Potremmo procedere con un secondo **if** dove, per condizione, andremmo a controllare la cosa *esattamente opposta*.

Quando dopo un **if** abbiamo un codice che deve essere eseguito al verificarsi della condizione opposta dell'**if** iniziale, ci viene in aiuto un'altra struttura (in modo da evitare un secondo e ridondante **if**): **else**.

[1] Con operatore binario, si intende che necessita di due operandi. Un operatore che lavora su un solo operando è detto "unario". Ci tengo a sottolineare che "binario" non significa, quindi, che opera sui bit. Un operatore che agisce sui bit è detto "bit a bit" o, in inglese, *bitwise* (un operatore binario, invece, è detto *binary*).

Il codice di prima, corretto, diventa:

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

int main(){
        int numero=0;
        cout << "Inserisci un numero: " << endl;
        cin >> numero; //cin legge da tastiera un valore e lo inserisce in
            numero
        if(numero%2==0){
                cout << "Il numero e' pari!" << endl;
        }
        else{
                cout << "Il numero e' dispari!" << endl;
        }
        return 0;
}
```

Esempio 4.4

Come funziona **else**? Si oppone ad un **if**. Se la condizione di entrata nell'**if** non è verificata, allora si entra nello scope dell'**else** (quest'ultimo, quindi, non ha bisogno di alcuna parentesi tonda contenente una condizione). È implicito che un **else** non ha senso se non è preceduto da un **if**.

NOTA Una piccola precisazione: se l'istruzione da eseguire dentro un **if** o un **else** è costituita da una singola riga di codice (se e solo se è una sola riga!), allora possiamo evitare le parentesi graffe dello scope. Te lo dico perché, per brevità del codice, spesso nei miei esempi scriverò così. Varrà lo stesso per i cicli, ma lo ripeterò.

Ti faccio notare che un **if** non deve per forza essere seguito da un **else**. Ci sono casi in cui, se la condizione non si verifica, non deve essere eseguito alcun codice alternativo: un caso è l'esempio del valore assoluto che avevo introdotto all'inizio. A dirla, tutta dovresti essere in grado di scrivere il programma da solo (fallo!), in ogni caso ti riporto una possibile soluzione:

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

int main(){
        int numero=0;
        cout << "Inserisci un numero: " << endl;
        cin >> numero;
        if(numero<0)
                numero=numero*(-1);
        cout << "Valore assoluto: " << numero << endl;
        return 0;
}
```

Esempio 4.5

Le strutture di controllo possono essere incluse una dentro l'altra, incastrate in una scalata di **if** ed **else**; ad esempio:

```
#include <iostream>
using namespace std;
int main(){
        int numero=0;
        cout << "Inserisci un numero maggiore di zero: " << endl;
        cin >> numero;
        if(numero<10){
                cout << "Il numero ha una singola cifra." << endl;
                if(numero%2==0){
                        cout << "Il numero e' pari. " << endl;
                        if(numero==2)
                                cout << "Il numero e' primo." << endl;
                }
        }

        return 0;
}
```

Esempio 4.6

Non è l'unica situazione dove puoi trovare tanti **if** e, ovviamente, **else**: nell'esempio di prima per ogni condizione avrei potuto mettere il complementare dentro un **else**); l'altra situazione è quella, mi vien da dire, "a cascata": tanti **if** scollegati e uno dopo l'altro.

Ti riporto un esempio molto semplice. Vedremo più avanti che esiste una struttura che è in grado di risolvere la stessa situazione in maniera elegante, ma solo in condizioni molto particolari.

```
#include <iostream>
using namespace std;
int main(){
        int c=0;
        cout << "Inserisci un numero da 1 a 5 corrispondente ad una lettera
            da 'a' a 'e': " << endl;
        cin >> c;
        if(c==1)
                cout << "a!" << endl;
        if(c==2)
                cout << "b!" << endl;
        if(c==3)
                cout << "c!" << endl;
        if(c==4)
                cout << "d!" << endl;
        if(c==5)
                cout << "e!" << endl;
        return 0;
}
```

Esempio 4.7

4.1.2 Operatori binari e logici

Abbiamo visto che **if** vuole, nelle parentesi tonde, un enunciato che produca un valore booleano. La cosa più banale è una variabile **bool**; questa non è, però, l'unica situazione che produce un risultato booleano: molto più notevoli sono gli operatori binari.

Gli operatori binari valutano una relazione tra due variabili, e a seconda del risultato restituiscono **true** o **false**. Nei codici precedenti ne ho introdotti un paio senza molte spiegazioni: vediamoli in dettaglio.

Chiedersi se una variabile è maggiore (o minore) di un'altra è una relazione binaria: se la variabile è effettivamente maggiore (rispettivamente minore) allora verrà restituito **true**, al contrario **false**.

Esistono operatori binari relativi alla matematica ed operatori puramente logici, vediamoli!

Gli operatori binari matematici che andremo ad utilizzare sono i seguenti (il codice in `C++` segue tra virgolette dopo la freccina):

- **Uguale → " == "**
  Come ho già accennato, c'è una profonda differenza tra "=" e "==". Il singolo uguale è un operatore di assegnazione. Ne abbiamo già discusso, non approfondirò di nuovo il significato. I programmatori del `C`, per distinguere l'operatore logico "uguale" dall'uguale di assegnazione hanno deciso che, per il primo, il simbolo sarebbe stato "==".
  Quest'operatore controlla se ciò che è posto a sinistra coincide con ciò che è posto a destra, in tal caso produce un valore **true**; se ciò non avviene produce un **false**.
  Stai attento a non confondere "==" con "=": se lo fai combini sicuramente un bel disastro nel tuo codice, hai capito perché?

  Importante: non usare *mai* quest'operatore tra variabili di tipo **float** o **double**. Quando abbiamo studiato i dati in virgola mobile, abbiamo visto che non sempre ritroviamo risultati esatti o quello che ci aspettiamo. Un'operazione che dovrebbe generare "0", nel computer potrebbe produrre un risultato prossimo a tale valore, ma non esattamente zero!!!
  Ti ricordi l'esempio di codice che ti avevo proposto per farti vedere un caso in cui accade esattamente quel problema? Un programma come quello che segue non funzionerà mai:

```
#include <iostream>
using namespace std;
int main(){
        float a=0.3, b=0.2, c=0.1;
        float d=a-b-c;
        if(d==0)
                cout << "Risultato corretto! " << endl;
        return 0;
}
```

Esempio 4.8

  Questo è un programma inutile e stupido, ma se nel tuo codice, scrivendo programmi seri, usi dei numeri in virgola mobile con "==" potresti creare errori per nulla facili da individuare.
  Usa l'operatore "==" solo con **int**, **short**, **long**, **char** e **bool**.

- **Diverso** → **“ != ”**
  Non c'è molto da spiegare: è l'esatto opposto dell'operatore precedente. Il punto esclamativo nega ciò che segue: significa “non uguale”. Questo operatore confronta ciò che è a sinistra con ciò che è a destra: se differiscono produce un valore **true**.
  Giusto per completezza: esiste una parola riservata per indicare ciò che in simboli è “!=”: **not_eq**; non ti capiterà praticamente mai di trovarla, ma nel caso sai cos'è...
  Vale la stessa considerazione di prima sul non usare numeri in virgola mobile (anche perché, essenzialmente, quest'operatore è identico al precedente: semplicemente produce risultati opposti).

- **Maggiore** → **“ > ”**
  Funziona esattamente come il relativo operatore matematico. Inoltre, è molto meno problematico nell'uso con i numeri a virgola mobile dei precedenti operatori: l'unico caso critico è quello in cui, di nuovo, due numeri dovrebbero essere uguali (e quindi restituire **false**) ma in realtà uno è leggermente maggiore dell'altro. È comunque un caso raro e, perciò, è comune usare questo operatore anche con i **float** e i **double**.

- **Maggiore o uguale** → **“ >= ”**
  Anche in questo caso, stesse funzionalità dell'operatore matematico e quasi nessun problema con i **float** e i **double**.

- **Minore** → **“ < ”**
  Stesse considerazioni fatte per “>”.

- **Minore o uguale** → **“ <= ”**
  Stesso discorso del “>=”.

Abbiamo visto praticamente tutti gli operatori binari matematici. Ci sono, però, ancora due operatori binari, diciamo “logici”: *and* e *or*.

Questi due operatori, a differenza dei precedenti, a sinistra e a destra hanno dei valori booleani e stabiliscono delle relazioni tra gli stessi restituendo un nuovo valore booleano. Vediamo in dettaglio come funzionano.

- **and**: se sia a destra che a sinistra vi sono due **true**; restituisce a sua volta **true**, se i due valori sono diversi o entrambi **false**, allora restituisce **false**.
  Nel `C` non esisteva la parola riservata **and**: il simbolo per rappresentarla era “&&”. Nel codice scritto da qualche nostalgico (o pezzi di codice derivanti dal `C`), potresti trovare questo simbolo: non spaventarti, è un semplice **and**.

- **or**: affinché sia prodotto un valore **true**, è sufficiente che solo uno dei due valori sia **true** (ovviamente possono esserlo entrambi); per cui, l'unico caso in cui viene prodotto un valore **false** è quando entrambi sono **false**.
  Anche **or** non esisteva nel `C`: il relativo simbolo era “||”.

- Ti aspettavi due punti e invece ecco il terzo! Voglio fare una precisazione: “**!**” può essere usato in situazioni diverse dal “!=”. Lo possiamo usare per negare una qualsiasi operazione che produca un valore booleano.
  Il punto esclamativo è, infatti, un operatore logico vero e proprio che nega ciò che segue ma non è un operatore binario: prende un solo argomento.

Di nuovo, per completezza, sappi che esiste una parola riservata per indicare "!": **not**; ti sarà raro trovarla, ma nel caso...
Se scriviamo:

```
if(!( a>0 and a<10))
```

Esempio 4.9

stiamo imponendo che `a` deve essere minore di 0 e maggiore di 10 (estremi inclusi). In questo caso, avremmo potuto semplicemente invertire i versi. Ci sono situazioni, però, in cui torna comodo (ad esempio per negare il valore che ritornano alcune funzioni, nei prossimi capitoli vedrai qualche esempio).
Ovviamente puoi usarlo per negare anche una semplice variabile booleana, esempio:

```
bool var=true;
if(var)
        //codice
if(!var)
        //codice
```

Esempio 4.10

Il programma entra nel primo **if**, mentre nel secondo no.

if, **and** e **or** spesso tornano utili per effettuare dei controlli, potremmo dire di "sicurezza", sugli input (e varie altre cose). Ad esempio, nel programma scritto prima in cui si chiedeva un numero da 1 a 5 da convertire in lettera, si potrebbe inserire un "check" sul fatto che effettivamente il numero inserito sia compreso tra 1 e 5. Provo a farti vedere:

```
#include <iostream>
using namespace std;
int main(){
        int c=0;
        cout << "Inserisci un numero da 1 a 5 corrispondente ad una lettera
            da 'a' a 'e': " << endl;
        cin >> c;
        if(c<1 or c>5){
                cout << "Hai inserito un numero non valido!" << endl;
                return 0; //Con questo comando ritorno al sistema operativo
                    0, interrompo qui il programma, non continua!
        }
        /* L'if precedente si sarebbe potuto scrivere anche cosi':
        if(!(c>=1 and c<=5)){
                cout << "Hai inserito un numero non valido!" << endl;
                return 0;
        }
        Spesso in informatica esistono piu' modi, ugualmente funzionali, per
             scrivere la stessa cosa: sta al tuo gusto, e a cosa ti viene in
            mente prima, decidere come comportarti.
        */
        if(c==1)
                cout << "a!" << endl;
        if(c==2)
```

```
                cout << "b!" << endl;
        if(c==3)
                cout << "c!" << endl;
        if(c==4)
                cout << "d!" << endl;
        if(c==5)
                cout << "e!" << endl;
        return 0;
}
```

Esempio 4.11

4.2 Cicli

Può capitare di dover compiere istruzioni estremamente ripetitive, tutte uguali o nelle quali cambia veramente poco. Invece di scrivere migliaia di righe di codice, è stato introdotto un utilissimo strumento: il ciclo.

Come funziona un ciclo? È composto da due parti: un blocco di codice che viene ripetuto in maniera ciclica e un blocco di codice che valuta una condizione: finché la condizione si verifica (o non si verifica, a seconda del tipo di ciclo) il ciclo continua.

È importante notare una cosa: i cicli sono di tre tipi diversi ma sono sempre intercambiabili. Mi spiego meglio: ognuna delle tre tipologie è più adatta ad alcune situazioni, ma ogni ciclo può essere riscritto usandone un altro.

A volte ti sembrerà che un problema possa essere risolto con cicli diversi, e non stai sbagliando! È veramente così (anche se, in alcune situazioni, alcuni cicli sono molto più adatti di altri). La scelta del ciclo è solo tua: da un lato potresti scegliere quello più elegante (che richiede meno righe di codice), dall'altro seguire la tua inclinazione; io, per esempio, tendo ad abusare del ciclo **for**, probabilmente mi piace più degli altri!

4.2.1 *while*

Per spiegartelo preferisco partire da un esempio di codice, poi lo analizziamo insieme.

```
#include <iostream>
using namespace std;
int main(){
        int numero=1; //inizializzo ad uno per poter entrare nel ciclo, vedi
             la spiegazione
        while(numero !=0){
                cout << "Inserisci un numero (zero per chiudere il programma
                    ): ";
                cin >> numero;
                if(numero%2==0) //controllo il resto della divisione
                        cout << "E' pari!" << endl;
                else
                        cout << "E' dispari!" << endl;
        }
        return 0;
}
```

Esempio 4.12

In breve, il programma funziona così: si avvia, chiede di inserire un numero, analizza se è pari o dispari e continua così finché non si inserisce zero.

Dovrebbe esserti tutto chiaro nel resto del programma, le uniche perplessità potrebbero essere, giustamente, nel **while**.

Il ciclo **while** inizia con la relativa parola riservata, quindi è seguito da una parentesi tonda. All'intero di questa vi è un codice che analizza una condizione, e se come risultato viene prodotto un **true** si entra nelle parentesi graffe: il codice contenuto in esse viene eseguito. Arrivati alla fine delle graffe, **while** testa di nuovo la condizione, e così via fino a che non si produce un **false**. In tal caso non si entra più nelle graffe: il programma procede con ciò che segue oltre il ciclo.

Da notare una cosa: prima di entrare nel ciclo si verifica la condizione, e questo vuol dire che potenzialmente non si potrebbe mai entrare nel ciclo. Nell'esempio precedente, se avessi inizializzato `numero` a zero non saremmo mai entrati nel ciclo **while** e il nostro programma sarebbe stato decisamente inutile.

Torniamo all'esempio. In quel caso, qual era il vantaggio e la necessità di usare un ciclo?

Poniamo che il nostro utente sia così negato per la matematica da avere una quantità di numeri non ben definita da processare per decidere se sono pari o dispari (ironizzo: è un modo per dire che l'informatica non è così stupida, si può scrivere di meglio... Ma come al solito cerco esempi banali per scrivere codici brevi). Abbiamo un problema: quando scriviamo il programma non sappiamo quanti numeri vorrà analizzare l'utente, quindi: o lui per ogni numero riavvia il programma (ma sarebbe un programma terribile, nessuno comprerebbe mai qualcosa del genere!) oppure ci ingegniamo per scrivere un programma che si modifichi a run-time, in base alle necessità dell'utente.

Ecco qui che entra in gioco il ciclo: potenzialmente il programma va avanti all'infinito, continua a chiedere numeri all'utente da analizzare; è quest'ultimo che interrompe il programma quando è soddisfatto.

Esistono molti altri usi di cicli, alcuni dei quali non dipendono dall'utente, ma da condizioni del computer, o da risultati di calcoli effettuati dal computer. A breve cercherò di presentarti qualche esempio più significativo.

Scopes e cicli Una caratteristica comune a tutti i cicli è che ogni singola ripetizione rappresenta uno *scope*. Questo cosa significa? Una variabile dichiarata all'interno delle graffe del ciclo è visibile solo e soltanto in quel singolo *scope*: ad ogni ripetizione nasce, viene usata e alla fine della graffa muore; alla ripetizione successiva, nasce di nuovo. Per cui, se avevi assegnato un valore alla ripetizione precedente, non sarà più visibile!

Questo esempio dovrebbe chiarire:

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

int main(){
        int i=0;
        while(i<10){
                int j=1;
                i=j;
                j++;
```

```
        }
        return 0;
}
```

Esempio 4.13

Il ciclo non si interrompe mai: `j` ad ogni ciclo rinasce e averlo incrementato alla ripetizione precedente non è servito a nulla[2].

4.2.2 *do... while*

Nel ciclo **while**, come abbiamo visto, prima di entrare nelle graffe viene testata la condizione, in questo modo non è detto che si entri nel ciclo.

Potremmo desiderare, a volte, che avvenga il contrario: per prima cosa si entri, almeno una volta, nel ciclo, dopo di che, per decidere se entrarci di nuovo, dobbiamo testare la condizione, e così via.

Riprendiamo l'esempio di prima: affinché si entrasse nel **while** abbiamo dovuto inizializzare la variabile `numero` al valore di uno. Dopo un po' che si programma, nel corso di Informatica 1, inizia a diventare automatico inizializzare le variabili numeriche a zero; ricordarsi di inizializzarla ad un valore diverso può essere non del tutto banale. Insomma, tutto questo per dire: non sarebbe più comodo inizializzare la variabile a zero ed avere un ciclo che, almeno una volta, chiede all'utente di inserire un numero e poi testa la condizione?

Esiste il ciclo **do... while**; riscrivo il programma di prima utilizzandolo, poi, come al solito, spiego tutto.

```
#include <iostream>
using namespace std;
int main(){
        int numero=0;
        do{
                cout << "Inserisci un numero (negativo per chiudere il
                    programma): ";
                cin >> numero;
                if(numero%2==0) //controllo il resto della divisione
                        cout << "E' pari!" << endl;
                else
                        cout << "E' dispari!" << endl;
        }while(numero >=0);
        return 0;
}
```

Esempio 4.14

Il programma, arrivato alla parola riservata **do**, entra nel ciclo, esegue il codice e, solo successivamente, arriva al **while**, dove testa la condizione e decide se entrare in un nuovo ciclo.

[2] Se un tuo programma entra in *loop* e lo vuoi interrompere, premi Control+C.

Ti faccio notare una differenza rispetto al ciclo **while**: nel **do... while** è *necessario*, alla fine, un bel punto e virgola!

Per concludere: i due cicli analizzati finora sono estremamente simili, la differenza è che il secondo esegue almeno una volta il codice tra le graffe.

Come accennato, i vari cicli sono interscambiabili. Se vuoi fare un esercizio, per convincerti, prova a scrivere quest'ultima versione del programma (con `numero` inizializzato a 0) con un ciclo **while**. Dovrai inserire qualcosa di nuovo: prima, dentro o dopo? E cosa?

4.2.3 *for*

Passiamo all'ultimo ciclo, il quale, apparentemente, è tutta un'altra cosa. Un esempio:

```
#include <iostream>
using namespace std;
int main(){
        unsigned int num=0;
        cout << "Programma per conto alla rovescia, da che numero vuoi
            partire?: ";
        cin >> num;
        for(int i=0; i<num; i++){
                cout << num-i << endl;
        }
        cout << "Zero!" << endl;
        return 0;
}
```

Esempio 4.15

Il ciclo **for** è composto da due parti: una parentesi tonda e la solita parentesi graffa con il codice da eseguire.

La parentesi tonda, però, è profondamente diversa da quelle degli altri due cicli, vediamo perché.

Come puoi notare, vi sono tre parti distinte separate da un punto e virgola. La prima parte serve per dichiarare ed inizializzare una variabile, detta contatore; generalmente viene chiamata `i`, ma puoi chiamarla come vuoi (se hai tanti **for** uno dentro l'altro, devi per forza dare nomi diversi); io l'ho inizializzata a zero, ma il valore può essere qualsiasi (spesso si sceglie un valore comodo e sensato per quello che dobbiamo fare). La seconda parte è quella in cui viene testata la condizione, nello specifico che la nostra `i` sia minore di `num`: finché ciò si verifica le iterazioni continuano. La terza parte effettua un'operazione sulla variabile contatore: per esempio io l'ho incrementata (specifico: ad ogni iterazione viene incrementata).

Il concetto è che `i` parte da 0 e ad ogni iterazione si incrementa di 1. Quando arriva al valore `num`-1 (e quindi il programma può ancora entrare nel ciclo) `i` viene incrementato di 1: al successivo controllo non è più minore di `num`, così si esce dal ciclo **for**.

A volte, le variabili contatore partono da un valore, tipo 10, e vengono decrementate; insomma, non per forza dobbiamo incrementarla!

Ti accorgerai dell'utilità dei cicli, in particolare del **for**, parlando di *array*.

Ti faccio notare che, così come con gli **if**, se abbiamo una sola riga di codice, anche nei cicli possiamo evitare le parentesi graffe.

Ora come ora, probabilmente, ti sarà difficile vedere come il ciclo **for** possa essere equivalente agli altri due. Per capirlo ci serve uno strumento molto utile: il **break**.

4.2.4 Gli enunciati *break* e *continue*

Abbiamo visto che tutti i cicli controllano *una* condizione, solo una! E se noi volessimo controllarne un'altra? Se all'accadere di un particolare evento (uno nuovo, non la condizione del ciclo) volessimo interrompere il ciclo? È possibile: la combinazione di un **if** e un enunciato **break** lo permette.

Quando in un ciclo il programma incontra un enunciato **break**, a qualsiasi iterazione si trovi, il ciclo viene interrotto, si esce dallo *scope* e si continua nel codice del programma.

Un **break** può essere davvero utilissimo: possiamo usarlo come controllo sulle iterazioni del ciclo (ad esempio interromperlo se dura troppo), oppure per interruzioni forzate da parte dell'utente, ecc...

Propongo un esempio: è un programma per calcolare il fattoriale di un numero. Quest'operazione è molto delicata, perché genera numeri enormi con una facilità notevole. Per non rischiare l'overflow possiamo inserire un check nel ciclo: quando il nostro calcolo raggiunge una determinata cifra lo interrompiamo.

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

int main(){
        unsigned int num=0; //unsigned per avere numero massimo piu' grande!
        unsigned int appo=0; //creo una variabile d'appoggio per il calcolo
        cout << "Inserisci un numero di cui calcolare il fattoriale: ";
        cin >> num;
        appo=1; //per il calcolo del fattoriale
        for(int i=1; i<=num; i++){
                appo=appo*i;
                if(appo>=100000000){ //scelgo questo numero perche' mi
                    sembra ragionevole rispetto alle dimensioni della
                    variabile
                        cout << "Numeri troppo grandi!" << endl;
                        break;
                }
                if(i==num) //se abbiamo incontrato il break questo non viene
                     eseguito mai
                        cout << num << "! = " << appo << endl;
        }
        return 0;
}
```

Esempio 4.16

Ti accorgerai, se provi a compilare il programma, che con numeri molto piccoli superiamo già il limite imposto.

Potremmo scrivere un controllo sull'overflow decisamente migliore, che ci permetta di utilizzare i limiti dell **unsigned int** fino in fondo. Se hai voglia, te lo lascio per esercizio!

Passiamo ora al **continue**. In realtà è molto più raro da usare, trova molte meno applicazioni, e sinceramente nel corso di Informatica 1 non l'ho mai visto usare. Comunque, io te l'accenno: non si sa mai che tu lo possa trovare utile in qualche tuo codice.

L'enunciato **continue** dice: interrompi questa iterazione, non continuare con il codice nello scope del ciclo, e vai all'iterazione successiva. In poche parole, a differenza del **break** non interrompe il ciclo, ma solo la singola iterazione dove viene incontrato: fa passare il programma alla successiva.

Un esempio molto semplice:

```
#include <iostream>
using namespace std;
int main(){
        int salta=0;
        cout << "Inserisci un numero da 1 a 10: quale numero vuoi saltare
            nel conto alla rovescia? " ;
        cin >> salta;
        for(int i=10; i>0; i--){
                if(i==salta)
                        continue;
                cout << i << endl;
        }
        cout << "Zero! " << endl;
        return 0;
}
```

Esempio 4.17

Visti questi elementi, ti è più chiaro come un **for**, insieme ad un **if** e **break**, possa sostituire un **while**? Fai qualche prova!

4.2.5 I cicli infiniti

Un particolare ciclo è il cosiddetto "ciclo infinito". Un ciclo che, in teoria, non si interrompe mai: non esiste una condizione di uscita. O meglio: non esiste come condizione inclusa nel ciclo, ma esiste tramite un **if-break**.

I cicli infiniti sono utili quando a priori non sappiamo quanti cicli dovremo fare, oppure se le condizioni di uscita sono tante (e di pari importanza).

Possiamo scrivere cicli infiniti sia usando **for** che **while** (e, ovviamente, **do...while**):

```
        //Ciclo for infinito
        for( ; ; ){
                espressione da eseguire;
                if(condizione di uscita)
                        break;
                if(altra condizione di uscita)
                        break;
                ... //Altre condizioni?
        }
```

```
        //Ciclo while infinito
        while(true){
                espressione da eseguire;
                if(condizione di uscita)
                        break;
                if(altra condizione di uscita)
                        break;
                ... //Altre condizioni?
        }
```

Esempio 4.18

Ho messo più condizioni per rendere chiaro che possono essere tante, in realtà ne basta una sola (ma almeno una... se no, buona attesa per la fine del programma!).

Il ciclo **for** infinito ha un vantaggio: può implementare in maniera elegante un contatore di iterazioni. Ti mostro:

```
        for(int i=0; ; i++){
                espressione;
                if(condizione di uscita)
                        break;
        }
```

Esempio 4.19

Mancando la parte centrale del **for**, non vi è un limite alle iterazioni; la presenza della variabile `i`, invece, serve da contatore: può essere utile all'interno del ciclo.

Ovviamente, in tutti questi cicli, l'**if** non deve essere per forza alla fine, ma lo puoi posizionare all'inizio, in mezzo, dove vuoi!

Passiamo al dovuto esempio di codice:

```
#include <iostream>
using namespace std;
int main(){
        const int numero=17;
        int risposta=0;
        cout << "Indovina il numero segreto! Prova con un numero da 1 a 20.
            Inserisci 0 per chiudere il programma." << endl;
        for( ; ; ){
                cout << "Inserisci un tentativo: ";
                cin >> risposta;
                if(risposta == 0)
                        return 0; //chiudo direttamente il programma
                if(risposta == numero)
                        break; //interrompo il ciclo infinito
                cout << "Hai sbagliato, prova di nuovo! " << endl;
        }
        cout << "Bravo, hai indovinato! " << endl;
        return 0;
}
```

Esempio 4.20

Come vedi, posso uscire da un ciclo non solo con l'enunciato **break**, ma anche con un **return** (che, in questo caso, chiude il programma).

L'utilità del ciclo infinito, in questo caso, è che abbiamo due condizioni di uscita di pari importanza. È, in realtà, una pura questione di stile: avremmo potuto scrivere un normalissimo **while** e un ulteriore **if** per la seconda condizione di uscita. Sta sempre a te decidere come comportarti (anche se ci sono occasioni in cui il ciclo infinito è davvero più comodo)!

Dovrebbe essere tutto chiaro, così come il fatto che possiamo complicare il programma inserendo un **if** per controllare che l'input sia effettivamente un numero tra 0 e 20 ed avvertire l'utente in caso contrario.

4.2.6 *switch... case*

Questo costrutto schematizza una serie di **if**. Si presta per risolvere in maniera elegante alcune situazioni particolari.

Ad esempio, se abbiamo una variabile che può assumere valori discreti e per ogni valore vogliamo accada una cosa diversa, lo **switch-case** è perfetto! È un costrutto che si presta benissimo per situazione di "multi selezione".

Ma come è strutturato? In generale è composto così:

```
switch(variabile){
        case valore-intero-1:
                espressione;
                break;
        case valore-intero-2:
                espressione;
                break;
        ...
        ...
        ...
        case valore-intero-n:
                espressione;
                break;
        default:
                espressione;
}
```

Esempio 4.21

Subito dopo **switch** vi è una parentesi tonda, all'interno della quale c'è la variabile che può assumere una serie di valori discreti. Segue una parentesi graffa, in cui abbiamo diversi **case**, uno per ogni valore. Dopo ogni **case**, infatti, vi è il valore, che può essere o un numero intero o una lettera (la quale va racchiusa tra singoli apici, come 'a'). Nota che, quello che segue a **case**, deve essere un valore costante: non può essere un'espressione del tipo `"nuovavariabile/17"` (che dipende dal valore, variabile, di `nuovavariabile`). Il valore viene concluso da due punti, dopo i quali vi è l'espressione che vogliamo venga eseguita. Alla fine vi è un **break**: la sua presenza ci porta alla fine della parentesi graffa. Senza di lui passeremmo al **case** successivo, anche se non è quello desiderato.

Particolare è la presenza del **default**: se la variabile assume un valore diverso da quello di ogni **case**, viene eseguito il codice che segue al **default** (può anche non seguire niente, se non vogliamo venga eseguito qualcosa).

Questa struttura è particolarmente comoda per menù o situazioni di scelta multipla.

Ti ripropongo il programma per stampare a video le lettere che avevo presentato con l'**if**. L'ho scritto utilizzando molti elementi presentati finora: ciclo **for** infinito, **break** e **continue**. Dai un occhio: spero ti sia tutto chiaro!

```
#include <iostream>
using namespace std;
int main(){
        int c=0;
        for( ; ; ){
                cout << "Inserisci un numero da 1 a 5 corrispondente ad una
                    lettera da 'a' a 'e': ";
                cin >> c;
                switch(c){
                        case 1:
                                cout << "a!" << endl;
                                break;
                        case 2:
                                cout << "b!" << endl;
                                break;
                        case 3:
                                cout << "c!" << endl;
                                break;
                        case 4:
                                cout << "d!" << endl;
                                break;
                        case 5:
                                cout << "e!" << endl;
                                break;
                        default:
                                cout << "Hai inserito un numero non ammesso,
                                     inserisci di nuovo!" << endl;
                                continue; //passo alla successiva iterazione
                                     del for: evito il break di uscita
                }
                break; //sono uscito correttamente dallo switch: interrompo
                    il for infinito!
        }
        return 0;
}
```

Esempio 4.22

In questo caso, non sappiamo quante volte l'utente sbaglierà ad inserire il numero: finché questo non è corretto dovremo continuare a chiederglielo. Un ciclo infinito si presta molto bene a questa situazione! Nel caso tutto vada bene, alla fine dello **switch**, vi è un **break** che interrompe il ciclo infinito. Se, invece, le cose non vanno come dovrebbero, nel **default** vi è un **continue** che fa saltare il **break**: si passa all'iterazione successiva!

Avremmo potuto usare, al posto di un ciclo infinito, un **while** con, come condizione, un controllo sul numero inserito: finché rimane diverso da quelli ammessi, continuano le iterazioni. Questi sono modi diversi per scrivere lo stesso programma, ma l'ho detto: a me piace il ciclo **for**...

CAPITOLO 5

Puntatori

Un puntatore è un altro tipo di dato, ma è molto particolare: mentre gli **int** contengono interi, i **float** numeri in virgola mobile e così via, i puntatori contengono *indirizzi di memoria*. Esistono più tipologie di puntatori, uno per ogni tipo di dato già definito. Un altro modo di immaginarli, infatti, è come una variabile che "punta" all'area di memoria occupata da un'altra variabile. Per questo motivo esistono puntatori ad **int**, puntatori a **float**, a **char**, in realtà a qualsiasi tipo di dato esistente (sia quelli standard, che quelli definiti dall'utente; un esempio lo vedrai nel capitolo 9).

Per istanziare una variabile puntatore ad uno specifico tipo di dato è necessario apporre un "$*$" dopo il tipo di dato.

```
int main(){
        int a; //a e' un intero
        int* b; //b e' un puntatore ad intero

        reuturn 0;
}
```

Esempio 5.1

Prima di procedere possiamo farci una domanda: ma quanta memoria occupa un puntatore? Un puntatore ad **int** occupa lo spazio di un **int**, quello di un **char** lo spazio di un **char** e così via? La risposta è assolutamente no.

Per quanto esistano diversi tipi di puntatori (e successivamente cercheremo di capire perché) occupano tutti lo stesso spazio: ovvero quanto è necessario per rappresentare un indirizzo di memoria. In un sistema operativo a 32 bit gli indirizzi di memoria occupano proprio 32 bit (4byte), mentre in un sistema operativo a 64 bit occupano proprio 64 bit (8 byte). Ormai, tutti i computer hanno sistemi operativi a 64 bit, per cui possiamo assumere che i puntatori occupano, di norma, 8 byte.

Se vuoi controllare prova ad eseguire questo:

```
//Dimensione puntatori
```

```
#include <iostream>
using namespace std;
int main(){
        int* a;
        double* b;
        char* c;
        float* d;
        cout << "Dimensione Puntatori:"
                << "ad intero: " << sizeof a << endl
                << "a double: " << sizeof b << endl
                << "a char: " << sizeof c << endl
                << "a float: " << sizeof d << endl;
        return 0;
}
```

Esempio 5.2

Nell'esempio 5.2 ci sono un po' di cose nuove, che non riguardano i puntatori. La prima che noterai è il particolare uso di *cout*: si può "spezzettare" (usarlo così serve per una maggiore leggibilità del codice: tutto sulla stessa riga diventa pesante!).

L'altra cosa nuova è l'operatore "**sizeof**": come puoi immaginare, restituisce la dimensione della variabile alla sua destra. Oltre che sulle variabili, lo puoi utilizzare anche sui tipi di dato, ma sono necessarie le parentesi: "**sizeof**(**int**)". L'esempio 2.2.8 è stato ottenuto proprio con questo operatore.

5.1 L'operatore "&"

Bene, ma come si assegna ad un puntatore un indirizzo di memoria?

Finora, tranne quando ti ho spiegato come sono fatti gli indirizzi di memoria, non ci siamo mai trovati a maneggiarli. Esiste un operatore, "&", la cui funzione è quella di restituire l'indirizzo di memoria della variabile che segue.

```
//Assegnazione di un indirizzo
#include <iostream>
using namespace std;

int main(){
        float n=12;
        float* p;
        cout << "Valore di n: " << n << endl<<"Indirizzo di n: " << &n <<
            endl;

        p=&n;
        cout << "Valore di p: " << p <<endl<< "Indirizzo di p: " << &p <<
            endl;

        return 0;
}
```

Esempio 5.3

Prima di analizzare il codice ritengo significativo dare un occhio all'output:

```
Valore di n:12
Indirizzo di n: 0x7ffd746cc36c
Valore di p: 0x7ffd746cc36c
Indirizzo di p: 0x7ffd746cc360
```

Nel codice ho scritto "`p=&n`", ovvero ho assegnato l'indirizzo di "n" a "p" (il che è lecito, visto che quest'ultimo è un puntatore a float, una variabile che contiene indirizzi di float). Se osservi l'output noterai che indirizzo di "n" e valore di "p" sono proprio uguali.

Nell'ultima riga ho voluto far notare che pure "p" ha un indirizzo, che è diverso dal suo valore. Nulla vieta, dunque, di ipotizzare l'esistenza di un puntatore ad un puntatore... Ma è un altro discorso: avremo modo di analizzare questo fatidico oggetto più avanti, parlando di matrici.

5.2 Dereferenziare i puntatori: l'operatore ∗

Se i puntatori avessero la sola utilità di immagazzinare indirizzi di memoria, sarebbero decisamente di scarso interesse... Ci deve essere un qualche modo per dire "vai a leggere l'area di memoria a cui punti". E in effetti esiste: è l'operatore "∗".

Scrivo un breve codice, per poi trarre spunto nella spiegazione:

```
//operatore *
#include <iostream>
using namespace std;

int main(){
        int* p1;
        int *p2; //anche questa scrittura e' valida

        int a=1;
        p1=&a;

        cout << "valore di a: " << a << endl;
        cout << "Indirizzo di a: " << &a << endl;
        cout << "Valore p1: " << p1 << endl;
        cout << "Valore dell'oggetto puntato da p1: " << *p1 << endl; //sto
            derefenziando il puntatore: vado a leggere l'area di memoria a
            cui punta

        *p1=4; //leggi: scrivi quattro nell'area di memoria puntata da p1
        cout << "Valore di a: " << a << endl;

        return 0;
}
```

Esempio 5.4

Dove l'output è:

```
valore di a: 1
Indirizzo di a: 0x7ffd77a24f64
Valore p1: 0x7ffd77a24f64
Valore dell'oggetto puntato da p1: 1
Valore di a: 4
```

Analizziamo punto per punto:

- **int *p2** Anche questa scrittura è corretta. Mettere l'asterisco vicino alla variabile o al tipo di puntatore è una questione, quasi sempre, di gusto. Forse la scrittura "**int*** p" rende più chiara una cosa: così come in "**int** n" **int** è il tipo di dato mentre "n" è la variabile vera e propria, in un puntatore il tipo di dato è "**int***" e "p" è la variabile. L'asterisco fa parte del tipo di dato, non della variabile. C'è solo un caso in cui si è costretti a porre l'asterisco vicino alla variabile e non al tipo di dato:

  int *p1, *p2, *p3;

  Ovvero, quando sulla stessa riga dichiariamo più puntatori: bisogna esplicitare che anche p2 e p3 sono puntatori ad intero. La scrittura "**int*** p1, p2, p3;" fa si che vengano istanziati, invece, un puntatore ad intero e due interi (rispettivamente p1 e p2,p3).

- **p1=&a** Come già visto, stiamo assegnando l'indirizzo di `a` al puntatore `p1`, e questo dovrebbe rendere chiaro che la variabile è proprio "`p1`" e non `*p1` (abbiamo sulla sinistra un puntatore ad intero, sulla destra un indirizzo di un intero: stesso tipo di dato, assegnazione lecita).

- **cout « *p1** Ecco qui l'altro uso fondamentale di "*": quello di operatore per dereferenziare un puntatore. Spiego subito i paroloni. L'operatore "*", posto prima di una variabile puntatore, significa "accedi all'area di memoria puntata". Essenzialmente stiamo leggendo l'area di memoria riservata ad "int a", la variabile iniziale; *p, ora, *è* la variabile iniziale ("*p", non "p"!), è quella stessa identica cella di memoria, nulla di differente.

  Ora, dovrebbe essere chiaro perché il tipo di dato non è semplicemente ***** bensì è **int***: utilizzando l'operatore di dereferenziazione, come potrebbe sapere il computer *quanta* memoria andare a leggere? Ricordati: ogni byte della memoria ha un indirizzo, e se un puntatore contiene un indirizzo, quest'ultimo è quello del *primo* byte della variabile puntata. Ma, come ben sai, spesso le variabili occupano ben più di un byte: gli int, ad esempio, ne occupano quattro. Per cui, se "p" è un puntatore ad interi e scriviamo "cout << *p << endl" il computer sa che deve andare all'indirizzo contenuto in "p" e, dato che punta ad un intero, deve leggere quattro byte. Se "p" fosse stato un puntatore a double, il computer avrebbe letto otto byte, e così via.

- ***p1=4** Cosa succede? Ormai l'avrai capito meglio di me: il computer sta scrivendo il valore "4" nell'area di memoria puntata da "p1". Non deve stupire, quindi, che quando successivamente stampiamo il valore di "a" quest'ultimo è proprio "4".

È chiaro, quindi, che i puntatori possono essere "a qualsiasi tipo di dato": esistono sia puntatori ai tipi di dato standard, che a tipi di dato definiti dal programmatore (ne vedremo un esempio nel capitolo sulle struct); esistono perfino i puntatori a puntatore (alla fine anche lui è una variabile che occupa un'area di memoria ma, come accennato, ne riparleremo prossimamente...); ecco alcuni esempi:

```
....
char* c;
float* f;
bool* b;
int** i; //puntatore a puntatore ad int, fai finta di non aver visto niente,
     li studieremo poi...
....
```

Esempio 5.5

Rimane un'ultima cosa da dire: e come si inizializza un puntatore a "zero"? Come si dice "non puntare da nessuna parte"? È semplice: si utilizza la parola chiave "NULL". Un esempio:

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

int main(){
        int a=1;
        int* p=&a;
        int* p2=NULL; //p2 inizializzato a puntare ''da nessuna parte''
        *p=2; //cambio valore in a
        p=NULL; //ora p non punta da nessuna parte
        cout << *p << endl; //provo ad accedere all'area di memoria puntata
            ora, cosa succede?

        return 0;
}
```

Esempio 5.6

L'unico commento necessario è che, dopo aver assegnato un puntatore a "NULL", cercare di accedere all'area di memoria puntata non ha alcun senso (come accedo al niente?): se provi ad eseguire il codice noterai, infatti, che il programma va in segmentation fault.

I puntatori, per ora, ti possono sembrare uno strumento astratto e poco utile, ma vedremo che la loro potenza è ben maggiore del poter semplicemente avere più variabili che si riferiscono allo stesso indirizzo di memoria. Essi mostrano tutta la loro forza con gli *array* e l'*allocazione dinamica* (capitolo 6) e con le funzioni (capitolo 10).

CAPITOLO 6

Array e Matrici

Dobbiamo scrivere un programma per analizzare i dati di un esperimento: stiamo misurando una grandezza e per questioni di statistica vogliamo ripetere la misura mille volte, quindi procedere all'analisi dati. Il nostro programma dovrà gestire mille dati. Con le conoscenze informatiche acquisite finora dovremmo scrivere qualcosa di questo tipo:

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

int main(){
        float a, b, c,...; //Mille float
        //Per non parlare poi del riempimento dei dati:
        a=2.1;
        b=5.3;
        ...
        ...

        //Fosse necessaria la media avremmo bisogno di:
        float media=a+b+c+.......;
        media/=1000.;

        return 0;
}
```

Esempio 6.1

In poche parole qualcosa di impossibile, o se non altro completamente inutile (allora tanto vale usare carta e penna...).

Se cerchiamo sul dizionario la parola "array" troviamo traduzioni del tipo: "schieramento", "insieme", "disposizione". Un insieme? Schieramento? I nostri mille dati, tutti dello stesso tipo, cos'altro sono se non proprio questo?

Ecco, un *array* è uno "schieramento" di dati tutti dello stesso tipo: in poche parole, un "vettore". Nel capitolo sulla rappresentazione in memoria (capitolo 3) ho presentato una breve

introduzione, orientata alla loro struttura nella memoria; in ogni caso, facciamo un breve riassunto dei concetti presentati.

Un *array* è un insieme contiguo in memoria di elementi dello stesso tipo. Cosa vuol dire "contiguo"? Nella memoria gli elementi sono "uno dopo l'altro", non sparsi ma ben ordinati. Se abbiamo un *array* di dieci interi, il compilatore ci assicura che essi occuperanno un blocco di memoria unitario, in cui il secondo elemento segue immediatamente il primo e così via.

6.1 Array statici

Il primo tipo di *array* che incontriamo è l'*array statico*. Un *array* statico è una collezione di elementi dello stesso tipo, la cui dimensione è definita a priori nel codice e che non può cambiare a "run-time" (ovvero durante l'esecuzione del programma).

Come al solito, partendo da un esempio di codice, cerchiamo di capire come funzionano gli *array* statici. Il programma 6.1 potrebbe assumere la seguente forma con gli *array*:

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

int main(){
        float dati[1000]; //Dichiaro un array di float di 1000 elementi


        for(int i=0; i<1000; ++i){
                dati[i]=... //un qualche codice che riempie i dati. Nel
                    capitolo su In/out impareremo a leggere da file, i dati
                    dell'esperimento potrebbero essere in un file
        }

        float media=0;
        for(int i=0; i<1000; ++i)
                media+=dati[i];

        media/=1000;
        cout << "Media: " << media << endl;

        return 0;
}
```

Esempio 6.2

Prendendo spunto da questo codice, vediamo le caratteristiche principali degli *array*.

6.1.1 Dichiarare array

Nella prima riga del *main* vi è "**float** dati[1000]": stiamo dichiarando un *array* di **float** di mille elementi. Il nome della variabile *array* di **float** è "`dati`".

Un array può essere di qualsiasi tipo di dato: **float**, **double**, **char**, **bool**, **int**, ecc... Il numero di elementi è racchiuso nelle parentesi quadre che seguono il nome dell'*array*; è importantissimo notare che questo numero deve essere definito a "compile-time": il compilatore deve sapere quanto sarà grande l'*array*, per cui tra le quadre non possiamo mettere variabili che vengono modificate durante l'esecuzione del programma. Sono ammessi, invece: macro,

variabili **const** (entrambi non si modificano durante l'esecuzione) o semplicemente numeri. Perché dovremmo usare una macro o una variabile **const**? Per comodità: poniamo di sapere che il nostro programma lo compileremo più volte, cambiando sempre il numero di dati che vogliamo analizzare. Se ogni volta dobbiamo modificare il "1000" in tutti i punti del codice in cui compare, diventa assai laborioso. Definendo una macro o una variabile **const** la questione si semplifica:

```
#include <iostream>
//definisco una macro
#define N 100
using namespace std;
//Alla riga della macro e' equivalente la seguente
//const int N=100; e, come spiegato nel relativo capitolo, ha i suoi
    vantaggi

int main(){
        float dati[N]; //N non cambiera' mai nell'esecuzione del programma,
            sia che si usi la macro sia che si usi il const int, quindi e'
            accettabile per un array statico

        for(int i=0; i<N; ++i)
                ...
        ...
        ...
        return 0;
}
```

Esempio 6.3

Ogni volta che vogliamo ricompilare il nostro programma cambiando la dimensione dell'*array*, ci basta cambiare la macro (o variabile **const**): di conseguenza cambierà la dimensione dell'*array*, i cicli **for**, e tutti i punti in cui N ci serve, senza dover modificare ulteriormente il codice.

Riassumendo:

```
#include <iostream>
#define N 10
int main(){
        const int n=25;
        bool b[n];      //array di 25 elementi di bool
        float f[N];     //array di 10 float
        double d[50];   //array di 50 double
        int dim=...     //codice che a run time determina il valore di dim
        int array[dim]; //ERRORE! La dimensione non puo' essere determinata
            a run time
        return 0;
}
```

Esempio 6.4

Gli array vanno inizializzati! Quando dichiariamo un *array* il compilatore ci assicura soltanto di riservare un'area contigua di memoria. Quest'ultima, però, non è inizializzata. Ti

ricordi? C'è dentro "sporcizia", valori a caso. Per cui, quando dichiari un *array*, ricordati di inizializzarlo: riempilo dei valori che ti servono, o utilizza un ciclo **for** per, ad esempio, inizializzare tutti gli elementi a zero.

6.1.2 Indici e... out of range!

Scrivendo **dati[i]=...** stiamo richiamando l'i-esimo elemento del nostro *array*. Nota una cosa e imprimitela benissimo in testa: negli esempi di codice, l'indice del ciclo **for** parte da zero. Questo non è un caso: l'indicizzazzione degli *array* parte proprio da zero! Il primo elemento è "`dati[0]`", il secondo è "`dati[1]`' e, se N è la dimensione dell'*array*, l'ultimo elemento è "`dati[N-1]`. E se scrivo "`dati[N]`"? Semplicemente "esco dall'*array*". Stai molto attento: nel `C++` non vi è alcun controllo sui limiti degli indici, né a run-time, né tanto meno a compile-time. Per cui cosa accade? L'*array* è un'area contigua di elementi di memoria, e scrivere "`array[x]`" mi fa accedere ad un elemento, ma se sforo l'indice il programma si sposta oltre: accede alle aree di memoria successive. C'è un problema: queste aree di memoria non sono dell'*array* e, spesso, non sono neanche del programma. Sforare i limiti dell'*array* ti fa leggere dati senza alcun senso, o ti fa scrivere in zone non tue: se non sono neanche del programma il sistema operativo ti "uccide"[1], cioè molto semplicemente non ti lascia scrivere nella memoria che non ti appartiene e il programma va in "segmentation fault". In lettura, invece, il sistema operativo è più tollerante, e questo è un problema: vai a leggere dati assolutamente senza senso, e i risultati saranno ancora meno sensati (e non ti accorgerai dell'errore perché nulla ti fermerà o ti avviserà!).

In conclusione stai molto attento agli indici, perché puoi scrivere stupidaggini immense e il compilatore non si lamenterò, ma i risultati non saranno piacevoli.

6.1.3 Puntatori e Array

Quando dichiariamo un *array* scriviamo "**float** dati[N]" e diciamo che dati è un *array* di N **float**. Ma quindi la variabile "`dati`" cos'è? Finora tutte le variabili potevamo stamparle a video; siamo curiosi e vogliamo provare a farlo anche con una variabile di "tipo *array*". Verrà stampato tutto il contenuto? Cosa succederà? Presi dall'interesse scriviamo questo codice:

```
#include <iostream>
using namespace std;
int main(){
        int array[10];
        cout << "Contenuto dell'array?: " << array << endl;
        return 0;
}
```

Esempio 6.5

Eseguiamo e...

```
Contenuto dell'array?: 0x7ffc3cb3e290
```

[1] Almeno per quanto riguarda i sistemi Unix.

Un indirizzo di memoria? Ma quindi un *array* è un puntatore? Ecco: essenzialmente si[2].

Dunque, cerchiamo di rimettere assieme i pezzi. Un puntatore ad **int** è una variabile che contiene un indirizzo di memoria; se lo dereferenziamo il computer va a quell'indirizzo di memoria e legge quattro byte (il contenuto della celletta puntata e delle tre successive) perché sa che un **int** ha quella dimensione.

Infine, sappiamo che un *array* è un insieme di elementi dello stesso tipo, rappresentati in memoria da celle contigue.

Se abbiamo un *array* di 10 **int** ci basta sapere l'indirizzo di memoria del byte iniziale del primo dato; individuato questo, per accedere agli elementi successivi basta spostarsi di 4 byte per il secondo dato, di altri 4 byte per il terzo e così via. È esattamente quello che fa il computer: di un *array* conosce solo l'indirizzo di memoria del primo byte, ma sapendo il tipo di dato, sa esattamente come leggere il primo elemento e i successivi.

Riassumendo, per accedere al N-esimo dato dell'*array* (con N che parte da zero), se X è la dimensione in byte del tipo di dato, il meccanismo è il seguente: partendo dal primo byte dell'*array* (puntato dal puntatore) mi sposto di $N \cdot X$ byte e leggo X byte.

Per convincerti del concetto guarda questo codice:

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

int main(){
        int array[5];

        cout << "Array:\t\t\t\t" << array << endl;
        for(int i=0; i<5; ++i)
                cout << "Indirizzo " << i+1 << "-esimo elemento:\t" << &(
                    array[i]) << endl;

        return 0;
}
```

Esempio 6.6

Da cui si ottiene il seguente output:

```
Array:                          0x7ffc8a1a7b40
Indirizzo 1-esimo elemento:     0x7ffc8a1a7b40
Indirizzo 2-esimo elemento:     0x7ffc8a1a7b44
Indirizzo 3-esimo elemento:     0x7ffc8a1a7b48
Indirizzo 4-esimo elemento:     0x7ffc8a1a7b4c
Indirizzo 5-esimo elemento:     0x7ffc8a1a7b50
```

Se parti dal primo indirizzo e sommi quattro byte ottieni il secondo e così via.

[2] In realtà è un puntatore in "sola lettura": a differenza di un puntatore normale non puoi modificarlo (cambiare l'indirizzo di memoria a cui punta).

Concludendo, dunque, possiamo dire che la scrittura "**int** array[N]" istruisce il compilatore a riservare una zona di memoria contigua di N **int** (o del tipo di dato in questione), e ad allocare un puntatore ad **int** (la variabile *array*) riempiendolo con l'indirizzo del primo byte dell'area di memoria. Quando nel codice usiamo le parentesi quadre, tipo "array[2]" queste fungono da operatore di dereferenziazione del puntatore "array+2" (parti dal puntatore iniziale e spostati di due puntatori ad **int**, quindi di otto byte): leggi il contenuto del puntatore "array+2". Scrivere "*array" e "array[0]" a rigore è equivalente. Se sei curioso prova: esiste anche un'aritmetica dei puntatori, per cui puoi scrivere cose come "*(array+3)" che equivale a "array[3]" (accedi al quarto elemento dell'array), ma ovviamente la seconda scrittura è estremamente più chiara e comoda...

Per capire meglio la questione, ti propongo un esercizio: alloca un *array* statico, riempilo e poi usa dei puntatori per accedere agli elementi. Prova anche ad utilizzare le parentesi quadre con i puntatori normali, insomma, giocaci un po'!

6.2 Matrici: array di array

Una matrice è un oggetto che ti viene presentato in maniera formale nel corso di Geometria.

Nei termini più banali possibili è una tabella di numeri. Ecco una generica matrice M di dimensione m x n:

$$M = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix}$$

Dove i vari x_{mn} sono numeri. Un esempio più pratico potrebbe essere la seguente matrice di interi di dimensioni 2x3:

$$A = \begin{bmatrix} 1 & 5 & -2 \\ -5 & 0 & 1 \end{bmatrix}$$

Vedere le matrici come "tabelle", per i nostri scopi può essere piuttosto scomodo. È più conveniente immaginarle come "*array* di *array*" (*array* bidimensionali). La matrice A, ad esempio, può essere vista come due *array* di tre **int** ciascuno, oppure come tre *array* di due **int** ciascuno.

Ma a cosa può servire un "*array* di *array*"? In Fisica, ad esempio, ci sono miriadi di situazioni potenzialmente interessate. Pensiamo un insieme di n particelle, ognuna delle quali descritta da m parametri. Bene, per rappresentare in memoria questo sistema è conveniente utilizzare n *array* di m **float** (o **double**, o **int**, o quello che serve).

Dovremmo, quindi, scrivere qualcosa di questo tipo:

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

const int n=100;
int main(){
        float v1[n];
        float v2[n];
        float v3[n];
        ...
        ...
```

```
        float vm[n]; //dobbiamo dichiarare m array di float

        return 0;
}
```

Esempio 6.7

Se m diventa molto grande il nostro codice perde di funzionalità: è una situazione molto simile a quella dell'esempio 6.1 che ci ha portato a definire l'*array*.

L'esempio 6.7 può essere risolto così:

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

int main(){
        const int n=100;
        const int m=50;

        float matrix[m][n];

        return 0;
}
```

Esempio 6.8

Cos'è "**float** matrix[m][n]"? Un *array* di m elementi, ognuno dei quali è un *array* di n float. Altro modo per vederlo, è intenderlo come una matrice m x n di **float**.

Come accediamo agli elementi? Come si modificano? Cicli **for** alla mano, cerchiamo di fare ordine mentale (in realtà non è nulla di nuovo rispetto a quanto già visto, solo che abbiamo due dimensioni al posto di una):

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

int main(){
        const int m=3;
        const int n=4;

        int matrix[m][n];
        for(int i=0; i<m; ++i){ //scorro sull'array di array, se i=0 sono
            sul primo array, se vale 1 sul secondo e cosi via
                for(int j=0; j<n;++j) //Scorro sugli elementi di ogni array,
                     nota: uso la variabile j (diversa da i)! ricordi il
                    discorso degli scope? i e' visibile qua dentro?
                        matrix[i][j]=i*n+j; //riempio
        }

        //ora voglio stampare come fosse una tabella, una matrice di m righe
             e n colonne
        for(int i=0; i<m; ++i){
                for(int j=0; j<n; ++j)
                        cout << matrix[i][j]<<"\t"; //spazio di tab tra
                            elementi
```

```
            cout << endl; // mando a capo alla fine di ogni riga
        }
        return 0;
}
```

Esempio 6.9

L'output di questo programma è seguente:

```
0       1       2       3
4       5       6       7
8       9       10      11
```

Come vedi, una volta che si pensa la matrice come un *array* di *array*, non cambia molto da quanto abbiamo già visto: bisogna solo tenere presente che vi sono due indici, ricordarsi cosa rappresenta ciascun indice e stare attenti a non confondere le variabili iterative dei cicli **for** incapsulati.

6.2.1 Rappresentazione della matrice in memoria

Ma in memoria, la nostra matrice come è fatta? Abbiamo visto che la memoria è una sorta di "striscia" continua di bit. Alla linearità della memoria non si scappa, e anche la matrice, sebbene abbia più dimensioni, è immagazzinata in una striscia continua e lineare di bit.

Dovrebbe essere chiaro, quindi, che l'uso dell'*array* multidimensionale è semplicemente un'astrazione per il programmatore ma, in memoria, una matrice è un semplice *array* di lunghezza $n \cdot m$.

Se abbiamo **int** matrix[10][50], il computer riserva una striscia di memoria di 500 interi; scrivendo matrix[3][0] il compilatore sa che deve spostarsi di $50 \cdot 4 \cdot 4$ byte (50 è la dimensione di ogni "colonna", quindi stiamo chiedendo di accedere alla quarta riga e ogni elemento occupa quattro byte, perché è un **int**). Potenzialmente potremmo usare solo vettori e farci noi il ragionamento di "spostarci" di **m** elementi quando vogliamo passare da una riga all'altra, ma sicuramente quest'astrazione è comoda. Vedrai che per le matrici dinamiche il discorso cambia un po'.

6.2.2 Un esempio concreto sui pericoli dell'out of range

Come ho accennato, non sempre quando si esce dai limiti degli *array* il sistema operativo uccide il programma; a volte, semplicemente, si va a scrivere in aree di memoria che appartengono al programma (ma non all'*array*!), il sistema operativo lo lascia fare e accadono cose poco piacevoli.

Ti propongo un esempio concreto, per capire bene l'importanza di questo argomento.

Il programma che segue è composto in questo modo: ho un *array* di "$n+1$" dati, su questi vengono eseguiti dei conti (matematici, di fisica, non ci importa: nel codice che ho scritto non ci sono) per poi riempire una matrice "n x n" di risultati. Lanciando il programma ci

accorgiamo che i risultati non hanno alcun senso; per cercare di capire perché decidiamo di stampare a video il contenuto del vettore di dati.

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

int main(){
        const int n=6;
        int v[n+1];      //vettore di dati
        int matr[n][n]; //matrice di risultati

        for(int i=0; i<=n; ++i)
                v[i]=i; //inizializzo l'array al valore che ci serve, nell'
                    esempio uso "i" giusto per riempirlo di qualcosa

        //Altro codice...
        //...

        for(int i=0; i<=n; i++){
                for(int j=0; j<=n; ++j)
                        matr[i][j]=0; //inizializzo gli elementi della
                            matrice tutti a zero, saranno i calcoli del
                            programma a riempirla
        }

        //Calcoli vari del programma...
        //...
        //...

        //Qualcosa non funziona!
        //Per capirci di piu' controllo cosa c'e' nel vettore iniziale
        for(int i=0; i<=n; ++i)
                cout << v[i] << endl;

        return 0;
}
```

Esempio 6.10

L'output con il contenuto di "v" è il seguente:

```
0
0
0
0
0
0
0
```

Com'è possibile? Dovrebbe essere pieno di 0,1,2... Che è successo?

Dopo aver sbattuto la testa a lungo, senza venirne a capo, ci accorgiamo che la matrice è di dimensione n x n mentre noi nel ciclo **for** stiamo riempiendo fino all"n+1" esimo elemento. Possibile che questa sia la causa di tutto? Ci viene un enorme sospetto: andando a scrivere nell'n+1 esimo vettore della matrice, stiamo uscendo dalla sua area di memoria. Forse abbiamo scritto nell'area di memoria di `v`?

Per rispondere alla domanda inseriamo il seguente codice nel programma:

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

int main(){
        const int n=6;
        int v[n+1];      //vettore di dati
        int matr[n][n]; //matrice di risultati

        for(int i=0; i<=n; ++i)
                v[i]=i; //inizializzo l'array al valore che ci serve, nell'
                    esempio uso "i" giusto per riempirlo di qualcosa

        //Altro codice...
        //...

        for(int i=0; i<=n; i++){
                for(int j=0; j<=n; ++j)
                        matr[i][j]=0; //inizializzo gli elementi della
                            matrice tutti a zero, saranno i calcoli del
                            programma a riempirla
        }

        //CODICE di controllo, che succede alle aree di memoria?
        cout << "Indirizzo v: \t\t" << v << endl; //stampo l'indirizzo del
            primo elemento del vettore v
        cout << "Indirizzo matr[n-1]: \t"<< matr[n-1] << endl; //matr[n-1] e
            ' un vettore (puntatore), l'n-esimo vettore: stampo l'indirizzo
            dell'area di memoria del primo elemento dell'n-esimo vettore
        cout << "Indirizzo matr[n]:\t" << matr[n] << endl; //Indirizzo "out
            of range"

        //...
        //...

        return 0;
}
```

Esempio 6.11

L'output, sul mio computer, è il seguente:

```
Indirizzo v:            0x7fff65f443a0
Indirizzo matr[n-1]:    0x7fff65f44388
Indirizzo matr[n]:      0x7fff65f443a0
```

Cos'è successo? Come vedi, "`matr[n]`" (su cui andiamo a scrivere per errore: non appartiene alla matrice!) ha lo stesso indirizzo di memoria di `v`. È chiaro, quindi, che nel ciclo **for** in cui inizializziamo a zero la matrice, stiamo erroneamente scrivendo sul vettore v, cancellando i dati originali. Abbiamo commesso un errore con gli indici ma, "sfortunatamente", non usciamo dalla memoria del programma e il sistema operativo non si lamenta.

In un programma grande, in cui la dichiarazione delle variabili e la loro inizializzazione è molto lontana nel codice, accorgersi dell'errore degli indici potrebbe essere difficile, lungo ed estenuante (apparentemente sembra che la scrittura nel vettore `v` non abbia funzionato). Stai sempre molto attento: possono succedere cose davvero spiacevoli.

6.3 Operazioni su array: qualche algoritmo

È arrivato il momento di addentrarci (o meglio: di sfiorare!) in quella parte dell'informatica che si occupa di algoritmi. In questa sezione non impareremo nessuno strumento del linguaggio `C++`, ma solo alcune applicazioni. Se vogliamo svolgere un'operazione, un algoritmo non è altro che l'"istruzione" da dare al computer per eseguirla; il linguaggio, invece, è solo una possibile lingua con cui scrivere quest'istruzione. Vediamo alcuni casi interessanti che riguardano gli *array*.

Un *array* è un insieme di elementi e, su questi, possiamo essere interessati a svolgere calcoli e operazioni. Se abbiamo *array* di numeri, ad esempio, possiamo voler trovare la media, la deviazione standard, la varianza, ecc... Queste problematiche non hanno nulla di particolarmente complesso sul piano dell'informatica, sono formule che hai imparato nei vari corsi di statistica e laboratorio, per cui li lascio a te.

Ci sono alcune operazioni, invece, proprie di un vettore: pensaci, hai un insieme di numeri, cosa può venirti voglia di fare? Ordinare il vettore può essere molto utile!

Il riordino di un *array* di dati è un argomento molto più complesso di quello che sembra: esistono tantissimi algoritmi e noi cercheremo di capirne uno molto semplice. Prima, però, partiamo dalle basi: come si trova l'elemento massimo (o il minimo) di un *array*?

6.3.1 Elemento massimo e minimo

Il nostro vettore ha n elementi, disposti casualmente. Prendiamo il primo elemento, ce lo "appuntiamo" e lo confrontiamo con i successivi; se ne troviamo uno maggiore, lasciamo perdere il primo elemento, ci salviamo quest'altro elemento e lo confrontiamo con quelli che seguono. Procediamo così fino alla fine del vettore. Arrivati all'ultimo elemento saremo certi che quello che ci siamo salvati sarà maggiore di tutti.

Il codice è semplice, "**int** array[n]" è un vettore di n elementi interi disposti casualmente:

```
int array[n];
int tmp=0; //variabile temporanea di appoggio
...
...
tmp=array[0]; //salvo il primo elemento per confrontarlo con i
    successivi
for(int i=1; i<n; ++i){//nota: i parte da 1, perche'?
        if(array[i]>tmp)
                tmp=array[i];
}
```

Esempio 6.12

Alla fine, la variabile "`tmp`" conterrà l'elemento massimo.

Prova a scrivere il codice per trovare l'elemento minimo, a questo punto dovrebbe essere estremamente facile.

Ti invito a notare una cosa (che tornerà utile quando parleremo di **struct**): affinché abbia senso parlare di elementi minimi e massimi è necessario che sul tipo di dato dell'*array* sia definito il concetto di ordinamento, sia a livello astratto (cosa vuol dire ordinare pere e banane?) sia a livello concreto (quando scriviamo "array[i]>tmp" il computer *sa* cosa significa?).

6.3.2 Ordinamento

Il problema dell'ordinamento di un vettore è estremamente importante in moltissimi campi dell'informatica applicata: Fisica, Matematica, economia, database di vario tipo, social network, ecc...

Per quanto sembri banale, ordinare un vettore è computazionalmente pesante: un cattivo algoritmo può essere molto più lento di uno buono, e costare secondi di esecuzione in più.

L'algoritmo che andremo ad analizzare è un algoritmo abbastanza cattivo, ma è molto semplice da capire e da implementare: il *selection sort*. Il numero di operazioni da effettuare, per ordinare completamente un vettore, è asintotico ad N^2, dove N è il numero di elementi. Ci sono algoritmi, come il *quick sort*, che nelle situazioni migliori sono asintotici a $N\ln(n)$. Nel corso viene riportato il codice del *quick sort*, ma è complicato da implementare e non è richiesto per superare l'esame, per cui in queste note lo tralasciamo.

Selection Sort

Ho voluto introdurre la ricerca di elementi massimi e minimi perché il *selection sort* si basa esattamente su questo procedimento. Si cerca l'elemento minimo e lo si mette al primo posto, quindi si parte dalla seconda posizione dell'*array* e da lì si cerca di nuovo l'elemento minimo e lo si sposta al secondo posto, quindi si passa alla terza posizione, ecc...

Come vedi è molto semplice, ma il numero di operazioni è notevole (per ogni posizione bisogna scorrere il vettore fino alla fine). Vediamo il codice: al solito, "**int** array[n]" è un *array* di n **int** casuali. L'unica vera "difficoltà" sta nel non perdersi per strada gli elementi del vettore durante gli scambi.

```
int array[n];
int tmp=0; //variabile di appoggio
int index_min=0; //variabile di appoggio per gli indici del vettore

...
...

for(int i=0; i<n-1; ++i) { //perche' minore di n-1?
        min = array[i];
        index_min=i; //index_min tiene traccia della posizione del
            minimo
```

```
                for(int j=i+1; j<n; ++i){ //perche' j=i+1? (ricorda il
                    codice della ricerca del minimo)
                        if(array[j] < min) {
                                index_min=j; //index_min tiene traccia del
                                    nuovo minimo
                                min=array[j]; //min e' uguale al nuovo
                                    minimo
                        }
                } //uscito da questo for ho trovato il minimo da mettere
                    nella i-esima posizione
                //ora devo scambiare l'elemento minimo con quello all'i-
                    esima posizione
                tmp=array[i]; //mi salvo l'elemento contenuto nell'i-esima
                    posizione
                array[i]=array[index_min]; //sposto nell'i-esima posizione
                    il minimo
                array[index_min]=tmp;

        } //passo alla posizione successiva
```

Esempio 6.13

Domanda: se il mio *array* è un *array* di **float** (o un altro tipo di dato) chi tra `min`, `index_min` e `tmp` deve essere dichiarato come **float**? E chi no?

Come esercizio scrivi il codice per ordinare **float** e **double**, quindi riscrivi tutto per ordinare in modo decrescente.

CAPITOLO 7

In/Out: i file

Abbiamo visto nel primo capitolo che un computer è dotato di dispositivi di input e output. Una possibilità che ci è garantita dal sistema operativo è di comunicare con essi.

Prova ad immaginare il dispositivo di input, ad esempio, come un tuo amico lontano con cui comunicare; per farlo hai bisogno di aprire un canale di comunicazione, come telefono, mail, chat, qualcos'altro. Un programma fa la stessa cosa: per comunicare con un dispositivo di in/out deve aprire un canale di comunicazione, uno *stream*.

Uno *stream* è un *oggetto*[1] piuttosto complicato, con tante belle proprietà e caratteristiche; ne vedremo giusto le funzioni base. Una piccola nota: questo è l'unico capitolo in cui useremo strumenti propri del `C++`. Nel `C`, infatti, non esiste un'entità assimilabile allo *stream* (la comunicazione con i dispositivi di in/out avviene sfruttando direttamente le librerie di sistema).

Un canale deve essere dichiarato, "costruito", "testato" e quindi usato (lo impareremo nella sezione 7.2). Esistono, però, alcuni canali che sono già aperti e testati, pronti ad essere usati. Li abbiamo già ampiamente sfruttati, per cui facciamo giusto un breve ripasso; quindi, affronteremo la comunicazione con i file, che è il vero obiettivo di questo capitolo.

7.1 Canali standard

Ci sono due cose che in un computer non possono mancare: la tastiera e lo schermo (o dei loro sostituiti: quando mi connetto via *ssh* cosa uso...?). Includendo la libreria "*iostream*", tre canali ci vengono automaticamente aperti e dati a disposizione: *cout*, *cin* e *cerr*. I primi due sono stati rapidamente analizzati in 2.7.

Cerr, invece, sta per "console error". È un canale di output che stampa sempre su terminale. L'unica differenza rispetto a *cout* è che non è bufferizzato. Per dirla in soldoni: usandolo per piccole stringhe, il sistema garantisce che, quando tutto si inchioda, lui dovrebbe continuare a

[1] L'essenza di questo concetto, per essere compresa appieno, avrebbe bisogno di strumenti che vanno oltre il contenuto di queste note. Quando l'anno prossimo studierai le *classi*, diversi strumenti di questo capitolo ti appariranno molto più chiari.

funzionare (dovrebbe... dipende da quanto sei bravo a mandare in crisi il computer!); in poche parole è da usare per stampare messaggi d'errore. Ma puoi anche usare *cout*, l'uso di *cerr* è una finezza. Si adopera allo stesso modo: cerr << stringa << endl;.

Bisogna sottolineare una problematica dei canali di input, a cui *cin* non si sottrae. Se `a` è un intero e noi scriviamo cin >> a; il contenuto di *cin* viene "buttato" in `a`. Ma se il contenuto di *cin* non c'entra molto con il tipo di dato di `a` (ad esempio scrivi "sadckjasndjcnd" e premi invio), cosa accade? Succede una cosa poco piacevole: lo *stream* si "rompe".

Un'ultima avvertenza, questa volta su *cout*: per quanto il discorso delle performance vada oltre lo scopo di queste note, sappi che *cout* ha un costo computazionale non indifferente. Se in un ciclo **for** stampi a video tutto quello che succede, il tempo di esecuzione crescerà inutilmente. Per provare, ho appena eseguito un programma che semplicemente incrementa una variabile in un ciclo **for** dieci milioni di volte: senza stampare a video il valore della variabile ad ogni ciclo, il tempo di esecuzione è stato di 0.03 secondi; stampando "Il valore di j è: `valore`" il tempo è risultato essere di 32.23 secondi. Tempo buttato via!

7.2 Comunicazione con i file

Cos'è un file? Un file non è altro che un insieme di dati che, però, vive sull'hard disk e non sulla RAM: i file sono permanenti, non muoiono dopo un riavvio.

Il canale che è in grado di comunicare con un file è "*fstream*" (file stream). Costui, poi, ha due "figli": "*ifstream*" (per leggere file: input file stream) e "*ofstream*" (per scrivere su file: output file stream). Per disporre di questi strumenti dobbiamo includere la libreria che li contiene con "#**include** <fstream>".

7.2.1 Scrivere su file

Partiamo da un esempio concreto; nel codice che segue **int** vec[10] è un vettore di interi che vogliamo scrivere su un file di nome `dati.txt`:

```
#include <iostream>
#include <fstream>      //ofstream
#include <cstdlib>      //exit()
using namespace std;

int main(){
        int vec[10];
        //Riempio vec in qualche modo...
        //...

        ofstream out; //dichiaro lo stream out
        out.open("dati.txt"); //apro il canale verso il file dati.txt (che
            crea, non esistendo)
        if(out.fail()){ //controllo che l'apertura sia andata a buon fine
                cerr << "Apertura del canale di output fallita!" << endl;
                exit(1); //se e' fallita esco dal programma
        }

        for(int i=0; i<10; ++i)
```

```
                out << vec[i] << endl; //nota uso out e non cout!! out e' il
                    canale verso il file!
        out.close(); //chiudo il canale
        return 0;
}
```

Esempio 7.1

Come vedi, per prima cosa dichiaro un oggetto di tipo "*ofstream*". *Ofstream* è un dato un po' particolare, dotato di membri che sono funzioni. Per accedere ai membri di un dato si scrive `nomedato.membro`. Ad esempio, `open()` è una funzione[2] che, come argomento (ciò che è tra le parentesi) vuole una stringa[3]: il nome del file verso cui aprire il canale di comunicazione[4]. Volendo, puoi fare un passaggio in meno ed esplicitare il file verso cui aprire il canale direttamente nella dichiarazione della variabile: ofstream out("dati.txt").

Nella riga successiva segue un **if**. Questo if-statement viene trascurato da buona parte degli studenti, ma è fondamentale! L'apertura di un canale verso un file non sempre ha successo, vi sono molte ragioni per cui potrebbe fallire. È buona norma, prima di usare un canale, controllare che sia funzionante; il metodo `fail()` (out. fail ()) restituisce una variabile booleana: **true** se il canale è "rotto", **false** se è funzionante. Se il canale è "rotto" entriamo nell'**if**: qui ho stampato, usando *cerr*, un messaggio di errore per avvisare l'utente di ciò che è avvenuto. Quindi, tramite la funzione `exit(1)`, interrompo il programma. Exit è contenuta nella *C standard library* (#**include** <cstdlib>); vuole tra le parentesi un numero da restituire al sistema operativo (puoi metterne uno qualsiasi, ora come ora...), allo stesso modo del "return 0" del *main*. La differenza tra exit e return è che la prima interrompe il programma in qualsiasi punto essa venga chiamata; return, invece, ritorna al blocco di codice precedente: nel *main* "uccide" il programma ma, all'interno di una funzione (lo vedremo...), no.

A questo punto, se tutto è andato a buon fine, possiamo usare il canale di output; niente di strano: si usa esattamente come useresti *cout*, semplicemente scrivendo al suo posto il nome del nuovo *stream*.

Ultimo step, anch'esso trascurato dagli studenti ma importantissimo: una volta che lo *stream* non serve più va chiuso (un esempio: se non chiudi lo *stream* in scrittura su un file e, successivamente, provi ad aprire uno *stream* in lettura su quel file l'operazione fallirà sempre, ci può essere un solo *stream* per volta)! Il codice è `nomestream.close()`.

Stai attento ad una cosa: uno *stream* aperto in questo modo ha questo "difetto": se il file esiste già ed è pieno, lo sovrascrive in maniera irreversibile (meglio saperlo: sono sicuro che non vuoi aprire un *ofstream* sulla tua tesi...).

Puoi dire ad *ofstream* di aprirsi in "append mode", ovvero di scrivere alla fine del file senza cancellare ciò che esiste già; è necessario questo codice:

```
        ofstream out;
        out.open("dati.txt", std::ofstream::app);
```

Esempio 7.2

[2] Affronteremo il discorso delle funzioni approfonditamente nel capitolo 10

[3] Fino al C++98 questa funzione voleva una stringa del C, ovvero puntatore a char. Se devi pass,are con una variabile il nome del file, usa questo tipo di dato.

[4] Se il file non si trova nella cartella corrente devi inserire il percorso completo, per chiarimenti vedi A

7.2.2 Leggere da file

Poniamo di avere un file "`dati.dat`" e di sapere che vi sono 10 **float** che vogliamo caricare su un vettore. Noterai che che il codice che segue è estremamente simile all'esempio 7.2.1.

```
#include <iostream>
#include <fstream>      //ifstream
#include <cstdlib>      //exit()
using namespace std;

int main(){
        int vec[10];
        ifstream in; //dichiaro lo stream in
        in.open("dati.dat"); //apro il canale verso il file dati.dat
        if(in.fail()){ //controllo che l'apertura sia andata a buon fine
                cerr << "Apertura del canale di input fallita!" << endl;
                exit(1); //se e' fallita esco dal programma
        }

        for(int i=0; i<10; ++i)
                in>>vec[i]; //uso in come userei cin
        in.close(); //chiudo il canale
        return 0;
}
```

Esempio 7.3

Per prima cosa dichiaro uno *stream* di input *ifstream*, quindi lo apro verso il file di destinazione. A questo punto controllo che tutto sia andato bene; negli *ifstream* è ancora più importante: ti basta scrivere il file con una lettera sbagliata e l'apertura del canale fallisce (non posso aprire uno *stream* in lettura verso un file inesistente!). Senza l'if-statement potresti passare ore a cercare di capire perché il tuo programma non funziona (quando, invece, compila senza problemi!) senza venirne a capo, mentre l'errore era banalissimo...

Per capire come un file viene letto, è utile immaginarsi una sorta di vecchio giradischi, in cui vi è la testina che si muove sul file: ogni volta che leggiamo una variabile la testina avanza, e prima o poi arriva alla fine del file. Se in un vecchio giradischi, una volta arrivato alla fine, lasciavi girare il disco, non ripartiva la prima canzone, semplicemente "girava a vuoto". Con i file accade qualcosa di simile: se, una volta arrivato alla fine, continui a leggere dal file, non riprendi dalla prima variabile, bensì... che succede? Prova!

Ma come faccio a capire di essere arrivato alla fine di un file? E a "spostare la testina indietro"?

Alla fine di ogni file vi è un carattere speciale, detto "*end of file*" (abbreviato in EOF); uno *stream* di input è in grado di leggere questo carattere (che aprendo un file con un editor di testo non vediamo) e di comunicarci se è arrivato alla fine del file. Ecco il codice:

```
        ifstream in("input.txt");
        ...
        if(in.eof()){
                //do something
        }
        ...
```

Esempio 7.4

Semplicemente il metodo "`eof()`" restituisce **true** se abbiamo raggiunto la fine del file, **false** in caso contrario.

Potremmo essere interessati, dopo aver letto una prima volta il file, a tornare all'inizio per leggerlo di nuovo (nel capitolo 8 ne vedremo un importante esempio). Ecco il comando per "riportare indietro la testina".

```
ifstream in("input.txt");
...
in.clear();
in.seekg(0);
...
```

Esempio 7.5

Il metodo "`clear()`" resetta tutte le *flag* (eof, ad esempio è un *flag*, che si era settata su **true**, con "`clear()`" l'azzeriamo). Il metodo "`seekg()`" cerca la posizione tra le parentesi e vi sposta la "testina"; nel nostro caso la posizione 0, ovvero il primo bit del file.

Ti scrivo un piccolo esempio di codice in cui leggo due volte il file: la prima per trovare l'elemento minimo, la seconda per trovare l'elemento massimo (è un esempio stupido, anche perché potrebbe essere più intelligente caricare i dati in un vettore e poi lavorarci sopra, oppure cercare massimo e minimo nello stesso ciclo **while**[5]; nel capitolo 8 vedremo un utilizzo molto più importante).

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

int main(){
        ifstream in;
        int tmp=0, max=0, min=0;
        in.open("dati.dat");
        if(in.fail()){
                cerr << "Errore apertura file di input." << endl;
                return 1;
        }
        in >> min;
        while(!in.eof()){
                in >> tmp;
                if(tmp < min)
                        min=tmp;
        }
        in.clear();     //pulisco le flag
        in.seekg(0);    //porto la testina a zero per leggere dall'inizio
        in >> max;
        while(!in.eof()){
```

[5] L'unico caso che mi viene in mente in cui questo codice è sensato è quando dobbiamo elaborare un set di dati ben più grande della RAM. In quel caso, caricare i dati in un *array* è infattibile. È ancora inutile trovare massimo e minimo in due letture successive del file, ma potrebbe capitare di dover fare due elaborazioni dei dati che non possono avvenire simultaneamente.

```
                in >> tmp;
                if(tmp > max)
                        max=tmp;
        }
        cout << "Min:\t" << min << "\nMax:\t" << max << endl;
        return 0;
}
```

Esempio 7.6

7.2.3 Opzionale: una versione generalizzata

Come ti ho detto all'inizio *ofstream* e *ifstream* sono figli del tipo di dato *fstream*; possiamo evitare di usarli separatamente ma utilizzare *fstream* dicendogli di specializzarsi durante l'apertura. Può essere comodo se, in un programma, dobbiamo sia scrivere su un file che leggerlo: in questo modo dichiariamo una sola variabile e rischiamo di perderci meno cose per strada. Ecco un esempio:

```
#include <iostream>
#include <fstream>      //ofstream
#include <cstdlib>      //exit()
using namespace std;

int main(){
        fstream stream; //dichiaro lo stream senza definire se out o in (
            fstream)
        stream.open("dati.txt", std::fstream::in); //apro il canale
            specificando che e' un canale di input
        if(stream.fail()){ //controllo che l'apertura sia andata a buon fine
                cerr << "Apertura del canale di output fallita!" << endl;
                exit(1); //se e' fallita esco dal programma
        }
        //leggo da file e faccio qualcosa
        //...
        stream.close();
        stream.open("dati.txt", std::fstream::out | std::fstream::app); //
            apro uno stream in out che non sovrascrive il file dati.txt ma
            scrive in fondo, le diverse opzioni si concatenano con ``|''
        if(stream.fail()){ //controllo che l'apertura sia andata a buon fine
                cerr << "Apertura del canale di output fallita!" << endl;
                exit(1); //se e' fallita esco dal programma
        }
        //scrivo qualcosa sul file
        //...
        stream.close();
        return 0;
}
```

Esempio 7.7

L'unica cosa in più da dire è che le diverse opzioni da passare a "`open()`" si concatenano con una o più "pipe": il carattere "|".

CAPITOLO 8

Allocazione Dinamica

Ci sono situazioni in cui, a priori, non è possibile conoscere la quantità di dati che dobbiamo immagazzinare. Per esempio: il nostro programma prende in ingresso numeri da tastiera ed effettua operazioni, ma non sappiamo quanti numeri l'utente vorrà inserire; oppure abbiamo un file pieno di dati di un esperimento, conosciamo che tipo di dati sono, ma non sappiamo *quanti* sono; e così via...

La maniera naïve di affrontare il problema consiste nel dire "sicuramente avrò meno dati di...", quindi alloco un *array* (o matrice) statico molto molto grande, così da poterlo riempire quanto mi serve. Insomma: abbondo per stare sicuro.

È evidente che questo approccio ha diversi problemi. In primis il discorso dell'efficienza: è chiaro che se "per star sicuro" alloco un *array* di un milione di elementi (quando alla fine dei conti ne andrò ad usare poco più di mille) non è una grande idea (oltre al fatto che la memoria è limitata). Mi ritrovo un programma molto più "pesante" di quanto avrei bisogno che fosse. In ultima analisi, questa soluzione del problema è ben poco elegante[1]...

L'altra soluzione, molto più elegante e performante, è di creare a "*run-time*" un *array* della dimensione necessaria.

Vediamo di capire come!

8.1 Gli operatori *new* e *delete*

Quando dichiariamo "**int** a", il compilatore riserva quattro byte per la variabile `a`. Per poter trattare vettori di dimensione definita a run-time, abbiamo bisogno di rendere questo passaggio, implicitamente eseguito dal compilatore, esplicito e controllato da noi. L'operatore che

[1] A dirla tutta, ci sono situazioni in cui questa soluzione è assai funzionale. Per esempio, sappiamo che al massimo avremo 10 dati, forse 9, forse 8, forse 2, probabilmente cambieranno durante il programma, magari raggiungendo i 10 elementi, magari no. Ecco, in quel caso, con pochi dati, di quantità variabile durante l'esecuzione, può essere conveniente allocare un vettore di grandezza tot che al più verrà usata.

permette di farlo è "**new**"[2]. *New* alloca la memoria necessaria per il tipo di dato che gli segue e restituisce un puntatore che punta il primo byte di memoria allocata. Vediamo un esempio per chiarirci le idee:

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

int main(){
        int* ptr;
        ptr=new int;
        *ptr=2;
        cout << "Indirizzo dell'intero allocato dinamicamente: " << ptr <<
            endl;
        cout << "Contenuto di ptr: " << *ptr << endl;
        return 0;
}
```

Esempio 8.1

La riga "ptr=**new int**" mostra esplicitamente ciò che ho detto: **new** alloca lo spazio necessario ad un **int** e restituisce l'indirizzo di memoria di quest'ultimo.

Questo esempio è estremamente simile all'esempio 5.1, ma c'è un'enorme differenza: là usavamo il puntatore per leggere una variabile statica, qui invece usiamo il puntatore per puntare ad un'area di memoria allocata ad hoc. Stai molto attento; se perdi traccia di quest'indirizzo di memoria non hai modo di recuperarlo. Se assegni a `ptr` un nuovo indirizzo, l'area di memoria che contiene 2 è persa per sempre (confrontando con l'esempio 5.1, là la memoria era "legata" alla variabile statica, non vi era modo di perderla).

Per rendere più leggera l'esposizione, l'esempio 8.1 è scritto in maniera profondamente scorretta: cerchiamo di capire cosa c'è di sbagliato. Abbiamo detto che con **new** ci occupiamo noi di allocare la memoria al posto del compilatore. Ma chi la dealloca? Il compilatore? No! Sempre noi. Cosa vuol dire? Deallocare significa "liberare" la memoria, significa dire al sistema operativo: "Non mi serve più, è di nuovo tua". Se non viene deallocata, quella memoria rimane riservata e non può essere utilizzata da altri. Non liberare la memoria che non utilizziamo più è una pessima abitudine: riempiamo la ram di sporcizia[3]!

Per deallocare la memoria dobbiamo usare l'operatore **delete**. Delete libera la memoria puntata dal puntatore che gli segue. Riscrivo il codice dell'esempio 8.1 con quest'ultimo passaggio:

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

int main(){
        int* ptr;
        ptr=new int;
        ...
        ...
```

[2]Ho mentito: anche *new* e *delete* sono solo del C++, ma a differenza degli stream, dove nel C non c'è nulla di equivalente, il C ha due funzioni molto simili: malloc() e free().

[3]E si rischia di andare incontro ad un *memory leak*, ovvero esaurire la memoria per non averla liberata quando non ci serviva più.

```
        delete ptr; //libero la memoria puntata da ptr!
        return 0;
}
```

Esempio 8.2

Non dimenticarti mai di **delete**: oltre a farti perdere punti all'esame, dimenticarti di lui è davvero una pessima abitudine e perfino pericolosa (nella prossima sezione nomino qualche pericolo).

8.2 Intermezzo: di più su Stack e Heap

In questa sezione cercherò di accennare qualcosa di più sui due tipi di memoria esistenti, la *stack* e la *heap*. È un argomento interessante ma, diciamo, non strettamente necessario (le basi, tra l'altro, le ho già accennate nella sezione 3.2). Salta alla prossima sezione se preferisci!

Abbiamo visto che i programmi vivono nella RAM, abbiamo studiato la questione degli indirizzi di memoria, e come gestirli. Nella sezione precedente hai visto che le variabili possono essere allocate dinamicamente; nell'astrazione attuata dal sistema operativo, in un programma, la memoria statica e la memoria dinamica sono ben distinte e divise in aree diverse: la prima vive nella *stack*, la seconda nella *heap*.

Stack La *stack* è la memoria statica, è una memoria ordinata, composta un po' come una pila di fogli (ti ricordi? è una memoria "deframmentata", le variabili sono contigue...). Ogni elemento sta sopra l'altro, al suo posto, senza muoversi e senza buchi tra una variabile e l'altra. La *stack* viene organizzata a "compile-time": quando il programma inizia l'esecuzione le variabili esistono già, ogni *scope* ha il suo spazio. Essendo ordinata, e "statica" il sistema operativo sa benissimo dove si trovano tutte le variabili e le raggiunge rapidamente, quando chiediamo di scriverci o di leggerle (addirittura, se questo ti dice qualcosa, spesso viene mappata nella cache della CPU, diventando così rapidissima). Ha però due grossi difetti. primo: come ripetuto mille volte, è statica, quindi non possiamo ingrandire vettori, creare vettori non definiti precedentemente, ecc ecc... È il motivo per cui la dimensione dell'*array* statico deve essere nota a priori (con una macro, un numero o una variabile **const**): la sua dimensione deve essere già "scritta" prima dell'avvio del programma. Il secondo difetto è che la *stack* è limitata: in diverse distribuzioni di Linux la sua dimensione è intorno agli 8 mega byte[4] (puoi chiedere al sistema quanto è grande scrivendo sul terminale "`ulimit -a`"). Questo ci pone delle problematiche: non possiamo definire *array* statici troppo grandi o finiremo per esaurire la memoria[5]: se provi a definire un "**double** vector[2000000]" in un programma (circa 16mb) e ad eseguirlo, prima che succeda qualsiasi altra cosa, esso andrà in "`segmentation fault`" e verrà ucciso dal sistema operativo (in quanto sta cercando di rompere le regole dell'OS).

Heap La *heap* è la memoria dinamica. È virtualmente illimitata (in questo senso: il limite è fisico, cioè la dimensione della RAM, non è a livello di software[6]); la sua dimensione cambia a "run-time",

[4] Nei sistemi unix –Linux, MacOS, etc...– il limite della *stack* è imposto dal sistema operativo. È un limite dell'ambiente, non definito nel programma. In Windows, invece, la questione è diversa: è il compilatore che definisce il limite all'interno del programma –di default dovrebbe essere sui 2mb, ma non sono ferrato–, il sistema operativo non pone limiti.

[5] In realtà ogni *thread* ha la propria *stack* –altro suo pregio e motivo di velocità–, quindi un programma può avere più di 8 mb, ma sto decisamente divagando...

[6] Questo non è vero su Windows, dal momento che di default la RAM utilizzabile da un programma è limitata a 2GB, e tale è la *heap*.

ed è completamente gestita dal programmatore[7]. È una memoria disordinata, un po' come un mare di di sassolini: ogni elemento è in posti casuali (dove c'è spazio), è frammentata[8]; per questo motivo il sistema operativo non riesce a tenerla sotto controllo (cambia in continuazione) e mapparla sarebbe estremamente difficile. Questa caratteristica la rende più lenta della *stack*: per accedere ad ogni elemento la CPU lo deve, in qualche modo, cercare, non sa già dove si trova. Ovviamente, però, ha anche dei vantaggi: è molto più grande ed è dinamica, per cui possiamo creare *array* enormi a "run-time" (con un paio di giga byte di RAM possiamo definire un vettore dinamico di **double** di due ordini di grandezza più grande di uno della *stack*), cancellarli, crearne di nuovi ecc ecc...

Il fatto di essere gestita dal programmatore è un'arma a doppio taglio: ci dà più libertà, ma dobbiamo comportarci bene. Quando allochiamo qualcosa, poi dobbiamo deallocarlo, pulire la memoria; se non lo facciamo rimane occupata e inutilizzata. Continuare ad allocare memoria senza deallocarla può portare a *memory leak*: finiamo per esaurire tutta la memoria senza che la stiamo effettivamente usando!

Altro problema, già accennato, è che siamo noi a tenere traccia della memoria (e non la CPU), grazie ai puntatori; se però ne perdiamo le tracce, quell'elemento di memoria è perso per sempre. Stai attento!

Ora, non spaventarti, questi discorsi sono validi finché il programma "vive", ma nel momento in cui si interrompe, sia *stack* che *heap* vengono liberate, per cui la memoria che abbiamo indebitamente sporcato e occupato ritorna libera. Oltre al fatto che Linux è piuttosto "cattivo" nella gestione della memoria: se facciamo sporcizie troppo grosse, nella *heap*, ci uccide malamente (ad esempio, se proviamo a scrivere nello spazio non nostro[9]).

8.3 Array dinamici

Certo, allocare una singola variabile in modo dinamico, a questo livello, ha ben poca utilità[10]: a noi interessa molto di più la questione dei vettori. Vediamo come si alloca un *array* dinamico:

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

int main(){
        int *v;
        int n=15; //n non e' ne una macro ne una variabile const
        v=new int[n];
        for(int i=0; i<n; ++i)
                v[i]=i;
        ...
        ...
        delete[] v;
        return 0;
}
```

Esempio 8.3

[7] Questo è vero nel C/C++, in altri linguaggi, come Java, è gestita anche lei in maniera automatica.

[8] Mentre la *stack* è riservata per ogni *thread*, ogni funzione ha il suo spazio e così via, la *heap* è la stessa condivisa da tutti.

[9] Purtroppo Linux uccide il processo solo se "sgarriamo troppo", non sempre (e questo può portare ad errori gravi): analizzare esattamente quando, però, è un discorso un po' troppo tecnico

[10] Imparerai nel corso di Trattamento Numerico dei Dati Sperimentali, che allocare una singola variabile, nel mondo degli "oggetti", può essere invece estremamente utile per sfruttare una funzione del C++: il polimorfismo.

Quali sono le principali differenze rispetto all'allocazione di una singola variabile? Come puoi notare, dopo l'operatore **new**, al tipo di dato segue una doppia parentesi quadra contente la dimensione dell'*array* che vogliamo allocare. Essenzialmente, diciamo a **new** di riservare nella *heap* una striscia contigua di n interi, quindi **new** ci restituisce un puntatore al primo byte di questi interi.

Ora capisci perché nel capitolo 6 ho insistito tanto nel farti capire che un *array* è un puntatore: per costruire un *array* dinamico usiamo proprio questo tipo di dato. Una volta allocato il vettore dinamico lo usiamo esattamente come avremmo fatto con un vettore statico (accedendo agli elementi usando le parentesi quadre), e ci dimentichiamo del fatto che, in realtà, è un puntatore (in soldoni: una volta allocato, un *array* dinamico si usa esattamente come uno statico).

Quando liberiamo la memoria, però, dobbiamo ricordarci che è un *array* e non un singolo elemento: `delete[] v`, le parentesi quadre indicano all'operatore **delete** di cancellare tutta la memoria legata al puntatore `v` e non semplicemente il primo elemento (se ci dimentichiamo le parentesi, viene liberato il primo elemento e gli altri rimangono allocati).

È implicito che possiamo allocare un *array* di qualsiasi tipo, ci basta avere un puntatore a quel tipo di dato ed esplicitare a **new** che memoria vogliamo: ad esempio, per un vettore di **double** scriveremo `v=new double[n]` e così via.

8.3.1 Ingrandire e rimpicciolire vettori

Potremmo essere interessati a cambiare la dimensione del nostro vettore durante l'esecuzione del programma. Ipotizziamo di voler ingrandire l'*array*: purtroppo non esiste alcun modo per farlo. Il motivo è alquanto semplice: abbiamo visto che la *heap* è deframmentata, l'operatore **new** trova un pezzo di memoria contigua in cui allocare l'*array*, ma nessuno ci dice che dopo questa memoria ci sia ancora spazio libero; magari l'ha "incastrato" in mezzo a qualcos'altro. Se vogliamo ingrandire il nostro vettore, l'unico modo è allocarne uno nuovo della dimensione desiderata, copiare tutti gli elementi del vecchio vettore e deallocare quest'ultimo.

```
#include <iostream>
using namespace std;
int main{
        int *v=new int[10];
        //uso il vettore
        ...
        //ora lo voglio ingrandire in un vettore di 100 elementi
        int *big_v=new int[100];
        for(int i=0; i<100; ++i){
                if(i<=10)
                        big_v[i]=v[i];
                else //inizializzo a zero gli elementi successivi del
                    vettore
                        big_v[i]=0;
        }
        delete[] v; //libero la memoria di v
        //ora posso usare big_v
        ...
        ...
        delete[]big_v;
        return 0;
}
```

Esempio 8.4

Nell'esempio precedente, oltre ad aver copiato il primo vettore nel secondo, ho inizializzato a zero gli elementi che seguono. Anche i vettori, se non li riempi subito, è buona norma inizializzarli!

Il C++ non prevede uno strumento per rimpicciolire un *array* dinamico. Anche per rimpicciolire un *array* devi crearne uno nuovo più piccolo e copiarci gli elementi del vecchio, ma a volte è un po' una fatica sprecata: se hai un *array* di 100 elementi e, ad un certo punto, ne usi solo 70, niente di grave, semplicemente ignori gli ultimi 30. Allocarne uno nuovo, probabilmente, ti costa molto di più. Se invece hai un vettore da diversi megabyte e, ad un certo punto, hai bisogno di una decina di elementi, il discorso è diverso.

8.3.2 Alcuni errori

- La prima cosa da evitare è riallocare il puntatore:

```
#include <iostream>
using namespace std;
int main(){
        int *p=new int[10];
        for(int i=0; i<10;++i)
                p[i]=i;
        p=new int[5]; //alloco nuovamente p
        for(int i=0; i<5; ++i)
                cout << p[i] << endl;
        delete[]p;
        return 0;
}
```

Esempio 8.5

Cosa viene stampato? I valori inizialmente assegnati al vettore di dieci elementi? No, "sporcizia": viene stampato il vettore di cinque elementi non inizializzato. Ricorda: **new** alloca la memoria richiesta alla sua destra e restituisce l'indirizzo del primo byte; in poche parole, `p` punta alla nuova memoria e il primo vettore viene perso per sempre, rimanendo irraggiungibile nel limbo della *heap*.

- Un altro errore è cercare di liberare della memoria già liberata:

```
#include <iostream>
using namespace std;
int main(){
        int* v=new int[10];
        int* p;
        p=v; //ora p punta alla memoria di v, posso usare p come vettore
        //ad esempio riempio il vettore di dati
        for(int i=0; i<10; ++i)
                p[i]=i;
        ...
```

```
        ...
        delete[] p; //rimuovo la memoria puntata da p
        delete[] v; //rimuovo la memoria puntata da v, ma attento: e' la
            stessa puntata da p, che succede?
        return 0;
}
```

Esempio 8.6

È un errore comune che ho visto commettere più volte: essenzialmente si alloca un vettore dinamico, quindi si fa puntare un secondo puntatore alla sua area di memoria (potrebbe essere utile dentro una funzione, lo vedremo più avanti). Arrivato il momento di utilizzare **delete**, si libera la memoria del secondo puntatore, infine, dimenticandosi che il primo punta alla stessa memoria, si libera anche questo.

La conseguenza è nefasta: *core dumped*, il sistema operativo ci uccide malamente. Il messaggio di errore che riceverai è simile al seguente (non leggerlo tutto, te lo riporto semplicemente perché, se ti capiterà, avrai un'idea di cosa è successo):

```
*** Error in `./mem': double free or corruption (fasttop): 0x0000000001f71060 ***
======= Backtrace: =========
/usr/lib/libc.so.6(+0x70c4b)[0x7f0a4acabc4b]
/usr/lib/libc.so.6(+0x76fe6)[0x7f0a4acb1fe6]
/usr/lib/libc.so.6(+0x777de)[0x7f0a4acb27de]
./mem[0x400b75]
/usr/lib/libc.so.6(__libc_start_main+0xf1)[0x7f0a4ac5b291]
./mem[0x4009da]
======= Memory map: ========
00400000-00401000 r-xp 00000000 08:02 9175556                            /home/lorenzo/esempi/mem
00601000-00602000 r--p 00001000 08:02 9175556                            /home/lorenzo/esempi/mem
00602000-00603000 rw-p 00002000 08:02 9175556                            /home/lorenzo/esempi/mem
01f5f000-01f91000 rw-p 00000000 00:00 0                                  [heap]
7f0a44000000-7f0a44021000 rw-p 00000000 00:00 0
7f0a44021000-7f0a48000000 ---p 00000000 00:00 0
7f0a4ac3b000-7f0a4add0000 r-xp 00000000 08:02 2234468                    /usr/lib/libc-2.24.so
7f0a4add0000-7f0a4afcf000 ---p 00195000 08:02 2234468                    /usr/lib/libc-2.24.so
7f0a4afcf000-7f0a4afd3000 r--p 00194000 08:02 2234468                    /usr/lib/libc-2.24.so
7f0a4afd3000-7f0a4afd5000 rw-p 00198000 08:02 2234468                    /usr/lib/libc-2.24.so
7f0a4afd5000-7f0a4afd9000 rw-p 00000000 00:00 0
7f0a4afd9000-7f0a4afef000 r-xp 00000000 08:02 2236025                    /usr/lib/libgcc_s.so.1
7f0a4afef000-7f0a4b1ee000 ---p 00016000 08:02 2236025                    /usr/lib/libgcc_s.so.1
7f0a4b1ee000-7f0a4b1ef000 r--p 00015000 08:02 2236025                    /usr/lib/libgcc_s.so.1
7f0a4b1ef000-7f0a4b1f0000 rw-p 00016000 08:02 2236025                    /usr/lib/libgcc_s.so.1
7f0a4b1f0000-7f0a4b2f3000 r-xp 00000000 08:02 2234551                    /usr/lib/libm-2.24.so
7f0a4b2f3000-7f0a4b4f2000 ---p 00103000 08:02 2234551                    /usr/lib/libm-2.24.so
7f0a4b4f2000-7f0a4b4f3000 r--p 00102000 08:02 2234551                    /usr/lib/libm-2.24.so
7f0a4b4f3000-7f0a4b4f4000 rw-p 00103000 08:02 2234551                    /usr/lib/libm-2.24.so
7f0a4b4f4000-7f0a4b66c000 r-xp 00000000 08:02 2230235                    /usr/lib/libstdc++.so.6.0.22
7f0a4b66c000-7f0a4b86c000 ---p 00178000 08:02 2230235                    /usr/lib/libstdc++.so.6.0.22
7f0a4b86c000-7f0a4b876000 r--p 00178000 08:02 2230235                    /usr/lib/libstdc++.so.6.0.22
7f0a4b876000-7f0a4b878000 rw-p 00182000 08:02 2230235                    /usr/lib/libstdc++.so.6.0.22
7f0a4b878000-7f0a4b87c000 rw-p 00000000 00:00 0
7f0a4b87c000-7f0a4b89f000 r-xp 00000000 08:02 2234467                    /usr/lib/ld-2.24.so
7f0a4ba55000-7f0a4ba5b000 rw-p 00000000 00:00 0
7f0a4ba9d000-7f0a4ba9e000 rw-p 00000000 00:00 0
7f0a4ba9e000-7f0a4ba9f000 r--p 00022000 08:02 2234467                    /usr/lib/ld-2.24.so
7f0a4ba9f000-7f0a4baa0000 rw-p 00023000 08:02 2234467                    /usr/lib/ld-2.24.so
7f0a4baa0000-7f0a4baa1000 rw-p 00000000 00:00 0
7ffd58191000-7ffd581b2000 rw-p 00000000 00:00 0                          [stack]
7ffd581ee000-7ffd581f0000 r--p 00000000 00:00 0                          [vvar]
7ffd581f0000-7ffd581f2000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
Aborted (core dumped)
```

La prima riga riporta un "double free or corruption", è un piccolo *hint* di Linux: probabilmente hai liberato la memoria due volte. L'ultima riga è un *core dumped*, la memoria è stata completamente liberata e il processo terminato. Come vedi sono molte righe di

errore, che riportano gli indirizzi di memoria in cui qualcosa è andato storto. Insomma, non andare nel panico: il problema potrebbe essere semplice!

- L'ultimo errore che può capitare è il *bad alloc*. A priori non è un errore di programmazione, e la causa è semplice. Ad esempio:

```
#include <iostream>
using namespace std;
int main(){
        double *v=new double[10000000000000000]; //memoria sproposita ta,
            quanti giga sono?
        delete[]v;
        return 0;
}
```

Esempio 8.7

Eseguendo il programma sul mio pc ricevo il seguente messaggio di errore:

```
terminate called after throwing an instance of 'std::bad_alloc'
what():  std::bad_alloc
Aborted (core dumped)
```

Stiamo richiedendo più memoria di quella disponibile in RAM (oppure non c'è abbastanza memoria contigua: ricordati che la *heap* è deframmentata). Viene quindi sollevata un'eccezione[11]e il programma viene terminato. Ora, nell'esempio precedente ho richiesto una quantità di memoria impensabile, ma potrebbe capitarti di dover allocare un paio di giga di memoria (niente di assurdo!) e non averla a disposizione: in quel caso non hai scritto nulla di male, semplicemente... cambia pc!

8.4 Matrici dinamiche

Per allocare dinamicamente una matrice possiamo fare due cose. La prima è linearizzare la matrice: ridurla ad un vettore di dimensione n per m e rinunciare alla comodità del doppio indice. La seconda è utilizzare un artificio per "costruire" in memoria una matrice e poter utilizzare il doppio indice. Quando nel corso di Informatica 1 si parla di matrici dinamiche, ci si riferisce alla seconda situazione.

Se ti ricordi, nel capitolo sugli *array*, ho accennato al fatto che una matrice può essere vista come un vettore di vettori. Ecco! È esattamente questa la strada da seguire: dobbiamo costruire un *array* di *array*.

[11]Le eccezioni sono uno degli strumenti più belli e potenti del C++ che lo distinguono dal C. Permettono di gestire situazioni non note a priori, errori che non dipendono dal programmatore ma, ad esempio, dall'ambiente ed imprevedibili (come l'insufficienza di memoria). La gestione delle eccezioni permette di evitare che i programmi "muoiano" quando qualcosa va storto; essa è fondamentale, ad esempio, in ambito di sistemi embedded. Un'eccezione non gestita, invece, porta alla fine del programma (come nel nostro caso). Purtroppo le eccezioni esulano dagli scopi del corso di Informatica 1...

Usando un puntatore a puntatore, allochiamo un *array* di puntatori (per esempio di dimensione n) Fatto ciò, per ogni puntatore, allochiamo un vettore di dati (ad esempio di dimensione m). Abbiamo costruito, così, un *array* di n *array* grandi m: una matrice n per m!

```
#include <iostream>
using namespace std;
int main(){
        int n, m;
        n=2;
        m=3;
        int **matr;
        matr=new int*[n]; //alloco il vettore di puntatori
        for(int i=0; i<n; ++i)
                matr[i]=new int[m]; //per ogni puntatore del vettore alloco
                    un vettore di interi
        //ora riempio la matrice
        for(int i=0; i<n;++i)
                for(int j=0; j<m; ++j)
                        matr[i][j]=i*m+j;
        //stampo
        for(int i=0; i<n;++i){
                for(int j=0; j<m; ++j)
                        cout << matr[i][j]  << "\t";
                cout << endl;
        }

        //dealloco, attenzione a come!
        for(int i=0; i<n; ++i)
                delete[] matr[i]; //dealloco ogni vettore di dati
        delete[] matr; //dealloco il vettore di puntatori
        return 0;
}
```

Esempio 8.8

Analizziamo il codice: tutto parte da **int** **matr, un puntatore a puntatore. La riga più strana è, probabilmente matr= **new int***[n]: stiamo allocando un vettore di puntatori ad interi; tale *array* può essere puntato solo da un puntatore a puntatore. Ti è chiaro il perché, vero? A **new** segue il tipo di dato da allocare, quindi **int***.

Nel ciclo **for** che segue, allochiamo, per ogni puntatore dell'*array*, un vettore di interi di dimensione m. Successivamente riempiamo e leggiamo la matrice: come vedi si utilizza esattamente come una matrice statica, nulla di nuovo.

Infine, dobbiamo liberare la memoria. Il processo si esegue al contrario: prima deallochiamo i vettori di interi e, per ultima cosa, il vettore di puntatori.

Per quanto riguarda l'ingrandire e il rimpicciolire matrici, valgono gli stessi discorsi fatti in 8.3.1; per esercizio, però, prova a scrivere tu il codice per ingrandire (e rimpicciolire) una matrice!

CAPITOLO 9

Struct

Dobbiamo rappresentare un punto materiale: cosa ci serve? Sicuramente almeno le sue coordinate cartesiane e la sua massa (se è un punto statico). Tradotto in C++, avremo bisogno di quattro **float** o **double**, a seconda della precisione che desideriamo. Potremmo, quindi, dichiarare quattro variabili **float** o quattro *array*, se abbiamo più punti materiali. Utilizzando questa via, però, perdiamo l'unitarietà del concetto "punto materiale". Un caso analogo potrebbe essere, ad esempio, quello delle particelle dove, oltre alle caratteristiche del punto materiale, vi è la carica, lo spin, il nome, ecc... È facile trovare molte circostanze simili.

In poche parole, stiamo considerando situazioni in cui vi è un "oggetto" descritto da più variabili. Nulla ci vieta di dichiarare le diverse variabili e di essere noi programmatori a ricordarci che compongono un oggetto più astratto, oppure, possiamo sfruttare uno strumento del C: la **struct**.

9.1 Definire un nuovo tipo di dato

Vediamo le **struct** all'opera:

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

//Definisco la struttura punto materiale
struct puntoMateriale{
        float x;
        float y;
        float z;
        float m;
};

int main(){
        puntoMateriale punto1; //dichiaro una variabile di tipo
            puntoMateriale

        //accedo agli elementi della struttura
```

```
        punto1.x=2.1;
        punto1.y=-7.8;
        punto1.z=0.1;
        punto1.m=1E12;

        ...
        ...

        return 0;
}
```

Esempio 9.1

Quando definiamo una struttura, stiamo in tutti i sensi creando un nuovo tipo di dato. La scrittura **struct** name{...} definisce il tipo di dato `name` (il nome che segue alla parola chiave **struct**). A questo punto, `name` esisterà nel nostro codice e potrà essere usato esattamente come siamo soliti fare con i tipi di dato built-in (es. **float, int**, ecc...).

Come vedi, a seguire il nome della struttura, vi è una coppia di parentesi graffe: al loro interno mettiamo tutte le variabili che vogliamo che la nostra struttura contenga (nel nostro caso, quattro **float** con i loro rispettivi nomi). Una volta chiuse le graffe, ricordati di concludere con il punto e virgola (è uno dei pochissimi casi in cui, dopo la graffa, ci vuole).

Ora, il tipo di dato definito esiste e possiamo utilizzarlo nel *main*, ma come? Niente di nuovo: `nome nomevariabile` (nel nostro caso `puntoMateriale punto1`).

9.1.1 L'operatore punto (".")

La nostra variabile `punto1`, però, è un po' strana: contiene quattro **float**, e noi vogliamo poterci accedere. Nell'esempio 9.1 ho scritto, ad esempio, "`punto1.x`": l'operatore "punto" serve ad accedere agli elementi della struttura che, nel nostro caso, sono x, y, z e m.

Che sia ben chiaro: `punto1` è una variabile di tipo `puntoMateriale` (definita dal programmatore all'inizio del codice); `punto1.x`, invece, è a tutti gli effetti un **float** (e come tale va usato).

Ricapitolando, una struttura è una sorta di "collezione" di elementi racchiusi dentro un nuovo tipo di dato definito dal programmatore. Questi elementi non devono essere dello stesso tipo. Inoltre, ognuno di essi ha il proprio nome; ad esempio, possiamo avere:

```
struct example{
        char* pointer1;
        long num;
        double* pointer2;
};
```

Esempio 9.2

Se dichiariamo nel *main* "example foo", il dato `foo` è una variabile di tipo `example`. Per assegnare all'elemento `pointer1` un indirizzo di un **char**, scriveremo "foo.pointer1=&address". Per accedere alla variabile puntata, il codice sarà: "*(foo.pointer1)='a'" (nota le parentesi tonde: sono utili per ricordarsi che `foo.pointer1` è un puntatore a **char** e che è lui a dover essere

dereferenziato, non semplicemente `pointer1`, che di per sé non è niente).

A volte, potrebbe capitarti di vedere questa scrittura (fuori dal *main*):

```
struct example{
        char* pointer1;
        long num;
        double* pointer2;
} foo;
```

Esempio 9.3

Stiamo contemporaneamente definendo il tipo di dato `example` e dichiarando la variabile `foo` che, essendo fuori dal *main*, sarà una variabile globale.

Attento ai nomi! Una variabile non può avere lo stesso nome del suo tipo di dato e, così come **float float** non ha senso, anche questo è illegale:

```
struct dato{
        int intero;
};

int main(){
        dato dato; //illegale!
        return 0;
}
```

Esempio 9.4

Al contrario, per quanto possa confondere, una variabile può avere lo stesso nome di un elemento della struttura della quale è un dato:

```
struct dato{
        int intero;
};

int main(){
        dato intero;
        intero.intero=3; //brutto, ma lecito
}
```

Esempio 9.5

Sempre molto ingannevole, ma lecito, è quanto segue:

```
struct dato{
        char dato; //molto brutto, ma lecito
};
```

Esempio 9.6

Ma, se puoi (e puoi), evita cose di questo tipo.

9.2 Struct e array

Possiamo sia definire *array* di **struct** che **struct** contenenti *array*. Di per sé, nessuna delle due cose è complicata, ma bisogna stare un po' attenti all'uso delle parentesi quadre nei due casi. Vediamo un esempio:

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

struct vettore{
        double vec[100];
        unsigned short utilizzati; //quanti posti del vettore abbiamo usato
};

int main(){
        vettore v1;
        vettore v[10];
        v1.vec[5]=12.2; //assegnazione al sesto elemento del array di double
             contenuto in v1
        v[3].utilizzati=0; //assegnazione al membro unsigned short della
            quarta struct del array di struct
        v[1].vec[12]=1.1; //assegnazione al tredicesimo elemento del array
            di double contenuto nel secondo elemento del array di struct
        return 0;
}
```

Esempio 9.7

La **struct** `vettore` contiene un *array* di **double** di 100 elementi. La variabile `v1` è un singolo dato di tipo `vettore`: per accedere al membro `vec` usiamo l'operatore *punto*. Dovrebbe esserti chiaro che `v1.vec` è un puntatore a **double**: se vogliamo accedere agli elementi puntati, dobbiamo usare le parentesi quadre in fondo (quindi `v1.vec[i]`). Un altro modo di vedere la questione è che stiamo usando l'operatore *punto* per accedere al membro *array* di cui dobbiamo specificare l'elemento che vogliamo.

Passiamo a considerare "`v[3]. utilizzati`": `v` è un *array* di strutture; siccome vogliamo accedere ad un suo elemento le parentesi quadre vanno dopo il nome dell'*array*. A questo punto, `v[3]` è una singola struttura (la quarta dell'*array*), per cui utilizziamo il punto per accedere all'elemento `utilizzati`.

L'ultimo caso è "`v[1].vec[12]`": come prima, `v` è un *array* di strutture, per cui per accedere ad un suo elemento dobbiamo usare le parentesi quadre subito dopo. Quindi, vogliamo accedere ad un elemento di "`vec`" il quale, a sua volta, è un *array*: dobbiamo usare le parentesi quadre anche dopo di esso.

9.3 Struct, puntatori, allocazione dinamica e memoria

Una volta definita una **struct**, esiste anche il tipo di dato puntatore ad essa (d'altro canto se esiste l'*array*...).

Non c'è nulla di nuovo, in realtà: solo una sintassi un po' particolare.

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

```
struct example{
        int n;
        float* x;

};

int main(){
        example* p=new example; //alloco una variabile di tipo example
        float y=1.1;
        //assegno ai membri di *p i valori
        (*p).n=12;
        (*p).x=&y;
        //stampo i valori
        cout << (*p).n << endl;
        cout << *((*p).x) << endl;

        delete p;
        return 0;
}
```

Esempio 9.8

Come detto, niente di nuovo, solo una sintassi un bel po' "elaborata". In particolare, vorrei richiamare la tua attenzione sull'ultima riga: voglio stampare il valore del **float** puntato dal membro della nostra struttura. Per prima cosa, essendo una variabile dinamica, dereferenzio il puntatore `p` (e quindi scrivo `(*p)`): a questo punto accedo al membro `x`. Ora, il mio `(*p).x` è, a sua volta, un puntatore da dereferenziare, quindi serve un ulteriore asterisco, da cui la scrittura `*((*p).x)`.

9.3.1 L'operatore "->"

Inutile dire che la scomodità di questa sintassi è notevole. Per rendere più semplici ed esplicite le cose, è stato introdotto l'operatore "`->`" (una sorta di freccia). Il codice dell'esempio 9.3 diviene:

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

struct example{
        int n;
        float* x;
};

int main(){
        example* p=new example; //alloco una variabile di tipo example
        float y=1.1;
        //assegno ai membri di *p i valori
        p->n=12;
        p->x=&y;
        //stampo i valori
        cout << p->n << endl;
        cout << *(p->x) << endl;
```

```
        delete p;
        return 0;
}
```

Esempio 9.9

Quando abbiamo un puntatore ad una **struct** (o, quando le imparerai, una classe), possiamo accedere ai membri della struttura puntata utilizzando `puntatore->membro` al posto del più artificioso `(*puntatore).membro`.

Array dinamico di struct Se, invece, abbiamo un *array* dinamico di **struct**, non dobbiamo usare nessuna freccia, ma solo le semplici quadre (che, come sai benissimo, fungono già loro da operatore di dereferenziazione). Il seguente codice dovrebbe essere ovvio:

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

struct example{
        int n;
        float x;
};

int main(){
        unsigned n=10;
        example *p=new example[n];
        for(int i=0; i<n; ++i){
                p[i].n=i;
                p[i].x=0.1*i;
        }

        delete[] p;
        return 0;
}
```

Esempio 9.10

9.3.2 L'operatore *sizeof*

Abbiamo la seguente struttura:

```
struct example{
        int n;
        double x;
        float y;
};
```

Esempio 9.11

E ci chiediamo: ma quanto spazio occupa in memoria?

In realtà, la risposta è semplice: la somma dello spazio occupato dai suoi membri[1] (quindi 4+8+4 byte)[2].

Ci sono situazioni in cui il calcolo della dimensione di una struttura potrebbe diventare un po' complicato: ad esempio, strutture dentro strutture di cui, magari, neanche conosciamo i membri. Per ovviare a queste problematiche, esiste un comodo operatore: **sizeof**[3].

Sizeof vuole alla sua destra una variabile, oppure, un tipo di dato; in questo secondo caso, dobbiamo racchiudere il tipo di dato tra parentesi tonde. L'operatore, quindi, restituisce la dimensione in byte di ciò che segue. Vediamo **sizeof** all'opera:

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

struct example1{
        float x;
        int n;
        double y;
};

struct example2{
        example1 s1, s2;
        int a,b,c;
        char l,m;
        short n;
};


int main(){
        example2 ex;
        example1* p=new example1;
```

[1]Questo è corretto in linea teorica. Nella pratica, in realtà, *non è vero*. Quella che sto per fare è una nota molto tecnica: se non ti interessa, non leggerla e dimentica quello che hai appena letto prendendo per corretto che la dimensione di una *struct* sia la somma dei suoi elementi.

Per motivi di efficienza della CPU, gli elementi nella memoria vanno allineati a particolari byte e dimensioni. Ad esempio, abbiamo visto che su una CPU da 64 bit, il bus di dati è da 8 byte: leggendo 8 byte per volta, è conveniente che una *struct* occupi multiplo di 8 byte o un sottomultiplo (2, 4, 8, 16, ecc...). Allo stesso modo, per alcuni tipi di dato, è conveniente iniziare a particolari byte. Se ipotizziamo di contare dal byte numero "zero", le variabili vengono allineate in multipli di: 1 byte per i *char*, 2 byte per gli *short*, 4 byte per gli *int* e *float* e 8 per i *double* e *long* (ad esempio, uno *short* lo possiamo trovare al byte zero, al byte due, ecc; ma non al byte uno, tre, ecc). Per chiarire, una struttura contenente un *int* e un *char*, su un'architettura intel a 64 bit, occupa molto probabilmente otto byte e non cinque (per allinearsi al bus di 8 byte); oppure, in una *struct* contenente un *char* e uno *short* viene inserito un byte "fasullo" tra i due dati per far iniziare lo *short* al terzo byte (occupando, alla fine, 4 byte e non 3). Questa problematica viene detta "data alingment", e l'utilizzo di byte non significativi per allineare i dati è chiamato "byte padding". Non è facile, a priori, sapere la dimensione della *struct* perché non sempre il compilatore decide di usare l'intero *bus* di dati (a volte, per *struct* piccole usa un sottomultiplo); al contrario, l'allineamento dei dati standard ai byte richiesti avviene sempre.

Tutto questo discorso, per l'esame, dimenticatelo! Ogni volta che si chiedono dimensioni e indirizzi di *struct* ci si riferisce alla regola teorica che la sua dimensione è la somma dei suoi elementi.

[2]E una *struct* vuota? *struct example{}* quanto spazio occupa? La risposta è un byte. Secondo te perché? Potrebbe occuparne zero? E meno di un byte? Questa è, ovviamente, una particolare eccezione: la norma è che una *struct* occupa lo spazio occupato dai suoi membri.

[3]Con l'operatore *sizeof*, puoi verificare il discorso della nota precedente riguardo al fatto che, a volte, la dimensione delle *struct* non è equivalente alla somma dei suoi dati. Negli esempi di questo capitolo, invece, ho scelto situazioni particolari in cui la dimensione della *struct* è esattamente la somma dei suoi elementi.

```
        cout << "Dimensione del tipo di dato ''int'': " << sizeof (int) <<
            endl;
        cout << "Dimensione della variabile ex: " << sizeof ex << endl;
        cout << "Dimensione della variabile puntata da p: "<< sizeof *p <<
            endl;
        cout << "Dimensione del tipo di dato ''example1'': " << sizeof (
            example1) << endl;
        cout << "Dimensione di un puntatore a ''example1'' " << sizeof p<<
            endl;

        delete p;
        return 0;
}
```

Esempio 9.12

Sul mio pc l'output è il seguente:

```
Dimensione del tipo di dato ''int'':        4
Dimensione della variabile ex:              48
Dimensione della variabile puntata da p:    16
Dimensione del tipo di dato ''example1'':   16
Dimensione di un puntatore a ''example1'':  8
```

Dove le dimensioni sono, come detto, in byte. L'unico commento che ci tengo a fare riguarda l'ultimo valore: su una macchina a 64 bit (come è il mio computer), un puntatore a qualsiasi tipo di dato avrà sempre la stessa dimensione: 8 byte (che sono, appunto, 64 bit).

Una parte di un esercizio di un tema d'esame (25 febbraio del 2015) era:

```
Dato questo codice:

struct strumento{
    char nome[5];
    int tipo;
    double fattore[2];
};

Qual è la lunghezza complessiva in byte di un elemento di tipo strumento?
```

Ovviamente, essendo uno scritto, non puoi usare **sizeof**. Prova a rispondere!

9.3.3 Struct e indirizzi di memoria

Se hai capito la sottosezione precedente, questa non dovrebbe aggiungere praticamente nulla di nuovo. Nel capitolo sulla rappresentazione in memoria, abbiamo imparato come si comportano gli indirizzi per i tipi di dato standard. E per le **struct** cosa succede? Abbiamo visto che la dimensione di una **struct** è la somma dei suoi elementi: gli indirizzi seguono in maniera naturale. Ad esempio, abbiamo un array di **struct**, ognuna delle quali occupa 16 byte. Se il primo elemento ha come indirizzo 0x7ffe31f60ad0 per il secondo ci spostiamo di 16 byte, quindi 0x7ffe31f60ae0, per il terzo di altri 16 byte, quindi 0x7ffe31f60af0, e così via.

Se poi siamo interessati agli indirizzi degli elementi della **struct**, partendo dal suo indirizzo, ci basta spostarci degli n byte di ogni elemento. Ad esempio, sia:

```
struct foo{
        char a,b;
        int c;
};

int main(){
        foo dato;
        return 0;
}
```

Esempio 9.13

Se l'indirizzo di `dato` è 0x7ffe14f62b10, per avere `dato.a` ci spostiamo di 2 byte, quindi 0x7ffe14f62b12, lo stesso per `data.b`, quindi 0x7ffe14f62b14, e di altri 4 byte per `data.c`, quindi 0x7ffe14f62b18.

Per esercitarti, prova a risolvere il seguente esercizio di un tema d'esame (Gennaio 2017):

Data la struttura:

```
struct punto{
    double x;
    double y;
};
```

dire che cosa significa la dichiarazione
punto v[10];

Se l'indirizzo di v[0] è un valore esadecimale che termina con a3cd, scrivere le parti finali degli indirizzi di: v[2] e v[4]

9.4 Il C++ e gli oggetti

All'inizio di questo capitolo, se ci hai fatto caso, ho scritto che le **struct** sono una caratteristica del `C`. Un vero "oggetto", oltre ad essere caratterizzato da delle variabili, è caratterizzato da delle *proprietà*

che nella programmazione si sintetizzano in "funzioni". Le particelle, oltre che avere una posizione, una massa, una carica, una velocità, un nome, possono interagire tra di loro, influenzarsi, annichilarsi, ecc... Nel `C` tutte questa caratteristiche andrebbero implementate con codice esterno alla struttura, il `C++`, invece, ha introdotto il concetto di "oggetto" dotato di proprietà interne e non esterne: è nata la *classe* (il codice che implementa le proprietà è dentro la classe e non all'esterno, la classe diviene un oggetto nella sua interezza). Le classi rendono il `C++` un linguaggio orientato agli oggetti e non, come invece è il `C`, un linguaggio solamente procedurale e imperativo. L'essere orientato agli oggetti è la più grande differenza del `C++`, che lo rende un linguaggio di livello più alto rispetto al `C`. La possibilità di scrivere sia ad alto livello che a basso livello rende il `C++` un linguaggio estremamente potente e versatile. Purtroppo, gli oggetti, la vera caratteristica del `C++`, non sono un obiettivo di questo corso e, in quanto tali, non li affronteremo.

Abbiamo già usato qualche oggetto: gli *stream*. Se ricordi, abbiamo sfruttato alcune funzioni come "open()" utilizzando l'operatore di accesso "punto": `stream.open("nomefile")`. Nel `C`, se lo *stream* fosse stato una struttura, avremmo, probabilmente, utilizzato una funzione esterna e lo *stream* non l'avrebbe contenuta esso stesso (sarebbe stato, semplicemente, un'accozzaglia di dati racchiusi in una struttura). Può sembrare una sottigliezza, ma gli "oggetti" hanno rivoluzionato la programmazione rendendola molto più sintetica, meno dispersiva, con una faccia diversa e più organizzata: la maggior parte dei linguaggi più moderni sono orientati agli oggetti.

CAPITOLO 10

Funzioni

Nel capitolo sugli *array*, abbiamo imparato ad ordinarli, trovare l'elemento massimo, calcolarne media, varianza, ecc...

Ipotizziamo di scrivere un codice che riempie un vettore di dati, lo rielabora, trova l'elemento massimo, lo rielabora ancora, trova di nuovo l'elemento massimo e così via per diverse ripetizioni. Avremmo bisogno di un codice (abbozzato) di questo tipo:

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

int main(){
        double* array;
        usigned int dim=0;
        double max;

        //1) Carico da file
        ...

        //2) Eseguo dei conti
        ...

        //3) Calcolo il massimo
        for(int i=1; i<dim; ++i){
                if(array[i]>max)
                        tmp=array[i];
        }
        ...

        //4) Eseguo altri conti
        ...

        //5) Calcolo nuovamente il massimo
        for(int i=1; i<dim; ++i){
                if(array[i]>max)
```

```
                tmp=array[i];
        }
        ...
        //e cosi' via...
        ...

        return 0;
}
```

Esempio 10.1

Come puoi notare, il codice per calcolare il massimo è sempre lo stesso. Semplicemente lo ripetiamo più volte ove serve. L'algoritmo per il massimo ha un codice breve, ma potremmo averne di molto più lunghi e complessi, la cui ripetizione confonderebbe inutilmente (oltre ad essere piuttosto scomoda da riscrivere n volte).

Alla fine dei conti, l'algoritmo "massimo" vuole in ingresso un vettore e la sua dimensione e restituisce un elemento dell'*array*. È una sorta di "$massimo = f(vettore, dimensione)$". Possiamo "rubare" alla matematica il concetto di funzione: un oggetto che, preso in argomento delle variabili, esegue qualcosa e, se necessario, restituisce dei valori.

Una funzione è proprio questo: una porzione di codice isolata in una "scatola", dotata del proprio spazio dei nomi. La funzione può essere chiamata ovunque da "terzi", eseguendo il suo compito.

Quando chiamiamo una funzione, è come se il nostro codice principale si interrompesse momentaneamente per entrare in un nuovo ambiente: la funzione. Questa viene caricata in RAM, con un proprio spazio dei nomi: viene eseguito il codice della funzione, e quindi, una volta concluso, ritorna qualcosa al chiamante e il codice principale continua.

Sin dal tuo primo "Hello World" hai usato una funzione: `int main()`. *Main*, definita così, è una funzione che in argomento non prende niente e restituisce al chiamante un intero. Ma chi è quest'ultimo? È il sistema operativo (ricordi il "`return 0`"?). *Main* è la funzione principale, colei che può chiamare tutte le altre, il punto di partenza e di fine (quindi che non può mancare) e sopra a lei c'è solo il sistema operativo.

Ma vediamo come definire nuove funzioni e, quindi, come utilizzarle.

10.1 Dichiarare, definire e usare una funzione

La prima cosa da definire è un'"interfaccia": bisogna dire al compilatore che forma ha la nostra funzione.

L'interfaccia della funzione è chiamata *prototipo*. Il prototipo della funzione ricerca del massimo potrebbe essere il seguente:

```
double Max(double [], unsigned int);
```

Esempio 10.2

o anche:

```
double Max(double array[], unsigned int dim);
```

Esempio 10.3

Il prototipo di una funzione ha lo scopo di rendere chiaro come è fatta e come si utilizza: nel nostro caso, vuole per argomento (ciò che è tra le parentesi) un *array* di **double** e un **unsigned int** e restituisce un **double**.

Il prototipo è una sorta di facciata, in cui vediamo che argomenti vuole, il suo nome e cosa restituisce (tra le parentesi basta il tipo di dato: il nome della variabile, nel prototipo, è facoltativa).

Il prototipo deve essere scritto prima del *main*: quando andiamo a chiamare una funzione, il compilatore deve conoscere già il prototipo (gli basta questo, non è necessario che sappia come è implementata).

Come si utilizza la funzione di cui abbiamo visto il prototipo (esempio 10.3)? Una funzione va "chiamata", farlo è semplicissimo:

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

//Definisco il prototipo, una buona cosa e' metterci prima un commento che
    spiega la funzione:
//Find maximum element of array which has dimension dim
double Max(double array[], unsigned int dim);

int main(){
        double* array;
        unsigned int n=0;
        double max=0;

        //carico in qualche modo gli elementi in array (che diviene un
            vettore dinamico) e n diventa diverso da zero
        ...
        ...
        max=Max(array, n); //gli array non si passano usando le parentesi
            quadre, ma solo con il loro nome: passiamo in realta' il
            puntatore

        //posso anche evitare di usare una variabile, le funzioni possono
            apparire dentro i ''cout'':
        cout << "Valore massimo: " << Max(array,n) << endl;

}
```

Esempio 10.4

Ma *Max* come è fatta? In tutto questo discorso, in effetti, manca il *corpo* della funzione. Il corpo è la vera e propria definizione: un blocco di codice dove specifichiamo come agisce la funzione. Ad esempio:

```
#include <iostream>

//Prototipo
```

```
double Max(double array[], unsigned int dim);
//main
int main(){
        ...
}

//corpo della funzione
double Max(double array[], unsigned int dim){ //variabili della funzione
        double tmp=0; //variabili della funzione
        for(int i=1; i<dim; ++i){
                if(array[i]>max)
                        tmp=array[i];
        }
        return tmp;
}
```

Esempio 10.5

Prima cosa da sottolineare: nel corpo della funzione il nome delle variabili del suo argomento (ciò che è tra le tonde) non è più facoltativo, lo devi mettere! Perché? Come ho già accennato, una funzione ha il suo spazio dei nomi: vede solo le variabili locali (e globali), ma non quelle delle altre funzioni (ad esempio non vede quelle del *main*, che è un'altra funzione). Le sue variabili sono definite o nell'argomento o dentro il suo codice. Per farla breve, "**unsigned int** dim" è una variabile locale visibile all'interno del corpo della funzione, che nasce quando questa viene chiamata e muore quando finisce.

Per chiarire il concetto:

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

int glo=4; //variabile globale

int foo(int);

int main(){
        int a=1, b=0;
        b=foo(a);
        cout << d << endl; //provo a stampare una variabile di foo, ERRORE!
            non viene vista
}

int foo(in c){
        int d=3;
        cout << glo << endl; //ok, variabile globale, viene vista
        cout << c << d <<endl;//ok, variabili locali
        cout << a << endl; //NON compila! E' una variabile di un altra
            funzione, non viene vista!
        return d;
}
```

Esempio 10.6

Una volta entrati nelle graffe, una funzione non è nulla di nuovo: è un blocco di codice in cui possiamo mettere tutto quello che abbiamo visto finora: è semplicemente un codice separato dal resto.

In realtà, il prototipo non deve per forza esistere ed essere separato dal corpo, ma si può avere anche il solo corpo a monte del *main*, ad esempio:

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

//corpo della funzione
double Max(double array[], unsigned int dim){ //variabili della funzione
        double tmp=0; //variabili della funzione
        for(int i=1; i<dim; ++i){
        if(array[i]>max)
                tmp=array[i];
        }
        return tmp;
}

int main(){
        ...
}
```

Esempio 10.7

Qual è, allora, lo scopo di scrivere il prototipo? È una questione di leggibilità: se avessimo molte definizioni di funzioni prima del *main*, ci troveremmo una grande quantità di codice prima di arrivare a quest'ultimo (insomma, sarebbe molto dispersivo). Quando affronteremo le librerie, poi, vedremo che spesso all'utente interessa sapere soltanto come è fatta l'*interfaccia* della funzione e non come è implementata. Il prototipo ci chiarisce come dobbiamo usarla e, se è presente un breve commento, che cosa fa: in un paio di righe spiega tutto il necessario; il corpo, invece, è lungo e dispersivo.

10.1.1 *return variabile;*

L'ultima riga della funzione, però, è qualcosa di nuovo: "**return** d;". Abbiamo visto che la funzione "**double** Max(**double**[], **unsigned int**)" restituisce un **double**. Ecco, il **return** istruisce il computer riguardo a cosa deve restituire al chiamante: nel nostro caso il contenuto della varibile `d`. Ovviamente, quando incontriamo l'istruzione **return**, l'effetto è che la funzione termina restituendo il valore della variabile che segue al **return** (oppure, non restituendo niente: vedremo a breve).

Un concetto importante è che una funzione può restituire al più un solo valore: se vogliamo "ritornare" due variabili non possiamo. Vedremo nella sezione 10.3 un modo per poter avere più informazioni da una funzione; il **return**, però, non può farlo.

Detto ciò, all'interno del corpo della funzione possono esserci più istruzioni **return**: semplicemente la prima che si incontra viene eseguita (ritornando, quindi, al chiamante e terminando la funzione). Ad esempio:

```
//funzione molto elementare (si puo' fare di meglio...) che calcola se un
    numero e' primo o no
bool prime_number(unsigned long n){
```

```
        for(int i=2; i<n/2; ++i){
                if(n%i==0)//se il resto della divisione e' zero allora e' un
                    divisore
                        return false;
        }

        return true;
}
```

Esempio 10.8

Come vedi, in questo esempio sono presenti due **return**: il primo ad essere incontrato viene eseguito. In poche parole, in una funzione posso avere n **return** (ad esempio in diversi **if**, **switch...case**, ecc...), l'unica condizione è che questi devono restituire lo stesso tipo di dato: quello specificato nel prototipo della funzione.

10.1.2 *void funcion*

Non per forza una funzione deve restituire qualcosa, può anche essere il "nulla": **void**.

Un prototipo di una funzione che non restituisce niente potrebbe essere:

```
void foo(int n);
```

Esempio 10.9

Ma, all'interno del corpo, dobbiamo ancora usare la parola chiave **return**? La risposta è sia si che no, in questo senso: non siamo obbligati, ma possiamo usarla.

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

void Print(int n){
        cout << "La variabile inserita e': " << n <<endl;
}

int main(){
        int tmp=0;
        cout << "Inserisci una variabile: ";
        cin >>tmp;
        Print(tmp);
        return 0;
}
```

Esempio 10.10

Come vedi, in questo esempio la funzione non necessita di un **return** (in quanto non restituisce niente): quando l'esecuzione del suo codice arriva alla fine, la funzione termina e il chiamante riprende il controllo.

L'uso del **return**, nelle funzioni che ritornano **void**, può essere utile nel caso di **if** (o altri check) per restituire prematuramente il controllo al chiamante:

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

```
void Print_Positive(int n){
        if(n<0)
                return; //esce dalla funzione
        cout << "La variabile inserita e': " << n <<endl;
}

int main(){
        int tmp=0;
        cout << "Inserisci una variabile positiva: ";
        cin >>tmp;
        Print(tmp);
        return 0;
}
```

Esempio 10.11

Una funzione, inoltre, può non avere alcun argomento: questo si indica con le parentesi tonde vuote[1]. Ad esempio:

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

void hello(){
        cout<< "Hello World! " << endl;
}

int main(){
        hello();
        return 0;
}
```

Esempio 10.12

Ovviamente, una funzione può avere una combinazione di queste caratteristiche: il *main*, ad esempio, non ha alcun argomento ma restituisce un intero.

10.1.3 Funzioni e array

Array in argomento

Se vogliamo passare ad una funzione un *array* monodimensionale, sia dinamico che statico, è semplicissimo, le due scritture sono equivalenti:

```
void Array(int []);
//uguale a:
void Array(int *);
```

Esempio 10.13

[1]Nel vecchio C, invece, bisognava metterci un void all'interno (tipo "int foo(void)"): le parentesi tonde vuote significavano un numero di argomenti indefinito.

Il perché della loro equivalenza (o, in altri termini, perché possiamo usare un puntatore per passare *array*) dovrebbe essere chiaro.

La prima scrittura, forse, ha il pregio di rendere più chiaro che la nostra funzione vuole un *array* e non un semplice puntatore.

C'è da sottolineare che sia con il puntatore che con le quadre, la dimensione dell'*array* non è comunicata alla funzione, per cui spesso dobbiamo passarle con un'altra variabile anche questa informazione.

Nel *main* una funzione del tipo "**void** Array(**int** []) (o con il puntatore) si usa così:

```
int main(){
        int v[15];
        Array(v); //passo solo il nome dell'array: perche'? (ricorda
            equivalenza tra [] e * : v cos'e'?)
        return 0;
}
```

Esempio 10.14

Restituire array

Per restituire un *array*, l'unico modo è usare un puntatore:

```
int* InizializzaZero(int array[], unsigned int dim){
        for(unsigned int i=0; i<dim; ++i)
                array[i]=0;
        return array;
}
```

Esempio 10.15

Infatti, la scrittura "**int**[] InizializzaZero (**int** array [], unigned **int** dim)" produce un errore di compilazione (come detto: devi restituire un puntatore).

Matrici e array multidimensionali

Per quanto riguarda gli *array* multidimensionali, la questione è diversa a seconda che sia una matrice dinamica o statica. Perché? Una matrice dinamica, per come l'abbiamo costruita noi, è un puntatore a puntatore; una matrice statica, invece, no! Ti ricordi? In memoria, è un semplice *array*; il doppio indice è solo un'astrazione per il programmatore.

Nel caso di una funzione che prende in argomento una matrice dinamica e che la restituisce, il prototipo potrebbe essere:

```
int** matr(int** m, unsigned int righe, unsigned int colonne);
```

Esempio 10.16

Niente di strano, quindi.

Le funzioni in cui compaiono *array* multidimensionali statici, invece, sono un po' scomode e meno versatili.

Abbiamo visto che per un *array* monodimensionale possiamo tralasciare la dimensione nel prototipo. Invece, per quanto riguarda il caso a più dimensioni, il compilatore deve "sapere" come convertire l'*array* lineare della memoria ad un *array* multidimensionale per il programmatore. In poche parole, possiamo tralasciare solo la dimensione più a sinistra, le altre vanno fissate (perché? Prova a pensarci!).

Se abbiamo un *array* a due dimensioni (ad esempio, 4 e 5) il prototipo sarà del tipo:

```
void matr(int m[][5]);
```

Esempio 10.17

A tre dimensioni:

```
void matr(int m[][4][5]);
```

Esempio 10.18

Inoltre, ritornare un *array* statico a più dimensioni ha una sintassi davvero allucinante, per non confonderti le idee non te la riporto neanche: semplicemente, è meglio evitare situazioni del genere (o usare matrici dinamiche)[2].

Insomma, se usiamo *array* multidimensionali statici, le relative funzioni sono utilizzabili solo per *array* di cui la maggior parte delle dimensioni è fissata. Molto meglio, per quanto riguarda le funzioni, usare le matrici dinamiche: in quel caso, infatti, una funzione nata per una matrice a due dimensioni è valida qualsiasi esse siano.

10.2 La chiamata a funzione

Ciò che segue è un piccolo discorso opzionale su cosa succede quando chiamiamo una funzione.

Ho già accennato che, chiamando una funzione, questa viene caricata in memoria. Ma cosa significa? Quando compiliamo il codice, il file eseguibile contiene dentro di sé le funzioni ben distinte ognuna dall'altra, come se fossero dei blocchi unitari, ognuno con le proprie variabili e il proprio spazio dei nomi. Insomma, ogni funzione è un blocco di linguaggio macchina separato dal resto. Quando la funzione viene chiamata, ad esempio, dal *main*, il suo blocco di codice viene caricato nella RAM e la CPU passa dal *main* a lei, quasi come se fosse a sua volta un piccolo programma. Se la funzione ne chiama un'altra, il blocco di codice di quest'ultima viene a sua volta caricato in RAM e così via. Se chiamiamo n volte una funzione, il suo blocco di codice viene chiamato n volte. Ogni qual volta una funzione termina la sua esecuzione, il suo blocco di codice viene eliminato dalla RAM, e lo spazio liberato (insieme a quello di tutte le sue variabili). È una struttura "*FIFO*" ("First In First Out).

Chiamare una funzione, quindi, non è semplicemente eseguire il suo codice, ma comporta tutto un insieme di operazioni legate al caricamento in memoria: per farla breve, non è gratis, ha un costo computazionale[3]. Il discorso delle performance, come al solito, è una finezza. C'è un caso, però, un po' patologico che può essere utile nominare. Le funzioni vengono caricate nella *stack* (non nella *heap*) che, come abbiamo visto, non è particolarmente grande (te lo ricordo: di default, su Linux, è grande circa 8MB). Quando usiamo la ricorsione (questa tecnica sarà spiegata, se sei interessato, nei complementi a

[2] No, non ci riesco, se proprio sei curioso ecco come andrebbe scritta: **int** (∗matr(**int** m[][5]))[5]. Insomma, meglio lasciar stare...

[3] Questo non vale per le cosiddette "inline functions": dalla struttura identica alle funzioni, ma differenti in quanto non sono distinte dal chiamante ma sono codice fisicamente presente al suo interno.

questo capitolo, 10.B), una funzione chiama se stessa n volte; il problema, però, è che la nuova funzione viene chiamata prima che la vecchia termini l'esecuzione. Per cui, alla fine dei conti, saranno caricate in RAM, contemporaneamente, n funzioni: esaurire la *stack* non è impossibile.

Ma, allora, qual è il vantaggio delle funzioni se sono computazionalmente più pesanti del codice lineare? Il primo vantaggio è che rendono la vita più facile al programmatore (ma se fosse l'unico, allora basterebbe far lavorare qualcuno, tipo il preprocessore, per incollare il codice della funzione nel chiamante, risparmiando, così, la chiamata). Il secondo vantaggio (fondamentale) è comprensibile con il seguente esempio: abbiamo il nostro *main* che chiama qualche milione di volte una funzione. Il codice macchina avrà la seguente forma: ci saranno il *main* e la funzione compilati e distinti in due blocchi indipendenti di eseguibile; quindi, ogni volta che chiamiamo la funzione, vi sarà un'istruzione per la CPU di caricare in RAM il blocco di codice della funzione. In totale, l'eseguibile sarà relativamente piccolo: il codice del *main* più quello della funzione e poco altro. Se invece l'essenza delle funzioni fosse un semplice artificio per rendere la vita più facile al programmatore e venisse semplicemente incollata nel chiamante, il *main* avrebbe un codice enorme: ogni volta che chiamiamo la funzione verrebbe aggiunto tutto il suo codice. Il binario compilato, quindi, avrebbe dimensioni notevoli e un semplice programma che, ad esempio, calcola qualche milione di volte la media di dati, peserebbe troppo per essere caricato in RAM.

10.3 Passaggio di argomenti: *by value, by pointer e by reference*

Abbiamo visto che ogni funzione ha un proprio spazio dei nomi a cui appartengono le sue variabili. Anche gli argomenti sono variabili locali: scrivendo "**void** foo(**int** b)", `b` nasce con la funzione e vive solo al suo interno.

Il seguente esempio dovrebbe chiarire questo concetto fondamentale:

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

void incrementa(int a){ //la variabile passata alla funzione nella sua
    chiamata viene copiata dentro la variabile locale ''int a''
        a+=1; //stiamo modificando la copia locale
        cout << "Valore locale di a: " << a << endl;
}

int main(){
        int a=17;
        incrementa(a); //il valore di a viene copiato dentro la variabile
            locale: la a del main non viene toccata
        cout << "Valore di a nel main: " << a << endl; //e' rimasta immutata
        return 0;
}
```

Esempio 10.19

E se volessimo modificare il valore della variabile del *main*? Potremmo, ad esempio, restituire il valore dopo averla incrementata:

```
int incrementa(int a){
        return a+1;
}
```

Esempio 10.20

Nel *main*, dovremmo chiamare la funzione così: a=incrementa(a). In questo modo, la variabile del *main* verrebbe copiata dentro quella locale della funzione: tutti i calcoli sarebbero eseguiti su quest'ultima e, alla fine, il suo valore verrebbe restituito al *main*.

Per quanto funzionante, questo metodo ha un difetto: se vogliamo modificare il valore di più variabili del chiamante?

Se sei stato attento, magari ti è venuto in mente qualcosa... cosa ne dici dei tediosi puntatori?

Vediamo un esempio:

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

void incrementa(int* a){
        *a+=1; //incremento la variabile puntata da a
}

int main(){
        int a=17;
        incrementa(&a); //passo alla funzione l'indirizzo di a
        cout << "Valore di a nel main: " << a << endl;
        return 0;
}
```

Esempio 10.21

Con questo metodo, dopo aver chiamato la funzione, il valore di `a` del *main* sarà 18.

Cerchiamo di capire cosa accade. La variabile locale, in questo caso, è un puntatore: quando passiamo alla funzione l'indirizzo di `a`, quest'ultimo viene copiato nella variabile locale. Vi è ancora la distinzione tra variabile locale e quella del chiamante (se modificassimo il contenuto del puntatore `a`, ovvero il suo indirizzo, il valore verrebbe modificato solo localmente: l'indirizzo della variabile del chiamante rimarrebbe immutato), però la funzione possiede l'indirizzo di una variabile che appartiene al chiamante. Anche se non è sua quindi, tramite l'indirizzo è in grado di accedervi (dereferenziando il puntatore) e modificarla.

Il vantaggio di passare le variabili *by pointer* è che possiamo modificare n variabili del chiamante e non solo una, un po' come se potessimo restituire un numero arbitrario di variabili[4].

Inutile dirlo: la sintassi con l'uso dei puntatori è alquanto scomoda. Il `C++` ha introdotto un oggetto molto interessante che semplifica la vita: la *reference*. Senza entrare nei dettagli di cos'è, ti basta sapere come si usa nelle funzioni per apprezzarne la comodità:

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

[4]Ad un lettore attento potrebbe saltare all'occhio anche un altro dettaglio: se modifichiamo le variabili usando i puntatori, al posto che restituendole, il computer deve fare un passaggio in meno. Ma l'efficienza non è il nostro scopo...

```
void incrementa(int& a){ //passaggio by reference, prende da sola l'
    indirizzo della variabile passata dal chiamante
        a+=1; //avendo passato la reference del chiamante e' come se stessi
            lavorando su di lei, e non su una variabile locale
}

int main(){
        int a=17;
        incrementa(a); //passo normalmente a, ma incrementa si prende la
            reference, quindi a viene incrementata
        cout << "Valore di a nel main: " << a << endl;
        return 0;
}
```

Esempio 10.22

In poche parole, quando passiamo gli argomenti *by reference*, il meccanismo del puntatore viene reso automatico ed implicito: la funzione va a prendersi l'indirizzo della variabile passata dal chiamante e, quindi, lavora direttamente su di lei e non su di una copia locale[5].

Invece, quando ha senso utilizzare il passaggio *by value*? Nelle situazioni in cui non vogliamo che la variabile del chiamante sia modificata, ovvero quando ci fa piacere che la funzione lavori su di una copia locale. Ad esempio, una funzione che calcola la media ha senso definirla così:

```
float media(float array[], unsigned int n);
```

Esempio 10.23

La variabile `n` non deve essere modificata, per cui la passiamo per valore e non per referenza (o puntatore).

10.3.1 Allocazione dinamica dentro a funzioni

Normalmente, se passiamo un *array* ad una funzione e lo modifichiamo al suo interno, la modifica verrà vista anche dal chiamante. Perché? Ricorda: passare un *array* significa passare un puntatore, e accedere agli elementi dell'*array* significa dereferenziare il puntatore. Quello che stiamo passando per valore è il puntatore, ma l'indirizzo di memoria, nel chiamante e nella funzione, sarà lo stesso: se accediamo agli elementi puntati sono uguali sia in uno che nell'altro, una modifica nella funzione ha effetto anche nel chiamante.

Esempio banale:

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

void inizializza(int array[], unsigned int dim){ //al solito array[] e'
    equivalente a *array
```

[5] Ma quindi passare argomenti *by pointer* o *by reference* sono solo due scritture diverse ma con lo stesso effetto? Ciò è quasi sempre vero. L'anno prossimo, studiando gli oggetti, vedrai che a volte è vantaggioso passare *const reference*. Passare "**const** &" ha un effetto profondamente diverso rispetto a passare "**const** *". Se te ne ricorderai, riflettici!

```
        for(unsigned int i=0; i<dim; ++i)
                array[i]=0; //la modifica ha effetto sull'elemento puntato
                    da array e i successivi
}

int main(){
        int* a=new int[12];
        inizializza(a, 12);
        for(int i=0; i<12; ++i)
                cout << a[i] << endl; // sono stati modificati dalla
                    funzione!
        return 0;
}
```

Esempio 10.24

C'è una situazione, però, in cui passare un puntatore non ha alcun effetto sulla variabile del chiamante. È il caso in cui vogliamo allocare dinamicamente un puntatore del *main* dentro una funzione, ingenuamente potremmo scrivere questo:

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

void alloca_inizializza(int *p, unsigned int dim){
        p=new int[dim];
        for(unsigned int i=0; i<dim; ++i)
                p[i]=0;
}

int main(){
        int *v;
        alloca_inizializza(v, 20);
        for (int i=0; i<20; ++i)
                v[i]=2; //con buone probabilita' andremo in segmentation
                    fault!
        //...
        //...
        delete[] v; //se non siamo andati prima in seg fault, ci andiamo
            sicuramente qui
        return 0;
}
```

Esempio 10.25

Alla prima lettura potrebbe sembrarti tutto corretto, eppure (te lo assicuro!), è sbagliato. Ma dov'è l'errore? Rifletti: quando passiamo la variabile `v` alla nostra funzione, il suo contenuto viene copiato dentro la variabile locale `p`; essendo entrambi puntatori, il contenuto di `v` è un indirizzo di memoria (casuale, non essendo stato inizializzato). Dentro la funzione chiamiamo l'operatore **new**, il quale alloca della memoria e restituisce a `p` l'indirizzo del primo byte. Ecco l'errore! L'indirizzo viene restituito a `p` che è una variabile locale, il puntatore `v` del *main*, invece, non viene modificato: quando usciamo dalla funzione, la memoria rimane allocata ma il suo indirizzo è perso con la fine di `p`. Quando, poi, nel *main* andiamo a modificare l'area di memoria puntata da `v`, stiamo toccando memoria potenzialmente non nostra (`v` non è stato

inizializzato: a chi punta?). Se non andiamo qui in *segmentation fault* ci andremo sicuramente quando chiamiamo l'operatore **delete**: `v` non punta nessuna memoria allocata.

Insomma, non solo perdiamo per strada l'*array* che abbiamo allocato, ma poi ci facciamo anche terminare dal sistema operativo.

Come risolvere? La questione è che dobbiamo modificare il puntatore presente nel *main*: possiamo usare un puntatore a puntatore o una *reference*. Il primo caso (sintatticamente scomodo) lo lascio da affrontare a te, se ne hai voglia. Invece, analizziamo la *reference* a puntatore:

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

void alloca_inizializza(int* &p, unsigned int dim){ //p e' una reference a
    puntatore ad interi
        p=new int[dim]; //l'indirizzo del primo byte di memoria non e' piu'
            assegnato ad una variabile locale: p e' una reference, sara' il
            puntatore del chiamante ad essere modificato
        for(unsigned int i=0; i<dim; ++i)
                p[i]=0;
}

int main(){
        int *v;
        alloca_inizializza(v, 20);
        for (int i=0; i<20; ++i)
                v[i]=2; //nessun problema! v, essendo stato preso by
                    reference, e' stato modificato: punta al vettore allocato
                     dinamicamente nella funzione
        //...
        //...
        delete[] v; //tutto liscio!
        return 0;
}
```

Esempio 10.26

Tutto chiaro? La *reference* è molto più agevole di un puntatore a puntatore; inoltre, se volessimo allocare dinamicamente una matrice in una funzione, avremmo bisogno di un puntatore a puntare a puntatore (complicato, eh?). Usando una *reference* si ritorna a quanto già visto nel capitolo sull'allocazione dinamica di matrici.

10.3.2 Stream e funzioni

Ci sono oggetti, nel `C++`, che non sono *copiabili*. Uno di questi è lo *stream*. Un canale non è altro che una virtualizzazione di un *bus* di dati fisico che collega la RAM ad una piccola porzione dell'hard disk. Ovviamente, questo canale nel computer è unico. Se potessimo copiare lo *stream* avremmo, all'interno del programma, più copie di un oggetto che in realtà è unico e univoco: sarebbe un assurdo e potrebbe portare a situazioni molto poco piacevoli. Se non ti fidi prova a copiare uno *stream* nel codice: verrai insultato malamente dal compilatore.

Per questo motivo, uno *stream* può essere passato solo *by reference* o *by pointer* in quanto, passandolo *by value*, creeremmo una copia locale all'interno della funzione.

10.4 Overload di funzioni

Quando definiamo una funzione, a livello di "linguaggio", stiamo dando un nome ad un'azione.

Vi sono situazioni in cui ci verrebbe da chiamare allo stesso modo azioni simili ma leggermente diverse. Ad esempio, l'azione "mangia" cambia a seconda dell'*argomento* che dobbiamo mangiare: se è una pasta usiamo la forchetta, per un budino il cucchiaio, con pizza le mani, ecc... Se in italiano esistessero nomi diversi per ogni "sfumatura" del verbo mangiare, sarebbe un vero disastro lessicale.

Anche il C++ permette di associare ad una "parola" diverse "sfumature" e comportamenti. Questa caratteristica è detta *function overloading*.

Con questa tecnica, possiamo definire due o più funzioni indipendenti con lo stesso nome ma con liste di argomenti diverse. Ecco un esempio:

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

void Stampa(int n){
        cout << "Il numero inserito e': " << n << endl;
}

void Stampa(char a){ //overloading!
        cout << "La lettera inserita e': " << a << endl;
}

int main(){
        char c='b';
        Stampa(c); //il compilatore capisce quale funzione usare a causa
            dell'argomento
        Stampa(2); //idem
        return 0;
}
```

Esempio 10.27

Avendo argomenti diversi, il compilatore è in grado di "scegliere" la funzione corretta durante la chiamata. Un'altra differenza possibile può essere, ad esempio, un numero di argomenti diverso (ma c'è una possibile soluzione alternativa che tra poco vedremo).

Ciò che invece il C++ si rifiuta di accettare è una differenza nel solo *return value*. Mentre quando invochiamo una funzione l'argomento con cui la chiamiamo è esplicito, la variabile ritornata, tecnicamente, è indipendente da cosa vi è a sinistra della funzione. Sarebbe estremamente pericoloso accettare una cosa di questo tipo:

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

void Stampa(int n){
        cout << "Il numero inserito e': " << n << endl;
}

int Stampa(int n){ //ERRORE: non si puo' fare!
        cout << "Ti ho restituito la variabile n " << endl;
        return n;
}
```

```
int main(){
        int n=Stampa(1);
        return 0;
}
```

Esempio 10.28

Nel `C` qualcosa del genere era accettabile, ma causava complicazioni notevoli in fase di *debugging*. Il `C++` si rifiuta di compilare un *overload* nella sola *return variable*.

Nota che, invece, è possibile *ritornare* valori differenti se la lista degli argomenti è diversa:

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

void Stampa(char a){
        cout << "La lettera inserita e': " << a << endl;
}

int Stampa(int n){ //return value diverso ma accettabile: la lista degli
    argomenti e' diversa, non vi e' ambiguita' in compilazione
        cout << "Il numero inserito e': " << n << endl;
        return n;
}

int main(){
        int n=Stampa(2);
        return 0;
}
```

Esempio 10.29

Questi esempi di *overloading* sono assai stupidi, ma ci sono casi in cui è davvero comodo utilizzare questa tecnica. Ad esempio, quando si definiscono tipi di dato diversi da quelli built-in e si vogliono adattare anche a loro funzioni già costruite per i dati standard:

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

//definisco la struttura puntoR2
struct puntoR2{
        float x;
        float y;
};

int max(int array[], unsigned int dim){
        int tmp=0;
        tmp=array[0];
        for(int i=1; i<dim; ++i){
                if(array[i]>tmp)
                        tmp=array[i];
        }
        return tmp;
}
//restituisce il punto R2 con l'ordinata massima
```

```
puntoR2 max(puntoR2, unsigned int dim){
        puntoR2 tmp=0;
        tmp=array[0];
        for(int i=1; i<dim; ++i){
                if(array[i].y>tmp.y) //definisce il concetto di ordinamento
                    per la struttura
                        tmp=array[i];
        }
        return tmp;
}

int main(){
        //...
        //...
        return 0;
}
```

Esempio 10.30

Nell'esempio precedente l'*overload* è comodissimo: avremmo potuto dare nomi diversi alle funzioni (es `maxR2`), ma il codice si sarebbe complicato nel momento di chiamare la funzione (dovendoci ricordare quale funzione usare ogni volta). Grazie all'*overload*, non ci accorgiamo di avere tante funzioni diverse.

Vi è una cosa importante da notare, non puoi definire due funzioni con il prototipo uguale (anche se il corpo è differente). Farlo causerà sempre un errore in compilazione.

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

int DoSomething(int n){
        return ++n;
}

int DoSomething(int n){ //ERRORE!! Stesso prototipo del precedente come fa
    il compilatore a scegliere quale usare?
        return --n;
}

int DoSomething(int n){ //ERRORE!! Funzione gia' definita (identica alla
    prima), errore in compilazione
        return ++n;
}

int main(){
        //...
        return 0;
}
```

Esempio 10.31

Chiara la differenza tra *overloading* di funzione ed errore di programmazione?

10.4.1 Default arguments

Una caratteristica piuttosto utile del C++ è di poter assegnare degli argomenti default ad una funzione, ad esempio:

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

void Stampa(int n, unsigned int n_volte=1){ //n_volte e' un argomento di
    default: se non lo assegno vale 1
        for(unsigned int i=0; i<n_volte; ++i)
                cout << "Valore inserito: " << n << endl;
}

int main(){
        //posso chiamare la funzione stampa in due diversi modi:
        Stampa(2); //stampa 2 una volta (n_volte non viene assegnato, quindi
             e' l'argomento di default)
        Stampa(2,100); //stampa 2 cento volte (sovrascrivo l'argomento di
            default)
        return 0;
}
```

Esempio 10.32

In una funzione posso avere n argomenti default, nella chiamata a funzione, però posso lasciare vuoti solo gli argomenti più a destra: non è ammesso assegnare un valore ad un argomento e lasciare quello di default alla sua sinistra. Per chiarire meglio:

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

void foo(int n=0, bool b=true, float f=1.1){
 //...
 //...
}

int main(){
        //questo e' ammesso:
        foo(); //tutti gli argomenti vengono passati alla funzione con il
            valore di default

        //anche questo e' ammesso
        foo(1, false); //f rimane 1.1

        //errore! non posso assegnare un valor a b senza che tutti gli
            argomenti alla sua sinistra abbiano un valore assegnato nella
            chiamata
        foo(false); //non ho assegnato n ma ho assegnato b

        return 0;
}
```

Esempio 10.33

Ultima cosa da dire: se dividi il prototipo dal corpo della funzione, gli argomenti di default devono comparire solo nel prototipo. Ad esempio:

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

//prototipo: ha gli argomenti di default
void foo(int n=0, bool b=true, float f=1.1);

int main(){
        //...
        //...
        return 0;
}

//nel corpo non devo piu' assegnare il valore agli argomenti, in caso
    contrario ho errore di compilazione
void foo(int n, bool b, float f){
        //...
        //...
}
```

Esempio 10.34

10.5 Alcune funzioni utili

Di seguito riporto alcuni esempi concreti di funzioni. In particolare, una funzione per caricare da file un *array* di dati con dimensione non nota a priori.

10.5.1 Lettura da file: dimensione non nota

Abbiamo un file, `dati.dat`, che contiene un numero di dati che non ci è noto. Sappiamo solo il tipo di dato, per esempio **int**. Ecco il codice:

```
#include <iostream>
#include <fstream>      //ifstream
#include <cstdlib>      //exit()
using namespace std;

void CaricaInteri(const char *nomefile, int* &v, unsigned int &dim){ //
    passiamo il nome del file come una stringa del C: un'array di caratteri
    costante
        ifstream in(nomefile);
        int tmp=0;
        if(in.fail()){
                cerr << "Errore di apertura canale in lettura con " <<
                    nomefile << endl;
                exit(1);
        }

        //Per prima cosa devo contare i dati su file
        dim=0; //inizializzo il numero dei dati a zero
        in >> tmp; //leggo un dato
```

```
        while(!in.eof()){ //se non ho letto il carattere end of file
            continuo il ciclo
                dim++; //incremento il numero di dati
                in >> tmp; //leggo altro dato
        }
        //ora ho il numero di dati, contenuto in dim, quindi alloco un
            vettore di quella dimensione e riporto la testina della lettura
            del file a zero
        v=new int[dim];
        in.clear();
        in.seekg(0);
        //ora carico i dati
        for(int i=0; i<dim;++i)
                in>>v[i];
        in.close(); //chiudo il canale
}

int main(){
        int *v;
        unsigned int n;
        CaricaInteri("dati.dat", v, n);
        //uso il mio vettore
        //...
        //...
        delete[] v; //ricordati di pulire la memoria usata!
        return 0;
}
```

Esempio 10.35

Questa funzione non dovrebbe farti sorgere grandi dubbi: è un riassunto di quanto detto nel presente capitolo e nei precedenti. Puoi provare a scrivere una funzione simile che lavori, ad esempio, su un file di **float**. Per esercizio può essere utile anche scrivere le funzioni che, al posto di leggere, scrivono su file.

10.5.2 Random numbers

In questa sottosezione impareremo, più che altro, ad usare una funzione già scritta, ma in ogni caso conoscerla è importante.

La questione dei numeri casuali, in Fisica e in Matematica, ricopre un'importanza fondamentale. In Matematica diversi metodi numerici per risolvere, ad esempio, integrali o equazioni differenziali, richiedono l'uso di numeri generati casualmente. In Fisica, invece, moltissime simulazioni di esperimenti o di fenomeni fisici partono proprio da numeri casuali generati secondo varie distribuzioni.

In queste note impareremo a produrre numeri detti "pseudo-casuali": la funzione che useremo genera numeri che appaiono casuali ma in realtà non lo sono; in altri termini, questa funzione non andrebbe *mai* usata per fare cose serie. D'altro canto, come può una macchina deterministica come un computer produrre numeri veramente casuali?[6]

[6] In realtà, Linux ha un file molto particolare, */dev/random* (o il più versatile */dev/urandom*), dentro cui sono immagazzinati numeri *realmente* casuali (e da dove escono? Linux, ad esempio, registra il momento in

La funzione incriminata è *rand()*, che è contenuta nella libreria `cstdlib`: è un generatore di numeri interi pseudo casuali uniformemente distribuiti tra 0 e `RAND_MAX` (una costante definita sempre in `cstdlib`).

Il generatore di numeri casuali lineare ha bisogno di essere "inizializzato" con un *seme* (*seed*, in inglese). A partire da questo seme genera i numeri successivi. Se il seme fosse sempre lo stesso, genererebbe ogni volta *gli stessi numeri* (per questo viene detto "pseudo-casuale"). La funzione *srand(int)* inizializza il *seme* all'intero che gli passiamo in argomento. Un piccolo "trucchetto" è usare come *seme* il tempo del sistema operativo. La funzione *time(NULL)* (contenuta nella libreria `ctime`) restituisce i secondi passati dal primo gennaio del 1970 (quindi, a secondi di distanza, siamo sicuri che i nostri numeri casuali saranno completamente diversi). Una volta inizializzato il *seme*, possiamo generare i numeri, vediamo come:

```
#include <iostream>
#include <ctime>        //time(NULL)
#include <cstdlib>      //rand(), srand()
using namespace std;

int main(){
        int v[5]; //generiamo cinque numeri casuali
        srand(time(NULL)); //la funzione srand inizializza il seme all'
            intero restituito da time
        for(int i=0; i<5; ++i)
                v[i]=rand(); //generiamo i numeri pseudocasuali
        for(int i=0; i<5;++i)
                cout << v[i] << endl; //li stampiamo
        return 0;
}
```

Esempio 10.36

Nel codice precedente generiamo cinque numeri casuali. Se riesci ad eseguire due volte di seguito il programma abbastanza rapidamente, ti accorgerai che i cinque numeri della prima esecuzione saranno gli stessi della seconda: se è passato meno di un secondo il *seme* è lo stesso! Oppure, per verificarlo, scrivi un codice che genera n numeri casuali e, come *seme*, metti un numero costante, tipo 2: ogni volta che esegui il programma i numeri sono gli stessi.

A questo punto, siamo pronti a scrivere una funzione che genera, ad esempio, **float** uniformemente distribuiti tra due numeri che specifichiamo noi (e non interi compresi tra zero e RAND_MAX), vediamo come:

```
#include <iostream>
#include <ctime>                    //time(NULL)
#include <cstdlib>                  //rand(), srand()
using namespace std;

//Ricorda di inizializzare il seme prima di chiamarla!!!
float generaFloat(float min, float max){
        float r;
        r = (max-min)*rand()/RAND_MAX + min; //genera numeri uniformemente
            distribuiti tra min e max, perche' non e' necessario mettere (
```

cui tu inserisci una chiavetta usb, oppure il momento in cui c'è un errore di sistema dovuto a cause esterne, o il movimento del mause, ecc ecc... tutti eventi casuali). Sono byte indipendenti, e quindi vanno letti come char (la cui dimensione è, appunto, un byte), se sei interessato all'argomento scrivi "man random.4" sul terminale.

```
            float)RAND_MAX? Ricorda le operazioni vengono eseguite da
            sinistra a destra
        return r;
}

int main(){
        float* v;
        float min=0, max=0;
        unsigned int n=0;
        cout << "Inserisci quanti numeri generare: ";
        cin >> n;
        cout << "Inserisci minimo: ";
        cin >> min;
        cout << "Inserisci massimo: ";
        cin >> max;
        v=new float[n];
        srand(time(NULL));
        for(unsigned int i=0; i<n; ++i)
                v[i]=generaFloat(min, max);

        cout << "Numeri casuali tra " << min << " e " << max << endl;
        for(unsigned int i=0; i<n; ++i)
                cout << v[i] << endl;
        delete[] v;
        return 0;
}
```

Esempio 10.37

Ora prova a scrivere una funzione che genera numeri interi uniformemente distribuiti tra un massimo e un minimo arbitrari. Come risolveresti i problemi di divisione tra interi, conversione tra **float** e **int** e l'essere certo che nessuno dei due estremi sia più probabile dell'altro?

È possibile scrivere le funzione per i numeri casuali **float** e **int** in modo che una sia l'overload dell'altra?

Nel corso di Trattamento Numerico dei Dati Sperimentali, imparerai a generare numeri casuali distribuiti secondo funzioni arbitrarie, in particolare con distribuzione gaussiana (che, come saprai, è onnipresente in Fisica). Va detto che, in realtà, tutto il necessario (e anche ben di più, con tanto di distrubuzioni binomiali, T di student, ecc...) è già presente dentro la libreria "`random`", nata con lo standard del `C++` del 2011. Sempre in quella libreria, sono definiti diversi generatori di numeri casuali di cui, alcuni, migliori della funzione *rand()* che abbiamo usato[7]. Se sei interessato all'argomento, potresti dare un occhio alla reference online (http://www.cplusplus.com/reference/random/).

[7] Addirittura, vi è un oggetto, ovvero *std::random_device*, che su Linux va a leggere */dev/urandom* (che ho già citato nella nota 6) che, quindi, genera numeri non deterministici, se l'entropia del sistema è sufficientemente alta. Allora perché si usano numeri pseudo-casuali? Essenzialmente per motivi di performance: il computer è velocissimo a generare un numero pseudo-random, mentre "random_device" può essere lento (in attesa che l'entropia del sistema aumenti).

10.5.3 Calcolo del tempo di esecuzione

Per scoprire cosa sta andando storto, o semplicemente imparare a capire qualcosa di più del C++ e degli algoritmi, può essere utile imparare a misurare i tempi di esecuzione del nostro programma o di porzioni del codice.

Di nuovo, dobbiamo usare una funzione già scritta da altri: tutti gli strumenti necessari sono contenuti nella libreria standard del C (`#include <stdlib>`).

Il modo più preciso per calcolare il tempo trascorso è di misurare i cicli di *clock* che la CPU compie. Ti sarà capitato di notare, a fianco del nome dei processori, scritte del tipo "2.4GHz": indicano la velocità della CPU, in altri termini il numero di *clock* che è in grado di elaborare in un secondo. Senza entrare nei dettagli, un *clock* è il calcolo più elementare possibile di una CPU: il più piccolo "step" di pensiero che il processore sà fare.

Un programma può chiedere alla CPU di contare i cicli di *clock* che trascorrono, e quindi, dividendo per il periodo di un ciclo, si ottiene il tempo reale passato.

Un esempio dovrebbe chiarire tutto:

```
#include <iostream>
#include <cstdlib> //clock(), clock_t, CLOCKS_PER_SEC
using namespace std;

int main(){
        clock_t start, end; //definisco due variabili di tipo clock_t:
            possono immagazzinare un dato che rappresenta i clock della cpu

        start=clock(); //la funzione clock ritorna il clock attuale della
            cpu: assegno a start il clock iniziale

        //Codice di cui voglio calcolare il tempo di esecuzione
        //...
        //...
        //...
        end=clock(); //assegno ad end il clock finale di esecuzione
        float time=(float)(end-start)/CLOCKS_PER_SEC; //calcolo il tempo
            trascorso (in secondi) dividendo il delta clock per il numero di
            clock per secondo (variabile definita per ogni CPU). NOTA: sia
            clock_t che CLOCKS_PER_SEC sono di tipo intero (motivo del (float
            ))
        cout << "Tempo trascorso: " << time << " secondi." << endl;
        return 0;
}
```

Esempio 10.38

10.5.4 Funzioni di ordinamento e statistica

Alcune "azioni" che abbiamo già studiato consistono nel trovare l'elemento massimo e minimo di un vettore, ordinarlo, trovarne la media, la varianza, la deviazione standard, ecc... Inutile dire che un buon esercizio per te potrebbe essere scrivere quello di scrivere le relative funzioni!

COMPLEMENTI

10.A Argomenti del *main*: *argc* e *argv*

Abbiamo visto che pure il *main* è una funzione: è la madre di tutte le funzioni. Finora è sempre stato del tipo "**int** main()", ovvero una funzione **void** in argomento che restituisce un **int**. Esiste, in realtà, un *overload* del *main*, ovvero: "**int** main(**int** argc, **char**** argv)". Come vedi, questo *main* prende due argomenti, e prima di analizzare cosa sono, ci chiediamo: da chi li prende? Dal chiamante, ovvero il sistema operativo che, nei nostri casi, è rappresentato dal terminal.

Questo *overload* del *main* ci permette di passare dalla riga di comando degli argomenti al nostro programma (quindi delle informazioni, dei dati). Il primo, `argc`, è un intero che rappresenta il numero di argomenti passati. Il secondo è un *array* di *array* di caratteri che, tradotto, è un'*array* di stringhe.

Argc, in realtà, non dobbiamo passarlo esplicitamente, è il sistema operativo che se ne occupa: conta lui il numero di argomenti che stiamo passando. Noi dobbiamo solamente passare le stringhe di caratteri.

Ecco un esempio banale:

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

int main(int argc, char** argv){
        cout << "Numero di argomenti: " << argc << endl;
        for(int i=0; i<argc; ++i)
                cout << argv[i] << endl;
        return 0;
}
```

Esempio 10.39

Chiamando il *main* come "`./main ciao come stai?`", otteniamo il seguente output:

```
Numero di argomenti: 4
./main
ciao
come
stai?
```

In poche parole, il sistema operativo automaticamente conta per noi il numero di argomenti: sono quattro (il primo è nome del programma). Quindi, stampiamo i quattro elementi del vettore di stringhe *argc* (e, come vedi, il primo elemento è proprio il nome del programma).

Gli argomenti del *main* possono diventare molto utili: ad esempio, per passare al programma il nome del file su cui scrivere, oppure il numero di passi da fare in un ciclo, ecc...

Un esempio un po' più significativo può essere il seguente (nota: la funzione *atoi*, contenuta in `ctstlib`, converte una stringa ad un intero). Il programma vuole come argomenti il nome del file su cui scrivere e il numero di dati casuali da generare:

```
#include <iostream>
#include <cstdlib> //atoi
#include <fstream>
using namespace std;

int main(int argc, char** argv){
        if(argc!=3){//controllo che il numero di argomenti sia esattamente
            il necessario
                cout << "Usage: " << argv[0] << " nomefile numerodati" <<
                    endl; //./main nomefile numerodati
        return 1;
        }
        int ndati=atoi(argv[2]); //converto il numero di dati passato da
            riga di comando ad un intero: gli argomenti del main sono array
            di char
        ofstream out(argv[1]); //apro un canale di out usando il nomefile
            passato da argomento al main
        srand(time(NULL));
        for(int i=0; i<ndati; ++i)
                out << rand() << endl;
        out.close();
        return 0;
}
```

Esempio 10.40

Come vedi, ho controllato che il numero di argomenti passati al *main* sia esattamente quello richiesto dal programma: nessuno mi assicura che l'utente finale non passi più o meno argomenti del necessario, e una situazione del genere, spesso, può portare a comportamenti non definiti che è meglio evitare.

10.B La Ricorsione

La ricorsione è una tecnica molto interessante degli algoritmi. È strutturata implementando un algoritmo che chiama e richiama se stesso fino a che non si verifica una condizione base. Ovviamente, è importantissimo definire un caso base che faccia uscire dal *loop* infinito che, in caso opposto, si genererebbe.

La ricorsione permette di scrivere algoritmi semplici ed eleganti per problemi che, utilizzando una soluzione iterativa, richiederebbero un codice più complesso.

Il vantaggio della ricorsione è l'eleganza e la semplicità dell'algoritmo, ma purtroppo, siccome la chiamata a funzione ha un costo elevato, è una tecnica inefficiente. Al contrario, gli equivalenti algoritmi iterativi sono assai più efficienti, ma spesso decisamente più complicati da scrivere.

L'esempio più classico che si riporta parlando di ricorsione è il calcolo del fattoriale di un numero. Per calcolare $n!$ possiamo invocare $n \cdot (n-1)!$, ricordandoci che $0! = 1! = 1$ (il caso base).

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

int fattoriale(int n){
        if(n> 1)
                return n*fattoriale(n-1); //richiamo me stesso
        else
                return 1; //caso base!
}

int main(){
        int n=0;
        cout <<"Inserisci un numero di cui calcolare il fattoriale: ";
        cin>>n;
        cout <<"Il fattoriale di " << n << " e': " << fattoriale(n) << endl;
        return 0;
}
```

Esempio 10.41

Nota che il calcolo del fattoriale comporta molto rapidamente l'*overflow*, utilizzando **int**.

Un altro esempio classico è la serie di Fibonacci, che potresti provare ad implementare per esercizio.

Anche i vettori, a volte, si prestano ad essere affrontati per ricorsione. Ad esempio, prova a scrivere un algoritmo ricorsivo che trova il massimo di un *array*.

Da notare che un *loop* infinito nella ricorsione porta prima o poi ad esaurire la *stack* e, quindi, all'interruzione del programma da parte del sistema operativo. Al contrario, un *loop* infinito in un algoritmo iterativo, generalmente, non comporta l'esaurimento della memoria ma solo l'uso completo e inutile della CPU: il nostro programma si "inchioda" all'infinito (e ci conviene "ucciderlo" con control+c per uscire).

10.C Variabili *static*

Abbiamo visto che ogni funzione ha il proprio spazio dei nomi: le variabili nascono quando la funzione viene chiamata e muoiono quando termina.

Esiste un tipo di variabile che "sopravvive" anche dopo la fine della funzione e che rimane accessibile nelle successive chiamate: la variabile **static**.

Una variabile definita **static** viene allocata e inizializzata una sola volta, ovvero la prima volta che chiamiamo la funzione: quando questa termina, la variabile rimane allocata. In poche parole, è una variabile che è condivisa tra tutte le successive chiamate a funzione, con l'effetto di "salvare" il dato tra una chiamata e l'altra. Può, quindi, essere utile per tenere traccia di informazioni.

Vediamo un esempio:

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

```
void conta(){
        static int contatore=1;
        cout << "Questa funzione e' stata chiamata " << contatore << " volte
            ." << endl;
        ++contatore;
}

int main(){
        for(int i=0; i<5; ++i)
                conta();
        return 0;
}
```

Esempio 10.42

L'output del programma sarà:

```
Questa funzione e' stata chiamata 1 volte.
Questa funzione e' stata chiamata 2 volte.
Questa funzione e' stata chiamata 3 volte.
Questa funzione e' stata chiamata 4 volte.
Questa funzione e' stata chiamata 5 volte.
```

Ommettendo la parola chiave **static**, avremmo stampato per cinque volte "Questa funzione e' stata chiamata 1 volte.", in quanto la variabile sarebbe morta e rinata ad ogni chiamata della funzione.

Da notare due cose. La prima è che, se inizializziamo una variabile **static**, questo avviene solo alla prima chiamata della funzione (il "`contatore=1`" dell'esempio ha effetto solo la prima volta che la funzione viene invocata). La seconda è che la parola chiave **static**, all'interno di una funzione, non aumenta la visibilità della variabile[8]: sopravvive sì in chiamate successive, ma non diventa visibile (e accessibile) al di fuori del corpo della funzione. Se cerco di accederci dal *main*, ad esempio, otterrò un errore di compilazione.

[8] Al contrario, la parola chiave **static** modifica l'ambito di visibilità per una variabile globale: lo restringe al file in cui è dichiarata. Una variabile globale **static** è visibile solo nel file in cui è dichiarata; leggendo il capitolo sulle librerie, capirai meglio cosa si intende per "file" di codice.

CAPITOLO 11

Librerie

In un file "main.cpp", abbiamo definito una serie di funzioni di statistica per lavorare sui dati di un esperimento. Qualche giorno dopo, facciamo un nuovo esperimento e, per l'analisi dati, dobbiamo scrivere un programma; le funzioni scritte il giorno precedente ci sono utili e quindi le copiamo dal vecchio "main.cpp". Ogni volta che vogliamo fare un po' di statistica, copiamo il codice da vecchi file. Decisamente scomodo!

Inoltre, il nostro "main.cpp" potrebbe avere decine, se non centinaia, di funzioni, e le sue dimensioni diventerebbero spropositate: difficilmente leggibile e poco chiaro.

Per fortuna il `C++` (così come il `C` e moltissimi altri linguaggi) ha uno strumento comodissimo: le librerie.

Una libreria non è altro che una collezione di funzioni, oggetti e variabili "stand alone" (di per sé una libreria non fa nulla). Possiamo, però, avere tutto ciò che contiene a disposizione nel nostro "main.cpp" (o in un qualsiasi altro file di codice).

Inutile dire che, sin dal nostro primo "hello world", abbiamo usato librerie (*iostream*, ad esempio): ormai dovresti sapere benissimo come si includono (o, più correttamente, come si includono gli *header*) e si usano. La domanda lecita, invece, è: come si definiscono?

11.1 Definire una libreria: *header* e *.cpp*

In una libreria, è fondamentale separare la dichiarazione dalla definizione; l'interfaccia dall'implementazione.

Una libreria è composta da due file (almeno): uno è detto "*header*" e contiene le interfacce (prototipi, dichiarazioni, ecc...), l'altro (che termina in un'estensione del `C++` come .cpp) contiene tutte le definizioni.

Un concetto sottile è che, potenzialmente, l'utente finale potrebbe avere accesso solo all'*header*. Una libreria è po' come un computer: se siamo l'utente finale lo vediamo da fuori, lo possiamo usare e sfruttare le sue funzioni, ma normalmente non vediamo come è fatto dentro. Se poi siamo curiosi, vi sono computer che possiamo aprire, smontare e studiare senza

che scada la garanzia; altri sono più restrittivi e, se lo facciamo, perdiamo la garanzia. Di alcune librerie, dette "proprietarie", ad esempio, non possiamo vedere il file ".cpp", in altre, dette "open-source", siamo liberi di farlo. Ma sto divagando.

Torniamo alla nostra libreria con funzioni di statistica, e prepariamo due file: "statistica.h" (l'*header*) e "statistica.cpp".

```
#ifndef STATISTICA_H
#define STATISTICA_H

//Funzione che calcola la media, in ingresso un vettore di float e la sua
    dimensione: restituisce la media
float Media(float [], unsigned int);

//funzione che calcola varianza
//...

//funzione che....
//...

#endif /*STATISTICA_H*/
```

Esempio 11.1: statistica.h

```
#include "statistica.h" //file locale, non libreria standard: ci vogliono
    gli apici e non i <...>

float Media(float v[], unsigned int n){
        float media=0;
        for(unsigned int i=0; i<n; ++i)
                media+=v[i];
        media/=n;
        return media;
}

//...   (definizione delle altre funzioni)


//...
```

Esempio 11.2: statistica.cpp

Ecco fatto: al di là di quei curiosi "*#ifndef ...*", "*#define ...*" e "*#endif*" non c'è nulla di strano: un file contiene i prototipi delle funzioni, l'altro le definizioni.

Per capire il significato e il motivo di quelle direttive preprocessore, dobbiamo prima studiare come agisce il compilatore.

Abbiamo il nostro "main.cpp" in cui, ad esempio, usiamo la funzione *Media()*:

```
#include <iostream>
#include "statistica.h" //file locale: ci vogliono apici e non <...>
using namespace std;

int main(){
```

```
        float v[3]={1.01,6,12.73};
        cout << "Media: "<< Media(v, 3) << endl;
        return 0;
}
```

Esempio 11.3: main.cpp

Come ho sottolineato, per includere l'*header* di qualsiasi libreria che non sia "standard" (cioè che non appartenga agli standard del `C` o `C++`), dobbiamo usare le virgolette al posto dei segni di maggiore e minore. Inoltre, se il file non è nella cartella dove stiamo lavorando, tra le virgolette ci vuole il *path* completo! (Vedi l'appendice A sull'uso di Linux per ulteriori informazioni sul significato di "percorso completo").

Proviamo a compilare con "`g++ main.cpp -o main`": per tutta risposta riceviamo uno strano messaggio:

```
g++     main.cpp   -o main
/tmp/ccRF16WN.o: In function ‘main’:
main.cpp:(.text+0x2f): undefined reference to ‘Media(float*,unsigned int)’
collect2: error: ld returned 1 exit status
```

Cos'è successo?

Se ti ricordi, il preprocessore è "stupido": quando scriviamo *#include "statistica.h"*, semplicemente prende tutto ciò che è contenuto in *statistica.h* e lo incolla al posto della direttiva presente nel *main*. Quindi, il file finale composto dal preprocessore si ritrova con il prototipo delle funzioni ma non l'implementazione, dato che non è contenuta in *statistica.h* ma in un altro file.

Il compilatore prova a compilare ma gli manca un pezzo: nel *main* c'è una referenza (una sorta di cartellino con scritto "qui devi usare ecc...") ad una certa funzione "Media(float[], unsigned int)", ma da nessuna parte riesce a trovare una "spiegazione" di come questa è fatta. Vedi l'ultima riga? "*ld returned 1 exit status*": *ld* è il *linker*, colui che deve mettere insieme i pezzi, il quale non ha trovato tutto il necessario e fallisce.

Sono sicuro che stai pensando: ma perché non mettiamo le definizioni dentro l'*header*? Assolutamente da non fare!

Il modo giusto di procedere è la cosiddetta *compilazione parziale*.

11.1.1 Compilazione parziale: preprocessore, linker e *#ifndef*

Prova a scrivere sul terminale "`g++ -c main.cpp`": il compilatore, senza lamentarsi, creerà un misterioso file: *main.o*. Se provi ad eseguirlo, a seconda del tuo sistema operativo, potresti ottenere un errore del tipo "*bash: ./main.o: cannot execute binary file: Exec format error*" oppure "*bash: ./main.o: Permission denied*".

Main.o è un file *oggetto*, un file compilato parzialmente[1]. Il compilatore crea codice macchina dove possibile ma, quando incontra la chiamata a funzione "Media()", non sapendo come è fatta, lascia una "reference": una sorta di frase con scritto "qui manca qualcosa, bisogna attaccarcelo" (il "-c" istruisce il compilatore a procedere in questo modo). Nota: il compilatore, con l'opzione "-c" non si lamenta solo se è presente, da qualche parte, il prototipo della funzione (ad esempio nel .h che includiamo). Se, invece, non vi è né definizione né prototipo, anche con un "-c", produrrà un errore del tipo: "*error: 'funzione' was not declared in this scope*"

Torniamo al nostro *statistica.cpp* e facciamo la stessa cosa: "`g++ -c statistica.cpp`". Il compilatore crea codice macchina delle funzioni (un altro file oggetto: *statistica.o*), solo loro senza nient'altro . Come si "incollano" i due file oggetto? È il *linker* l'unico in grado di farlo: se scriviamo "`g++ main.o statistica.o -o main`" il linker cerca la referenza alla funzione mancante nel *main.o*; la trova in *statistica.o* e incolla i due pezzi per creare il codice finale ed eseguibile[2]. Tutto chiaro? Un po' macchinoso ma niente di complesso.

Ma quindi, quel *#ifndef ...* a cosa serve?

La nostra libreria (*statistica.h*) potrebbe essere inclusa in una seconda libreria (diciamo, per esempio, *tuboKundt.h*), la quale, a sua volta, è inclusa nel *main*. Il preprocessore quando incontra *#include "statistica.h"* nel *main* incolla lì tutto e fa lo stesso quando lo trova dentro *tuboKundt.h*. Poi, nel *main*, incontra *#include "tuboKundt.h"* e incolla lì tutto, creando una doppia copia di alcune cose presenti in *statistica.h* (particolarmente sensibili a questi errori sono le definizioni di nuovi tipi di dato come le **struct**).

Una ridefinizione porta ad un errore fatale in compilazione, del tipo:

```
In file included from foo.h:1:0,
                 from main.cpp:2:
media.h:1:8: error: redefinition of ‘struct foo’
   struct foo{
          ^~~~~~
In file included from main.cpp:1:0:
```

È importante istruire il preprocessore a non incollare ovunque il *.h*, ma a farlo solo una volta. La scrittura:

```
#ifndef STATISTICA_H
#define STATISTICA_H
...
...
#endif /*STATISTICA_H*/
```

Esempio 11.4

[1] È scritto, quindi, in "linguaggio macchina". Quando usiamo una libreria closed-source, ci vengono forniti l'header e il file oggetto: in questo modo non possiamo sapere come è implementata (e neanche copiare/modificare/chissà che altro...).

[2] Il *linker*, in ogni caso, fa sempre qualcosa di simile: il nostro programma deve sempre essere "linkato" con le librerie di sistema per poter diventare un vero e proprio file eseguibile.

dice, in poche parole: "se non è definita la macro STATISTICA_H (if not defined, *ifndef*), allora definiscila (*define*)", Alla fine del *.h*, vi è la conclusione della definizione "*endif*" (il /*...*/ è un commento per il programmatore: definizione di cosa?). Quando il preprocessore trova l'*include*, cerca di copiare il *.h* ma, se la macro STATISTICA_H non è stata definita, allora procede (e la definisce), se invece è stata già definita (quindi ha già incolato tutto altrove), salta alla fine del'*endif* e va oltre (con il risultato di non incollare tutto due volte). "STATISTICA_H" è scritto maiuscolo per ricordare che è una macro preprocessore, come da convenzione del `C++`.

Per esercizio, scriviti le tue librerie di statistica, di algoritmi su vettori, ecc...

11.2 Il Makefile

Cambiamo programma, ora abbiamo un *main* che usa diverse librerie. Per creare l'eseguibile, dobbiamo compilare parzialmente ogni libreria e quindi linkare tutto insieme; una sorta di:

```
g++ -c main.cpp
g++ -c lib1.cpp
g++ -c lib2.cpp
g++ -c lib2.cpp
...
...
g++ main.o lib1.o lib2.o lib3.o .... -o main
```

Estremamente scomodo, tanto più che, ogni volta che cambiamo un file di codice, dobbiamo ricompilarlo e poi chiamare di nuovo il linker (l'ultimo comando).

Make è una piccola (ma complessa) utility che permette di automatizzare la compilazione: per farlo, ha bisogno di un file di "comandi", detto *makefile*.

Al nostro livello, scrivere un *makefile* è semplicissimo. Ipotizziamo di avere i file "main.cpp", "statistica.h", "statistica.cpp", "ordinamento.h" e "ordinamento.cpp"; il *makefile* andrebbe scritto così:

```
compila: main.o statistica.o ordinamento.o
	g++ main.o statistica.o ordinamento.o -o main
main.o: main.cpp
	g++ main.cpp -c
statistica.o: statistica.h statistica.cpp
	g++ statistica.cpp -c
ordinamento.o: ordinamento.h ordinamento.cpp
	g++ ordinamento.cpp -c
esegui: main
	./main
clean:
	rm *.o main
```

Esempio 11.5: makefile

Il *makefile* definisce una serie di regole. Una *regola* è una parola seguita dai due punti (tipo "compila:'). Dopo i due punti vanno inserite le cose di cui ha bisogno, ad esempio i file necessari per la compilazione (nel nostro caso tutti i file oggetto). Il comando associato alla regola si definisce la riga sottostante, dopo uno spazio di tab (che è obbligatorio, perché senza il comando non viene riconosciuto): nel nostro caso è un comando per il compilatore.

Nel mio *makefile*, successivamente, ho definito le regole per costruire i file oggetto.

La regola deve avere il nome del file oggetto in questione, ad esempio, "statistica.o". Di cosa ha bisogno? Dell'*header* e del *.cpp*. Quindi, a capo e con uno spazio di tab, vi è il comando di compilazione.

Come lavora *make*? È molto più intelligente di quanto possa sembrare. Quando scriviamo "*make compila*", controlla la data in cui è stato creato il file eseguibile e la confronta con le date di ultima modifica dei file di cui ha bisogno la regola; se non sono stati modificati da quando *main* è stato compilato, allora non fa nulla (non ci sarebbe bisogno di ricompilare!). Se le date non coincidono, invece, passa alle regole che definiscono i vari "*.o*". Prendiamo "statistica.o": per prima cosa controlla la data di ultima modifica di "statistica.h" e "statistica.cpp". Sono più nuovi di ".o"? Se sì compila, se no, no, e così via. Questa tecnica è molto utile quando abbiamo centinaia, se non migliaia, di file di codice da compilare: se modifichiamo un solo file vengono ricompilati giusto un paio di file e non tutti (ad esempio: compilare ROOT impiega circa mezz'ora, se gli sviluppatori modificano un file e vogliono testarlo, non devono aspettare tutto quel tempo in attesa che la nuova modifica abbia effetto).

Se scriviamo sul terminal "make" senza un comando, lui esegue il primo della lista, per cui è comodo aver scritto per primo il comando finale della catena di compilazione. Da notare che se non definiamo tutte le regole, *make* non sa come fare e si ferma.

L'ultima regola che ho definito è *clean* (che va chiamato con "make clean"): rimuove tutti i file oggetto e il file eseguibile, utile per forzare la ricompilazione di tutto (o per pulire la cartella). Attento, è "*.o": se ti dimentichi il ".o" e lasci solo "*", potresti combinare un vero disastro eliminando tutto ciò che è nella cartella!

Può sembrarti molto scomodo dover scrivere il *makefile* (che deve chiamarsi proprio "makefile" ed essere nella cartella in cui lanciamo il comando "make"), ma il vantaggio è che va scritto solo una volta: poi ci basta chiamare *make* al posto di dover riscrivere decine di comandi di compilazione.

CAPITOLO 12

Containers: strutture di dati

TO BE WRITTEN

12.1 La Queue

12.2 Il Vettore

CAPITOLO 13

ROOT CERN

ROOT è un framework sviluppato al CERN per l'analisi dati. È utilizzato per plottare (disegnare) dati su grafici, visualizzare istogrammi, fittare dati, ecc...

Essenzialmente è una libreria che, una volta inclusa, permette al nostro programma di rappresentare graficamente numeri e dati, oltre ad essere dotata di diverse altre funzioni. Per sapere come installare ROOT, prova a leggere l'appendice C.

La figura 13.1, ad esempio, è realizzata con ROOT: è un plot di dati sperimentali raccolti da un rivelatore di raggi gamma. In poche parole, rappresenta lo spettro di radiazioni gamma emesse da una sorgente di radio 226: con ROOT possiamo scrivere programmi in `C++` che producono output di questo tipo.

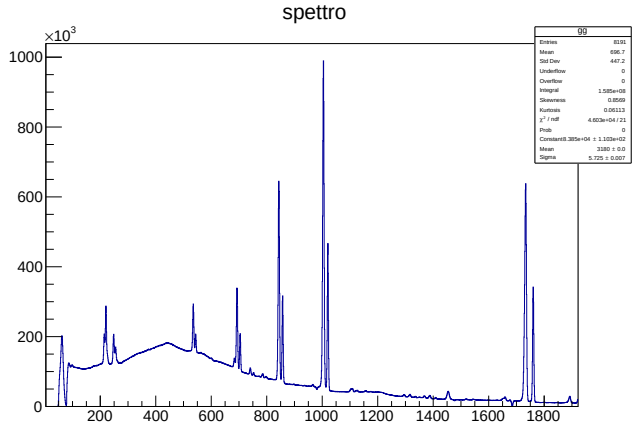

Figura 13.1: Radio 226

Questo capitolo sarà decisamente "hands on": c'è poco da capire, semplicemente dobbiamo imparare ad usare una libreria scritta da qualcun altro.

13.1 Librerie da includere, flag di compilazione, TApplication e TCanvas

Per poter usare ROOT, dobbiamo includere le librerie necessarie per avere a disposizione alcuni strumenti nel nostro codice.

Due liberie che non possono mai mancare sono "`TApplication.h`" e "`TCanvas.h`". La prima contiene il "motore" di ROOT (la classe TApplication), la seconda lo strumento "TCanvas", ovvero le "tele" su cui disegnare gli oggetti di ROOT.

Purtroppo, mentre il compilatore sa benissimo dove andare a cercare le librerie standard (come "`iostream`"), non ha idea di dove si trovano le librerie di ROOT. Vi sono due comandi che restituiscono al compilatore un'indicazione di dove andarle a cercare: "*root-config --cflags*" e "*root-config --libs*": la prima indica il path degli *headers*, la seconda le librerie da includere.

È comodo definire due macro nel makefile. Le macro vengono espanse di fianco ai vari comandi, un esempio può essere il seguente (per un programma composto da "main.cpp", "lib.h" e "lib.cpp"):

```
INCS=`root-config --cflags` #macro
LIBS=`root-config --libs`   #macro

compila: main.o lib.o
	g++ main.o lib.o -o main ${INCS} ${LIBS}
main.o: main.cpp
	g++ main.cpp -c ${INCS}
lib.o: lib.h lib.cpp
	g++ lib.cpp -c ${INCS}
esegui:
	./main
clean:
	rm *.o main
```

Esempio 13.1

Come vedi, ho definito due macro: *INCS* e *LIBS*. Nel makefile si definiscono con "nome=contenuto". Da notare che il comando di ROOT è incluso tra apici gravi (control+apostrofo): le parole tra apici gravi non sono semplici stringhe ma sono comandi.

A questo punto, si possono richiamare le macro con "${nome}": la macro viene espansa lì dove è riportata (viene effettuata una sostituzione con il comando). Nota che *INCS* (gli *headers*) sono presenti ovunque, mentre *LIBS* (le librerie) serve solo quando deve operare il linker.

Nel *main*, dobbiamo dichiarare ed usare TApplication e TCanvas:

```
#include <iostream>
#include "TApplication.h"       //Non e' una lib standard, vanno usate le
    virgolette
#include "TCanvas.h"
using namespace std;

int main(){
	TApplication app("app", 0,0); //vanno passati tre argomenti: una
	    stringa con il nome e due zeri
	TCanvas canvas;
```

```
        //...
        //...

        app.run(); //alla fine si usa questo per far partire ROOT
        return 0;
}
```

Esempio 13.2

Tutti i programmi che usano ROOT devono avere almeno queste righe di codice. Ora cerchiamo di disegnare qualcosa sulla tela!

13.2 Plot e Istogrammi

Utilizzando ROOT, impareremo a realizzare plot di dati, istogrammi e, infine, a disegnare funzioni su plot.

Un plot di dati è, ad esempio, quello in figura 13.2, che rappresenta una Lorentziana. Un

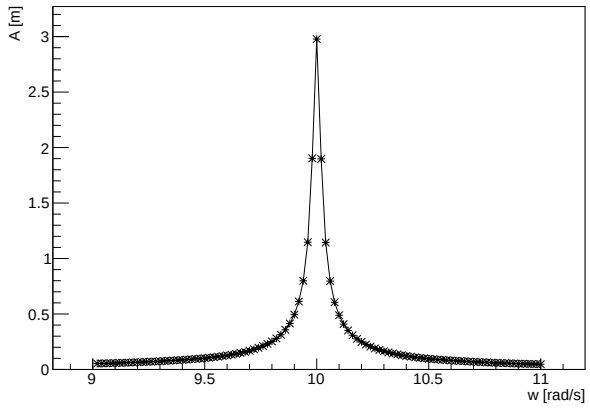

Figura 13.2: Plot: una lorentziana

istogramma, invece, è quello in figura 13.3.

13.2.1 TH1F

Per disegnare un istogramma, la libreria da usare è "TH1F.h". Vediamo il codice:

```
#include <iostream>
#include "TApplication.h"
#include "TCanvas.h"
#include "TH1F.h"
using namespace std;

int main(){
        TApplication app("app", 0,0);
        TCanvas canvas;
        int n_bin=10; //numero di bin in cui dividere l'istogramma, il
            numero di ''barre'' dell'istogramma
```

```
        float min=3.3, max=6.7; //valore minimo e massimo per l'ascissa dell
            'istogramma (i ''canali'')
        TH1F histo("Istogramma", "Somme", n_bin, min, max); //nome da far
            apparire nella tabellina riassuntiva, nome da far apparire in
            alto, numero di bin, canale minimo e canale massimo
        //...
        //...
        for(int i=0; i<n_punti; ++i)
                histo.fill(v[i]); //v e' un vettore contenente i nostri
                    punti
        canvas.cd(); //seleziono la canvas su cui disegnare
        histo.Draw(); //disegno l'istogramma
        app.Run(); //faccio partire ROOT
        return 0;
}
```

Esempio 13.3

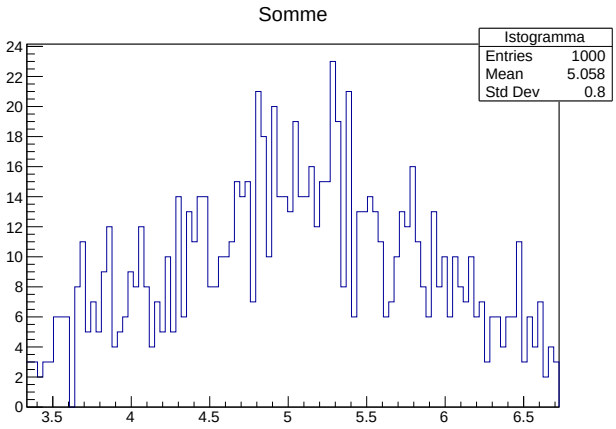

Figura 13.3: Istogramma

Quando dichiariamo un oggetto, la classe istogramma vuole come parametri (tra le tonde) i seguenti elementi: il nome da far apparire nella tabella riassuntiva (in alto a destra della figura 13.3), il nome da far apparire in alto nella canvas ("Somme" in figura 13.3), il numero di bin in cui suddividere l'istogramma (il numero di barre) e, infine, il valore del bin minimo e massimo (ad esempio, sempre in figura 13.3 sono rispettivamente 3.3 e 6.7). A questo punto, riempiamo l'istogramma con i nostri dati tramite il metodo *Fill(*dato_da_inserire*)*: la classe *TH1F*, da sola, posiziona il dato nel bin corretto e costruisce l'istogramma (sulle ordinate vi è, quindi, il numero di dati per ogni bin).

Infine, dobbiamo selezionare la canvas su cui vogliamo disegnare (possiamo, infatti, avere più canvas nello stesso programma) e, tramite il metodo *Draw()*, stampare l'istograma. Quindi, dobbiamo far "partire" l'applet di ROOT con il metodo *Run()* della classe *TApplication*.

Un'alternativa consiste nel riempire noi i bin senza lasciare che venga fatto in modo automatico: dobbiamo usare il metodo *SetBinContent(*i, n_dati*)* che permette di inserire n_dati nell'i-esimo bin (dove i, tanto per uniformarsi al C++, parte da 1...).

Il seguente esempio riempie casualmente i cinque bin di un istogramma, e il risultato è mostrato in figura 13.4:

```
#include <cstdlib>
#include <ctime>
#include "TApplication.h"
#include "TCanvas.h"
#include "TH1F.h"
using namespace std;

int main(){
        TApplication app("app", 0,0);
        TCanvas canvas;
        TH1F histo("Dati", "Istogramma", 5, 0,5); //nome nella tabella dei
            dati, nome in alto, numero di bin, valore bin minimo, valore bin
            massimo (in questo caso questi ultimi due valore non servono a
            niente: siamo noi a riempire)

        srand(time(NULL));
        for(int i=0; i<5;++i) //riempio ogni bin con un numero di dati
            casuale
                histo.SetBinContent(i+1, rand()); //i parte da 1
        canvas.cd();
        histo.Draw();
        app.Run();

        return 0;
}
```

Esempio 13.4

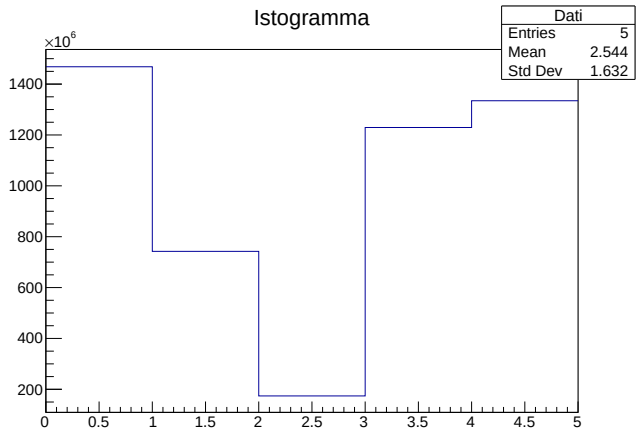

Figura 13.4: Istogramma casuale

Per informazione, la figura all'inizio del capitolo (13.1) è stata realizzata proprio in questo modo: è stata usata la classe *TH1F* riempiendo manualmente i bin (in ogni bin vi è il numero di particelle che ha colpito il rivelatore a quella data energia). Non si nota che è un istogramma in quanto i bin sono più di ottomila.

13.2.2 TGraph

La classe *TGraph* (contenuta nella libreria *TGraph.h*) permette, invece, di realizzare un plot di dati. Molto semplicemente, si utilizza il metodo *SetPoint(i, x, y)*: i è l'i-esimo punto (che questa volta, invece, parte da zero), x è l'ordinata e y l'ascissa. Vediamo un esempio.

Il codice che segue disegna 100 punti con ordinata e ascissa uniformemente distribuite tra 0 e 1:

```
#include <cstdlib>
#include <ctime>
#include "TApplication.h"
#include "TCanvas.h"
#include "TGraph.h"
using namespace std;

int main(){
        TApplication app("app", 0,0);
        TCanvas canvas;
        TGraph grafico;
        srand(time(NULL));
        for(int i=0; i<100;++i)
                grafico.SetPoint(i,rand()/(float)RAND_MAX, rand()/(float)
                    RAND_MAX); //Perche' ho scritto il (float)? Se non lo
                     metto che numeri (o meglio: che numero) vengono generati?
        canvas.cd();
        grafico.Draw("A*"); //quando disegno un grafico devo dargli delle
            opzioni
        app.Run();
        return 0;
}
```

Esempio 13.5

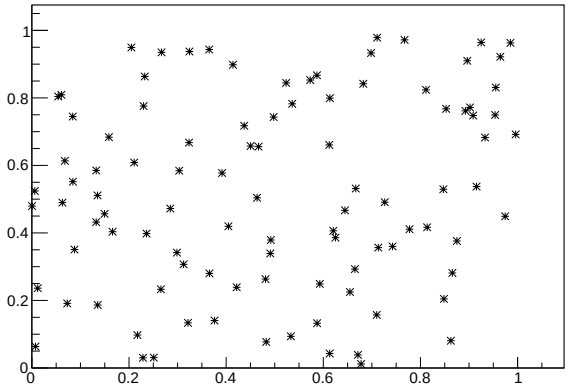

Figura 13.5: Plot di dati random

L'unica differenza rispetto agli esempi precedenti è questa: quando disegno un grafico devo dire alla classe *TGraph* come rappresentare i punti. Per farlo, si passa una stringa al metodo

Draw: la "A" indica di disegnare gli assi del grafico; l' "*" indica di disegnare i punti con degli asterischi. Il grafico in figura 13.2 aveva in più l'opzione "L" che indica di collegare i punti con una riga continua: infatti, avevo passato la stringa "AL*".

Il risultato del codice dell'esempio 13.5 è la figura 13.5.

13.2.3 Nomi degli assi e funzioni

Potremmo desiderare di mettere grandezze e unità di misura sugli assi. La classe *TGraph* supporta questa funzionalità tramite un metodo che restituisce un oggetto *TAxis*, il quale permette di inserire nomi sugli assi (va inclusa la libreria *TAxis.h*). Vediamo come:

```
#include <iostream>
#include "TApplication.h"
#include "TCanvas.h"
#include "TGraph.h"
#include "TCanvas.h"
#include "TAxis.h"
using namespace std;

int main(){
        TApplication app("app", 0,0);
        TCanvas canvas;
        TGraph grapico;

        //riempio il grafico
        //...
        //...

        canvas.cd();
        grafico.GetXaxis()->SetTitle("V [V]"); //Nome all'asse X
        punti.GetYaxis()->SetTitle("I [mA]"); //Nome all'asse Y
        grafico.Draw("A*");
        app.Run();
        return 0;

}
```

Esempio 13.6

L'esempio 13.7 permette di dare i nomi agli assi come in figura 13.6. Da notare cosa ho scritto: si accede al metodo *GetXaxis()* (o Y) dell'oggetto *TGraph*. Questo, quindi, restituisce un puntatore ad un oggetto della classe *TAxis*: perciò, dobbiamo usare l'operatore "->" per accedere al metodo *SetTitle("nome")*.

L'ultima cosa che ci rimane da imparare è disegnare una funzione matematica sopra ad un grafico. La questione è piuttosto semplice: si utilizza la classe *TF1* contenuta nella libreria *TF1.h*. Ecco un esempio:

```
#include <iostream>
#include "TApplication.h"
#include "TCanvas.h"
#include "TGraph.h"
#include "TCanvas.h"
#include "TAxis.h"
```

```
#include "TF1.h"
using namespace std;

int main(){
        TApplication app("app", 0,0);
        TCanvas canvas;
        TGraph grapico;

        //riempio il grafico
        //...
        //...

        canvas.cd();
        grafico.GetXaxis()->SetTitle("V [V]");
        punti.GetYaxis()->SetTitle("I [mA]");
        grafico.Draw("A*");

        //Funzone retta
        TF1 retta("Retta", "[0]*x+[1]",min , max); //in ordine: nome,
            formula parametrica, definita tra minimo e massimo
        float m=10.;
        float q=0.;
        retta.SetParameter(0, m); //assegno valore al parametro "[0]"
        retta.SetParameter(1, q); //assegno valore al parametro "[1]"
        retta.Draw("SAMEL"); //SAME sta per sulla stessa canvas selezionata
            prima, sopra al grafico, L per usa una linea
        app.Run();
        return 0;
}
```

Esempio 13.7

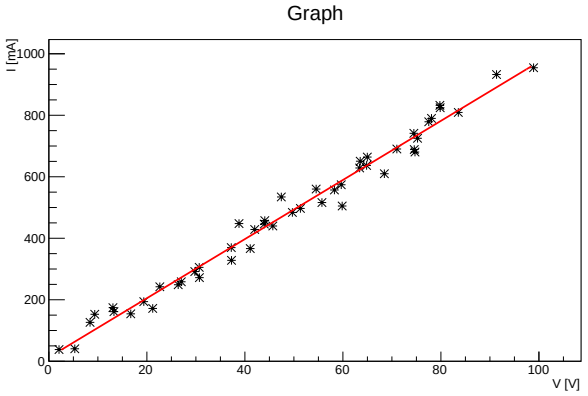

Figura 13.6: Fit lineare

ROOT, oltre alle funzioni analizzate, ha tante altre caratteristiche. Se vai sul sito di ROOT e accedi alla reference (https://root.cern.ch/documentation), puoi trovare informazioni su tutto quello che è disponibile e su come usarlo. Va detto che la reference di ROOT non è eccellente e spesso è davvero difficoltoso capire come utilizzare i suoi strumenti.

Una nota conclusiva su ROOT

ROOT non è l'unica soluzione che permette di affrontare le problematiche del plot di dati, disegnare funzioni, analisi dati, ecc... A dirla tutta, vi sono linguaggi orientati esclusivamente a queste finalità, alcuni molto semplici da utilizzare.

Per l'analisi dati, ad esempio, *Matlab* e *Octave* possono risultare molto più potenti, efficaci e facili da usare (i fit, anche delle funzioni più esotiche, si realizzano in qualche di riga di codice). Un linguaggio di programmazione di livello più alto del `C++` che dispone di bellissime librerie orientate al calcolo scientifico è *Python* (librerie utilissime sono *matplotlib*, *scipy* e *numpy*). Se, invece, devi solo plottare dati o graficare funzioni (facendo, magari, qualche fit) il leggerissimo e semplice programma *Gnuplot* può essere una soluzione valida. Se, al contrario, vuoi usare il computer per risolvere problemi di Analisi matematica, Geometria, ecc..., magari in maniera analitica e non numerica, sapere dell'esistenza di *Mathematica* (purtroppo non free) potrebbe esserti molto utile.

Ovviamente non sto insinuando che tu debba impararti tutti questi linguaggi ma, se ti intriga conoscere strumenti diversi, perché non dare un'occhiata? Il vantaggio di conoscere più linguaggi è di saper scegliere, di fronte ad un problema, quello che porta alla soluzione in maniera più rapida, efficiente e comoda.

APPENDICE A

Rapida guida all'uso di Linux

Nel corso di laurea in Fisica andrai inevitabilmente a scontrarti con Linux[1]. In primis, per questo esame, per cui è il caso di imparare sin da subito alcuni comandi base: non è piacevole brancolare nel buio!

Lo strumento più potente ed importante di questo sistema operativo è il *terminale*. Cos'è? Normalmente siamo abituati ad interfacciarci con il nostro PC per via grafica: visualizziamo le cartelle, i file, i contenuti... Il terminal è il computer privato di grafica! In realtà, su Linux, la grafica è un'aggiunta, ma l'intero sistema operativo esiste a prescindere da essa.

Apri un *terminale* e cominciamo! Ah già, come? Ctrl+alt+T su Ubuntu, oppure cercalo nelle utility di sistema. Ti apparirà una finestra vuota con in alto a sinistra un *nome@altro-nome altro*. Il primo è il tuo nome utente, dopo la @ c'è il nome del computer. "Altro" (che non necessariamente è presente) tipicamente indica la cartella dove ti trovi(vedi A.1). Ci siamo: il modo per interfacciarsi al computer, privato della grafica, è scrivendo comandi (per dare un comando, scrivilo e premi invio) e ricevendo output di soli caratteri.

Il comando base, che spesso viene ignorato, è **man**: sta per *manual*. Prova a scrivere "man man" e, come per tutti i comandi, premere invio. Ti apparirà la *manual page* del comando scritto alla destra di "man" (che in questo caso è di nuovo "man"): in poche parole il manuale del comando. Se non sai come funziona un comando o non ti ricordi come si usa, il modo più rapido ed efficace è scrivere proprio "man comando" e ti verrà mostrato il suo manuale. Per uscire da una *man page* schiaccia "q" (quit).

[1]Ma cos'è Linux? È un "kernel", il cuore di un sistema operativo. Anche Android usa Linux come kernel. Quando noi diciamo "Linux", sottointendiamo, invece,"GNU/Linux", una famiglia di sistemi operativi. Di GNU/Linux esistono varie "distribuzioni": Debian, Ubuntu, Arch, Manjaro, ecc.... Per fare un parallelo, Windows è una famiglia di sistemi operativi, Windows 10 è un sistema operativo, e Windows NT è il kernel di Windows. In queste note parlo, per semplicità, di Linux come sistema operativo: permettimi l'abuso di linguaggio.

A.1 Il filesystem

Il filesystem, senza entrare troppo nei dettagli, è come vengono organizzati in memoria file e cartelle.

Linux si comporta come un albero: partendo dalla *radice* –*"root"*–, il filesystem si dirama nelle varie cartelle –*"directory"*–. Dentro ogni cartella sono contenuti i file e le cartelle successive. Una pulce nell'orecchio: su Linux tutto è un file, anche le cartelle.

A.1.1 Navigare nel filesystem

Se sul tuo terminale scrivi "**ls**" (che pressapoco significa "list directory contents"), puoi visualizzare il contenuto della cartella in cui ti trovi.

Ogni cartella è quindi un ramo dell'albero; partendo dalla *radice*, la quale è rappresentata da / (lo slash).

In figura A.1 puoi vedere la rappresentazione grafica della *radice* con le sue sottocartelle. In figura A.2, invece, c'è l'output del terminale. Come vedi, quest'ultimo è composto dai nomi delle sottocartelle e dei file contenuti nella cartella in cui ti trovi.

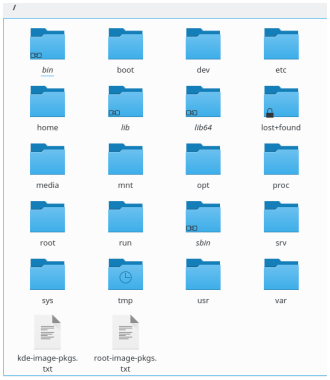

Figura A.1

Figura A.2

Non ci interesseremo a cosa sono tutte queste cartelle, ce n'è solo una che ti deve importare: "home". Dentro vi sono le cartelle base di tutti gli utenti, la loro *home*. Ogni utente può accedere solo alla propria.

Ma come ci si sposta tra le cartelle? Il comando è **cd** (change directory), che alla sua destra vuole la cartella in cui ci vogliamo spostare. Abbiamo visto che il simbolo della *radice* è lo *slash*, il simbolo della tua home è invece la tilde (`~`). I comandi `cd ~` e `cd /` ti faranno spostare, rispettivamente, nella mia *home* e nella *radice*. Se lanci "`cd`" senza alcun argomento, ti sposterai nella tua *home* (è equivalente alla tilde).

Bisogna subito imparare l'importantissima differenza tra *percorso assoluto* e *percorso relativo*.

Poniamo di essere l'utente "pippo" e di avere una cartella, nella nostra *home*, chiamata "fuffa". Se ci troviamo nella *home* il comando per spostarsi in fuffa sarà `cd fuffa`. Perché? Semplicemente, il percorso *relativo* di questa cartella rispetto a dove ci troviamo è "fuffa", si trova lì! Se invece ci troviamo in un posto qualsiasi (diverso dalla *home*), possiamo scrivere il percorso *assoluto* di fuffa. Il comando sarà `cd /home/pippo/fuffa`: i percorsi assoluti sono indicati partendo dalla *radice*, e le sottocartelle si separano con degli slash. Quindi, partendo dalla *radice*, a grandi linee, significa "spostati in *home* (che contiene le *home* di tutti gli utenti), quindi in pippo (la *home* di pippo), quindi in fuffa".

Chiaro? Il *percorso assoluto* è espresso indicando tutte le sottocartelle a partire dalla *radice*, il *percorso relativo* è espresso indicando le sottocartelle a partire da dove ci troviamo.

Spesso lavoriamo in cartelle che appartengono alla nostra *home*; per abbreviare, al posto del percorso completo possiamo usare `cd ~/percorso`: la tilde iniziale esprime il fatto che il percorso deve essere inteso a partire dalla nostra *home*.

In Linux ci sono due cartelle particolarissime, presenti ovunque: la cartella "punto" (si, proprio " . ") e la cartella "punto punto" (" .. "). La prima rappresenta la cartella stessa in cui siamo (se sono nella cartella fuffa e scrivo `cd .` mi sposto nella cartella in cui sono, ovvero non mi sposto). La cartella ".." rappresenta quella precedente nell'albero: se sono in "fuffa" e scrivo `cd ..` mi sposto in "pippo" (la cartella precedente).

Fai dunque attenzione: i comandi `cd fuffa/fuffa1/fuffa2` e `cd /fuffa/fuffa1/fuffa2` sono diversi! (Questa differenza è fondamentale, e fonte di errori enormi!)

Mentre i comandi `cd fuffa/fuffa1/fuffa2` e `cd ./fuffa/fuffa1/fuffa2` sono uguali! Per il comando `cd`, mettere o non mettere uno slash alla fine del percorso è indifferente: `cd fuffa` e `cd fuffa/` sono equivalenti (e volendo anche `cd fuffa/.`). Complicato? Con un po' di pratica diventerà tutto ovvio.

E se ci siamo persi? Il comando "**pwd**" restituisce il percorso assoluto della nostra posizione.

A.1.2 Modificare il filesystem

Ora che abbiamo visto come muoverci nel filesystem di Linux è arrivato il momento di capire come creare, rimuovere e modificare cartelle e file.

Il comando "**mkdir** *nomecartella*" crea una cartella nell'attuale posizione con il nome alla sua destra (mkdir: make directory, facile!). È anche possibile esplicitare il percorso dove vogliamo creare la cartella: "`mkdir /home/pippo/nuovacartella`" è un comando valido (supponendo che vogliamo creare *nuovacartella* dentro la cartella *pippo*).

Per creare un file vuoto il comando è "**touch** *nomefile*". Una nota: in Linux, le estensioni non hanno alcun significato; per cui, un file di testo può essere chiamato "file", "file.txt", "file.dat", "file.quellochevuoi", ecc...

Per aprire (ed eventualmente creare, se non esiste) un file con il nostro editor di testo preferito, il comando è "editor nomefile", per cui, ad esempio usando gedit: "`gedit file.txt`". Il problema di un comando di questo tipo è che il terminale lancia l'editor di testo e rimane in uno stato di attesa: fino a che l'editor non viene chiuso non possiamo più agire con il terminale. Esiste un trucco: porre una "&" alla fine del comando, in questo modo ci viene restituito il terminale dal programma lanciato e possiamo continuare ad utilizzarlo (nota, però, che se chiudiamo il terminale muore anche il programma). Quindi, alla fine dei conti, il comando più efficace per aprire un file con gedit è "`gedit file &`".

Per rimuovere un file, il comando è "**rm** *nomefile*" (rm sta per remove). Per rimuovere una cartella dobbiamo renderlo ricorsivo, ovvero rimuovere anche tutti i contenuti della cartella; per farlo si usa "**rm -r** *nomecartella*". ATTENTO: su Linux rm non sposta in alcun "cestino", elimina senza possibilità di ritorno.

Per copiare un file usiamo: "**cp** *percorso-iniziale percorso-di-arrivo*". Se vogliamo copiare il file nella stessa cartella allora useremo "`cp nomefile nomecopia`". Per le cartelle bisogna copiare anche tutti i file contenuti, quindi usare cp in modo ricorsivo: "**cp -r** *percorso-cartella-da-copiare percorso-copia*".

L'ultimo comando base utile è "**mv**" (move); mv è ambivalente, può significare sia "rinomina" sia "sposta" (ma, se ci rifletti bene, in realtà sono la stessa cosa: "mv" cambia il percorso nel filesystem). Se scriviamo "`mv vecchionome nuovonome`" stiamo rinominando: se ci pensi, un indirizzo nel filesystem è del tipo /x/y/z/a/b/... Scrivendo "vecchionome" e "nuovonome" sottintendiamo che tutto quello che viene prima rimane immutato, cambia solo l'ultima parte dell'indirizzo, ovvero il nome! Per le cartelle non bisogna utilizare il "-r" (prova a riflettere: perché non bisogna rendere il comando ricorsivo? Perché basta cambiare l'indirizzo della cartella e non anche di tutte le cose contenute?).
Se scriviamo "`mv vecchio-percorso nuovo-percorso`" stiamo spostando il file o la cartella.

Per tutti i comandi elencati in A.1.2 usare *percorso assoluto* o *relativo* è indifferente.

A.2 Secure Shell: Linux in remoto

Come ti avranno detto nel corso di Informatica, puoi accedere in remoto al laboratorio di calcolo e lavorare da casa.

Un enorme pregio di Linux è quello che si può accedere ad una *shell* (un *terminale*) in remoto. Esistono anche programmi per Windows che ti permettono di farlo ma, data la mia scarsissima conoscenza di quel sistema operativo, in queste note non troverai istruzioni per farlo... Chiedo venia e ti rimando al sito del corso di Informatica dove puoi cercare "putty". Se invece sei un Mac-user o hai una partizione con Linux, allora è facilissimo: "`ssh nomeutente@nomeserver`". Se ti vuoi collegare al laboratorio di calcolo, il comando sarà: "`ssh tuonomeutente@tolab.fisica.unimi.it`" (generalmente il tuo nome utente è quello della mail di Ateneo). Se vuoi avere anche una sessione grafica (ad esempio per aprire programmi come gedit, altrimenti disponi solo del terminale) devi usare "`ssh -X`" (X volutamente maiuscola).

Una volta entrato nel server di destinazione, è come se ti trovassi fisicamente su quel computer: il terminale è di quel computer e valgono tutti i comandi spiegati precedentemente. Quando hai finito di lavorare e desideri ritornare sul tuo computer ti basta usare il comando "`exit`".

Un altro comando utile è "**scp**" (secure copy). Molto semplicemente, si comporta come il comando cp, tenendo conto, però, che uno dei due indirizzi è sul server di destinazione. Ecco degli esempi pratici:

- Copiare un file locale su un server di destinazione:

  `scp indirizzo_file nomeutente@nomeserver:indirizzo_destinazione` (nessuno spazio dopo i due punti!)

- Copiare un file remoto sul mio computer:

  `scp nomeutente@nomeservr:indirizzo_file_destinazione indirizzo_su_mio_computer`

Due precisazioni: la prima è che per copiare una cartella bisogna usare il comando ricorsivo "`scp -r`"; la seconda precisazione è che gli indirizzi sul server di destinazione sono relativi alla tua *home*, ovviamente puoi inserirli in maniera completa, utilizzando quelli assoluti.

APPENDICE B

La reference online

In tutti gli esempi affrontati, abbiamo avuto modo di utilizzare funzioni contenute in alcune librerie, dette *standard*, del `C++` (iostream, cstdlib, fstream, ...).

La libreria del `C++` è estremamente più estesa di quanto abbiamo visto e usato, ha funzioni di ogni tipo: dai contenitori al multithreading, dai numeri complessi agli algoritmi di base.

Ovviamente, per quanto un programmatore sia esperto, non è richiesta la conoscenza di tutta la libreria: per questo motivo, sono presenti diverse reference online, di cui la più valida forse è http://www.cplusplus.com/reference/.

A sinistra, nel menù a tendina, trovi tutte le librerie raggruppate in cinque macro categorie: *C Library* (tutte quelle che iniziano con `<c...>`), *Containers* (classi di vettori, code, liste ecc...), *Input/output* (iostream, fstream, sstream -per lo stream di stringhe-), *multithreading* (libreria piuttosto avanzata che richiederebbe un intero corso dedicato per poterla usare con disinvoltura) e *other* (contiene un po' di tutto, tra cui cose utilissime ed estremamente interessanti, come *string*, *complex*, *random*, *chrono*, ecc...).

Prova ad aprire, ad esempio, la macro categoria *C library* e qui selezionare la libreria *ctime*. Ti si aprirà una pagina in cui sono elencate tutte le funzioni, le classi, i tipi di dato e le costanti definite nella libreria.

Selezioniamo una voce, ad esempio, la funzione *time*. Ti si aprirà una pagina divisa in sezioni: la descrizione generale con il prototipo della funzione; i parametri da passare; il *return value* e, infine, un esempio concreto di come usare la funzione.

Ti consiglio vivamente di consultare spesso la reference quando vuoi capire di più su funzioni e classi varie o per avere un'idea di cosa offre la libreria del `C++`: programmando tanto è uno strumento fondamentale!

Concludo con un piccolo suggerimento. Spesso ti chiederai "ma esiste una funzione che faccia...?", gli informatici ti direbbero: GIYF, ovvero Google Is Your Friend. Cerca su internet e, con ottime probabilità, troverai la risposta alla tua domanda; quindi, una volta scoperto il nome della funzione, puoi controllare sulla reference come si usa.

APPENDICE C

Installazione di ROOT

Installare ROOT è, in realtà, molto più semplice di quanto si pensa.

Il procedimento è diverso a seconda del sistema operativo che stai usando:

- Su Linux e Mac la questione è un po' "laboriosa" ma semplice e non presenta criticità. Vai alla sezione C.1 se sei un utente Linux e alla sezione C.2 se usi Mac.
- Se programmi su Windows (tipo con Visual Studio?) allora... auguri! No, seriamente: auguri!!
  Se ci riesci allora complimenti. Se invece non hai idea di come fare, fatti una partizione con Linux, oppure installati Linux in VirtualBox, ma lascia stare Windows!

C.1 Installare su Linux

Per prima cosa vai a questo indirizzo `https://root.cern.ch/downloading-root`, quindi seleziona, sotto le "Lastest ROOT releases" la "PRO Release". Nella pagina che ti si aprirà cerca, sotto "binary distributions", se c'è il nome della tua distribuzione Linux (ad esempio Ubuntu 16 –che sta per 16.04–, o Ubuntu 14). Si aprono due possibilità: la tua distribuzione è presente o no, nel primo caso vai a C.1.1 se no a C.1.2.

C.1.1 La tua distribuzione c'è!

Clicca sul link sotto il nome della tua distribuzione; ti si scaricherà un file compresso. Una volta scompattato devi scegliere una cartella in cui installare ROOT. Puoi installarlo nella tua Home, oppure, se non vuoi avere la sua cartella tra le scatole in /opt (in questo caso dovrai usare "sudo" prima dei comandi!). Sposta la cartella scompattata nella sua destinazione finale ("`mv root destinazione`, o "`sudo mv root /opt/.`"), a questo punto il gioco è praticamente fatto: devi solo dire a Linux dove si trova ROOT. Per farlo apri con un editor il file "`~/.bashrc`" (ad esempio: "`gedit ~/.bashrc''`"), quindi, in fondo al file, inserisci "`. PERCORSO/root/bin/thisroot.sh`" (se la cartella di root si chiama in altro modo metti il suo nome al posto di "root"). NOTA: il punto e lo spazio all'inizio sono voluti, non dimenticarli!

Salva il file, esci dal terminale, apri un nuovo terminale e scrivi "`root`", se il programma si apre ROOT è installato e hai finito!

C.1.2 Sei sfortunato...

Al CERN non hanno pensato di fare una versione precompilata di ROOT per la tua distribuzione di Linux, poco male[1]!

Scarica, sotto "source distribution", il file compresso. Una volta scompattato devi scegliere una cartella in cui installare ROOT. Puoi installarlo nella tua *home*, oppure, se non vuoi avere la sua cartella tra le scatole in */opt* (in questo caso dovrai usare "*sudo*" prima dei comandi!). Sposta la cartella scompattata nella sua destinazione finale ("`mv root-versione destinazione`", o "`sudo mv root-versione /opt/.`").

Ora, bisogna compilare ROOT, ti avviso impiegherà un bel po' di tempo, e userà un sacco di risorse, per cui cerca di attaccare la presa del PC e dotarti di pazienza. Per la compilazione avrai bisogno di alcuni pacchetti, assicurati di avere tutto il necessario dando un'occhiata alla pagina https://root.cern.ch/build-prerequisites.

Portati dentro la cartella di ROOT, quindi lancia "`./configure`", se qualche pacchetto necessario manca ti verrà segnalato. Se tutto va a buon fine lancia "`make -j4`" (il numero che segue la "j" è il numero di processori da usare, quattro è un buon numero).

Quando avrà finito devi solo dire a Linux dove si trova ROOT. Per farlo apri con un editor il file "`~/.bashrc`" (ad esempio: "`gedit ~/.bashrc''`"), quindi, in fondo al file, inserisci:

"*. PERCORSO/root-versione/bin/thisroot.sh*" (dove, al posto di "root-versione" metti il nome della cartella di ROOT). NOTA: il punto e lo spazio all'inizio sono voluti, non dimenticarli!

Salva il file, esci dal terminale, apri un nuovo terminale e scrivi "`root`", se il programma si apre ROOT è installato e hai finito!

C.2 Installare su Mac

Per prima cosa vai a questo indirizzo https://root.cern.ch/downloading-root, quindi seleziona, sotto le "Lastest ROOT releases" la "PRO Release". Nella pagina che ti si aprirà, sotto "binary distributions", cerca il nome la versione del tuo *OsX* (es. 10.10), e del tuo compilatore *clang*. Quindi, scarica il file *.tar.gz* relativo ad essi.

A questo punto, scompatta l'archivio e posiziona la cartella di ROOT dove preferisci. Apri un terminale e spostati al suo interno, quindi scrivi `cd bin`, e poi `pwd`. Copia il percorso che ti viene restituito e, con un editor di testo, apri il file *.bash_profile* che si trova nella tua *home* (se non esiste crealo). In fondo al file scrivi "*. percorso_copiato_precedentemente/thisroot.sh.*", salva ed esci dal terminale. NOTA: il punto e lo spazio all'inizio sono voluti, non dimenticarli!

Ora, se ne apri uno nuovo, avrai ROOT perfettamente istallato e funzionante (prova a scrivere `root` per controllare).

[1] Se usi Arch Linux o una sua derivata puoi trovare ROOT precompilato nei repository AUR. Usa yaourt per cercarlo e potrai installarlo come un semplice pacchetto evitandoti tutto quello che segue.

APPENDICE D

Un esame di laboratorio risolto

Questa appendice l'ho realizzata per dare un'idea di come andrebbe scritto un codice in linea con le richieste di Informatica 1. Ho cercato di usare lo stile e gli strumenti più semplici possibili: il minimo richiesto per superare l'esame a pieni voti! Volendo si possono usare tanti altri strumenti: classi, template, ecc ecc... Ma se sai queste cose, guardare questo svolgimento per te è probabilmente inutile! A proposito del "tutto giusto": mi sono concentrato sul codice, per quanto riguarda la correttezza dei risultati... non assicuro nulla!

Cos'è apprezzato nell'esame di Informatica 1, ovvero cosa devi fare per prendere un bel voto?

- I risultati devono essere corretti (ma questo è ovvio...).
- Il codice deve essere comprensibile e leggibile; potrà sembrarti una cavolata, ma ad ogni parentesi graffa dopo dai i dovuti spazi di tab per renderlo ordinato!
- Il codice deve essere ben strutturato: devi scrivere il meno possibile nel main, dove invece dovresti limitarti a poco più di semplici chiamate a funzione. Ogni volta che puoi, e che ha senso farlo, scrivi una funzione che esegua un'insieme di operazioni.
- Dividi il codice in librerie: le funzioni non tenerle nello stesso file del main, ma crea una, o piu', librerie.
- Scrivi un makefile ben fatto e funzionante.
- Stai attento a quelli che possono sembrarti dettagli ma che sono importanti in un buon codice: ogni volta che apri uno stream ad un file controlla che non sia corrotto ma che sia funzionante; quando non ti serve più lo stream ricordati di chiuderlo e di non lasciarne nessuno in sospeso; dopo che hai allocato la memoria, quando non serve più, liberala!
- Sicuramente apprezzato: nei passaggi oscuri, commenta un minimo il codice per semplificarne la lettura, potrebbe aiutare il professore a capire cosa avevi in mente.

Un consiglio: se non sei molto pratico, durante l'esame ogni volta che scrivi un pezzo di codice compilalo e provalo. Non cercare di scrivere tutto e solo alla fine provare a compilare: è praticamente impossibile scrivere tanto codice senza fare neanche un piccolo errore. Trovare un errore in un pezzetto di codice è più facile che capire perché un intero programma non funziona.

Procedi per piccoli passi: scrivi una funzione (tipo carica da file) e testala, se è corretta salva tutto e vai avanti. Se ad un certo punto si romperà tutto –in stile segmentation fault o cose strane– saprai che al passo precedente funzionava tutto e ti basterà tornare indietro o aggiustare il nuovo pezzo di codice.

Non c'è nulla di peggio di scrivere l'intero programma, testarlo per la prima volta dieci minuti prima della consegna e ritrovarsi come output un segmentation violation.

E, soprattutto, meglio un programma non completo (ma che compila) di uno finito ma che non compila (rischio bocciatura!)!

D.1 21 Luglio 2015

Il testo è il seguente:

INFORMATICA – 21 luglio 2015

Cognome e nome _____

Matricola _____ Firma _____

Il file `poligoni.dat`, che si trova in `/home/comune/20150721_Dati`, contiene la descrizione di un numero imprecisato di poligoni in R2, ogni poligono e' descritto attraverso le coppie (x,y) delle coordinate dei suoi vertici, considerati ordinatamente, uno dopo l'altro, a partire da uno qualunque. Ogni gruppo di coordinate dei vertici di un poligono e' preceduto da un intero che specifica il numero dei vertici del poligono considerato. La fine delle descrizioni di poligoni e' individuata da uno zero (0 significa che il prossimo poligono ha 0 vertici, quindi il file e' finito).

1) Per ognuno dei poligoni descritti sul file si vuole sapere se e' intrecciato o no, per fare questo si suggerisce di costruire una funzione che dati due segmenti individui l'eventuale intersezione delle rette su cui giacciono e verifichi se tale intersezione e' all'interno di entrambi o no.

2) Considerando il complesso di tutti i vertici dei poligoni presenti sul file individuare l'ascissa minima e quella massima e costruire un istogramma che evidenzi quanti punti cadono nelle strisce verticali ottenute dividendo l'intervallo [xmin-.5,xmax+.5] in 10 parti.

Visualizzare a video il risultato di ciascun punto. Tutti i risultati in formato alfanumerico dovranno essere salvati in un file `risultati.dat`.

Inserire tutti i file necessari alla compilazione del programma in una cartella dal nome `Cognome_Matricola`. La cartella dovrà contenere anche un `makefile` che consenta di compilare ed eseguire il programma usando, rispettivamente, `make compila` e `make esegui`.

La cartella dovrà essere copiata in `/home/comune/20150721_ProvaLab`

La valutazione terra` conto della strutturazione del codice (uso funzioni, compilazione separata ecc...) e della qualità dei risultati.

Il nostro codice è diviso in: `makefile`, `main.cpp`, `funzioni.h`, `funzioni.cpp`; ti riporto anche `poligoni.dat` se vuoi provare sui dati dell'esame.

makefile

```
#come e' costituito ogni comando di un makefile?
# nomecomando: oggetti necessari al comando
#       comando (preceduto da spazio di tab, obbligatorio!)


 #questa riga e la prossima sono dei comandi di root per dire al compilatore
      che flag usare in compilazione
INCS=`root-config --cflags`
LIBS=`root-config --libs`

#esempio: compila ha bisogno dei file oggetto "main.o" e "funzioni.o"
compila: main.o funzioni.o
        g++ main.o funzioni.o -o main ${INCS} ${LIBS} #quando faccio il
            linking, ovvero creo l'eseguibile finale, devono esserci entrambi
             i comandi di root, vengono chiamati con dollaro e dentro le
            parentesi graffe il nome delle variabili (quello che precede l'
            uguale alle righe sopra)

#main.o ha bisogno dei file di codice "main.cpp" e "funzioni.h" (che e'
    incluso in main.cpp)
main.o: main.cpp funzioni.h
        g++ main.cpp -c ${INCS} #nella compilazione parziale dei file che
            includono le librerie di root ci vuole il comando flags (che noi
            abbiamo assegnato ad INCS)
funzioni.o: funzioni.cpp funzioni.h
        g++ funzioni.cpp -c
esegui: main
        ./main
```

Esempio D.1

main.cpp

```
#include "funzioni.h"   //importo la libreria, essendo un file locale uso le
     virgolette
#include <iostream>     //cout, endl, ecc...
#include <fstream>      //scrittura su file

//Librerie di ROOT
#include "TH1F.h"       //histogramma monodimensionale
#include "TApplication.h" //per far partire la "app" di root
#include "TCanvas.h"    //per poter usare le canvas, le tele su cui
    disegnare


using namespace std; //per evitare di dover anteporre "std::" a tutte le
    funzioni facenti parte del namespace std; e' un argomento probabilmente
```

```
    non toccato nel corso di calcolo 1, non dimenticarti questa linea pero'
    se non sai come fare altrimenti!

int main(){
        Poligono* poligoni; //vettore dinamico di poligoni
        unsigned int npoligoni=0; //numero di poligoni che carichero'
        ofstream out; //dichiaro un canale di uscita per un file su cui
            scrivere tutti i risultati
        out.open("risultati.dat");
        if(out.fail()){ //controllare che lo stream non sia rotto e' indice
            di buono stile di programmazione, magari non ti sembrera'
            fondamentale ma fidati: all'esame sara' apprezzato!
                cout << "Errore apertura file ''risultati.dat'', interrompo
                    il programma." << endl;
                return 1;
        }


        CaricaDaFile("poligoni.dat", poligoni, npoligoni); //carico i dati
            da file, vedi funzioni.h e .cpp


        cout << "Numero di poligoni " << npoligoni << endl;
        out << "Numero di poligoni " << npoligoni << endl;
        for(unsigned int i=0; i<npoligoni; ++i){ //ciclo tutti i poligono
                if(Intrecciato(poligoni[i])){ //e controllo per ogni
                    poligono se e' intrecciato
                        cout << "Il poligono n'"<< i+1<< " e' intrecciato."
                            << endl;
                        out << "Il poligono n'"<< i+1<< " e' intrecciato."<<
                             endl;
                }
        }

        //Per la seconda parte del programma prima di tutto creo un vettore
            contenente tutti i punti (dovendo ora lavorare solo su quelli),
            quindi  ordino per x crescenti per semplificarmi la vita
        unsigned int npunti=0;
        PuntoR2* punti;
        GetPunti(poligoni,npoligoni, punti, npunti);


        //la memoria allocata dal vettore di poligoni non mi serve piu'
            nella seconda parte del programma, la libero!
        for(unsigned int i=0; i<npoligoni; ++i)
                delete[] poligoni[i].vertici;
        delete[] poligoni;


        SelsortCrescente(punti, npunti);
        cout << "Ascissa minore: x = " << punti[0].x<< endl; //avendo
            ordinato per x crescenti e' il primo del vettore
        out << "Ascissa minore: x = " << punti[0].x << endl;
        cout << "Ascissa massima: x = " << punti[npunti-1].x<< endl; //e' l'
            ultimo del vettore
        out << "Ascissa massima: x = " << punti[npunti-1].x<< endl;
```

```
        out.close(); //di nuovo: chiudere i canali per file e' indice di
            buono stile di programmazione, non dimenticarlo!



        TApplication app("", 0, 0); //dichiara l'applicazione di root
        TCanvas tela("Istogramma", "Strisce verticali"); //dichiara una tela
        TH1F histo("", "", 10, punti[0].x-0.5, punti[npunti-1].x+0.5); //
            dichiara un istogramma contenente 10 bin, punto minimo "punti[0].
            x-0.5", punto massimo "punti[npunti-1].x+0.5"
        for(unsigned int i=0; i<npunti; ++i)
                histo.Fill(punti[i].x); //riempio l'istrogramma con tutti i
                    punti

        delete[] punti;  //non mi serve piu', libero la memoria


        tela.cd();       //seleziono la tela su cui disegnare
        histo.Draw();    //disegno l'istogramma
        app.Run();       //faccio partire root

        return 0;
}
```

Esempio D.2

funzioni.h

```
#ifndef FUNZIONI_H
#define FUNZIONI_H
#include <fstream>      //lettura e scrittura su file
#include <iostream>     //cout, endl...
#include <cstdlib>      //exit()


using namespace std;


struct PuntoR2{ //definisco una struttura per il tipo di dato utile per
    rappresentare un punto in r2
        float x;
        float y;
};

struct Poligono{ //struttura per rappresentare un poligono
        PuntoR2* vertici; //puntatore a PuntoR2 per creare dinamicamente un
            vettore di punti
        unsigned int nvertici; //per tenere a memoria il numero di vertici
};

void CaricaDaFile(const char* nomefile, Poligono* &poligoni, unsigned int&
    npoligoni); //funzione per caricare i dati da file: come argomento vuole
    il nome del file, un puntatore a struct Poligono passato per referenza
```

```
    per crare dinamicamente un vettore di poligoni, e un intero in cui
    salvare il numero di poligoni caricati

bool Intrecciato(Poligono poligono); // funzione che trova se un poligono e'
     intrecciato, se si restituisce true, altrimenti false.

void GetPunti(Poligono* poligoni,unsigned int npoligoni, PuntoR2* &punti,
    unsigned int &npunti); //funzione per creare un vettore di punti
    contenente tutti i punti dei poligoni, in ingresso il vettore in poligoni
    , la sua dimensione e, passati per referenza, un puntatore a PuntoR2 con
    cui creare dinamicamente il vettore e la sua futura dimensione



//Probabilmente hai gia' scritto nel corso un algoritmo di ordinamento, per
    esempio il selsort, devo pero' adattarlo (se non ho usato template) al
    tipo di dato su cui vado ad operare (PuntoR2 nel nostro caso) -ad esempio
    , cosa vuol dire ">" per il nostro tipo di dato?...- , per come l'ho
    implementato io ordina per x crescenti!
void SelsortCrescente(PuntoR2 v[], unsigned int dim); //in ingresso un
    vettore e la sua dimensione


#endif /* FUNZIONI_H */
```

Esempio D.3

funzioni.cpp

```
#include "funzioni.h" //devo includere l'header delle funzioni, il file ".h"
     della libreria, con le definizioni di funzioni

void CaricaDaFile(const char* nomefile, Poligono* &poligoni, unsigned int&
    npoligoni){
        ifstream in; //canale per lettura su file

        in.open(nomefile);
        if(in.fail()){
                cout << "Errore apertura file ''"<< nomefile << "''. Chiudo
                    il programma." << endl;
                exit(1);
        }



        //ho aperto il file, ora mi occupo di caricare i dati, ma prima devo
             contarli!

        unsigned int nvertici=0; //per contare i vertici in lettura
        PuntoR2 appo; //variabile di appoggio per contare i dati
        npoligoni=0; //inizializzo a zero il numero di poligoni

        in >> nvertici;
```

```
        while(nvertici!=0 and !in.eof()){ //per leggere nvertici non deve
            essere zero e non devo essere alla fine del file
                for(unsigned int i=0; i<nvertici; ++i){
                        in >> appo.x;
                        in >> appo.y;
                }
                npoligoni++; //ho letto quanti vertici ha un poligono
                in >>nvertici; //leggo quanti vertici ha il successivo, se e
                    ' diverso da zero il ciclo continua.
        }


        //ho contato il numero di poligoni, posso creare il vettore di
            poligoni:
        poligoni=new Poligono[npoligoni];

        //ora posso caricare i dati, porto la testina del file al punto
            iniziale
        in.clear();
        in.seekg(0);

        for(unsigned int i=0; i<npoligoni; ++i){
                in >> nvertici;
                poligoni[i].vertici=new PuntoR2[nvertici]; //alloco il
                    vettore di vertici di ogni poligono, dovro' ricordarmi ad
                     un certo punto di fare un bel delete...
                poligoni[i].nvertici=nvertici;
                for(unsigned int j=0; j<nvertici; ++j){
                        in >> (poligoni[i]).vertici[j].x;
                        in >> (poligoni[i]).vertici[j].y;
                }
        }
        in.close();//ho finito di caricare i dati, devo ricordarmi di
            chiudere il file!
}


bool Intrecciato(Poligono poligono){

        float a1, a2, c1, c2;//per il calcolo delle rette
        float x_intersezione;
        for(unsigned int i=0; i<poligono.nvertici-1; ++i){
                a1=(poligono.vertici[i].y-poligono.vertici[i+1].y)/(poligono
                    .vertici[i].x-poligono.vertici[i+1].x); //coef. angolare
                    prima retta
                c1=poligono.vertici[i].y-poligono.vertici[i].x*a1; //
                    costante prima retta
                for(unsigned int j=i+2; j<poligono.nvertici-1; ++j){ //j=i
                    +2: non considero il lato immediatamente successivo
                        a2=(poligono.vertici[j].y-poligono.vertici[j+1].y)/(
                            poligono.vertici[j].x-poligono.vertici[j+1].x);
                            //coef.angolare seconda retta
                        c2=poligono.vertici[j].y-poligono.vertici[j].x*a2;
                            //costante seconda retta
                        if(a1==a2)
```

```
                                continue; //le rette sono parallele, di
                                    sicuro nessuna intersezione
                        x_intersezione=(c2-c1)/(a1-a2); //interseco le due
                            rette e trovo l'ascissa di intersezione
                        if((x_intersezione>poligono.vertici[i].x and
                            x_intersezione<poligono.vertici[j].x) or (
                            x_intersezione<poligono.vertici[i].x and
                            x_intersezione>poligono.vertici[j].x)) //non so
                            quale tra le due x e' la maggiore e minore,
                            quindi testo entrambe le possibilia'
                                return true;
                }
        }
        return false; //non ho trovato nessuna intersezione
}


void GetPunti(Poligono* poligoni, unsigned int npoligoni, PuntoR2* &punti,
    unsigned int &npunti){
        npunti=0;
        for(unsigned int i=0; i<npoligoni; ++i)
                npunti+=poligoni[i].nvertici; //conto i punti totali

        punti= new PuntoR2[npunti]; //di nuovo: quando non mi servira' piu'
            andra' fatto un bel delete!
        unsigned int counter=0; //contatore
        for(unsigned int i=0; i<npoligoni; ++i){
                for(unsigned j=0; j<poligoni[i].nvertici; ++j){
                        punti[counter]=poligoni[i].vertici[j];
                        counter++;
                }

        }

}


void SelsortCrescente(PuntoR2 v[], unsigned int dim){
        unsigned int posMin;
        PuntoR2 min;
        PuntoR2 appo;
        for (unsigned int j=0;  j<dim-1; ++j) {
                min = v[j];
                posMin=j;
                for (unsigned int i=j+1; i<dim; ++i){
                        if (v[i].x < min.x) {
                                posMin=i;
                                min=v[i];
                        }
                }
                appo=v[j];
                v[j]=v[posMin];
                v[posMin]=appo;
        }
}
```

Esempio D.4

poligoni.dat

```
5
0               5
3               7.2
7               0
1               2.4
2               8
4
0               3
5               6
6               3
3               2
4
3.2             1
7.1             1
8.4             5
5               7.1
3
1               1
3.3             5
4.2             2
4
2               2
5               2.1
5               5.1
2               5
4
3               3
6               6
6               2
3               4
4
0               3.2
5               6
6.2             3
3.2             0
4
3.3             1
7               1
8               5.1
5.1             7
3
1               1
3               5
4               2
5
0.2             5
3.1             7
7               0
1               2
2.4             8
4
0               3.1
5.2             6.3
6               3
```

```
3                   1
4
3.2                 1
7.1                 1.2
8                   5
5                   7
0
```

Esempio D.5