Metodi informatici della Fisica

C++12: appendice++

Susanna.Costanza@unipv.it
Andrea.Negri@unipv.it

Direttive di preprocessore

- Sono indicate dal prefisso #
  - Sono eseguite prima della compilazione

Direttive di preprocessore

- #include <file>
  - il precompilatore includerà nel file corrente il contenuto del file che segue la direttiva

```
#include <iostream>    // Inclusione file di sistema
#include "MioHeader.h" // Inclusione file utente
```

- #define etichetta valore
  - il precompilatore sostituirà all'interno del codice tutte le occorrenze di "etichetta" con il contenuto di "valore", es:

```
#define ALPHA 137.035 // Definiz. etichetta ALPHA
```

  - e' anche possibile creare macro,
    cioè dei #define che accettano parametri

Direttive di preprocessore

- #ifdef etichetta pezzoDiCodice #endif
  - #ifdef (#ifndef) e #endif permettono di includere (escludere) dalla compilazione blocchi di codice in base all'esistenza o meno di una certa etichetta
- L'istruzione cout seguente

```
#ifdef DEBUG
std::cout << "Variabile x = " << x << std::endl;
#endif
```

  è effettivamente compilata solo se
  - o l'etichetta DEBUG è stata definita con #define in una parte a monte del codice
  - o se si aggiunge l'etichetta DEBUG ai parametri di compilazione tramite il parametro **-D**

```
esempio> g++ -DDEBUG myfile.cxx
```

Preprocessore ed header file

A.h

```
int A();
...
```

B.h

```
#include "A.h"

float B();
...
```

source.cxx

```
#include "A.h"
#include "B.h"

int main()
{
  ...
}
```

```
prompt> g++ source.cxx
```

- In questo esempio la dichiarazione della funzione A() è inclusa **2 volte** e verrebbe quindi processata 2 volte dal compilatore
  - La 1a perché A.h è incluso direttamente da source.cxx
  - La 2a volta perché A.h è incluso da B.h (che a sua volta è stato incluso da source.cxx)
- Se ciò accade il compilatore da **errore** a causa di una **ri-dichiarazione** della funzione A()
  - Il problema può essere prevenuto con #ifndef e #define

Direttive di preprocessore

- Con **`#ifndef`**, **`#endif`** e **`#define`** è possibile evitare che le dichiarazioni in un header file vengano processate più volte dal compilatore

```
#ifndef etichetta
#define etichetta
pezzoDiCodice
#endif
```

  - Es: g++ vede le “linee di codice” qui sotto **una sola** volta

```
#ifndef  CICCIO
#define  CICCIO
linee di codice // Il compilatore vedrà queste linee
linee di codice // una sola volta: al secondo eventuale
linee di codice // passaggio l'etichetta CICCIO sarà già
linee di codice // definita e il blocco ignorato
#endif
```

Opzioni di g++

Opzioni di g++

- **`-o`** nome
  - Definisce il nome dell'eseguibile che verrà creato dal linker
- **`-E`**
  - Arresta g++ dopo la fase di pre-compilazione.
  - Si utilizza solo per esigenze didattiche o di debugging
- **`-S`**
  - Arresta g++ dopo la fase di compilazione propriamente detta, cioè prima di chiamare l'assembler
  - l'output sono file in linguaggio assembly
- **`-c`**
  - Arresta g++ prima della fase di linking, cioè dopo aver chiamato l'assembler
  - l'output sono file oggetto (.o), cioè file in formato binario ma non ancora eseguibili (manca il linking appunto)

Opzioni di g++

- **`-I` path**
  - Specifica la path di ricerca degli header file che descrivono librerie non di sistema. Esempio nel caso di root
    `-I/usr/include/root`
- **`-L` path**
  - Specifica la path di ricerca delle librerie non di sistema. Serve al linker per trovare le librerie necessarie. Per esempio nel caso di root
    `-L/usr/lib/root`
- **`-l`libreria**
  - Specifica la libreria da linkare. Es: per linkare la libreria `libHist.so` che si trova in `/usr/lib/root/` occorre aggiungere
    `-L/usr/lib/root -lHist`
- **`-shared -fPIC`**
  - Serve per creare una libreria (shared library). Come libm.so. Esempio
    g++ -o libmia.so `-shared -fPIC mia.cxx`

Opzioni di g++: -std

- Il C++ è un linguaggio in evoluzione
  - C++98
    - → C++11
      - → C++14
        - → C++17
          - → C++20
  - E man mano i compilatori si adeguano

- Per utilizzare le nuove funzionalità del lunguaggio occorre informare il compilatore utilizzando il parametro **-std=c++??**
  - Esempio per utilizzare lo standard g++14

```
shell> g++ -std=c++14 file.cxx
```

Auto

- **Auto** è una direttiva disponibile da C++11
- Con la direttiva **auto** si chiede al compilatore di dedurre automaticamente il tipo in base a ciò che c'è a destra dell'= di assegnazione
  - Per es. qui, il compilatore deduce che y è float

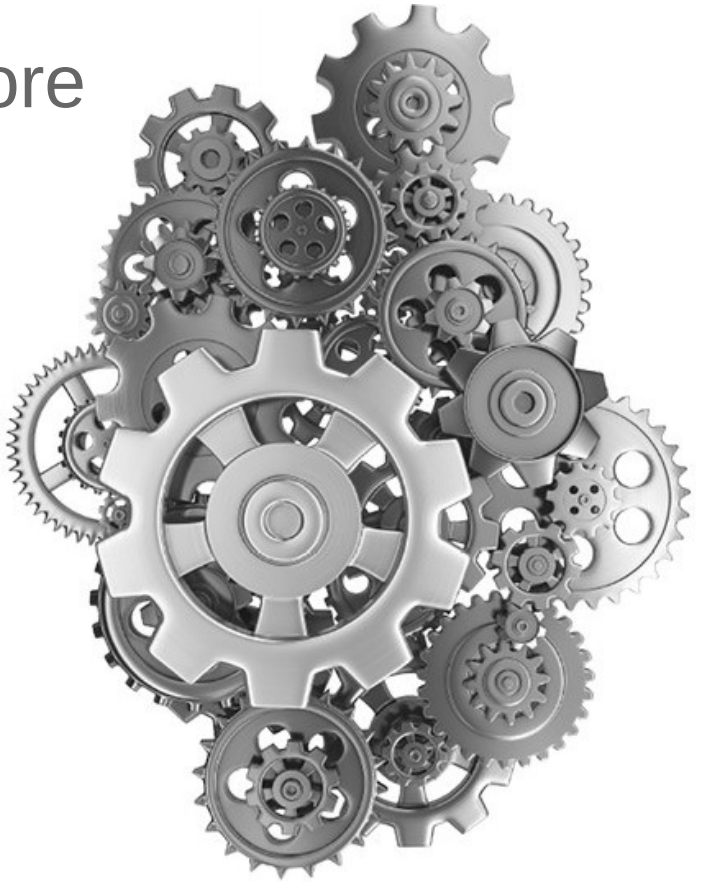
```
std::vector<float> v(10,0);
float x = v[1]; // x con tipo esplicito
auto y  = v[2]; // y con tipo dedotto
```

- E permette di semplificare i loop sui container
  - Qui per "loop-are" sugli elementi di v basta

```
for(auto i: v) // Loop sugli elementi i di v
  std::cout << i << std::endl;
```

Make

- **make** è un'utility che automatizza la conversione dei file da una forma ad un'altra, risolvendo le dipendenze e invocando programmi esterni per fare il lavoro
  - esempio g++ o un altro compilatore
- Usato principalmente per gestire la compilazione di progetti con più di un file
  - Permette di ricompilare i file oggetto interessati da un cambiamento nei rispettivi file sorgente, ma non gli altri file



Makefile

- Se vi capiterà di imbattervi in un progetto software da compilare troverete nell'archivio (solitamente un tar file)
  - i file header e sorgente
  - un file chiamato **Makefile**
- Per compilare il progetto sarà sufficiente eseguire make

```
[fisica@unipv] make
```

  - Chi ha scritto il **Makefile** ha già specificato come chiamare g++ per creare le librerie e gli eseguibili del progetto

Esempio di Makefile

- Per progetto contenente 3 file sorgente e 2 header
  - NB: lo spazio ad inizio riga è una tabulazione **<Tab>**

```
[fisica@unipv] cat Makefile
# Riga di commento.
CC=g++
CFLAGS=-c -Wall

hello: main.o factorial.o hello.o
    $(CC) main.o factorial.o hello.o -o hello

main.o: main.cpp
    $(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
    $(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
    $(CC) $(CFLAGS) hello.cpp

clean:
    rm -rf *o hello
```

Questi non sono spazi ma un <tab>

Assegnazione di variabili

L'eseguibile hello dipende dai file elencati

Comando per creare hello (con linking)

main.o dipende da main.cpp

Comando per creare main.o (compilazione) nel caso main.cpp sia stato modificato

Eseguendo “make clean” viene cancellato l'eseguibile hello e tutti i file oggetto .o

Void e NULL

Void e NULL

- Il qualificatore **void** viene utilizzato per indicare l'assenza di un parametro o di un tipo ritornato
  - Se utilizzato come tipo ritornato significa che la funzione non ritorna alcun valore

```
void funzione();
```

  - Se utilizzato nei parametri di una funzione significa che la funzione non ha parametri

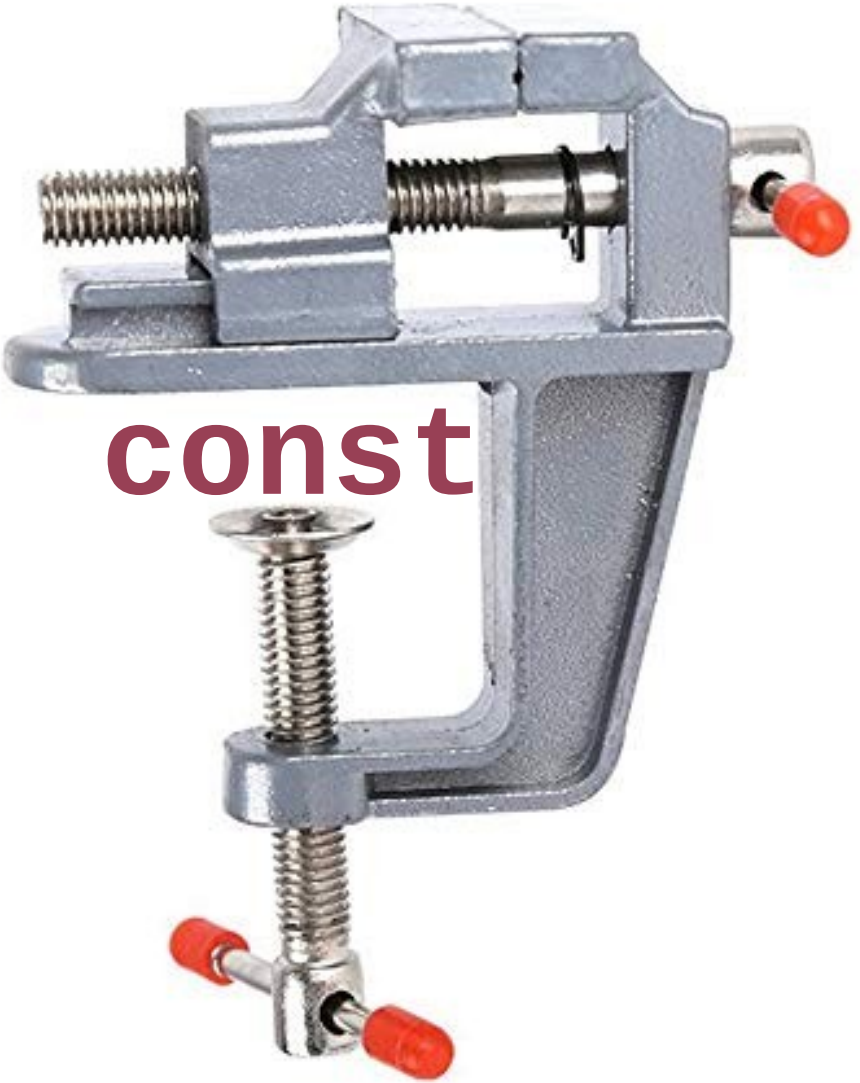
```
int funzione(void);
```

  - Se utilizzato come tipo di un puntatore, significa che il puntatore è senza tipo definito (è universale)

```
void *p;
```

- **NULL** indica invece un puntatore nullo
  - Cioè che punta all'indirizzo zero

Qualificatore const



Variabili const

- Quando usato nella dichiarazione di una variabile il qualificatore **const** indica che la variabile non può essere modificata
  - Cioè la variabile è read-only
  - Ogni tentativo di modificarla conduce ad un errore di compilazione

```
const int i = 7;
i = 2;   // error: assignement
         // of read-only variable
```

- Quindi, se è presente **const**, l'assegnazione va fatta contestualmente alla dichiarazione
  - Ogni modifica successiva da errore

const e puntatori

- **const** sui puntatori ha due possibili significati
- Indicare che il puntatore punta a una variabile const
  - Obbligatorio per potere puntare ad un oggetto costante
  - Sintassi: `const tipo* nome;`

```
const int  x = 5;
const int* p = &x; // Puntatore a 'const int'
*p = 88;           // Errore tentativo di modifica
                   // di un oggetto const int
```

- Indicare che è il puntatore stesso ad essere costante
  - Cioè si vuole che il puntatore non possa essere spostato
  - Sintassi: `tipo* const nome;`

```
int x = 5;
int* const p = &x; // Puntatore costante a 'int'
*p = 88;  // Ok, modifica di un oggetto non const
p++;      // Errore, incremento di un puntatore const
```

Metodi const

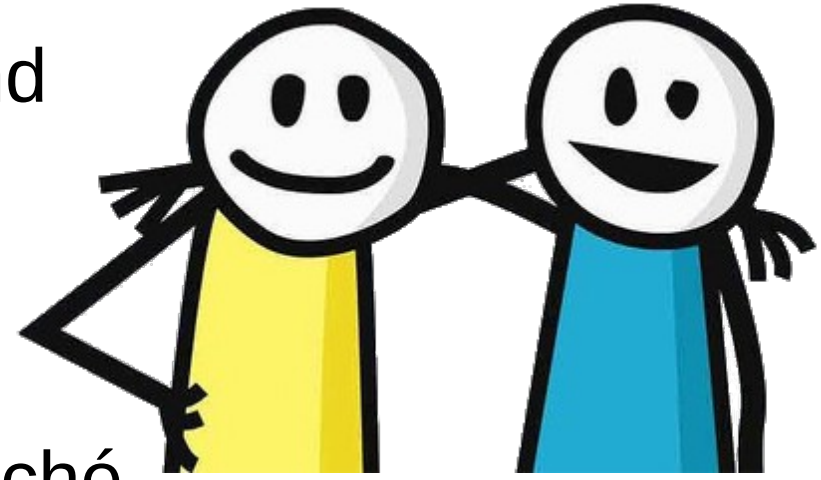
- Un oggetto **const** può chiamare solo metodi definiti **const**
  - Un metodo è const se c'è il qualificatore const dopo ()

```
class Dummy                 // Classe minimale
{
 public:
  Dummy() { m_a = 1;}       // Costruttore
  void f()       {;}        // Metodo che non fa niente
  void g() const {;}        // Metodo const che non fa niente
 private:
  int m_a;
};
int main()
{
  Dummy       a;  // Oggetto di tipo Dummy
  Dummy const c;  // Oggetto costante di tipo Dummy
  a.f();  // OK
  a.g();  // OK
  c.f();  // ERRORE: 'passing ..... discards qualifiers'
          // Un oggetto const puo' chiamare solo metodi definiti const
  c.g();  // OK, il metodo g() e' dichiarato const, puo' agire su c
  return 0;
}
```

Friend

Friend

- La direttiva **friend** è usata per fornire **accesso** “esterno” ai dati membro **privati** di una classe
  - Funziona come l'amicizia sui social network
- Una classe può dichiarare friend
  - una funzione esterna
  - un metodo di un'altra classe
  - una classe intera



- Da usare con moderazione perché contrario al paradigma dell'incapsulamento
  - E' necessario solo in casi particolari
  - Per es nell'overloading di operator<<() che è una funzione esterna alle classi, ma che potrebbe aver necessità di accedere ai dati membro private di esse

Friend

```
class Cosa // Classe minimale
{
 public:
  Cosa(int x) { m_data = x;} // Costruttore

  // La classe dichiara che funz() e' sua friend
  friend void funz(Cosa c);
 private:
  int m_data;     // Dato membro privato
};

// Funzione esterna alla classe Cosa, ma che quest'ultima
// ha riconosciuto come friend
void funz(Cosa c)
{  // Amica di Cosa, quindi può accede ai dati privati
   std::cout << "Dato privato: " << c.m_data << std::endl;
}

int main()
{
  Cosa k(7777); // Oggetto della classe Cosa
  funz(k);      // Chiamata della funzione funz()
  return 0;
}
```

friend.cxx

formattazione

iomanip

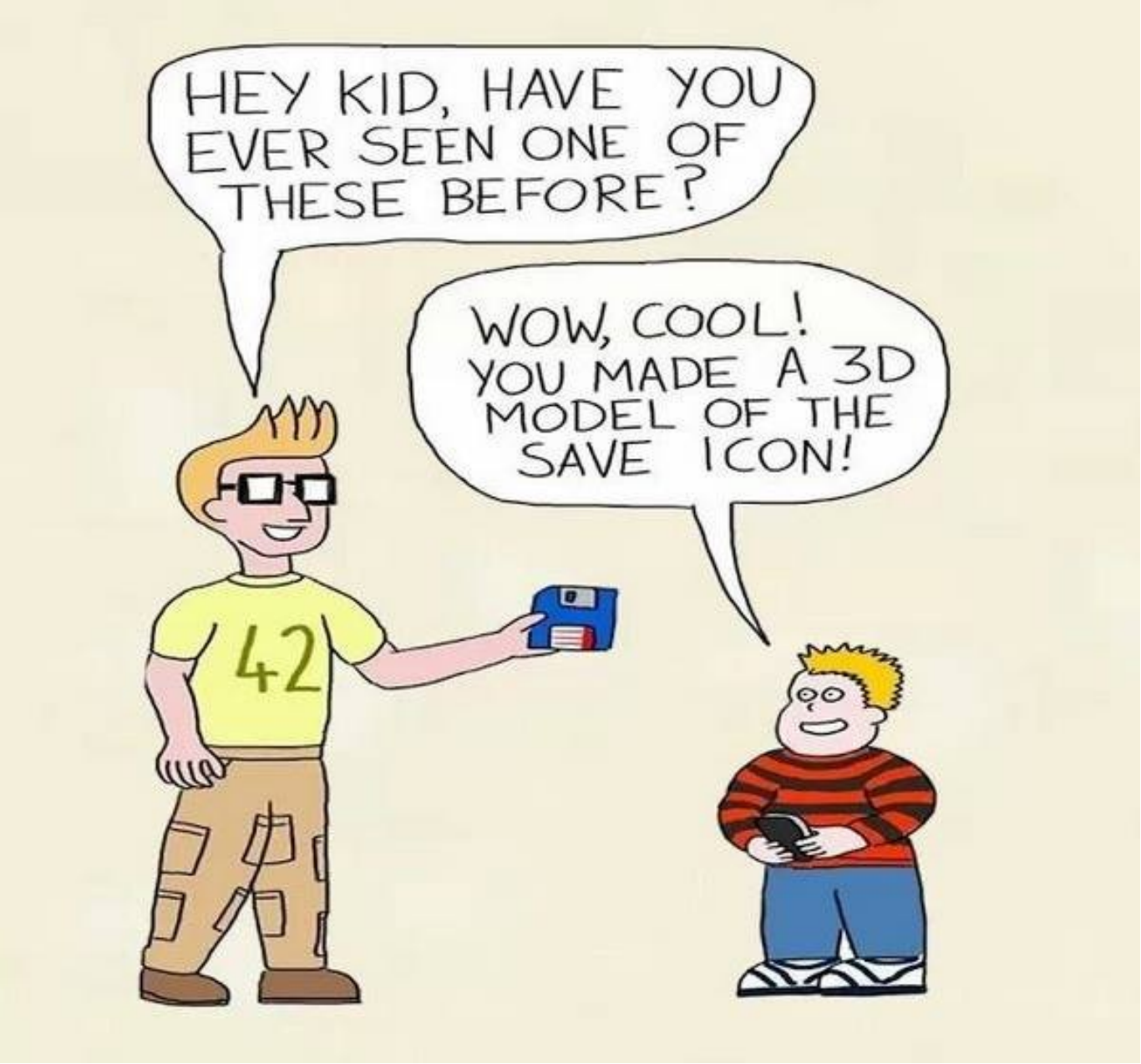
- Controllo **formattazione** dell'output possibile attraverso manipolatori definiti in <iostream> o <iomanip>

```
#include <iostream>
#include <iomanip>

int main ()
{
  double a = 0.123456789;
  double b = 7.;
  std::cout << a << " " << b << std::endl;

  std::cout.precision(4); // Definisce la precisione
  std::cout << a << " " << b << std::endl;

  // std::setw() definisce la larghezza di campo
  std::cout << std::setw(10) << a << " "
            << std::setw(10) << b << std::endl;
  std::cout << std::fixed      << a << " " << b << std::endl;
  std::cout << std::scientific << a << " " << b << std::endl;

  return 0;
}
```

```
[negri@marvin]$./a.out
0.123457 7
0.1235 7
    0.1235          7
0.1235 7.0000
1.2346e-01 7.0000e+00
```

iomanip.cxx

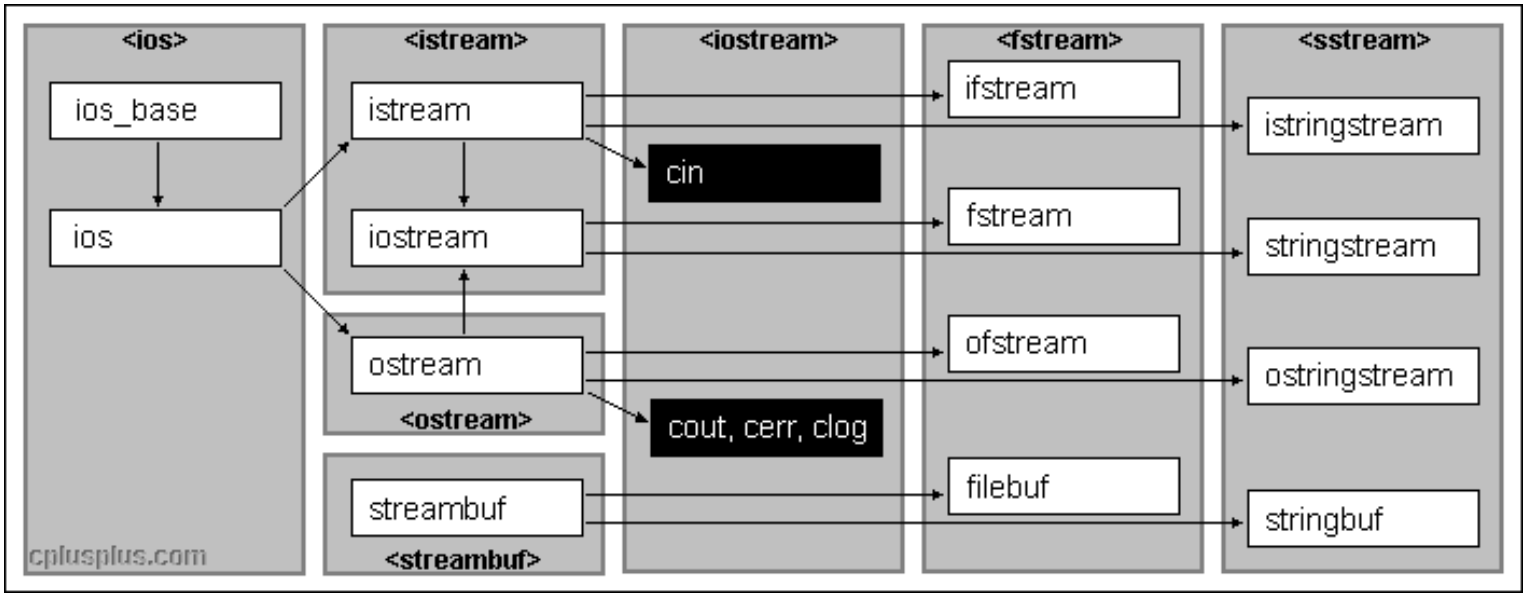
I/O da file


HEY KID, HAVE YOU EVER SEEN ONE OF THESE BEFORE?
WOW, COOL! YOU MADE A 3D MODEL OF THE SAVE ICON!
42

C++: I/O da file

- I/O da file gestito da classi **ifstream** e **ofstream**
  - Dichiarate nell'header file **`fstream`**
- Qui solo introduzione alle funzionalità base per lettura e scrittura di soli file di testo
  - Vedi www.cplusplus.com, links:ifstream, ofstream

Tabella ereditarietà classi C++ per I/O



Scrittura su file di testo

- Simile all'uso di std::cout

```
#include <fstream>   // Include file

int main()
{
  // Creazione oggetto ofstream out associato al
  // file "file.txt" trattato come file di testo
  std::ofstream out("file.txt");

  int i = 42;
  float f = 2.1425;

  // Stessa sintassi di cout
  out << "Prova " << i << "\t" << f << std::endl;

  out.close(); // Chiusura del file

  return 0;
}
```

writeToTextFile.cxx

Lettura da file di testo

- Utilizzo simile a std::cin
  - Es: assumiamo file con numeri su due colonne

```
#include <fstream>
#include <iostream>
int main()
{
  // Creazione oggetto classe ifstream con nome file
  std::ifstream in("fileConNumeriSu2Colonne.txt");

  float x, y;       // Variabili di lettura
  // Leggi fino a che non si arriva ad End Of File
  while(!in.eof()) // eof() da vero se il file è finito
  {
    in >> x >> y;  // lettura di due numeri su x e y
    std::cout << x << "\t" << y << std::endl;
  }
  in.close() // Chiusura del file
  return 0;
}
```

readFromTextFile.cxx

stringstream

- Esiste anche una classe che permette di scrivere su una **stringa** come se fosse uno **stream** di I/O
  - stringstream definita in <sstream>
  - Permette di concatenare ad una stringa tipi differenti, per esempio un intero

```
#include <iostream>
#include <sstream>
int main ()
{
  int i = 42;
  std::ostringstream o; // Costruttore di ostringstream

  o << "Stringa" << i;  // Appende testo ed intero alla stream
  o << " " << 137.03;   // Appende un float alla stream

  std::string s = o.str(); // Salva contenuto stream in una string
  std::cout << s << std::endl;

  return 0;
}
```

StringStream.cxx

Scelte di default

- Valori di **default** per gli **argomenti** di funzioni/metodi
  - valori che assumeranno se non esplicitati nella chiamata

Argomenti di default

- Un valore è di default se nella dichiarazione della funzione la variabile è seguita da un'assegnazione
  - Es: f() con 3 argomenti, gli ultimi 2 con valore di default

```
void f(int a, char c = 'a', float f = 7.42);
```

  - Tali valori sono utilizzati nel caso la funzione sia chiamata senza gli argomenti corrispondenti

```
f(3, 'x', 8.15); // a = 3, c='x', f = 8.15
f(4, 'y');       // a = 4, c='y', f = 7.42 (default)
f(9);            // a = 9, c='a' (def), f=7.42 (def)
```

- NB: nella dichiarazione della funzione gli argomenti di default devono seguire tutti gli altri argomenti
  - Altrimenti il compilatore non può identificare il prototipo corretto da chiamare

Metodi di default

- Alcuni metodi sono generati **implicitamente** dal compilatore, nel caso non siano dichiarati esplicitamente nella classe
  - Esempio: costruttori senza parametri, costruttore di copie, distruttore
- Costruttore **senza parametri**, necessario per
  - la creazione di array
  - per l'allocazione dinamica
  - se non esistono altri costruttori
  - se un oggetto della classe in questione è membro di un'altra e il costruttore di quest'ultima non contiene il membro nella lista di inizializzazione

Metodi di default

- Costruttore di **copie**
  - E' il metodo chiamato quando si utilizza l'operatore di assegnazione. Es:

```
X(const &X); // Costruttore di copie della classe X
```

  - Il costruttore di copie di default in genere fa ciò che ci si attende, a meno che non ci sia di mezzo l'allocazione dinamica
- **Distruttore**
  - E' chiamato ogni qual volta un oggetto esce dal suo scopo (corpo di una funzione, di un loop, di un if, ecc)
  - Il distruttore va esplicitato nel caso sia necessario deallocare memoria allocata dinamicamente nella classe

Casting

Casting

- Conversione del tipo di una variabile
  - **Implicito**: es. se si divide un int per un float, il compilatore promuove implicitamente l'int a float
  - **Esplicito**: cioè forzato dal programmatore
- Casting C-style
  - Già utilizzato per promuovere un int a float (per avere risultati "reali" nelle divisioni tra interi)

```
int i   = 7;
float f = (float) i; // Sintassi C-style
```

- La sintassi C++ è più "esplicita"

```
float f = static_cast<float>(i); // C++style
```

Casting

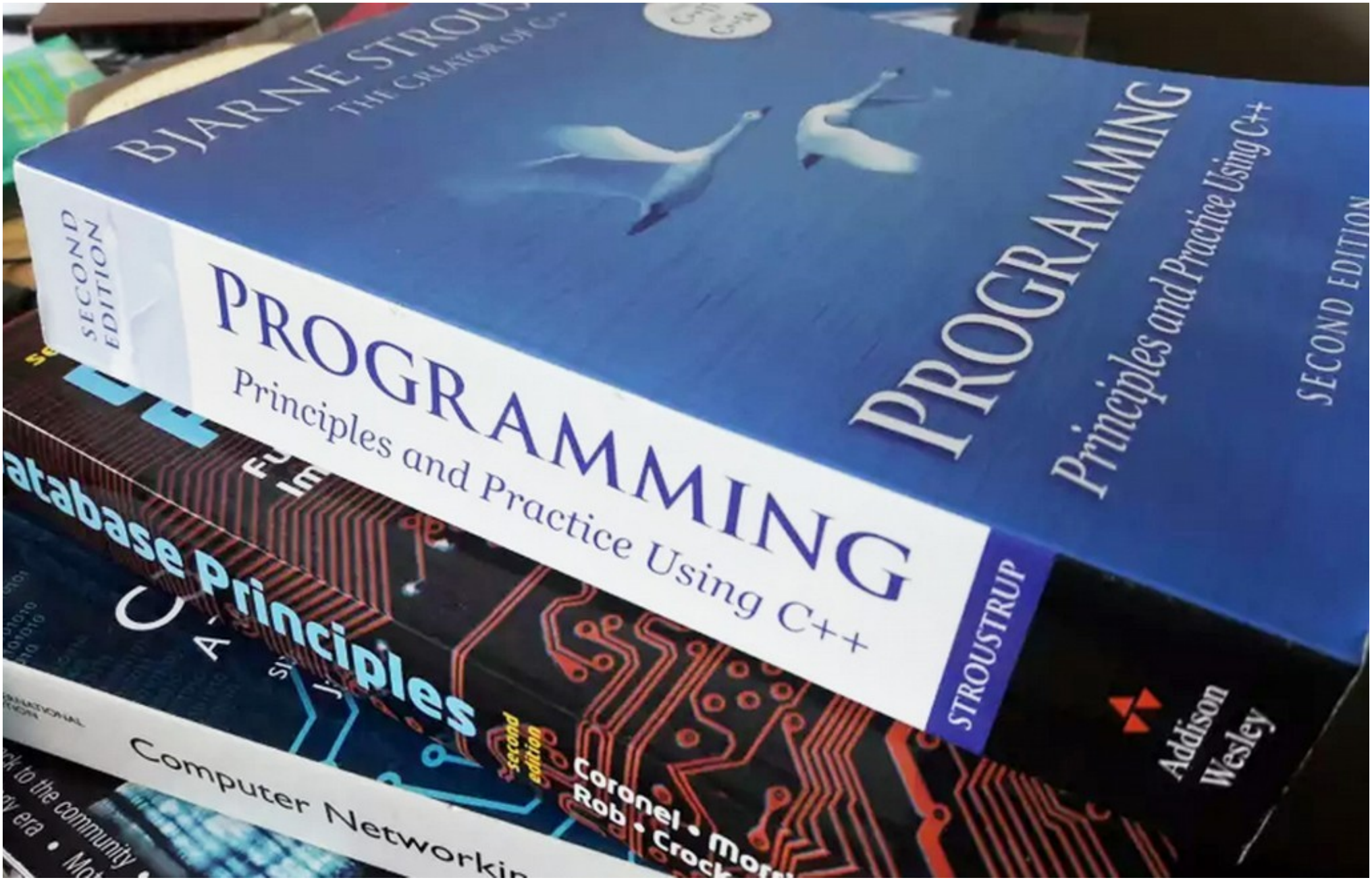
- Con i **puntatori** è possibile forzare ∀ tipo di **cast**
  - Qualsiasi puntatore è semplicemente una singola casella di memoria che contiene un indirizzo
  - Il tipo è quello dell'oggetto puntato
- A volte può servire **reinterpretare** il puntatore
  - Ma è sempre un'operazione **pericolosa**: uno deve sapere cosa sta facendo
  - Esempio (NB: non particolarmente utile)

```
string *q = new string("Stringa a caso");
int    *p = reinterpret_cast<int>(q);
```

- Rimozione del const

```
const_cast<tipo>(puntatore const a tipo);
```

What else?


BJARNE STROUSTRUP
THE CREATOR OF C++
PROGRAMMING
Principles and Practice Using C++
SECOND EDITION
STROUSTRUP
Addison Wesley
Database Principles
Computer Networkin
second edition

C++: cosa manca?

- Studio della **S**tandard **T**emplate **L**ibrary
- **Exception** e gestione degli errori
- **Singleton**: avere una sola istanza di una classe
- **Network** programming
- Gestione dei **segnali**
- Librerie **boost**
- Programmazione **multi-threading**
  - Esecuzione parallela di un programma
- **Estensioni** del C++: 11, 14, 17, 20
- E … qualche altro centinaio di pagine

Inserzione pubblicitaria

Se siete interessati

- **Tecniche Digitali** di **Acquisizione Dati**
  - io e Bob (Roberto Ferrari) per LT e LM
  - per la realizzazione di esperimenti di fisica
- Programma (syllabus) (kiro)
  - **Gestione** e **formattazione** dei dati
  - Progr. parallela: **multi-thread** e **GPU**
  - Progr. microprocessori: **Arduino**
  - Progr. schede elettroniche: **FPGA**
  - **Network** programming
  - ….