Lezione I: C++ ripasso ed approfondimenti

Laboratorio di Metodi Computazionali e Statistici (2023/2024)

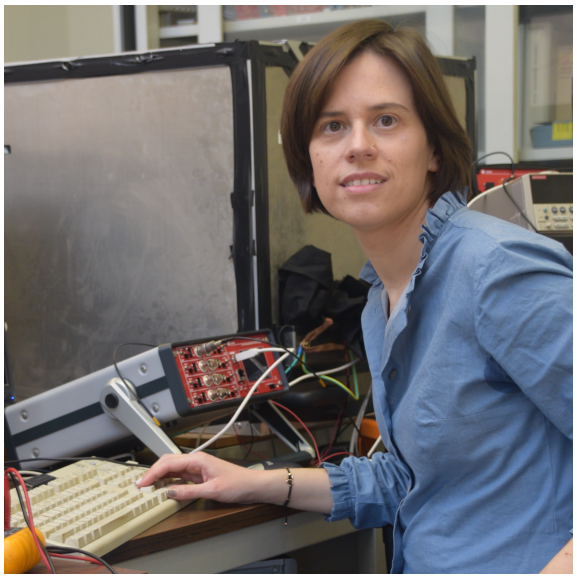
Fabrizio Parodi

DIFI

Docenti del corso

Docenti

Roberta Cardinale, S817, tel. 010 353-6209, roberta.cardinale@ge.infn.it
Fabrizio Parodi, S823, tel. 010 353-6657, fabrizio.parodi@ge.infn.it
Stefano Passaggio, L100, tel. 010 353-6047, stefano.passaggio@ge.infn.it



Roberta Cardinale (UniGe)

Fabrizio Parodi (UniGe)

Stefano Passaggio (INFN)

Obiettivi del corso

Docenti

Roberta Cardinale, S817, tel. 010 353-6209, roberta.cardinale@ge.infn.it
Fabrizio Parodi, S823, tel. 010 353-6657, fabrizio.parodi@ge.infn.it
Stefano Passaggio, L100, tel. 010 353-6047, stefano.passaggio@ge.infn.it

- Integrazione di programmazione/strumenti informatici
  - Programmazione OO in C++
    - Recap. di C++
    - Standard Template Library
    - Ereditarietà e polimorfismo
  - Scripting
    - shell scripting
    - make
  - Python
    - linguaggio base
    - cenni a classi
  - Applicativi/librerie
    - Matlab
    - ROOT

Obiettivi del corso

Docenti

Roberta Cardinale, S817, tel. 010 353-6209, roberta.cardinale@ge.infn.it
Fabrizio Parodi, S823, tel. 010 353-6657, fabrizio.parodi@ge.infn.it
Stefano Passaggio, L100, tel. 010 353-6047, stefano.passaggio@ge.infn.it

- Integrazione di programmazione/strumenti informatici
- Soluzione numerica di equazioni differenziali
  - Soluzione di equazioni differenziali ordinarie (riprendendo quanto iniziato a Lab. Calcolo): applicazioni sistemi a molti corpi, Equazione di Shrodinger (metodo di Numerov).
  - Soluzioni di equazioni differenziali alle derivate parziali (Potenziali, Onde, Calore, Meccanica quantistica).
  - Esperienze:
    - Sistemi gravitazionali a più corpi
    - Equazione di Shrodinger indipendente dal tempo
    - Equazioni differenziali alle derivate parziali (potenziale di un condensatore reale, propagazione del calore, equazione di Schrodinger dipendente dal tempo (pacchetti d'onda)).

Obiettivi del corso

Docenti

Roberta Cardinale, S817, tel. 010 353-6209, roberta.cardinale@ge.infn.it
Fabrizio Parodi, S823, tel. 010 353-6657, fabrizio.parodi@ge.infn.it
Stefano Passaggio, L100, tel. 010 353-6047, stefano.passaggio@ge.infn.it

- Integrazione di programmazione/strumenti informatici
- Soluzione numerica di equazioni differenziali
- Introduzione ai metodi di Monte Carlo
  - Generazione di numeri casuali
  - Generazione di sequenze di numeri casuali secondo una data distribuzione di probabilità
  - Integrazione MonteCarlo

Obiettivi del corso

Docenti

Roberta Cardinale, S817, tel. 010 353-6209, roberta.cardinale@ge.infn.it
Fabrizio Parodi, S823, tel. 010 353-6657, fabrizio.parodi@ge.infn.it
Stefano Passaggio, L100, tel. 010 353-6047, stefano.passaggio@ge.infn.it

- Integrazione di programmazione/strumenti informatici
- Soluzione numerica di equazioni differenziali
- Introduzione ai metodi di Monte Carlo
- Strumenti statistici avanzati per l'analisi dati
  - Integrazione su propagazione degli errori, misure correlate, matrice degli errori.
  - Metodo di massima verosimiglianza (binned, unbinned). Stima puntuale, stima di intervalli. Limiti. Test d'ipotesi. Cenni a metodi di classificazione multivariata (Machine Learning).

Obiettivi del corso

Docenti

Roberta Cardinale, S817, tel. 010 353-6209, roberta.cardinale@ge.infn.it
Fabrizio Parodi, S823, tel. 010 353-6657, fabrizio.parodi@ge.infn.it
Stefano Passaggio, L100, tel. 010 353-6047, stefano.passaggio@ge.infn.it

- Integrazione di programmazione/strumenti informatici
- Soluzione numerica di equazioni differenziali
- Introduzione ai metodi di Monte Carlo
- Strumenti statistici avanzati per l'analisi dati.
  Esperienze:
  - Simulazione/progettazione di un semplice esperimento
  - Misura di grandezze fisiche fondamentali attraverso misure elettriche:
    - e/h dalla curva IV di un LED (acquisizione ed analisi dati)
    - e/k dalla curva IV di un transistor (acquisizione ed analisi dati)
    - h/k dalla radiazione di corpo nero (lampadina) (acquisizione ed analisi dati)
    - e da esperimento di Millikan (analisi dati)

Calendario corso e primi appuntamenti

- Lezioni: 2 ore lunedì (9:00-11:00) e mercoledì (9:00-11:00)
- Laboratorio: giovedì e venerdì pomeriggio (14:00-18:00, due turni): 7 esperienze + 4 hands-on. Frequenza obbligatoria.
- Lezioni in presenza (online su Teams in caso di allerta meteo). Registrazioni disponibili su richiesta, per studenti lavoratori.
- Esercitazioni in aula informatica a postazione singola o doppia (max 30 per turno). Compilate il form su aulaweb per la scelta del giorno (scadenza 29/09). Prima esercitazione: 5 e 6/10.

Modalità d'esame

Composizione del voto:

$$Voto = 0.35(scritto) + 0.35(orale) + 0.3(val.\ itinere:\ \ frequenza\ +\ \ bonus)$$

- Valutazione in itinere (frequenza): 18 se si completano 7 esperienze su 7 e 4 "hands-on" su 4 (-2 per ogni esperienza persa, -1 per ogni "hands-on" perso). Voto minimo per ammissione allo scritto: 15. Le esperienze possono recuperate nel numero massimo di due (la prima parte della sesta non può essere recuperata)
- Valutazione in itinere (bonus):
  - al termine di ogni esercitazione verrà assegnato un compito a caso da svolgere singolarmente. Valutazione
    - 0: non consegnato o identico (o molto simile) a un altro elaborato
    - 1/2/3: elaborato originale (in funzione dei risultati ottenuti)

    Attenzione: questo non vuol dire che non dobbiate collaborare... Anzi ! Ma dovete essere in grado di re-interpretare personalmente la strategia concordata con i colleghi. Valutazione finale: 0-15
- Esame al calcolatore (soluzione di 2 problemi su 3 in 2 ore). Ammissione all'orale con voto ≥ 15.
- Esame orale

Materiale didattico

- Il programma tocca molti argomenti per cui non è banale indicare un unico testo di riferimento: è sufficiente il materiale su AW del corso.
- Canale Teams: codice: *s0w2gsy*.

Installazione del software sul proprio pc

- Istruzioni su aulaweb: provate subito oggi pomeriggio !
- Sessione aperta per problemi installazione software: 28/09 e 29/09 (14:00-16:00) in aula 603
- Canale Teams apposito, comune primo/terzo anno: codice: *31nc918*. Utilizzatelo per segnalare problemi generali (canali separati per le diverse piattaforme).
- Tutor dedicato per aiuto per installazione SW: Lucrezia Boccardo.

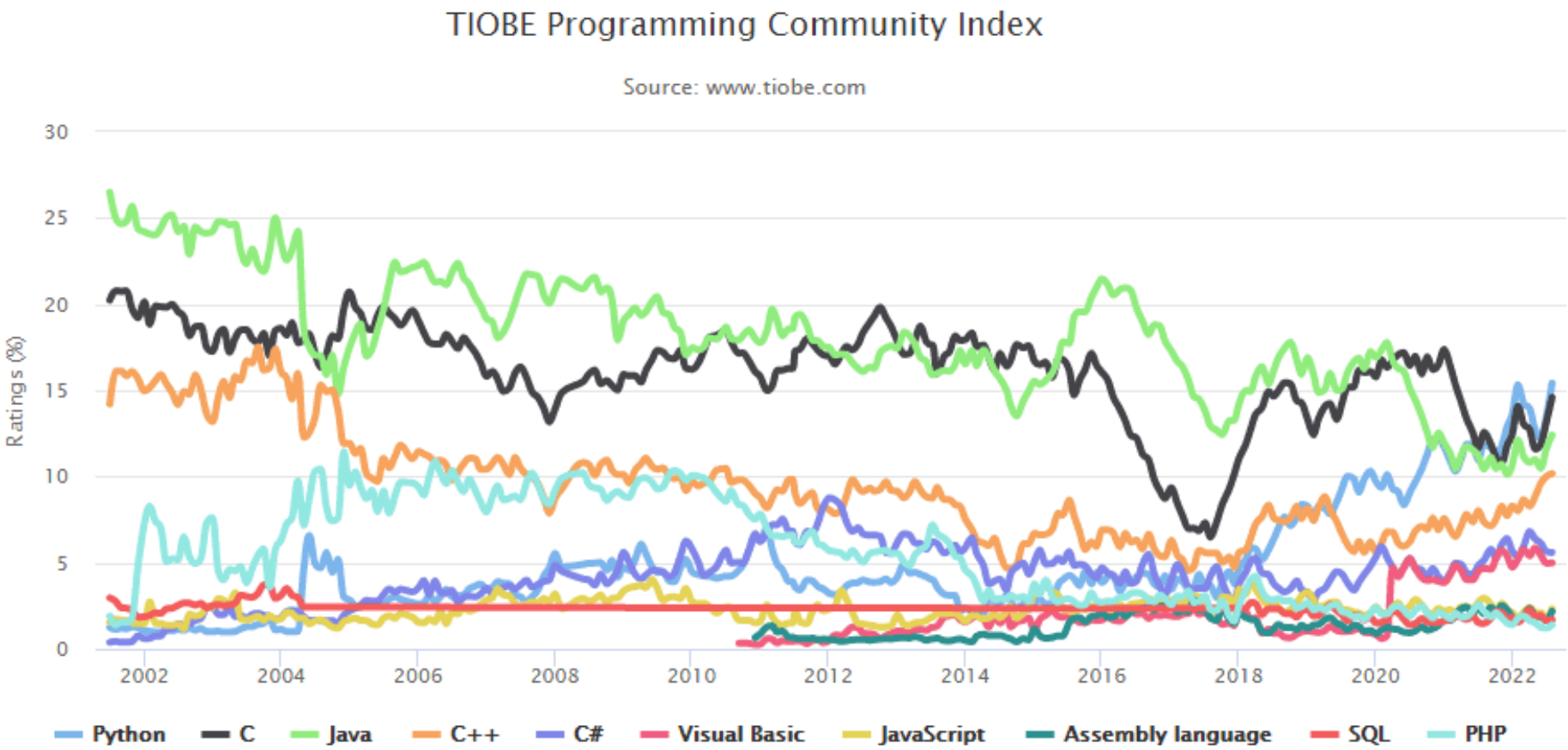
Programmazioni, disegno e complessità

- Scopo del software: risolvere un particolare problema fisico
  - problemi numerici, analisi dati, simulazione di fenomeni e apparati.
- La crescita esponenziale negli ultimi decenni della potenza di calcolo ha permesso di affrontare problemi sempre più complessi
- Come conseguenza il software deve diventare sempre più potente e strutturato (per risolvere problemi complessi)
  - Google $\sim$ 2000M linee di codice (C, C++, Java, Python, PHP)
  - Windows $\sim$ 50M linee di codice (C, C++, C#)
  - Linux $\sim$ 22M linee di codice (C)
  - Esperimento di fisica delle particelle (tipicamente, in fisica, la realtà con più software centralizzato) $\sim$ 6M linee di codice (C++, Python)

Filosofia di programmazione

- Per descrivere efficacemente problemi complessi occorre scomporre il codice in piccoli moduli, minimizzando le dipendenze tra i moduli.
- Tradizionalmente questo scopo è raggiunto tramite programmazione strutturata (C, Fortran, Pascal)
  - Modulatità e struttura del software garantita da funzioni che incapsulano (sotto) algoritmi
  - Le funzioni tuttavia sono spesso poco adatte per organizzare un codice molto complesso (che gestisce dati ed algoritmi)
- Linguaggi orientati agli oggetti (C++, Java, ..) vanno un passo oltre
  - Raggruppano i dati e le funzioni associate in oggetti (trattamento più naturale)
  - Incrementano modularità e riduzione della dipendenza aumentando la riusabilità del codice.
  - Promuovono l'astrazione degli strumenti di calcolo

Diffusione linguaggi di programmazione (TIOBE)



Programmazione Procedurale vs Object-Oriented

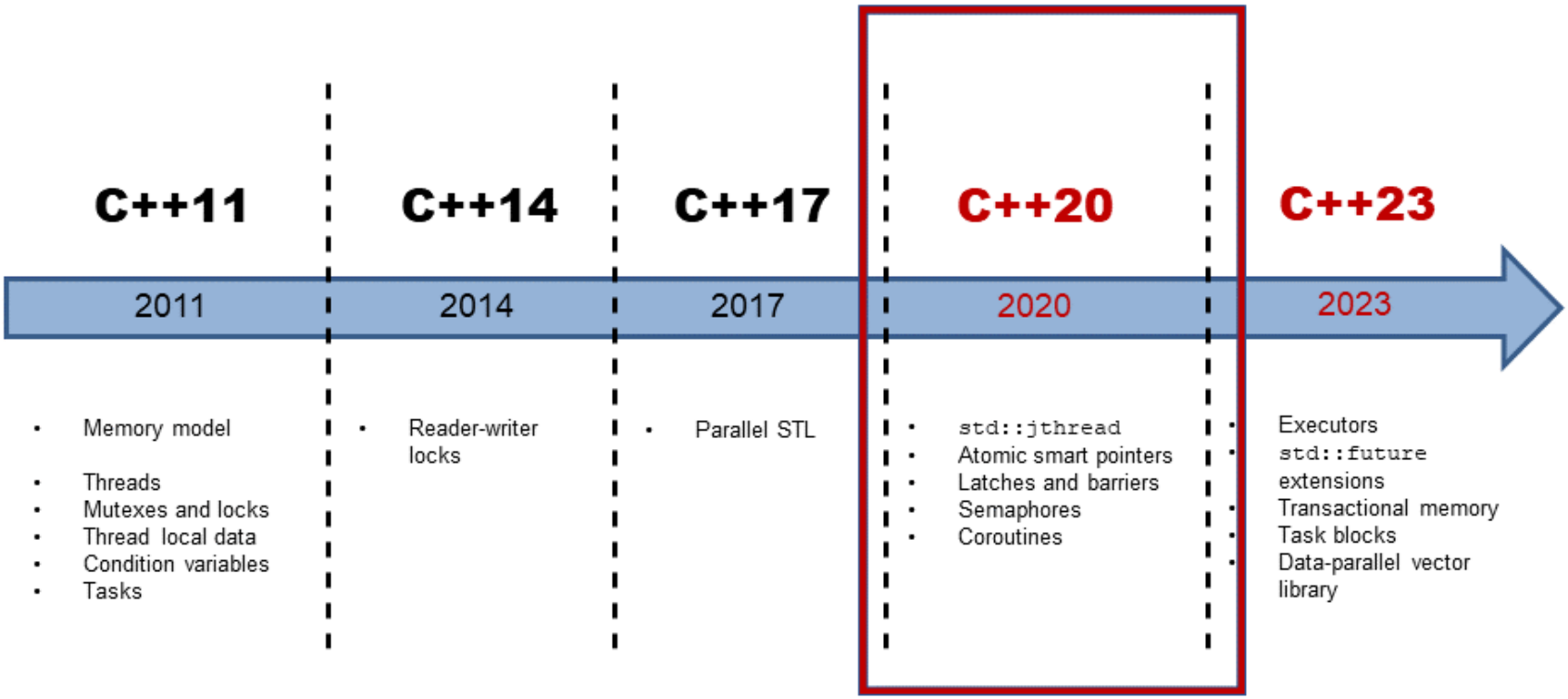
- I linguaggi procedurali agiscono in modo sequenziale su insiemi di dati modificandoli
  - I dati sono oggetti passivi e il programma consiste in una successione di istruzioni e funzioni che agiscono sui dati
- Nella programmazione orientata agli oggetti (OO) si supera la divisione tra istruzioni e dati e si parte dal concetto primitivo di oggetto
  - Un oggetto possiede uno stato (i dati) e dei metodi per la comunicazione con altri oggetti
  - Lo stato di un oggetto è modificabile tramite i suoi metodi
  - Il programma è composto da oggetti che interagiscono tra loro

Perché C++

- Linguaggio OO basato su C
  - Permette di sfruttare tutto le librerie di accesso diretto del C per interfacciare strumenti/acquisizione dati.
  - Permette di prendere dimestichezza con comandi comuni a tutti i linguaggi della C-family (C#, Java, etc...)
- Vantaggi/svantaggi:
  - Se usato nel modo giusto è il più performante tra tutti i linguaggi OO.
  - La sintassi è complessa. Solo sfruttandola bene si ottiene le prestazioni migliori. Per questo è necessario conoscere a fondo il C++ per utilizzarlo efficacemenre.
- C++ (OO) è usato per progetti di larga scala dove le prestazioni sono essenziali
  - C++ standard in grandi aziende e, in fisica, in esperimenti di fisica delle particelle (dovendo organizzare il coding di molti sviluppatori l'approccio OO è essenziale).
  - Quasi tutti i tool di alto livello (Matlab, Python,..) sono basati su codice C/C++.
  - Se il vostro programma è di $\mathcal{O}(100)$ line e le prestazioni non sono critiche altri linguaggi (Python, C++ procedurale) possono essere una valida alternativa.

C++ è "vivo"

Prima definizione nel 1998, aggiornamenti regolari ogni 3 anni.



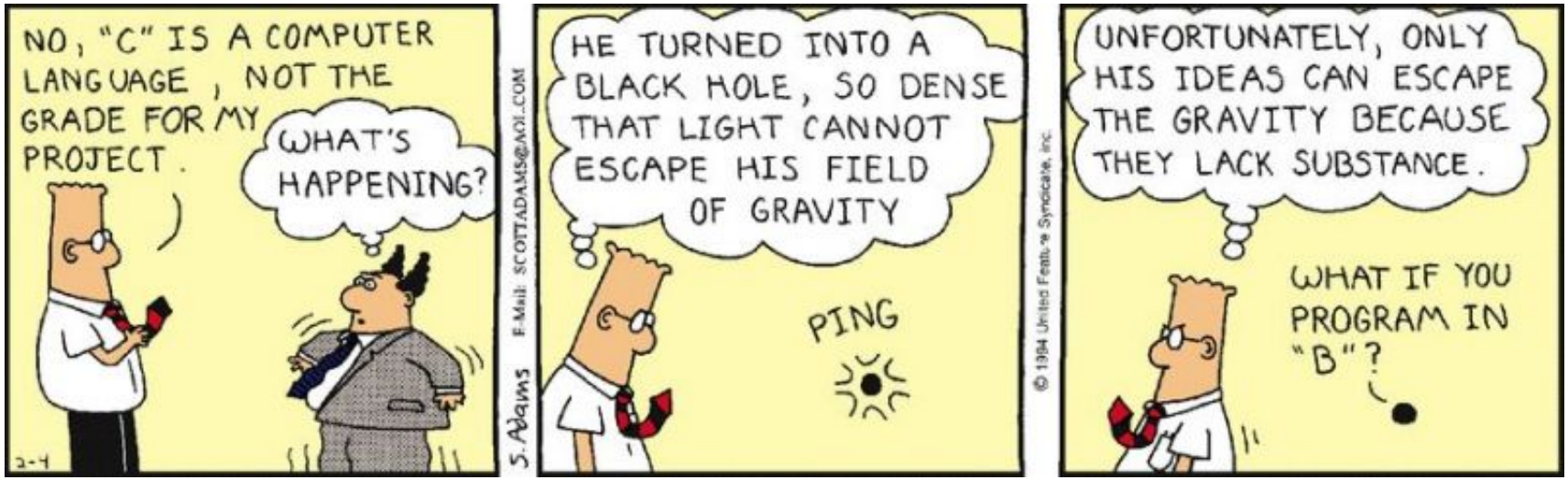
- Sulle macchine in aula informatica il default è C++14.
- Per cambiare versione del compilatore basta aggiungere in compilazione l'opzione

```
> g++ --std=c++17
```

Convenzioni di programmazione C++

- Le convenzioni, come tali, sono utili (non essenziali) per aumentare la leggibilità/uniformità del codice
- Dato il mio background vi mostrerò le convenzioni più comuni nella programmazione C++ in Fisica (in particolare in grandi collaborazioni internazionali)
- Tuttavia, da quest'anno, mostrerò anche le convenzioni di `GOOGLE`

Breve revisione di C/C++ procedurale (+classi)



Concetti della programmazione Object-Oriented

- Classe:
  - Definizione di nuovi tipi attraverso l'utilizzo dei tipi base e altre classi
  - Unificazione tra rappresentazione dei dati e metodi per la loro manipolazione
  - Gli oggetti sono le istanze delle classi
- Incapsulamento:
  - Lo stato dell'oggetto (la sua rappresentazione) non è "esposto" all'utente, ma solo ai metodi della classe
- Ereditarietà
  - Possibilità di derivare nuove classi da quelle già esistenti, prevenendo inutili duplicazione di codice
  - Le classi derivate ereditano caratteristiche e proprietà delle classi base
- Polimorfismo
  - La frase chiave è "un'interfaccia più metodi": possibilità di definire un singola interfaccia (per es. il nome di un metodo) che viene mappata su molteplici definizioni ognuna specifica per un determinato tipo
  - Tre tipologie: overloading, template e polimorfismo run-time

Dichiarazione di una classe

La dichiarazione di una classe prevede diverse regioni di visibilità

```
1  class ExampleClass{
2    public:
3    Type variable;
4    ...
5    protected:
6    Type variable;
7    ...
8    private:
9    Type variable;
10   ...
11 };
```

private
: parte accessibile solo ai metodi della classe ma non all'esterno di essa, in genere contiene la rappresentazione della classe (incapsulamento)

public
: la parte accessibile anche all'esterno della classe contiene i metodi che si vuole siano chiamabili all'esterno della classe

protected
: ... lo vedremo più in là.

Esempio di Classe

Ad esempio nella classe *string* (che avete usato spesso)

- la parte private contiene i dati membro della classe (la rappresentazione): nella fattispecie
  - almeno un array di char per contenere i caratteri della stringa
  - un intero per contenere le dimensioni attuali dell'array
- la parte public dove sono dichiarati i metodi pubblici della classe
  - ad es. `length()` (dimensione), `c_str()` (conversione a stringa C), etc...

Strutture

- Una struttura (struct) è una classe in cui tutti i membri sono pubblici

```
1       struct ExampleStruct{
2         Type variable;
3         Type variable;
4         Type variable;
5         ...
6       };
```

- Si suggerisce in genere di usare una struct solo per semplici collezioni di dati, se c'è la necessità di funzioni che agiscano sugli oggetti si consiglia di usare la classe (anche convenzione `GOOGLE-STYLE`).

Costruzione della classe Vector3

Riprendiamo l'esempio fatto al primo anno (classe *Vector3*):

- Le classi (come le funzioni, del resto) devono essere sia dichiarate che implementate (definite)
  - È possibile sia dichiarare che implementare una classe nello stesso blocco di istruzioni e farlo nello stesso file che contiene il main del programma
  - È però buona norma mettere la dichiarazione di ogni classe in un file header specifico e le implementazioni in file .cpp corrispondenti
- Nella fattispecie avremo tre file
  - Vector3.h: che contiene la dichiarazione della classe
  - Vector3.cpp: che contiene l'implementazione di ciò che è stato dichiarato in Vector3.h
  - main.cpp: che servirà a validare e usare la classe Vector
- Compilazione:

```
g++ -o main main.cpp Vector3.cpp
```

  oppure in modalità compilazione separata:

```
g++ -c Vector3.cpp
g++ -o main main.cpp Vector3.o
```

Dichiarazione della classe

- Dichiarazione di una nuova classe (che finirà in Vector3.h):

```
class Vector3{ // Nuova classe chiamata Vettore
 public:       // Interfaccia pubblica
   ...         // Metodi pubblici
 private:      // Parte privata: incapsulamento
   ...         // Dati membro (rappresentazione)
};             // NB: ';' alla fine!
```

- Rappresentazione (dati membro)
  - Essendo un numero vettore composto da tre reali, la rappresentazione sarà un'array a tre componenti o una terna di variabili (la prima scelta è più conveniente perché permette di utilizzare i loop) che andranno nella parte private della dichiarazione della classe

```
double m_v[3];
```

- Convenzioni
  - Per distinguere facilmente le variabili membro dalle altre variabili del programma è buona norma (non obbligatorio ma fortemente caldeggiato) usare il prefisso `m_` (ma si trova anche `_` o `__`) per i nomi dei dati membro.
    `GOOGLE-STYLE`: "trailing snake" postfisso `_` (nel caso specifico `v_[3]`).
  - Per i nomi composti: CamelCase (ad esempio: PuntoMateriale, in questo caso coincide con il `GOOGLE-STYLE`).

Dichiarazione della classe

```
1  // File Vector3.h
2  class Vector3{
3  public:
4    // per ora vuota
5  private:
6    double m_v[3];
7  };
```

```
1  // File Vector3.cpp
2  #include "Vector3.h"
3  // Conterra' la definizione dei
       metodi
```

```
1  // File main.cpp
2  #include "Vector3.h"
3  int main(){
4    return 0;
5  }
```

```
g++ -o main main.cpp Vector3.cpp
```

Costruttore

- Il costruttore è un metodo particolare: viene chiamato nel momento in cui un oggetto appartenente alla classe viene creato. Il nome del metodo deve coincidere con quello della classe (Vector3) e non deve avere tipo (neanche void). Si possono usare parametri che vengono utilizzati per inizializzare l'oggetto.
- Se il costruttore non è definito il sistema lo definisce automaticamente. Ma, attenzione, se definite un costruttore non default dovete aggiungere anche quello default. Senza di questo è, di fatto, impossibile costruire un array (new usa il costruttore default).
- Di fatto il costruttore non fa altro che inizializzare l'oggetto e lo può fare in due modi:
  - nel suo "corpo"
  - attraverso un lista di inizializzazione

  D'ora in poi useremo la lista di inizializzazione. Il vantaggio principale è che assicura che le variabili siano inizializzate prima di essere utilizzate.
- Si può utilizzare l'overloading per diminuire il numero di diversi costruttori da dichiarare

Costruttore (assegnazione nel corpo)

```
1  // File Vector3.h
2  class Vector3{
3  public:
4    Vector3(double x, double y, double z);
5    Vector3();
6  private:
7    double m_v[3];
8  };
```

```
1  // File Vector3.cpp
2  #include "Vector3.h"
3  Vector3::Vector3(double x, double y, double z){
4    m_v[0]=x;
5    m_v[1]=y;
6    m_v[2]=z;
7  }
8  Vector3::Vector3(){
9    m_v[0]=0;
10   m_v[1]=0;
11   m_v[2]=0;
12 }
```

Costruttore (assegnazione con lista)

```
1  // File Vector3.h
2  class Vector3{
3  public:
4    Vector3(double x, double y, double z):m_v{x,y,z}{}
5    Vector3():m_v{0,0,0}{}
6  private:
7    double m_v[3];
8  };
```

Attenzione la sintassi usata per l'array è C++11.

In generale per una variabile privata la sintassi di inizializzazione è

```
1  myClass(double var):m_var(var);
```

Costruttore (assegnazione con lista e overloading)

```
1  // File Vector3.h
2  class Vector3{
3  public:
4    Vector3(double x=0, double y=0, double z=0):m_v{x,y,z}{}
5  private:
6    double m_v[3];
7  };
```

Esempio (per tutti e tre i casi)

```
1  // File main.cpp
2  #include "Vector3.h"
3  int main(){
4    Vector3 a;          // costruttore default
5    Vector3 b(1,1,1); // costruttore non default
6    Vector3 c(1);       // comp. y e z a zero
7    Vector3 *v = new Vector3[10]; // 10 Vector3 creati
8                                  // con il costruttore default
9  }
```

Funzione di accesso: setters & getters

- Aggiungiamo i metodi per accedere (getters) o modificare il vettore (setters)

```
1  // File Vector3.h
2  class Vector3{
3  public:
4    ...
5    double X();      // getter
6    void X(double); // setter
7    ...
8  private:
9    double m_v[3];
10 };
```

```
1  // File Vector3.cpp
2  #include "Vector3.h"
3  ...
4  double Vector3::X(){
5    return m_v[0];
6  }
7  void Vector3::X(double x){
8    m_v[0]=x;
9  }
10 ...
```

```
1  // File main.cpp
2  #include <iostream>
3  #include "Vector3.h"
4  using namespace std;
5  int main(){
6    Vector3 a(1,1,1); // crea vettore (1,1,1)
7    a.X(2);
8    cout << "La comp. x vale " << a.X() << endl;
9    return 0;
10 }
```

Il costruttore di copia

- Ogni classe ha due costruttori: il costruttore e il costruttore di copia

```
1    myClass();
2    myClass(const myClass &);
```

- Il costruttore di copia viene chiamato ogni qual volta si crea una copia di un oggetto

```
1    myClass x(y);
```

- Così come per il costruttore, se il costruttore di copia non è definito il sistema lo definisce automaticamente.
- Quando viene chiamato il costruttore di copia, questi effettua una copia dello stato attuale dell'oggetto in un nuovo oggetto della stessa classe.

Il distruttore

- Il distruttore ha la forma

  `~myClass()`

  Ce ne può essere al massimo uno per classe. Se il distruttore non è definito esplicitamente è creato automaticamente.
- Viene chiamato quando si esce dal campo di visibilità di un oggetto della classe allocato staticamente o quando si applica l'operatore **delete** ad un oggetto della classe allocato dinamicamente.
- Esempio:

```
1  int main(){
2    myClass *v = new myClass[10];
3    myClass a;
4    if (...){
5      myClass b;
6    }              // -> agisce il distruttore di b
7    delete[] v;    // -> agisce il distruttore su v[]
8  }                // -> agisce chiamato il distruttore di b
```

- In realtà il distruttore default spesso non fa nulla (chiama il distruttore dei suoi "componenti").

L'operatore di assegnazione

- Permette di copiare un oggetto in altro

```
1 Classe& operator=(const Classe &)
```

come per costruzione e distruzione, se non esiste, viene creato. L'operatore si occupa di copiare lo stato di un certo oggetto in un altro.

- Esempio:

```
1       myClass b,c;
2       myClass a = b; // operatore di copia
3       c = b;         // operatore di assegnazione
```

Definizione esplicita di distruttore, costruttore di copia, operatore di assegnazione

- Quando una classe contiene, tra i sui membri privati, oggetti/variabili allocate dinamicamente allora è necessario la definizione esplicita di distruttore, costruttore di copia, assegnazione:
- Ad esempio:

```
1  // File VectorN.h
2  class VectorN{
3  public:
4    VectorN(int n=3);
5  private:
6    int m_n;
7    double *m_v;
8  };
```

```
1  // File VectorN.cpp
2  VectorN(int n){
3      m_n  = n;
4      m_v = new double[n];
5  }
```

Distruttore

- Il distruttore "default" si limiterebbe a cancellare la zona di memoria riservata al puntatore `m_v`
- Il distruttore non banale, necessario per liberare la memoria, è:

```
1 VectorN::~VectorN(){
2   delete[] m_v;
3 };
```

Costruttore di copia

- Il costruttore di copia "default" copierebbe i membri privati (cioè il puntatore) creando così un oggetto vettore che punterebbe alla stessa zona di memoria del vettore originario.
- Per evitare ciò (costruendo così una vera copia):

```
1   VectorN::VectorN(const VectorN &a){
2     m_n = a.m_n;
3     m_v = new double[m_n];
4     for (int i=0;i<n;i++)
5       m_v[i] = a.m_v[i];
6   }
```

Operatore di assegnazione

- L'operatore di assegnazione "default" copierebbe l'indirizzo da un puntatore all'altro (assegnato e assegnando identici).
- Si definisce quindi il metodo:

```
1  Classe& VectorN::operator=(const VectorN &a){
2    if (m_n != a.m_n){
3      cout << "Error message " << endl;
4    } else {
5      for (int i=0;i<n;i++)
6        m_v[i] = a.m_v[i];
7    }
8  }
```

Regole del Tre

La regola del tre è una regola empirica di programmazione in C++ che afferma che:

- *Se la classe definisce esplicitamente almeno uno tra distruttore, costruttore di copia e operatore di assegnamento, allora tutti e tre i metodi devono essere definiti esplicitamente*

La motivazione di questa regola nasce dal fatto che, usualmente, le motivazioni che richiedono l'implementazione di uno di questi metodi (altrimenti implementati per default) implica l'implementazione anche degli altri due.

Definiamo gli operatori matematici

- Il C++ consente di (ri)definire il modo in cui gli operatori (+, -, *, /, ...) agiscono sugli elementi di una classe.
- Se a e b sono oggetti di una classe, allora per definire il comportamento di

  ```
  c = a + b;
  d = -a;
  ```

  si deve pensare che l'operazione sia scritta come

  ```
  c = a.operator+(b);
  d = a.operator-;
  ```

  e definire gli operatori in maniera opportuna.
- Nel caso di operatori binari (nel senso che operano tra due operandi), la definizione dell'operatore spetta alla classe che compare a sinistra.
- Se necessario l'operatore può anche essere definito come funzione (non come metodo)

  ```
  c = operator+(a,b);
  ```

Definiamo gli operatori matematici

```
// File Vector3.h
class Vector3{
public:
  ...
  Vector3 operator+(Vector3);
  Vector3 operator-();
  ...
private:
  double m_v[3];
};
```

```
// File main.cpp
#include <iostream>
#include "Vector3.h"
using namespace std;
int main(){
  Vector3 a(1,1,1),b(1,2,1);
  Vector3 c = a+b;
  Vector3 d = -a;
  ...
  return 0;
}
```

```
// File Vector3.cpp
#include "Vector3.h"
...
Vector3 Vector3::operator+(
    Vector3 b){
  Vector3 res;
  for (int i=0;i<3;i++)
    res.m_v[i] = m_v[i] + b.m_v
        [i];
  return res
}

Vector3 Vector3::operator-(){
  Vector3 res;
  for (int i=0;i<3;i++)
    res.m_v[i] = -m_v[i];
  return res
}
...
```

Moltiplicazione per scalare

- Definizione V*α

```
1  // File Vector3.h
2  class Vector3{
3  public:
4    ...
5    Vector3 operator*(double);
6    ...
7  private:
8    double m_v[3];
9  };
```

```
1  // File main.cpp
2  #include <iostream>
3  #include "Vector3.h"
4  using namespace std;
5  int main(){
6    Vector3 a(1,1,1);
7    Vector3 c = a*2;
8    ...
9    return 0;
10 }
```

```
1  // File Vector3.cpp
2  #include "Vector3.h"
3  ...
4  Vector3 Vector3::operator*(
       double b){
5    Vector3 res;
6    for (int i=0;i<3;i++)
7      res.m_v[i] = m_v[i]*b;
8    return res
9  }
```

Moltiplicazione per scalare

- Definizione V*α.. e di α*V

```
1  // File Vector3.h
2  Vector3 operator*(double,
       Vector3);
3  class Vector3{
4  public:
5    ...
6    Vector3 operator*(double);
7    ...
8  private:
9    double m_v[3];
10 };
```

```
1  // File main.cpp
2  #include <iostream>
3  #include "Vector3.h"
4  using namespace std;
5  int main(){
6    Vector3 a(1,1,1);
7    Vector3 c = a*2;
8    Vector3 d = 2*a;
9    ...
```

```
1  // File Vector3.cpp
2  #include "Vector3.h"
3  ...
4  Vector3 operator*(double b,
       Vector3 V){
5    return V*b;
6  }
7
8  Vector3 Vector3::operator*(
       double b){
9    Vector3 res;
10   for (int i=0;i<3;i++)
11     res.m_v[i] = m_v[i]*b;
12   return res
13 }
```