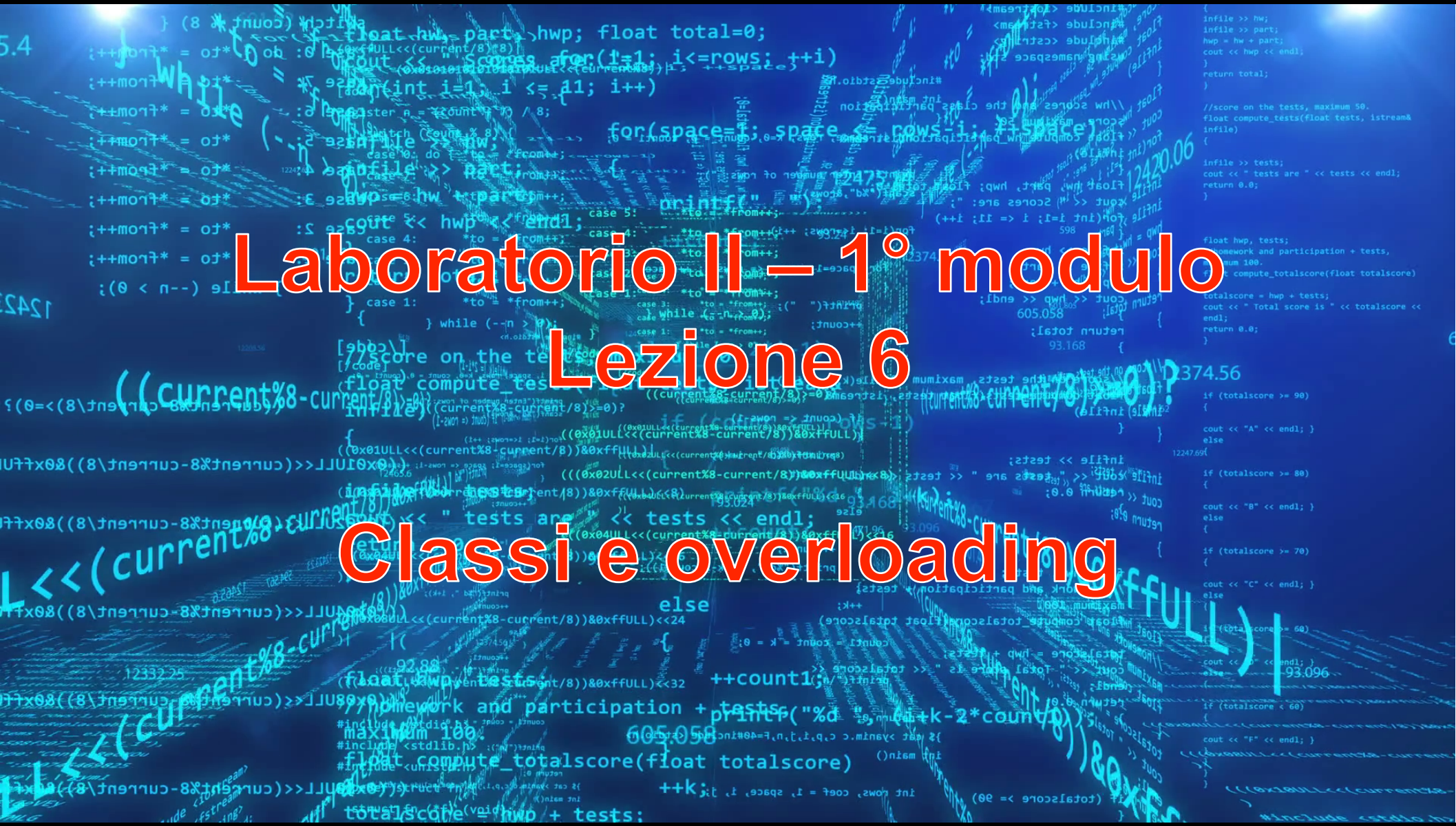
Laboratorio II – 1° modulo
Lezione 6
Classi e overloading

UNIVERSITA' DEGLI STUDI DI MILANO BICOCCA

Indice

- Ancora sulle classi:
  - Diversi tipi di costruttori: default constructor, copy constructor, …
  - Come ridefinire gli operatori **=**, **+**, ecc …
- Overloading degli operatori nelle classi del C++
- Overloading delle funzioni in C++

- Esercizi:
  - La classe dei numeri complessi
  - La classe data del calendario

Tanti modi di istanziare

```
int A;
int B = 10;
int numCopy(B);
int numCopy = B;

// Istogramma di "default"
istogramma DefaultHisto;

// Costruttore con valori di
// inizializzazione
istogramma MyHisto(Nbin,min,max);

// Copy constructor
istogramma HistoCopy1(MyHisto);
```

E` possibile istanziare le variabili numeriche (e.g. `int`) in diversi modi:

- senza valore di inizializzazione
- con valore di inizializzazione
- inizializzandole al valore di un'altra variabile dello stesso tipo

Allo stesso modo è possibile istanziare gli oggetti di una classe in diversi modi:

- senza parametri (default)
- con parametri di inizializzazione
- inizializzandoli uguali a un altro oggetto dello stesso tipo

Per ogni modo di istanziare esiste un relativo costruttore:
tutti i costruttori sono da scrivere esplicitamente

Diversi costruttori (file `.h`)

Default constructor: viene creato un oggetto i cui data member sono inizializzati con parametri di default

```
istogramma();
```

Costruttore: i data member dell'oggetto sono inizializzati a partire da uno o più parametri passati come argomenti del costruttore

```
istogramma(int Nbin, float min, float max);
```

Copy constructor: i data member dell'oggetto sono inizializzati uguali a quelli di un oggetto dello stesso tipo già esistente (`istogramma original`), passato per referenza

```
istogramma(const istogramma& original);
```

Default constructor (file .cc)

Default constructor: viene creato un oggetto i cui data member sono inizializzati con parametri di default

```
istogramma::istogramma()
{
  Nbin_p       = 0;
  min_p        = 0.;
  max_p        = 0.;
  step_p       = 0.;
  binContent_p = NULL;
  underflow_p  = 0;
  overflow_p   = 0;
  entries_p    = 0;
  sum_p        = 0.;
  sumSq_p      = 0.;
  std::cout << "Default constructor" << std::endl;
}
```

Copy constructor (file `.cc`)

Copy constructor: i data member dell'oggetto sono inizializzati uguali a quelli di un oggetto dello stesso tipo già esistente (`istogramma original`), passato per referenza

```
istogramma::istogramma(const istogramma& original)
{
  Nbin_p       = original.Nbin_p;
  min_p        = original.min_p;
  max_p        = original.max_p;
  step_p       = original.step_p;
  binContent_p = new int [Nbin_p];
  for (int i = 0; i < Nbin_p; i++)
    binContent_p[i]=original.binContent_p[i];
  underflow_p = original.underflow_p;
  overflow_p  = original.overflow_p;
  entries_p   = original.entries_p;
  sum_p       = original.sum_p;
  sumSq_p     = original.sumSq_p;
  std::cout << "Copy constructor" << std::endl;
}
```

- I data member **`private`** dell'**`oggetto original`** sono accessibili

Tutti i data member di **`original`** vengono copiati

- Gli elementi dell'array vengono copiati uno ad uno con un ciclo `for`

Utilizzo nel `main`

```
// Istogramma di "default"
istogramma DefaultHisto;


const int Nbin  = 30;
const float min = 0;
const float max = 10;
// Costruttore con valori di inizializzazione
istogramma MyHisto(Nbin, min, max);


// Copy constructor
istogramma HistoCopy1(MyHisto);
...
istogramma HistoCopy1 = MyHisto;
```

Attenzione: default constructor chiamato senza le parentesi (altrimenti il compilatore lo interpreta come dichiarazione di una nuova funzione)

Esiste anche questo modo per eguagliare due istogrammi, cioè tramite l'operatore =

L'operator=

Vediamo ora come fare per modificare i data member di un istogramma già esistente, rendendoli uguali a quelli di un altro istogramma tramite la ridefinizione e l'uso dell'operatore **=**

Esempio:

```
istogramma DefaultHisto;
istogramma MyHisto (Nbin, min, max);
...
DefaultHisto = MyHisto;
```

Poiché il tipo `istogramma` non è un tipo di *default* del `C++`, per il calcolatore non è ovvio come uguagliare due istogrammi tramite l'operatore binario **=**
È necessario quindi implementare un metodo apposito che gestisca correttamente la copia di tutti i valori dei data member di un oggetto istogramma in un altro oggetto istogramma

L'operator=

Prototipo: file .h

Come per il copy constructor, l'argomento è una *reference* a un oggetto istogramma `const`

```
// operator=
istogramma& istogramma::operator=(const istogramma& original);
```

Implementazione: file .cc

Ritorna una *reference* ad un oggetto istogramma

```
// operator=
istogramma& istogramma::operator=(const istogramma& original)
{
  Nbin_p       = original.Nbin_p;
  min_p        = original.min_p;
  max_p        = original.max_p;
  step_p       = original.step_p;
  binContent_p = new int [Nbin_p];
  for (int i = 0; i < Nbin_p; i++)
    binContent_p[i] = original.binContent_p[i];
  underflow_p = original.underflow_p;
  overflow_p  = original.overflow_p;
  entries_p   = original.entries_p;
  sum_p       = original.sum_p;
  sumSq_p     = original.sumSq_p;
  std::cout << "Operator =" << std::endl;
  return *this;
}
```

`this` è un puntatore che contiene l'indirizzo dell'oggetto della classe che ha invocato il metodo

Nel nostro caso l'`operator=` restituisce l'oggetto (`*this`) per poter eseguire: `oggetto a = b = c = d;`

L'operator=

Vi sarete chiesti quale sia motivo per restituire una referenza e non un valore in `istogramma& istogramma::operator=(const istogramma& original);` Sostanzialmente è per risparmiare operazioni. Provate questo semplice programma con e senza la referenza:

```
#include <iostream>
using namespace std;
class A {
public:
  A(int n) { // Constructor
    cout << __PRETTY_FUNCTION__ << endl;
    n_p = n; }
  // Destructor
  ~A() { cout << __PRETTY_FUNCTION__ << endl; }

  A(const A& a) { // Copy constructor
    cout << __PRETTY_FUNCTION__ << endl;
    n_p = a.n_p; }

  // Overloading operator=
  A& operator=(const A& a) {
    n_p = a.n_p;
    return *this; }
  void get() {
    cout << "n_p = "
         << n_p << endl; }
private:
  int n_p;
};

int main() {
  A a(1);
  A aa(10);
  aa = a;
  aa.get();
  return 0; }
```

Output con referenza

```
A::A(int)
A::A(int)
n_p = 1
A::~A()
A::~A()
```

Output senza referenza

```
A::A(int)
A::A(int)
A::A(const A &)
A::~A()
n_p = 1
A::~A()
A::~A()
```

Utilizzo nel `main`

```
istogramma a;
istogramma b (Nbin, min, max);
```

Costruisce due oggetti `istogramma`

```
a = b;
// Equivalentemente
// a.operator=(b);
```

Viene chiamato l'`operator=()` dell'oggetto `a` → l'oggetto `a` viene cosi modificato ed uguagliato all'oggetto `b`

```
istogramma c;
c = a = b;
// Equivalentemente
// c.operator=(a.operator=(b));
```

`b` → `a` → `c` : l'istogramma `b` viene copiato nell'istogramma `a`, che a sua volta viene copiato nell'istogramma `c`

Overloading degli operatori

Il caso che abbiamo appena descritto dell'`operator=` costituisce un esempio del cosiddetto *overloading* degli operatori

Si usa lo stesso simbolo (e.g. `=`, `+`, `-`, `*`, `/`, …) per implementare operazioni differenti a seconda del tipo di variabile

- E.g.: per il calcolatore è diverso eseguire una divisione tra `int` rispetto ad una divisione tra `double`

Possiamo ridefinire (i.e. fare l'*overload*) della maggior parte degli operatori disponibili in `C++` in modo da poterli utilizzare con i nuovi tipi definiti dal programmatore mediante classi

- E.g.: possiamo implementare la somma fra istogrammi, scrivendo un metodo che sommi il contenuto bin per bin e restituisca il nuovo istogramma ottenuto

Overloading degli operatori

- Quando realizziamo l'overloading di un operatore di fatto stiamo implementando un nuovo metodo della classe
- Al posto di inventare un nome per questo nuovo metodo della classe riciclamo uno dei simboli già usati in C++ per le operazioni sui tipi standard

Esempio:

- Per sommare due oggetti istogramma potrei implementare un metodo `Somma`, oppure, equivalentemente, realizzare l'overloading dell'`operator+`

Prototipi: file `.h`

```
istogramma Somma(const istogramma& histo2);
```

Utilizzo nel `main`

```
istogramma SumH = HistoA.Somma(HistoB);
```

```
istogramma operator+(const istogramma& histo2);
```

```
istogramma SumH = HistoA + HistoB;
```

- Grazie all'overloading degli operatori il codice diventa più sintetico ed intuitivo

Overloading degli operatori

L'`operator+` (come anche il metodo `Somma`) non ha bisogno di ricevere in input entrambi gli istogrammi da sommare. Infatti, essendo un metodo della classe, agisce sull'oggetto della classe che sta a sinistra del `+` (`HistoA`) e gli serve ricevere in input solo l'istogramma a destra del `+` (`HistoB`)

Prototipi: file `.h`

Utilizzo nel `main`

```
istogramma Somma(const istogramma& histo2);
```

```
istogramma SumH = HistoA.Somma(HistoB);
```

```
istogramma operator+(const istogramma& histo2);
```

```
istogramma SumH = HistoA + HistoB;

// Oppure equivalentemente

istogramma SumH = HistoA.operator+(HistoB);
```

Restituisce un oggetto di tipo `istogramma` che conterrà la somma dei due istogrammi

Overloading operator+ (.cc)

```
// Utilizzo nel main
istogramma SumH = HistoA + HistoB;


// Operator+
istogramma istogramma::operator+(const istogramma& histo2)
{
  // Creo un nuovo istogramma, identico a histo2
  istogramma SumHisto(histo2);

  if (Nbin_p != histo2.Nbin_p || min_p != histo2.min_p || max_p != histo2.max_p)
    {
      std::cout<<"Errore! I due istogrammi hanno binning differente!\n";
      return SumHisto;
    }

  SumHisto.underflow_p += underflow_p;
  SumHisto.overflow_p  += overflow_p;
  SumHisto.entries_p   += entries_p;
  SumHisto.sum_p       += sum_p;
  SumHisto.sumSq_p     += sumSq_p;

  for (int i = 0; i < Nbin_p; i++)
    SumHisto.binContent_p[i] += binContent_p[i];

  return SumHisto;
}
```

HistoB corrisponde a **histo2** nel .cc

HistoA è l'oggetto a cui viene applicato l'operator+, quindi nel .cc le variabili Nbin_p, min_p, max_p, ... e tutti gli altri data member contengono i valori dei parametri di HistoA

- SumHisto.binContent_p[i] è inizializzato ai conteggi di histo2 (cioè HistoB)
- Verifica che i due istogrammi abbiano lo stesso binning
- binContent_p[i] contiene i conteggi nei bin di HistoA

Restituisce la copia di SumHisto (non la sua *reference*) perché SumHisto è definito solo nello scope di operator+()

Overloading dei costruttori

Nella prima parte della lezione abbiamo visto che possono coesistere diversi tipi di costruttori della classe istogramma. Alla luce di quanto visto a proposito dell'overloading, possiamo dire che quello era un esempio di overloading dei costruttori

```
// Default constructor
istogramma();

// Costruttore
istogramma(int Nbin, float min, float max);

// Copy constructor
istogramma(const istogramma& original);
```

Puntatore ad un array contenente i valori degli estremi dei bin → così si possono costruire istogrammi con bin di larghezza variabile

Volendo, si possono implementare anche altri costruttori con diversi parametri di inizializzazione, ad esempio:

```
istogramma(int Nbin, const float* xbins)
```

Overloading delle funzioni

Analogamente si possono realizzare l'**overloading delle funzioni**, in modo che la stessa funzione agisca su differenti tipi o anche su un numero diverso di argomenti. Sarà poi il compilatore a scegliere a quale funzione far riferimento in base agli argomenti con cui verrà chiamata la funzione

Esempio: la **funzione raddoppia**

```
#include <iostream>

int raddoppia (int a) {return a*2);

int main()
{
  int   A = 2;
  float B = 2.22;
  std::cout << raddoppia(A) << std::endl;
  std::cout << raddoppia(B) << std::endl;
  return 0;
}
```

La funzione `raddoppia` definita per un `int` non funziona correttamente se cerco di utilizzarla con un `float`

Overloading delle funzioni

E` possibile implementare la funzione raddoppia anche per i `float`, usando lo stesso nome

```
#include <iostream>

int   raddoppia (int a)   {return a*2);
float raddoppia (float a) {return a*2);

int main()
{
  int   A = 2;
  float B = 2.22;
  std::cout << raddoppia(A) << std::endl;
  std::cout << raddoppia(B) << std::endl;
  return 0;
}
```

L'OVERLOADING consiste nell'avere una o più funzioni con lo stesso nome che agiscono su argomenti di diverso tipo

Esercizi

Esercizio 1: Scrivere la classe dei numeri complessi utilizzando l'header file (file `.h`) mostrato nella slide seguente

- Implementare i metodi:
  - `Re()` → parte reale
  - `Im()` → parte immaginaria
  - `Mod()` → modulo
  - `Rho()` e `Theta()` → descrizione polare
  - `Print()` → stampa parte reale e parte immaginaria

- Implementare gli operatori
  - `=(), +(), -(), *(), /()`
  - `=(const double& original)` // Operazioni con double
  - `^(const int& potenza)` // Elevamento a potenza intera

Esercizi

```
#ifndef COMPLESSO_H
#define COMPLESSO_H

class complesso
{
 public:
  // Constructors
  complesso  ();
  complesso  (const double& re, const double& im);
  complesso  (const complesso& original);
  ~complesso (); // Destructor

  // Operator overloads
  complesso& operator= (const complesso& original);
  complesso& operator= (const double& original);
  complesso  operator+ (const complesso& original);
  complesso  operator- (const complesso& original);
  complesso  operator* (const complesso& original);
  complesso  operator/ (const complesso& original);
  complesso  operator^ (const int& potenza);
...
```

```
...
  // Member functions
  double Re()    const;
  double Im()    const;
  double Mod()   const;
  double Theta() const;
  void   Print() const;

  // Data members
 private:
  double re_p;
  double im_p;

};

#endif
```

Esercizi

Esercizio 2: Scrivere la classe data del calendario utilizzando l'header file (file `.h`) mostrato nella slide seguente

- Oltre ai costruttori e operatori indicati nel file `.h`, implementate i seguenti metodi:
- `void stampa()` → scrive a terminale la data nel formato `GG/MM/AAAA`
- `void imposta(int day, int month, int year)` → reimposta giorno, mese e anno della data secondo i parametri di input
- `bool valida()` → restituisce `true/false` a seconda che la data inserita sia formalmente corretta oppure no (e.g. 34 febbraio 2018)
- `int diff_giorni(const data other)` → calcola il numero di giorni che intercorrono tra la data inserita e un'altra data, passata come parametro di input
- `bool Bisestile()` → restituisce `true/false` a seconda che l'anno della data inserita sia bisestile oppure no. Un anno è bisestile se è divisibile per 4 ma non per 100, oppure se è divisibile per 400: e.g. il 2000 era bisestile ma il 1900 no
- `const char* giorno_settimana()` → restituisce il giorno della settimana (si consiglia di sfruttare il metodo `diff_giorni` per calcolare il numero di giorni trascorsi rispetto ad una data di riferimento: ad esempio sappiamo che il `25/12/2023` sarà un sabato)

Esercizi

```
#ifndef DATA_H
#define DATA_H

class data
{
 public:
  data ();
  data (int day, int month, int year);
  data (const data& other);
  data& operator= (const data& other);
  bool operator== (const data& other) const;
  bool operator<  (const data& other) const;

  void stampa  () const;
  void imposta (int day, int month, int year);

  bool valida     () const;
  bool Bisestile  () const;
  int diff_giorni (const data other) const;
  const char* giorno_settimana () const;

  // Inline implementation
  bool operator!= (const data& other) const { return !this->operator==(other); }
  bool operator>  (const data& other) const { return other.operator<(*this);   }
  bool operator<= (const data& other) const { return !this->operator>(other);  }
  bool operator>= (const data& other) const { return !this->operator<(other);  }
...
```

```
...
 private:
  int year_p, month_p, day_p;
};

#endif
```