

Laboratorio II – 1° modulo

Lezione 6

Indice

- Ancora sulle **classi**:
 - Diversi tipi di costruttori (default constructor, copy constructor ...)
 - Operator=
- **Overloading delle funzioni** in C++
- **Overloading degli operatori** nelle classi del C++
- Esercizi:
 - La classe dei numeri complessi
 - La classe “data”

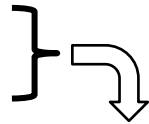
Tanti modi di istanziare

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

Possiamo istanziare (=definire) le **variabili numeriche** (es: `int`) in diversi modi:

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

```
// istogramma di "default"  
istogramma DefaultHisto;  
// costruttore con valori di inizializzazione  
istogramma MyHisto (Nbin, min, max);  
// copy constructor  
istogramma HistoCopy (MyHisto);  
istogramma HistoCopy2 = MyHisto;
```



Sintassi diverse con significato equivalente

Allo stesso modo possiamo 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!

Diversi costruttori (file .h)

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


```
//default constructor  
istogramma();
```

Costruttore: gli attributi dell'oggetto sono inizializzati a partire da uno o più parametri passati come argomenti del costruttore

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

Copy constructor: gli attributi dell'oggetto sono inizializzati uguali a quelli di un oggetto dello stesso tipo già esistente (istogramma original), passato per *referenza*

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



Default constructor (file .cc)

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

```
//default constructor
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: gli attributi dell'oggetto sono inizializzati uguali a quelli di un oggetto dello stesso tipo già esistente (istogramma original), passato per referenza

```
//copy constructor
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;
}
```

Gli attributi *private* dell'**oggetto original** sono accessibili

Tutti gli attributi 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 HistoCopy (MyHisto);  
istogramma HistoCopy2 = MyHisto;
```

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

L'operator=

E se all'interno del mio programma volessi modificare gli attributi di un istogramma già esistente, rendendoli uguali a quelli di un altro istogramma?

Esempio:

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

Vogliamo gestire questa operazione con l'operatore =

Cosa significa uguagliare un istogramma ad un altro?

Se per noi la risposta è intuitiva, per il calcolatore non lo è affatto, perché il tipo `istogramma` non è un *default* del C++, ma una classe che abbiamo inventato noi

È necessario implementare un metodo apposito che gestisca correttamente la copia di tutti i valori degli attributi di un oggetto istogramma in un altro oggetto istogramma

L'operator=

Prototipo: **file .h**

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

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

Implementazione: **file .cc**

```
//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;  
}
```

L'output è una *reference* a un oggetto istogramma

`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 fare:

oggetto a = b = c = d;

Utilizzo nel main

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

Costruisco due
oggetti istogramma

```
a = b;
```

Viene chiamato l'`operator=()`
dell'oggetto `a` \rightarrow l'oggetto `a` viene così
modificato ed uguagliato all'oggetto `b`

```
istogramma c;  
c = a = b;
```

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

Overloading di operatori

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

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

- **Es.:** per il calcolatore è diverso eseguire una divisione tra `int` rispetto ad una divisione tra `double` quando eseguiamo la somma fra `int`

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 la creazione di classi

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

Overloading di operatori

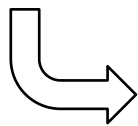
- Quando realizziamo l'overloading di un operatore di fatto stiamo implementando un nuovo metodo della classe
- Al posto di inventare un nome per questo nuovo metodo della classe ricicliamo uno dei simboli già usati in C++ per le operazioni sui tipi standard

Esempio:

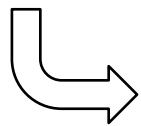
- Per sommare due oggetti istogramma potrei implementare un metodo `Somma`, oppure, equivalentemente, realizzare l'overloading dell'`operator+`

Prototipi: **file .h**

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



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



Utilizzo nel **main**

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

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

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

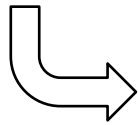
Overloading di operatori

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`

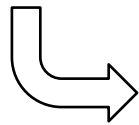
Utilizzo nel `main`

```
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);
```

In output restituisce un oggetto di tipo `istogramma` (che conterrà la somma dei due istogrammi)

Overloading operator+ (.cc)

Nel main:

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

HistoB corrisponde a histo2 nel .cc

//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;
```

```
}
```

HistoA è l'oggetto a cui viene applicato l'operator+



Nel .cc, le variabili Nbin_p, min_p, max_p ... e tutti gli altri attributi contengono i valori dei parametri di HistoA

- Verifica che i due istogrammi abbiano lo stesso binning (altrimenti la somma non è ben definita)
- Prima dell'operazione +=, SumHisto.binContent_p[i] è inizializzato ai conteggi di histo2 (cioè HistoB)
- binContent_p[i] contiene i conteggi nei bin di HistoA

Restituisco l'oggetto istogramma (non la sua *reference*) perché l'oggetto SumHisto è definito solo nel .cc e verrà distrutto dopo essere stato copiato in SumH (che è definito nel main).

Overloading di funzioni

Analogamente posso realizzare **l'overloading delle funzioni**, in modo che la stessa funzione agisca su differenti tipi o anche su un numero diverso di argomenti

- Sarà il compilatore a scegliere a quale funzione far riferimento in base agli argomenti che avremo passato in fase di scrittura del codice

Esempio: la **funzione raddoppia**

```
#include <iostream>
```

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

```
int main() {  
    int A=2;  
    int 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** se cerco di utilizzarla con un `float`

Overloading di funzioni

Posso re-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;  
    int B=2.22;  
    std::cout << raddoppia(A) << std::endl;  
    std::cout << raddoppia(B) << std::endl;  
    return 0;  
}
```

L'**OVERLOADING** consiste nell'avere una funzione con lo stesso nome, ma che agisce su argomenti di diverso tipo

Overloading dei costruttori

Nella prima parte della lezione abbiamo visto che possono coesistere diversi tipi di costruttore della classe istogramma: sono tutti metodi che hanno lo stesso nome (istogramma), ma parametri di input di tipi differenti:

```
//default constructor  
istogramma();  
  
//costruttore  
istogramma(int Nbin, float min, float max);  
  
//copy constructor  
istogramma(const istogramma& original);
```

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

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

Numero di bin

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

Esercizi

- **Esercizio 1:** Scrivete la classe dei numeri complessi utilizzando l'header file (file `.h`) mostrato nella slide seguente e scaricabile dalla pagina e-learning di questa lezione
 - Implementate i metodi:
 - `Re()` → parte reale
 - `Im()` → parte immaginaria
 - `Mod()` → modulo
 - `Rho()` e `Theta()` → descrizione polare
 - `Print()` → `Re + i Im`
 - Implementate 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 (); // distruttore

    // 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);

    // object methods
    double Re() const;
    double Im() const;
    double Mod() const;
    double Theta() const;
    void Print() const;

private:
    // attributes
    double re_p;
    double im_p;
};
#endif
```

Esercizi

- **Esercizio 2:** Scrivete la classe *data del calendario*, utilizzando l'header file (`data.h`) allegato alla pagina e-learning di questa lezione

Oltre ai costruttori e agli 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: ad esempio 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 25/12/2017 sarà un lunedì)

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;

    // metodi gia' implementati nel file .h
    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
```