



Corso di Laboratorio 2 – Programmazione C++

Silvia Arcelli

27 Ottobre 2014

AVVISO

- **Nuovo calendario** turni di laboratorio :

I prova: 11, 12, 13 Novembre 2014 ore 9-12

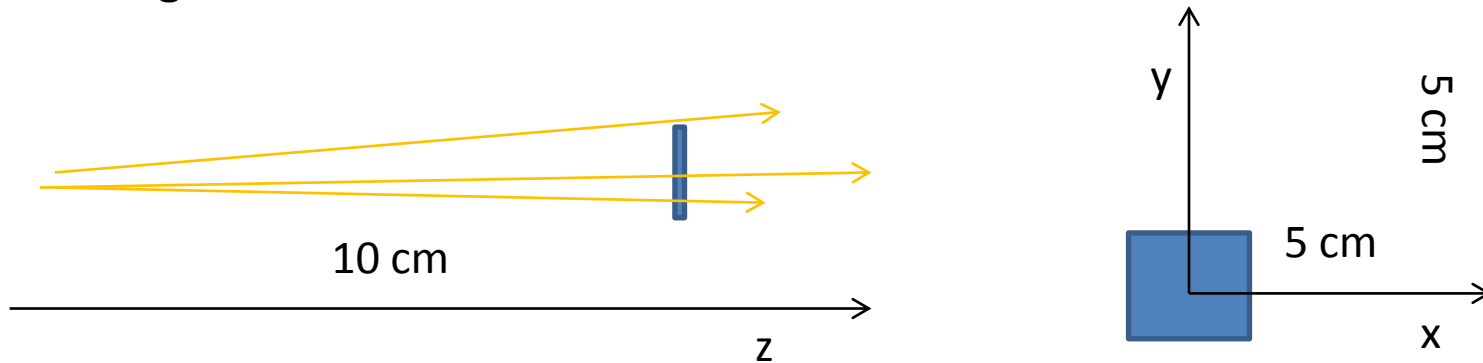
II prova: 18, 19, 20 Novembre 2014 ore 9-12

III prova: 25, 26, 27 Novembre 2014 ore 9-12

IV prova: 2, 3, 4 Dicembre 2014 ore 9-12

ESERCIZIO

- In un esperimento di fisica si misurano le posizioni di impatto x e y di particelle che incidono su un piano di rivelatore posto a una distanza fissa $L=10$ m dalla sorgente. Il piano, di dimensione 5 cm $\times$ 5 cm, è una matrice di rivelatori a silicio con 100 canali nella direzione x e 100 canali nella direzione y , e il suo centro geometrico passa per l'asse z , su cui giace anche la sorgente



- Le particelle sono emesse in un piccolo angolo solido e percorrono il tragitto dalla sorgente al rivelatore seguendo una traiettoria rettilinea a velocità costante.

ESERCIZIO

- Se una particella incide su un determinato canale, alla particella si assegna come posizione quella del centro geometrico del canale, e come incertezza quella definita dalla deviazione standard di una distribuzione uniforme.
- Si vuole scrivere del software dedicato a contenere l'informazione posizionale delle particella sul rivelatore, per una successiva analisi e memorizzazione. Scrivere una classe che
 - conservi l'informazione della posizione della particella congiuntamente al suo errore , implementando metodi che consentano di accedere a tale informazione
 - Abbia un metodo che consenta di correggere la misura di posizione del singolo canale secondo una dipendenza di tipo lineare $a+bx$ e $a'+b'xy$ per entrambe le coordinate, in cui i coefficienti di calibrazione, 4 per ogni canale, sono estratti da un file esterno. Si vuole conservare l'informazione dei coefficienti per ogni canale .

Overloading degli Operatori

- Nell'ambito della definizione dei metodi di una classe, è anche contemplata la possibilità di fare **l'overloading di operatori** (ad esempio, se voglio sommare due numeri complessi di cui ho una rappresentazione come classe posso ridefinire l'operatore + tra i metodi della classe che descrive il numero complesso).
 - L'overloading di un operatore deve soddisfare **un opportuno insieme di requisiti**:
 - Si può solamente eseguire l'overloading di operatori per cui **esiste già un simbolo** nel linguaggio.
 - **Non è possibile** modificare la **precedenza** di un operatore e non è possibile modificarne **l'arietà** (unario/binario) o **l'associatività** (a destra/a sinistra),
-

Overloading degli Operatori

- Non e` concessa la possibilita` di eseguire l'overloading di alcuni operatori, ad esempio
 - l'operatore ternario (? :),
 - l'operatore sizeof ,
 - l'operatore di accesso a un membro, (. , *)
 - l'operatore di risoluzione di ambito (::)
 - In generale è possibile ridefinire un operatore sia come **funzione globale** che **come funzione membro**, tuttavia alcuni operatori devono sempre essere implementati come funzioni membro :
 - operatore di assegnamento (=),
 - operatore di sottoscrizione ([])
 - l'operatore freccia (->).
-

Overloading degli Operatori

- Alcuni operatori sono automaticamente ridefiniti (ad es. l'operatore di assegnamento =)
 - E' possibile ridefinire l'operatore chiamata di funzione `()` , molto usato per rappresentare una operazione "dominante" che viene fatta su oggetti di una certa classe
 - Non c'e` alcuna restrizione riguardo il contenuto del corpo di un operatore: un operatore altro non e` che un tipo particolare di funzione e tutto cio` che puo` essere fatto in una funzione puo` essere fatto anche in un operatore. Buona norma è tuttavia **rispettare la semantica originaria** dell'operatore.
-

Overloading degli Operatori

La sintassi per la definizione degli operatori come funzioni membro è:

```
< ReturnType > ClassName::operator@( < ArgumentList > ) { < Body > }
```

- La keyword operator, seguita dal simbolo dell'operatore (@), è **obbligatoria**.
 - Il tipo di ritorno è **arbitrario** (dipende da cosa fa e come si ridefinisce l'operatore)
 - Come argomenti del metodo si possono inserire tanti argomenti quanti sono gli operandi corrispondenti all'operatore, **meno uno** (che è l'oggetto corrente a cui ci si può riferire con **this**). Ad esempio, l'operatore somma come funzione membro ha un solo argomento. Un operatore unario come l'incremento postfisso o prefisso ha 0 argomenti.
-

Overloading degli Operatori

Esempio di overloading di operatori (- unario, somma binaria, assegnamento) come funzioni membro della classe Complex:

```
class Complex {  
public:  
    Complex(float re, float im);  
    Complex operator-() const; // - unario  
    Complex operator+(const Complex& B) const;  
    Complex & operator=(const Complex& B);  
private:  
    float Re;  
    float Im;  
};
```

```
Complex::Complex(float re, float im = 0.0)  
    { Re = re; Im = im; }  
Complex Complex::operator-() const  
    { return Complex(-Re, -Im); }  
Complex Complex::operator+(const Complex& B) const  
    { return Complex(Re+B.Re, Im+B.Im); }  
Complex& Complex::operator=(const Complex& B)  
    { Re = B.Re; Im = B.Im; return *this; }
```

Overloading degli Operatori

Nell'esempio appena mostrato si ha che:

- E' stato fatto l'overload del -(meno) unario, e il compilatore capisce che si tratta del meno unario dalla lista di argomenti vuota. Per operatori unari, è in generale raccomandato implementare l'overload come funzione membro
- Il + (o -) binario invece, come funzione membro, deve avere un argomento

```
Complex A, B;  
/* ... */  
Complex C = A + B; // equivalente a Complex C(A.operator+( B));
```

- Infine viene ridefinito l'operatore di assegnamento che ritorna un riferimento (oltre ad avere il parametro passato per riferimento), in modo da poter concatenare piu' assegnamenti evitando la creazione di temporanei (i reference non richiedono nuova memoria).
-

Operatore di Assegnamento

L'operatore di **assegnamento**:

```
MyClass& MyClass::operator=(const MyClass & B){/*...*/}
```

ha una particolarità che lo rende simile al costruttore:

- se in una classe non ne viene definito uno esplicitamente il compilatore ne fornisce uno in modo **automatico**.
 - Così come nel caso del costruttore di copia, è necessario **definire esplicitamente l'operatore** di assegnamento ogni qual volta la classe contenga **puntatori**, onde evitare condivisioni di memoria.
-

Operatore di Assegnamento

In particolare, quando si ridefinisce l'operatore di assegnamento è bene tenere presente queste due “regole”:

- Evitare che un oggetto assegni a se stesso (se una operazione di assegnamento comporta la deallocazione di risorse, e` facile che un autoassegnamento porti l'oggetto in uno stato inconsistente)
 - Ritornare un reference a `*this`.
-

Operatore di Assegnamento

Esempio di operatore di assegnamento “fragile” rispetto ad autoassegnamenti (quando scrivo A=A, per intenderci):

```
class MyClass {  
public:  
    MyClass(){/*---*/}  
    MyClass& operator=(const MyClass& value);  
private:  
    char* Str;  
};  
    MyClass& MyClass::operator=(const MyClass& value) {  
delete[] Str;  
Str = new char[strlen(value.Str)+1];  
strcpy(Str, value.Str);  
/*--*/  
}
```

Operatore di Assegnamento

- Ritornare un riferimento a `*this` e non a “value” è importante perché il valore generalmente restituito è un reference a non const (altrimenti dovete definire anche il valore di ritorno come const, ma questo poi vi porrà altre limitazioni....)
- Eliminare il const dal parametro non è una soluzione poiché impedirebbe l'assegnamento a istanze costanti.
- In generale dunque la struttura di un buon operatore di assegnamento è:

```
MyClass& MyClass::operator=(const MyClass& value) {  
    if (this==&value) return *this; //protegge vs autoassegazioni  
    /* ... */  
    return *this; }
```

Overloading degli Operatori

- Per eseguire l'overloading di un operatore come funzione globale si utilizza una sintassi molto simile:

```
< ReturnType > operator@( < ArgumentList > ) { < Body > }
```

- In questo caso, ovviamente, come argomenti del metodo si hanno tanti argomenti quanti sono gli operandi corrispondenti all'operatore. La somma definita come funzione globale ha in questo caso due argomenti. L'ordine degli argomenti riflette l'ordine degli operandi nell'operatore che si ridefinisce.
-

Overloading degli Operatori-Funzioni Globali

Esempio di overloading di un operatore (somma) come funzione globale:

```
class Complex {  
public:  
/*---*/  
friend Complex operator+(const Complex& , const Complex& );  
private:  
float Re;  
float Im;  
};  
Complex operator+(const Complex& A, const Complex& B) {  
Complex Result;  
Result.Re = A.Re + B.Re;  
Result.Im = A.Im + B.Im;  
return Result;  
}
```

Dichiarazioni Friend

- La dichiarazione friend consente a una funzione non membro l'accesso diretto ai membri (attributi e/o metodi) privati di una classe (overloading operatori di I/O, ma anche in generale in situazioni di cooperazione fra classi)
- La keyword friend puo` essere applicata anche a un identificatore di classe, abilitando cosi` una classe intera. **non ha importanza** la sezione (pubblica, protetta o privata) in cui tale dichiarazione e` fatta.

```
class MyClass {  
    /* ... */  
    friend class AnotherClass;  
};
```

- in tal modo qualsiasi membro di AnotherClass puo` accedere agli attributi/metodi privati di MyClass. Da usare con cautela, è una “violazione” dell'incapsulamento.
-

Overloading degli Operatori-Funzioni Globali

Definito l'operatore, e' possibile utilizzarlo secondo la sintassi riservata agli operatori:

```
Complex A, B;  
/* ... */  
Complex C = A + B;
```

Notate che qui si fa uso del costruttore di copia di Complex (esplicitamente definito o fornito di default dal compilatore) e non l'operatore di assegnamento. Il codice appena mostrato è equivalente a:

```
Complex A, B;  
/* ... */  
Complex C(operator+(A, B));
```

Overloading degli Operatori

Sostanzialmente, non c'è differenza tra un operatore definito globalmente e uno con le stesse funzionalità definito come funzione membro:

- nel primo caso l'operatore, se necessario, **viene dichiarato friend** di tutte le classi cui appartengono i suoi argomenti (deve poter operare su tutti i membri dell'oggetto);
- nel caso di una funzione membro, **il primo argomento è sempre l'istanza corrente** e non è esplicito. Per l'eventuale secondo argomento dell'operatore, può essere necessario dichiarare la classe che definisce l'operatore come membro proprio **friend** nella classe cui appartiene il secondo argomento.

Per il resto non ci sono differenze per il compilatore, i due approcci sono ugualmente efficienti.

Overloading degli Operatori

Non sempre è possibile utilizzare una funzione membro, ad esempio se si vuole implementare il flusso su stream della propria classe è necessario ricorrere ad una funzione globale, perché l'operando sinistro **non è del tipo della classe**.

```
class Complex {  
public:  
    Complex(float a, float b):Re(a),Im(b){}  
private:  
    float Re, Im;  
    friend ostream& operator<<(ostream& o, Complex& C);  
};  
ostream& operator<<(ostream& o, Complex& C)  
{ o << C.Re << " + i " << C.Im;  
  return o;  
}
```

```
int main (){  
    Complex A(1,3);  
    /* ... */  
    cout << A << endl;  
    return 0;  
}
```

Operatore << come funzione globale + dichiarazione friend