

Corso di ***Grafica Computazionale***

Note di C++

Docente:
Massimiliano Corsini

Laurea Specialistica in Ing. Informatica

Facoltà di Ingegneria

Università degli Studi di Siena



Note Iniziali

- Il C++ non ha il concetto dei packages
- In C++ si deve dire esplicitamente in quali file il compilatore deve cercare le definizioni che gli servono:
 - `#include<...>`
- Possono esistere funzioni globali (cioè che non fanno parte di nessuna classe).
- Il C++ non è banalmente “il C con le classi”(!!!)
- Il C++ è un linguaggio fortemente tipizzato !!!



Note Iniziali

- I sorgenti C++ sono splittati in
 - **header (.h)**: che contengono le *dichiarazioni* delle classi (e quindi la loro interfaccia)
 - **program files (.cpp)**: che contengono le *definizioni* dei vari metodi delle classi (e quindi le loro implementazioni)



Classi

```
[foo.h]
class Foo          // dichiarazione della classe Foo
{
public:            // sezione pubblica
    Foo();        // dichiarazione del costruttore

private:         // inizio sezione 'privata'
    int m_num;   // dichiaraz. variabile privata di
                // tipo intero
};
```

```
[foo.cpp]
#include "foo.h"

Foo::Foo()        // definizione del costruttore
{
    m_num = 5;   // inizializzazione della variabile
}
```



Classi

[foo.h]

```
class Foo {
public:
    Foo();                // costruttore
    ~Foo();               // distruttore
    int myMethod(int a, int b); // un metodo pubblico
}; // notate il ; alla fine della dichiarazione !!
```

[foo.cpp]

```
#include "Foo.H"
#include <iostream>

Foo::Foo() // scope operator
{
    int a = myMethod(5,2);
    std::cout << "a = " << a << std::endl; // stampo il risultato sullo
                                           // stream di output
}

Foo::~~Foo()
{
    std::cout << "Il distruttore dovrebbe pulire la memoria..." << std::endl;
}

int Foo::myMethod(int a, int b)
{
    return a+b;
}
```



Note sulla Sintassi

- Notare l'uso dello ***scope operator*** `::`
 - Serve a dire a quale classe appartiene il metodo che sto definendo
- `std::` è il ***namespace*** standard
- Attenzione al `;` dopo la dichiarazione di classe!
 - Altrimenti può capitare che l'errore venga segnalato molto dopo...



Public, Private e Protected

- I membri di una classe possono essere dichiarati **public**, **private** e **protected** a seconda dell'accesso che si vuole permettere
- **Public** → possono essere acceduti liberamente
- **Protected** → possono essere acceduti solo da funzioni membro della classe di appartenenza e dalle funzioni membro delle classi derivate
- **Private** → possono essere acceduti solo da funzioni membro della classe di appartenenza



Costruttori ed Inizializzazione

- L'inizializzazione delle variabili membro di una classe viene fatta nel costruttore della classe (come in Java)
- A differenza di Java i membri di una classe possono essere inizializzati prima della chiamata del costruttore.
 - Meccanismo della lista di inizializzatori
 - Paradigma RAI (Resource Acquisition Is Initialization)



Lista Inizializzatori

```
class Foo
{
public:
    Foo();
protected:
    int m_a, m_b;
private:
    double m_x, m_y;
};
=====
#include "foo.h"
#include <iostream>
using namespace std; // utilizzo il namespace standard (std)

// inizializzazione con initializer list
Foo::Foo() : m_a(1), m_b(4), m_x(3.14), m_y(2.718)
{
    std::cout << "Inizializzazione terminata." << std::endl;
}

// inizializzazione senza initializer list
Foo::Foo()
{
    m_a = 1; m_b = 4; m_x = 3.14; m_y = 2.718;
    std::cout << "Inizializzazione terminata." << std::endl;
}
```



Distruttori

- In uno dei precedenti esempi oltre al costruttore era presente anche un metodo `Foo::~~Foo`
- Questo metodo è chiamato distruttore non ha parametri e viene invocato quando un'istanza di una classe viene distrutta.
- Come il costruttore non ritorna nulla
- Il distruttore può essere invocato automaticamente oppure esplicitamente



Overloading

- È possibile avere più di una funzione con lo stesso nome
- Il C++ risolve il conflitto su quale funzione chiamare usando il tipo dei parametri con i quali la funzione viene chiamata
- Bisogna fare attenzione ai cast impliciti che possono far scegliere al compilatore un metodo al posto di un altro
- In certi casi la chiamata rimane ambigua e il programmatore deve forzare la scelta con un cast esplicito



Overloading e Cast Impliciti

```
include <iostream>
using namespace std;

void Foo::print(int a)
{
    cout << "int a = " << a << endl;
}

void Foo::print(double a)
{
    cout << "double a = " << a << endl;
}
```

- **“foo” è un’istanza della classe “Foo”**
 - `foo.print(5);`
output: int a = 5
 - `foo.print(5.5)`
output: double a = 5.5



Parametri di Default

```
class Foo
{
public:
    Foo();
    void setValues(int a, int b=5)

    •protected:
        int m_a, m_b;
};

void Foo::setValues(int a, int b)
{
    m_a=a;
    m_b=b;
}
```

- **“foo” è un’istanza della classe “Foo”**
foo.setValues(4);
- **È equivalente a scrivere:**
foo.setValues(4,5);



Parametri di Default

- Ricordatevi sempre che i parametri senza default DEVONO precedere tutti i parametri con default
- Non si può saltare parametri nel mezzo di una chiamata
- I parametri di default si specificano nel .h e non nel .cpp



Ereditarietà

```
[foo.h]
class A
{public:
    A(int something);
};

class B : public A
{public:
    B(int something);
};

[foo.cpp]
#include "foo.h"

A::A(int something)
{}

B::B(int something) : A(something)
{}

```



Ereditarietà

- Simile a java
- Il costruttore della classe derivata prende i parametri per costruire la classe base

```
B::B(int something) : A(something)
```
- Esiste anche l'ereditarietà multipla
 - Class B : public A , public C
- È buona norma non abusare dell'ereditarietà multipla



Polimorfismo

- Concettualmente *polimorfismo* significa mandare agli oggetti lo stesso messaggio ed ottenere comportamenti diversi a seconda del contesto di utilizzo
- Un metodo membro si dice **overriden** se ha lo stesso nome e gli stessi argomenti ma appartiene a classi diverse
- Quando la chiamata viene risolta in fase di compilazione siamo in presenza dell'**early binding**
- Esempio: (B classe derivata di A)
 a.display(); // early binding
 b.display(); // early binding



Polimorfismo

- Se però nella classe A la funzione `display()` fosse dichiarata con lo specificatore *virtual* (*funzione virtuale*) allora la scelta della chiamata sarebbe rinviata in fase di esecuzione (***late binding***) realizzando il polimorfismo vero e proprio dal punto di vista concettuale (l'oggetto polimorfa il suo comportamento a seconda del contesto)



Esempio di Polimorfismo

- Shape è la classe base (interfaccia) di Triangle, Quadrilateral, Hexagon
- ```
Shape *shape = myshape;
shape->draw(); // il compilatore, a run-time
 // risolve la chiamata
```



# Funzioni Virtuali

```
class A
{
public:
 A();
 virtual ~A();
 virtual void foo();
};
```

```
class B : public A
{
public:
 B();
 virtual ~B();
 virtual void foo();
};_
```



# Funzioni Virtuali

- Possibilità di definire classi non istanziabili (che quindi servono solo a definire un'interfaccia) con funzioni pure virtual (equivalente alla keyword 'abstract' in Java)

```
class FooInterface
{
public:
 // metodo "pure virtual"
 virtual int abstractMethod() = 0;
}
```

- Grazie allo scope operator si può accedere ai membri overridden come si vuole



# Chiamate a Funzioni Virtuali

```
#include "foo.h"
#include <iostream>
using namespace std;

A::foo()
{
 cout << "A::foo() called" << endl;
}

B::foo()
{
 cout << "B::foo() called" << endl;
 A::foo();
}
```

- **“b” è un’istanza della classe “B”**

```
b.foo();
```

- **Output**

```
B::foo() called
A::foo() called
```



# Puntatori e Memoria

- Alcune ovvietà:
  - La memoria è organizzata in celle, ognuna delle quali è associata ad un numero unico detto *indirizzo*
  - Ogni variabile è memorizzata in un certo numero di celle
  - Un puntatore è un indirizzo di memoria
  - Un puntatore ad una variabile è l'indirizzo di memoria della prima cella in cui la variabile è memorizzata
  - I puntatori sono variabili



# Dichiarazione Puntatori

- Come si dichiara un puntatore?
  - Operatore \* tra il tipo e la variabile
  - Esempio. `int* myIntegerPointer;`
  - È un puntatore ad un intero, ossia una variabile che contiene l'indirizzo di una variabile di tipo intero





# Puntatori e Memoria

- Come si fa a far puntare un puntatore a qualcosa?
- Ossia, come si fa a far sì che una variabile puntatore contenga come valore l'indirizzo di un'altra variabile?
- Operatore &:

```
int* myIntegerPointer;
int myInteger = 1000;
myIntegerPointer = &myInteger;
```



# Puntatori e Memoria

- Come si fa a modificare quel che punta un puntatore?
  - Come si fa a modificare la variabile il cui indirizzo è memorizzato in un certo puntatore (e non quindi il puntatore stesso)?
  - Come si fa a dereferenziare un puntatore?
- Ancora \* (inteso come operatore di dereferenziazione)

```
int* myIntegerPointer;
int myInteger = 1000;
myIntegerPointer = &myInteger;
```

`myIntegerPointer` → indirizzo di memoria di `myInteger`

`*myIntegerPointer` → l'intero all'indirizzo di memoria dato da `myIntegerPointer`



# Puntatori e Memoria

```
#include <iostream>
using namespace std;

int main(int argc, char **argv)
{
 int myInteger = 1000;
 int *myIntegerPointer = &myInteger;

 // stampa il valore di myInteger
 cout << myInteger << endl;

 // dereferenzia il puntatore ed aggiunge 5
 // all'intero puntato
 *myIntegerPointer += 5;

 // stampa il valore dell'intero dopo avere
 // modificato il suo contenuto tramite il puntatore
 cout << myInteger << endl;
}
```



# Altri Esempi

- Cosa succede se si dereferenzia un puntatore e si memorizza in un'altra variabile?

```
int myInteger = 1000;
int* myIntegerPointer = &myInteger;
int mySecondInteger = *myIntegerPointer;
```

- Cosa succede se cambio il valore di myInteger?  
Cambia anche mySecondInteger?



# Altri Esempi

```
#include <iostream>
using namespace std;

int main(int argc, char **argv)
{
 int myInteger = 1000;
 int *myIntegerPointer = &myInteger;
 int mySecondInteger = *myIntegerPointer;

 // stampo il valore di myInteger
 cout << myInteger << endl;

 // aggiungo 5 tramite dereferenziazione a myInteger
 *myIntegerPointer += 5;

 // stampo di nuovo il valore di myInteger
 cout << myInteger << endl;

 // cosa è successo a mySecondInteger?
 cout << mySecondInteger << endl;
}
```



## Puntatori alla stessa variabile

---

- Si può avere più puntatori alla stessa variabile.
  - Cambiando il valore della variabile memorizzata a quel indirizzo, ovviamente il cambiamento è *visto* da tutti i puntatori a quella variabile



# Puntatori a Puntatori

- Siccome i puntatori sono variabili se ne può avere e memorizzare l'indirizzo

```
int myInteger = 1000;
int* myIntegerPointer = &myInteger;
int** myIntegerPointerPointer;

myIntegerPointerPointer = &myIntegerPointer;
```

- `(*myIntegerPointerPointer) == myIntegerPointer` → l'indirizzo di memoria di `myInteger`
- `(**myIntegerPointerPointer) ==` quel che è memorizzato all'indirizzo `myIntegerPointer` → `myInteger`



# Puntatori ad Oggetti

- [Foo.H]  
class Foo {  
public:  
    Foo();                    // costruttore  
    Foo(int a, int b);      // altro costruttore  
    ~Foo();                  // distruttore  
  
    void bar();              // random method  
    int blah;  
};
- `Foo* myFooInstance = new Foo(0, 0)`





# Puntatori ad Oggetti

- `Foo* myFooInstance = new Foo(0, 0)`
- Per usare l'oggetto creato (e.g. accedere ai suoi metodi e membri pubblici) occorre dereferenziarlo  
`(*myFooInstance).bar();`
  - Oppure usando l'operatore freccia `->`  
`myFooInstance->bar();`



# Istanze di Oggetti

---

- In C++ è possibile dichiarare (ed ottenere) oggetti senza fare `new` esplicite o usare puntatori:
- `Foo myFooInstance(0,0);` dichiara una variabile di tipo `foo` e chiama il costruttore;
- Se si voleva usare il costruttore di default:

```
Foo myFooInstance;
```

oppure equivalentemente:

```
Foo myFooInstance();
```



# Istanze di Oggetti

- Per accedere a membri e funzioni pubbliche di un istanza si fa come in Java.
- ```
myFooInstance.bar();  
myFooInstance.blah = 5;
```
- Istanze di oggetti possono essere create anche senza essere associate esplicitamente ad una variabile:
- ```
// supponiamo che Bar abbia il metodo setAFoo(Foo foo) ...
Bar bar;
bar.setAFoo(Foo(5,3)); // l'istanza di Foo viene creata e
 // passata alla funzione
```



# Istanze di Oggetti

- Come i puntatori, le istanze possono essere variabili locali o variabili membri di una classe;
- Il costruttore può essere chiamato nella lista di inizializzatori del costruttore della classe



# Istanze di Oggetti

```
[Bar.H]
```

```
#include "Foo.H" // da qualche parte avrò la dichiarazione di Foo
```

```
class Bar {
```

```
public:
```

```
 Bar(int a, int b);
```

```
private:
```

```
 Foo m_foo; // istanza di Foo
```

```
};
```

```
[Bar.C]
```

```
Bar::Bar(int a, int b) : m_foo(a,b) // Foo::Foo(int,int)
```

```
 // m_foo
```

```
{
```

```
 Foo fooLocal; // questa istanza di Foo è una variabile locale
```

```
 // ...
```

```
}
```



# References

- Supponiamo di voler riferire un'istanza di un oggetto con più di un nome.
- Una soluzione sono i puntatori
- Una seconda soluzione, più sicura, è di usare i references



# References

```
[main.C]
#include <iostream>
using namespace std;

int main(int argc, char **argv)
{
 int foo = 10;
 int& bar = foo;

 bar += 10;
 cout << "foo is: " << foo << endl;
 cout << "bar is: " << bar << endl;

 foo = 5;
 cout << "foo is: " << foo << endl;
 cout << "bar is: " << bar << endl;
}
```



# References

- References assomigliano ai puntatori ma sono diversi dal punto di vista ***semantico***
- Possono essere assegnati SOLO alla creazione
- Non possono essere NULL
- Si accede al loro contenuto senza operatori di dereferenziazione
- Possono essere utilizzati liberamente nel passaggio dei parametri





# References e Classi

- Siccome i references possono essere assegnati solo alla creazione, references che sono membri di una classe **devono** essere assegnati nella lista di inizializzatori del costruttore della classe

- [bar.h]

```
class Foo;
```

```
class Bar
```

```
{
```

```
private:
```

```
 Foo & m_fooBar; // a reference to a Bar
```

```
public:
```

```
 Bar(Foo & fooToStore) :
```

```
 m_fooBar(fooToStore) {}
```

```
};
```



# Da puntatori a variabili

```
// fooPointer è un puntatore a Foo
Foo* fooPointer = new Foo(0, 0);

// il puntatore viene dereferenziato ed una copia
// dell'istanza puntata assegnata a myFooInstance
Foo myFooInstance = *fooPointer;
```

**Modificando myFooInstance NON si modifica anche  
l'oggetto puntato da fooPointer(!!)**



# Copiare un Oggetto

- Di default, un'istanza di una classe può essere copiata
- L'operazione (di default) copia di una classe copia ogni membro della classe
- Esempio:

```
Date d = today; // inizializzazione
 // tramite copia
```
- Se questo non è il comportamento desiderato si può ricorrere alla definizione di un ***costruttore di copia***.



# Costruttore di Copia

- Particolare tipo di costruttore utilizzato quando si vuole inizializzare un oggetto con un altro dello stesso tipo.
- Spesso usato nel passaggio di parametri.
- ```
class Foo {  
    // copy constructor  
    Foo(const Foo&);  
};
```



Costruttore di Copia (esempio)

```
void func()  
{  
    Table t1;  
    Table t2 = t1; // inizializzazione tramite copia  
    Table t3;  
  
    t3 = t2;      // copia tramite assegnazione  
}
```

Il costruttore di Table viene chiamato 2 volte (per t1 e per t3). Poichè t2 è inizializzato tramite copia il suo costruttore non viene chiamato. Al suo posto viene chiamato il costruttore di copia.

Distinguate tra assegnazione e copia(!)



Costruttore di Copia (esempio)

```
class Table
{
    // ...
    Table (const Table&);           // copy constructor
    Table& operator=(const Table&); // copy assignment
}
```

```
Table::Table(const Table& t)
{
    p = new Name[sz=t.sz];
    for (int i=0; i < sz; i++) p[i] = t.p[i];
}
```

```
Table& Table::operator=(const Table& t)
{
    if (this != &t) { // check auto-assegnamento
        delete [] p;
        p = new Name[sz=t.sz];
        for (int i=0; i < sz; i++) p[i] = t.p[i];
    }
    return *this;
}
```



Costruttore di Copia

- Il copy constructor e l'assegnazione possono e in certi casi devono differire come abbiamo visto.
- Infatti, il copy constructor deve inizializzare la memoria non inizializzata, mentre l'operatore di assegnazione deve tener conto di avere a che fare con un oggetto ben costruito.



Occhio alla memoria(!)

- Considerando l'esempio precedente, *ipotizzando che gli elementi delle tabelle siano allocati nel costruttore tramite delle new*, quello che accade è che il distruttore viene chiamato 3 volte con conseguenze potenzialmente disastrose.
- Inoltre l'assegnazione $t3 = t2$ causa un memory leak.



Memory Management

- Due grandi categorie di storage:
- **Local**, memoria valida solo all'interno di un certo scope (e.g. dentro il corpo di una funzione), lo stack;
- **Global**, memoria valida per tutta l'esecuzione del programma, lo heap.



Local Storage

- {
 int myInteger; // memoria per un intero allocata
 // ... uso myInteger ...

 Bar bar; // memoria per la classe Bar allocata
 // ... uso bar ...
}
- ‘{’ e ‘}’ sono i delimitatori di un blocco in c++, Non è detto che corrisponda ad una funzione...
- Non possiamo usare bar, o myInteger fuori dallo scope del blocco!



Global Storage

- Per allocare memoria nel global storage (e.g. per avere puntatori ad oggetti la cui validità persista sempre) si usa l'operatore new.



Local e Global Storage (esempio)

```
[main.cpp]
#include "bar.h" // supponiamo di avere una classe Bar

int main(int argc, char *argv[])
{ // puntatore a Bar non inizializzato → 'garbage'
  Bar *p;
  { // creiamo un'istanza della classe Bar
    p = new Bar();
    if (p == 0) {
      // memory allocation failed
      return 1;
    }
    Bar p2; // p2 è un'istanza di Bar
  }
  // Bar è allocata nel global storage, possiamo continuare
  // ad utilizzare l'istanza
  p->myFunction();
  p2.myFunction(); // p2 esiste solo nel blocco di cui sopra
}
```



Operatore *Delete*

- ***In C++ lo sviluppatore deve esplicitamente gestire le risorse allocate.*** La deallocazione della memoria deve essere chiamata ***esplicitamente !!***
- Per cancellare dalla memoria gli oggetti allocati con `new` si deve utilizzare l'operatore `delete`
- Solo oggetti creati con `new` possono essere deallocati con `delete`.



Memory Management

```
[Foo.h]
#include "MyUsefulObject.h"

class Foo {
private:
    MyUsefulObject* m_Obj;
public:
    Foo() {}
    ~Foo() {}
    void funcA() {
        m_Obj = new MyUsefulObject;
    }

    void funcB() {
        // MyUsefulObject is used...
    }

    void funcC() {
        // ...
        delete m_Obj;
    }
};
```



Memory Management

```
main()
{
    Foo myFoo;        // viene create un'istanza locale della
                    // classe Foo

    myFoo.funcA(); // m_Obj viene creato

    // ...
    myFoo.funcB(); // Ok
    // ...
    myFoo.funcB(); // Ok
    // ...

    myFoo.funcC(); // m_Obj viene deallocato
}
```



Memory Management

Facoltà di
Ingegneria

```
main
{
    Foo myFoo;

    //...
    myFoo.funcB(); // errore(!!) MyUsefulObject non è stato allocato

    myFoo.funcA(); // MyUsefulObject è allocato ed assegnato a m_Obj
    myFoo.funcA(); // MyUsefulObject viene allocato di nuovo
                    // (un'altra istanza)
                    // e ri-assegnato al puntatore, si perde traccia
                    // del vecchio MyUsefulObject → MEMORY LEAK(!!)

    //...
    myFoo.funcB();

} // un altro MEMORY LEAK!! MyUsefulObject non viene deallocato.
```




Memory Management

```
[Foo.h]
#include "MyUsefulObject.h"

class Foo {
public:
    Foo() {}
    ~Foo() {}
    void funcA()
    {
        MyUsefulObject myobj = new MyUsefulObject;

        myobj->method1();
        myobj->method2();
        //...
    } // MEMORY LEAK(!!) → abbiamo creato
    // MyUsefulObject ma non lo abbiamo deallocato e
    // lui sopravvive allo scope
};
```



Domande?