

Costruzione di Interfacce Lezione 11 Dal Java al C++

cignoni@iei.pi.cnr.it
<http://vcg.iei.pi.cnr.it/~cignoni>

Caveat

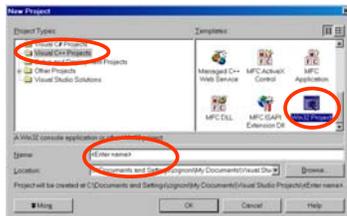
- ❖ Se volete provare quanto vedremo oggi ricordatevi che stiamo facendo programmi command line style e quindi il progetto .net va costruito nel giusto modo.

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

2

Creare il progetto

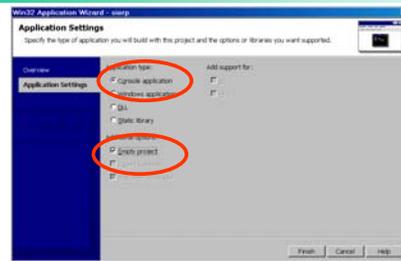


28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

3

Setting progetto



28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

4

Aggiungere un file al progetto



28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

5

Hello World

```
[Hello.java]
package hello; // says that we are part of a package named hello
public class Hello // declare a class called Hello
{
    public static void main(String args[]) // declare the function main
    { // that takes an array of Strings
        System.out.println("Hello world!"); // call the static method
    } // println on the class System.out
} // with the parameter "Hello world!"

[Hello.C]
#include <iostream> // include declarations for the "cout" output stream
using namespace std; // the cout stream is in the std namespace
// this tells the compiler to look in the std
// namespace, you can also write std::cout
int main(int argc, char *argv[]) // declare the function main that
{ // takes an int and an array of strings
    cout << "Hello world!" << endl; // and returns an int as the exit code
} // this inserts "Hello world!"
// output stream // and a newline character into the cout
}
```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

6

Differenze Salienti

- ❖ Esistono funzioni globali (che non fanno parte di alcuna classe): main()
- ❖ 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<... >

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

7

Classi

```
[Foo.java]
public class Foo // declare a class Foo
{
    protected int m_num; // declare an instance variable of type int
    public Foo() // declare and define a constructor for Foo
    {
        m_num = 5; // the constructor initializes the m_num variable
    }
}

[Foo.H]
class Foo // declare a class Foo
{
public: // begin the public section
    Foo(); // declare a constructor for Foo
protected: // begin the protected section
    int m_num; // declare an instance variable of type int
};

[Foo.C]
#include "Foo.H"

Foo::Foo() // definition for Foo's constructor
{
    m_num = 5; // the constructor initializes the m_num
              // instance variable
}

```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

8

Differenze Salienti

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

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

9

```
[Foo.H]
class Foo {
public:
    Foo();
    ~Foo();
    int myMethod(int a, int b);
}; // note the semicolon after the class declaration!

[Foo.C]
#include "Foo.H"
#include <iostream>

Foo::Foo() // scope operator :: helps define constructor for class Foo
{
    cout << "I am a happy constructor that calls myMethod" << endl;
    int a = myMethod(5,2);
    cout << "a = " << a << endl;
}

Foo::~~Foo()
{
    cout << "I am a happy destructor that would do cleanup here." << endl;
}

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

```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

10

Sintassi

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

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

11

Costruttori e liste di inizializzatori

- ❖ Come in Java, l'inizializzazione delle variabili membro di una classe viene fatta nel costruttore della classe.
- ❖ Come in Java il costruttore può avere vari parametri
- ❖ A differenza di Java i membri di una classe possono essere inizializzati prima della chiamata del costruttore.
 - ❖ Meccanismo della lista di inizializzatori

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

12

Inizializer list

```
[Foo.H]
class Foo
{
public:
    Foo();
protected:
    int m_a, m_b;
private:
    double m_x, m_y;
};

[Foo.C] // with initializer list
#include "Foo.H"
#include <iostream>
using namespace std;

Foo::Foo() : m_a(1), m_b(4), m_x(3.14), m_y(2.718)
{
    cout << "The value of a is: " << m_a << endl;
}

// o equivalentemente

Foo::Foo()
{
    m_a = 1; m_b = 4; m_x = 3.14; m_y = 2.718;
    std::cout << "The value of a is: " << m_a << endl;
}
28 Ott 2002      Costruzione di Interfacce - Paolo Cignoni      13
```

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

28 Ott 2002 Costruzione di Interfacce - Paolo Cignoni 14

Overloading

- ❖ Come in java è possibile avere più di una funzione con lo stesso nome
- ❖ C++ usa il tipo di parametri per decidere quale funzione chiamare.
- ❖ Attenzione, ai cast impliciti che il C++ può fare e che potrebbero far scegliere al compilatore un metodo al posto di un altro

28 Ott 2002 Costruzione di Interfacce - Paolo Cignoni 15

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;
}

On an instance "foo" of type "Foo", calling
foo.print(5);
will output
int a = 5
whereas
foo.print(5.5)
will output
double a = 5.5
```

28 Ott 2002 Costruzione di Interfacce - Paolo Cignoni 16

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;
}

if we have an instance "foo" of class "Foo" and we did the
following:
foo.setValues(4);
it would be the same as if we had coded:
foo.setValues(4,5);
```

28 Ott 2002 Costruzione di Interfacce - Paolo Cignoni 17

Parametri di default

- ❖ Nota 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 nell .cpp

28 Ott 2002 Costruzione di Interfacce - Paolo Cignoni 18

Ereditarietà

```
❖ [Foo.H]
class A
{public:
  A(int something);
};

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

[Foo.C]
#include "Foo.H"
#include <iostream>
using namespace std;

A::A(int something)
{
  cout << "Something = " << something << endl;
}

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

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

19

Ereditarietà

- ❖ Simile a java
- ❖ Normalmente è sempre public
- ❖ Nota importante, per poter passare parametri al costruttore della classe da cui si deriva si deve utilizzare la sintassi della initializer list
`B::B(int something) : A(something)`
- ❖ Esiste anche l'ereditarietà multipla
`class B : public A, public C`

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

20

Funzioni Virtuali

```
public void someMethod() {
  Object obj = new String("Hello");
  String output = obj.toString(); // calls String.toString(),
  // not Object.toString()
}
```

- ❖ Polimorfismo, java sa che obj è in realtà una stringa e chiama il metodo giusto
- ❖ Il tipo dell'oggetto va controllato a runtime.
- ❖ Leggero overhead a runtime, contrario alla filosofia del c++,
- ❖ In C++ di default non funziona come in java
- ❖ Se quando si subclassa, si fa override di una funzione viene chiamata quella tipo della variabile usata...

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

21

Funzioni Virtuali

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

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

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

22

Sintassi Virtual

- ❖ Accesso a membri overridden: si usa lo scope operator ::
- ❖ 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 Foo
{
public:
  virtual int abstractMethod() = 0; // The "virtual" and "= 0"
  // are the key parts here.
}
```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

23

Virtual

```
#include "Foo.H"
#include <iostream>
using namespace std;
```

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

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

So, if we have an instance "b" of class "B", calling

`b.foo();`

will output

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

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

24

Puntatori e Memoria

- ❖ In c++ le variabili si dichiarano (e si inizializzano) come in java

```
int number=0;
```
- ❖ In realtà per tipi non primitivi le cose sono diverse...
- ❖ 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

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

25

Puntatori e Memoria

- ❖ Come si dichiara un puntatore?
 - ❖ Operatore * tra il tipo e la variabile
 - ❖ E.g.

```
int* myIntegerPointer;
```
 - ❖ È un puntatore ad intero, e.g. una variabile che contiene l'indirizzo di una variabile di tipo intero.

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

26

Puntatori e Memoria

- ❖ Come si fa a far puntare un puntatore a qualcosa?
- ❖ (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;
```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

27

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 lo * (inteso come operatore di dereferenziazione)

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

```
myIntegerPointer means "the memory address of myInteger."  
*myIntegerPointer means "the integer at memory address  
myIntegerPointer."
```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

28

Puntatori e Memoria

```
#include <iostream>  
  
using namespace std;  
  
int main(int argc, char **argv  
int myInteger = 1000;  
int *myIntegerPointer = &m  
  
// print the value of the integer before changing it  
cout << myInteger << endl;  
  
// dereference pointer and add 5 to the integer it points to  
*myIntegerPointer += 5;  
  
// print the value of the integer after  
// changing it through the pointer  
cout << myInteger << endl;  
}
```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

29

Altri esempi

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

```
int myInteger = 1000; // set up an integer with value 1000  
int* myIntegerPointer = &myInteger; // get a pointer to it  
int mySecondInteger = *myIntegerPointer; // now, create a second integer  
// whose value is that of the integer  
// pointed to by the above pointer
```

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

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

30

```

#include <iostream>
using namespace std;

int main(int argc, char **argv) {

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

    // declare another integer whose value is the same as the integer
    // at memory address <myIntegerPointer>
    int mySecondInteger = *myIntegerPointer;

    // print the value of the first integer before changing it
    cout << myInteger << endl;

    // dereference the pointer and add 5 to the integer it points to
    *myIntegerPointer += 5;

    // print the value of the integer after changing
    // it through the pointer
    cout << myInteger << endl;

    // print the value of the second integer
    cout << mySecondInteger << endl;
}

```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

31

- ❖ 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

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

32

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`

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

33

## Puntatori a oggetti

- ❖ [Foo.H]
- ```

class Foo {
public:
    Foo(); // default constructor
    Foo(int a, int b); // another constructor
    ~Foo(); // destructor

    void bar(); // random method
    int blah; // random public instance
variable
};

```
- ❖ in java
- ```

Foo myFooInstance = new Foo(0, 0);

```
- ❖ In C++
- ```

Foo* myFooInstance = new Foo(0, 0)

```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

34

Puntatori a oggetti

- ❖ in java
- ```

Foo myFooInstance = new Foo(0, 0);

```
- ❖ In C++
- ```

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();

```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

35

Istanze di oggetti

- ❖ In java il solo modo di creare oggetti e di fare new e memorizzare il riferimento ottenuto in una variabile
 - ❖ In C++ è possibile dichiarare (ed ottenere) oggetti senza fare new esplicite o usare puntatori
- ```

Foo myFooInstance(0, 0);

```
- ❖ Dichiarare una variabile di tipo foo e chiama il costruttore;
  - ❖ Se si voleva usare il costruttore di default
- ```

Foo myFooInstance;
oppure equivalentemente:
Foo myFooInstance();

```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

36

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 associarle esplicitamente ad una variabile:

```
❖ // ... suppose the class Bar defines  
// the method setAFoo(Foo foo) ...  
Bar bar;  
bar.setAFoo( Foo(5,3) ); // pass an instance of Foo
```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

37

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

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

38

```
[Bar.H]  
#include "Foo.H" // must include Foo.H since  
                // we declare an instance of it  
  
class Bar {  
public:  
    Bar(int a, int b);  
protected:  
    Foo m_foo; // declare an instance of Foo  
};  
  
[Bar.C]  
Bar::Bar(int a, int b) : m_foo(a,b) // call Foo::Foo(int,int)  
                                // and initialize m_foo  
{  
    Foo fooLocal; // create another instance of Foo, this time  
                // as a local var  
  
    // do something with the two Fools, m_foo and fooLocal  
}
```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

39

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

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

40

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: %d\n" << foo << endl;  
    cout << "bar is: %d\n" << bar << endl;  
}
```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

41

References

- ❖ References sono come puntatori solo che:
- ❖ Possono essere assegnati SOLO alla creazione
- ❖ Non possono essere null
- ❖ Si accede al loro contenuto senza operatori di dereferenziazione

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

42

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 {
protected:
    Foo & m_foo; // declare an reference to a bar
public:
    Bar(Foo & fooToStore) : m_foo(fooToStore) {}
};
```

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

43

Da puntatori a variabili

```
Foo* fooPointer = new Foo(0, 0); // create a pointer to
// Foo and give it a value
Foo myFooInstance = *fooPointer; // dereference the pointer
// and assign it to myFooInstance;
// A copy is made (!)
```

Modificando myFooInstance NON si modifica anche l'oggetto puntato da fooPointer

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

44

Costruttore di copia

- ❖ Particolare tipo di costruttore utilizzato quando si vuole inizializzare un oggetto con un altro dello stesso tipo.

- ❖ Usato nel passaggio di parametri...

- ❖

```
class Foo {
    Foo(const Foo &classToCopy); // copy constructor
};
```

- ❖

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

45

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.

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

46

Local Storage

- ❖

```
{
    int myInteger; // memory for an integer allocated
    // ... myInteger is used here ...

    Bar bar; // memory for instance of class Bar allocated
    // ... bar is used here ...
}
```

- ❖ '{' 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!

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

47

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.

28 Ott 2002

Costruzione di Interfacce - Paolo Cignoni

48

```

[Bar.H]
class Bar {
public:
    Bar();
    Bar(int a);
    void myFunction(); // this method would be defined
                        // elsewhere (e.g. in Bar.C)
protected:
    int m_a;
};
Bar::Bar() { m_a = 0; }
Bar::Bar(int a) { m_a = a; }

[main.C]
#include "Bar.H"
int main(int argc, char *argv[])
{ // declare a pointer to Bar; no memory for a Bar instance is
  // allocated now p currently points to garbage
  Bar * p;
  { // create a new instance of the class Bar (*p)
    // store pointer to this instance in p
    p = new Bar();
    if (p == 0) {
        // memory allocation failed
        return 1;
    }
  }
  // since Bar is in global storage, we can still call methods on it
  // this method call will be successful
  p->myFunction();
}
28 Ott 2002      Costruzione di Interfacce - Paolo Cignoni      49

```

Delete esplicite

- ❖ In java si alloca con new e la memoria viene liberata automaticamente dal garbage collector
- ❖ In c++ NO. Ci deve pensare l'utente a disallocare esplicitamente quel che ha esplicitamente allocato con una new
- ❖ delete p; // memory pointed to by p is deallocated

Solo oggetti creati con new possono essere deallocati con delete.

28 Ott 2002 Costruzione di Interfacce - Paolo Cignoni 50

Memory Management

```

[foo.H]
#include "Bar.H"

class Foo {
private:
    Bar* m_barPtr;
public:
    Foo() {}
    ~Foo() {}
    void funcA() {
        m_barPtr = new Bar;
    }
    void funcB() {
        // use object *m_barPtr
    }
    void funcC() {
        // ...
        delete m_barPtr;
    }
};
28 Ott 2002      Costruzione di Interfacce - Paolo Cignoni      51

```

Memory Management

```

{
    Foo myFoo; // create local instance of Foo
    myFoo.funcA(); // memory for *m_barPtr is
                  allocated

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

    myFoo.funcC(); // memory for *m_barPtr is
                  deallocated
}
28 Ott 2002      Costruzione di Interfacce - Paolo Cignoni      52

```

Memory Management

```

{
    Foo myFoo;
    // ...
    myFoo.funcB(); // oops, bus error in funcB()

    myFoo.funcA(); // memory for *m_barPtr is allocated

    myFoo.funcA(); // memory leak, you lose track of
the memory previously // pointed to by m_barPtr when new
instance stored
    // ...
    myFoo.funcB();
} // memory leak! memory pointed to by m_barPtr in
myFoo is never deallocated
28 Ott 2002      Costruzione di Interfacce - Paolo Cignoni      53

```