C++ Programming

Classes, Constructors, Operator overloading

M1 Math
Michail Lampis
michail.lampis@dauphine.fr

Classes

- These slides demonstrate a number of C++ features related to classes:
  - Data encapsulation
  - Operator overloading
  - Constructors/Destructors and memory management
- We will use two running examples:
  - The Complex class (continued from last slide set)
  - The Vector class (on next slide set)
  - See also sections 3.2.3 and 4.1 of notes for alternative versions

A complex class

- Recall that a complex number is a + ib

```
class Complex {
private: //not needed – private is default
    double a,b;
public: //list of methods to manipulate complex nums
    double abs( );
    ...
}
```

Using complex numbers

```
…
Complex z1, z2; //OK
z1.a = 2.2; z1.b = 3.4; //meaning z1 = 2.2 + 3.4i
//ERROR!!
```

- Remember: a,b are **private** fields of the Complex class. They cannot be read/written from outside the methods of the class

Getter/Setter functions

- Solution: define an interface (== collection of methods) to read/write the private fields of a class

```
class Complex {
double a,b;
public:
    double read_a() { return a; }
    double read_b() { return b; }
    void set_a(double new_a) { a = new_a; }
    void set_b(double new_b) { b = new_b; }
}
```

Using getter/setter functions

```
Complex z1,z2;
z1.set_a(2.3);
z1.set_b(3.7);
//Now z1 = 2.3 +3.7i
…
cout << “z1 = “ << z1.read_a() << “ + i “ <<
z1.read_b();
```

Getter/Setter functions

- Solution: define an interface (== collection of methods) to read/write the private fields of a class

```
class Complex {
double a,b;
public:
    double read_a() { return a; } //What about { return this->a }; ??
    double read_b() { return b; }
    void set_a(double new_a) { a = new_a; }
    void set_b(double new_b) { b = new_b; }
}
```

Why??

- It probably seems like this is a **stupid** solution
  - Why not just make a,b public?
  - Then we could simply read/write them when we want. More efficient and simpler
- Answer: in **object-oriented programming** we want the interface of a class to be **independent** of its implementation.
- This principle is called **data encapsulation.** This is the reason fields are by default private.

Data encapsulation

- Why is data encapsulation useful?
  - Recall: there are two ways to represent a complex number: z = a + ib or z = r e^(iθ) (polar coordinates)
- Suppose that you have partly written a program that uses the current Complex class (with a,b)
- Then you decide that you prefer polar coordinates (perhaps they give smaller rounding errors for some operations)
- If z1.a = 2.3 is allowed, **you have to rewrite your whole program!**
- Otherwise, you may just need to rewrite the get/set functions and everything else will work.

Example

```
class Complex {
//double a,b; Changed my mind!
double r, theta;
public:
    double read_a() { return a; } //This doesn't work any more!!!
    double read_b() { return b; }
    void set_a(double new_a) { a = new_a; }
    void set_b(double new_b) { b = new_b; }
}
```

Example

```
class Complex {
//double a,b; Changed my mind!
double r, theta;
public:
    double read_a() { return r*cos(theta); }
    double read_b() { return r*sin(theta); }
    void set_a(double new_a) { ... }
    void set_b(double new_b) { ... }
}
```

Operator overloading

- We now know enough to make a reasonable implementation of a Complex class
- We would probably like to have an add function

  Complex add(Complex z1,Complex z2)
- Wouldn't it be a lot easier if we could just write z3 = z1+z2?
- Problem: + is generally not defined for classes
- But we can define it!

Overloading

- Overloading is when we use the same function name/symbol to refer to several different operations.
- C++ understands what we mean by the **type** of the parameters used.
- This can be done also for standard C++ operators, such as +,-,=,[ ],...
- We have already seen this: cout << x , vs. 5 << 2

A simple Complex class

```
class Complex {
        double a,b;
   public:
        Complex(double, double);
        Complex();
        ~Complex();
        Complex operator+(Complex );
        friend ostream& operator<<(ostream &, Complex );
};
```

Explanation

- One thing to notice in the definition of the previous class

  Complex operator+(Complex);

- This is the line that allows us to redefine the + operator and replace with our own method
- Another thing to notice: We defined two methods named Complex (overloading!) and one named ~Complex. What do these do?

Constructors

- A method that has the same name as the class is a **constructor.** Such a method will be called every time a new object of the class is created
  - Automatically on the stack (via declaration)
  - With new
- If more than one exist, the appropriate one is called by looking at the number/type of parameters given

Defining a constructor

```
Complex::Complex(double a, double b)
{
        this -> a = a;
        this -> b = b;
}
...
Complex z1(2.2, 3.3); //Calls the constructor!
```

Default constructor

- If we don't define any constructor, C++ gives one for free
  - The default constructor takes no parameters and **does nothing!**
- If we define one constructor then the default constructor (that takes no parameters) is not defined
  - We have to write it ourselves, if we want it...

Default constructor

```
Complex::Complex()
{
        this - > a = 0.0;
        this - > b = 0.0;
}
```

Destructor

- A class method whose name is ~(class name) is a destructor.
- There is only one destructor
- It takes no parameters
- You cannot call it directly!
- Will be called automatically when an object is destroyed:
  - By going out of scope (end of function)
  - With delete operator

Example

- For this class we don't really need a destructor
  - Main use: cleaning up memory management. Do a delete for every new, etc.
- For debugging purposes, let's see how often a destructor is called.

```
Complex::~Complex()
{
        cout << "Destructor!" << a << "+i" << b <<  endl;
}
```

A simpler Complex class

```
lass Complex {
        double a,b;
   public:
        Complex(double, double);
        Complex();
        ~Complex();
        Complex add(Complex ); //Addition
        friend ostream& operator<<(ostream &, Complex );
};
```

Exercise

- Implement the add method of the previous slide
  - The result should be the sum of the current complex number (this) and the complex number given as parameter

Solution

```
Complex Complex::add(Complex z2)
{
        Complex res(this->a+z2.a, this->b+z2.b);
        return res;
}
…
Complex z1(2.2,3.3);
Complex z2(4.4,5.5);
z1.add(z2); //This computes the Complex 6.6,8.8
```

Solution with overloading

```
Complex Complex::operator+(Complex& z2)
{
        Complex res(this->a+z2.a, this->b+z2.b);
        return res;
}
```

Friends

- What about this?

  friend ostream& operator<<(ostream &, Complex);
- A friend method is a method that **does not belong to the class** but can still access its private elements.
- In this case, we define a friend method to be able to print complex numbers.
- We are also overloading the << operator!

Friend implementation

```
ostream& operator<<(ostream &out, Complex z)
{
        return out << z.a << " +i" << z.b;
}
```

A program that uses Complex

```
int main()
{
        Complex z1(2.2,3.3);
        cout << z1 << endl;
        Complex z2(4.4,5.5);;
        cout << z2 << endl;
        Complex z3(0,0);
        z3 = z1 + z2;
        cout << z3 << endl;
}
```

Question

- What is the output of the previous program?
  - In particular, remember that we set the destructor to print a message.
  - How many times was the destructor called?

Output

```
2.2 +i3.3
Destructor!2.2+i3.3
4.4 +i5.5
Destructor!4.4+i5.5
Destructor!0+i0
Destructor!6.6+i8.8
Destructor!4.4+i5.5
6.6 +i8.8
Destructor!6.6+i8.8
Destructor!6.6+i8.8
Destructor!4.4+i5.5
Destructor!2.2+i3.3
```

Avoiding overhead

- The reason that the destructor is called so many times is that temporary copies of Complex numbers are made when they are passed as parameters or returned.
- This is bad! (OK for such a simple class)
- One way to avoid: use more pointers
  - Need to change code a lot, harder to read
- Alternative: use references
  - References are pointers without the *

Complex with Refs

```
class Complex {
        double a,b;
        public:
        Complex(double, double);
        Complex();
        ~Complex();
        Complex operator+(Complex& );
        friend ostream& operator<<(ostream &, Complex& );
};
```

Improved?

- Previous program now only calls ~Complex 4 times
  - Good!
- Consider this piece of code:

  z3 = z1 + z2;
- Will this work?
  - z4 = z1 + z2 + z3;
  - z4 = z1 + (z2 + z3); //Is this the same??