Advanced C++ programming

Introduction, Classes, Pointers and rerefences, Operators

Giuseppe Lipari - Luigi Palopoli

CRIStAL - Université de Lille
Embedded Intelligence and Robotic Systems - Università di Trento

Outline

Course Contents

Classes

Pointers and dynamic memory

References

Copy Constructor

Operator Overloading

Outline

Course Contents

Classes

Pointers and dynamic memory

References

Copy Constructor

Operator Overloading

Pre-requisites

To understand this course, you should at least know the basic C syntax

- functions declaration and function call,
- global and local variables
- pointers (will do again during the course)
- structures

Conventions

When explaining a concept:

- if nothing is said, the concept is valid for all standard C++ language versions (e.g. starting from C++98)
- if C++11 is specified, the concept is valid starting from C++11
  - when compiling with g++ or clang++, you must specify the option `-std=c++11`
- if C++14 is specified, the concept is valid only from C++14 (and not for C++11, or C++98)
  - when compiling with g++ or clang++, you must specify the option `-std=c++14`
- if C++17 is specified, the concept is only valid C++17

Outline

Course Contents

Classes

Pointers and dynamic memory

References

Copy Constructor

Operator Overloading

Class

- Class is the main construct for building new types in C++
  - A class is almost equivalent to a struct with functions inside
  - In the C-style programming, the programmer defines structs, and global functions to act on the structs
  - In C++-style programming, the programmer defines classes with embedded functions

```
class MyClass {
  int a;
public:
  int myfunction(int para
};
```

Class declaration

Member variable

Member function

Remember the semicolon!

Object construction

- An object is an instance of a class
- An object is created by calling a special function called *constructor*
  - A constructor is a function that has the same name of the class and no return value
  - It may or may not have parameters;
  - It is invoked in a special way

```
class MyClass {
public:
  MyClass() {
    cout << "Constructor"<<endl;
  }
};
MyClass obj;
```

Declaration of the constructor

Object construction

- An object is an instance of a class
- An object is created by calling a special function called *constructor*
  - A constructor is a function that has the same name of the class and no return value
  - It may or may not have parameters;
  - It is invoked in a special way

```
class MyClass {
public:
  MyClass() {
    cout << "Constructor"<<endl;
  }
};
MyClass obj;
```

Declaration of the constructor

Invoke the constructor to create an object

Constructor - II

A class can have many constructors

```
class MyClass {
    int a;
    int b;
public:
    MyClass(int x);
    MyClass(int x, int y);
};

MyClass obj;
MyClass obj1(2);
MyClass obj2(2,3);

int myvar(2);
double pi(3.14);
```

This is an error, there is no constructor without parameters

Calls the first constructor

Calls the second constructor

Same syntax is valid for primitive data types

Default constructor

- Rules for constructors
  - If you do not specify a constructor, a default one with no parameters is provided by the compiler
  - If you provide a constructor (any constructor) the compiler will not provide a default one for you
- Constructors are used to initialise members

```
class MyClass {
    int a;
    int b;
public:
    MyClass(int x, int y)  {
        a = x; b = 2*y;
    }
};
```

Initialization list

- Members can be initialised through a special syntax
  - This syntax is preferable (the compiler can catch some obvious mistake)
  - Use it whenever you can (i.e. almost always)

```
class MyClass {
  int a;
  int b;
public:
  MyClass(int x, int y) :
        a(x), b(y)
  {
        // other initialisation
  }
};
```

A comma separated list of constructors, following the :

Accessing members

- Members of one object can be accessed using the classical dot notation, similarly to structs in C and objects in Java

```
class MyClass {
public:
  int a;
  int f();
  void g(int i, int ii);
};

MyClass x;
MyClass y;

x.a = 5;
y.a = 7;
x.f();
y.g(5, 10);
```

Assigning to a member variable of object x

Accessing members

- Members of one object can be accessed using the classical dot notation, similarly to structs in C and objects in Java

```
class MyClass {
public:
   int a;
   int f();
   void g(int i, int ii);
};

MyClass x;
MyClass y;

x.a = 5;
y.a = 7;
x.f();
y.g(5, 10);
```

Assigning to a member variable of object x

Assigning to a member variable of object y

Accessing members

- Members of one object can be accessed using the classical dot notation, similarly to structs in C and objects in Java

```
class MyClass {
public:
  int a;
  int f();
  void g(int i, int ii);
};

MyClass x;
MyClass y;

x.a = 5;
y.a = 7;
x.f();
y.g(5, 10);
```

Assigning to a member variable of object x

Assigning to a member variable of object y

Calling member function f() of object x

Accessing members

- Members of one object can be accessed using the classical dot notation, similarly to structs in C and objects in Java

```
class MyClass {
public:
  int a;
  int f();
  void g(int i, int ii);
};

MyClass x;
MyClass y;

x.a = 5;
y.a = 7;
x.f();
y.g(5, 10);
```

Assigning to a member variable of object x

Assigning to a member variable of object y

Calling member function f() of object x

Calling member function g() of object y

Implementing member functions

- Unlike Java and Python, you can implement a member function (including constructors) in a separate .cpp file

complex.hpp

```
class Complex {
  double real_;
  double img_;
public:
  ...
  double module() const;
  ...
};
```

complex.cpp

```
double Complex::module()
{
  double temp;
  temp = real_ * real_ +
         img_ * img_;
  return temp;
}
```

- The `::` operator is called scope resolution operator
- member variables and functions can be accessed without *dot* or *arrow*

Question

Code decomposition

Which part of the implementation should be in the *.cpp* and in *.hpp*?

Question

Code decomposition

Which part of the implementation should be in the *.cpp* and in *.hpp*?

- The *.hpp* file is actually an interface. Something to communicate what your module is about.
- It should contain declarations and comments to document the code (e.g., using doxyGen)
- There are two exception: inline functions and template. In theory all inlines and template implementation should be contained in the *.hpp* file....but we will find good workarounds.

Uniform Initialisation (C++11)

- Starting from C++11, it is possible to initialise an object with 3 different syntaxes:

```
int a = 0;

int a(0);

int a{0};
```

- In this case, the three are equivalent
- The last one is called uniform initialisation because it is the most general
  - although it cannot be used everywhere, as we will see...

Uniform initialisation (C++11)

- The reason for introducing uniform initialisation is more apparent for object constructors

```
Widget w1(7);

Widget w2;

Widget w3();

Widget w4{7};

Widget w5{};
```

Calls the constructor taking one int argument

Uniform initialisation (C++11)

- The reason for introducing uniform initialisation is more apparent for object constructors

```
Widget w1(7);

Widget w2;

Widget w3();

Widget w4{7};

Widget w5{};
```

Calls the constructor taking one int argument

Calls the default constructor

Uniform initialisation (C++11)

- The reason for introducing uniform initialisation is more apparent for object constructors

```
Widget w1(7);

Widget w2;

Widget w3();

Widget w4{7};

Widget w5{};
```

Calls the constructor taking one int argument

Calls the default constructor

Declares a function!

Uniform initialisation (C++11)

- The reason for introducing uniform initialisation is more apparent for object constructors

```
Widget w1(7);

Widget w2;

Widget w3();

Widget w4{7};

Widget w5{};
```

Calls the constructor taking one int argument

Calls the default constructor

Declares a function!

Calls the constructor taking one int argument

Uniform initialisation (C++11)

- The reason for introducing uniform initialisation is more apparent for object constructors

```
Widget w1(7);
Widget w2;
Widget w3();
Widget w4{7};
Widget w5{};
```

Calls the constructor taking one int argument

Calls the default constructor

Declares a function!

Calls the constructor taking one int argument

Calls the default constructor

Class member initialisation (C++11)

- Since C++11 it is possible to initialise member variables directly in the declaration with default values

```
class A {
  int a=0;
  int b{0};
  Widget obj{"ObjName"};
public:
  A() {}
  A(int x) : a(x), b(x) {}
};

A obj1;

A obj2{3};
```

- Notice the use of the uniform initialisation syntax
- NB: You cannot use parenthesis in default initialisation of member variables
- the value of `obj1.a` is equal to 0 because the object is initialised by the default constructor
- the value of `obj2.a` is equal to 3 because the object is initialised by the second constructor

Access control keywords

- A member can be:
  - private: only member functions of the same class can access it; other classes or global functions can't
  - protected: only member functions of the same class or of derived classes can access it: other classes or global functions can't
  - public: every function can access it

```
class MyClass {
private:
    int a;
public:
     int c;
};
```

```
MyClass data;

cout << data.a;    // ERROR!
cout << data.c;    // OK
```

Why access rules?

Meaning of the three access rules

When do you want to use *private* vs *public* vs *protected*?

Why access rules?

Meaning of the three access rules

When do you want to use *private* vs *public* vs *protected*?

- The source code is also a way for communicating your intent.

Why access rules?

Meaning of the three access rules

When do you want to use *private* vs *public* vs *protected*?

- The source code is also a way for communicating your intent.
- By *public* you want to say "use me, I am part of the class interface
- By *private* you want to say "Do not meddle wiht me. I am part of the implementation. I could change in the future"
- By *protected* you want to say, "I am part of the implementation, but all the subclasses share it"

Access control rules

- Default is private
- An access control keyword defines access until the next access control keyword

```
class MyClass {
    int a;
    double b;
public:
      int c;

      void f();
      int getA();
private:
      int modify(double b);
};
```

private (default)

public

private again

Access control and scope

```
int xx;

class A {
    int xx;
public:
    void f();
};
```

global variable

member variable

```
void A::f()
{
    xx = 5;
    ::xx = 3;

    xx = ::xx + 2;
}
```

Access control and scope

```
int xx;

class A {
    int xx;
public:
    void f();
};
```

```
void A::f()
{
    xx = 5;
    ::xx = 3;

    xx = ::xx + 2;
}
```

global variable

member variable

access member xx

Access control and scope

```
int xx;

class A {
    int xx;
public:
    void f();
};
```

global variable

member variable

```
void A::f()
{
    xx = 5;
    ::xx = 3;

    xx = ::xx + 2;
}
```

access member xx

access global xx

Friends

```
class A {
   friend class B;
   int y;
   void f();
public:
   int g();
};

class B {
   int x;
public:
   void f(A &a);
};

void B::f(A &a)
{
    x = a.y;
    a.f();
}
```

B is friend of A ...

... hence B can access private members of A

Friend operators

- Global functions and operators can be friend of a class
- Also, a single member function can be declared friend

```
class A {
    friend B::f();
    friend h();
    int y;
    void f();
public:
    int g();
};
```

friend member function

friend global function

- It is better to use the *friend* keyword only when it is really necessary because it breaks the access rules .

  "Friends, much as in real life, are often more trouble than they're worth." – Scott Meyers

- When you use friend classes and functions you are creating a strong coupling between the friens.
- This means that is you decide to change the design our the implmentation of one of the friends, you have to do so for all its friends

Example of friend global function

```
class A;

void print(A obj);

class A {
  int data;
  friend void print(A obj);
public:
  A() : data(0) {}
};
void print(A obj) {
  cout << obj.data << endl;
}
```

Forward class declaration

Prototype of global function

Friend declaration

Using class private variable

Outline

Course Contents

Classes

Pointers and dynamic memory

References

Copy Constructor

Operator Overloading

Pointers and arrays

- A pointer is a variable that can hold a memory address
- The name of an array is equivalent to a constant pointer to the first element
- With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";

cout << *name << endl;

char *p = name;

p++;

assert(p == name+1);

while (*p != 0)
    cout << *(p++);
    cout << endl;
```

Pointers and arrays

- A pointer is a variable that can hold a memory address
- The name of an array is equivalent to a constant pointer to the first element
- With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";

cout << *name << endl;

char *p = name;

p++;

assert(p == name+1);

while (*p != 0)
    cout << *(p++);
    cout << endl;
```

prints "G"

Pointers and arrays

- A pointer is a variable that can hold a memory address
- The name of an array is equivalent to a constant pointer to the first element
- With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";

cout << *name << endl;

char *p = name;

p++;

assert(p == name+1);

while (*p != 0)
    cout << *(p++);
    cout << endl;
```

prints "G"

declares a pointer to the first element of the array

Pointers and arrays

- A pointer is a variable that can hold a memory address
- The name of an array is equivalent to a constant pointer to the first element
- With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";

cout << *name << endl;

char *p = name;

p++;

assert(p == name+1);

while (*p != 0)
    cout << *(p++);
    cout << endl;
```

prints "G"

declares a pointer to the first element of the array

Pointer arithmetic: increments the pointer, now points to "i"

Pointers and arrays

- A pointer is a variable that can hold a memory address
- The name of an array is equivalent to a constant pointer to the first element
- With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";

cout << *name << endl;

char *p = name;

p++;

assert(p == name+1);

while (*p != 0)
    cout << *(p++);
    cout << endl;
```

prints “G”

declares a pointer to the first element of the array

Pointer arithmetic: increments the pointer, now points to “i”

this assertion is correct

Pointers and arrays

- A pointer is a variable that can hold a memory address
- The name of an array is equivalent to a constant pointer to the first element
- With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";

cout << *name << endl;

char *p = name;

p++;

assert(p == name+1);

while (*p != 0)
    cout << *(p++);
    cout << endl;
```

prints "G"

declares a pointer to the first element of the array

Pointer arithmetic: increments the pointer, now points to "i"

this assertion is correct

zero marks the end of the string

Pointers and arrays

- A pointer is a variable that can hold a memory address
- The name of an array is equivalent to a constant pointer to the first element
- With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";

cout << *name << endl;

char *p = name;

p++;

assert(p == name+1);

while (*p != 0)
    cout << *(p++);
```

prints “G”

declares a pointer to the first element of the array

Pointer arithmetic: increments the pointer, now points to “i”

this assertion is correct

zero marks the end of the string

Dynamic memory

- Dynamic memory is managed by the user
- In C:
  - to allocate memory, call function `malloc`
  - to deallocate, call `free`
  - Both take pointers to any type, so they are not type-safe
- In C++
  - to allocate memory, use operator `new`
  - to deallocate, use operator `delete`
  - they are more type-safe

The new operator

- The `new` and `delete` operators can be applied to primitive types and user-defined classes
- `operator new` automatically calculates the size of memory to be allocated

Allocates an integer pointed by p

```
class A { ... };

int *p = new int(5);

A obj;

A *q = new A();

delete p;
delete q;
```

The new operator

- The `new` and `delete` operators can be applied to primitive types and user-defined classes
- `operator new` automatically calculates the size of memory to be allocated

```
class A { ... };

int *p = new int(5);

A obj;

A *q = new A();

delete p;
delete q;
```

Allocates an integer pointed by p

A statically allocated object (no new!)

The new operator

- The `new` and `delete` operators can be applied to primitive types and user-defined classes
- `operator new` automatically calculates the size of memory to be allocated

```
class A { ... };

int *p = new int(5);

A obj;

A *q = new A();

delete p;
delete q;
```

Allocates an integer pointed by p

A statically allocated object (no new!)

It does two things:
1) Allocates memory for an object of class A
2) calls the constructor of A()

The new operator

- The `new` and `delete` operators can be applied to primitive types and user-defined classes
- `operator new` automatically calculates the size of memory to be allocated

```
class A { ... };

int *p = new int(5);

A obj;

A *q = new A();

delete p;
delete q;
```

Allocates an integer pointed by p

A statically allocated object (no new!)

It does two things:
1) Allocates memory for an object of class A
2) calls the constructor of A()

Deallocates the memory pointed by p

The new operator

- The `new` and `delete` operators can be applied to primitive types and user-defined classes
- `operator new` automatically calculates the size of memory to be allocated

```
class A { ... };

int *p = new int(5);

A obj;

A *q = new A();

delete p;
delete q;
```

Allocates an integer pointed by p

A statically allocated object (no new!)

It does two things:
1) Allocates memory for an object of class A
2) calls the constructor of A()

Deallocates the memory pointed by p

It does two things:
1) Calls the *destructor* for A
2) deallocates the memory pointed by q

Destructor

- The destructor is called just before the object is deallocated.
- It is always called for all objects (allocated on the stack, in global memory, or dynamically)
- If the programmer does not define a destructor, the compiler automatically adds one by default (which does nothing)

```
class A {
  ...
public:
  A() { ... }  // constructor
  ~A() { ... } // destructor
};
```

The destructor never takes any parameter

Why a destructor ?

- A destructor is useful when an object dynamically allocates memory, so that it can deallocate it when the object is deleted

```
class A { ... };

class B {
    A *p;
public:
    B() {
        p = new A();
    }
    ~B() {
        delete p;
    }
};
```

- The memory is allocated when the object is created ..
- ... and it is deallocated when the object is deleted

Example with destructor

examples/destructor.cpp

When is the destructor called ?

- The destructor is called automatically every time the object is destroyed
  - For statically allocated global objects: before the program terminates
  - For automatic objects (i.e. allocated on the stack): when the function (or the enclosing block) terminates
  - For dynamically created objects with new: when the delete is called.

New and delete for arrays

- To allocate an array, use this form

```
int *p = new int[5]; // allocates an array of 5 int
...
delete [] p;         // notice the delete syntax

A *q = new A[10];    // allocates an array of 10
...                  // objects of type A
delete [] q;
```

- In the second case, the default constructor is called to build the 10 objects
- Therefore, this can only be done is a default constructor (without arguments) is available

Null pointer

- The address 0 is an invalid address
  - (no data and no function can be located at 0)
- therefore, in C/C++ a pointer to 0 is said to be a *null pointer*, which means a pointer that points to nothing.
- Dereferencing a null pointer is always a bad error (null pointer exception, or segmentation fault)
- In C, the macro `NULL` is used to mark 0, or a pointer to 0
  - however, 0 can be seen to be of integer type, or a null pointer
- In C++11, the null pointer is indicated with the constant `nullptr`
  - this constant cannot be automatically converted to an integer

Pointer to objects

examples/pointerarg.cpp

Pointer to objects

examples/pointerarg.cpp

What happened:

- function `g()` takes an object, and makes a copy
  - `c` is a copy of `obj`
  - `g()` has no side effects, as it works on the copy
- Function `h()` takes a pointer to the object
  - it works on the original object `obj`, changing its internal value

Outline

Course Contents

Classes

Pointers and dynamic memory

References

Copy Constructor

Operator Overloading

References

- In C++ it is possible to define a reference to a variable or to an object

```
int x;             // variable
int &rx = x;       // reference to variable

MyClass obj;       // object
MyClass &r = obj;  // reference to object
```

- WARNING!
  - C++ uses the same symbol & for two different meanings!
  - when used in a declaration/definition, it denotes a reference
  - when used in an expression, it is an operator to obtain the address of a variable in memory

References vs pointers

- There is quite some difference between references and pointers

```
MyClass obj;         // the object
MyClass &r = obj;    // a reference
MyClass *p;          // a pointer
p = &obj;            // p takes the address of obj

obj.fun();        // call method fun()
r.fun();          // call the same method by reference
p->fun();         // call the same method by pointer

MyClass obj2;     // another object
p = & obj2;       // p now points to obj2
r = obj2;         // compilation error!
                  // Cannot change a reference!
MyClass &r2;      // compilation error!
                  // Reference must be initialized
```

Reference vs pointer

- In C++, a reference is an *alternative name* for an object

Pointers	References
May be uninitialised	Must be initialised
Pointers are like other variables	Not a "variable", just an alias
Can have a pointer to `void`	No references to void
Can be assigned arbitrary values	Cannot be assigned any value
It is possible to do arithmetic	Cannot do arithmetic

Reference example

examples/referencearg.cpp

- Notice the differences:
  - Method declaration: `void h(MyClass &c);` instead of `void h(MyClass *p)`
  - Method call: `h(obj);` instead of `h(&obj)`
  - In the first case, we are passing a reference to an object
  - In the second case, the address of an object
- References are much less powerful than pointers
- However, they are much safer than pointers
  - The programmer cannot accidentally misuse references, whereas it is easy to misuse pointers

This

- Inside a class method, it is possible to use the this special pointer to the object

```
Complex &Complex::add_to(const Complex &c)
{
  this->real_ += c.real_;
  this->img_  += c.img_;
  return *this;
}
// ...

Complex a(1,0);
Complex b(0,1);
Complex c(1,1);
Complex d;
d.add_to(a).add_to(b).add_to(c);
```

Equivalent to `real_ += c.real_;`

This

- Inside a class method, it is possible to use the this special pointer to the object

```
Complex &Complex::add_to(const Complex &c)
{
  this->real_ += c.real_;
  this->img_  += c.img_;
  return *this;
}
// ...

Complex a(1,0);
Complex b(0,1);
Complex c(1,1);
Complex d;
d.add_to(a).add_to(b).add_to(c);
```

Equivalent to `real_ += c.real_;`

Returns the reference to this object

This

- Inside a class method, it is possible to use the this special pointer to the object

```
Complex &Complex::add_to(const Complex &c)
{
  this->real_ += c.real_;
  this->img_  += c.img_;
  return *this;
}
// ...

Complex a(1,0);
Complex b(0,1);
Complex c(1,1);
Complex d;
d.add_to(a).add_to(b).add_to(c);
```

Equivalent to `real_ += c.real_;`

Returns the reference to this object

Chaining of method calls: the second call is executed on the object modified by the first call

Outline

Course Contents

Classes

Pointers and dynamic memory

References

Copy Constructor

Operator Overloading

Copying objects

- In a previous example, function g() is taking an object by value

```
void g(MyClass c) {...}
...
g(obj);
```

  - The original object is copied into parameter c ;
  - The copy is done by invoking the *copy constructor*

```
MyClass(const MyClass &r);
```

- If the user does not define it, the compiler will define a default one for us automatically
  - The default copy constructor just performs a bitwise copy of all members

Example

- Let's add a copy constructor to `MyClass`, to see when it is called

  `examples/copy1.cpp`
- Now look at the output
  - The copy constructor is automatically called when we call `g()`
  - It is not called when we call `h()`

Usage

- The copy constructor is called every time we initialise a new object to be equal to an existing object

```
MyClass ob1(2);     // call constructor
MyClass ob2(ob1);   // call copy constructor
MyClass ob3 = ob2;  // call copy constructor
MyClass ob3{ob2};   // call copy constructor
```

- We can prevent a copy by making the copy constructor private:

```
// can't be copied!
class MyClass {
    MyClass(const MyClass &r);
public:
    ...
};
```

Copy constructor in C++11

- In the new standard C++11, the copy constructor can be *hidden* by using keyword "= delete" after the member declaration

```
// can't be copied!
class MyClass {
public:
    MyClass();
    MyClass(const MyClass &r) = delete ;
    ...
};
```

Hint

When starting the implementation of a class, disable copy constructor and assignment operator by default, since in most cases you will not need it. This will allow you to catch some additional errors.

Const references

- Let's analyse the argument of the copy constructor

```
MyClass(const MyClass &r);
```

- It means:
  - This function accepts a reference
  - however, the object will not be modified: it is treated as a *constant* within the scope of the function
  - The compiler checks that the object is not modified by checking the *constness* of the methods
  - As a matter of fact, the copy constructor does not modify the original object: it only reads its internal values in order to copy them into the new object
- If the programmer by mistake tries to modify a field of `r`, the compiler will give an error

Outline

Course Contents

Classes

Pointers and dynamic memory

References

Copy Constructor

Operator Overloading

Operator overloading

- An operator in C++ is a function
  - binary operator: a function that takes two arguments
  - unary operator: a function that takes one argument
- The syntax is the following:
  - `Complex &operator+=(const Complex &c);`
- Of course, if we apply operators to predefined types, the compiler does not insert a function call

```cpp
int a = 0;
a += 4;

Complex b;
Complex c(1,3);
b += 5;
b.+=(5);
c = b;
```

Default constructor

Constructor

Sum operator

Equivalent to b += 5;

Assignment operator

A complete example

```
class Complex {
  double real_;
  double imaginary_;
public:
  Complex();                        // default constructor
  Complex(double a, double b = 0); // constructor
  ~Complex();                       // destructor
  Complex(const Complex &c);        // copy constructor

  double real() const;             // member function
  double imaginary() const;        // member function
  double module() const;           // member function
  Complex &operator =(const Complex &a);  // assignment operator
  Complex &operator+=(const Complex &a);  // sum operator
  Complex &operator-=(const Complex &a)); // sub operator
};

Complex operator+(const Complex &a, const Complex &b);
Complex operator-(const Complex &a, const Complex &b);
```

To be member or not to be...

- In general, operators that modify the object (like ++, +=, –, etc...) should be member
- Operators that do not modify the object (like +, -, etc,) should not be member, but friend functions
- Let's write `operator+` for complex: `examples/complex.cpp`
- Not all operators can be overloaded
  - we cannot "invent" new operators,
  - we can only overload existing ones
  - we cannot change number of arguments
  - we cannot change precedence
  - . (dot) cannot be overloaded

Copy constructor and assignment operator

- The assignment operator looks very similar to the copy constructor

```
Complex c1(2,3);
Complex c2(2);
Complex c3 = c1;

c2 = c3;
c1 += c2;

cout << c1 << "    "
     << c2 << "    " << c3 << "\n";
```

Copy constructor

Assignment

- The difference is that c3 is being defined and initialized, so a constructor is necessary;
- c2 is already initialised

The add function

- Now suppose we want to write the sum operator to sum two complex numbers
- First try

```
Complex operator+(Complex a, Complex b)
{
  Complex z(a.real() + b.real(),
                       a.imaginary() + b.imaginary());
  return z;
}
```

- This is not very good programming style!

Usage

- Let's see what happens when we use our *add function*

```
Complex c1(1,2),c2(2,3),c3;

c3 = c1+c2;

// ...

Complex operator+(Complex a, Complex b)
{
  Complex z(a.real() + b.real(),
    a.imaginary() + b.imaginary());
  return z;
}
```

Usage

- Let's see what happens when we use our *add function*

```
Complex c1(1,2),c2(2,3),c3;

c3 = c1+c2;

// ...

Complex operator+(Complex a, Complex b)
{
  Complex z(a.real() + b.real(),
    a.imaginary() + b.imaginary());
  return z;
}
```

c1 and c2 are copied (through the copy constr.) into a and b

Usage

- Let's see what happens when we use our *add function*

```
Complex c1(1,2),c2(2,3),c3;

c3 = c1+c2;

// ...

Complex operator+(Complex a, Complex b)
{
  Complex z(a.real() + b.real(),
    a.imaginary() + b.imaginary());
  return z;
}
```

z is constructed

c1 and c2 are copied (through the copy constr.) into a and b

Usage

- Let's see what happens when we use our *add function*

```
Complex c1(1,2),c2(2,3),c3;

c3 = c1+c2;

// ...

Complex operator+(Complex a, Complex b)
{
  Complex z(a.real() + b.real(),
    a.imaginary() + b.imaginary());
  return z;
}
```

z is copied into a temp. object

z is constructed

c1 and c2 are copied (through the copy constr.) into a and b

Usage

- Let's see what happens when we use our *add function*

```
Complex c1(1,2),c2(2,3),c3;

c3 = c1+c2;

// ...

Complex operator+(Complex a, Complex b)
{
  Complex z(a.real() + b.real(),
    a.imaginary() + b.imaginary());
  return z;
}
```

The temp. object is assigned to c3 calling the assignment operator

z is copied into a temp. object

z is constructed

c1 and c2 are copied (through the copy constr.) into a and b

Usage

- Let's see what happens when we use our *add function*

```
Complex c1(1,2),c2(2,3),c3;

c3 = c1+c2;

// ...

Complex operator+(Complex a, Complex b)
{
  Complex z(a.real() + b.real(),
    a.imaginary() + b.imaginary());
  return z;
}
```

The temp. object is destroyed

The temp. object is assigned to c3 calling the assignment operator

z is copied into a temp. object

z is constructed

c1 and c2 are copied (through the copy constr.) into a and b

7 function calls are involved!

Improvement

- Let's pass by const reference:

```
Complex c1(1,2),c2(2,3),c3;

c3 = c1+c2;
```

```
Complex operator+(const Complex& a, const Complex& b)
{
    Complex temp(a.real() + b.real(),
        a.imaginary() + b.imaginary());
    return temp;
}
```

- We saved 2 function calls
  - Notice that c1 and c2 cannot be modified anyway
- The compiler optimizes the temporary, so finally the code becomes very efficient

Strange operators

You can overload:

- new and delete
  - used to build custom memory allocate strategies
- operator[]
  - for example, in vector<>...
- operator,
  - You can write very funny programs!
- operator->
  - used to make smart pointers

How to overload new and delete

```
class A {
    ...
public:
    void* operator new(size_t size);
    void operator delete(void *);
};
```

- Every time we call new for creating an object of this class, the overloaded operator will be called
- You can also overload the global version of new and delete

How to overload * and ->

- This is the prototype

```
class Iter {
  ...
public:
  Obj operator*() const;
  Obj *operator->() const;
};
```

- Why should I overload `operator*()`?
  - to implement iterators!
- Why should I overload `operator->()`?
  - to implement smart pointers

Output on streams

- It is possible to overload operator«() and operator»()
- This can be useful to output an object on the terminal
- Typical way to define the operator

```
ostream & operator<<(ostream &out, const MyClass &obj);
```

Example

- An example is worth a thousands words

```
class MyClass {
    int x;
    int y;
public:
    MyClass(int a, int b) : x(a), y(b) {}
    int getX() const;
    int getY() const;
};

ostream& operator<<(ostream& out, const MyClass &c) {
    out << "[" << c.getX() << ", " << c.getY() << "]";
    return out;
}

int main() {
   MyClass obj(1,3);
   cout << "Object: " << obj << endl;
}
```