Object Oriented Software Design

References, copy constructor, operators

Giuseppe Lipari
`http://retis.sssup.it/~lipari`

Scuola Superiore Sant'Anna – Pisa

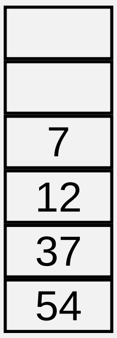
November 24, 2010

Outline

Stack of integers

- Let us implement a Stack of integers class

```
Stack stack;
...
stack.push(12);
stack.push(7);
...
cout << stack.pop();
cout << stack.pop();
```



First, define the interface

```
class Stack {
   ...
public:
    Stack();
    ~Stack();

    void push(int a);
    int pop();
    int peek();
    int size();
};
```

Constructor & destructor

Now the implementation

- Now we need to decide:
    - how many objects can our stack contain?
    - we can set a maximum limit (like 1000 elements)
    - or, we can dynamically adapt
- computer memory is the limit
    - Let’s first choose the first solution notice that this decision is actually part of the interface contract!

Fixed size stack

```
 class Stack {
public:
    Stack(int size);
    ~Stack();

    int push(int a);
    void pop();
    int size();
private:
    int *array_;
    int top_;
    int size_;
};
```

Constructor

- The constructor is the place where the object is created and initialised
    - Every time an object is defined, the constructor is called automatically
    - There is no way to define an object without calling the constructor
    - Sometime the constructor is called even when you don’t suspect (for example for temporary objects)
- It’s a nice feature
    - it forces to think about initialization

Constructor for stack

- The constructor is a function with the same name of the class and no return value
- It can have parameters:
    - in our case, the `max_size` of the stack

```
class Stack {
public:
    Stack(int size);
    ...
};
```

```
Stack::Stack(int size)
{
    array_ = new int[size];
    top = 0;
}
```

The new operator

- In C++, there is a special operator, called `new` to dynamically allocate memory

```
Stack::Stack(int size)
{
    array_ = new int[size];
    size_ = size;
    top_ = 0;
}
```

Creates an array of *size* integers

Destructor

- When the object goes out of scope, it is destructed
  - among the other things, its memory is de-allocated

```
class Stack {
    ...
    ~Stack();
    ...
};
```

```
 Stack::~Stack()
{
    delete []array_;
}
```

When are they called?

```
Stack::Stack(int size)
{
    size_ = size;
    array_ = new int[size_];
    top_ = 0;
    cout << "Constructor has been called\n!";
}

Stack::~Stack()
{
    delete []array_;
    cout << "Destructor has been called\n";
}
```

```
int main()
{
    cout << "Before block\n";
    {
        Stack mystack(20);
        cout <<  "after constructor\n";
        ...
        cout << "before block end\n";
    }
    cout << "After block\n";
}
```

Default constructor

- A constructor without parameters is called default constructor
  - if you do not define a constructor, C++ will provide a default constructor that does nothing
  - if you do provide a constructor with parameters, the compiler does not provide a default constructor

```
Stack s1;

Stack s2(20);
```

Error!! No default constructor for Stack!

Ok, calling the user-defined constructor

Default constructor

- We did not define a default constructor on purpose
  - in our interface, we cannot construct a Stack without knowing its size
- However it is possible to define different constructors using overloading
  - usually, we need to provide several constructors for a class
- The compiler always provide a destructor, unless the programmer provides it

Implementing the Stack interface

- The complete code:
  - `./examples/13.cpp-examples/stack1/stack.h`
  - `./examples/13.cpp-examples/stack1/stack.cpp`
  - `./examples/13.cpp-examples/stack1/stack_main.cpp`
- Improving the implementation: using a list to remove the need for the fixed size
  - `./examples/13.cpp-examples/stack2/stack.h`
  - `./examples/13.cpp-examples/stack2/stack.cpp`
  - `./examples/13.cpp-examples/stack2/stack_main.cpp`

Pointers

- We have seen that we can define a pointer to an object

```
class A { ... };

A myobj;
A *p = &myobj;
```

- Pointer p contains the address of `myobj`

Pointers - II

- As in C, in C++ pointers can be used to pass arguments to functions

```
void fun(int a, int *p)
{
  a = 5;
  *p = 7;
}
...
int x = 0, y = 0;
fun(x, &y);
```

- After the function call, x=`0` while y `= 7`
- x is passed by value (i.e. it is copied into a)
- y is passed by address (i.e. we pass its address, so that it can be modified inside the function)
- Syntax is not very nice

Another example

pointerarg.cpp

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

class MyClass {
    int a;
public:
    MyClass(int i) { a = i; }
    void fun(int y) { a = y; }
    int get() { return a; }
};

void g(MyClass c) {
    c.fun(5);
}

void h(MyClass *p) {
    p->fun(5);
}

int main() {
    MyClass obj(0);

    cout << "Before calling g: obj.get() = " << obj.get() << endl;
    g(obj);
    cout << "After calling g: obj.get() = " << obj.get() << endl;
    h(&obj);
    cout << "After calling h: obj.get() = " << obj.get() << endl;
}
```

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
- It depends on what you want to do!
- However, the syntax is not nice

More on pointers

- It is also possible to define pointers to functions:
  - The portion of memory where the code of a function resides has an address; we can define a pointer to this address

```
void (*funcPtr)();          // pointer to void f();
int (*anotherPtr)(int)      // pointer to int f(int a);

void f(){...}

funcPtr = &f();  // now funcPtr points to f()
funcPtr = f;     // equivalent syntax

(*funcPtr)();    // call the function
```

Pointers to functions – II

- To simplify notation, it is possible to use typedef:

```
typedef void (*MYFUNC)();
typedef void* (*PTHREADFUN)(void *);

void f() { ... }
void *mythread(void *) { ... }

MYFUNC funcPtr = f;
PTHREADFUN pt = mythread;
```

- It is also possible to define arrays of function pointers:

```
void f1(int a) {}
void f2(int a) {}
void f3(int a) {}
...
void (*funcTable []) (int) = {f1, f2, f3}
...
for (int i =0; i<3; ++i) (*funcTable[i])(i + 5);
```

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
```

- `r` is a reference to object `obj`
  - WARNING!
  - C++ uses the same symbol & for two different meanings!
  - Remember:
    - when used in a declaration/definition, it is a reference
    - when used in an instruction, it indicates the address of a variable in memory

References vs pointers

- There is quite a 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
```

- Once you define a reference to an object, the same reference cannot refer to another object later!

Reference vs pointer

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

Pointers

- Pointers are like other variables
- Can have a pointer to `void`
- Can be assigned arbitrary values
- It is possible to do arithmetic

References

- Must be initialised
- Cannot have references to void
- Cannot be assigned
- Cannot do arithmetic

- What are references good for?

Reference example

referencearg.cpp

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

class MyClass {
    int a;
public:
    MyClass(int i) { a = i; }
    void fun(int y) { a = y; }
    int get() { return a; }
};

void g(MyClass c) {
    c.fun(5);
}

void h(MyClass &c) {
    c.fun(5);
}

int main() {
    MyClass obj(0);

    cout << "Before calling g: obj.get() = " << obj.get() << endl;
    g(obj);
    cout << "After calling g: obj.get() = " << obj.get() << endl;
    h(obj);
    cout << "After calling h: obj.get() = " << obj.get() << endl;
}
```

Differences

- 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
  - It is quite easy to misuse pointers

Copying objects

- In the previous example, function `g()` is taking a 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
  - Remember: this is not a deep copy!

Example

copy1.cpp

```
class MyClass {
    int a;
public:
    MyClass(int i) : a(i) {
        cout << "Constructor" << endl;
    }
    MyClass(const MyClass &r) {
        cout << "Copy constructor" << endl;
        a = r.a;
    }
    void fun(int y) { a = y; }
    int get() { return a; }
};
```

- 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
```

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

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

const references

- Let's analyse the argument of the copy constructor

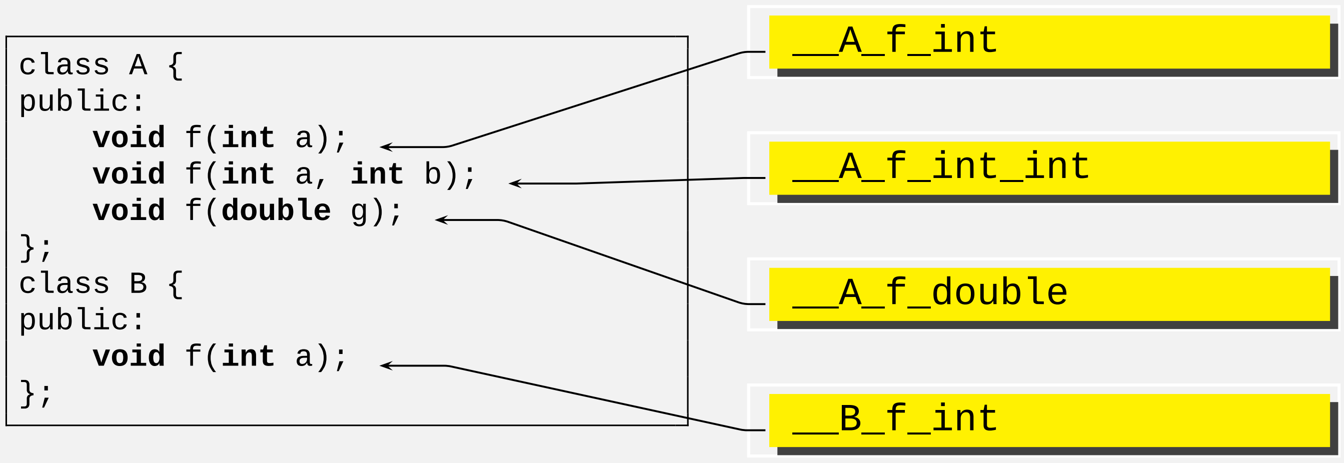
```
MyClass(const MyClass &r);
```

- The const means:
  - This function accepts a reference
  - however, the object will not be modified: it is *constant*
  - 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 the original object, the compiler will give an error

Function overloading

- In C++, the argument list is part of the name of the function
  - this mysterious sentence means that two functions with the same name but with different argument list are considered two different functions and not a mistake
- If you look at the internal name used by the compiler for a function, you will see three parts:
  - the class name
  - the function name
  - the argument list

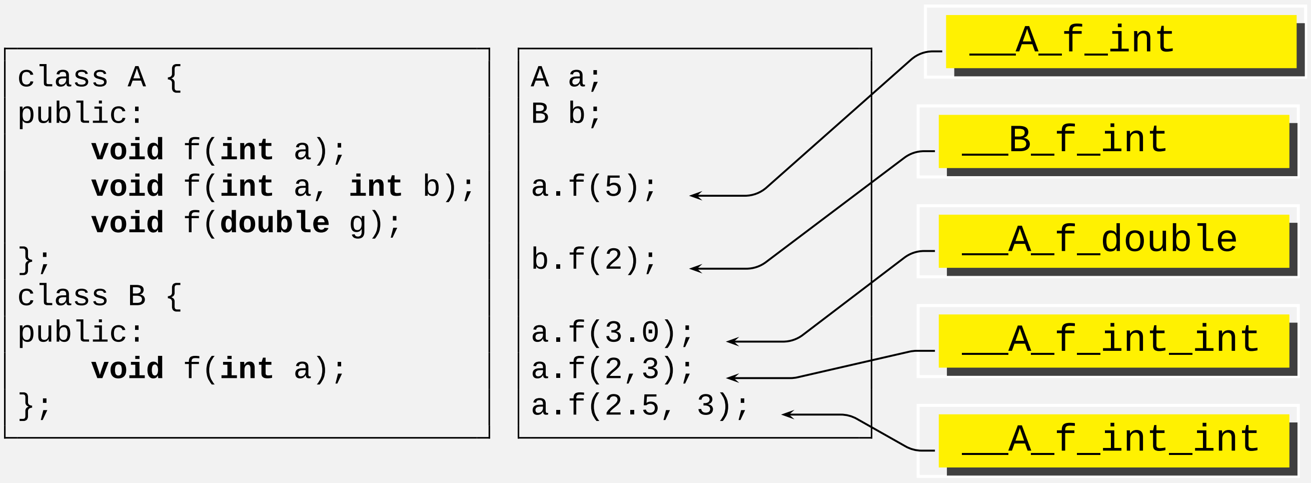
Function overloading



```
class A {
public:
    void f(int a);
    void f(int a, int b);
    void f(double g);
};
class B {
public:
    void f(int a);
};
```

- To the compiler, they are all different functions!
- beware of the type...

Which one is called?



```
class A {
public:
    void f(int a);
    void f(int a, int b);
    void f(double g);
};
class B {
public:
    void f(int a);
};
```

```
A a;
B b;

a.f(5);

b.f(2);

a.f(3.0);
a.f(2,3);
a.f(2.5, 3);
```

Return values

- Notice that return values are not part of the name
    - the compiler is not able to distinguish two functions that differs only on return values!

```
class A {
    int floor(double a);
    double floor(double a);
};
```

- This causes a compilation error
- it is not possible to overload a return value

Default arguments in functions

- Sometime, functions have long argument lists
- some of these arguments do not change often
    - we would like to set default values for some argument
    - this is a little different from overloading, since it is the same function we are calling!

```
int f(int a, int b = 0);

f(12);    // it is equivalent to f(12,0);
```

Exercise

- What happens?

overload.cpp

```
#include <iostream>

class A {
public:
//    void f(int a, int b=0);
    void f(int a);
};

void A::f(int a) {
    std::cout << "Called f(int)" << std::endl;
}

//void A::f(int a, int b) {
//    std::cout << "Called f(int, int)" << std::endl;
//}

int main() {
    A a;
    a.f(5);
}
```

Constants

- In C++, when something is declared `const`, it means that it cannot change. Period.
- Now, the specific uses of `const` are a lot:
  - Don't to get lost! Keep in mind: `const` = cannot change
- Another thing to remember:
  - constants must have an initial (and final) value!

Constants - I

- As a first use, const can substitute the use of **`#define`** in C
  - whenever you need a constant global value, use const instead of a define, because it is clean and it is type-safe

```
#define PI 3.14                    // C style

const double pi = 3.14;            // C++ style
```

- In this case, the compiler does not allocate storage for pi
- In any case, the const object has an *internal linkage*

Constants - II

- You can use `const` for variables that never change after initialisation. However, their initial value may be decided at run-time

```
const int i = 100;
const int j = i + 10;

int main()
{
    cout << "Type a character\n";
    const char c = cin.get();
    const char c2 = c + 'a';
    cout << c2;

    c2++;
// ERROR! c2 is const!
}
```

compile-time constants

run-time constants

Constant pointers

- There are two possibilities
  - the pointer itself is constant
  - the pointed object is constant

the pointer is constant

```
int a
int * const u = &a;

const int *v;
```

the pointed object is constant (the pointer can change and point to another const int!)

- Remember: a const object needs an initial value!

const function arguments

- An argument can be declared constant. It means the function can't change it
  - It's particularly useful with references

```
class A {
public:
    int i;
};

void f(const A &a) {
    a.i++;      // error! cannot modify a;
}
```

- You can do the same thing with a pointer to a constant, but the syntax is messy

Passing by const reference

- Remember:
  - we can pass argument by value, by pointer or by reference
  - in the last two cases we can declare the pointer (or the reference) to refer to a constant object: it means the function cannot change it
  - Passing by constant reference is equivalent, from the *semantic* point of view, to passing by value:
    - the original object is not modified
    - however no copy need to be made
  - From an implementation point of view, passing by const reference is much faster

Constant member functions

```
class A {
    int i;
public:
    int f() const;
    void g();
};
void A::f() const
{
    i++;
// ERROR! this function cannot
               // modify the object!
    return i;
}
```

The compiler can call only const member functions on a const object!

```
const A a = ...;

a.f();      // Ok
a.g();      // ERROR!!
```

Constant return value

- This is tricky! We want to say: “the object we are returning from this function cannot be modified”
  - This is meaningless when returning predefined types
  - Will see more on this later

```
const int f1(int a) {return ++a;}

int f2(int a) {return ++a;}

int i = f1(5);    // legal
i = f2(5);

const int j = f1(5); // also legal
const int k = f2(5); //also legal
```

these two functions are equivalent!

Class Complex

- Suppose you need to write a mathematical library, and you need to do calculations with the type *complex*
  - A complex number consists of a real part and an imaginary part
- Let’s start by writing a class that represent the data type

complex/complex.h

```
#include <iostream>

class Complex {
    double re;
    double im;

public:
    Complex();
    Complex(double a);
    Complex(double a, double b);
    Complex(const Complex &a);

    double real() const;
    double imaginary() const;
    double module() const;
};
```

Adding two complex

- Now we need to implement functions to do simple mathematical operations on objects of type Complex
  - You want to sum, subtract, multiply by a scalar, etc.
  - You want also to assign the result of these operations to other complex objects
  - To do this, in Java you need to write methods like this:

```
class Complex {
    ...
public:
    ...
    Complex add(Complex b) {
         re += b.re; im += b.im;
         return this;
    }
    ...
}
```

Operators

- In the previous case, it means that giving an object complex (`this`), you can add another object (b) to it

```
...
Complex a, b;
...
a.add(b);
```

- However, it would be more natural to just use the normal operator $+=$ to achieve the same result, as follows

```
a += b;
```

- C++ allows to do this, by using *operator overloading*

Operator oveloading

- After all, an operator is like a function
  - binary operator: takes two arguments
  - unary operator: 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

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

Complex b = 0;
b += 5;        // function call
```

Complex interface

complex2/complex.h

```
#include <iostream>

class Complex {
    double re;
    double im;

    friend std::ostream &operator<<(std::ostream& o, const Complex &a);

public:
    Complex();
    Complex(double a);
    Complex(double a, double b);
    Complex(const Complex &a);

    double real() const;
    double imaginary() const;
    double module() const;

    Complex &operator=(const Complex &a);
    Complex &operator+=(const Complex &c);
    Complex &operator-=(const Complex &c);
};

std::ostream &operator<<(std::ostream& o, const Complex &a);
const Complex operator+(const Complex &a, const Complex &b);
```

Assignment operator

- The assignment operator, `operator=()` is used to assign one value to an object
- The result of the assignment is a value (remember that an assignment is an expression!)
- This is why it needs to have a return type

```
Complex& Complex::operator=(const Complex &a)
{
  re = a.re;
  im = a.im;
  return *this;
}
```

Its parameter is a constant references (it will not be changed by the function)

In this case, it returns the same object after the assignment

Implementation of +=

complex2/complex.cpp

```
Complex& Complex::operator=(const Complex &a)
{
  re = a.re;
  im = a.im;
  return *this;
}

Complex& Complex::operator+=(const Complex &a)
{
  re += a.re;
  im += a.im;
  return *this;
}

Complex& Complex::operator-=(const Complex &a)
{
  re -= a.re;
  im -= a.im;
  return *this;
}
```

Operator $<<$

- We use the operator $<<$ to output our number on the screen
  - It takes a ostream (a class in the iostream library) and a complex, and returns a reference to a ostream.
  - cout derives from ostream, but also all the classes that implement files
  - It means that the same function is used also to output on a file
- It returns a reference to the same ostream after the operation. This allows chaining!

```
ostream& operator<<(ostream &o, const Complex &a)
{
  o << "{" << a.re << "," << a.im << "}";
  return o;
}
```

It's a global function!

Uses the standard $<<$ operator on the internal members and strings

The plus

- The + operator is slightly different:
  - it does not modify its arguments, but returns a **new value**

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

- It returns a const object
- The object is created right away (*temporary*) and will soon be destroyed
- Observe the strange use of the constructor!

Summing

- Now, let's observe what happens in this line of code:

```
Complex a, b, c;
...
a = b + c;
```

- Steps:
  - `operator+()` is called passing the references to b and c
  - The operator will create a temporary object, initialising it to the values takes from b and c
  - The assignment operator `operator=()` is called on object a, passing the temporary object by reference
  - The temporary object is destroyed

Why returning a const?

- We return a const object because we want to avoid things like this:

```
(a+b) += c;
```

- The result of the expression on the right is a temporary object, so it should not be modified
- However, we invoke a method on it (`operator+=()`), which tries to modify it
- Since we declared that the temporary is constant, the compiler will produce a compilation error
  - *error: passing 'const Complex' as 'this' argument of 'Complex& Complex::operator+=(const Complex&)' discards qualifiers*
- Try to remove the qualifier `const` and see what happens

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 global functions (they can be declared `friend` for optimising access)
- 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
- All others, YES!