Conception Objets Avancée

Operators

Giuseppe Lipari

CRIStAL - Université de Lille

Outline

Outline

Operator overloading

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

Default constructor

Constructor

Sum operator

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 for many reasons!
    - can you list them?

Usage

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

```
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

```
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

```
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

```
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

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

c3 = c1+c2;
```

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

```
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

The temp. object is destroyed

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

c3 = c1+c2;
```

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

z is copied into a temp. object

```
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

7 function calls are involved!

First 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 already saved 2 function calls
- Notice that c1 and c2 cannot be modified anyway

Temporaries

- Why the compiler builds a temporary?
  - because he doesn't know what we are going to do with that object
  - consider the following expression:

```
Complex c1(1,2), c2(2,3), c3(0,0);

c3 += c1 + c2;
```

first, the add is called

second, `operator+=` is called, passing the temporary

- the compiler is forced to build a temporary object of type Complex and pass it to `operator+=` by reference, which will be destroyed soon after `operator+=` completes

Optimization on return

- For the return object,
  - the compiler optimizes by eliminating the construction of the local variable temp
  - therefore, there is only an object that is built and then copied when assigned to the caller

Outline

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 operator []

- the prototype is the following:

```
class A {
    ...
public:
    A& operator[](int index);
};
```

- add operator [] to you Stack class
  - the operator must never go out of range

How to overload new and delete

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

- Everytime 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);
```

- An example is worth a thousands words

Example

```
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;
}
```

Outline

Inlines

- Performance is important
  - if C++ programs were not fast, probably nobody would use it (too complex!)
  - Instead, by knowing C++ mechanisms in depth, it is possible to optimize a lot
  - One possible optimizing feature is inline function

Example

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

    inline double real() const {return real_;}
    inline double imaginary() const {return imaginary;}
    inline double module() const {
        return real_*real_ + imaginary_*imaginary_;
    }
    Complex& operator =(const Complex& a);    // assignment operator
    Complex& operator+=(const Complex& a);    // sum operator
    Complex& operator-=(const Complex& a));   // sub operator
};
```

What is inlining

- when the compiler sees inline, it tries to substitute the function call with the actual code
  - in the complex class, the compiler substitutes a function call like `real()` with the member variable `real_`

```
Complex c1(2,3), c2(3,4), c3;

c1.real();
```

- we save a function call!
- in C this was done through macros
  - macros are quite bad. Better to use the inlining!
  - the compiler is much better than the pre-compiler

Excessive use of inlines

- Of course, inline function must be defined in the header file
  - otherwise the compiler cannot see them and cannot make the substitution
  - sometime the compiler refuses to make inline functions
- People tend to use inlines a lot
  - first, by using inline you expose implementation details
  - second, you clog the interface that becomes less readable
  - Finally, listen to what D.Knuth said:

  Premature optimization is the root of all evil

- So
  1. first design and program,
  2. then test,
  3. then optimize ... and test again!

Outline

Type conversion via constructor

- If you define a constructor that takes as its single argument an object (or reference) of another type, that constructor allows the compiler to perform an automatic type conversion.
- For example,

```
class One {
public:
  One() {}
};

class Two {
public:
  Two(const One&) {}
};

void f(Two) {}

int main() {
  One one;
  f(one); // Wants a Two, has a One
}
```

Another example

```
class AA {
  int ii;
public:
  AA(int i) : ii(i) {}
  void print() { cout << ii << endl;}
};
void fun(AA  x) {
  x.print();
}
int main()
{
   fun(5);
}
```

- The integer is "converted" into an object of class `AA`

Preventing implicit conversion

- To prevent implicit conversion, we can declare the constructor to be `explicit`

```
class AA {
  int ii;
public:
  explicit AA(int i) : ii(i) {}
  void print() { cout << ii << endl;}
};
void fun(AA  x) {
  x.print();
}
int main()
{
   fun(5); // error, no implicit conversion
   fun(AA(5)); // ok, conversion is explicit
}
```

Type conversion through operator Class()

- This is a very special kind of operator:

```
class Three {
  int i;
public:
  Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
  int x;
public:
  Four(int xx) : x(xx) {}
  operator Three() const { return Three(x); }
};

void g(Three) {}

int main() {
  Four four(1);
  g(four); // calls the conv. operator
  g(1);    // Calls Three(1,0)
}
```