C++

Operator overloading

Introduction

- Most languages (like C or Java) have operators only defined for built-in types
- In C++, operators are like any other functions and can be overloaded
- For example, you can redefine the operator “+” for working on your own class (for example: Matrix, String)

C++ operators

- All C++ operators can be overloaded except: “.” “::” “?:” “sizeof” “.*”
- Only existing operator can be overloaded (you can not create new operators)
- When overloading an operator you can not change: its arity, its associativity, and its precedence
- Operators are already defined for built-in types and you can not change this
- You can only overload operators for user defined classes
  - As a consequence: at least one of the arguments of the overloaded operator should be a user defined class

The Operator syntax

- In C++, “+” is only syntactic sugar for a function named “operator+”
- Same for other operators:
  - “*” -> “operator*”
  - “[]” -> “operator[]” etc
- These functions can be overloaded as any other functions
- The expression “a+b” is equivalent to the function call “operator+(a, b)”

Example of non-member operator function overloading

```
class Vector {
 private:
   float _x, _y;
 public:
   Vector(float x, float y) : _x(x), _y(y) {}
   float x() const { return _x; }
   float y() const { return _y; }
};

Vector operator+ (const Vector& a, const Vector& b)
{ return Vector(a.x()+b.x(), a.y()+b.y()); }

// …
Vector a(1.0f, 2.0f);
Vector b(1.0f, 1.0f);
Vector c = a + b;
```

Member operator function

- We can also have operators as members (methods) of a class
- They are called like other class methods with the dot syntax
- Example: the class Vector can have a method named “operator+”
  - The expression “a+b” would now be equivalent to “a.operator+(b)”
  - Be careful that to keep the arity of “+”, the method operator+ has only one argument now

Example with member operator function

```
class Vector {
 private:
   float _x, _y;
 public:
   Vector(float x, float y) : _x(x), _y(y) {}
   float x() const { return _x; }
   float y() const { return _y; }
   Vector operator+ (const Vector& b) {
     return Vector(_x + b.x(), _y + b.y());
    }
};

// …
Vector a(1.0f, 2.0f);
Vector b(1.0f, 1.0f);
Vector c = a + b;
```

Member vs non-member operator function

- If you have two objects a and b of type Vector, when the compiler sees: “a+b” it will try to look for “operator+(a,b)” or “a.operator+(b)”
- You can not define an operator as a method of a class and as an outside function if it leads to an ambiguity as above

Member vs non-member operator function: how to decide ?

- There is no rule working all the time.
- Operators “=“ (assignment), “[]” (indexing) and “()” (call) should be defined as class member (this is actually a C++ rule)
- If the left argument of the operator is an object of other type, it must be a non-member functions
- For commutative operators like “+”, “-” or “*” it is in general better to define them as non-member functions

Overloading ++ and --

- “++” and “--” are a bit different because they have a prefix and postfix form:
  - Example: int a=0; a++; ++a;
- Since they have two definitions, C++ give us two signatures to overload them
- Both are called operator++() but:
  - The prefix version takes no parameter
  - The postfix version takes a dummy “int”

Example of overloading prefix and postfix “++”

```
class Number {
private:
 int x;
public:
 Number() : x(0) {}
 Number& operator++ (); // prefix
 Number operator++ (int); // postfix
};

Number& Number::operator++() {
  x = x + 1;
  return *this;
}
Number Number::operator++ (int) {
  Number ret = *this;
  ++(*this);
  return ret;
}
```

```
// Example of usage:
//
// …
Number n1;
n1++; // calls Number::operator++(int)
++n1; // calls Number::operator++()
```

Useful example

- To finish a useful example to add some printing facilities to your class
- Note: ostream is the base class for all output stream (including cout and cerr). It is defined in the header <ostream>

```
std::ostream& operator<< (std::ostream& os, const Vector& v)
{
  os << v.x() << ", " << v.y();
  return os;
}

// …
Vector a(1.0, 2.0);
std::cout << a << std::endl; // calls operator<<(operator<<(cout, a), endl)
```