

Operator overloading: extra examples

CS319: Scientific Computing (with C++)

Niall Madden

**Week 8: some extra
examples, to supplement
what was covered in class**

- 1 Eg 1: Points in the (x, y) -plane
 - Overloading operator+
 - Overloading operator+ again
- 2 Unary operators
- 3 The Conditional Operator ?:
- 4 Prefix and Postfix ++
- 5 Overloading ++
 - Example: Date
- 6 friend functions
- 7 Another example: complex numbers
- 8 Constants
- 9 friends again

Eg 1: Points in the (x, y) -plane

We will create an object to represent points in the (x, y) -plane:

- with two private members: floats x and y ,
- constructor to initialise x and y ; it will have default values.
- a public function `Get`, used to find the values of x and y ,
- a public function `Set` that can be used to set the values of x and y ,
- a public overloaded operator `+` to add two points.

Eg 1: Points in the (x,y)-plane

Our first version of class declaration is shown opposite.

The constructor function is defined in the declaration of the class, and has default values.

01Points.cpp

```
1 class Point {  
  private:  
3   float x,y;  
  public:  
5   Point(float i=0, float j=0) { x=i; y=j;};  
   void Set(float i, float j) {x=i; y=j;};  
7   void Get(float &i, float &j) {i=x; j=y;};  
};
```

If an object of type `Point` is declared as `Point a;` then `a.Point()` is called as if it was `Point(0.0, 0.0)`.

If `a` is declared as: `Point a(-2,3);` then these two floats are passed to the constructor.

The public `Set()` allows us to set the values of `a.x` and `a.y`

In the definition of `Get()`, the integer variables are passed **by reference** and not by value – therefore the value of `i` and `j` are modified by the function.

Now we add the code for the + operator.

First, to the class definition we add the declaration of the operator:

```
Point operator+(Point b);
```

So now the class definition is

```
2 class Point {  
  private:  
    float x,y;  
4 public:  
    Point(float i=0, float j=0) { x=i; y=j;};  
6    void Set(float i, float j) {x=i; y=j;};  
    void Get(float &i, float &j) {i=x; j=y;};  
8    Point operator+(Point b);  
};
```

Notice that it seems to suggest that + takes just one argument...

And then we give the definition:

```
10 Point Point::operator+(Point b)
11 {
12     Point temp;
13     temp.x = x + b.x;
14     temp.y = y + b.y;
15     return temp;
16 }
```

The first thing to notice is that, although **+** is a binary operator, our function takes only one argument.

This is because, when we call the operator, e.g., $c = a + b$ then **a** is passed **implicitly** to the function and **b** is passed **explicitly**. Therefore **a.x** is known to the function simply as **x**.

The temporary object **temp** is used inside the object to store the result. It is this object that is return.

Neither *a* or *b* are modified..

We'll now try overloading the `+` operator again, for the purpose of introducing the `*this` pointer.

Suppose that we have a point $b = (3.1, 2.2)$. If we write $a = b + 0.5$ and mean that we want to set $a = (3.6, 2.7)$.

```
class Point
{
private:
    float x,y;
public:
    Point(float i=0, float j=0) { x=i; y=j;};
    void Set(float i, float j) {x=i; y=j;};
    void Get(float &i, float &j) {i=x; j=y;};
    Point operator+(Point b);
    Point operator+(float p);
};
```

And the rest of the code would be

```
Point Point::operator+(float p)
{
    Point temp;
    temp.x = this->x + p;
    temp.y = this->y + p;
    return temp;
}
```

or

```
Point Point::operator+(float p)
{
    Point temp(*this);
    temp.x += p;
    temp.y += p;
    return temp;
}
```

Unary operators

So far we have discussed just the **binary** operator `+`. That is, it is an example of an operator that takes two arguments.

But many operators in C/C++ are *Unary*: they take only one argument.

The most common examples of unary operators are `++` and `--`, but we'll first over load the `-` (minus) operator. Note that this can be used in two ways:

- `c = a - b` (binary)

- `c = -a` (unary).

In the second case here, “minus” is an example of a *prefix* operator. These are the easiest.

Unary operators

```
////////////////////  
// Overloading minus  
Point Point::operator-(Point b)  
{  
    Point temp;  
    temp.x = x - b.x;  
    temp.y = y - b.y;  
    return temp;  
}  
  
Point Point::operator-(void)  
{  
    Point temp;  
    temp.x = -x;  
    temp.y = -y;  
    return temp;  
}
```

The Conditional Operator ?:

C and C++ have a selection of

- unary operators, e.g., `+`, `-`, `!`, `++` and `--`.
- And binary operators, e.g., `+`, `-`, `*`, `/`, `%`, `=`, `<`, `>`, `+=`, etc.

But it has only one **ternary** operator (one that takes **3** arguments), and that is the conditional operator `?:`.

Syntax: *Cond ? Op of Cond True : Op if Cond False*

The Conditional Operator ?:

See 02Ternary.cpp for more details

```
8  int Score;
   std::string Grade, Outcome;

   std::cout << "Enter your score: ";
12  std::cin >> Score;

14  (Score >= 40) ? Grade="Pass" : Grade="Fail";
   std::cout << "You have " << Grade << "ed." << std::endl;

   // Alternative
18  std::cout << "Enter another score: ";
   std::cin >> Score;
20  Outcome = (Score >= 40) ? "will not" : "will";
   std::cout << "You " << Outcome << " have to come back in August!"
22          << std::endl;
```

Prefix and Postfix ++

We all know that in C/C++ one can use the ++ and -- operators in prefix and post fix forms, e.g.,

- ++a (prefix)
- a++ (postfix)

But what is the difference?

In C/C++, all expressions – including assignments such as a++ – must evaluate as something. So

- ++a evaluates as a+1, the value *after* incrementation,
- a++ evaluates as a, the value *before* incrementation.

```
2 int a, b;  
  a=1;    b=(a++);  
  std::cout << "a=" << a << ", b=" << b << endl;  
  
  a=1;    b=(++a);  
6  std::cout << "a=" << a << ", b=" << b << endl;
```

output

a=2, b=1

a=2, b=2

Prefix and Postfix ++

And there is a further distinction between the preincrement (**++a**) and postincrement (**a++**) operators: the preincrement operator can be used as the left operand in an assignment.

```
// Use prefix and postfix ++
2 #include <iostream>
  int main(void)
4 {
    int a, b;

    a=10; b=20;
8 // The following would be illegal
  // (a++)=b;
10  (++a)=b; // <- But this is OK!
    std::cout << "a=" << a << ", b=" << b << std::endl;
12  return(0);
}
```

Overloading ++

Overloading both the prefix and postfix versions of ++ involve a number of technicalities, specifically

- (1) There has to be a way of distinguishing between the two forms. This is achieved by giving the postincrement operator a dummy `int` as an argument. That way the two versions have different **signatures**.
- (2) We may wish to maintain the usual way the operators work (i.e., whether they evaluate as the original or incremented version, and if they can be used as **left values** in assignment operations). This is achieved as follows:
 - (a) The postincrement operator makes a copy of the object before the increment, and then returns the (original, unincremented) copy.
 - (b) Typically, preincrement operator will return a reference (address), and the postincrement operator will return a value. This allows the preincrement op to be used as the left value in an assignment.

The example we'll use to study this is a class to implement the current date.

03Date++.cpp

```
1 // CS319 Week 9 (extras)
2 // Example of overloading the ++ operator
3 // Implemented class represents a calendar date
4 #include <iostream>
5
6 int StandardDaysMonth[] =
7     {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
8
9 class Date {
10 public:
11     Date (int d=1, int m=1, int y=1901);
12     void SetDate(int dd, int mm, int yy);
13     int GetDay(void) {return day;};
14     int GetMonth(void) {return month;};
15     int GetYear(void) {return year;};
16     bool IsLeapYear(void);
17
18     Date &operator++(void); // prefix version
19     Date operator++(int Dummy); // postfix
20
21 private:
22     int day, month, year;
23     int *DaysMonth;
24     void Increment(void);
25 };
```

```
28 // Constructor
   Date::Date(int d, int m, int y)
30 {
    DaysMonth = new int [13];
32     for (int i=0; i<=12; i++)
        DaysMonth[i] = StandardDaysMonth[i];
34     SetDate(d,m,y);
    }

38 bool Date::IsLeapYear(void)
   {
40     if ( (year%400 == 0) ||
        ( (year%4 == 0) && (year%100 != 0)))
42         return true;
        else
44         return false;
    }
```



```
48 void Date::SetDate(int d, int m, int y)
   {
       year=y;
50     if (IsLeapYear())
           DaysMonth[2]=29;

       month = ( (m>=1) && (m<=12) ) ? m : 1;

       day = ( (d>=1) && (d<=DaysMonth[month]) ) ? d : 1;
56 }
```

```
58 // Will call this in both prefix and postfix versions.
void Date::Increment(void)
60 {
    if (day != DaysMonth[month])
62         day++;
    else if ( month != 12 )
64     {
        month++; day=1;
66     }
    else
68         SetDate(1,1,year+1); // Need to set the leap year
}
```

```
72 Date &Date::operator++(void) // Prefix version
73 {
74     Increment();
75     return(*this);
76 }
77
78 Date Date::operator++(int Dummy) // Postfix version
79 {
80     Date temp=*this;
81     Increment();
82     return(temp);
83 }
```

```
84 int main(void)
85 {
86     Date today(01,03,2017);
87     Date tomorrow, NextMonday;

88     cout << "Today is " << today.GetDay() << "/" <<
89         today.GetMonth() << "/" << today.GetYear() << endl;

90     tomorrow=(++today);
91     cout << "Tomorrow is " << tomorrow.GetDay() << "/" <<
92         tomorrow.GetMonth() << "/" << tomorrow.GetYear() << endl;

93     for (int i=1; i<5; i++)
94         ++today;

95     NextMonday = today;
96     cout << "Next Monday is " << NextMonday.GetDay() << "/" <<
97         NextMonday.GetMonth() << "/" << NextMonday.GetYear() << endl;

98     cout << endl;

99     return(0);
100 }
```

friend functions

In all the examples that we have seen so far, the only functions that may access private data belonging to an object has been a member function/method of that object.

However, it is possible to designate non-member as being **friends** of a class.

For non-operator functions, there is nothing that complicated about **friends**. However, care must be taken when overloading operators as **friends**.

In particular:

- All arguments are passed explicitly to **friend** functions/operators.
- Certain operators, particularly the **Put to <<** and **Get from >>** operators can only be overloaded as friends.

04DatePutTo.cpp

```
1 class Date // New version to overload <<
2 {
3     friend std::ostream &operator<<(std::ostream &, const Date &);
4
5     public:
6         Date (int d=1, int m=1, int y=1901);
7     .
8     .
9     .
10    std::ostream &operator<<(std::ostream &output, const Date &d)
11    {
12        std::string MonthName[13]={ "", "Jan", "Feb", "Mar", "Apr",
13        "May", "Jun", "Jul", "Aug", "Sep",
14        "Oct", "Nov", "Dec"};
15        output << d.day << "/" << MonthName[d.month] << "/"
16        << d.year;
17        return(output);
18    }
```

Another example: complex numbers

In order to increase the number of examples of classes that you have studied, and to give a framework to consider some technical issues, we introduce a `class` for **Complex Numbers**.

A **complex number** is

$$z = a + bi \text{ where } a \text{ and } b \text{ are real numbers and } i = \sqrt{-1}.$$

So our class will have two private data elements for the **real** and **imaginary** parts, as well as methods to get and set their values.

Another example: complex numbers

05Complex.cpp

```
class Complex
{
private:
    float real, imag;
public:
    Complex (float r=0.0, float i=0.0) {Set(r,i);};
    float GetReal(void) {return(real);};
    float GetImag(void) {return(imag);};
    void Set(float r, float i=0.0);
};

void Complex::Set(float r, float i)
{
    real=r; imag=i;
}
```

Constants

Constant Variables: We are familiar with the idea of a `const` “variable”. This is one whose value can not be changed after initialisation.

Constant Functions: In `C++` one may also force a class method to be constant. This is done by placing the keyword `const` at the end of the line containing the function prototype and header. The effect is that the function cannot modify the object.

Furthermore, if you wish to call a method belonging to a constant object, then the method must be constant.

In our example, we might define `const Complex I(0,1)` to represent $i = \sqrt{-1}$. But a call to `I.GetReal()` or `I.GetImag()` as presented above would give an error, such as

```
error: passing 'const Complex' as 'this' argument of  
'float Complex::GetImag()' discards qualifiers
```


Constants

```
class Complex
{
    private:
        float real , imag;
    public:
        Complex (float r=0.0, float i=0.0) {Set(r,i);};
        float GetReal(void) const;
        float GetImag(void) const;
        void Set(float r, float i=0.0);
};

float Complex::GetReal(void) const
{
    return(real);
}
float Complex::GetImag(void) const
{
    return(imag);
}
```

friends again

Let's return to the idea of a `friend` function. This is a function that does not belong to the class, but still has access to private elements of a class.

There are two reasons you might want a function to be a `friend` of a class

- **Efficiency:** If the called function is not a `friend`, it will have to use some method to access private members. This can be slower than being able to access it directly.
- **Overloading:** When overloading an member operator for a class, the left argument must be an object of that class. So, for example,

```
Complex x(2.0, 3.0), z;  
float f=3.0;  
z = x*x; // * operator can be a member  
z = x*f; // * operator can be a member  
z = f*x; // * operator can't be a member
```

friends again

It is because the left argument of an overloaded operator must be an object of the class, that we can only overload the **stream insertion** operator `<<` and **stream extraction** operator `>>` as friends.

To see how this is done, have a look at [06ComplexMult.cpp](#)

This example also shows the three ways of overloading the `*` operator.

Note that,

- for a member operator, the left argument is implicit, and only the right argument is listed explicitly,
- for a non-member function, both the left and right arguments must be listed.