CPS235 Object Oriented Programming in C++

LAB6 BESE-15B *1st April 2010*

Operator Overloading in C++

Objectives

By the end of this lab, you should be able to understand how to overload different unary and binary operators in C++

What is Overloading

Overloading is a component of polymorphism in C++. It lets a programmer use one function name for multiple functions. The compiler chooses what implementation to use based on the arguments provided in the call. Functions can be overloaded to handle different types of data and different numbers of parameters. You can also overload certain operators, including +, /, <<, and even [ ]. This can add power and ease of use to user defined data types.

Function Overloading

Briefly, ***function overloading*** means that two or more functions share the same name but their parameters are different. In this situation, the functions that share the same name are said to be ***overloaded*** and the process is called function ***overloading.*** The number and types of a function's parameters are called the function's ***signature***. Together a function's name and its signature uniquely identify it.

Operator Overloading

You can overload operators in the same way that you overload functions. There are only three things you need to know:

1. Can this operator be overloaded?
2. How many parameters does the operator require (1 - unary, 2 - binary)?
3. Where do I overload the operator (as class member functions or non-member functions)?

1. Which operators can be overloaded?

Once you know whether you can overload an operator you name the overloaded function by adding `operator` infront of the symbol. For example to overload + you would use the function name `operator+`.

The following table is the predefined set of C++ operators that may be overloaded:

TABLE OF OVERLOADABLE OPERATORS

+	-	*	/	%	^	&	|	~
!	,	=	<	>	<=	>=	++	--
<<	>>	==	!=	&&	||	+=	-=	/=
%=	^=	&=	|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]			

You cannot overload **`::`** , **`.*`** , **`.`** **or** **`? :`**

NOTE: the point of operator overloading is to be intuitive. You do not want to assign a confusing operator name. For instance, you do not want to overload the "+" when the function actually performs subtraction.

2. Number of Parameters

The number of parameters an operator takes depends on the operator and where you overload it. We will start with operator overloading outside of a class.

2.1 Non-member Operator Overloading

Most operators you can overload are either binary or unary. Binary operators have a left-hand side and a right-hand side usually written ***LHS*** and ***RHS***. For non-member functions each side is a parameter. The LHS is the first parameter, the RHS is the second.

```
return_type operatorbinary(type LHS, type RHS)
```

Unary operators have only one parameter.

```
return_type operatorunary(type param)
```

Some operators can be both binary and unary. Which overloaded definition is used is determined by context and function signature. Suppose you wanted to overload subtraction and negation. Suppose we have some Fraction class objects declared.

```
fraction f1(10,10), f2(5,3), f3, f4;
```

These expressions:

```
f3 = f1 -f2; //subtraction
-f1         // negation
```

are transformed into function calls like these by the compiler:

```
operator-(f1,f2);     //subtraction
operator-(f1);        //negation
```

The following example shows how you would overload operators + using the non-member function:

```
class fraction
{
  private:
      int numer, denom;
  public:
    fraction(int n = 0, int d = 1);
    friend fraction operator+(const fraction&,const fraction&);
};

fraction::fraction(int n, int d)
{
  numer = n;
  denom = d;
}

// fraction friend function
fraction operator+(const fraction& f1,const fraction& f2)
{
fraction temp(f1.numer*f2.denom + f2.numer*f1.denom,f1.denom * f2.denom);
  return temp;
}
void main()
{
  fraction f(3,4);
  fraction g(2,3);
  fraction h = f + g;
}
```

Note that non-member functions do not have access to private data. C++ has the keyword ***friend*** you can use as a workaround to this restriction.

2.2 Member Operator Overloading

When you overload an operator as a member function, the LHS of binary operations and the only parameter of unary operations is implied. That parameter is assumed to be the calling object. If subtract and negate were defined as member operators of the fraction class then

`f1 - f2` would become `f1.operator-(f2)`

whereas

`-f1` would become `f1.operator-()`

The changes you need to overload the non-member operators + above as member operators follow. The function prototype becomes

```
fraction operator+(fraction f);
```

Now change the function definitions as follows

```
fraction operator+(const fraction& f1,)
{
      fraction temp(f1.numer * denom + numer *f1.denom,f1.denom * denom);
      return temp;
}
```

Notice that a member function has access to the private data members of parameters that belong to the same class.

3. Where to overload an operator

Usually it is up to you to decide where to overload an operator. Sometimes you have no choice; you are required by some limitation to define an operator as a class member or as a non-member.

Forced Non-Member Operator Overloading

You must use a non-member implementation when the LHS is either not an object (int, double) or you do not have access to its class definition (cin, strings).

Forced Member Operator Overloading

Assignment =, subscript [ ], call (), and member access (-> and ->*) operators **must** be defined as member functions.

x----------x----------x----------x

Some programmers prefer to define operators on object/non-object pairs as non-member functions.

```
#include <math.h>
//……other header files
class twoD{
      private:
            int x;
            int y;
      public:
      //...constructors and other member functions
      friend bool operator<(twoD, int);
      friend bool operator<(int a,twoD);
};

bool operator<(twoD b, int a)
{
   if (sqrt(b.x*b.x+b.y*b.y) < a) return true;
   return false;
}
```

```
bool operator<(int a, twoD b)
{
   if (sqrt(b.x*b.x+b.y*b.y) > a) return true;
   return false;
}
```

Exercise

We need to write an application that reads two times (times are represented by two integers: hours and minutes).
Then the program finds out which time is later time. After that it calculates the time difference between these times. Finally the program displays the smaller (earlier) time and the time difference (duration) in the format

```
starting time was 11:22
duration was     1:04
```

The main function that does these things looks as follows:

```
int main(void) {
    Time time1, time2, duration;
    time1.read("Enter time 1");
    time2.read("Enter time 2");
    if (time1.lessThan(time2)) {
      duration = time2.subtract(time1);
      cout << "Starting time was ";
      time1.display();
    }
    else {
      duration = time1.subtract(time2);
      cout << "Starting time was ";
      time2.display();
    }
    cout << "Duration was ";
    duration.display();
}
```

Now you need to define and implement class Time so that the program becomes working. As can be seen from the main function, Time has the following member functions:

```
read that is used to read time (minutes and hours) from the keyboard.
lessThan that is used to compare two times.
subtract that is used to calculate time difference between two times.
display that is used to display time in the format hh:mm.
```

Develop the program further so that you implement the following operators for the class

```
< (less than)
- (subtract)
<< (output) and
>>(input)
```

After overloading these operators, the main function should look like:

```
void main() {
    Time time1, time2, duration;

    cout << "Enter time 1";
     cin >> time1;
    cout << "Enter time 2";
     cin >> time2;
     if (time1 < time2) {
      duration = time2 - time1;
      cout << "Starting time was " << time1;
    }
    else {
      duration = time1 - time2;
      cout << "Starting time was " << time2;
    }
    cout << "Duration was " << duration;
}
```

Add pre-increment and post-increment operators for your time class (operators add one minute to the time). Test the operators using the following simple test program:

```
void main() {
    Time time;
    cout << "Enter time";
       cin >> time;
       cout << time++ << endl;
    cout << ++time << endl;
}
```