CS115 Lab: C++ Overloading

Highlights of this lab:

In this lab you will learn how to overload functions and operators. You will be able to recognize and write overloaded functions and you will be able to define your own operators. Specifically, the following topics will be covered:

- What is Overloading
- Function Overloading
- Operator Overloading

Lab Exercise:

- Answer questions about overloading
- Complete a program by:
  - identifying and working with overloaded functions
  - writing and testing some overloaded operators

Click the little computer above for a detailed description.
NOTE: Your lab instructor will tell you what will be marked for this lab.

1. 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.

2. Function Overloading

Briefly, ***function overloading*** means 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.

2.1 Parameter Type Overloading

This refers to the situation where the overloaded functions have the ***same number*** of parameters, but the ***types*** are ***different***.

```
// File name: /pub/class/115/ftp/cpp/inheritance/OverloadSingle.cpp
// Purpose:   Demonstrate single parameter overloading

//Overloaded functions with same number of parameters, but
//different types.
void timesTwo(int &num);
void timesTwo(double &num);

int main(void)
{
  int    A = 1;
  double B = 1.1;

  timesTwo(A); // Changes A to 2
  timesTwo(B); // Changes B to 2.2
}

// handle int type
void timesTwo(int &num)
{
  num = num * 2;
}

// handle double type
void timesTwo(double &num)
{
  num = num * 2.0;
}
```

In the above example, there are two versions of the `timesTwo` function; one deals with integers and the other deals with double. When we call the function with different data types, the compiler is able to decide which version of the `timesTwo()` function to call by comparing the type of the argument in the function call with the type of the parameter of the available function versions. If the call argument is of type `int` then the function version with an `int` parameter is called. Similarly if the call argument is of type `double` then the function version with a `double` parameter is called.

2.2 Parameter Number Overloading

This refers to the situation where the types of parameters of the overloaded functions may or may not be the same and the ***number*** of parameters is ***different***.

Let's look at the following example:

```
// File name: /pub/class/115/ftp/cpp/inheritance/OverloadMultiple.cpp
// Purpose:   Demonstrate multiple parameter overloading

#include <iostream>
using namespace std;

void add(int i, int j);
void add(int i, double j);
void add(int i, int j, int k);

int main(void)
{
  int    A = 1, B = 2, C = 3;
  double D = 1.1;

  add(A, B); //  i + j => add prints 3
  add(A, D); //  i + 1.1 => add prints 2.1
  add(A, B, C); // i + 2 + 3 => add prints 6
}

void add(int i, int j)
{
      cout << "Result: " << i + j << endl;
}
void add(int i, double j)
{
      cout << "Result: " << i + j << endl;
}
void add(int i, int j, int k)
{
      cout << "Result: " << i + j + k << endl;
}
```

The above program gives the following results:

```
barcolay[9]% g++ -o OverloadMultiple OverloadMultiple.cpp
barcolay[9]% ./OverloadMultiple
Result: 3
Result: 2.1
Result: 6
barcolay[10]%
```

2.3 Overloading Member Functions

Nothing special is required to overload member functions. You simply need to declare and define member functions with the same name but different signatures. Consider the following example:

```
//in point.h
class Point
{
  private:
   int coord[2];  //represents x and y

  public:
    .
    .
    .
   //add scalar to "coord"
   void addCoord(int);
   //add another coordinate to "coord"
   void addCoord(const Point &);
    .
    .
    .
//in matrix.cpp
void Point::addCoord(int a)
{
...
}
void Point::addCoord(const Point & otherCoord)
{
...
}
```

How do you identify a member function from a non-member function?

3. Operator Overloading

The idea of operator overloading is that you create a special named function with the `operator` keyword infront of the symbol. For example to overload `+` you would use the function name `operator+`.

The following table (taken from *C++ Primer*, Lippman and Lajoie) 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.

Review Unary and Binary

3.2 Overloading ==

Let's say you have the `Date` class that you created in a previous lab. You want to write:

```
if (date1 == date2)
      cout << "The dates are the same";
```

In the following subsections, we will look at two ways of implementing this

3.2.1 Overloading As a Member Function

You can modify the `sameDay` function to the following:

```
bool Date::operator==(const Date& myDate) const
{
   if (month == myDate.month && day == myDate.day && year == myDate.year)
      return true;
   else
      return false;
}
```

Don't forget to modify the prototype in the `Date.h` file. Change `sameDay` to `operator==`.

The compiler does something behind the scenes for you. When it sees `date == date2`, it checks if `operator==` exists as a member function. If it does, then the compiler writes:

```
date1.operator==(date2)  //the compiler automagically writes this for you
```

You can actually write this in the code, but wouldn't you agree that `date1 == date2` is more intuitive and easy to read. Thus, the point of operator overloading!

3.2.2 Overloading As a Non-member Function

Above, we overloaded the `==` as a member function. We could have chosen to define the `operator==` as a non-member function with two arguments. What would that look like?

```
bool operator==(const Dates myDate1, const Dates myDate2)
{
   if (myDate1.getMonth() == myDate2.getMonth() &&
       myDate1.getDay() == myDate2.getDay() &&
       myDate1.getYear() == myDate2.getYear())
       return true;
   else
       return false;
}
```

What are the differences?

- No scope resolution (`::`)
- Two arguments instead of one
- Using the getters instead of accessing `month`, `day` and `year` directly

Why these differences?

This time, when the compiler sees date1 == date2 in main, it will determine that there is a non-member function and rewrite the code as:

```
operator==(date1,date2) //the compiler automagically writes this for you
```

3.3 Overloading ++

Let's say you have the `Date` class that you created in a previous lab. You want to write:

```
++date1
```

This will increment the current date1 object (in essence adding one to the day). In the following subsections, we will look at two ways of implementing this.

3.3.1 Overloading As a Member Function

```
Date Date::operator++()
{
      if (day == endOfMonth())  //endOfMonth returns 28, 29, 30 or 31 depending on month
      {
            if (month == 12)
            {
                  year += 1;
                  month = 1;
            }
            else
                  month += 1;
            day = 1;
      }
      else
            day += 1;
      return *this;
}
```

Why are there no arguments in this approach to overloading?

What would the compiler write automagically for you?

3.3.2 Overloading As a Non-Member Function

```
Date operator++(Dates myDate)
{
      if (myDate.getDay() == myDate.endOfMonth())
      {
            if (myDate.getMonth() == 12)
            {
                  myDate.setYear(myDate.getYear()+1);
                  myDate.setMonth(1);
            }
            else
                  myDate.setMonth(myDate.getMonth()+1);
            myDate.setDay(1);
      }
      else
            myDate.setDay(myDate.getDay()+1);

      return myDate;
}
```

What are the differences? Why?

3.4 Do I overload the operator as a non-member or member function?

To summarize, sometimes you have a choice of defining the overloaded operator function as a member function or a non-member function. It is a matter of preference as to which you find more comfortable and easy. However, there are two situations where you are forced to use only one of the two options:

When the left-hand side is not an object (for instance, when the left-hand side is a integer) such as:

- `1 + date1`

you must define it as a **non-member function**

For the following operators, you can only define them as **member functions**

- Assignment `=`
- Class member access `->` or `->*`
- Subscripting `[ ]`
- Function call `( )`

You will get a compiler error if you try and overload these operators as non-member functions.

Some programmers prefer to define operators on object/non-object pairs as non-member functions. This way the definition that cannot be a member is defined similarly to the one that can like this:

```
#include <math.h>
//...

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;
}
```

4. Lab Exercise — Overloading

Part 1

1. Change the `sameDay` function into a overloaded == operator. This will be a **member** function as demonstrated in the notes. Remember that you will want to compare the month, day, and year.
2. Add an overloaded ++ operator (as a **member** function, shown in the notes)
3. Just for fun, change the `printDate` function into an overloaded << operator as a non-member. The prototype in the .h will look like this:

```
ostream& operator<<(ostream&, const Dates&);
```

   The function itself will look like this:

```
ostream& operator<<(ostream& os, const Dates& myDate)
{
      os << "Date is: " << myDate.getMonth()
         << "/" << myDate.getDay()
         << "/" << myDate.getYear() << endl;
      return os;
}
```

4. Explain why `operator<<` must be overloaded as a non-member.

Part 2

Overload the + operator so that the following lines of code run in main:

```
      Date date2, date4;
      date3 = date1 + 92;
      date4 = 5 + date2;
```

More Details:

- Define two functions. The prototypes will look like this:

```
Date operator+ (const Dates, int);
Date operator+ (int, const Dates);
```

  Are they member or non-member functions? (Notice how many arguments they have)

- Both functions will create a temporary date and return it.
- Think about reusing as much code as possible.
  a. For one implementation of operator+, can you call the operator++ function?
  b. For the second implementation of operator+, can you call the first operator+ function?
  c. Together these two functions should be no more that 10 lines of code in the bodies. If you make it longer, you are working too hard.
- Which of the two functions **must** be defined as a non-member? Why?

Sample Run (after Part 2)

```
a.out
Date is: 0/0/0
Please enter integer month, day, and year separated by spaces:  1 1 1
Please enter integer month, day, and year separated by spaces:  2 2 2
Date is: 1/1/1
Date is: 2/2/2
The dates are different.
Before addition
Date is: 1/2/1
Date is: 2/2/2
After addition
Date is: 1/2/1
Date is: 2/2/2
Date is: 3/2/1
Date is: 2/3/2
```