CMPT 225 - Lab 4 - Operator Overloading

Lab Objectives

In our Lab 3, we saw how to design and implement a data collection ADT class that used a concrete data type (CDT) made of dynamically allocated memory and provided most of the necessary methods. We did say "most" because we still needed to add the overloaded assignment (=) operator to our List ADT class.

In this lab, we shall explain how to overload operators. This will allow us to complete our List class from Lab 3 and do Assignment 2 by overloading assignment (=) operators.

What does *overloading* mean?

Overloading is a mechanism by which a programming language, such as C++, allows a function, a method, or an operator to have different behaviours in the same scope.

Constructors are examples of overloaded methods. One can define several constructors within the same class (scope), and each of these constructors would differ in the number and data type of their parameters and in their behaviour, constructing objects of the class differently. For example, one can define the following constructors for the Profile class:

```
// Default Constructor
// Description: Creates a Profile object.
// Postcondition: userName, name, email and birthday set to defaultStrValue.
Profile::Profile() : userName(defaultStrValue), name(defaultStrValue), email(defaultStrValue), birthday(defaultStrValue) { }

// Parameterized Constructor
// Description: Creates a Profile object with the given userName.
// Postcondition: If aUserName is not valid, then the member attribute userName it set to defaultStrValue.
//                All other member attributes are set to defaultStrValue.
Profile::Profile(string aUserName) : name(defaultStrValue), email(defaultStrValue), birthday(defaultStrValue) {

  this->setUserName(aUserName);
}

// Parameterized Constructor
// Description: Creates a Profile object with the given userName, name, email and birthday.
// Postcondition: If aUserName is not valid, then the member attribute userName is set to defaultStrValue.
//                All other member attributes are set to the given parameters, respectively.
Profile::Profile(string aUserName, string aName, string anEmail, string aBirthday) :
                 name(aName), email(anEmail), birthday(aBirthday) {

  this->setUserName(aUserName);
}
```

where each constructor constructs an object of the Profile class differently, i.e., initializing the data members of the Profile class with different values by using the given parameters and **defaultStrValue**.

The "+" operator is an example of an overloaded operator. One can implement different behaviours for this operator depending on the type of its operands (integers, floats, strings, Profile objects, etc.). For example, the compiler already knows how to add two integers and how to add two floats, but we need to tell it how to add two Profile objects by overloading the "+" operator for the Profile class.

In general, when you call an overloaded function, method, or operator, the compiler determines the most appropriate definition (i.e., implementation) to execute, by comparing the argument types (or operand types, in the case of an operator) you have used to call the function, method or operator with the parameter types specified in their prototype (or header or declaration). The process of selecting the most appropriate overloaded function, method, or operator is called **overload resolution** process.

Operator Overloading: overloading operators ==, <, and >

Even though we can overload functions, methods, and operators, in this lab, we shall focus on the overloading of operators.

Let's first have a look at the way Profile class from our Assignment 1 overloads the operators ==, <, and >:

```
// Overloaded Operators
// Description: Comparison (equality) operator. Compares "this" Profile object with "rhs" Profile object.
//              Returns true if both Profile objects have the same userName.
bool Profile::operator==(const Profile & rhs) {

    // Compare both Profile objects
    return ( this->userName == rhs.getUserName() );

}

// Description: Greater than operator. Compares "this" Profile object with "rhs" Profile object.
//              Returns true if the userName of "this" Profile object is > the userName of "rhs"
//              Profile object, i.e., the userName of "this" Profile object goes after
//              the userName of "rhs" Profile object when ordered in ascending alpha order.
```

```
bool Profile::operator > (const Profile & rhs) {

  return ( this->userName > rhs.getUserName() );
}

// Description: Lesser than operator. Compares "this" Profile object with "rhs" Profile object.
//              Returns true if the userName of "this" Profile object is < the userName of "rhs"
//              Profile object, i.e., the userName of "this" Profile object goes before
//              the userName of "rhs" Profile object when ordered in ascending alpha order.
bool Profile::operator < (const Profile & rhs) {

  return ( this->userName < rhs.getUserName() );
}
```

Having overloaded these three operators for the Profile class allows the client code to ascertain whether an object of the Profile class is equal (==), smaller (less) than (<), or greater than (>) another object of the Profile class, based on the lexicographic order (or, in plain English, alphabetical order) of their search key (i.e., **userName** attribute).

When our code makes use of these operators, as we have in the following code fragment taken from the method **search(...)** of the MyADT class:

```
...

  // Get index of array in which target may be found.
  index = target.getSearchKey() - 'a';

  // Go directly to array of elements with same first letter as target's and ...
  if ( elements[index] != nullptr && elementCount[index] > 0 ) {
    for (unsigned int eachMember = 0; eachMember < elementCount[index] && !found; eachMember++) {

      // ... search for target - see Profile class overloaded operator ==
      if ( elements[index][eachMember] == target ) {
        result = &elements[index][eachMember];
        found = true;
      }
    }
  }

...
```

the left and the right operands of the operator == in **elements[index][eachMember] == target**, as seen above, are objects of the Profile class.

The fact that the left operand is an object of the Profile class allows us to overload these three operators as **methods** (public methods) of the Profile class because this Profile object (i.e., the left operand) is the one "calling" this method. Using our example above, **elements[index][eachMember]** is calling the overloaded operator == of the Profile class. (Well, for an operator, "calling this method" means "being the left operand of this operator".) Furthermore, this left operand is represented by the pointer **this** within the implementation (overloading) of the operator ==, as you can see in the corresponding overloaded operator == code above, where **this**, in **this->userName**, refers to the Profile object **elements[index][eachMember]** when the code fragment **elements[index][eachMember] == target** in the **search(...)** method above is executed.

The right operand **target** in **elements[index][eachMember] == target** is passed to the overloaded operator as an argument, more specifically, as the argument **const Profile & rhs**, where **rhs** stands for **right-hand side**, i.e., the right-hand side operand of the operator.

Operator Overloading: overloading operator <<

Let's now have a look at the overloaded **stream insertion** << operator from the Profile class from our Assignment 1. Here is its prototype:

```
// Description: Prints the content of "this" Profile object as follows:
//              <userName>, <name>, <email>, born on <birthday>
friend ostream & operator<<(ostream & os, const Profile & p);
```

and here is its implementation:

```
// Description: Prints the content of "this" Profile object as follows:
//              <userName>, <name>, <email>, born on <birthday>
ostream & operator<<(ostream & os, const Profile & p) {

    os << p.userName << ", " << p.name << ", " << p.email << ", born on " << p.birthday << endl;
    return os;

}
```

Notice the differences between the way this operator has been overloaded and the way the other operators (==, <, and >) have been overloaded.

Indeed, overloading the << operator is a little more complicated than overloading the ==, the <, and the > operators. There are a couple of reasons for this.

The first reason is that the << operator is itself an overloaded operator (overloaded by the **ostream** class). In theory, we could get access to **iostream.h** and **iostream.cpp** and add code to these files so that **cout** recognizes objects of Profile class type, but this is not a good idea because we would have to do this for every new class we create. Instead, we write a function that overloads the << operator for the Profile class - a much more **OO** (Object-Oriented) way of achieving our goal of printing the content of a Profile object (for testing purposes).

The second reason why overloading the << operator is a little more complicated is because its left operand is **not** an object of the Profile class but an object of the ostream class:

```
cout << elements[index][eachMember] << endl;
```

Notice the left operand of the leftmost << operator in the above output statement is **cout**, i.e., an object of the ostream class, not an object of the Profile class.

Question: What is the effect of having the left operand not being an object of the Profile class?

Answer: The effect is that we cannot "call" this operator using an object of the Profile class as we do when we call a method of the Profile class. In other words, we cannot overload this operator as a method of the Profile class as we did with the other three operators. Instead, we need to implement it as a **friend** function of the Profile class as opposed to a method of this class. And we do this by prepending the keyword **friend** to the prototype in the header file of the Profile class (Profile.h) as you can see in the stream insertion << operator's prototype above.

This keyword indicates that the function is designated as a friend of the Profile class and as such has access to its private data members without being part of the class itself. Since the friend function is **not a member method** of the Profile class, we do not have to precede the function header with **Profile::** when we implement it in the **Profile.cpp** file and we cannot use the pointer **this** in its body (i.e., implementation).

Question: Why does this friend function have 2 parameters?

In order to answer this question, let's have another look at the way this operator is called:

```
cout << elements[index][eachMember] << endl;
```

Notice again its left operand: **cout**. Since **cout** is not an object of the Profile class, it must be passed to the overloaded << operator as an argument. In this example, notice that **cout** matches the first parameter of the friend function in its declaration above, i.e., **ostream & os**. Note the name change. In general, arguments can be named differently than their matching parameters. The right (i.e., second) operand (**elements[index][eachMember]**) in the example above is an object of the Profile class and matches the second parameter, **const Profile & p**, of the friend function in its declaration above.

Question: Why does this friend function return **ostream &**?

Answer: Remember that we can chain the << operator as illustrated in the following example:

```
cout << "Here is a new Member to our FriendsBook: " << newProfile << endl;
```

Such chained statement is read from left to right such that the example above is equivalent to:

```
((cout << "Here is a new Member to our FriendsBook: ") << newProfile) << endl;
```

The above statement executes as follows: first, the expression **cout << "Here is a new Member to our FriendsBook: "** executes. This is to say: the << operator overloaded by the **string** class executes and returns an **ostream** object. This object becomes the left operand of the second section of the statement above, i.e., **<< newProfile;**. Our friend function from the Profile class is then executed since the left operand is an ostream object and the right operand is an object of the Profile class. Remember, the overload resolution process would match this "calling" of the << operator, having these two arguments, with the prototype in the Profile class **ostream & operator<<(ostream & os, const Profile & p)** as its two parameters match the two arguments. After printing the content of the newProfile object, it returns an **ostream** object, which becomes the left operand of the rest of the **cout** statement, i.e., **<< endl;**. Finally, this last section of the output would be executed, adding a new line to the output.

So, why does this friend function return **ostream &**? Because our overloaded version of this << operator must behave as expected, i.e., it must return an **ostream** object so that our overloaded version of the << operator can also be chained as expected (and as illustrated in the example above).

Operator Overloading - Let's practice!

Exercise 1

Let's practice overloading the << operator by modifying our MyADT class from Assignment 1. First, let's copy our entire Assignment 1 directory into a new **Lab4** directory since we do not want to alter the code we wrote for our Assignment 1 before we submit it. Then **cd** into this **Lab4** directory and replace MyADT's **print( )** method with a friend function overloading the << operator.

As you are implementing this friend function, remember that the word **friend** is not used as part of the function header in the implementation file because the code in the implementation file is outside the class definition. The keyword **friend** is only needed within the scope of the class. Also, remember that this friend function cannot use the pointer **this** in its implementation.

Exercise 2

At the end of Lab 3, we said: *You should now be able to design and implement a data collection ADT class that uses a concrete data type (CDT) made of dynamically allocated memory and provide most of its necessary methods, i.e., special member functions. We say "most" because we still need to add the overloaded assignment operator to our List ADT class.*

Well, we are now in a position to complete this data collection List ADT class from our Lab 3 by adding an assignment (=) operator to its **Public Interface**. Go ahead and modify your List class from Lab 3 by implementing (overloading) its assignment (=) operator. How would you test your overloaded assignment operator using the **test.cpp** class?

You may find that List's copy constructor and this new overloaded assignment operator have a lot in common. If this is the case, what would be the best way to deal with this common code? Would refactoring their common code into a private method, which both methods would call, a way of dealing with this situation? Remember, having duplicated code is not a good idea!

Lastly, even though we are not asked to overload the assignment (=) operator for MyADT class in Assignment 1, we can practice doing so by overloading this operator for the MyADT class in our **Lab4** directory.

Enjoy!

/