Friend Operator Overloading

1. Objectives

After you complete this experiment you will be able to overload an operator as a friend function of a class.

2. Introduction

When you overload an operator you provide your personal implementation for the operator. It comes in handy when you create a new object and you need to refine the implementation of an existing operator to match your needs. Overloading operators is really effective when the original meaning of the operator can be ported to the new object. For example, the "+" operator should mean some type of addition will take place. For strings and chars addition is concatenation; for doubles and ints it is numeric summation and for a new type of array class it may be the summation of all the corresponding elements in two adjacent arrays. Overloading C++ operators is an optional topic. However, we think it is important that you understand how to implement operator overloading, so we will cover it.

3. Definitions & Important Terms

We will define several terms you need to know to understand classes. They are as follows:

a. The **arity** of an operator is the number of parameters (operands) it requires. The arity of an operator cannot change.
b. A **friend** is a **non-member** function that has access to the state (private area) of a class. The class must give the function permission to be its friend.
c. **Chaining** occurs when a C++ statement contains several instances of the same overloaded operator.

4. Declaration Syntax

```
friend return_type function_name(formal parameter list);
//place this form of a prototype in your class declaration
```

Notice the location of the friend statement in the class declaration and the syntax for the implementation of the friend function body in the following code:

```
class Class_name
{
public:
      constructors
```

```
        destructor
        member functions
              accessors
              mutators
        friend return_type function_name(formal parameter list);_
        public data

  private:
        helper functions
        data
  };

  --------------------------------

  return_type function_name(formal parameter list)
  {
        body
  }
```

More information on classes can be found in your course textbook and on the web.

Member Operator Overloading

5. **Objectives**

   After you complete this experiment you will be able to overload an operator as a member function of a class.

6. **Introduction**

   When you overload an operator you provide your personal implementation for the operator. It comes in handy when you create a new object and you need to refine the implementation of an existing operator to match your needs. Overloading operators is really effective when the original meaning of the operator can be ported to the new object. For example, the "+" operator should mean some type of addition will take place. For strings and chars addition is concatenation; for doubles and ints it is numeric summation and for a new type of array class it may be the summation of all the corresponding elements in two adjacent arrays. Overloading C++ operators is an optional topic. However, we think it is important that you understand how to implement operator overloading, so we will cover it.

7. **Definitions & Important Terms**

   We will define several terms you need to know to understand classes. They are as follows:

   d. The **arity** of an operator is the number of parameters (operands) it requires. The arity of an operator cannot change.
   e. A **non-member** function of a class does not have access to the state (private area) of a class.
   f. **Chaining** occurs when a C++ statement contains several instances of the same overloaded operator.
   g. The **current object** is the object that performed the function invocation.
   h. **"this"** is a pointer to the current object.
   i. **All member functions of a class have access to the "this" pointer.**
   j. **Non-member and friend functions of a class do not have access to the "this" pointer.**
   k. **"*this"** is the current object.
   l. When an object is passed **"implicitly"** it is passed through the "**this" pointer**.
   m. When an object is passed **"explicitly"** it is passed through its corresponding formal parameter.

8. Declaration Syntax

Notice that the non-member function does not have a prototype in the class declaration and the syntax for the implementation of the non-member function body in the following code:

```
class Class_name
{
public:
      constructors
      destructor
      member functions
            accessors
            mutators
      public data
private:
      helper functions
      data
};
--------------

return_type Class_name::function_name(formal parameter list)
{
      body
}
```

More information on classes can be found in your course textbook and on the web.

Non-member Operator Overloading

9. Objectives

After you complete this experiment you will be able to overload an operator as a non-member function of a class.

10. Introduction

When you overload an operator you provide your personal implementation for the operator. It comes in handy when you create a new object and you need to refine the implementation of an existing operator to match your needs. Overloading operators is really effective when the original meaning of the operator can be ported to the new object. For example, the "+" operator should mean some type of addition will take place. For strings and chars addition is concatenation; for doubles and ints it is numeric summation and for a new type of array class it may be the summation of all the corresponding elements in two adjacent arrays. Overloading C++ operators is an optional topic. However, we think it is important that you understand how to implement operator overloading, so we will cover it.

11. Definitions & Important Terms

We will define several terms you need to understand in order to implement operator overloading as a non-member function. They are as follows:

n. The **arity** of an operator is the number of parameters (operands) it requires. The arity of an operator cannot change.
o. A **non-member** function of a class does not have access to the state (private area) of a class.
p. **Chaining** occurs when a C++ statement contains several instances of the same overloaded operator.

12. Declaration Syntax

Notice the location of the non-member function of the class in the class declaration and the syntax for the implementation of the non-member function body in the following code:

```
class Class_name
{
public:
      constructors
      destructor
      member functions
```

```
            accessors
            mutators
      public data
private:
      helper functions
      data
};
---------------------
return_type function_name(formal parameter list)
{
      body
}
```

More information on classes can be found in your course textbook and on the web.