LANGARA COLLEGE
DEPARTMENT OF COMPUTING SCIENCE AND INFORMATION SYSTEMS
CPSC 1160 - ALGORITHMS AND DATA STRUCTURES I
Assignment 05 – Lab 05
October 19, 2017

Instructions

- This assignment is worth 10 points, and is due on October 26 at 04:00 PM.
- As extra mark up to 10% (1 point) will be considered for more professional implementations.
- All the program files (*.cpp* and *.h* files) are required to be put in a folder named **Lab05**; the whole folder then should be submitted as a single zipped file on D2L.

Operator Overloading Functions

- **Benefits**
  Operator overloading simply means defining a function for a particular operator in a class. Without overloading an operator, the objects of your class in a client code cannot participate as the operand of that specific operator unless there is a default definition for the operator. For instance, for an imaginary class like `MyClass` with no operator overloading function, an expression like `c1 + c2` where `c1` and/or `c2` are objects of `MyClass` will results an error in the client code. The main reason for this error is that the compiler cannot find any function that defines the functionality of the operator "+". So, defining a function for the "+" operator (i.e. overloading the "+" operator) can be very beneficial for the clients of `MyClass`. Notice that, even if your class has already defined a function like `add()` to do the same operation, overloading this operator will be still beneficial as it might be much more convenient for your clients to simply use an operator instead of having to go through the syntax of a function call. Even for an operator like the assignment operator "=" which can be used with all types of objects (`c1 = c2;` performs a member-wise shallow copy where `c1` and `c2` are both objects of `MyClass`), it might be beneficial to overload this operator for example to make `c1` a deep copy of `c2` instead.

- **Categories**
  It should be noticed that operator overloading functions are first functions. So, as any other C++ function they can be categorized into *non-member* (*free*) functions and *member* functions. While non-member functions do not belong to any class, member functions always belong to a particular class, and therefore have access to other class members specially the data members. As a result, the input parameters of a member function are usually less than the input parameters of a non-member function with the same functionality.

  Specifically for an operator overloading function of a particular class, use the following rule of thumb to recognize if it should be defined as a member and/or non-member function. If the operator is:

  1. A unary operator whose only operand is an object of the class is implemented as a *member* function (e.g. ++, −−, unary −, etc.).

  2. A binary operator whose left operand is **always** an object of the class is implemented as a *member* function (e.g. +=, =, etc.).

3. A binary operator whose left operand can be of a type which is implicitly convertible to an object of the class is implemented as a *non-member function* (e.g. + in a numeric-type class).

4. A binary operator whose left operand is an object of another class while its right operand is an object of the current class is implemented as a *non-member* function (e.g. stream insertion operator <<, stream extraction operator >>).

   Note: In this case the non-member function needs to be declared as a `friend`.

- **Implicit Type Conversion Functions**
  It is possible to make the compiler capable of automatic type conversion from/to the class type to/from another type (e.g. a primitive data type).

  - **Converting to the class type** is possible by declaring a constructor which accepts only one parameter of the convertible type. For example adding the following constructor to the `MyClass` class, and provide an appropriate implementation for it can be used for implicit conversion from `int` to an object of type `MyClass`:

    ```
    MyClass(int intParameter){ … }
    ```

  - **Converting from the class type** is doable if an appropriate type cast operator is overloaded in the class. For instance, an object of the `MyClass` class will be convertible to a `double` if the following member function is added to the class with an appropriate implementation:

    ```
    operator double(){ … return someDoubleValue; }
    ```

    Notice that like constructors there is no return type (even `void`) required for the type cast overloading functions. The compiler assumes that data of the correct type is returned.

- **Implementation Steps**
  As mentioned above, operator overloading functions are functions. So, like any other function they have a name, a list of parameters, a return type (value), and a body. However, since these functions have special meaning to the compiler, they are named in a specific way: the keyword "`operator`" followed by the operator name/symbol (e.g. `operator+`, `operator=`, `operator[]`, `operator new`, etc.).

  Aside from naming, the operator overloading functions are actually implemented in the same steps as all other functions:

  1. Identify the list of parameters and the appropriate parameter passing method for each parameter in the list (i.e. pass by value or reference, if the `const` keyword should be used, etc.). To do so, answer questions like:

     - What data is accessible? (i.e. global variables, data members of the same class, etc.)

     - What data is required to be passed from the caller of the class?

     - Which of the passed parameters are expected to be changed (and returned) to the caller?

     - How to save time and memory (i.e. efficiency) in parameter-passing?

  2. Identify the function return type accordingly:

- What is required to be returned by the name of the function? (i.e. using the `return` keyword)

- If any, what is its type? Is it a pointer? A primitive? A specific object?

- How it should be returned? By value? By reference? Does this function return an "Lvalue" object? Would the return object be used in a chain of expressions?

3. Code the function body by implementing an algorithm that produces the expected results from the function inputs.

Note: Improper use of the above-mentioned functionality can lead to the ambiguity errors. To see an example, refer to the last paragraph of the section14.10.2 of the textbook on Page 540.

The Boolean ADT [3 points]

According to the above explanation, complement the Boolean ADT of the Assignment 4 by adding appropriate operator overloading functions.

The Array ADT [4 points]

Utilize C++ templates to implement the Integer Array ADT of the assignment 4 as a generic Array ADT. Complement your implementation by overloading appropriate operators.

Bitwise Operators [3 points]

Use the bitwise operators to write a program that reads an integer (`a`) in the range of 0-255 along with a bit number (`n`) in the range of 1-8 from the keyboard, and displays the following information on the screen:

1. If the n^{th} bit of the binary representation of "`a`" is zero or one?

2. What is the decimal integer which has the same binary representation as `a` but its n^{th} bit is toggled.

3. What is the decimal integer which has the same binary representation as `a` but its n^{th} bit is 0.