CIS023 Laboratory 2 Assignment Mockup

Exercise L2-1 Chap 13 Overloading and Templates

Please ensure your class header files have the exact same function names as listed below. If your class definitions do not match, I will be unable to test them. When submitting your lab assignment submit only the .cpp and .h files you created or modified. Be sure you have coded the cout statement as described in the Grading Rubric inside the contructor.

In this exercise, first redefine the class `rectangleType` by declaring the instance variable as protected and then overload additional operators as defined in parts A through C.

A. Overload the pre– and post– increment and decrement operators to increment and decrement, respectively, the length and width of a rectangle by one unit. (Note that after decrementing the length and width, they must be positive.)
B. Overload the binary operator **-** to subtract the dimensions of one rectangle from the corresponding dimensions of another rectangle. If the resulting dimensions are not positive, output an appropriate message and do not perform the operation.
C. The operators **==** and **!=** are overloaded by considering the lengths and widths of `rectangles`. Redefine the functions to overload the relational operator by considering the areas of `rectangles` as follows: two `rectangles` are the same, if they have the same area; otherwise, the rectangles are not the same. Similarly, `rectangle yard1` is greater than `rectangle yard2` if the area of `yard1` is greater than the area of `yard2`. Overload the remaining relational operators using similar definitions.
D. Write the definitions of the functions to overload the operators defined in parts A through C.
E. Write a test program that tests various operations on the `class rectangleType`.

rectangleType.h

```
    double area() const;
    double getLength() const;
    double getWidth() const;
    double perimeter() const;
    void   setDimension(double l, double w);

    friend ostream& operator<<(ostream&, const rectangleType &);
    friend istream& operator>>(istream&, rectangleType &);

    rectangleType operator + (const rectangleType &) const;
    rectangleType operator - (const rectangleType &) const;
    rectangleType operator * (const rectangleType&) const;
    rectangleType operator ++ ();
    rectangleType operator ++ (int);
    rectangleType operator -- ();
```

```
    rectangleType operator -- (int);
    bool operator == (const rectangleType&) const;
    bool operator != (const rectangleType&) const;
    bool operator <= (const rectangleType&) const;
    bool operator < (const rectangleType&) const;
    bool operator >= (const rectangleType&) const;
    bool operator > (const rectangleType&) const;
    rectangleType();
    rectangleType(double l, double w);
```

Exercise L2-2 Chap 14 Exception Handling

Write a program that prompts the user to enter time in **12:00:00AM** thru **11:59:59PM** (12-hour notation). The program then outputs the time in **00:00:00** thru **23:59:59** (24 – hour notation). Your program must contain three exception classes: **invalidHr, invalidMin, invalidSec**. If the user enters an invalid value for hours, then the program should throw and catch an invalid HR object. Follow similar conventions for the invalid values of minutes and seconds.

Each exception class (**invalidHr, invalidMin, invalidSec**) should have 2 constructors:

The default constructor that supplies the standard error message and a parameterized constructor which takes a string variable to initialize the error message.

Your derived exception classes should also supply the necessary overloads for returning an error message.

Name the header files the same as the class name with the **.h** file extension.

Exercise L2-3 Chap 15 Recursion

(**Ackermann's function**) The Ackermann's function is defined as follows:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } n = 0 \\ A(m - 1, A(n, n-1)) & \text{otherwise} \end{cases}$$

In which ***m*** and ***n*** are nonnegative integers. Write a recursive function to implement Ackermann's function. Also write a program to test a function. What happens when you call the function with ***m*** = 4 and ***n*** = 3?

Answer this question: How many times has the Ackermann Function been recursively called when ***m*** = **3** and ***n*** = **5**?