In this laboratory exercise, you will explore the use of template classes and exception handling in C++.

**Exercise 1.** The following program contains a simple class that is designed to allow a user to enter an integer value within a specific range. The code prompts the user with a message that explains the limits of the valid range of values, but it does not force the user to enter legal values. We have examined a number of approaches for dealing with this problem. One approach is to place an assert statement in the getValue function to terminate the program when an illegal input occurs. An alternative approach is to put a loop inside the getValue function that prompts the user repeatedly until a valid value is received. Neither of these approaches is satisfactory, because they force the user to adopt a specific approach to dealing with the error. If you compare the execution of a program with the operation of a business, you will notice that both consist of a hierarchy of classes and functions. In a business, we expect the people in the upper layers of the company (managers) to set policies and the people in lower positions (workers) to implement the policy. We want the same type of behavior in our programs, but both of our approaches to handling errors lead to an inversion of authority -- the bottom layer (implementation of the class) is making decisions for the top-layer (user's application program). A better approach to managing this problem is to use an exception handler that passes control back to a top-level function when an error occurs so the user can make the decision to terminate the program or continue processing.

**TASK:** Add an exception handler (try/catch/throw) to the class that throws an error message (e.g. “value out of range”) in the getValue function that is caught and handled in the main program. Implement two version of the program. In version 1, throw the error in getValue, catch the error in the main, display an error message in the main, and then allow the program to terminate. In the second version, perform the same basic actions (throw and catch) but keep re-invoking the getValue function from the main program until the user enters a valid value.

```cpp
// Starter Code
class subRange
{
    public:
        subRange( int, int );
        int getValue();
    private:
        int lower, upper;
};
subRange::subRange( int low, int high )
{
    lower = low;
    upper = high;
}

int subRange::getValue()
{
    int v;
    cout << "Enter value [ " << lower << ", " << upper << "]: ";
    cin >> v;
    return v;
}

void main()
{
    subRange x(1,10);
    cout << x.getValue() << endl;
}
Exercise 2. Add an exception handler to the following program that catches a bad_alloc exception if a user attempts to allocate an array object and insufficient memory is available to handle the request. Test your solution using the main program shown.

// Starter Code
class array
{
    public:
        array( int = 1 );
    // other functions not needed for this assignment
    private:
        int* elts;
        int count;
};

array::array( int sz )
{
    count = sz;
    elts = new int[ count ];
}

void main( )
{
    array* ptr[10];

    for ( int i = 0; i < 10; i++ )
    {
        ptr[i] = new array( 100000000L );
    }
}

Exercise 3. Take the first version of the code that you completed for exercise 1 and convert the subRange class from a class that allows a user to enter integers to a template class that allows a user to enter a value in a sub-range for any simple data type (e.g. int, double, char). Test your class by declaring variables of type subRange<int>, subRange<double>, and subRange<char> and input values within and outside the allowed range.