Error Handling with Exceptions

In this lab you will:

- Write exception classes
- Catch and handle exceptions

You are given an `IntStack` class that has plenty of error checking. However, whenever an error occurs, the program exits. This is bad because you don't want `IntStack` to handle the problem — you want to handle the problem in `main()`. The general rule is that only `main()` should decide if the program should exit. For example, if the stack is empty when we try to pop a value, main may know a way to handle it and can avoid exiting.

```
#include <vector>
#include <iostream>
using namespace std;

class IntStack
{
public:
        // MaxSize should be an unsigned int, but for the sake of example...
        IntStack(int MaxSize)
        {
                if (MaxSize < 0)
                {
                        cerr << "Cannot create a negative size stack" << endl;
                        exit(1);
                }

                data.resize(MaxSize);
                cur_index = 0;
        }

        void push(int new_data)
        {
                if (cur_index == data.size())
                {
                        cerr << "Push to full stack" << endl;
                        exit(1);
                }
                else
                {
                        data.at(cur_index) = new_data;
                        cur_index++;
                }
        }

        int pop()
        {
                if (cur_index == 0)
                {
                        cerr << "Pop from empty stack" << endl;
                        exit(1);
                }
                else
                {
                        // pop off the int and return it
                        cur_index--;
                        return data.at(cur_index);
                }
        }

private:
        vector<int> data;
        unsigned int cur_index;
};

int main()
{
        // Testing Constructor
        IntStack c_test(-10);

        c_test.push(3);
        c_test.push(4);
        c_test.pop();
        c_test.pop();


        // Testing push
        IntStack push_test(5);

        for (unsigned int i = 0; i < 7; i++) {
                push_test.push(i);
        }


        // Testing pop
        IntStack pop_test(2);

        pop_test.push(1);
        pop_test.push(2);
        pop_test.pop();
        pop_test.pop();
        pop_test.pop();

        cout << "Completed!" << endl;
}
```

The `IntStack` class works as follows. When constructed, it allocates a vector of a specified size. This `size` will not change and limits the number of items that can go in the stack. `IntStack` also has two functions that are usually included with stacks: `push()` and `pop()`. `push()` adds an item to the stack; this can cause an error if the stack doesn't have any more room. `pop()` removes the most recently added item; this can cause an error if the stack is empty. The stack keeps track of the top with an unsigned integer called `cur_index`. This number stores the index that will be pushed onto next. For example, if `cur_index` is 0, the stack is empty and the next pushed item will be placed in index 0 of the vector.

The `main()` function includes three tests that specifically trigger each error case.

Handle the constructor's error case

An error occurs when a negative size is passed to the constructor. At the top of `exceptions.cpp`, write an `InvalidSize` class to represent this error, it should inherit from the `invalid_argument` class (from the STL exception hierarchy). This exception should take a c-string in its constructor and call the `invalid_argument` constructor by passing in the c-string via the constructor initialization list. Modify the constructor so that it throws this exception with an error message passed to the exception's constructor *rather than* printing an error message and calling `exit(1)`. Write the code in `main()` to catch the exception and use `cerr` to print out the message returned by `what()`

Handle push's error case

An error occurs if `push()` is called when the stack is full. At the top of `exceptions.cpp`, write a `StackFull` class to represent this error, and inherit from the `runtime_error` class (from the STL exception hierarchy). This exception should take a c-string and an int in its constructor; the c-string is an error message, and the int is the argument to `push()` that was *not* able to be pushed onto the stack. In the constructor initialization list, call the `runtime_error` constructor by passing in the c-string error message. The class should have a `GetValue()` accessor that returns the value (it inherits an accessor `what()` that returns the string, so you don't have to write it). Modify `push()` so that it throws this exception with an error message and value passed to the exception's constructor *rather than* printing an error message and calling `exit(1)`. Write the code in `main()` to catch the exception and use `cerr` to print out the message returned by `w` `hat()` and the value returned by `GetValue()`.

Handle pop's error case

An error occurs if `pop()` is called when the stack is empty. At the top of `exceptions.cpp`, write a `StackEmpty` class to represent this error, it should inherit from the `runtime_error` class (from the STL exception hierarchy). This exception should take a string in its constructor. In the constructor initialization list, call the `runtime_error` constructor by passing in the c-string error message. Throw this exception with an error message passed to the exception's constructor *rather than* printing an error message and calling `exit(1)`. Write the code in `main()` to catch the exception and use `cerr` to print out the message returned by `what().`