CISP 400 - Lab Assignment #10
Due Thursday, December 6, 2018 at 11:59 PM

Exception processing is an important part of writing C++ code, since so much code depends on it. The language also depends on it. However, it is not always the easiest thing in the world to come up with an error condition we can use with it; not all errors result in a thrown exception, and it can sometimes be difficult to force an error condition. Although it's a bit easier when using file I/O, we have not yet covered that. As such, we are going to have to put together our own program that throws artificial exceptions.

This assignment covers:

- Throwing exceptions
- Practical use of abstract classes
- Building custom exceptions for throwing
- Stack unwinding (limited)

PROGRAM SPECIFICATIONS:

This lab is a demonstration program of how to write try-catch blocks and the basic use of the throw keyword in concert with these programming structures.

CLASSES:

This lab involves creating several different classes, using inheritance for most of it.

AnnounceDestroyed:

This is a very simple class. In its constructor, it announces that it is created. In its destructor, it announces it is destroyed. Both use cout for the announcements.

ExceptionThrower:

This is an ***abstract*** class. It has one member function, declared virtual. This function is launchException(). It has a return type of void and no parameters. It is a pure virtual function, so you will have to add that to the class declaration (remember, you cannot instantiate this class!). It should also have a destructor that is declared virtual. The destructor code can be an empty function, but it should be present.

IntThrower:

This is a subclass of ExceptionThrower.

Its constructor takes in a single int value which is stored in a private variable. It has code for launchException(). When launchException is run, it throws using that private variable.

MsgThrower:

This is a subclass of ExceptionThrower.

Its constructor takes in a single string value which is stored in a private variable. It has code for launchException(). When launchException is run, it throws using that private variable.

CustomThrower:

This is a subclass of ExceptionThrower.

Its constructor takes in a single string value and a single int value. Both of these are stored in private variables. When launchException is run, it has a local variable of the CustomException type. The constructor for CustomException is given these private variables to initialize it. The function then throws that CustomException object.

CustomException:

This is a class that is not a subclass of any other class.

Its constructor takes in a single string value and a single int value. It stores these as private variables. It also has a public member function, printContents(). printContents() displays the contents of the string and the number that were provided at construction. It does this using cout to display it for the user in order to explain the (fake) error.

HELPER FUNCTIONS:

This program has one helper function as well, activateException(). activateException() takes in a reference to an ExceptionThrower type object, which we will call currentThrower.

activateException has a single shared pointer of type AnnounceDestroyed. This pointer should be initialized using make_shared or the constructor. Then the function calls currentThrower.launchException(). Also note that you might get a compiler warning because the object that this pointer is pointing to will not have any code explicitly run or variables used.

At the very end of the function there should be code that declares an exception was attempted, but did not occur. If your program is running properly, this statement will never execute, because execution will not properly return to activateException, and will resume at the catch block (or, if there is no appropriate catch block anywhere in the stack, it will crash).

MAIN PROGRAM:

The main program includes the following constants:

```
const int INT_THROWER_CODE = -1;
const string MSG_THROWER_STRING = “Error!”;
const int CUSTOM_THROWER_CODE = -99;
const string CUSTOM_THROWER_STRING = “Major error!”;
```

The main program starts with an int named exceptionCounter, which is initialized to 0.

The main program creates several objects as local variables. These include:

1. an instance of IntThrower (using INT_THROWER_CODE as the parameter to its constructor), which we will call LauncherInt
2. an instance of MsgThrower (using MSG_THROWER_STRING as the parameter to its constructor), which we will call LauncherMsg
3. an instance of CustomThrower (using CUSTOM_THROWER_CODE and CUSTOM_THROWER_STRING as the parameters to its constructor), which we will call LauncherCustom

The main program has three try-catch blocks. For the try block for each, it:

1. Announces that an exception test is about to take place (using cout to display this message)
2. Uses activateException with one of the ExceptionLauncher subclass instances (LauncherInt, LauncherMsg and LauncherCustom, respectively
3. Announces that an exception did not take place (using cout to display this message). Note that this particular statement will never run, if the code is working correctly.

Organize these try-catch blocks in this order: LauncherInt, LauncherMsg, and LauncherCustom.

Each try block has a corresponding catch block. The catch block:

1. catches the appropriate type of exception (int for LauncherInt, string for LauncherMsg, CustomException for LauncherCustom)
2. prints out the contents of that value, on its own line. In the case of an int parameter or a string parameter, just use cout. For LauncherCustom, use the printContents() member function for the CustomException that you caught.
3. Increments exceptionCounter

At the very end, the program uses cout to announce how many exceptions were count using the counter, and then exits the program.

TIPS:

- Remember to throw by value, and catch by reference.
- Do not dynamically allocate variables for exceptions – you only need one dynamic allocation for this entire program, using a shared pointer.
- Exceptions get very complicated very easily. Try to be straightforward in following the specifications.
- Organize your program so that the code for classes is generally in one place, e.g. one specific place for IntThrower, one for CustomThrower, etc.
- Remember that class forward declaration can save you quite a bit of trouble.
- This lab does not require any input from the user. It should be able to automatically complete all tests by itself.

DELIVERABLES:

Upload the .cpp file to Canvas by the due date. Please do not use separate files.