COMP 220
Lecture Notes 09
Exceptions and Exception Handling

October 20, 2016

These notes briefly cover exceptions and exception handling in C++.

Runtime Errors

Until now we've mostly assumed that arguments passed to procedures meet the preconditions. This is usually a dangerous assumption. You can use other procedures to check and enforce preconditions, sure, but what do you do in those procedures when they detect some data doesn't hold mustard? The answer is to raise an error to the user/client which in programming is called THROWING AN EXCEPTION.

When a procedure throws an exception it directs that exception to the procedure that called it. This process bypass the usual return mechanism. A procedure can throw an exception or return, but not both. The caller of the procedure that threw can either CATCH and handle the exception or do nothing and in doing so throw the exception up to its caller. If no procedure catches the exception and it makes its way to main, then the program with crash.

Crashing or ending a program when a run-time error occurs is often the right thing to do. Without exceptions, the operating system is going to do this in a very abrupt manner. If you had files open for output, then they won't be saved and you're going to lose some data. What's more, the OS might do nothing more than tell you a segmentation fault occurred without giving you any clue as to what happened[1]. Exception handling lets you take the time to save work, write error logs prior to closing down, and report to the user the nature of the problem.

[1] compare out-of-bounds vector access with at and operator[]

Exceptions in the Standard Library

The *throw* operator is used to throw exceptions. The C++ stdexcept[2] comes with a set of pre-defined exception types for many common program errors and these exception types are defined in a heirarchical manner like the I/O streams we've worked with previously.

[2] `http://www.cplusplus.com/reference/stdexcept/`

At the top of the exception hierarchy is std::exception. Next there are two categories of exceptions: *std::logic_error*[3] and *std::runtime_error*[4]. The later is used for errors that can only be caught at runtime where the former is used for things that could have been caught at compile

[3] `http://www.cplusplus.com/reference/stdexcept/logic_error/`

[4] `http://www.cplusplus.com/reference/stdexcept/runtime_error/`

time. Each of these types of exceptions has more specific variants that you are encouraged to investigate. In the event that you need a more specific type of error you can use Object-Oriented programming to extend the exception hierarchy with your own custom exception types.

Throwing Exceptions

Throwing exceptions is done with the *throw* operator which is used in the same fashion as *return*. Exceptions are constructed like other objects and typically take a single string type argument. That string is the error message/description.

Let's say you were doing some kind of calculator program and needed to detect divide by zero errors. When you detect the error you could use a throw statement like the one shown in figure 1.

```
1 throw std::runtime_error("Divide by Zero detected");
```

Figure 1: Throwing a generic divide by zero error

Catching Exceptions

Catching exceptions requires a *try..catch* statement. The try block contains the code the might throw an exception. It is followed by a series of catch blocks for the types of exceptions the code might generate. Each exception type includes a variable declaration for the exception.

```
1  try{
2    // code that might throw
3  }
4  catch( exception_type exception_name){
5    // what to do if exception of exception_type is thrown
6    // exception_name is a variable for the exception object
         thrown.
7  }
8  // more catch blocks if needed
```

Figure 2: Template for Try Catch statements

You can access the exception error message/description through the *what* method. Figure demonstrates exception handling.

The procedure *std::exit*[5] terminates the whole program using the normal program termination process and can be used if you aren't in *main* and cannot or don't want to return back to main and return from there.

[5] http://www.cplusplus.com/reference/cstdlib/exit/

```
try{
  procedure_that_might_throw();
}
catch( std::runtime_error e ){
  std::err << "Error Detected: " << e.what() << '\n';
  std::exit(EXIT_FAILURE); //ends program with error code
}
```

Testing with Exceptions and gTest

If you're designing procedures that might throw errors then you need to test that errors are thrown when they should be. The gTest library provides EXPECT statements for this purpose[6].

When a procedure might throw several types of exceptions you should use *EXPECT_THROW*.

[6] https://github.com/google/googletest/blob/master/googletest/docs/AdvancedGuide.md#exception-assertions

```
EXPECT_THROW( might_throw(5) , std::runtime_error );
EXPECT_THROW( might_throw(10) , std::logic_error );
```

If the type of the error isn't important to the test, you can use *EXPECT_THROW_ANY*.

```
EXPECT_THROW_ANY( might_throw(5) );
```

Finally, if you need or want to verify that no errors occur there is *EXPECT_NO_THROW*.

```
EXPECT_NO_THROW( might_throw(0) );
```