COMP 220: Data Structures **Lecture Notes 09**

Exceptions and Exception Handling

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 [1]. 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?

[1] Such as binary search (vector must be sorted), or stack `pop` (stack must be non-empty).

There are lots of different ways to handle errors, and different methods are popular and/or more useful in various languages. One option when you detect an error is simply to immediately end the program, using something like `std::exit()` [2] or `std::abort()` [3]. The obvious problem here is that sometimes an exception isn't the end of the world — it can be "handled". If the user enters a number instead of a string, for example, you may simply wish to print out an error message and have the user retry inputting a string. Crashing the program may also yield no error messages, or at best cryptic ones; descriptive explanations of what went wrong are always better. A more subtle issue is that your program may be using resources, such as reading from or writing to files, or network connections communicating to various servers over the internet. Although we haven't seen much yet, setting up these resources is complicated and often requires some kind of "cleanup" procedure when we're done. Simply exiting the program on an error doesn't do any resource cleanup for us.

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

[3] `http://www.cplusplus.com/reference/cstdlib/abort/`

Another options is to return some kind of special value from procedures that have discovered an error. For example, when we implemented our stack for non-negative integers, `top()` returned `-1` to indicate that the stack was empty. This was okay because we defined our stack to only hold non-negative integers. What if we wanted our stack to hold any integer? The general problem here is that many procedures return a result, but you also need them to return an error type when an error has occurred. So you either need a special error value, such as -1 in our case, or you need to define some special new type that can hold both the result AND an error, which is a pain and gets messy fast. The other issue with returning errors is that you must check the return type of each procedure and handle an error. This can quickly yield messy code, as long chains of procedures check for errors and pass a new error to their parent.

A popular way to raise errors to the user/client in C++ is called *throwing exceptions*. 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 will crash, similar to using `std::exit()` except that we may receive a more useful error message.

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[4]. 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.

[4] 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[5] comes with a set of pre-defined exception types for many common program errors and these exception types are defined in a hierarchical manner like the I/O streams we've worked with previously.

[5] `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*[6] and *std::runtime_error*[7]. The latter 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 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.

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

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

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.

```
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.

Figure 2: Template for Try Catch statements

```
try{
  // code that might throw
}
catch( exception_type exception_name){
  // what to do if exception of exception_type is thrown
  // exception_name is a variable for the exception object
       thrown.
}
// more catch blocks if needed
```

You can access the exception error message/description through the *what* method. Figure 3 demonstrates exception handling.

Figure 3:

```
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
}
```

The procedure *std::exit*[8] 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.

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

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[9].

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

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

```
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) );
```