

Interlude 3: Exceptions

History of handling errors

Before exceptions became more ubiquitous, there were other common ways of dealing with errors in programs.

Error codes

When checking for errors, programmers used to use a unique number code to represent each error. You might have seen this in early programs, where you'd get some "error #1423" without much more information. With these error codes, the programmers could do a search for that error number to find where in the code it crashed. However, for the users, an arbitrary number didn't really help us avoid broken features.

Return a boolean / nullptr

If you've ever used a C library, you might have experienced this. In C, library functions often would return "true" if successful, and "false" if unsuccessful. Or, if a function was responsible for allocating some memory or accessing some data, it might return "nullptr" if there were an error.

This is better than nothing; any programmer using the library can do error checks as they go through. Here are some examples using the SDL library:

```
if ( !SDL_SetHint( SDL_HINT_RENDER_SCALE_QUALITY, "1" ) )
{
    cerr << "error setting render" << endl;
    return false;
}

m_window = SDL_CreateWindow( winTitle.c_str(), SDL_WINDOWPOS_UNDEFINED,
                             SDL_WINDOWPOS_UNDEFINED, m_screenWidth, m_screenHeight, windowFlags );

if ( m_window == NULL )
{
    std::string error( SDL_GetError() );
    cerr << "error creating window!" << endl;
    return false;
}
```

Concepts

Exceptions

- What is an exception?
- How are exceptions different from using return values for functions to report whether something failed?

It is important to use exceptions when you're writing reusable code, or code that will be used by other programmers. Using exceptions, you can report that an error occurred and what *kind* of error it is. Then, with this information, the programmer using your tool can decide how best to resolve the error.

C++ exception classes

For each exception, write out what kind of scenario it would be good to use it for.

Exception	Scenario
invalid_argument	
length_error	
out_of_range	
overflow_error	
range_error	
underflow_error	
logic_error	
runtime_error	
exception	

You can also write your own exception class that inherits from any of these, to make a specialized error types.

Levels of using exceptions

Exceptions can be confusing at first. Remember that there are different “levels” where exceptions are used.

Detecting errors and throwing the exception

When writing a function that may result in an error, you will use an if statement to check for the error state, and **throw** an exception if detected. A function can throw multiple types of exceptions. For example:

```
string SmartDynamicArray::Get( int index ) const
{
    if ( index < 0 || index >= m_arraySize )
    {
        throw out_of_range( "Index is out of range" );
    }
    if ( m_data == nullptr )
    {
        throw runtime_error( "Array is null" );
    }

    return m_data[ index ];
}
```

Trying out a function and catching any exceptions

When you’re calling that risky function, you should surround it in a **try/catch** block. Within **try**, you call the function. Then, you can have one or more **catch** blocks to grab exceptions. You should order these from most-specific (the “youngest” child in the family) to most-generic (the “exception”) class itself.

```
try
{
    string obj = myArray.Get( 100 );
}
catch( out_of_range& ex )
{
    cout << "Error! " << ex.what() << endl;
}
catch( runtime_error& ex )
{
    cout << "Error! " << ex.what() << endl;
}
```

You can also just catch any other exceptions with `catch(...)`

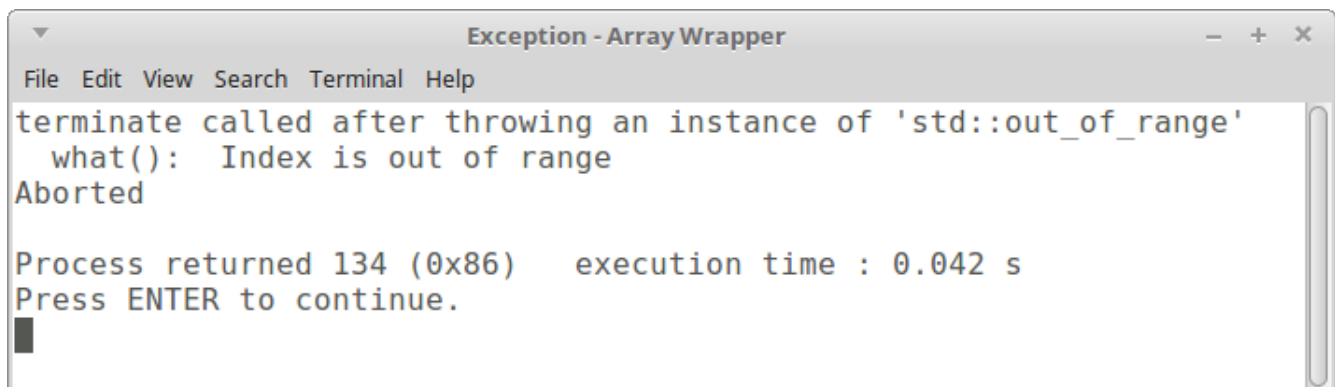
Passing exceptions up the chain

Let's say that you're writing a reusable class that uses somebody else's code. Your code **catches** an exception from the library you're using. If you can handle it within your structure – fine, so do. Otherwise, you'll want to pass the buck to the programmer that's using *your* structures. If you catch an exception, you can then throw the same exception.

```
catch( out_of_range& ex )  
{  
    throw ex;  
}
```

Not detecting exceptions

In the case where you don't detect an exception, the program will crash instead:



```
Exception - Array Wrapper  
File Edit View Search Terminal Help  
terminate called after throwing an instance of 'std::out_of_range'  
  what():  Index is out of range  
Aborted  
  
Process returned 134 (0x86)   execution time : 0.042 s  
Press ENTER to continue.  
█
```

Exception safety

Read through https://en.wikipedia.org/wiki/Exception_safety and describe each of the following...

Guarantee	Description
No-throw guarantee	
Strong exception safety	
Basic exception safety	
No exception safety	