# Exceptions

# What are exceptions?

- Exceptions are a feature in C++ that is used (can be used) when your code encounters an abnormal condition.

- C++ provides means of raising an exception (or throwing an exception), and handling an exception (or catching an exception).

# Example

- In the Stack example we have seen so far,

```
int  CArrayStack::Pop ( )
{
     int retVal = -1;
     // Lets keep it simple,
     // return -1 if stack is empty
     if ( ! IsEmpty() )
     {
          retVal = m_Array [m_NumElements ];
          m_NumElements --;
     }

    return retVal;
}
```

```
int  CArrayStack::Pop ( )
{
     int retVal = -1;
     if ( ! IsEmpty() )
     {
          retVal = m_Array [m_NumElements ];
          m_NumElements --;
     }
     else
          throw StackEmpty ();

    return retVal;
}
```

Notice that we don't return any default values if stack is empty, in fact no value is returned in that case, but an exception is thrown.

# Example: contd.

- Lets see the usage of Stack::Pop():
  - 
```
int main()
{
   try
   {
          Stack  s1;
          … // push elements onto the stack.

          int  value =  s1.Pop();
          cout  <<  "Pop returned "  <<  value <<  endl;
   }
   catch (StackEmpty & e)
   {
          cerr  <<  "Stack Empty exception caught"  <<  endl;
   }
}
```

NOTE: In C++, exceptions are non-resumptive, i.e., once an exception is thrown, control does not return to the statement after the one that throws the exception.

# What did we do in the pre-exception era?

- We had return codes, of course.
- Functions would return, say, 0 in case of no errors, but a non-zero code for an error condition.
- The programmer would need to check for any error code returned from a function call, before proceeding with the rest of the code.
- As a result, the error checking code has to be right after the function call, which means that error handling is mixed in with the actual code.
- In a non-trivial piece of code, this can lead to long term maintenance hassles.
- We still use return codes for normal functioning of code, i.e., we still write functions that have return values, and this is what we look at in non-error situations.
- But in an error condition, we can throw exceptions.

# The *throw* keyword

- *throw* is used to throw an exception (or raise an exception).

- This keyword is followed by the object that is thrown.
- This object is usually an instance of an Exception class.

- Note: In C++, it is most common to throw an object of a class type, but you can throw an object of any type ( E.g. integer, or float or an enum ). Not recommending you do that, but it is allowed by the language.

  - Example:
    - if ( ptr == NULL)
      throw 0;

# The *catch* keyword

- The *catch* keyword is used to catch an exception (handle an exception).
- This keyword is followed by the type of the exception that you want to catch.
- Example:
  - ```
    catch ( MyException & e )
    {
      // code to handle this exception, which may be something as simple
      // as writing an error string to stderr or to a file.
      // Release memory / resources, if any.
    }
    ```
    ```

# The *try* keyword

- The *try* keyword tells the compiler that you are planning to catch (handle) some exceptions.

- Actually the *catch* keyword that we just saw always occurs after a *try* block. This is usually known as the try-catch block.

- So, the previous example really looks like this:

```
try
{
    …  // some code that can throw exceptions.
    myObj.Foo();
    myObjPtr->Foo2();
}
catch ( MyException & e )
{
    // code to handle this exception, which may be something as simple
    // as writing an error string to stderr or to a file.
    // Release memory / resources, if any.
}
```

# Exception classes

- When we throw an exception, we usually throw an object of some class.
- This class is usually an exception class, which means that this is a class that you have developed in your code to represent some exception condition.
- Example:
  - In an earlier slide, we saw an example of the StackEmpty exception. StackEmpty is a class that we would have to write.
  - Typically, this class would be able to return an error message describing the error (exception) that it represents.

# Example

```
class StackEmpty
{
public:
    StackEmpty() { };

    const char *   GetDescription ()  const  throw();
};

inline const char * StackEmpty :: GetDescription () const throw()
{
    return  "Stack is empty";
}
```

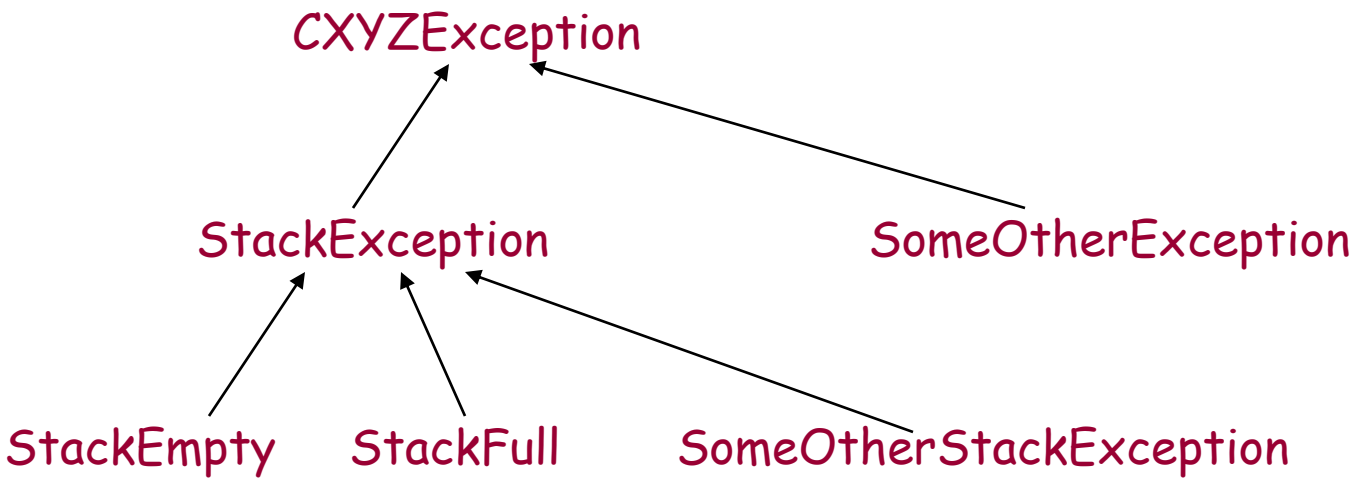
NOTE:

1. Usually we put the return type as a const char *, and not like a string * or string &. WHY ?

2. GetDescription() method (or any other method in this class) should not throw an exception.

# Hierarchy of exception classes

- We can also have a hierarchy of exception classes.
- Example:



# Hierarchy of exceptions

- There are some considerations when you are catching exceptions that are part of a class hierarchy.

- The order of catching exceptions should be bottom up as far as the class hierarchy is concerned.
- This means that in a list of catch clauses, you have to catch the deepest derived class first, and the base class last. **WHY ?**

- Example:

```
catch ( StackEmpty  &  e )
{
}
catch ( StackFull  &  e )
{
}
catch ( StackException  &  e )
{
}
```

# The catch all...

- If you want to catch any exception, then you have to specify this as follows:

```
catch (...)
{
   // code for handling.
}
```

As you would guess, in a list of catch statements, this should be the last catch statement.

# What after the catch?

- Inside a catch block, if there is a return statement, then control returns to the caller of the function that has the catch block ( NOT the caller of the function that threw the exception).

- If there is no return inside the catch block of statements, then execution of the function resumes at the statement that follows the last catch clause in the list.

- Example:

```
catch ( StackEmpty  &   e )
{
   cout << "message";
}
catch ( StackFull  &   e )
{
   cout << "message";
}
```

← Execution continues here.

```
catch ( StackEmpty  &   e )
{
   cout << "message";
   return;            ← Return to caller
}
catch ( StackFull  &   e )
{
   cout << "message";
   return;            ← Return to caller
}
```

# Stack unwinding

- When an exception is thrown, the code looks for a catch clause that will handle this exception.
- This handling catch clause may be declared in the same function that throws the exception, or in a different function up the function calling chain.
- In the process of searching for this handler, the functions that are in the call chain are exited, but not before destructing the objects that are on their stacks. (*This is what happens anyway when a function exits normally, i.e., local objects on the stack go out of scope and are destructed*).
- This process of searching up the call chain, looking for a handling catch clause, and in the process destructing the local objects on the functions stack is called *stack unwinding.*

# Example:

- Lets say function fooA () called fooB(), which called the Pop () method on a Stack object.
- Let the Pop() method throw a StackEmpty exception.
- Let the catch clause that handles this exception be in fooA().
- 

```
void fooA()
{
        try
        {
                fooB();
        }
        catch (StackException & )
        {
                // code to do whatever you want in this case.
        }
}

void fooB()
{
        Stack s1;
        int value  =  s1.Pop();              ← Lets say  a StackEmpty exception is thrown here.
        cout   <<  "Value popped from stack is " <<  value  << endl;
}
```

**NOTE**: fooB does not have a catch clause for StackEmpty exception, so fooB() is exited after unwinding its stack, and the exception is caught at the upper level in fooA()

# The rethrow

- In some situations, you may want to catch an exception, but cannot fully handle it.
- In such situations, you can catch that exception, do some actions, and then throw the same exception again for some caller up the chain hierarchy to handle it.
- See example for rethrow.

# What happens if no catch?

- If an exception is not caught, i.e., there is no catch clause that matches the exception thrown, the program will exit.

- Specifically, in such a case, a function called *terminate*() is called, which calls abort() to abort the program execution.

- It is possible to override the default behavior of terminate() if you wanted to (but this is not common).

# Destructors

- You should never throw an exception from within a destructor.
- This is because the destructor might have been called as a result of an exception thrown earlier ( stack unwinding ), and the run time system does not handle more than one exception being thrown at a time.

## Exceptions

- The error handling logic is separate from the actual functionality.
- Exceptions cannot be ignored, they need to be handled at some level in the program (else the program exits).
- You have one place in your function where you can place your cleanup code.
- Using try – catch blocks does cost you some in terms of performance since the compiler needs to add some extra code.

## Return Codes

- Error handling logic has to be right after the function call.
- Error condition can potentially be ignored in some cases, which can sometimes lead to subtle errors.
- You can still have one place, but you may end up with nested if statements (or goto statements).
- No extra code for return codes, however, using if statements to check for error codes does add code too…