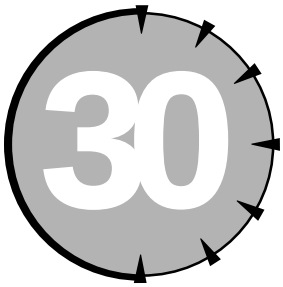
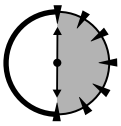
SESSION



Exceptions

Session Checklist:

- ✔ Returning error conditions
- ✔ Using exceptions, a new error-handling mechanism
- ✔ Throwing and catching exceptions
- ✔ Overloading an exception class



30 Min. To Go

In addition to the ubiquitous error-message approach to error reporting, C++ includes an easier and far more reliable error-handling mechanism. This technique, known as exception handling, is the topic of this session.

Conventional Error Handling

An implementation of the common example factorial function appears as follows.

This factorial function appears on the accompanying CD-ROM in a program called FactorialProgram.cpp.

```
// factorial - calculate the factorial of nBase which
//             is equal to nBase * (nBase - 1) *
//             (nBase - 2) * ...
int factorial(int nBase)
{
    // begin with an "accumulator" of 1
    int nFactorial = 1;

    // loop from nBase down to one, each time
    // multiplying the previous accumulator value
    // by the result
    do
    {
        nFactorial *= nBase;
    } while (--nBase > 1);

    // return the result
    return nFactorial;
}
```

While simple enough, this function is lacking a critical feature: the factorial of zero is known to be 1, while the factorial of negative numbers is not defined. The above function should include a test for a negative argument and, if so, indicate an error.

The classic way to indicate an error in a function is to return some value that cannot otherwise be returned by the function. For example, it is not possible for the returned value from factorial to be negative. Thus, if passed a negative number, the factorial function could return a –1. The calling function can check the returned value—if it's a negative, the calling function knows that an error occurred and can take the appropriate action (whatever that may be).

This is the way that error processing has been done ever since the early days of FORTRAN. Why change it now?

Why Do I Need a New Error Mechanism?

There are several problems with the error-return approach to error reporting. First, while it's true that the result of a factorial cannot be negative, other functions are not so lucky. Take logarithm for example. You can't take the log of a negative number

either, but logarithms can be either negative or positive — there is no value that could be returned from a `logarithm()` function that isn't a legal logarithm.

Second, there's just so much information you can store in an integer. Maybe –1 for "argument is negative" and –2 for "argument too large," but if the argument is too large, there is no way to store the value passed. Knowing that value might help to debug the problem. There's no place to store any more than the single error-return value.

Third, the processing of error returns is optional. Suppose someone writes `factorial()` so that it dutifully checks the argument and returns a negative number if the argument is out of range. If the code that calls that function doesn't check the error return, it doesn't do any good. Sure, we make all kinds of menacing threats such as, "You will check your error returns or else . . . " but we all know that the language (and our boss) can't do anything to us if we don't.

Even if I do check the error return from `factorial()` or any other function, what can my function do with the error? Probably nothing more than output an error message of its own and return another error indication to its calling function. Soon the code looks like this:

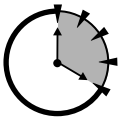
```
// call some function, check the error return,
// handle it, and return
int nErrorRtn = someFunc();
if (nErrorRtn)
{
    errorOut("Error on call to someFunc()");
    return MY_ERROR_1;
}

nErrorRtn = someOtherFunc();
if (nErrorRtn )
{
    errorOut("Error on call to someOtherFunc()");
    return MY_ERROR_2;
}
```

This mechanism has several problems:

- It's highly repetitive;
- It forces the user to invent and keep track of numerous error return indications; and
- It mixes the error-handling code in the normal code flow, thereby obscuring both.

These problems don't seem so bad in this simple example, but the complexity rises rapidly as the complexity of the calling code increases. After a while, there's actually more code written to handle errors, checking pesky codes, than the "real" code.

The net result is that error-handling code doesn't get written to handle all the conditions it should.



20 Min. To Go

How Do Exceptions Work?

C++ introduces a totally new mechanism for capturing and handling errors. Called *exceptions*, this mechanism is based on the keywords `try`, `throw`, and `catch`. In outline, it works like this: a function tries (`try`) to get through a piece of code. If the code detects a problem, it `throws` an error indication that the function can `catch`.

Listing 30-1 demonstrates how exceptions work.

Listing 30-1
Exceptions in Action

```
// FactorialExceptionProgram - output the factorial
//                             function with exception-
//                             based error handling
#include <stdio.h>
#include <iostream.h>

// factorial - calculate the factorial of nBase which
//             is equal to nBase * (nBase - 1) *
//             (nBase - 2) * ...
int factorial(int nBase)
{
    // if nBase < 0...
    if (nBase <= 0)
    {
        // ...throw an error
        throw "Illegal argument exception";
    }

    int nFactorial = 1;
    do
    {
        nFactorial *= nBase;
```

```
    } while (--nBase > 1);

    // return the result
    return nFactorial;
}

int main(int nArgc, char* pszArgs[])
{
    // call factorial in a loop. Catch any exceptions
    // that the function might generate
    try
    {
        for (int i = 6; ; i--)
        {
            cout << "factorial("
                 << i
                 << ") = "
                 << factorial(i)
                 << "\n";
        }
    }
    catch(char* pErrorMsg)
    {
        cout << "Detected an error of:"
             << pErrorMsg
             << "\n";
    }
    return 0;
}
```

The `main()` function begins with a block marked by the keyword `try`. One or more catch blocks appear immediately after the `try` block. The `try` keyword is followed by a single argument that looks similar to a function definition.

Within the `try` block, `main()` can do whatever it wants. In this case, `main()` enters a loop that calculates the factorial of progressively smaller numbers. Eventually the program passes a negative number to the `factorial()` function.

When our clever `factorial()` function detects a bogus request, it throws a character string containing a description of the error using the keyword `throw`. At that point, C++ begins looking for a `catch` block whose argument matches the object thrown. It does not finish the remainder of the `try` block. If C++ doesn't

find a `catch` in the current function, it returns to the point of the call and continues the search there. This process is repeated until a matching catch block is found or control passes out of `main()`.

In this example, the thrown error message is caught by the phrase at the bottom of `main()`, which outputs the message. The next statement executed is the return command, which terminates the program.

Why is the exception mechanism an improvement?

The exception mechanism addresses the problems inherent in the error-return mechanism by removing the error-processing path from the normal code path. Further, exceptions make error handling obligatory. If your function doesn't handle the thrown exception, then control passes up the chain of called functions until C++ finds a function that does handle the thrown exception. This also gives you the flexibility to ignore errors that you can't do anything about anyway. Only the functions that can actually correct the problem need to catch the exception. So how does this work?

Trying to Catch the Details Thrown at You

Let's take a closer look at the steps that the code goes through to handle an exception. When a throw occurs, C++ first copies the thrown object to some neutral place. It then looks for the end of the current `try` block. C++ then looks for a matching `catch` somewhere up the chain of function calls. The process is called *unwinding the stack*.

An important feature of stack unwinding is that as each stack is unwound any objects that go out of scope are destructed just as if the function had executed a return statement. This keeps the program from losing assets or leaving objects dangling.

Listing 30-2 is an example program that unwinds the stack.

Listing 30-2
Unwinding the Stack

```
#include <iostream.h>
class Obj
{
  public:
    Obj(char c)
```

```
    {
        label = c;
        cout << "Constructing object " << label << endl;
    }
   ~Obj()
    {
        cout << "Destructing object "  << label << endl;
    }

  protected:
    char label;
};

void f1();
void f2();

int main(int, char*[])
{
    Obj a('a');
    try
    {
        Obj b('b');
        f1();
    }
    catch(float f)
    {
        cout << "Float catch" << endl;
    }
    catch(int i)
    {
        cout << "Int catch" << endl;
    }
    catch(...)
    {
        cout << "Generic catch" << endl;
    }
    return 0;
}
```

Continued

Listing 30-2 *Continued*

```
void f1()
{
    try
    {
        Obj c('c');
        f2();
    }
    catch(char* pMsg)
    {
        cout << "String catch" << endl;
    }
}

void f2()
{
    Obj d('d');
    throw 10;
}
```

Output:

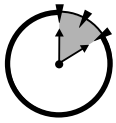
```
Constructing object a
Constructing object b
Constructing object c
Constructing object d
Destructing object d
Destructing object c
Destructing object b
Int catch
Destructing object a
```

First, are the four objects `a`, `b`, `c`, and `d` being constructed as control passes through each declaration before `f2()` throws the `int` 10. Because there is no `try` block defined within `f2()`, C++ unwinds `f2()`'s stack, causing object `d` to be destructed. `f1()` defines a `try` block, but it's only catch phrase is designed to handle `char*`, which does not match the `int` thrown, so C++ continues looking. This unwinds `f1()`'s stack resulting in object `c` being destructed.

Back in `main()`, C++ finds another `try` block. Exiting that block causes object |`b` to go out of scope. The first catch phrase is designed to catch `float`, which is

skipped because it doesn't match our `int`. The next catch phrase matches our `int` exactly, so control stops there. The final catch phrase, which would catch any type of object thrown, is skipped because a matching catch phrase was already found.

A function declared as `fn(...)` accepts any number of any type of arguments. The same applies to `catch` phrases; a `catch(...)` catches anything thrown at it.



10 Min. To Go

What kinds of things can I throw?

A C++ program can throw just about any type of object it wants. C++ uses simple rules for finding the appropriate catch phrase.

C++ searches the `catch` phrase that appears after the first `try` block that it finds. Each `catch` phrase is searched in order, looking for one that matches the object thrown. A "match" is defined using the same rules as used to define an argument match in an overloaded function. If no matching catch phrase is found, then the code continues to the next higher-level `try` block in an ever-outward spiral, until an appropriate catch is found. If no catch phrase is found, the program terminates.

The earlier `factorial()` function threw a character string; that is, it threw an object of type `char*`. The corresponding catch declaration matched by promising to process an object of type `char*`. However, a function can throw just about any type of object it wants to throw.

A program may throw the result of any expression. This means that you can throw as much information as you want. Consider the following class definition:

```
#include <iostream.h>
#include <string.h>

// Exception - generic exception-handling class
class Exception
{
  public:
    // construct an exception object with a description
    // of the problem along with the file and line #
    // where the problem occurred
    Exception(char* pMsg, char* pFile, int nLine)
    {
        this->pMsg = new char[strlen(pMsg) + 1];
```

```
        strcpy(this->pMsg, pMsg);

        strncpy(file, pFile, sizeof file);
        file[sizeof file - 1] = '\0';
        lineNum = nLine;
    }

    // display - output the contents of the
    //           current object to the output stream
    virtual void display(ostream& out)
    {
        out << "Error <" << pMsg << ">\n";
        out << "Occured on line #"
            << lineNum
            << ", file "
            << file
            << endl;
    }

  protected:
    // error message
    char* pMsg;

    // file name and line number where error occurred
    char file[80];
    int lineNum;
};
```

The corresponding throw looks like this:

```
throw Exception("Negative argument to factorial",
                __FILE__,
                __LINE__);
```

`__FILE__` and `__LINE__` are intrinsic `#define`s that are set to the name of the source file and the current line number within that file, respectively.

The corresponding catch is straightforward:

```
void myFunc()
{
    try
    {
         //...whatever calls
    }

     // catch an Exception object
     catch(Exception x)
     {
          // use the built-in display member function
          // to display the object contents to the
          // error stream
          x.display(cerr);
     }
}
```

The catch snags the `Exception` object and then uses the built-in `display()` member function to display the error message.

The version of factorial using the `Exception` class is contained on the accompanying CD-ROM as FactorialThrow.cpp.

The output from executing the factorial program is shown below.

```
factorial(6) = 720
factorial(5) = 120
factorial(4) = 24
factorial(3) = 6
factorial(2) = 2
factorial(1) = 1
Error <Negative argument to factorial>
Occurred on line #59,
          file C:\wecc\Programs\lesson30\FactorialThrow.cpp
```

The Exception class represents a generic error-reporting class. However, this class can be extended by subclassing from it. For example, I might define an InvalidArgumentException class that stores off the value of the invalid argument, in addition to the message and location of the error:

```
class InvalidArgumentException : public Exception
{
  public:
    InvalidArgumentException(int arg, char* pFile, int nLine)
      : Exception("Invalid argument", pFile, nLine)
    {
        invArg = arg;
    }

    // display - output the contents of the current
    //           object to the specified output object
    virtual void display(ostream& out)
    {
        // rely on the base class to output its information
        Exception::display(out);

        // now output our portion
        out << "Argument was "
            << invArg
            << "\n";
    }

  protected:
    int invArg;
};
```

The calling function automatically handles the new InvalidArgumentException because an InvalidArgumentException is an Exception and the display() member function is polymorphic.

The `InvalidArgumentException::display()` **relies on the base class** `Exception` **to output that portion of the object.**

Chaining Catch Phrases

A program may extend its error-handling flexibility by attaching multiple catch phrases to the same `try` block. The following code snippet demonstrates the concept:

```
void myFunc()
{
    try
    {
         //...whatever calls
    }
    // catch a simple character string
    catch(char* pszString)
    {
        cout << "Error:" << pszString << "\n";
    }
    // catch an Exception object
    catch(Exception x)
    {
        x.display(cerr);
    }

    // ...execution continues here...
}
```

In this example, if the thrown object is a simple string, it is captured by the first catch phrase, which outputs the string to the console. If the object is not a character string, it is compared to the `Exception` class. If the object is an `Exception` or any subclass of `Exception`, it is processed by the second phrase.

Because this process continues serially, the programmer must start with the more specific object type and continue to the more general. Thus, to do the following is a mistake:

```
void myFunc()
{
    try
    {
         //...whatever calls
    }
    catch(Exception x)
```

```
        {
            x.display(cerr);
        }
        catch(InvalidArgumentException x)
        {
            x.display(cerr);
        }
    }
```

Done!

Because an `InvalidArgumentException` is an `Exception`, control will never reach the second catch phrases.

The compiler does not catch this coding mistake.

Even if `display()` were a virtual function, it wouldn't make any difference in the above example. The `Exception` catch phrase would call the `display()` function to match the object's run-time class.

Because the generic `catch(...)` phrase catches any exception passing its way, it must appear last in the list of `catch` phrases. Any `catch` appearing after a generic `catch` is unreachable.

REVIEW

The C++ exception mechanism provides a simple, controlled, and extensible mechanism for handling errors. It avoids the logical complexity that can occur using the standard error-return mechanism. It also makes certain that objects are destructed properly as they go out of scope.

- The conventional error-handling technique of returning an otherwise illegal value to indicate the type of error has severe limitations. First, there is a limited amount of information that can be encoded. Second, the calling function is forced to handle the error, either by processing it or by returning the error, whether it can do anything about the error or not. Finally, many functions have no illegal value that can be returned.

- The exception-error technique enables functions to return a virtually unlimited amount of data. In addition, if a calling function chooses to ignore an error, the exception continues to propagate up the chain of functions until it finds one that can handle the error.
- Exceptions can be subclasses, giving the programmer increased flexibility.
- Catch phrases may be chained to enable the calling function to handle different types of errors differently.

QUIZ YOURSELF

1. Name three limitations of the error-return technique. (See "Conventional Error Handling.")
2. Name the three new keywords used by the exception handling technique. (See "How Do Exceptions Work.")