Error Handling and Exceptions

C++ has a mechanism for capturing and handling errors called exceptions. It uses the keywords try, throw and catch (which means you can’t use these words as variable names). In brief, it works like this: A function tries to get through a piece of code. If the code has a problem, it throws something (like a character string, number, or a class) that the calling function must catch and process. A simple example is

```cpp
#include <iostream>

//factorial - compute factorial
int factorial(int n)
{
    // you can’t handle negative values of n

    if (n < 0)
    {
        throw "Argument for factorial negative";
    }

    // go ahead and calculate factorial

    int accum = 1;
    while (n > 0)
    {
        accum *= n;
        n--;
    }
    return accum;
}

// any old function will do

void someFunc()
{
    try
    {
        // this will generate an exception
        cout << "Factorial of -1 is " << factorial(-1) << endl;

        // control will never get here
        cout << "Factorial of 10 is " << factorial(10) << endl;
    }
}
someFunc() starts out by creating a block of code that begins with the word try. Within this block, the program can do whatever it wants. In this case, someFunc() attempts to calculate the factorial of a negative number. The factorial() function detects the erroneous request and throws an error indication using the throw keyword. Control passes to the catch phrase, which immediately follows the closing brace of the try block. The second call to factorial() is not performed.

Exception handling is not used that much in scientific programming, but it is a good way to make your program idiot-proof. It is particularly useful if your program is to be used interactively. Suppose you are writing a program that produces windows and menus. If something goes awry, your windows will hang. Using exceptions allows the program (and windows) to exit gracefully or even recover by trying again.

The thing that is thrown is called an “exception.” Exceptions are caught by an “exception handler,” which appears immediately following the try block. Here is the syntax:

```cpp
try{
    ...
    // "throw" may be contained in a function that is called here
} catch(arguments){
    statement;   // exception handler body
    ...
}
```

There can be one or more catch phrases associated with a try block.

### Unwinding the Stack

Let’s take a closer look at how exceptions are handled. When the throw occurs, C++ first copies the thrown object to some neutral place. It then begins looking for the end of the current try block. If a try block is not found in the current function, the function’s execution is terminated and control passes to the calling function. A search is then made of that function. If no try block is found there, control passes to the function that called it, and so on up the stack of calling functions. This process is called “unwinding the stack.” Basically the computer searches for a try block by going to higher and higher levels of the program. As stack unwinding occurs, 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.
When the encasing try block is found, the code searches for the first catch phrase immediately following the closing brace of the try block. If the object thrown matches the type of object that the catch phrase expects, then control passes to that catch phrase. If not, a check is made of the next catch phrase. If no matching catch phrases are found, the code searches the next higher level for a try block in an ever outward spiral until an appropriate catch phrase can be found. If no catch phrase is found, the program is terminated.

Consider the following example:

```c++
#include <iostream.h>
class Obj
{
    public:
        Obj(char c)
        {
            label = c;
            cout << "Constructing object " << label << endl;
        }
        ~Obj()
        {
            cout << "Destructing object " << label << endl;
        }
    private:
        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;
}
void f1()  
{
   try  
   {
      Obj c('c');
      f2();
   }
   catch(char* pMsg)  
   {
      cout << "String catch" << endl;
   }
}

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

The output looks like this:

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 you see the four objects a, b, c, and d being constructed as control passes through each declaration before f2() throws int 10. Because no try block is defined in f2(), C++ unwinds f2()’s stack, causing object d to be destructed. f1() defines a try block, but its only catch phase is designed to handle char*, which does not match the int
thrown. Therefore C++ continues looking. It 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 floats that don't match our int, so it's skipped. The next catch phrase matches our int exactly, so control stops there. The final catch phrase, which would catch any object thrown, is skipped because a matching catch phrase was already found.

**Throwing Classes**

So far we have thrown phrases (char*) and integers. In fact we can throw any type of object, even classes. For example, suppose we make an Error class and use it in our factorial example:

```cpp
#include <iostream.h>

class Error {

public:
    Error(int nErrorCode, char* pReason): // constructor
        nCode(nErrorCode), Reason(pReason)
    {}

    void display(ostream& out) {
        out << "Error Code = " << nCode << " : " << Reason;
    }

private:
    int nCode;
    char* Reason;
};

//factorial - compute factorial
int factorial(int n) {
    // you can't handle negative values of n

    if (n < 0)
    {
        int ErrorCode = -1;

        // Throw Error class
        throw Error(ErrorCode,"Argument for factorial negative");
    }

    // go ahead and calculate factorial
```
int accum = 1;
while (n > 0)
{
    accum *= n;
    n--;
}
return accum;

void main()
{
    int result;

    try
    {
        // this will generate an exception
        result = factorial(-1);
        cout << "Factorial of -1 is " << result << endl;

        // control will never get here
        cout << "Factorial of 10 is " << factorial(10) << endl;
    }

    // control passes here
    catch(Error& error)
    {
        cout << "Fielded error: ";
        error.display(cout);
        cout << endl;
    }
}

We can make this even fancier by using inheritance and polymorphism. In particular, let’s make Exception a base class and a derived class called InvalidArgumentException. Let’s make a somewhat useful error class that tells the line number and file where the error occurred. We will have factorial throw the exception.

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

//Exception-generic exception handling class
class Exception
{
    public:
        Exception(char* pMsg, char* pFile, int nLine)
        {
            strcpy(msg,pMsg);
            msg[sizeof(msg)-1] = '\0'; // make sure msg is terminated
            strcpy(file,pFile);
            file[sizeof(file)-1] = '\0';
            lineNum = nLine;
        }

        virtual void display(ostream& out)
        {
            out << "Error: " << msg << endl;
            out << "Occurred on line number " << lineNum
                << ", in file called " << file << endl;
        }

    protected:
        // error message
        char msg[80];

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

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

        int argument()
        {return invArg;}

        virtual void display(ostream& out)
        {
            Exception::display(out);
        }
    }
out << "Argument was " << argument() << endl;

protected:
    int invArg;
};

//factorial - compute factorial
int factorial(int n)
{
    // you can't handle negative values of n

    if (n < 0)
    {
        int ErrorCode = -1;

        // Throw Exception class

        throw Exception("Argument for factorial negative", __FILE__,__LINE__);

        // __FILE__ and __LINE__ are intrinsic #defines that are set to the
        // name of the source file and the current line number in that file,
        // respectively.
    }

    else if(n>100)
    {
        // Throw derived class

        throw InvalidArgumentException(n, __FILE__, __LINE__);
    }

    // go ahead and calculate factorial

    int accum = 1;
    while (n > 0)
    {
        accum *= n;
        n--;
    }
    return accum;
}

void main()
{
int result;

try
{
    // this will generate an exception and throw the derived class
    result = factorial(150);
    cout << "Factorial of 150 is " << result << endl;

    // this will generate an exception
    result = factorial(-1);
    cout << "Factorial of -1 is " << result << endl;

    // control will never get here
    cout << "Factorial of 10 is " << factorial(10) << endl;
}

// control passes here
catch(Exception& x)
{
    // use the built-in display member function
    x.display(cerr);
}

Suppose that **InvalidArgumentException** is thrown. If we had written `catch(Exception x)`, we would only get the base class portion of **InvalidArgumentException** due to slicing. If we pass `Exception` by value, the default copy constructor for the base class is used to copy the base class part of **InvalidArgumentException** to catch. To avoid slicing, we pass `Exception` by reference.