Chapter 6

Exception Handling

Two common types of error in a program are:

1) **Syntax error** (arises due to missing semicolon, comma, and wrong prog. constructs etc)
2) **Logical error** (wrong understanding of the problem or wrong procedure to get the solution)

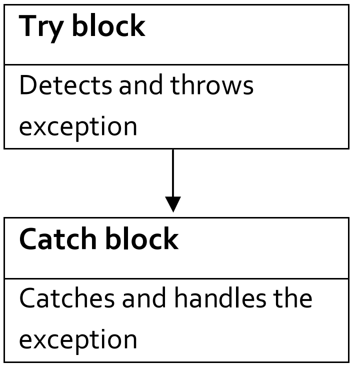
Exceptions

Exceptions are the errors occurred during a program execution. Exceptions are of two types:

- Synchronous (generated by software i.e. division by 0, array bound etc).
- Asynchronous (generated by hardware i.e. out of memory, keyboard etc).

Exception handling mechanism

- C++ exception handling mechanism is basically built upon three keywords namely, try, throw and catch.
- Try block hold a block of statements which may generate an exception.
- When an exception is detected, it is thrown using a throw statement in the try block.


Try block
Detects and throws exception
Catch block
Catches and handles the exception

- A try block can be followed by any number of catch blocks.

The general form of **try** and **catch** block is as follows:

```
try
{
        /* try block; throw exception*/
}
catch (type1 arg)
{
        /*  catch block*/
}

……………………
```

```
……………………
catch (type2 arg)
{
        /*  catch block*/
}
```

The exception handling mechanism is made up of the following elements:

- try blocks
- catch blocks
- throw expressions

Program 7.1 **Write a program to find x/y, where x and y are given from the keyboard and both are integers.**

Solution:

```
#include<iostream.h>
void main()
{
        int x, y;
        cout<<"enter two number"<<endl;
        cin>>x>>y;
        try
        {
                if(y!=0)
                {
                z=x/y;
                cout<<endl<<z;
                }
                else
                {
                throw(y);
                }
        }
        catch(int y)
        {
        cout<<"exception occurred: y="<<y<<endl;
        }
}
```

Output:

```
Enter two number
6  0
exception occurred:y=0
```

A **try** block can be localized to a function. When this is the case, each time the function is entered, the exception handling relative to that function is reset. For example, examine this program.

Program 7.2

```
#include <iostream.h>
void Xhandler(int test)
{
        try
        {
        if(test) throw test;
        }
        catch(int i)
        {
        cout << "Caught Exception #: " << i << '\n';
        }
}
void  main()
{
        cout << "Start\n";
        Xhandler(1);
        Xhandler(2);
        Xhandler(0);
        Xhandler(3);
        cout << "End";
}
```

Output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught Exception #: 3
End
```

As you can see, three exceptions are thrown. After each exception, the function returns. When the function is called again, the exception handling is reset.

It is important to understand that the code associated with a **catch** statement will be executed only if it catches an exception. Otherwise, execution simply bypasses the **catch** altogether. (That is, execution never flows into a **catch** statement.) For example, in the following program, no exception is thrown, so the **catch** statement does not execute.

```
#include <iostream.h>
void  main()
```

```
{
            cout << "Start\n";
            try
            {
            cout << "Inside try block\n";
            cout << "Still inside try block\n";
            }
            catch (int i)
             {
            cout << "Caught an exception -- value is: ";
            cout << i << "\n";
            }
            cout << "End";
}
```

Output:

```
Start
Inside try block
Still inside try block
End
```

Catching Class Types

An exception can be of any type, including class types that you create. Actually, in real-world programs, most exceptions will be class types rather than built-in types. Perhaps the most common reason that you will want to define a class type for an exception is to create an object that describes the error that occurred. This information can be used by the exception handler to help it process the error. The following example demonstrates this.

Program 7.3

```
#include <iostream.h>
#include <cstring.h>
class MyException
{
            public:
            char str_what[80];
            int what;
            MyException() { *str_what = 0; what = 0; }
            MyException(char *s, int e)
            {
            strcpy(str_what, s);
            what = e;
            }
};
void main()
{
            int i;
            try {
            cout << "Enter a positive number: ";
            cin >> i;
            if(i<o)
            throw MyException("Not Positive", i);
            }
            catch (MyException e) { // catch an error
            cout << e.str_what << ": ";
            cout << e.what << "\n";
            }
}
```

Output:

```
Enter a positive number: -4
Not Positive: -4
```

The program prompts the user for a positive number. If a negative number is entered, an object of the class **MyException** is created that describes the error. Thus, **MyException** encapsulates information about the error. This information is then used by the exception handler. In general, you will want to create exception classes that will encapsulate information about an error to enable the exception handler to respond effectively.

Using Multiple catch Statements

As stated, you can have more than one **catch** associated with a **try**. In fact, it is common to do so. However, each **catch** must catch a different type of exception. For example, this program catches both integers and strings.

Program 7.4

```
#include <iostream.h>
void Xhandler(int test)
{
            try
            {
            if(test) throw test;
            else throw "Value is zero";
            }
            catch(int i)
            {
```

```
            cout << "Caught Exception #: " << i << '\n';
            }
            catch(const char *str) {
            cout << "Caught a string: ";
            cout << str << '\n';
            }
}
void main()
{
            cout << "Start\n";
            Xhandler(1);
            Xhandler(2);
            Xhandler(0);
            Xhandler(3);
            cout << "End";
}
```

Output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught a string: Value is zero
Caught Exception #: 3
End
```

As you can see, each **catch** statement responds only to its own type.

Handling Derived-Class Exceptions

You need to be careful how you order your **catch** statements when trying to catch exception types that involve base and derived classes because a **catch** clause for a base class will also match any class derived from that base. Thus, if you want to catch exceptions of both a base class type and a derived class type, put the derived class first in the **catch** sequence. If you don't do this, the base class **catch** will also catch all derived classes. For example, consider the following program.

Program 7.5

```
#include <iostream.h>
class B
{
};
class D: public B
{
};
```

```
void main()
{
        D derived;
        try
        {
        throw derived;
        }
        catch(B b)
        {
        cout << "Caught a base class.\n";
        }
        catch(D d)
        {
        cout << "This won't execute.\n";
        }
}
```

Here, because **derived** is an object that has **B** as a base class, it will be caught by the first **catch** clause and the second clause will never execute. Some compilers will flag this condition with a warning message. Others may issue an error. Either way, to fix this condition, reverse the order of the **catch** clauses.

Exception Handling Options

There are several additional features and nuances to C++ exception handling that make it easier and more convenient to use. These attributes are discussed here.

Catching All Exceptions

In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type. This is easy to accomplish. Simply use this form of **catch**.

```
catch (...) {
// process all exceptions
}
```

Here, the ellipsis matches any type of data. The following program illustrates **catch (...)**.

Program 7.6

```
#include <iostream.h>
void Xhandler(int test)
{
        try
        {
        if(test==0) throw test; // throw int
        if(test==1) throw 'a'; // throw char
        if(test==2) throw 123.23; // throw double
```

```
		}
		catch(...)
		{
		cout << "Caught One!\n";
		}
}
void  main()
{
		cout << "Start\n";
		Xhandler(0);
		Xhandler(1);
		Xhandler(2);
		cout << "End";
}
```

Output:

```
Start
Caught One!
Caught One!
Caught One!
End
```

Rethrowing an Exception

If you wish to rethrow an expression from within an exception handler, you may do so by calling throw, by itself, with no exception. This causes the current exception to be passed on to an outer try/catch sequence. The most likely reason for doing so is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception and a second handler copes with another. An exception can only be rethrown from within a catch block (or from any function called from within that block). When you rethrow an exception, it will not be recaught by the same catch statement. It will propagate outward to the next catch statement. The following program illustrates rethrowing an exception, in this case a char * exception.

Program 7.7

```
#include <iostream.h>
void Xhandler()
{
		try
		{
		throw "hello"; // throw a char *
		}
		catch(const char *)
		{
		cout << "Caught char * inside Xhandler\n";
		throw ; // rethrow char * out of function
		}
}
void main()
{
		cout << "Start\n";
		try
		{
		Xhandler();
		}
		catch(const char *)
		{
		cout << "Caught char * inside main\n";
		}
		cout << "End";
}
```

Output:

```
Start
Caught char * inside Xhandler
Caught char * inside main
end
```

Understanding terminate( ) and unexpected( )

As mentioned earlier, **terminate()** and **unexpected()** are called when something goes wrong during the exception handling process. These functions are supplied by the Standard C++ library. Their prototypes are shown here:

void terminate( );

void unexpected( );

These functions require the header **<exception>**.

The **terminate()** function is called whenever the exception handling subsystem fails to find a matching **catch** statement for an exception. It is also called if your program attempts to rethrow an exception when no exception was originally thrown. The **terminate()** function is also called under various other, more obscure circumstances. For example, such a circumstance could occur when, in the process of unwinding the stack because of an exception, a destructor for an object being destroyed throws an exception. In general, **terminate()** is the handler of last resort when no other handlers for an exception are available. By default, **terminate()** calls **abort()** .

Assignment 6

Short Type Question

1. What is exception?
2. Differentiate between syntax error and logical error.

Long Type Questions

1. What is an exception? Describe the mechanism of exception handling with suitable example?
2. What is a generic catch block? What are the restrictions while using a generic catch block? Explain with an example.