UNIT 3 EXCEPTION HANDLING

Structure **Page Nos.**

3.0	Introduction	46
3.1	Objectives	46
3.2	Exceptions in C++ Programs	46
3.3	Try, Throw and Catch Expressions	48
3.4	Specifying Exception Types	53
3.5	Summary	58
3.6	Answers to Check Your Progress	58
3.7	Further Readings	58

3.0 INTRODUCTION

In the units covered so far, we have talked about various features provided by C++ to design different programs. The programs which are written correctly produce the desired output when input data is supplied to them. However, in some cases programs may behave in an unexpected manner. One of the reasons that may lead to this situation is when we provide inappropriate input data (something that the programmer never expected or anticipated). These situations are unexpected and that is why they are termed exceptions. Exceptions are different from syntactical and logical errors but they also cause the program to misbehave. Exceptions are usually encountered at run time. In order to ensure that programs work correctly under all conditions, we have to incorporate mechanisms to identify and handle different exceptions that may occur in a program.

This unit introduces the nature and type of exceptions that may occur in a C++ program. It then describes the steps in exception handling. Then the syntax and use of try, throw and catch expressions are explained with appropriate examples. This unit tries to present a comprehensive picture of the exception handling mechanisms in C++ and their use in designing correct and robust programs.

3.1 OBJECTIVES

At the end of the unit, you should be able to:

- know what is exceptions and how to handle them;
- use try, catch and throw expressions to identify and handle exceptions;
- describe the standard exception hierarchy;
- appreciate the usefulness of exception handling mechanisms in C++; and
- write robust and fault tolerant C++ programs.

3.2 EXCEPTIONS IN C++ PROGRAMS

The name ‘exception’ implies that the problem is one which occurs infrequently- if the “rule” is that a statement normally executes correctly, then the “exception to the rule” is that a problem occurs.

The term exception itself implies an unusual condition. Exceptions are anomalies that may occur during execution of a program. Exceptions are not errors (syntactical or logical) but they still cause the program to misbehave. They are unusual conditions which are generally not expected by the programmer to occur. An exception in this sense is an indication of a problem that occurs during a programs’s execution. The

typical exceptions may include conditions like divide by zero, access to an array outside its range, running out of memory etc.

For example: Consider the following code segment:

```
# include <iostream>
                    int main()
                    {
                      int x,y;
                      cout << "enter values of x & y \n";
                      cin >> x;
                      cin >> y;
                      cout << "result of x divided by y is:" <<
x/y;
                     }
```

This program reads values of variables x and y from standard input and produces output of x divided by y, which is then displayed on the standard output. However, please note that if the value of y read from the keyboard is '0', then the program witnesses a divide by zero situation. In this case, the result will be infinite and program terminates.

To deal with these unexpected conditions, C++ provides an exception handling mechanism that detects and handles exceptions in a predefined way. Exception handling mechanism was not part of the original C++ specifications. It was added later and now almost all compilers support this feature. C++ exception handling mechanism provide with a scheme to identify and handle predictable exceptions that may occur during program execution. It may kindly be noted that exception handling mechanism needs to be incorporated explicitly in the program to handle the exceptions and that it may be able to handle only those exceptions that are provisioned in exception handling statements.

Exceptions can be of two kinds: *asynchronous and synchronous*. The exceptions that are caused by events beyond the control of the program (such as keyboard interrupts) are called asynchronous exceptions. The other unusual conditions (such as overflow, out of range array index) that are part of the program are called synchronous exceptions. The C++ exception handling mechanism is designed to handle synchronous type of exceptions only. It provides a means to detect, report and act on an unusual condition. The exception handling mechanism C++ deals with exceptions by performing following tasks:

a. Identify the problem (**hit** the exception)
b. Inform that an exception has occurred (**throw** the exception)
c. Exception handler catches the exception information (**catch** the exception)
d. Exception handler takes corrective actions (**handle** the exception)

These tasks are incorporated in C++ exception handling mechanism in two segments, one to detect (**try**) and inform (**throw)** about the exception, and the other to catch the exception and take appropriate actions to handle it.

As indicated earlier, the C++ exception handling mechanism is built upon following three expressions:

- Try
- Throw
- Catch

The expression **try** is used to preface (enclose in braces) that block which may generate exceptions. This block is often termed as try block. The **throw** expression is invoked when an exception is detected. It informs the catch block that an exception has occurred. The exception handling code is enclosed in **catch** block. The catch block catches the exception thrown by the throw expression and handles it in a predefined manner. The Figure 3.1 further shows the use of these three expressions and their relationship:

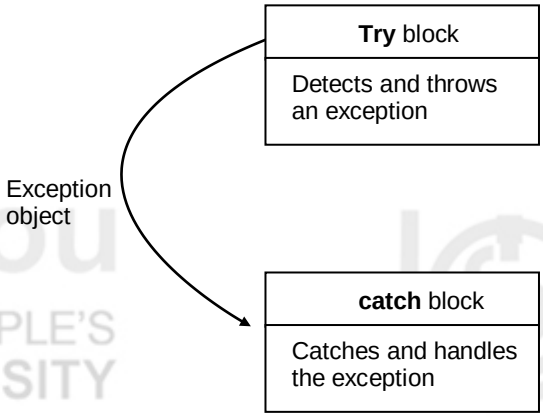

Figure 3.1 : Exception Handling

3.3 TRY, THROW AND CATCH EXPRESSIONS

The keyword 'try' is used to enclose the statements that may generate an exception. These statements begin with a try keyword and are enclosed in braces. When the try block encounters an exception, it uses the throw keyword to pass the information to the exception handling block. The exception handling block is enclosed within a block preceding the catch keyword. This catch block should immediately follow the try block that throws the exception. The general syntax of these statements and blocks is as follows:

```
……..
……..
try
{
  ……
  ……
  throw exception;
  ……
  ……
 }
catch (type arg)
{
  ……..
  ……..
  ……..
 }
……
……
```

As some statement within the try block of the program generates an exception, it is thrown using the throw statement. At this time, the program control transfers to the

catch block. The throw contains an argument named exception which is passed to the catch block as an argument. However, the catch block handles this exception only if the arguments passed from throw matches with the argument specified in the catch block. In case the exceptions that may be generated could be of different types, then multiple catch blocks will be required. This can be done by writing these catch blocks one after another. When an exception is thrown, the exception object is compared with the argument in the catch blocks written in succession. The first catch block matching the exception object is executed. We will see use of multiple catch blocks in the next part of the chapter. If the exception object thrown does not match with the argument specified in catch block, then the program gets terminated by automatic invocation of abort() function. Sometimes exceptions are thrown from functions that are invoked within the try block. The point at which this throw is executed is called throw point. After an exception is thrown to the catch block, control cannot return back to the throw point. The Figure 3.2 demonstrates this situation when a function invoked from within the try block throws an exception.

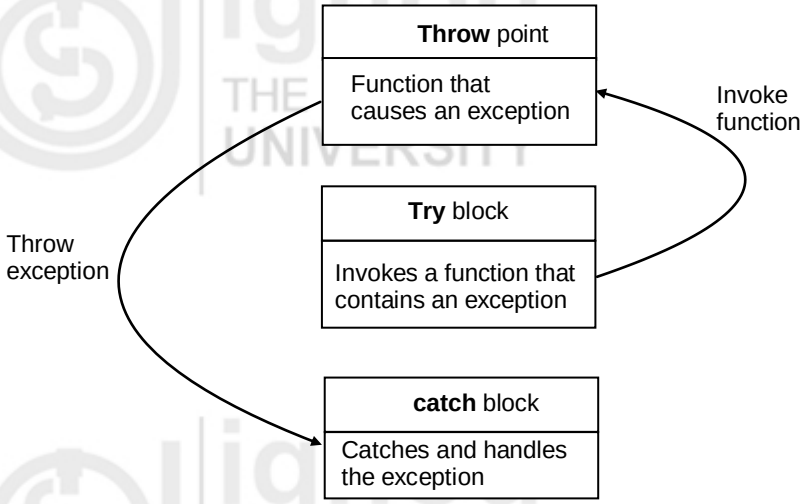

Figure 3.2 Throw, try and catch blocks for exception handling

We present following two code segments that demonstrate use of try, throw and catch expressions for exception handling. The first program presents a simple example of use of a single try and catch block. The second program presents an example where the exception is thrown from a function invoked from within the try block.

```
# include <iostream>
int main()
{
        int x,y;
        cout << "enter values of x & y \n";
        cin >> x;
        cin >> y;
        int a = x-y;
        try
        {
         if (a!=0)
         {
          cout << "Result (x/a) =" << x/a;
         }
         else
         {
          throw (a);
         }
        }
```

```
        catch (int i)
        {
          cout << "Exception caught a=" << a;
        }
        return(0);
        }
```

```
# include <iostream>
void divide (int x, int y, int z)
{
        cout << "inside function \n";
        if (x-y)!=0)
        {
          int r = z/ (x-y);
          cout << "result =" << r << "\n";
        }
        else
        {
          throw (x-y);
        }
 }

int main()
{
        Try
        {
         cout << inside try block \n";
         divide (10,20,30);
         divide(10,10,20);
        }
        catch (int i)
        {
          cout << "Caught the exception";
        }
        return(0);
}
```

You would see that in the first program, if input values are 20 and 15 then the program runs correctly without any exception with result being 4. On the other hand, if input given is 10 and 10, the program would detect an exception and display it as "0" value. Similarly, in the second program for the first invocation of divide (), the result will be -3, whereas the second invocation will result into an exception.

The throw statement can take multiple forms:

```
throw (exception);
throw exception;
throw;
```

The first two throw statements pass the exception object to the catch handler, whereas the third throw statement without any exception object is used to rethrow an exception from within a catch block.

A catch handler may rethrow the exception, without handling it, to the next catch block. This is equivalent to passing the exception to the next enclosing catch block within the scope of try/catch sequence. Please note that a rethrown exception is not

caught by the same catch handler or any other catch handler in that group, but by an appropriate catch handler in the outer try/catch sequence only. This also applies to those exceptions that may be detected within a catch handler block itself.

Multiple Catch Statements

We have seen earlier that a catch block looks like a function definition. It has a type argument which specifies the type of exception that this catch block may handle followed by statements (enclosed within braces) to process the exception. After executing these statements, the control goes to the statement immediately following the catch block. It is also possible that a program may have more than one exception condition. In such cases multiple catch statements are needed to handle the different types of exceptions. Now when an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that matches the thrown exception object type is executed. Rest of the catch blocks are then skipped and the control goes to first statement following the last catch block. This indicates that in case of multiple catch blocks matching the thrown exception, only the first matching catch block is executed. The following program demonstrates the use of multiple catch blocks:

```
#include<iostream>
void test(int x)
{
        try
        {
          if (x==1) throw x;
          else
            if (x==0) throw 'x';
          else
            if(x==-1) throw 1.0;
        }
        catch (char c)
        {
          cout << "caught a character \n";
        }
        catch(int m)
        {
          cout << "caught an integer \n";
        }

        catch(double d)
        {
          cout << "caught a double \n";
        }
}

int main()
{
        cout << "testing multiple catches \n";
        cout << "x==1 \n";
        test(1);
        cout << "x==0 \n";
        test(0);
        cout << "x==-1 \n";
        test(-1);
        cout << "x==2 \n";
        test(2);
```

```
        return(0);
}
```

As you can see this program has multiple catch blocks, each handling exception objects of different types. One integer, other character and the third one a double exception argument. The last invocation of test with argument 2 does not throw any exception and hence no catch block is invoked.

Catch all exceptions

There are many situations where it may be very difficult to anticipate in advance all possible types of exceptions that may occur in a program. C++ provides with a mechanism to cope with this problem in form of a catch (…) expression. This type of catch statement catches all exceptions, unlike only one exception being caught. The general syntax of use of this kind of catch expression is as follows:

```
catch(…)
{
//statements for processing
//all exceptions
}
```

The following example presents a case of use of this kind of catch expression:

```
# include <iostream>
void test (int x)
{
        try
        {
          if (x==0) throw x;
          if (x==-1) throw 'x';
          if (x==1) throw 1.0;
        }
        catch (…)
        {
         cout << "caught an exception \n";
        }
}
int main()
{
        cout << "testing generic catch \n";
        test (-1);
        test (0);
        test(1);
        return(0);
}
```

You may see that this program prints the line "caught an exception" three times as follows:

Output:

caught an exception
caught an exception
caught an exception

This is because all the exceptions (x getting values -1, 0 and 1) are caught by the same catch handler block. This is the reason why this kind of catch block is termed as *catch all expression.* This property can make this kind of catch all expression to be placed as default catch handler. This default handler may then be used to catch all those exceptions which are not handled explicitly. However, one must be careful to place this catch all expression in the last place of catch handlers.

☞ Check Your Progress 1

1) List five common examples of exceptions.

……

……

……

2) If no exceptions are thrown in a try block, where does control proceed to after the try block completes the execution?

……

……

……

3) What happens if an exception is thrown outside a try block?

……

……

……

4) What does the statement throw do?

……

……

……

5) What happens if several handlers match the type of thrown object?

……

……

……

3.4 SPECIFYING EXCEPTION TYPES

We have seen earlier that the exception type is reported by the throw statement to the catch handler, which then takes appropriate action on it. It is also possible to restrict a function to throw only certain specified exceptions. This can be done by adding a throw list clause to the function definition. The general syntax of doing this exception type specification is as follows:

```
type function (arg-list) throw (type-list)
{
……..
……..
}
```

The type-list after throw specifies the type of exceptions that may be thrown. An attempt to throw another type of exception will cause abnormal program termination. In case we want to prevent a function from throwing any exception at all, we may do so by making the type-list empty, i.e., simply writing throw ( ) in the function header line. However, these specifications operate only when the function is called back from a try block and it reports back the exceptions to that try block. It does not apply if exception is thrown within the function code itself. An example to demonstrate this scenario is as follows:

```
#include <iostream>
void test(int x) throw (int, double)
{
        if (x==0) throw 'x';
        else
          if (x==1) throw x;
          else
            if (x==-1) throw 1.0;
}
int main()
{
        try
        {
          cout << :testing throw specifications \n";
          cout << "x==0 \n";
          test(0);
          cout << "x==1 \n";
          test(1);
          cout << "x==-1 \n";
          test(-1);
          cout << "x==2 \n";
          test(2);
        }
        catch(cahr c)
        {
           cout << "caught a character \n";
        }
        catch (int m)
        {
          cout << "caught an integer \n";
        }
        catch (double d)
        {
          cout << caught a double \n";
        }
        return (0);
}
```

You may note that this program tries to throw a char exception object in the very first invocation of test(). This results in an abnormal termination of the program, since the test can throw only exceptions of type int and double.

Program Output:

```
testing throw specifications
x==0
caught a character
```

Throwing an exception that has not been declared in a function's exception specification causes a call to function unexpected. The compiler will not generate a compilation error if a function contains a throw expression for an exception not listed in the function's specification. An error occurs only when that function attempts to throw that exception at execution time. To avoid surprises at execution time, its better to carefully check the code to ensure that functions do not throw exceptions not listed in their specifications.

The *function unexpected* calls, the function registered with the function set_unexpected (defined in header file <exception>). If no function has been registered in this manner, function terminate is called by default. The function set_terminate can specify the function to invoke when terminate is called. Otherwise, terminate calls abort, which terminates the program without calling the destructors of any remaining objects of automatic or static storage class. This could lead to resource leaks when a program terminates prematurely.

Exceptions and Stack Unwinding

When an exception is thrown but not caught in a particular scope, the function call stack is "unwound" and an attempt is made to catch the exception in the next outer try….catch block. Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables in that function are destroyed and control returns to the statement that originally invoked that function. If a try block encloses that statement, an attempt is made to catch the exception. If a try block does not enclose the statement, stack unwinding occurs again. If no catch handler ever catches this exception, function terminate is called to terminate the program. The following programming example demonstrates stack unwinding:

```
#include <iostream>
#include <stdexcept>
using namespace std;

//function3 throws runtime error
void function3() throw (runtime_error)
{
  cout << "in function3" << endl;
// no try block, stack unwinding occurs, return control to function2
  throw runtime_error ("runtime_error in function3");
}

//function2 invokes function3
void function2() throw (runtime_error)
{
  cout << "function3 is called inside function2" << endl;
  function3(); // stack unwinding occurs, return control to function1
}

//function1 invokes function2
void function1() throw (runtime_error)
{
  cout << "function2 is called inside function1" << endl;
  function2(); //stack unwinding occurs, return control to main
}

int main()
{
  //invoke function1
```

```
  try
   {
     cout << "function1 is called inside main" << endl;
     function1();
    }
  catch (runtime_error & error)
   {
     cout << "exception occurred:"  << error.what() << endl;
     cout << " exception handled in main" << endl;
   }
}
```

The program will produce the following output:

function1 is called inside main
function2 is called inside function1
function3 is called inside function2
in function3
exception occurred: runtime_error in function3
exception handled in main

Exception handling and Constructors and Destructors

It is worth discussing that what happens if an exception is thrown while executing a constructor? Since the object is not yet fully constructed, its destructor would not be called once the program control goes out of the object's context. And, if the constructor had reserved some memory before the exception was raised, then there would be no mechanism to prevent such memory leak. Hence. Appropriate exception handling mechanism must be implemented pertaining to the constructor routine to handle exceptions that occur during object construction.
Where to catch the exception is another important issue here. Whether it should be done inside the constructor block or inside the main. If we allow the exception to be handled inside main then we would not be able to prevent the memory leak situation. Therefore, we must catch the exception within the constructor block so that we get chance to free up any reserved memory spaces. However, we must simultaneously rethhrow the exception to be appropriately handled inside the main block.

Exceptions and Inheritance

Like the normal inheritance concept, various exception classes can be derived from a common base class. If a ctach handler catches a pointer or reference to an exception objectof a base-class type, it also can catch a pointer or reference to all objects of class publicly derived from that base class- this allows for polymorphic processing of related errors. Using inheritance with exceptions enables an exception handler to catch related errors with concise notation. One approach is to catch each type of pointer or reference to a derived-class exception object individually, but a more concise approach is to catch pointers or references to base-class exception objects nstead. Also, catching pointers or references to derived-class exception object individually is error prone, especially if you forget to test explicitly for one or more of the derived-class pointer or reference types.

Standard Library Exception Hierarchy

As we know that exceptions fall nicely into a number of categories. The C++ standard library includes a hierarchy of exception classes. The hierarchy is headed by base-class exception (defined in header file <exception>), which contains virtual function what, which derived base classes can override to issue appropriate error messages. Immediate derived classes of base class exception include runtime_error and logic_error (both defined in header <stdexcept>), each of which has several derived classes. Also derived from exception are the exceptions thrown by C++ operators – for example, bad_alloc is thrown by new, bad_cast is thrown by dynamic_cast and

bad_typeid is thrown by typeid. Including bad_exception in the throw list of a function means that, if an unexpected exception occurs, function unexpected can throw bad_exception rather than terminating the program's execution or calling another function specified by set_unexpected. The class logic_error is the base class of several standard exception classes that indicate errors in program logic, whereas class runtime_error is the base class of several other standard exception classes that indicate execution-time errors. The Figure 3.3 below presents an overview of standard library exception classes.

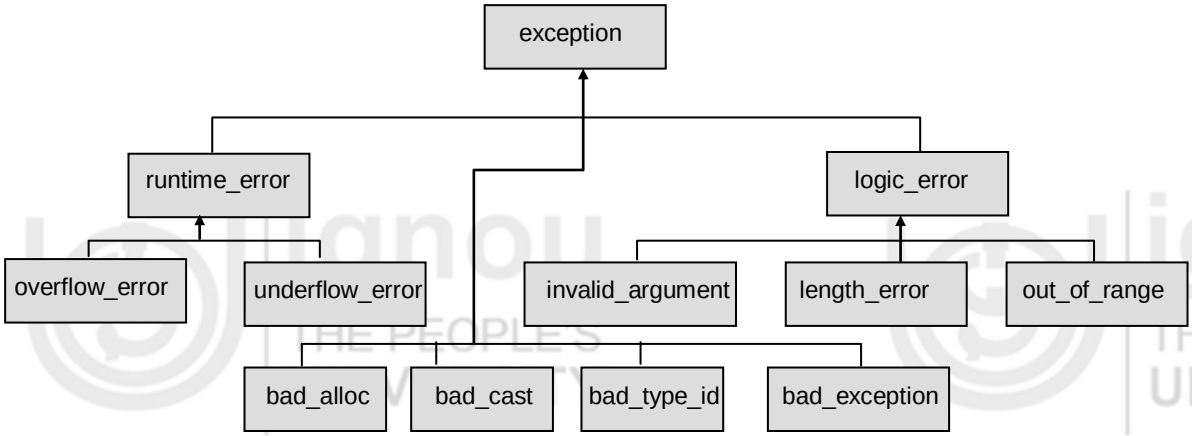

Figure 3.3 Some of the Standard Library Exception Classes

☞ Check Your Progress 2

1) What happens if no catch handler matches the type of a thrown object?

……

……

……

2) Must throwing an exception cause program termination?

……

……

……

3) What happens when a catch handler throws an exception?

……

……

……

4) How do you restrict the exception type that a function can throw?

……

……

……

5) What happens if a function throws an exception of a type not allowed by the exception specification for the function?

……

……

……

3.5 SUMMARY

Exceptions are a kind of problem that may occur when a program is executed. C++ provides an exception handling mechanism to handle synchronous exceptions. An exception is handled by a group of try, throw and catch expressions. The block that may cause a possible exception is enclosed within a try block. The exceptional situation occurring in the try block is reported by throw expression to an exception handling block, called catch block. When an exception is not caught the program is terminated. The throw expression passes the exception object to an appropriate catch block. A program may have more than one catch block to handle different kinds of exceptions. We can also have a catch all expression that can be used as a default handler. A catch block may also rethrow an exception without processing it. We may also restrict the type of exception objects that a function may throw. The try-throw-catch mechanism in C++ is quite useful to provide for dealing with unexpected situations that may occur at runtime in a program.

3.6 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) Insufficient memory to satisfy a new request, array subscript out of bounds, arithmetic overflow, division by zero, invalid function parameters.
2) The exception handlers (in the catch handlers) for that try block are skipped, and the program resumes execution after the last catch handler.
3) An exception thrown outside a try block causes a call to terminate.
4) It passes the exception object to the catch handler. If it occurs within a catch block, it rethrows the exception.
5) The first matching exception handler after the try block is executed.

Check Your Progress 2

1) This causes the search for a match to continue in the next enclosing try block if there is one. As this process continues, it might eventually be determined that there is no handler in the program that matches the type of thrown object. In this case, program is aborted.
2) No, but it does terminate the block in which the exception is thrown.
3) The exception will be processed by a catch handler (if one exists) associated with the try block (if one exists) enclosing the catch handler that caused the exception.
4) Provide an exception specification listing the exception type that the function can throw.
5) Function unexpected is called to terminate the program.

3.7 FURTHER READINGS

1) E. Balaguruswamy, *Object Oriented Programming with C++*, Tata McGraw Hill, 2010.
2) P. Deitel and H. Deitel, C++: *How to Program, PHI*, 7th edition, 2010.
3) B. Strousstrup, *Programming – Principles and Practices using C++*, Addison Wesley, 2009.