Discipline Courses-I
Semester-I
Paper: Programming Fundamentals
Unit-VII
Lesson: Exception Handling
Lesson Developer: Rakhi Saxena
College/Department: Deshbandhu College, University of Delhi

Table of Contents

1.1 Exception Handling -I

You may have written quite a few programs by now. Some of your programs when executed may have terminated unexpectedly with runtime errors. The errors could have occurred because the user entered an invalid input or an array index reference was out of range and so on.

Such an error situation that would be unusual for the program that is being processed and causes the program to terminate unexpectedly is called an exception. As a programmer, you should anticipate exceptions in your program that could lead to unpredictable results and handle them.

In this chapter you will learn to write fault tolerant programs that are able to deal with exceptions, and continue executing or terminate gracefully.

1.1.1 **Learning Objectives**

After reading this chapter you should be able to:

- Write fault tolerant programs that can deal with and recover from errors.
  - Define an exception.
  - Detect exceptions through a try block.
  - Throw exceptions using a throw specifier.
  - Catch exceptions with a catch block.

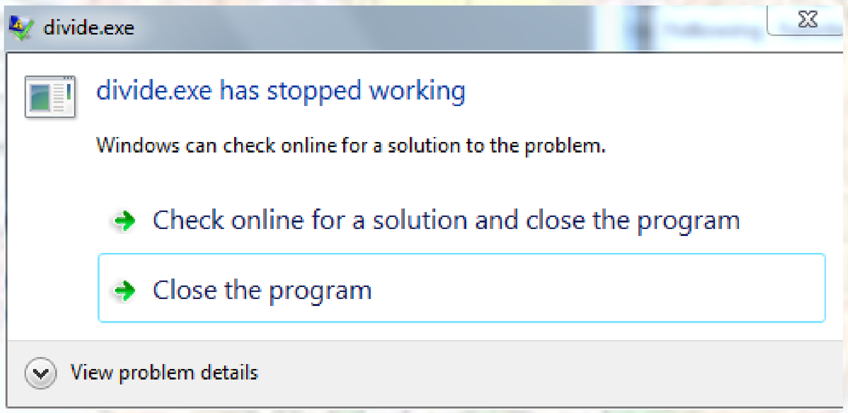
- Write exception handlers to deal with exceptions.
- Rethrow an exception without handling it.
- Describe how exceptions are handled within functions.
- Describe how exceptions that are thrown are propagated up the call stack .

1.1.2 Handling Errors

To write good code, it is essential to handle potential errors so that the program does not terminate unexpectedly. For example, consider the following function that takes two numbers as parameters, divides them and returns the quotient:

```
int divide(int dividend, int divisor)
{
     return dividend/divisor;
}
```

Since division by 0 in integer arithmetic causes a program to terminate prematurely, a call to the function divide such as divide(10,0) will cause the program to fail.


divide.exe

divide.exe has stopped working

Windows can check online for a solution to the problem.

Figure 1.1 – Program Failure

Note: If the function divide() had used floating point numbers instead of integers, there would not have been an error, since in floating-point arithmetic, division by zero is allowed — it results in positive or negative infinity, which would be displayed as INF or -INF.

The traditional way to handle errors is by using a return code. The function returns a zero if it is successful and a unique error code if it fails. For example, the function divide() could check whether the divisor is 0 or not before attempting to divide, and then return either -1 (on failure) or the quotient (on success) respectively.

```
int divide(int dividend, int divisor)
{
   if (divisor == 0)
     return -1;
   else
     return dividend/divisor;
}
```

However, using error codes has certain disadvantages – the calling function needs to check the return values from a function to determine whether it is a valid value or an error code before using the return value. Also, how does the calling function know whether a return value of -1 indicates an error or whether it is a valid return value? It is hard to tell without looking inside the function's code. The problem is that the program logic is mixed with the error handling logic.

C++ provides an exception handling mechanism to separate the error detecting code from the error handling code. An exception is an unexpected situation that occurs during the execution of a program. It is a way of flagging unexpected conditions or errors that have occurred in a program. The basic idea in exception handling is to

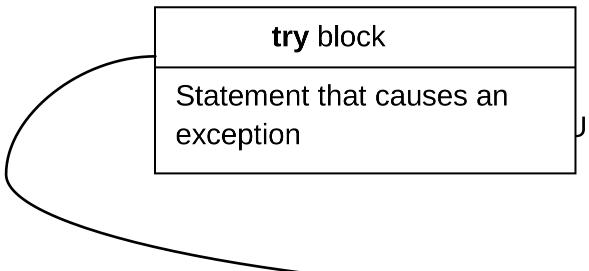
- Denote an exception block - Identify areas in the code where errors can occur and detect the problem
- Throw the exception – Signal that a problem has occurred
- Catch the exception - Receive the error information
- Handle the exception - Take corrective action to recover from the error

1.1.3 Exception Handling Keywords

An exception occurs in a program when an unusual situation – an error result or unpredictable behavior by the program - is encountered. For example, some exceptions that can occur are: integer division by zero, user entering invalid values as input, array index out of bounds, unexpected end of file encountered or a memory insufficient error.

A programmer should anticipate such exceptional situations that could result in unpredictable behavior. Instead of terminating abnormally, the program should handle the exception and then continue normal execution. C++ provides the following keywords to handle an exception:

1. **try -** A try block surrounds the part of the code that can generate exception(s).
2. **throw -** When an exception is detected, a throw statement signals that an exception has occurred. The exception is thrown using a typed object.
3. **catch –** The catch block follows a try block. The catch block contains the exception handler – that is, specific code that is executed when the exception occurs. There can be multiple catch blocks following a try block to handle different types of exceptions**.**



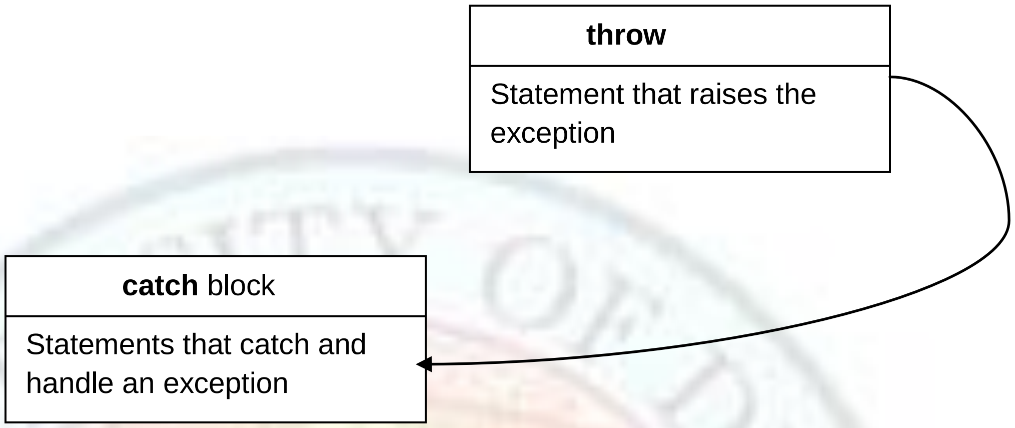

throw
Statement that raises the exception
catch block
Statements that catch and handle an exception

Figure 1.2 – try, throw and catch

The syntax of handling an exception is as below:

```
try {

        // Part of the program where an exception might occur
        throw exception object
}
catch (datatype1 argument1) {
        // Handle exception of the data type1
}
catch (datatype2 argument2) {
        // Handle exception of the data type2
}
```

If no exception is thrown during execution of the statements in the try block, the catch clauses that follow the try block are not executed. Execution continues at the statement after the last catch clause.
If an exception is thrown during execution of the statements in the try block, an exception object is created and thrown by the operand of the throw statement. The compiler now looks for a catch clause that can handle an exception of the type thrown. If a match is found the statements in that catch block are executed.

The following code fragment handles a division by zero exception:

```
try                                          //look for exceptions in this block
{
   if (divisor == 0)
      throw divisor;                         //signal that an exception has occurred
   int quotient = dividend/divisor;
   cout << "Result of division is: " << quotient;
}
catch (int exception_number)                 //catch the exception
{
   if (exception_number == 0)
```

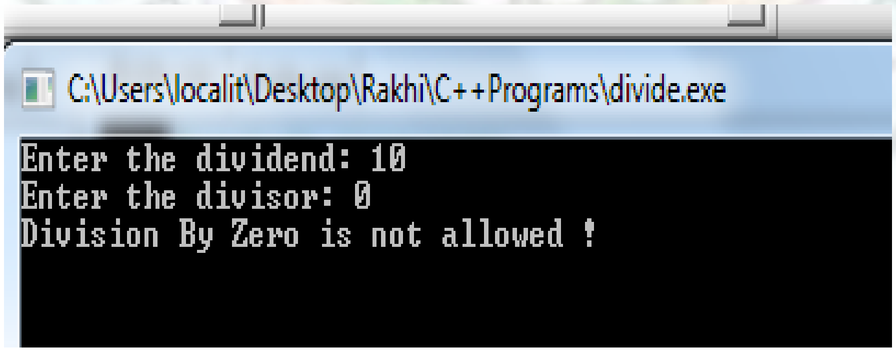
```
        cout << "Division By Zero is not allowed !\n";  //handle the exception
    }
```

We will learn more about each of these keywords in the following sections.

Value addition Source Code

Heading text DivideBy Zero Exception Handling

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

int main()
{
    int dividend, divisor;
    cout << "Enter the dividend: ";
    cin >> dividend;
    cout << "Enter the divisor: ";
    cin >> divisor;
    try
    {
        if (divisor == 0)
          throw divisor;
        int quotient = dividend/divisor;
        cout << "Result of division is: " << quotient;
    }
    catch (int exception_number)
    {
        if (exception_number == 0)
            cout << "Division By Zero is not allowed !" <<  endl;
    }
    getchar();
    return 0;
}
```



C:\Users\localit\Desktop\Rakhi\C++Programs\divide.exe

```
Enter the dividend: 10
Enter the divisor: 0
Division By Zero is not allowed !
```

Source: Self Made

1.1.4 Throwing Exceptions

An exception is thrown by using the **throw** keyword inside a **try** block. This is also called "raising" an exception. The throw keyword is followed by an object that signals the type of error that has occurred. For example, the following throw statement raises an exception and passes an integer value to the exception handler.

```
throw number;   //throw an integer variable that was defined previously
```

Any type of data can be thrown as an exception – a double, a string, a Boolean value, a user defined class and so on.

```
throw false;                    // throw a boolean value
throw -1;                       // throw a literal integer value
throw "Sorry: String too long"; //throw a literal character string
throw NewException(10);         //throw an object of the class NewException
```

The object that is thrown should signal some useful information to the exception handler – maybe the type of problem that has occurred or a description of the problem.

1.1.5 Detecting Exceptions

The try block looks for any exceptions that are thrown by statements within the block. This block of code is examined during execution to detect exceptions that may have been thrown by calls to functions within the block.

Try blocks and catch blocks work together — a try block detects exceptions that are thrown by statements within the try block, and routes them to the appropriate catch block for handling.

Value addition Know More!

Heading text Exceptions within Functions and Stack Unwinding

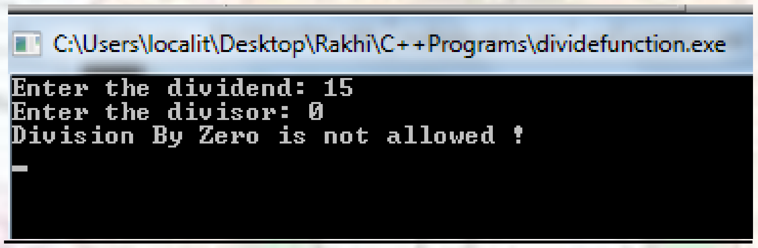
The throw statements do not have to be placed directly inside a try block due to the way exceptions propagate when thrown. The immediate caller of a function that throws an exception doesn't have to handle the exception if it doesn't want to. It can delegate that responsibility to its caller. This lets us employ exception handling in a modular manner.

For example, the following code demonstrates how the division by zero exception is handled when the division is performed within a function:

```
int divide(int num, int den)
{
   if (den == 0)
      throw den;
   return num/den;
}

int main()
```

```
{
    int dividend, divisor;
    cout << "Enter the dividend: ";
    cin >> dividend;
    cout << "Enter the divisor: ";
    cin >> divisor;
    try
    {
      cout << "Result of division is: ";
      cout << divide(dividend,divisor);
    }
    catch (int exception_number)
    {
       if (exception_number == 0)
           cout << "Division By Zero is not allowed !" <<  endl;
    }
    getchar();
    return 0;
}
```

Output:



```
C:\Users\localit\Desktop\Rakhi\C++Programs\dividefunction.exe
Enter the dividend: 15
Enter the divisor: 0
Division By Zero is not allowed !
_
```

The exception in this case is thrown inside the divide() function and the throw is not placed inside a try block. This means that the divide function just throws the exception but doesn't handle it. It delegates the responsibility of handling the exception to its calling function – in this case main(). Here, main() provides an exception handler for the exception that is thrown.

Thus, when an exception is raised, first, the program looks to see if the exception can be handled immediately (which means it was thrown inside a try block). If not, it immediately terminates the current function and checks to see if the caller will handle the exception. If not, it terminates the caller and checks the caller's caller. Each function is terminated in sequence until a handler for the exception is found, or until main() terminates. Thus the exception progressively moves up the call stack until it is either caught and handled, or until main() fails to handle the error. If nobody handles the error, the program typically terminates with an exception error.
This process of passing up an exception up the function call stack is called "**Stack Unwinding**".

Source: Self Made

1.1.6 Catching Exceptions

When an exception is thrown, the program execution stops at that point and control enters the catch block whose argument matches the type of the exception object thrown. The catch block is executed for handling the exception.

A try block must have at least one catch block attached to it, but can have multiple catch blocks listed in sequence. Thus multiple catch blocks can be chained together each with a different parameter type.
An ellipsis (...) as the parameter of catch will allow the handler to catch any exception no matter what the type of the throw exception is. Such a "catch all" handler can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last.

The following code fragment catches three types of exceptions and also provides a default exception handler:

```
try {

        // code where exception can occur
}
catch(int param) {
        cout << "An integer exception caught"<<param<<\n";
}
catch(double param) {
        cout << "An double exception caught"<<param<<\n";
}
catch(char param) {
        cout << "An character exception caught"<<param<<\n";
}
catch(…){
        cout << "Default exception caught\n";
}
```

Value addition – Frequently Asked Questions

Heading text Exception Handling

1. **What happens if the type of object that is thrown does not match the arguments in any catch block?**

If the type of the object that is thrown does not match any catch block, the program is aborted with the help of the **abort()** function.
If no handler at any level catches the exception, the special library function **terminate( )** is automatically called. By default, **terminate( )** calls the Standard C++ library function **abort( )** , which abruptly exits the program.

2. **What happens when no exception is detected and thrown?**
The control goes to the statement immediately after the catch block, i.e., the catch block is skipped. The program execution is resumed after the try catch block and not after the throw statement.

3. **Is it possible to nest try-catch blocks within an external try block?**
Yes, the internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

```
   try {
        try {
        // code here
     }
     catch (int n) {
        throw;
     }
   }
   catch (...) {
         cout << "Exception occurred";
 }
```

4. **Can the catch all handler(...) be used to handle any unanticipated problems?**
   The catch all handler can be used to wrap the contents of main(). In this case, if an unanticipated problem occurs in any function in the program and is not handled, it will eventually be passed on to main() which can print an error message, save the system state and terminate gracefully. For example:

```
   int main() {
       try {
           //function calls
       }
       catch(…) {
         cout << "Abnormal Termination\n";
       }
       //Save the state of execution
       return -1;
   }
```

Source: Self made

1.1.7 Rethrowing Exceptions

An exception handler can decide to re-throw an exception that it has caught without processing it. In such situations, simply invoke throw without any arguments as shown below:

```
throw;
```

This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.

For example, the throw statement in the catch block throws the exception back to the function from where this try/catch block was invoked.

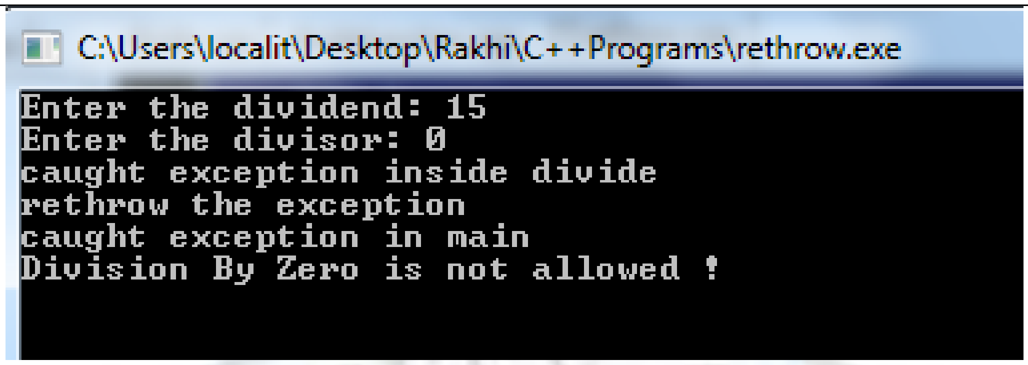
```
try {
      if (den == 0)
        throw den;
    } catch ( int exception_number)
    {
       cout << "Caught exception inside divide\n";
       cout << "Rethrowing the exception\n";
       throw;
    }
```

Value addition: Source Code

Heading text Rethrowing an Exception

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

int divide(int num, int den)
{
    try {
      if (den == 0)
        throw den;
    } catch ( int exception_number)
    {
       cout << "Caught exception inside divide"<<endl;
       cout << "Rethrowing the exception"<<endl;
       throw;
    }
    return num/den;
}

int main()
{
     int dividend, divisor;
     cout << "Enter the dividend: ";
     cin >> dividend;
     cout << "Enter the divisor: ";
     cin >> divisor;
     try
     {
        cout << "Result of division is: " << divide(dividend,divisor);
     }
     catch (int exception_number)
     {   cout << "Caught exception in main"<<endl;
         if (exception_number == 0)
             cout << "Division By Zero is not allowed !" <<  endl;
     }
     getchar();
     return 0;
}
```

Output



```
Enter the dividend: 15
Enter the divisor: 0
caught exception inside divide
rethrow the exception
caught exception in main
Division By Zero is not allowed !
```

Source: Self Made

1.2 Exception Handling -II

In the last section you learnt to handle exceptions using the try, throw and catch keywords. In this section you will learn how to restrict function throw types, some standard exceptions and how to create user defined exception objects.

1.2.1 Learning Objectives

After reading this chapter you should be able to:

- Handle exceptions in a program.
  - Restrict the types of exceptions that can be thrown by a function
  - Use the standard exception class and other classes derived from it.
  - Create and employ user defined exception classes.

1.2.2 Restricting Function throw Types

The **throw** clause as we have seen earlier signals the exception that has occurred. It can also be used in a function declaration to restrict the types of exceptions that can be thrown by a function. This is called "**Exception Specification".** For example, the following function declaration limits the type of exceptions that can be thrown by the testFunction() to int, char and double:

```
void testFunction(int test) throw(int, char, double)

    {
     if(test==0)
        throw test;              // throw int
     if(test==1)
        throw 'a';               // throw char
     if(test==2)
        throw 333.23;            // throw double
    }
```

Any type that does not appear in the throw() part of the declaration cannot be thrown by that function. This tells anyone that uses the function that there is a fixed list of exception types that the function may throw, and that these are the only types that need to be watched for in try blocks containing calls to the function.

If the throw clause is left empty without any type in the function declaration, the function is not allowed to throw any type of exceptions. For example:

```
void testFunction(int test) throw(); //no exceptions allowed
```

A function declaration without any throw clause in its declaration is allowed to throw all exception types.

```
void testFunction(int test);          //all exceptions allowed
```

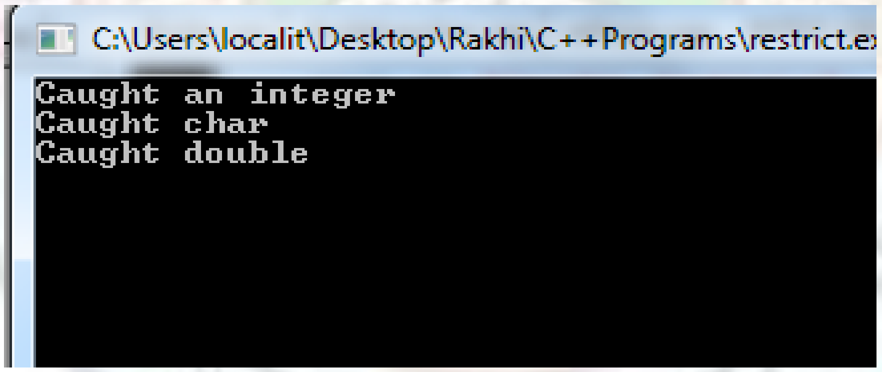
Value addition: Source Code

Heading text: Exception Specification - Restricting Function throw Types

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

void testFunction(int test) throw(int, char, double)
{
 if(test==0)
    throw test;               // throw int
 if(test==1)
    throw 'a';                // throw char
 if(test==2)
    throw 333.23;             // throw double
}

int main()
{

 try{
   testFunction(0);
 }
 catch(int i) {
   cout << "Caught an integer\n";
 }
 catch(char c) {
   cout << "Caught char\n";
 }
 catch(double d) {
   cout << "Caught double\n";
 }

 try{
   testFunction(1);
 }
```

```
        catch(int i) {
          cout << "Caught an integer\n";
        }
        catch(char c) {
          cout << "Caught char\n";
        }
        catch(double d) {
          cout << "Caught double\n";
        }

        try{
          testFunction(2);
        }
        catch(int i) {
          cout << "Caught an integer\n";
        }
        catch(char c) {
          cout << "Caught char\n";
        }
        catch(double d) {
          cout << "Caught double\n";
        }
        getchar();
        return 0;
}
```

Output



C:\Users\localit\Desktop\Rakhi\C++Programs\restrict.ex

```
Caught an integer
Caught char
Caught double
```

Source: Self Made

1.2.3 Standard Exceptions

The C++ Standard library defines a class called **exception** in the <exception> header file under the namespace std. This class contains a method what() that can be used to get the error message associated with the exception. The exception class serves as the base class for all exceptions thrown by classes and functions defined in the Standard Library.

For example, a bad_alloc exception class is a derived class of the exception class. A bad_alloc object is thrown by the new operator when it fails to allocate sufficient memory. The following code catches and handles such insufficient memory allocation errors:

```
try
{
      int * newarray = new int[1000];
}
catch (bad_alloc&  param)
{
     cout << "Error allocating memory." << param.what()<<endl;
}
```

The same exception can be handled by catching a reference to the exception class since bad_alloc is a derived class of exception.

```
try
{
      int * newarray = new int[1000];
}
catch (exception&  param)
{
     cout << "Error allocating memory." << param.what()<<endl;
}
```

Value addition: Did you Know?

Heading text Standard Exception Classes in C++

The **exception class** is the base class for the two exception classes:

- **logic_error** : Thrown because of a logical error in a program
- **runtime_error**: Thrown because of an error in a library function or in the run-time system.

The exception classes derived from the logic_error class are:

Exception	**Description**
domain_error	Indicates an invalid function argument
invalid_argument	Indicates an invalid argument to in C++ standard function
length_error	Object's length exceeds maximum allowable length
out_of_range	Reports an out of range reference to an array element

The exception classes derived from the runtime_error class are:

Exception	**Description**
range_error	Reports an out of range error in standard template library container
overflow_error	Reports an arithmetic overflow in standard template library container

underflow_error	Reports an arithmetic underflow in standard template library container

Other popular predefined exception classes are:

Exception	**Description**
bad_alloc	Reports a failure to allocate memory by new
bad_exception	Thrown when an exception type does not match any catch
bad_cast	Thrown by dynamic_cast when it fails with a reference type
bad_typeid	Thrown by typeid operator when its operand is a null pointer
ios_base::failure	Thrown by functions in the iostream library

Source: Self Made

Value addition: Did you Know?

Heading text Passing Exception Objects by Reference

The parameter caught by the catch clause can be passed either by value or by reference.

```
catch (exception param)   //passed by value
catch (exception &param) //passed by reference
```

When the parameter is passed by value, every time the exception is passed up the call stack, a copy of the parameter is made and sent to the next function. If the object is large this copying will waste time and memory. It is a better idea therefore to catch exceptions (other than basic types such as int, char or long) by reference. This will ensure that no copy of the object is made, thus saving time and memory.

Source: Self Made

1.2.4 User Defined Exception Classes

Just like the Standard library defines exception classes derived from the exception class, it is possible to define your own **exception classes** and to cause objects of those classes to be thrown whenever an exception occurs. An exception class is just a normal class that is designed specifically to be thrown as an exception.

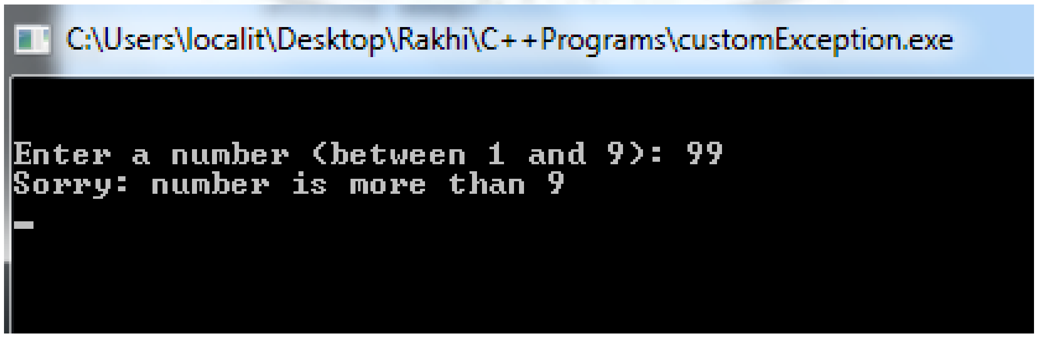
The following code fragment creates a custom exception class - RangeException. Objects of this class can be thrown when an exception occurs. The class has a string data member to hold a description of the error that has occurred. The default and the parameterized constructors assign a descriptive string to the data member.

```
class RangeException
{
    public:
            RangeException() {msg = "User defined Range Exception!";}
            RangeException(string m) {msg = m; }
            ~RangeException() throw() {}
            string message() { return msg; }
        private:
            string msg;
    };
```

Now suppose you are writing a program to accept user input within a certain range (say between 1 and 9). Then you could throw a RangeException object with an appropriate error message whenever the user input is out of range.

```
    int main()
    {
        int number;

        cout << "Enter a number (between 1 and 9): ";
        cin >> number;
        try
        {
          if (number < 1)
            throw RangeException("Sorry: number is less than 1");
        }
        catch(RangeException& e)
        {
            cout << e.message() << endl;
        }
        try
        {
          if (number > 9)
             throw RangeException("Sorry: number is more than 9");
        }
        catch(RangeException& e)
        {
            cout << e.message() << endl;
        }
        getchar();
        return 0;
    }
```

The following output will be displayed when the user enters a number outside the given range:



```
C:\Users\localit\Desktop\Rakhi\C++Programs\customException.exe

Enter a number (between 1 and 9): 99
Sorry: number is more than 9
_
```

Value addition: Did you Know?

Heading text Creating Custom Exception Classes by extending the standard exception class

Sometimes instead of writing an exception class from scratch, it is useful to extend the standard exception class. This way the derived custom class exception object can be caught by an exception class parameter.

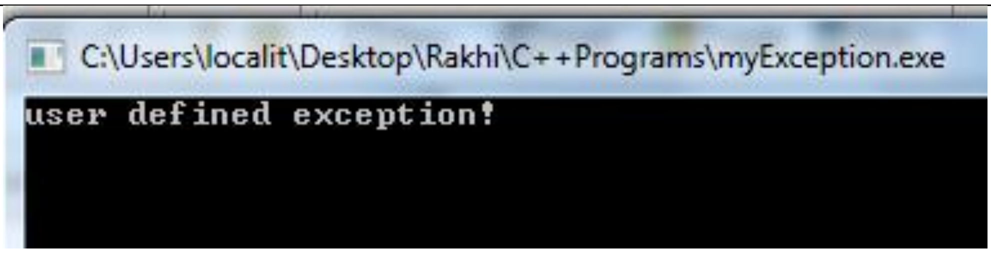
To create such a class, derive the base exception class and override the base exception::what() function that has the following signature:

```
const char * what() const throw()
```

The following program defines a myException class extended from the base exception class:

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

class myException:public exception
{
     public:
         myException(string m ="user defined exception!") : msg(m) {}
         ~myException() throw() {}
         const char* what() const throw() { return msg.c_str(); }

     private:
         string msg;
};

int main()
{
   try
   {
     throw myException();
   }
   catch(exception& e)
   {
       cout << e.what() << endl;
   }
   getchar();
   return 0;
}
```

<u>Output:</u>



```
user defined exception!
```

Source: Self Made

Value addition: Did you Know?

Heading text Exceptions in Constructors and Destructors

Exceptions can be very useful when writing constructors. If a constructor fails, simply throw an exception to indicate the object failed to create. The object's construction is aborted and its destructor is never executed.

However, exceptions should *not* be thrown in destructors. The problem occurs when an exception is thrown from a destructor during the exception being passed up the function call stack. If that happens, the compiler is put in a situation where it doesn't know whether to continue passing up the exception or to handle the new exception thrown by the destructor. The end result is that the program is terminated immediately.

Source: Self Made

Summary

- Traditional error handling employs error codes as return values to indicate success or failure.
- However, using error codes has certain disadvantages – the calling function needs to check the return values from a function to determine whether it is a valid value or an error code before using the return value.
- C++ provides an exception handling mechanism to separate the error detecting code from the error handling code.
- An exception is an unexpected situation that occurs during the execution of a program. It is a way of flagging unexpected conditions or errors that have occurred in a program.
- C++ provides the – try, throw and catch – keywords for exception handling.
- A try block surrounds the part of the code that can generate exception(s).
- When an exception is detected, a throw statement signals that an exception has occurred. The exception is thrown using a typed object.
- Any type of data can be thrown as an exception – a double, a string, a Boolean value, a user defined class and so on.

- The catch block contains the exception handler – that is, specific code that is executed when the exception occurs.
- There can be multiple catch blocks following a try block to handle different types of exceptions**.**
- If no exception is thrown during execution of the statements in the try block, the catch clauses that follow the try block are not executed. Execution continues at the statement after the last catch clause.
- When an exception is thrown, the program execution stops at that point and control enters the catch block whose argument matches the type of the exception object thrown. The catch block is executed for handling the exception.
- An exception progressively moves up the call stack until it is either caught and handled, or until main() fails to handle the error. If nobody handles the error, the program typically terminates with an exception error.
- An ellipsis (...) as the parameter of catch will allow the handler to catch any exception no matter what the type of the throw exception is.
- It is possible to nest try-catch blocks within an external try block. The internal catch block forwards the exception to its external level.
- An exception handler can decide to re-throw an exception that it has caught without processing it by simply invoking throw without any arguments. This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.
- A throw specifier in a function declaration is used to restrict the types of exceptions that can be thrown by a function.
- An empty throw clause (without any type in the function declaration) allows the function to throw any type of exceptions.
- A function declaration without any throw clause in its declaration is allowed to throw all exception types.
- The C++ Standard library defines a class called **exception** in the <exception> header file under the namespace std. The exception class serves as the base class for all exceptions thrown by classes and functions defined in the Standard Library.
- The exception class contains a method what() that can be used to get the error message associated with the exception.
- The exception class is the base class for the two exception classes:
    - logic_error : Thrown because of a logical error in a program
    - runtime_error: Thrown because of an error in a library function or in the run-time system.
- It is better to catch exceptions (other than basic types such as int, char or long) by reference. This will ensure that no copy of the object is made, thus saving time and memory.
- A user defined custom exception class is just a normal class that is designed specifically to be thrown as an exception.
- A custom exception class can also be derived from the standard exception class.
- If a constructor fails, simply throw an exception to indicate the object failed to create. The object's construction is aborted and its destructor is never executed.
- Exceptions should *not* be thrown in destructors since a call to the destructor during exception passing up the function call stack can cause the program to terminate immediately.

Exercises

1.1 Why should exception handling be used instead of different return values to signal an error?
1.2 Explain exception handling in C++ with an example.
1.3 Explain what happens during stack unwinding in exception handling.
1.4 Give two reasons that the exception declaration of a catch clause should declare a reference.
1.5 Create a class with a main( ) that throws an object of class myException inside a try block. Give the constructor for myException a string argument. Catch the exception inside a catch clause and print the string argument. Add a catch all clause and print a message there.
1.6 Write a function with an exception specification that can throw four exception types: a char, an int, a bool, and your own exception class. Catch each in main( ) and verify the catch. Derive your exception class from the standard exception class. Write the function in such a way that the system recovers and tries to execute it again.
1.7 Write a College class that has a Course class that is having troubles with its Student class. Use a try block in the College class constructor to catch an exception (thrown from the Student class) when its Course object is initialized. Throw a different exception from the body of the College constructor's handler and catch it in main( ).

Glossary

catch: block that contains the exception handler - specific code that is executed when the exception occurs.

exception: an unexpected situation that occurs during the execution of a program. It is a way of flagging unexpected conditions or errors that have occurred in a program.
throw: statement that signals that an exception has occurred by throwing a typed object.

try: block that surrounds the part of the code that can generate exception(s).

References

1. Works Cited

2. Suggested Reading

1. B. A. Forouzan and R. F. Gilberg, Computer Science, A structured Approach using C++, Cengage Learning, 2004.
2. R.G. Dromey, How to solve it by Computer, Pearson Education
3. E. Balaguruswamy, Object Oriented Programming with C++ , 4th ed., Tata McGraw Hill
4. G.J. Bronson, A First Book of C++ From Here to There, 3rd ed., Cengage Learning.
5. Graham Seed, An Introduction to Object-Oriented Programming in C++, Springer
6. J. R. Hubbard, Programming with C++ (2nd ed.), Schaum's Outlines, Tata McGraw Hill
7. D S Malik, C++ Programming Language, First Indian Reprint 2009, Cengage Learning
8. R. Albert and T. Breedlove, C++: An Active Learning Approach, Jones and Bartlett India Ltd.

3. Web Links

1.1 http://www.freshsources.com/Except1/ALLISON.HTM
1.2 http://msdn.microsoft.com/en-us/library/4t3saedz.aspx
1.3 http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=/com.ibm.xlcpp8a.doc/language/ref/exceptn.htm
1.4 http://www.informit.com/articles/article.aspx?p=31537
1.5 http://www.boost.org/community/error_handling.html
1.6 http://www.go4expert.com/forums/showthread.php?t=2890