16. EXCEPTION HANDLING

In C++ errors can be divided into two categories
1. Compile time errors and
2. Run time errors.

Compile time errors are syntactic errors which occurs during the writing of the program. Most common examples of compile time errors are missing semicolon, missing comma, missing double quotes, etc. They occurs mainly due to poor understanding of language or writing program without proper concentration to the program.

There are logical errors which are mainly due to improper understanding of the program logic by the programmer. Logical errors cause the unexpected or unwanted output..

Exceptions are runtime errors which a programmer usually does not except. They occurs accidentally which may result in abnormal termination of the program. C++ provides exception handling mechanism which can be used to trap this exception and running programs smoothly after catching the exception.

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: 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.

Some common causes of errors are
- division by 0, or values that are too large or small for a type
- no memory available for dynamic allocation
- errors on file access, for example, file not found

- attempt to access an invalid address in main memory
- invalid user input

Dividing two integers

```
#include <iostream>
using namespace std;
int main ()
{
int num1, num2, result;
for (int i = 0; i < 5; i++)
{
cout << "Enter an integer: ";
cin >> num1;
cout << "Enter another integer: ";
cin >> num2;
result = num1 / num2; // The statement that may create exception.
cout << "The result of division is: " << result << endl;
}
return 0;
}
```

Output:

```
Enter an integer: 6
Enter another integer: 0
```

A program that aborts with an error message

```
#include <iostream>
#include <cassert>
using namespace std;
int main ()
{
int num1, num2, result;
for (int i = 0; i < 5; i++)
{
cout << "Enter an integer: ";
cin >> num1;
cout << "Enter another integer: ";
cin >> num2;
//Handling divided by zero exception
if (num2 == 0)
{
cout << "No division by zero!. Program is aborted." << endl;
assert (false);
}
result = num1 / num2;
cout << "The result of division is: " << result << endl;
}
return 0;
}
```

Output:

```
Enter an integer: 7
Enter another integer: 0
No division by zero. Program is aborted.
```

```
#include <iostream>
using namespace std;
int main ( )
{
int num1, num2, result;
for (int i = 0; i < 4; i++)
{
cout << "Enter an integer: ";
cin >> num1;
cout << "Enter another integer: ";
cin >> num2;
if (num2 == 0)
{
cout << "Division cannot be done in this case." << endl;
}
else
{
result = num1 / num2;
cout << "The result of division is: " << result << endl;
}
}
return 0;
}
```

```
/*A program that uses the return value of a function to show *
the occurrence of a run-time error */

#include <iostream>
using namespace std;

// Function declaration
int quotient (int first, int second);

int main ()
{
int num1, num2, result;
for (int i = 0; i < 3; i++)
{
cout << "Enter an integer: ";
cin >> num1;
cout << "Enter another integer: ";
cin >> num2;
result = quotient (num1, num2);
if (result == −1)
{
cout << "Error, division by zero." << endl;
}
else
{
cout << "Result of division is: " << result << endl;
}
}
return 0;
}

// Function definition
int quotient (int first, int second)
{
if (second == 0)
{
return −1;
}
return (first / second);
}
```

1. The first approach is the worst. We let the program abort without any warning.
2. The second approach is better; the program will still abort, but the user will be notified.
3. The third approach is better than the first two because the pair that would cause the program to abort is ignored, and the program continues with the rest of the data. The problem with this approach is that the code for handling an error is mingled with the productive code of the program. In other words, the problem here is coupling. The error-handling code is so coupled with the code to do the job that it is difficult to distinguish between them.
4. The fourth approach is the best, but it cannot be applied in all cases. In addition, the principle of modular programming dictates that the return value of a function be used only for one purpose, not two. In this case, one value (−1) is used to report an error; other values are used to return the result of the calculation.

Exception Handling Approach

Exception handling is the process to handle the exception if generated by the program at run time. The aim of exception handling is to write code which passes exception generated to a routine which can handle the exception and can take suitable action. Any exception handling mechanism must have the following steps:

Step 1 : Writing exception class (optional).
Step 2 : Writing try block.
Step 3 : Throwing an exception.
Step 4 : Catching and handling the exception thrown.

1. The try Block

The exception is to be thrown to be written in the try block. Whenever an exception is generated in the try block, it is thrown. An exception is an object so we can say that an exception object is thrown. The throw keyword is used for throwing an exception. The usual practice of using the throw statement is as :

```
throw exception;
```

For throwing an exception and simply

```
throw;
```

For re-throwing an exception.

Syntax:

```
try
{
        statements;
        statements;
        statements;
        throw exception;
}
```

2. The catch Block

An exception thrown by try block is caught by the catch block. A try block must have at least one catch, though there can be many catch block for catching different types of exceptions. A catch block must have a try block prior written which will throw an exception. The catch block is used as :

Syntax:

```
try
{
        statements;
        statements;
        statements;
        throw exception;
}
catch (object or argument)
{
        statements for handling the exception;
}
```

The catch block catches any exception thrown by the try block. If exception thrown matches with the argument or object in the block, the statements written within the catch block and we say that exception thrown by try block was caught successfully by the catch block. After the successful execution of the catch block statements any statements following the catch block will be executed. If argument does not match with the exception thrown, catch couldn't handle it and this may results in abnormal program termination.

The try, throw and catch all together provide exception handling mechanism in C++. The exception is generated by the throw keyword which is written in the try block. Any exception generated within this try block is thrown using this throw keyword. Immediately following the try block, catch block is written. As soon as some run time errors occurs an exception is thrown by the try block using throw which informs the catch block that an error has occurred in the try block. This try block is also known as exception generated block. The catch block is responsible for catching the execution thrown by the try block. When try throw an exception, the control of the program passes to the catch block and if argument matches as explained earlier, exception is caught. In case no exception is thrown the catch block is ignored and control passes to the next statement after the catch block.

Program:

```
#include <iostream.h>
void main( )
{
        try
        {
                throw “Exception Demo”;
        }
        catch(char*E)
        {
                cout<<“Exception caught=”<<E<<endl;
        }
        cout<<“Continue after catch block”<<endl;
}
```

OUTPUT :

```
Exception caught= Exception Demo
Continue after catch block
```

Catching Division by Zero Exception

```
#include <iostream.h>
void main( )
{
        float x,y;
        cout<<“Enter two numbers\n”;
        cin>>x>>y;
        try
        {
                if(y!=0)
                        cout<<“Div=”<<x/y<<endl;
                else
                        throw(y);
```

```
          }
          catch(float E)
          {
                    cout<<“Caught an Exception \n”;
                    cout<<“y=”<<y<<endl;
          }
          cout<<“Out of try catch block \n”;
}
```

Throwing Exception int 1-b where a>b

```
#include <iostream>
int main( )
{
int x,y;
cout<<“Enter two numbers \n”;
cin>>x>>y;

try
{
          func(x,y);
}
catch(char*E)
{
cout<<“Caught an Exception\n”;
cout<<E<<endl;
}
cout<<“Out of try catch block\n”;
}

void func(int x, int y)
{
          bool m;
          m=x>y ?true :false;
          if(m==true)
          cout<<“Subtraction=”<<x-y<<endl;
          else
          throw(“subtraction not possible”);
}
```

Multiple Catch Statement With Single Try

```
#include <iostream>
int main( )
{
int num=10;
for(num=10;num<=30;num+=10)
{
try
{
if(num==20)
throw(“GOOD”);
else if(num<20)
throw num;
else if (num>20)
throw 2.25f;
}
catch(int E)
{
cout<<“Caught int Exception E=”<<E<<endl;
}
catch(char* E)
{
cout<<“Caught string Exception E=”<<E<<endl;
}
catch(float E)
{
cout<<“Caught float Exception E=”<<E<<endl;
}
}
cout<<“Outside try catch block”<<endl;
}
```

Write a program to generate exception in fuction.

Exception Handling With Class

For throwing an exception of say demo class type within the try block we may write

Syntax:

```
throw demo( );
```

Program:

// Exception Handling With Single Class

```
#include <iostream.h>
class demo
{
};
int main( )
{
        try
        {
                throw demo( );
        }
        catch(demo d)
        {
                cout<<"Caught exception of demo class \n”;
        }
}
```

OUTPUT :

```
Caught exception of demo class
```

Exception Handling With Two Classes

```
#include <iostream.h>
class demo1
{
};
class demo2
{
};

int main( )
{
        for(int i=1;i<=2;i++)
        {
                try
                {
                        if(i==1)
                        throw demo1( );
                        else if(i==2)
                        throw demo2( );
                }
        catch(demo1 d1)
        {
                cout<<“Caught exception of demo1 class \n”;
        }
        catch(demo2 d2)
        {
                cout<<“Caught exception of demo2 class \n”;
        }
}
}
```

Write a program for Exception Handling in constructor

Exception Handling With Inheritance

```
#include <iostream>
class demo1
{
};
class demo2 :public demo1
{
};
int main( )
{
for(int i=1;i<=2;i++)
{
try
{
if(i==1)
throw demo1( );
else if(i==2)
throw demo2( );
}
catch(demo1 d1)
{
cout<<“Caught exception of demo1 class \n”;
}
catch(demo2 d2)
```

Exception Handling In Constructor

```
#include <iostream>
class demo
{
int num;
public :
demo( )
{
try
{
throw 25;
}
catch(int E)
{
cout<<“Exception caught \n”;
num=E;
}
}

void show( )
{
cout<<“num=”<<num<<endl;
}
};
int main( )
{
demo d;
d.show( );
}
```

Catching All Exceptions

A single catch block can be used for catching all different types of exceptions. This is necessary in situations when we do not know in advance what particular types of exception may be thrown by the try block.

Syntax:

```
catch(…)
{
 Statements for handling exceptions;
}
```

Program:

```
#include <iostream>
int main( )
{
        int num = 10;
        for(num=10;num<=30;num+=10)
        {
                try
                {
                        if(num==20)
                                throw(“Good”);
                        else if(num<20)
                                throw num;
                        else if(num>20)
                                throw 2.25f;
                }
                catch(...)
                {
                        cout<<“Caught an Exception \n”;
                }
                cout<<“Outside try catch block\n”;
        }
}
```

Specifying Exception For A Function

We can specify the types of exception for a function to be thrown as a list passed to throw. In this way we can restrict a function to throw only exception specified in the list and not any other types of exception.

Syntax:

```
return type function-name(function parameters)
throw (data-types)
{
function definition;
}
```

In case a function does not want to throw an exception it may leave throw empty like throw ( ).

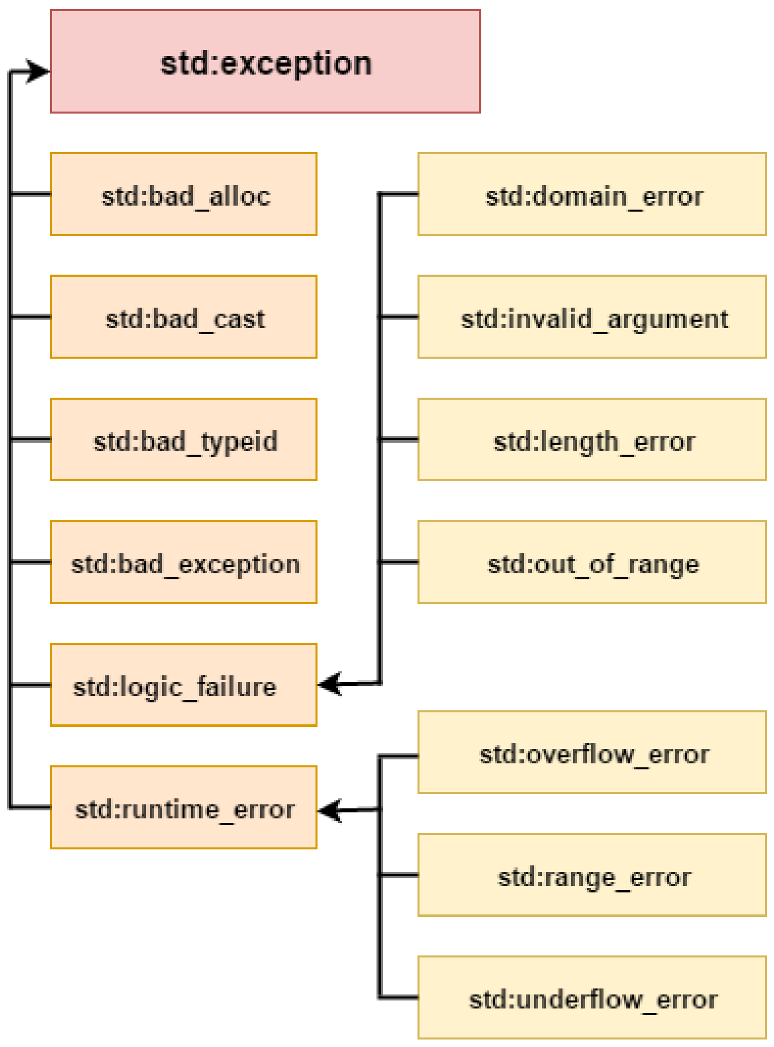
Program:

```
#include <iostream>
void demo( ) throw(int,char,char*)
{
        for(int i=0;i<=2;i++)
        {
                try
                {
                        switch(i)
                        {
                                case 0 :throw 123;
                                        break;
                                case 1 :throw ‘p’;
                                        break;
                                case 2 :throw “NMIMS UNIVERSITY”;
                                        break;
                        }
                }
        catch(int x)
        {
                cout<<“Caught an int exception :=”<<x<<endl;
        }
        catch(char x)
        {
                cout<<“Caught a char exception := “<<x<<endl;
        }
        catch(char* x)
        {
                cout<<“Caught a char* exception :=”<<x<<endl;
        }
}

int main( )
{
        demo( );
}
```

Standard Exceptions in C++

There are some standard exceptions in C++ under <exception> which we can use in our programs. They are arranged in a parent-child class hierarchy which is depicted below:



Exception	Description
exception	It is an exception and parent class of all standard C++ exceptions.
logic_failure	It is an exception that can be detected by reading a code.
runtime_error	It is an exception that cannot be detected by reading a code.
bad_exception	It is used to handle the unexpected exceptions in a c++ program.
bad_cast	This exception can be thrown by **dynamic_cast.**
bad_typeid	This can be thrown by **typeid.**
bad_alloc	This can be thrown by **new.**

Define New Exceptions

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

struct MyException : public exception {
   const char * what () const throw () {
      return "C++ Exception";
   }
};

int main() {
   try {
      throw MyException();
   } catch(MyException& e) {
      std::cout << "MyException caught" << std::endl;
      std::cout << e.what() << std::endl;
   } catch(std::exception& e) {
      //Other errors
   }
}
```