Object Oriented Programming Using Cpp

Topic:

Exception Handling

Chapter 10: Exception Handling

10.1 Introduction

The main objective of programming is to create error free programs. Errors remain in the program due to mainly due to poor understanding of programming. Errors may be categorized into following two types:

- Syntax error
- Logical error

Errors which happen during compile time are known as syntax error or compile time errors. On the other hand, errors which happen during execution of program are known as run time error. Syntax error is not a problem as it can be easily rectified. The main problem arises in case of run time error as it may cause minor or major damage to the system.

Formally, an **exception** is defined as a run time error which may cause abnormal termination of the program. C++ supports a mechanism known as exception handling to control exceptions in a program.

10.2 Exception Handling in C++

In C++, exception handling mechanism consists of following three components:

- try
- throw
- catch

The syntax is below:

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

```
catch(exception type)
{
        ……;
        ……;
        ……;
}
```

In the above syntax, try block contains those statements which may or may not cause exceptions. In case of no exception, the program proceeds in normal execution skipping the catch block and terminate after completion. If exception occurs then it is thrown using throw statement. The thrown exception is received by catch block. In catch block, user defines the steps what they want to do in case of exception.

The important thing is to note here is that catch block is always written immediately after try block. Again, the exception type inside throw statement and inside catch header must be same. Failing to do so will lead missing of exception and program will be terminated using abort() which is called implicitly by the compiler.

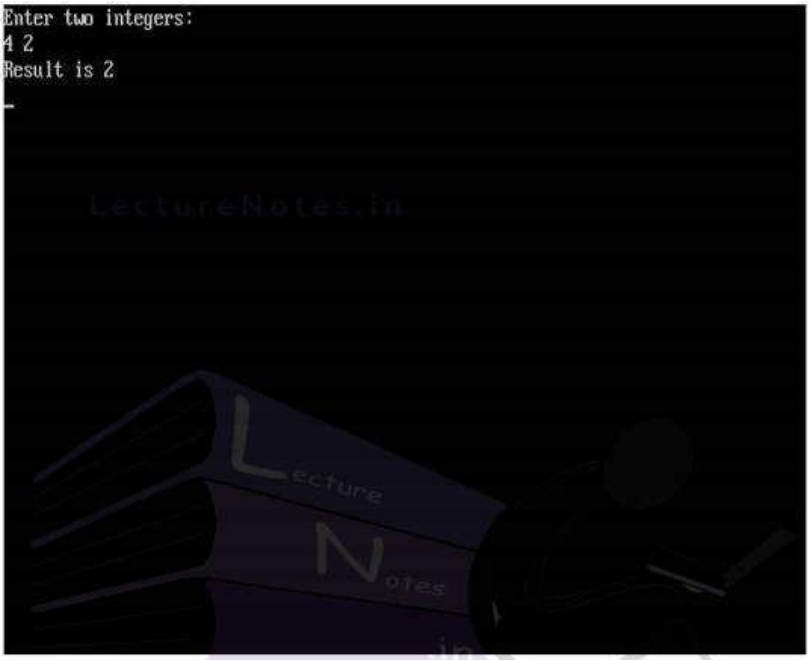
Program 67: Write a program in C++ to illustrate the concept of an exception in absence of exception handling mechanism.

Code:

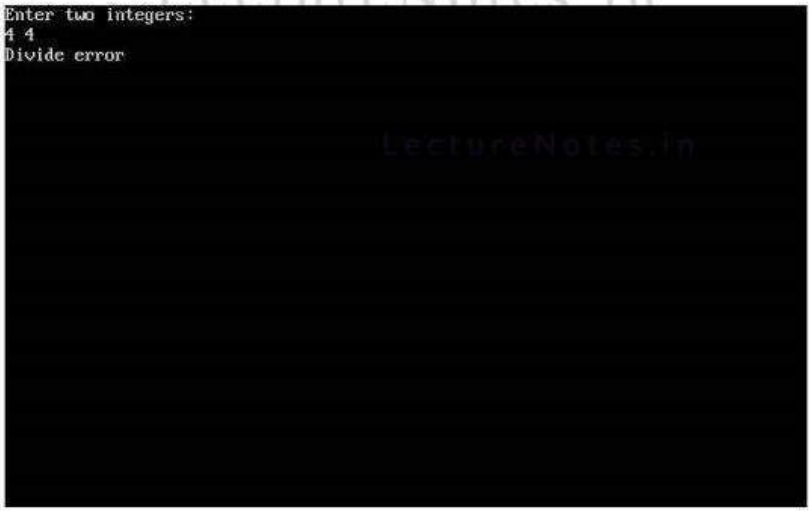
```
#include<iostream.h>
#include<conio.h>
void main()
{
        int n1,n2;
        clrscr();
        cout<<"Enter two integers:";
        cin>>n1>>n2;
        int x=n1-n2;
        int res=0;
        res=n1/x;
```

```
        cout<<"Result is "<<res<<endl;
        getch();
    }
```

O/P 1:



O/P 2:



Explanation: In the above program, the execution depends on the value of x. So, when user enters two different values x is non-zero and it calculates the result stored inside res. (O/P 1)

In O/P 2, when user enters two same values then x becomes zero which leads to the process of division by zero which causes an exception. Due to this system shows "Divide Error" message.

Now, program 68 explains the use of exception handling in the above program.

Program 68: Write a program in C++ to illustrate the concept of exception handling mechanism.

Code:

```
#include<iostream.h>
#include<conio.h>
void main()
{
        int n1,n2;
        clrscr();
        cout<<"Enter two integers:";
        cin>>n1>>n2;
        int x=n1-n2;
        int res=0;
        try
        {
                if(x != 0)
                {
                        res=n1/x;
                        cout<<"Result is "<<res<<endl;
                }
                else
                        throw(x);
```

```
        }
        catch(int      y)
        {
                cout<<"Exception caught as x becomes 0"<<endl;
                cout<<"Kindly enter two different values as input"<<endl;
        }
        getch();
}
```

O/P 1:

Enter two integers:

4 2

Result is 2

O/P 2:

Enter two integers:

4 4

Exception caught as x becomes 0

Kindly enter two different values as input

10.3 Multiple catch statements

Sometimes it may be possible that the type of exception may be not predictable. In this scenario, one catch block is not sufficient for exception handling. One simple solution is to write multiple catch block for different possible data types. However, this is not the best solution.

Syntax:

```
try
{
        //try section
}
```

```
catch(datatype 1)
{

}
catch(datatype 2)
{

}
……..……..
……..……..
catch(datatype n)
{

}
```

In this scenario, as soon as an exception is thrown, system searches for an appropriate catch block and it executes the corresponding catch block. After executing the matching catch block, control jumps to the first statement after last catch block.

Note: Kindly refer "OOP with ANSI and Turbo C++" by Ashok N. Kamthane book page no 584, program 15.2 for understanding of multiple catch statements.

10.4 Generic catch block

As we discussed earlier, writing multiple catch statements for different possible data types is not the best solution. C++ provides a new mechanism known as generic catch. A generic catch block is capable to handle any type of exception thrown by try block.

Syntax:

```
catch(...)
{

}
```

Program 68: A program to explain generic catch block

Code:

```
#include<iostream.h>
#include<conio.h>
voidnum(int  k)
{
        try
        {
                if(k==0)
                        throw(k);
                else if(k>0)
                        throw('P');
                else
                        throw(0.0);
        }
        catch(...)
        {
                cout<<"Exception is caught"<<endl;
        }
}
```

```
void main()
{
        num(0);
        num(-5);
        num(10);
        getch();
}
```

O/P:

Exception is caught

Exception is caught

Exception is caught

10.5 Rethrowing Exception

When an exception is thrown by a catch block to another catch block, the process is known as rethrowing of exception. It can be done using following syntax:

Syntax:

```
catch(exception type)
{

        throw;

}
```

10.6 Handling of Uncaught Exception

It may be possible that an exception may go uncaught. This may be because of either absence of a matching catch block or absence of generic catch block. These exceptions are known as uncaught exceptions.

C++ provides following two functions to control uncaught exceptions:

i. terminate() function: When an appropriate exception handler is not defined and an exception occurs then terminate() function gets invoked which implicitly invokes abort() function.

Ex:

```
class  ONE
{


};
class  TWO
{


};
void main()
{
        try
        {
                cout<<"An uncaught exception"<<endl;
                throw TWO();
        }
        catch(ONE)
        {
                cout<<"Exception for class ONE";
        }
}
```

Explanation: In the above example, an exception is thrown for class TWO using the statement throw TWO(). The catch block is defined to handle exception for class ONE. Hence, when exception is thrown, it does not find a matching catch block. Again, generic catch block is also not defined. In this case, the program is terminated automatically as abort() function is implicitly called by the compiler. Here, user does not have any control on the termination of the program.

ii. set_terminate() function: The set_terminate() function is used to transfer the control to the appropriate error handling function. This function requires only one argument which is function name where the control is to be transferred in case of any exception. If no function name is specified in the set_terminate() function, the program is terminated by an implicit call to abort() function.

Ex:

```
class  ONE
{

};
class TWO
{

};
void   my_terminate()
{
        cout<<"my_terminate function is invoked"<<endl;
}
void main()
{
        set_terminate(my_terminate);

        try()
        {
```

```
            throw(TWO);
        }
        catch(ONE)
        {
            cout<<"Class ONE exception"<<endl;
        }
    }
```


O/P:

my_terminate function is invoked

Explanation: Here, a set_terminate function is associated with a user defined function my_terminate(). When exception is thrown then the necessary matching catch block is not found and system executes the set_terminate function which invokes the my_terminate function.