Practical Lecture : Exception Handling

Quick Recap

Let's take a quick recap of previous lecture –

Today we are going to cover –

- Compile and run time polymorphism

- Virtual functions, Pure virtual functions

- virtual destructor

- Abstract classes and concrete class

- Self-Referential class

- Early binding and late binding, Dynamic constructors.

Today's Agenda

Today we are going to cover –

- Basics of exception handling

- Exception handling mechanism

- Throwing mechanism

- Catching mechanism

Let's Get Started-

Basics of Exception Handling

- It's very rare that a large program or software works correctly the first time. It might have errors.

- The two most common types of errors are:
  - -Compile time errors
  - -Runtime errors

- Compile time errors: Errors caught during compiled time is called Compile time errors. E.g
  Logical errors
  Syntactic errors (syntax errors)

- Run Time Errors - Programmers often come across some peculiar problems in addition logical or syntax errors. These are called exceptions.

Basics of Exception Handling

- Programmers can debug compile time errors by debugging and testing procedures.

- But runtime errors hinder normal execution of program. They are run-time anomalies or unusual logical conditions that may come up while executing the C ++ program.

- For example, User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed

- Consider the code given next, which may fail/crash at runtime on some systems.

Exception

```cpp
#include<iostream>
using namespace std;
int main()
{
    double var1, var2;
    cout<<"Enter two values"<<endl;
    cin>>var1 >>var2;
    cout<< var1 <<"/" <<var2 <<"=" <<var1/var2;
    return 0;
}
```
Output: Enter two values

4

0

4/0=inf

Note: Some compilers may terminate the program abruptly for divide by zero error.

Exception handling

As we have learnt , an exception is a problem that arises during the execution of a program.

A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exception handling is the process of handling errors and exceptions in such a way that they do not hinder normal execution of the system

In other words, Exceptions allow a method to react to exceptional circumstances and errors (like runtime errors) within programs by transferring control to special functions called handlers.

Exception handling

There are two types of exceptions:
   a)Synchronous,
   b)Asynchronous

Asynchronous exceptions are beyond the program's control, Disc failure etc. Those errors that are caused by events beyond the control of the program are called asynchronous exceptions.

Errors such as: out of range index and overflow fall under the category of synchronous type exceptions. For synchronized exceptions, C++ provides following specialized keywords for this purpose.

- **try**
- **throw**
- **catch**

All are case sensitive

Exception handling

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

try – A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

throw – A program throws an exception when a problem shows up. This is done using a throw keyword.

catch – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

Exception handling

The main motive of the exceptional handling concept is to provide a means to
    1. detect errors
    2. throw or report an exception and take appropriate action.

This mechanism needs a separate error handling code that performs the following tasks:

- Find and hit the problem (exception)

- Inform that the error has occurred (throw exception)

- Receive the error information (Catch the exception)

- Take corrective actions (handle exception)

syntax

The Catch blocks catching exceptions must immediately follow the try block that throws an exception.

```
try
{
    throw exception;
}

catch(type arg)
{
    //some code
}
```

How it works

If the try block throws an exception then program control leaves the block and enters into the catch statement of the catch block.

If the type of object thrown matches the argument type in the catch statement, the catch block is executed for handling the exception.

Divided-by-zero is a common form of exception generally occurred in arithmetic based programs.

Practice question

```cpp
#include<iostream>
using namespace std;
int main(){
    int a;
    double b;
    cout<< "Enter two integers "<<endl;
    cin>>a>>b;
    double d = 0;
    try    {
        if (b == 0)        {
            throw "Division by Zero not possible.";
        }
        else  {
            d=a/b;        cout<<d;
        }
    }
```

```
 catch (const char* error)  //This is used to catch the message thrown by try block
{
      cout << error << endl;
 }

    return 0;
}
```

In the code above, we are checking the divisor, if it is zero, we are throwing an exception message, then the catch block catches that exception and prints the message.

Doing so, the user will never know that our program failed at runtime, he/she will only see the message "Division by Zero not possible".

Note: Because we are raising an exception of type const char*, so while catching this exception, we have to use const char* in catch block

Practice question (revised)

```cpp
//The above program is written here in function call format.
#include<iostream>
using namespace std;
double division(int var1, double var2)
{
    if (var2 == 0) {
        throw "Division by Zero not possible.";
    }
    return (var1 / var2);
}
int main()
{

        int a;
        double b,d=0;
        cout<< "Enter two integers "<<endl;
        cin>>a>>b;
```

Practice question(revised)

```cpp
 try {
    d = division(a, b);
    cout << d << endl;
 }
 catch (const char* error) {
    cout << error << endl;
 }

    return 0;
}
```

Output:
Enter two integers
3 0
Division by Zero not possible

Try block

A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

The code which can throw any exception is kept inside(or enclosed in) a try block.

Then, when the code will lead to any error, that error/exception will get caught inside the catch block

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords.

A try/catch block is placed around the code that might generate an exception.

Code within a try/catch block is referred to as protected code.

Try block

```
try {
  // protected code
} catch( ExceptionName e1 ) {
  // catch block
} catch( ExceptionName e2 ) {
  // catch block
} catch( ExceptionName eN ) {
  // catch block
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Throwing exceptions

Exceptions can be thrown anywhere within a code block using throw statement.

The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Consider the following example of throwing an exception:

```
double division(int a, int b) {
  if( b == 0 ) {
    throw "Division by zero condition!";
  }
  return (a/b);
}
```

Catch exception

The catch block following the try block catches any exception.

You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
  // protected code
} catch( ExceptionName e ) {
  // code to handle ExceptionName exception
}
```

Above code will catch an exception of **ExceptionName** type.

Catch exception

catch block is intended to catch the error and handle the exception condition.

We can have multiple catch blocks to handle different types of exception and perform different actions when the exceptions occur.

For example, we can display descriptive messages to explain why any particular exception occured.

In the below program, if the value of integer in the array x is less than 0, we are throwing a numeric value as exception and if the value is greater than 0, then we are throwing a character value as exception. And we have two different catch blocks to catch those exceptions.

Practice question

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x[3] = {-1,2};
    for(int i=0; i<2; i++)
    {
        int ex = x[i];
        try
        {
            if (ex > 0)
                // throwing numeric value as exception
                throw ex;
```

```
 else

            // throwing a character as exception
            throw 'e';
    }
    catch (int ex)  // to catch numeric exceptions
    {
       cout << "Integer exception\n";
    }
    catch (char ex) // to catch character/string exceptions
    {
       cout << "Character exception\n";
    }
  }
}
Output: Character exception
Integer exception
```

Catch exception (Generalized)

Below program contains a generalized catch block to catch any uncaught errors/exceptions. catch(...) block takes care of all type of exceptions. In the below program, both the exceptions are being catched by a single catch block..

```
int main()
{
    int x[3] = {-1,2};
    for(int i=0; i<2; i++)
    {
        int ex = x[i];
        try
        {
            if (ex > 0)
                // throwing numeric value as exception
                throw ex;
```

Catch exception (Generalized)

```
 else

            // throwing a character as exception
            throw 'e';
      }
      catch (...)  // to catch numeric exceptions
      {
         cout << "Special exception\n";
      }


   }
}
```

Output:
Special exception
Special exception

Catch exception example

We can even have separate catch blocks to handle integer and character exception along with the generalized catch block.

```
int main()
{
    int x[3] = {-1,0,2}; //array of 3 values
    for(int i=0; i<3; i++)
    {
        int ex = x[i];
        try
        {
            if (ex > 0) //ex value is 2
                // throwing numeric value as exception
                throw ex;
```

Catch exception example

```
else if (ex < 0) //ex value is -1
        throw "EX";
     else  //ex value is 0
        // throwing a character as exception
        throw 'e';
   }
   catch (int ex)  // to catch numeric exceptions
   {
      cout << "Integer exception\n";
   }
   catch (char ex) // to catch character exceptions
   {
      cout << "Character exception\n";
   }
```

Catch exception example

```
 catch (...)  // to catch generalised exceptions
     {
         cout << "Special exception\n";
     }
  }
}
```

Output:
Special exception
Character exception
Integer exception

Catch exception example

There is a special catch block called 'catch all' catch(…) that can be used to catch all types of exceptions.

For example, in the above program, an int and char is thrown as an exception, there are catch blocks for int and char exceptions, but there is no catch block for const char* which is "EX" , so catch(…) block will be executed.

Practice question

Simple example to show exception handling and program flow:

```cpp
#include <iostream>
using namespace std;
int main()
{
  int x = -1;
  cout << "Before try \n";
  try {
    cout << "Inside try \n";
    if (x < 0)
    {
      throw x;
      cout << "After throw (Never executed) \n";
    }
  }
```

Practice question

```
}
  catch (int x ) {
    cout << "Exception Caught \n";
  }

  cout << "After catch (Will be executed) \n";
  return 0;
}
```

Output:
Before try
Inside try
Exception Caught
After catch (Will be executed)

Implicit type conversion doesn't happen for primitive types. For example, in the following program 'a' is not implicitly converted to int . What will be the output of the program?

```
int main() {
  try  {
    throw 'a';
  }
  catch (int x)  {
     cout << "Caught " << x;
  }
  catch (...)  {
     cout << "Default Exception\n";
  }
}
```

A. Caught
B. Default Exception
C. 'a' will be printed
D. Compilation error

Implicit type conversion doesn't happen for primitive types. For example, in the following program 'a' is not implicitly converted to int . What will be the output of the program?

```cpp
int main() {
  try  {
    throw 'a';
  }
  catch (int x)  {
     cout << "Caught " << x;
  }
  catch (...)  {
     cout << "Default Exception\n";
  }
}
```

A. Caught
B. Default Exception
C. 'a' will be printed
D. Compilation error

What will be the output of the following program?

```
int main()
{
  try {
    throw 'a';
  }
  catch (int x) {
    cout << "Caught ";
  }
}
```

A. 'a' will be displayed
B. Caught
C. Compilation error
D. Program terminates abnormally

What will be the output of the following program?

```
int main()
{
   try  {
      throw 'a';
   }
   catch (int x)  {
       cout << "Caught ";
   }
}
```

A. 'a' will be displayed
B. Caught
C. Compilation error
D. Program terminates abnormally

Note: If an exception is thrown and not caught anywhere, the program terminates abnormally.

MCQ

What should be put in try block?
1. Statements that might cause exceptions
2. Statements that should be skipped in case of an exception

A. Option 1
B. Option 1 & 2
C. Only 2
D. None of the above

What should be put in try block?
1. Statements that might cause exceptions
2. Statements that should be skipped in case of an exception

A. Option 1
B. Option 1 & 2
C. Only 2
D. None of the above

Guess the output

What would be output of the following program? Type the answer in chatbox

```cpp
int main()
{
  try   {
    throw 'a';
  }
  catch (int param)    {
    cout << "int exceptionn";
  }
  catch (...)    {
    cout << "Default exceptionn";
  }
  cout << "After Exception";
}
```

Guess the output

What would be output of the following program? Type the answer in chatbox

Output:
Default exception
After exception

Assignment

Write a program to create an array of 5 integers. Access this array using index variable. Handle the condition where programmer accidentally accesses any index of array which is out of bound using exception handling.

```
int main () {
  char myarray[10];
  try
  {
    for (int n=0; n<=10; n++)
    {
      if (n>9) throw n;
      myarray[n]='z';
    }
  }
  catch (int n)
  {
    cout << "Array out of bound Exception: " << endl;
  }
  return 0;
}
```

Any Questions ??
Any Questions??

Thank You!

See you guys in next class.