Exception Handling

LOVELY PROFESSIONAL UNIVERSITY

PUNJAB (INDIA)

Today’s Agenda

- Basics of exception handling
- Exception handling mechanism
- Throwing mechanism
- Catching mechanism
- Rethrowing an exception

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

```
#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

```
#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;
                }
         }
```

Practice question

```
 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)

```
//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)

```
     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

```
#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;
```

Practice question

```
 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:

```
#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)
```

MCQ

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

MCQ

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

MCQ

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

MCQ

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

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

Guess the output

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

```
 int main()
{
    try    {
       throw 'a';
    }
    catch (int param)     {
        cout << "int exceptionn"; //A
    }
    catch (...)     {
        cout << “Default exceptionn";//B
    }
    cout << "After Exception";//C
  }
```

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.

Assignment- Solution

```
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;
}
```

Rethrowing an exception

In C++, try-catch blocks can be nested.

Also, an exception can be re-thrown using “throw; ”

Rethrowing an expression from within an exception handler can be done by calling throw, by itself, with no exception.

This causes current exception to be passed on to an outer try/catch sequence.

An exception can only be rethrown from within a catch block.

When an exception is rethrown, it is propagated outward to the next catch block.

Consider the example given below. Revisit this slide after you go through example.

Rethrowing an exception

```
int main()
{
    try
    {
        try
        {
            throw 20;
        }
        catch (int n)
        {
            cout << "Inner Catchn";
            throw;
        }
    }
catch (int x)
    {
        cout << "Outer Catchn";
    }
    return 0;
}
```

Output:

```
Inner Catchn
Outer Catchn
```

Rethrowing an exception

```
#include <iostream>
using namespace std;
void MyHandler()
{
   try
   {
      throw "hello" ;
   }
   catch (const char*)
   {
   cout <<"Caught exception inside MyHandler\n";
   throw; //rethrow char* out of function
   }
}
```

Rethrowing an exception

```
int main()
{
   cout<< “Main start”;
   try
   {
       MyHandler();
   }
   catch(const char*)
   {
      cout <<”Caught exception inside Main\n”;
   }
       cout << “Main end”;
       return 0;
}
```

Rethrowing an exception

Output:

```
Main start
Caught exception inside MyHandler
Caught exception inside Main
Main end
```

Explanation: The try block in the main() function calls function MyHandler(). The try block in function MyHandler() throws an exception “Hello”. The handler catch (const char*) catches this exception. The handler then rethrows char* out of function with the statement throw to the next dynamically enclosing try block: the try block in the main() function. The generic handler in main catch(...) catches char* exception.

Exception in function calls

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

void fun(int *ptr, int x)  // Dynamic Exception specification
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}
```

Exception in function calls

```
int main()
{
    try {
       fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
}
```

Explanation: If the compiler encounters an exception in a try block, it will try each handler in order of appearance. If the run time cannot find a matching handler in the current scope, the run time will continue to find a matching handler in a dynamically surrounding try block. In function fun(), the run time could not find a handler to handle the exception of type E thrown. The run time finds a matching handler in a dynamically surrounding try block: the try block in the main() function.

Points to remember

A catch block of the form catch(...) must be the last catch block following a try block or an error occurs.

This placement ensures that the catch(...) block does not prevent more specific catch blocks from catching exceptions intended for them.

When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block. Refer next slide for example

If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class. If we put base class first then the derived class catch block will never be reached.

When an exception is thrown and not caught, the program terminates abnormally.

Exception in Constructor / Destructor

```
class Test {
public:
    Test() { cout << "Constructor of Test " << endl; }
    ~Test() { cout << "Destructor of Test " << endl; }
};
int main()
{
    try {
        Test t1; //creating object of Test class using default constructor
        throw 10;
    }
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}
```

```
Constructor of Test
Destructor of Test
Caught 10
```

Exception in inheritance

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

class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   try {
       throw d;
   }
   catch(Base b) {
        cout<<"Caught Base Exception";
   }
catch(Derived d)
//This catch block is NEVER executed
 {
        cout<<"Caught Derived Exception";
   }
   return 0;
}
```

Output:

```
Caught Base Exception
```

Note: Catching a base class exception before derived is not allowed by the compiler itself. Compiler might give warning about it, but compiles the code.

Exception in inheritance

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

class Base {};
class Derived: public Base {};
int main()
{
  Derived d;
  try {
     throw d;
  }
 catch(Derived d)
 {
      cout<<"Caught Derived Exception";
  }
}
```

```
catch(Base b) {
      cout<<"Caught Base Exception";
  }
```

Output:

```
Caught Derived Exception
```

Note: If we change the order of catch statements then both catch statements become reachable. Above is the modified program and it prints *“Caught Derived Exception”*

Standard exceptions

<This slide is only for knowledge, and won’t be included for exam>

C++ provides a list of standard exceptions defined in <exception> which we can use in our programs.

1. std::exception :An exception and parent class of all the standard C++ exceptions.
2. std::bad_alloc : This can be thrown by new.
3. std::range_error : This is occurred when you try to store a value which is out of range.
4. std::underflow_error: This is thrown if a mathematical underflow occurs.
5. std::overflow_error: This is thrown if a mathematical overflow occurs.

Advantages of exception handling

Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors.

These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable.

With try catch blocks, the code for error handling becomes separate from the normal flow.

Programmers can deal with them at some level within the program

If an error can't be dealt with at one level, then it will automatically be shown at the next level, where it can be dealt with.

Advantages of exception handling

Functions/Methods can handle any exceptions they choose: A function can throw many exceptions, but may choose to handle some of them.

The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.

In C++, a function can specify the exceptions that it throws using the throw keyword.

The caller of this function must handle the exception in some way (either by specifying it again or catching it)

Memory allocation failure

If memory allocation using new is failed in C++ then how it should be handled?

When an object of a class is created dynamically using new operator, the object occupies memory in the heap.

Below are the major thing that must be kept in mind:

1. What if sufficient memory is not available in the heap memory, and how it should be handled? - using try and catch block
2. If memory is not allocated then how to avoid the project crash? – prevent memory crash by throwing an exception

Memory allocation failure

```
#include <iostream>
using namespace std;
int main()
{
    // Allocate huge amount of memory
    long MEMORY_SIZE = 0x7fffffff;
     // Put memory allocation statement
    // in the try catch block
    try {
        char* ptr = new char[MEMORY_SIZE];
         // When memory allocation fails, below line is not be executed
        // & control will go in catch block
        cout << "Memory is allocated“ << " Successfully" << endl;
    }
```

Memory allocation failure

```
// Catch Block handle error
    catch (bad_alloc e) {

        cout << "Memory Allocation“ << " is failed: “    << e.what()   << endl;
    }

    return 0;
}
```

Output:

```
Memory Allocation is failed: std::bad_alloc
```

The above memory failure issue can be resolved without using the try-catch block. It can be fixed by using nothrow version of the new operator.

Memory allocation failure

The nothrow constant value is used as an argument for operator new and operator new[] to indicate that these functions shall not throw an exception on failure but return a null pointer instead.

By default, when the new operator is used to attempt to allocate memory and the handling function is unable to do so, a bad_alloc exception is thrown.

But when nothrow is used as an argument for new, and it returns a null pointer instead.

This constant (nothrow) is just a value of type nothrow_t, with the only purpose of triggering an overloaded version of the function operator new (or operator new[]) that takes an argument of this type.

Practice question

What will be the output of the following program?

```
class Base {};
class Derived: public Base {};
int main(){
  Derived d;
  try {
     throw d;
  }
  catch(Base b) {
      cout<<"Caught Base Exception";
  }
  catch(Derived d) {
      cout<<"Caught Derived Exception";
  }
 }
```

Practice question

What will be the output of the following program?

```
class Base {};
class Derived: public Base {};
int main(){
   Derived d;
   try {
       throw d;
   }
   catch(Base b) {
        cout<<"Caught Base Exception";
   }
   catch(Derived d) {
        cout<<"Caught Derived Exception";
   }
 }
```

Output:

```
Caught Base Exception
```

Practice question

What will be the output of the following program?

```
int main(){
    try    {
       throw 'a';
    }
    catch (int param)     {
        cout << "int exceptionn";
    }
    catch (...)     {
        cout << "default exceptionn";
    }
    cout << "After Exception";
    return 0;
}
```

Practice question

What will be the output of the following program?

```
int main(){
   try   {
     throw 'a';
   }
   catch (int param)    {
      cout << "int exceptionn";
   }
   catch (...)    {
      cout << "default exceptionn";
   }
   cout << "After Exception";
   return 0;
}
```

Output:

```
default Exception
After Exception
```

Practice question

What will be the output of the following program?

```
int main() {
    try    {
      throw 10;
    }
    catch (...)    {
       cout << "default exception";
    }
    catch (int param)    {
       cout << "int exception";
    }

    return 0;
}
```

Options:

1. default exception
2. int Exception
3. Compile error
4. default exception int exception

Practice question

What will be the output of the following program?

```
int main() {
    try    {
       throw 10;
    }
    catch (...)    {
        cout << "default exception";
    }
    catch (int param)    {
        cout << "int exception";
    }

    return 0;
}
```

Options:

1. default exception
2. int Exception
3. Compile error
4. default exception int exception

Practice question

Which of the following is true about exception handling in C++?

1. When an exception is rethrown, it is propagated outward to the next catch block.

2. A catch block of the form catch(...) must be the last catch block following a try block or an error occurs.

Options:

1. 1 only
2. 2 only
3. Both are true
4. Both are false

Practice question

Which of the following is true about exception handling in C++?

1. When an exception is rethrown, it is propagated outward to the next catch block.

2. A catch block of the form catch(...) must be the last catch block following a try block or an error occurs.

Options:

1. 1 only
2. 2 only
3. Both are true
4. Both are false

Practice question

What happens in C++ when an exception is thrown and not caught anywhere like in the following program?

```
#include <iostream>
using namespace std;
 int fun()
{
    throw 10;
}

 int main() {
   fun();
   return 0;
}
```

Options:

1. Compile error
2. Abnormal program termination
3. Program doesn’t print anything and terminates normally
4. None of the above

Practice question

What happens in C++ when an exception is thrown and not caught anywhere like in the following program?

```
#include <iostream>
using namespace std;
 int fun() throw (int)
{
   throw 10;
}

 int main() {
   fun();
   return 0;
}
```

Options:

1. Compile error
2. Abnormal program termination
3. Program doesn’t print anything and terminates normally
4. None of the above

Practice Question

Write a c++ program to accept a character from keyboard. If it is not an alphabet, not a number then throw an appropriate exception and catch it using multiple catch statements and generalized catch.

Hint:

```
If not an alphabet
    throw(“not an alphabet”)
Else if not a number
    throw “not a number
Else
    throw “Special char”
```

Practice Question

```
#include<iostream>
using namespace std;
int main(){
        char ch;
        cout<<"Enter a char";
        cin>> ch;
        try
        {
        If ((!isalpha(ch)) && (!isdigit(ch)))
                throw ch;
        else if (!isalpha(ch))
                throw "not an alphabet";
        else if (!isdigit(ch))
                throw 1;
        }
```

Practice Question

```
 catch(const char* ex)
{
           cout<<ex<<endl;
}
catch (int n)
{
 cout <<"not a number"<<endl;
}
catch (...)
{
cout<< " It is a special character";
 }
return 0;

}
```

Assignment

Write a c++ program to accept 5 numbers from user in an array and handle the exceptions for positive , -ve numbers and equal to 0 numbers using multiple try catch statements.

Sample Input: 1 -2 0 2 -4

Sample output:

```
1- Positive number
-2 – negative number
0 – Zero
2 -positive number
-4 – negative number
```

QNA Time

Any Questions??

Thank You!

See you guys in next class.