UNIT-V

C++ Exception Handling

Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

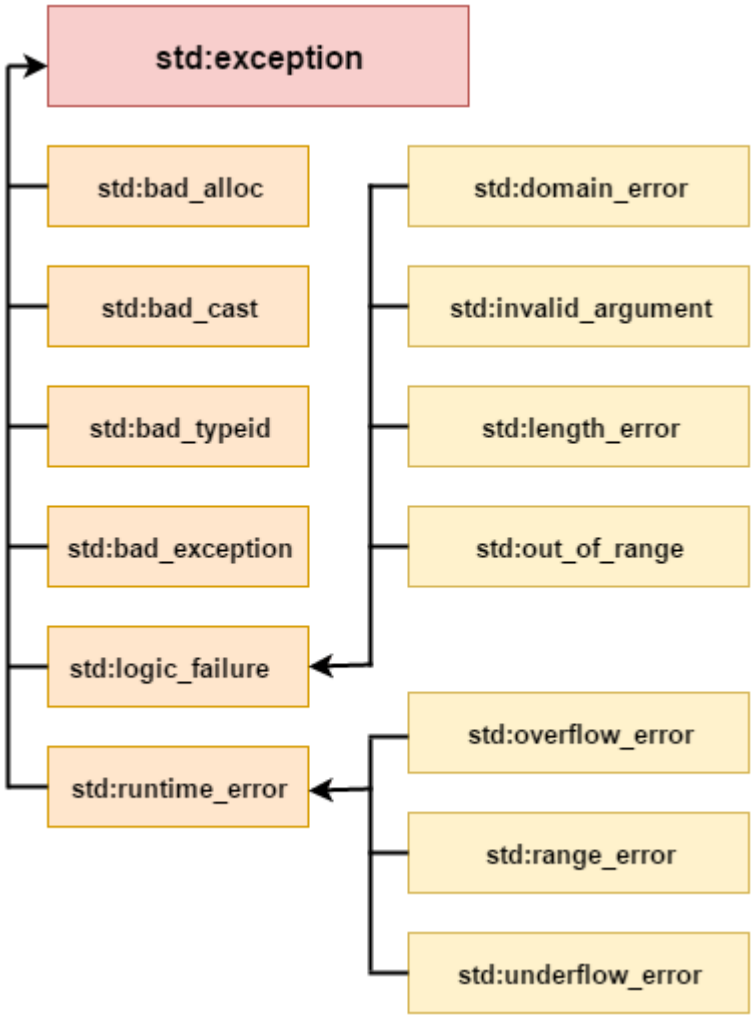
In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Advantage

It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

C++ Exception Classes

In C++ standard exceptions are defined in <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:



All the exception classes in C++ are derived from std::exception class. Let's see the list of C++ common exception classes.

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

C++ Exception Handling Keywords

In C++, we use 3 keywords to perform exception handling:

- try
- catch, and
- throw

C++ try and catch

Exception handling in C++ consist of three keywords: `try`, `throw` and `catch`:

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `throw` keyword throws an exception when a problem is detected, which lets us create a custom error.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

The `try` and `catch` keywords come in pairs:

Example

```
try {
  // Block of code to try
  throw exception; // Throw an exception when a problem arise
}
catch () {
  // Block of code to handle errors
}
```

Consider the following example:

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


int main() {
  try {
    int age = 20;
    if (age >= 18) {
      cout << "Access granted - you are old enough.";
    } else {
      throw (age);
    }
  }
  catch (int myNum) {
    cout << "Access denied - You must be at least 18 years old.\n";
    cout << "Age is: " << myNum;
  }
  return 0;
}
```

Example explained

We use the `try` block to test some code: If the `age` variable is less than `18`, we will `throw` an exception, and handle it in our `catch` block.

In the `catch` block, we catch the error and do something about it. The `catch` statement takes a **parameter**: in our example we use an `int` variable (`myNum`) (because we are throwing an exception of `int` type in the `try` block (`age`)), to output the value of `age`.

If no error occurs (e.g. if `age` is `20` instead of `15`, meaning it will be be greater than 18), the `catch` block is skipped:

```
#include <iostream>

using namespace std;


int main() {

  try {

    int age = 15;

    if (age >= 18) {

      cout << "Access granted - you are old enough.";

    } else {

      throw 505;

    }

  }

  catch (int myNum) {

    cout << "Access denied - You must be at least 18 years old.\n";

    cout << "Error number: " << myNum;

  }

  return 0;

}
```

OUTPUT:

```
Access denied - You must be at least 18 years old.
Error number: 505
```

When an exception is thrown, it is caught by its corresponding catch statement, which processes the exception. There can be more than one catch statement associated with a try. Which catch statement is used is determined by the type of the exception. That is, if the data type specified by a catch matches that of the exception, then that catch statement is executed (and all others are bypassed). When an exception is caught, arg will receive its value. Any type of data may be

caught, including classes that you create. If no exception is thrown (that is, no error occurs within the try block), then no catch statement is executed.

The general form of the throw statement is shown here:

throw exception;

throw generates the exception specified by exception. If this exception is to be caught, then throw must be executed either from within a try block itself, or from any function called from within the try block (directly or indirectly).

```
// A simple exception handling example.

#include <iostream>

using namespace std;

int main()

{

cout << "Start\n";

try

{ // start a try block

cout << "Inside try block\n";

throw 100; // throw an error

cout << "This will not execute";

}

catch (int i)

{ // catch an error

cout << "Caught an exception -- value is: ";

cout << i << "\n";

}

cout << "End";

return 0;
```

```
}
```

This program displays the following output:

```
Start
Inside try block
Caught an exception -- value is: 100
End
// This example will not work.
#include <iostream>
using namespace std;
int main()
{
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
throw 100; // throw an error
cout << "This will not execute";
}
catch (double i) { // won't work for an int exception
cout << "Caught an exception -- value is: ";
cout << i << "\n";
}
cout << "End";
return 0;
}
```

This program produces the following output because the integer exception will not be

caught by the catch(double i) statement. (Of course, the precise message describing abnormal termination will vary from compiler to compiler.)

```
Start
Inside try block
Abnormal program termination
```

An exception can be thrown from outside the try block as long as it is thrown by a function that is called from within try block. For example, this is a valid program.

```
/* Throwing an exception from a function outside the
try block.
*/
#include <iostream>
using namespace std;
void Xtest(int test)
{
cout << "Inside Xtest, test is: " << test << "\n";
if(test) throw test;
}
int main()
{
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
Xtest(0);
Xtest(1);
Xtest(2);
```

```
}
catch (int i) { // catch an error
cout << "Caught an exception -- value is: ";
cout << i << "\n";
}
cout << "End";
return 0;
}
```

This program produces the following output:

```
Start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught an exception -- value is: 1
End
```

A try block can be localized to a function. When this is the case, each time the function is entered, the exception handling relative to that function is reset. For example, examine this program.

```
#include <iostream>
using namespace std;
// Localize a try/catch to a function.
void Xhandler(int test)
{
try{
if(test) throw test;
```

```
}
catch(int i) {
cout << "Caught Exception #: " << i << '\n';
}
}
int main()
{
cout << "Start\n";
Xhandler(1);
Xhandler(2);
Xhandler(0);
Xhandler(3);
cout << "End";
return 0;
}
```

This program displays this output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught Exception #: 3
End
```

Catching Class Types

An exception can be of any type, including class types that you create. Actually, in real-world programs, most exceptions will be class types rather than built-in types. Perhaps the most common reason that you will want to define a class type for an

exception is to create an object that describes the error that occurred. This information can be used by the exception handler to help it process the error. The following example demonstrates this.

```
// Catching class type exceptions.
#include <iostream>
#include <cstring>
using namespace std;
class MyException {
public:
char str_what[80];
int what;
MyException() { *str_what = 0; what = 0; }
MyException(char *s, int e) {
strcpy(str_what, s);
what = e;
}
};
int main()
{
int i;
try {
cout << "Enter a positive number: ";
cin >> i;
if(i<0)
throw MyException("Not Positive", i);
```

```
}

catch (MyException e) { // catch an error

cout << e.str_what << ": ";

cout << e.what << "\n";

}

return 0;

}
```

Here is a sample run:

```
Enter a positive number: -4

Not Positive: -4
```

The program prompts the user for a positive number. If a negative number is entered, an object of the class MyException is created that describes the error. Thus, MyException encapsulates information about the error. This information is then used by the exception handler. In general, you will want to create exception classes that will encapsulate information about an error to enable the exception handler to respond effectively

Using Multiple catch Statements

As stated, you can have more than one catch associated with a try. In fact, it is common

to do so. However, each catch must catch a different type of exception. For example,

this program catches both integers and strings.

```
#include <iostream>

using namespace std;

// Different types of exceptions can be caught.

void Xhandler(int test)

{

try{

if(test) throw test;
```

```
else throw "Value is zero";
}
catch(int i) {
cout << "Caught Exception #: " << i << '\n';
}
catch(const char *str) {
cout << "Caught a string: ";
cout << str << '\n';
}
}
int main()
{
cout << "Start\n";
Xhandler(1);
Xhandler(2);
Xhandler(0);
Xhandler(3);
cout << "End";
return 0;
}
```

This program produces the following output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
```

Caught a string: Value is zero

Caught Exception #: 3

End

As you can see, each catch statement responds only to its own type.

In general, catch expressions are checked in the order in which they occur in a program. Only a matching statement is executed. All other catch blocks are ignored.

Catching All Exceptions

In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type. This is easy to accomplish. Simply use this form of catch.

```
catch(...) {
// process all exceptions
}
```

Here, the ellipsis matches any type of data. The following program illustrates catch(...).

```
// This example catches all exceptions.
#include <iostream>
using namespace std;
void Xhandler(int test)
{
try{
if(test==0) throw test; // throw int
if(test==1) throw 'a'; // throw char
if(test==2) throw 123.23; // throw double
}
catch(...) { // catch all exceptions
```

```
cout << "Caught One!\n";
}
}
int main()
{
cout << "Start\n";
Xhandler(0);
Xhandler(1);
Xhandler(2);
cout << "End";
return 0;
}
```

This program displays the following output.

```
Start
Caught One!
Caught One!
Caught One!
End
```

As you can see, all three throws were caught using the one catch statement.

One very good use for catch(...) is as the last catch of a cluster of catches. In this capacity it provides a useful default or "catch all" statement. For example, this slightly different version of the preceding program explicity catches integer exceptions but relies upon catch(...) to catch all others.

Handling Derived-Class Exceptions

You need to be careful how you order your catch statements when trying to catch exception types that involve base and derived classes because a catch clause for a base class will also match any class derived from that base. Thus, if you want to catch exceptions of both a base class type and a derived class type, put the derived class first in the catch sequence. If you don't do this, the base class catch will also catch all derived classes. For example, consider the following program.

```
// Catching derived classes.

#include <iostream>

using namespace std;

class B {

};

class D: public B {

};

int main()

{

D derived;

try {

throw derived;

}

catch(B b) {

cout << "Caught a base class.\n";

}

catch(D d) {

cout << "This won't execute.\n";

}

return 0;
```

```
}
```

Here, because derived is an object that has B as a base class, it will be caught by the first catch clause and the second clause will never execute. Some compilers will flag this condition with a warning message. Others may issue an error. Either way, to fix this condition, reverse the order of the catch clauses.

Rethrowing an Exception

```
// Example of "rethrowing" an exception.

#include <iostream>

using namespace std;

void Xhandler()

{

try {

throw "hello"; // throw a char *

}

catch(const char *) { // catch a char *

cout << "Caught char * inside Xhandler\n";

throw ; // rethrow char * out of function

}

}

int main()

{

cout << "Start\n";
```

```
try{

Xhandler();

}

catch(const char *) {

cout << "Caught char * inside main\n";

}

cout << "End";

return 0;

}
```

This program displays this output:

```
Start

Caught char * inside Xhandler

Caught char * inside main

End
```

OTHER EXAMPLES:

```
#include<iostream>

#include<conio.h>

using namespace std;

int main()

{

  int a,b;
```

```
  try
  {
     cout<<"enter tw numbers";
     cin>>a>>b;
     if(b==0) throw 0;
     cout<<"division="<<a/b;
  }
catch(int x)
{
cout<<"divison not possible";
}
} //main close
```

Nested try,catch example

```
using namespace std;
int main()
{
  try
   {
     cout<<"welcome\n";
        try
         {
           cout<<"to inner try block\n";
           throw 10;
         }
         catch(int x)
          {
            cout<<"throw working\n";
             throw x;
          }
}
catch(int y)
{
cout<<"outer try block \n";
}
```

```
}       //main close


using namespace std;
class test
{
  public:
        test()
        {
        cout<<"I am the constructor\n";
        }

        ~test()
        {
        cout<<"i am the destructor\n";
        }
};

int main()
{
  try
   {
        cout<<"welcome to exception handling\n";
        test t;
        throw 10;
        cout<<"testing destructor\n";
  }
catch(...)
{
cout<<"thank you";
}
}
```

C++ try/catch

In C++ programming, exception handling is performed using try/catch statement. The C++ **try block** is used to place the code that may occur exception. The **catch block** is used to handle the exception.

C++ example without try/catch

```
#include <iostream>
using namespace std;
float division(int x, int y) {
  return (x/y);
```

```
}
int main () {
  int i = 50;
  int j = 0;
  float k = 0;
    k = division(i, j);
    cout << k << endl;
  return 0;
}
```

Output:

```
Floating point exception (core dumped)
```

C++ try/catch example

```
#include <iostream>
using namespace std;
float division(int x, int y) {
  if( y == 0 ) {
    throw "Attempted to divide by zero!";
  }
  return (x/y);
}
int main ()
{
  int i = 25;
  int j = 0;
  float k = 0;
  try {
    k = division(i, j);
    cout << k << endl;
  }catch (const char* e) {
    cerr << e << endl;
  }
  return 0;
}
```

Output:

```
Attempted to divide by zero!
```

C++ User-Defined Exceptions

The new exception can be defined by overriding and inheriting **exception** class functionality.

C++ user-defined exception example

Let's see the simple example of user-defined exception in which **std::exception** class is used to define the exception.

```
#include <iostream>
#include <exception>
using namespace std;
class MyException : public exception{
   public:
      const char * what() const throw()
      {
         return "Attempted to divide by zero!\n";
      }
};
int main()
{
   try
   {
      int x, y;
      cout << "Enter the two numbers : \n";
      cin >> x >> y;
      if (y == 0)
      {
         MyException z;
         throw z;
      }
      else
      {
         cout << "x / y = " << x/y << endl;
      }
   }
   catch(exception& e)
   {
      cout << e.what();
   }
}
```

Output:

```
Enter the two numbers :
10
2
x / y = 5
```

Output:

```
Enter the two numbers :
10
0
Attempted to divide by zero!
```

```
-->
```

Note: In above example what() is a public method provided by the exception class. It is used to return the cause of an exception.

Stack Unwinding in C++

Here we will see what is the meaning of stack unwinding. When we call some functions, it stores the address into call stack, and after coming back from the functions, pops out the address to start the work where it was left of.

The stack unwinding is a process where the function call stack entries are removed at runtime. To remove stack elements, we can use exceptions. If an exception is thrown from the inner function, then all of the entries of the stack is removed, and return to the main invoker function.

Let us see the effect of stack unwinding through an example.

Example Code

```
#include <iostream>

using namespace std;


void function1() throw (int) { //this function throws exception

   cout<<"\n Entering into function 1";

   throw 100;

   cout<<"\n Exiting function 1";

}


void function2() throw (int) { //This function calls function 1

   cout<<"\n Entering into function 2";

   function1();

   cout<<"\n Exiting function 2";

}


void function3() { //function to call function2, and handle

   //exception thrown by function1

   cout<<"\n Entering function 3 ";
```

```
   try {
      function2(); //try to execute function 2
   }
   catch(int i) {
      cout<<"\n Caught Exception: "<<i;
   }
   cout<<"\n Exiting function 3";
}


int main() {
   function3();
   return 0;
}
```

Output

```
Entering function 3
Entering into function 2
Entering into function 1
Caught Exception: 100
Exiting function 3
```

Here we can see that the control stores info of function3, then enters into function2, and then into function1. After that one exception occurs, so it removes all of the information from stack, and returns back to function3 again.