EXCEPTION HANDLING

EXCEPTIONS

- Exceptions are run time anomalies or unusual conditions that a program may encounter during execution.
- Conditions such as
  - Division by zero
  - Access to an array outside of its bounds
  - Running out of memory
  - Running out of disk space
- It was not a part of original C++.
- It is a new feature added to ANSI C++.

EXCEPTIONS

- Exceptions are of 2 kinds
  - Synchronous Exception:
    - Out of range index
    - Over flow
  - Asynchronous Exception: Error that are caused by events beyond the control of the program
    - Keyboard interrupts

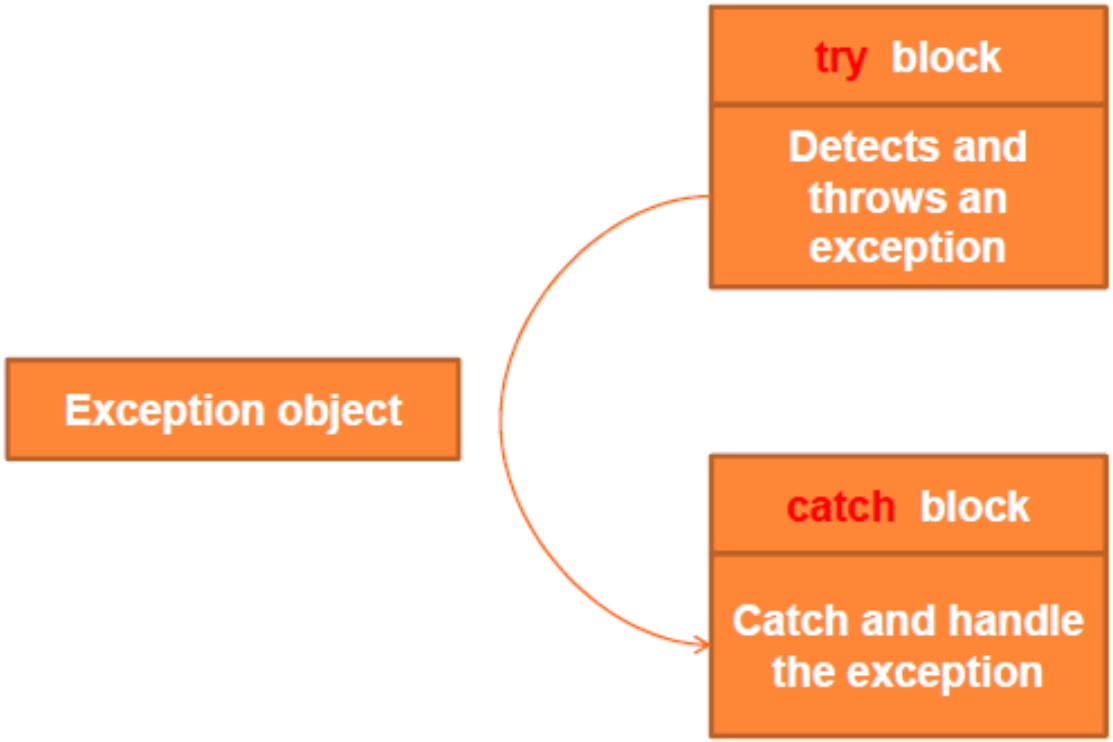
- In C++ only synchronous exception can be handled.

Exception Handling

- Exception handling
  - Can resolve exceptions
    - Allow a program to continue executing or
    - Notify the user of the problem and
    - Terminate the program in a controlled manner
  - Makes programs robust and fault-tolerant
- Exception handling mechanism
  - Find the problem (Hit the exception)
  - Inform that an error has occurred (Throw the exception)
  - Receive the error information (Catch the exception)
  - Take corrective action (handle the exception)

Exception handling mechanism

- It is basically build upon three keywords
  - Try
  - Throw
  - Catch



Exception handling mechanism

- The keyword **try** is used to preface a block of statements which may generate exceptions.
- When an exception is detected, it is thrown using a **throw** statement in the try block.
- A **catch** block defined by the keyword ‘catch’ catches the exception and handles it appropriately.
- The catch block that catches an exception must immediately follow the try block that throws the exception.

Exception handling mechanism

```
try
{
   …                          //Block of Statements which detects and
  throw exception;            // throw an exception
   …
}
catch(type arg)               //catch exception
{
…                             //Block of statements that handles exception
…
}
….
….
```

Example

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

Exception Handling

- Enclose code that may have an error in try block.
- Follow with one or more catch blocks. Each catch block has an exception handler
- If exception occurs and matches parameter in catch block, code in catch block executed
- If no exception thrown, exception handlers skipped and control resumes after catch blocks
- throw point - place where exception occurred Control cannot return to throw point

Exception Handling

- Exceptions are objects used to transmit information about a problem.
- If the type of the object thrown matches the arg type in the catch statement, the catch block is executed.

Example

```
#include<iostream>
using namespace std;
int main()
{int a,b;
cout<<"Enter the values of a and
b";
cin>>a>>b;
int x = a- b;
try
{
if(x!=0)
{cout<<"result(a/x) =
"<<a/x<<"\n";
}
else{
throw(x);}}
catch(int i)
{
cout<<"exception caught :
x =  "<<x<<"\n";
}
cout<<"End";
return 0;
}
```

First Run Output:

Enter the values of a and b
20 15
Result(a/x) =4
End

Second Run Output:

Enter the values of a and b
10 10
exception caught : x =0
End

Exception Handling

- Often, Exceptions are thrown by functions that are invoked from within the try blocks.
- The point at which the throw is executed is called the throw point.
- Once an exception is thrown to the catch block, control cannot return to the throw point.

// The try block is immediately followed by the catch block, irrespective of the location of throw point

```
#include<iostream>
using namespace std;
void divide(int x, int y, int z)
{
     cout<<"inside the function\n";
      if((x-y)!=0)
        { int  R=z/(x-y);
          cout<<"Result="<<R;}
      else{
       throw(x-y);
       }
}
```

```
int main()
{
  try{
      cout<<"inside try block\n";
      divide(10,20,30);
      divide(10,10,20);
      }
   catch(int i)            //catches exception
   {
    cout<<"\ncaught exception";
   }
return 0;
}
```

Output:

Inside try block
Inside the function
Result = -3
Inside the function
Caught exception

THROWING MECHANISM

- The **throw** statement can have one of the following 3 forms
  - throw(exception)
  - throw exception
  - throw //used to re-throw a exception
- The operand object exception can be of any type, including **constant**.
- It is also possible to throw an object not intended for error handling.

THROWING MECHANISM

- Throw point can be in a deeply nested scope within a try block or in a deeply nested function call.
- In any case, control is transferred to the catch statement.

CATCHING MECHANISM

- The type indicates the type of exception the catch block handles.
- The parameter arg is an **optional** parameter name.
- The catch statement catches an exception whose type matches with the type of the catch argument

```
catch(type arg)
{
…
…
}
```

CATCHING MECHANISM

- If the parameter in the catch statement is named, then the parameter can be used in the exception handling code.
- If a catch statement does not match the exception it is skipped.
- More than one catch statement can be associated with a try block.

CATCHING MECHANISM

- When an exception is thrown, the exception handlers are searched **in order** for a match.
- The first handler that yields a match is executed.
- If several catch statement matches the type of an exception the first handler that matches the exception type is executed.

Class type Example

```
// 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;}
```

Output:
Enter a positive number: -4
Not Positive: -4

Handling Derived-Class Exceptions

- **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.

Example

```
// 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;
}
```

Output:

Caught a base class.

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.

Multiple catch statements

```
try {
// try block
}
catch (type1 arg) {
// catch block
}
catch (type2 arg) {
// catch block
}
catch (type3 arg) {
// catch block
}
…
…
catch (typeN arg) {
// catch block}
```

Example(Multiple catches)

```
#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;
}
```

Output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught a string: Value is zero
Caught Exception #: 3
End
```

Catching All Exceptions

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

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

Catching All Exceptions

```
// 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;
}
```

Output:

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

RETHROWING AN EXCEPTION

- A handler may decide to rethrow the exception caught without processing it.
- In such a case we have to invoke **throw** without any arguments as shown below

  throw;
- This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after the enclosing try block

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;
}
```

Output:

Start
Caught char * inside Xhandler
Caught char * inside main
End

SPECIFYING EXCEPTION

- It is possible to restrict a function to throw certain specific exceptions by adding a **throw** list clause to the function definition.

```
type function(arg-list) throw(type-list)
{
… … …
… … …
… … …
}
```

SPECIFYING EXCEPTION

- The type-list specifies the type of exception that may be thrown.
- Throwing any other kind of exception will cause abnormal program termination.
- If you want to prevent a function from throwing any exception, you may do so by making the type-list empty.

Example

```
//demonstrates how we can restrict a function to throw only certain types and not all.
#include<iostream>
using namespace std;
void test(int x) throw(char,double)
{
 if(x==0)
    throw 'x'; //char
 else
  if(x==1)
    throw x;  //int
  else
  if(x==-1)
    throw 1.0;  //double
}
```

```
int main()
{ try{
     cout<<"Testing throw restrictions \n";
     cout<<"x==0\n";
     test(0);
     cout<<"x==1\n";
     test(1);
     cout<<"x==-1\n";
     test(-1);
     cout<<"x==2\n";
     test(2);}
   catch(char c)
{ cout<<"caught a character\n";}

 catch(int m)
{ cout<<"caught an integer\n";}

catch(double d)
{ cout<<"caught a double\n";}
return 0;}
```

Output:
Testing throw restrictions x==0
caught a character

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

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

int main()
{
   try {
      try  {
         throw 20;
      }
      catch (int n) {
          cout << "Handle Partially\n";
          throw;   //Re-throwing an exception
      }
   }
catch (int n) {
       cout << "Handle remaining ";
                }
   return 0;
}
```

Output:
Handle Partially
Handle remaining

//When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block

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

class Test {
public:
   Test() { cout << "Constructor of Test " << endl; }
  ~Test() { cout << "Destructor of Test "  << endl; }
};

int main() {
  try {
    Test t1;
    throw 10;
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
}
```

Output:
Constructor of Test
Destructor of Test
Caught 10