Exception Handling in C++

One of the advantages of C++ over C is Exception Handling. Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions: a)Synchronous, b)Asynchronous(Ex:which are beyond the program's control, Disc failure etc). C++ provides following specialized keywords for this purpose.
try: represents a block of code that can throw an exception.
catch: represents a block of code that is executed when a particular exception is thrown.
throw: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

Why Exception Handling?
Following are main advantages of exception handling over traditional error handling.

1) *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.

2) *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)

3) *Grouping of Error Types:* In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

C++ Exceptions:

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (throw an error).

C++ try and catch:

Exception handling in C++ consists 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:

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:

Exception Handling in C++

1) Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks.

```
#include <iostream>
usingnamespacestd;

intmain()
{
  intx = -1;

  // Some code
  cout << "Before try \n";
  try{
    cout << "Inside try \n";
    if(x < 0)
    {
      throwx;
      cout << "After throw (Never executed) \n";
    }
  }
  catch(intx ) {
    cout << "Exception Caught \n";
  }

  cout << "After catch (Will be executed) \n";
  return0;
}
```

Output:

```
Before try

Inside try
```

Exception Caught

After catch (Will be executed)

2) There is a special catch block called ‘catch all’ catch(…) that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so catch(…) block will be executed.

```
#include <iostream>
usingnamespacestd;

intmain()
{
   try  {
     throw10;
   }
   catch(char*excp)  {
      cout << "Caught "<< excp;
   }
   catch(...)  {
      cout << "Default Exception\n";
   }
   return0;
}
```

Output:

Default Exception

3) Implicit type conversion doesn’t happen for primitive types. For example, in the following program ‘a’ is not implicitly converted to int

```
#include <iostream>
usingnamespacestd;

intmain()
{
   try  {
     throw'a';
   }
   catch(intx)  {
      cout << "Caught "<< x;
   }
   catch(...)  {
      cout << "Default Exception\n";
   }
   return0;
}
```

Output:

Default Exception

4) If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch a char.

```
#include <iostream>
```

```
usingnamespacestd;

intmain()
{
   try   {
     throw'a';
   }
   catch(intx)  {
      cout << "Caught ";
   }
   return0;
}
```

Output:

```
terminate called after throwing an instance of 'char'
```

```
This application has requested the Runtime to terminate it in an

unusual way. Please contact the application's support team for

more information.
```

We can change this abnormal termination behavior by writing our own unexpected function.
5) A derived class exception should be caught before a base class exception. See this for more details.
6) Like Java, C++ library has a standard exception class which is base class for all standard exceptions. All objects thrown by components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type
7) Unlike Java, in C++, all exceptions are unchecked. Compiler doesn't check whether an exception is caught or not (See this for details). For example, in C++, it is not necessary to specify all uncaught exceptions in a function declaration. Although it's a recommended practice to do so. For example, the following program compiles fine, but ideally signature of fun() should list unchecked exceptions.

```
#include <iostream>
usingnamespacestd;

// This function signature is fine by the compiler, but not recommended.
// Ideally, the function should specify all uncaught exceptions and function
// signature should be "void fun(int *ptr, int x) throw (int *, int)"
voidfun(int*ptr, intx)
{
   if(ptr == NULL)
      throwptr;
   if(x == 0)
      throwx;
   /* Some functionality */
}

intmain()
{
   try{
     fun(NULL, 0);
```

```
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return0;
}
```

Output:

```
Caught exception from fun()
```

A better way to write above code

```
#include <iostream>
usingnamespacestd;

// Here we specify the exceptions that this function
// throws.
voidfun(int*ptr, intx) throw(int*, int) // Dynamic Exception specification
{
    if(ptr == NULL)
        throwptr;
    if(x == 0)
        throwx;
    /* Some functionality */
}

intmain()
{
    try{
      fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return0;
}
```

(**Note :** The use of Dynamic Exception Specification has been deprecated after C++11, one of the reason maybe because it can randomly abort your program. This can happen when you throw an exception of an another type which is not mentioned in the dynamic exception specification, your program will abort itself, because in that scenario program calls(indirectly) terminate(), and which is by default call abort()).

Output:

```
Caught exception from fun()
```

8) In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using “throw; ”

```
#include <iostream>
usingnamespacestd;

intmain()
{
    try{
        try{
```

```
            throw20;
        }
        catch(intn) {
            cout << "Handle Partially ";
            throw; // Re-throwing an exception
        }
    }
    catch(intn) {
        cout << "Handle remaining ";
    }
    return0;
}
```

Output:

```
Handle Partially Handle remaining
```

A function can also re-throw a function using same “throw; “. A function can handle a part and can ask the caller to handle remaining.
9) 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>
usingnamespacestd;

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

intmain()
{
    try{
        Test t1;
        throw10;
    }
    catch(inti) {
        cout << "Caught "<< i << endl;
    }
}
```

Output:

```
Constructor of Test

Destructor of Test

Caught 10
```

10) You may like to try Quiz on Exception Handling in C++.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.