Exception Handling

Exception Handling:

- Basics of Exception Handling
- Exception Handling Mechanism: The keyword try, throw and catch

Exception Handling:

Basics of Exception Handling

- Exception handling is a mechanism that separates code that detects and handles exceptional circumstances from the rest of your program. Note that an exceptional circumstance is not necessarily an error.
- Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers.
- 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.
- 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.
- To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a try-block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.
- An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

- 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.

Why Exception Handling?

Following are main advantages of exception handling over traditional error 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.

- **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)

- **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.

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>
using namespace std;
int main()
{
   int x = -1;
    // Some code
   cout << "Before try \n";
   try {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
       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)
```

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>
using namespace std;
int main()
{
   try  {
     throw 10;
   }
   catch (char *excp)  {
      cout << "Caught " << excp;
   }
   catch (...)  {
      cout << "Default Exception\n";
   }
   return 0;
}
```

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(try with multiple catch)

```
#include <iostream>
using namespace std;
int main()
{
   try  {
     throw 'a';
   }
   catch (int x)  {
      cout << "Caught " << x;
   }
   catch (...)  {
      cout << "Default Exception\n";
   }
   return 0;
}
```

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>
using namespace std;
int main()
{
   try  {
      throw 'a';
   }
   catch (int x)  {
       cout << "Caught ";
   }
   return 0;
}
```

OUTPUT:

Terminate called after throwing an instance of 'char'

This application has requested the Runtime to terminate

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

NESTED TRY BLOCK AND RETHROW AN EXCEPTION

```
#include <iostream>
using namespace std;
int main()
{
   try {
      try  {
         throw 20;
      }
      catch (int n) {
          cout << "Handle Partially ";
          throw;   //Re-throwing an exception
      }
   }
   catch (int n) {
       cout << "Handle remaining ";
   }
 return 0;
}
```

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.

6)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

Exception Handling – catching base and derived classes as exceptions:

- 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. For example, following C++ code prints “Caught Base Exception”

Example:

```
#include<iostream>
using namespace std;
 class Base {};
class Derived: public Base {};
int main()
{
  Derived d;
  // some other stuff
  try {
     // Some monitored code
     throw d;
  }
  catch(Base b) {
      cout<<"Caught Base Exception";
  }
  catch(Derived d) {  //This catch block is NEVER executed
      cout<<"Caught Derived Exception";
  }
  return 0;   }
```

In the above C++ code, if we change the order of catch statements then both catch statements become reachable. Following is the modifed program and it prints "Caught Derived Exception"

```
#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";
  }  return 0;  }
```

Exceptions In Constructors And Destructors

A copy constructor is called in exception handling when an exception is thrown fro the try block using the throw statement. The copy constructor mechanism is applied to initialize the temporary object. In addition, destructors are also executed to destroy the object. If an exception is thrown from the constructor, destructors are called only for those objects that are completely constructed.

19.7 Write a program to use exception handling with constructor and destructor.

```
#include<iostream.h>
#include<process.h>
class number
{
float x;
public :
number (float);
```

```
number() {};
~number()
{
cout<<"\n In destructor";
}
void operator ++ (int) // postfix notation
{ x++; }
void operator --() // prefix notation
{
--x; }
void show()
{
cout<<"\n x="<<x;
}
};
```

```
number :: number ( float k)
{
if (k==0)
throw number();
else
x=k;
}
void main()
{
try
{
number N(2.4);
cout<<“\n Before Increasing:”;
N.show();
cout<<“\n After Increasing:”;
N++; // postfix increment
N.show();
```

```
cout<<"\n After Decrementation:";
--N; // prefix decrement
N.show();
number N1(0);
}
catch (number)
{
cout<<"\n invalid number";
exit(1);
}
}
```

OUTPUT

```
Before Increasing:
x=2.4
After Increasing:
x=3.4
After Decrementation:
x=2.4
In destructor
In destructor
invalid number
```

Exception specification

- Older code may contain dynamic exception specifications. They are now deprecated in C++, but still supported. A dynamic exception specification follows the declaration of a function, appending a throw specifier to it. For example:

```
double myfunction (char param) throw (int);
```

- This declares a function called myfunction, which takes one argument of type char and returns a value of type double. If this function throws an exception of some type other than int, the function calls std::unexpected instead of looking for a handler or calling std::terminate.

- If this throw specifier is left empty with no type, this means that std::unexpected is called for any exception. Functions with no throw specifier (regular functions) never call std::unexpected, but follow the normal path of looking for their exception handler.

```
int myfunction (int param) throw(); // all exceptions call unexpected
int myfunction (int param);         // normal exception handling
```

C++ restrict a function to throw an exception.

Example –

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

// only throw ints, chars, and doubles
void f(int val) throw(int, char, double)
{
 if(val==0)
   throw val;
 if(val==1)
   throw 'a';
 if(val==2)
   throw 123.23;
}
int main(){
 try{
  f(0); // also, try passing 1 and 2 to f()
 }
 catch(int i) {
  cout << "Caught an integer\n";
 }
 catch(char c) {
  cout << "Caught char\n";
 }
 catch(double d) {
  cout << "Caught double\n";
 }
 return 0;
}
```

Standard exceptions

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called std::exception and is defined in the <exception> header. This class has a virtual member function called what that returns a null-terminated character sequence (of type char *) and that can be overwritten in derived classes to contain some sort of description of the exception.

```
// using  standard exceptions
#include <iostream>
#include <exception>
using namespace std;
class myexception: public exception
{
  virtual const char* what() const throw()
  {
    return "My exception happened";
  }
} myex;
int main () {
  try
  {
    throw myex;
  }
catch (exception& e)
  {
    cout << e.what() << '\n';
  }
  return 0;
}
```

output:

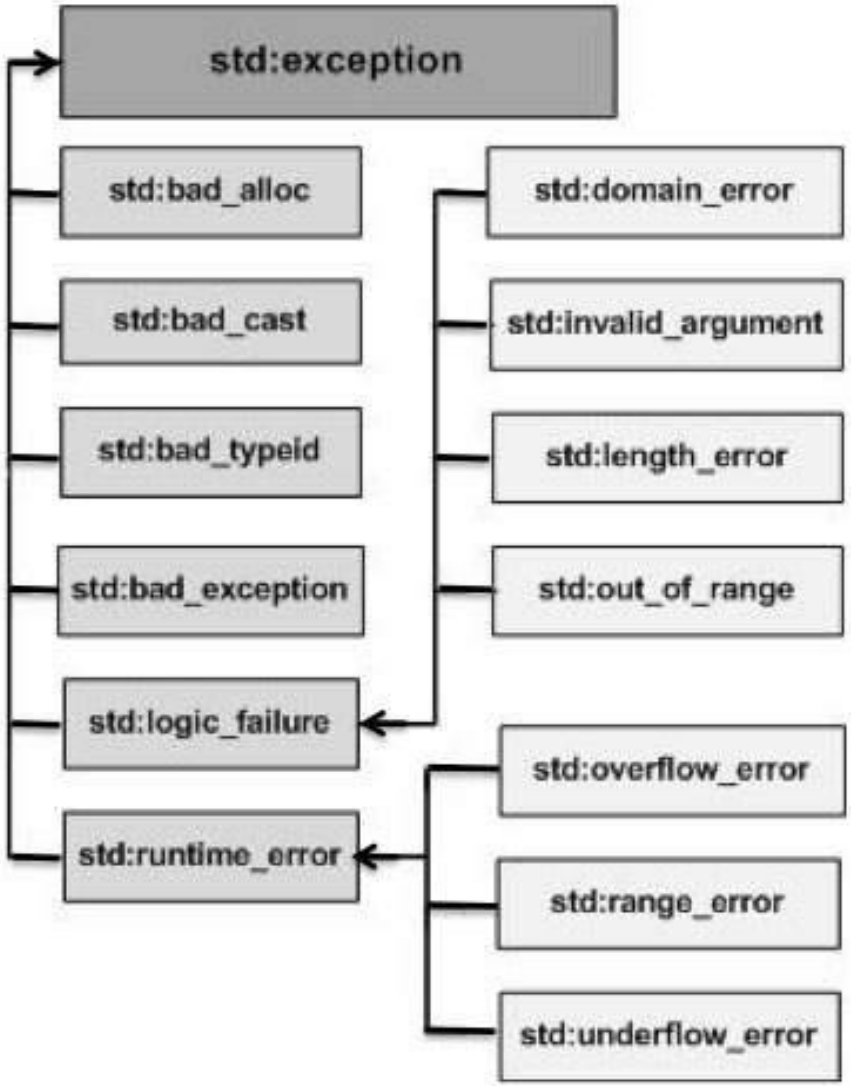
My exception happened.

Explanation:

We have placed a handler that catches exception objects by reference (notice the ampersand & after the type), therefore this catches also classes derived from exception, like our myex object of type myexception)

Standard Exceptions

C++ provides a list of standard exceptions defined in <exception> which we can use in our programs. These are arranged in a parent-child class hierarchy shown belo



std::exception

An exception and parent class of all the standard C++ exceptions.

std::bad_alloc is the type of the object thrown as exceptions by the allocation functions to report failure to allocate storage.

std::bad_cast

An exception of this type is thrown when a dynamic_cast to a reference type fails the run-time check (e.g. because the types are not related by inheritance)

std::bad_exception

This is useful device to handle unexpected exceptions in a C++ program.

std::bad_typeid

An exception of this type is thrown when a typeid operator is applied to a dereferenced null pointer value of a polymorphic type.

std::logic_error

An exception that theoretically can be detected by reading the code.

std::domain_error

This is an exception thrown when a mathematically invalid domain is used.

std::invalid_argument

This is thrown due to invalid arguments.

std::length_error

This is thrown when a too big std::string is created.

std::out_of_range

This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]().

std::runtime_error

An exception that theoretically cannot be detected by reading the code.

std::overflow_error

This is thrown if a mathematical overflow occurs.

std::range_error

This is occurred when you try to store a value which is out of range.

std::underflow_error

This is thrown if a mathematical underflow occurs.

Example:

std::bad_alloc

```
#include <iostream>
#include <new>
 int main()
{
    try {
        while (true) {
            new int[100000000ul];
        }
    } catch (const std::bad_alloc& e) {
        std::cout << "Allocation failed: " <<
e.what() << '\n';
    }
}
```

output:

```
Allocation failed: std::bad_alloc
```