Module-5

Exception Handling: Exception Handling Fundamentals, Catching Class Types, Using Multiple catch Statements, Handling Derived-Class Exceptions, Exception Handling Options, Catching All Exceptions, Restricting Exceptions, Rethrowing an Exception, Understanding terminate( ) and unexpected( ), uncaught_exception( ) Function, The exception and bad_exception Classes, Applying Exception Handling.
STL: Class template, An overview of STL, containers, Vectors, example

Exception:

- An exception is a problem that arises during the execution of a program
- Exceptions are the errors that occur at runtime. There are several reasons for occurrence of exceptions. Few of them.
    - Shortage of memory
    - Inability of open file
    - Exceeding the range of an array
    - Attempt to initialise an impossible value to an object.
- In such situations, the programmer has to take decision about resolving it. The programmer may choose to display an error message on the screen, or he/she may request the user to input better and/or proper data, or simply terminate program.
- The following section discuss the way to handle an exception

5.1 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.

```
try {
     ------------------
     throw val;
     ------------------
}
catch(data_type  arg)
{
     -----------------
}
```

- 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, and the syntax for using try/catch throw as follows

```
try{
      // protected code
      throw exception
}
catch( ExceptionName e1 ) {
   // catch block
}
```

Example:

```
// Exception handling example.

#include<iostream>
using namespace std;
int main(){
     cout<<"start";
     try{
          cout<<"inside try block";
          throw 100;
          cout<<"This will not execute";
}
catch(int i){
     cout<<"caught an exception – value is:";
     cout<<i;
}
cout<<"end";
return 0;
}
```

Output

```
start
inside try block
caught an exception –value is:100
end
```

5.2 Catching class types

- It is observed that catch block can receive any type of exception: basic data types, derived data types etc.
- Similarly, it is possible to have an exception of type class. That is, an object of a class can be thrown as an exception.

```
class test
{
   public:
     test(const char *s){
           cout<<s;
      }
};
int main(){
   int a;
```

```
    try {
       cout<<" enter a number";
       cin>>a;

         if(a<0)
           throw test("Negative");
         cout<<"positive";
        }
     catch(test ob){   }

 }
```

```
Output:
Enter a number: -5
Negative
```

5.3 Using multiple catch statements

- For single try block, there can be multiple `catch` blocks with different types of exception.

```
int main()
{
     int a;
     try{
          cout<<"enter a number";
          cin>>a;
          if(a>0)
               throw "Negative";
          else
               throw a;
     }
     catch(const char *s){
          cout<<s;
     }
     catch(int x){
          cout<<"\n the number is :"<<x;
     }

}
```

```
Output

Enter a number: -8

Negative
```

5.4 Handling derived-class exception

- 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"*

```
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 modified program and it prints "*Caught Derived Exception*"

```
class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   // some other stuff
   try {
       // Some monitored code
       throw d;
   }
   catch(Derived d) {
        cout<<"Caught Derived Exception";
   }
   catch(Base b) {
        cout<<"Caught Base Exception";
   }
   getchar();
   return 0;
}
```

5.5 Exception handling options

- There are several additional features of exception handling that made it easier and more convenient to use.
- These attributes are discussed here.
  - Catching all exception
  - Restricting exceptions
  - Rethrowing an exception

5.5.1 Catching all exceptions

- It is necessary to catch all the exception irrespective of their types. This can be achieved using three dots(. . .) within `catch`.

```
int main(){
  int a;
   try{
         cout<<"enter a number:";
         cin>>a;
          if(a<0)
                 throw "Negative";
           else if ( a==0 )
                  throw 'a';
            else
                   throw a;
}
catch(. . .){
    cout<<"exception caught";
}
```

```
Output:
Enter a number: -8
Exception caught
```

5.5.2 Restricting exceptions

- We can restrict the type of exception to be thrown, from a function to its calling statement, by adding throw keyword to a function definition.
- **The general form**

```
ret-type func_name(arg_list) throw(type_list){
       // ….
}
```

- Here, only those data types contained in the comma-separated *type_list* may be thrown by the function. Throwing any other type of expression will cause abnormal program termination.
- Attempting to throw an exception that is not supported by a function will cause the standard library function `unexcepted()` to be called. By default, this causes `abort()` to be called, which causes abnormal program termination.

- <u>Example:</u>

```
#include<iostream>
using namespace std;
void Xhandler(int test) throw (int, char, double)
{
     if(test==0)    throw test;    //throw int
     if(test==0)    throw 'a';     //throw char
     if(test==0)    throw 123.45;  //throw double
}
int main(){
     cout<<"Start";
     try{
          Xhandler(0);  //also, try passing 1 and 2 to Xhandler( )
     }
     catch(int i){
          cout<<”caught an integer”;
     }
     catch(char c){
          cout<<”caught an integer”;
     }
     catch(double d){
          cout<<”caught an integer”;
     }
     cout<<”end”;
     return 0;
}
```

- In this program, the function `Xhandler()` may only throw integer, character and double exceptions.
- If it attempts to throw any other type of exception, an abnormal program termination will occur.

5.5.3 Rethrowing an exception

- 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.
- Example: illustrates 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
   }
}
int main()
{
   cout<< “Main start”;
```

```
    try
    {
        MyHandler();
    }
    catch(const char*)
    {
       cout <<"Caught exception inside Main\n";
    }
        cout << "Main end";
        return 0;
}
```

Output:

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

5.6 Understanding terminate( ) and unexpected

- `terminate()` and `unexcepted()`are called when something goes wrong during the execution handling process. These functions are supplied by the standard c++ library. Their prototype are shown here.

```
void terminate();

void unexcepted();
```

- These functions require the header <exception>.
- The `terminate()` function is called whenever the exception handling subsystem fails to find a matching `catch` statement for an exception.
- It is also called if your program attempts to rethrow an exception when no exception was originally thrown.
- The `terminate()` function is also called under various other, more obscure circumstances. By default, `terminate()` calls `abort()`.
- The `unexcepted()` function is called when a function attempts to throw an exception that is not allowed by its throw list. By default, `unexcepted()` calls `terminate()`.

Setting the terminate and unexcepted Handlers

- The `terminate`() and unexcepted() functions simply call other functions to actually handle an error.
- As explained, by default, `terminate`() calls `abort`(), and `unexcepted`() calls `terminate`() . Thus, by default both functions halt the program execution when an exception handling error occurs.

- However, you can change the functions that are called by terminate and unexcepted. Doing so allows your program to take full control of the exception handling subsystem.
- Syntax:

```
terminate_handler set_terminate()terminate_handler
newhandler) throw();
```

- Here, *newhandler* is a pointer to the new terminate handler. The function returns a pointer to the old terminate handler. The new terminate handler must be of type `terminate_handler,` which is defined like this:

```
typedef void (*terminate_handler) ( );
```

- The only thing that your terminate handler must do is to stop program execution. It must not return to the program or resume it in any way.

- To change the unexepected handler, use set_unexecpected( ), shown here:

```
unexpected_handler set_unexpected(unexpected_handler
newhandler) throw();
```

- Here, *newhandler* is a pointer to the new unexpected handler. The function returns a pointer to the old unexpected handler. The new unexpected handler must be of type `unexpected_handler,` which is defined like this:

```
typedef void (*unexpected _handler) ( );
```

- This handler may itself throw an exception, stop the program, or call *terminate*(). However it must not return to the program.
- Both set_terminate( ) and set_unexpected() require the header <exception>.
- Here is an example that defines its own terminate( ) handler.

```
#include<exception.h>
void my_handler(){
     cout<<"inside new terminate handler";
     abort();
}
int main(){
     set_terminate(my_handler);
     try{
          cout<<"inside try block".
          throw 100;
     }
catch(double d){ //won't catch an int exception
     //…
     }
return 0;
}
```

```
Output:
inside try block
inside new terminate handler
abnormal program termination
```

5.7 uncaught_exception( ) function

- **The c++** exception handling subsystem supplies one other function that you may find useful:uncaught_exception( ). Its prototype is shown here:

```
bool uncaught_exception( );
```

- This function returns **true** if an exception has been thrown but not yet caught. Once caught, the function returns **false**.

5.8 The exception and bad_exception Classes

- When a function supplied by the c++ standard library throws an exception, it will be an object derived from the base class **exception**.
- An object of the class **bad_exception** can be thrown by unexcepted handler. These classes require the header **<exception>**.

5.9 Applying exception handling

- **Exception** handling is designed to provide a structed means by which your program can handle abnormal events. This implies that the error handler must do something rational when an error occurs.
- **Example:**

```
void divide(double a, double b);
int main(){
     double i,j;
     do{
          cout<<"Enter numerator (0 to stop)";
          cin>> i;
          cout<<"enter denominator: ";
          cin>> j;
          divide(i,j);
     }while(i!=0);
return 0;
}
void divide(double a, double b){
     try{
          if(!b) throw b;
          cout<<"result: "<<a/b;
     }
     catch(double b){
          cout<<"cant divide by zero";
     }
}
```

- While the preceding program is a very simple example, it does illustrate the essential nature of exception handling. Since division by zero is illegal, the program cannot continue if a zero is entered for second number. In this case, the exception is handled by not performing

5.10 Standard Template (STL)

5.10.1 class template

- The concept of templates is applicable even to classes. When a class uses generalized logic irrespective of the data type, then *generic/template* class is used.
- For example, to create a stack, the operations to be implemented are push and pop. The data type may be integer, float or character, the logic of push and pop functions remain same. In such situations, instead of creating separate class for each data type, the programmer can go for generic classes.
- In a generic class/template class, one has to define all the variables and functions used by that class. But the actual type of data to be operated will be specified as a parameter when objects of that class are created.
- The syntax of generic class declaration is-

```
template <class t_name>class class_name
{
     //body of generic class
};
```

- Once the generic class has been created, an object of that class can be created using the statement

```
class_name <type> obj;
```

- Here, the *type* specifies a particular type of data upon which the class will be operating

```
Template <class T>
class weight{
     private:
          T kg;
     public:
          void setData(T x)
          {
               kg=x;
          }
          T getData(){
               return kg;
          }
};
int main(){
     weight <int> obj1; //object with integer datatype
     obj1.setData(5);
     cout<<"value is"<<obj1.getData();
     weight <double> obj2; //object with double datatype

     obj2.setData(5.3345);
     cout<<"value is"<<obj2.getData();
}

Output:
Value is 5
Value is 5.3345
```

5.10.2 An overview of STL

- Standard Template Library (STL) is a collection of general-purpose classes and functions for commonly used algorithms and data structures like stacks, queue, lists, vectors etc.
- STL consists of function templates and class templates for several algorithms and hence they can be used for merely any type of data.
- In STL, there are three fundamental items viz. (containers, algorithms, iterators)

    - **Containers**: The objects that hold other objects are known as container classes.
        - For example, the vector class defines a dynamic array, deque creates a double ended queue, and list provides linear list.
        - There are two types of containers:
            - ***Sequence Container***: like list, vectors
            - ***Associative container***: which allow efficient retrieval of values-based on keys. Example: map

    - **Algorithms:** Algorithms act on containers and they provide the means through which the content of container can be manipulated.

    - **Iterators:** these are the objects similar to pointers. They give the ability to cycle through the contents of a container just similar to using a pointer to cycle an array.

5.10.3 Container class

- The containers are the STL objects that actually store data.
- Also, headers are necessary to use each container.
- The string class, which manages character strings, is also a container.
- The containers defined by STL are shown in below table.

Container	Description	Required header
bitset	A set of bits	<bitset>
deque	A double ended queue	<dequeu>

list	A linear list	<list>
map	Stores key/value pairs in which each key is associated with two or more value.	<map>
set	A set in which each element is unique	<set>
stack	A stack	<stack>
vector	A dynamic array	<vector>

5.10.4 vectors

- Vector is the most general-purpose container. It is dynamic array that can grow or shrink as needed.
- The elements of a vector can be accessed randomly but insertion and deletion are possible only at one end. So, it is a sequential and unidirectional.
- To use the STL container vector, a header file vector.h should be included in the program.
- Few of the member functions defined in vector class are given below in

Member Function	**Description**
begin()	Returns an iterator to the first element of the vector
clear()	Removes all the elements from the vector
empty()	Returns true, if the vector is empty, otherwise false
pop_back()	Deletes last element from the vector
push_back()	Inserts an element at the end of the vector
size()	Returns number of elements in the vector

5.10.5 Example

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

     int main()
     {
              vector<int> v;
              int n, element;

        cout << "Size of Vector=" << v.size() << endl;
        cout << "Enter No of elements to add :" << endl;
        cin>>n;

      cout << "Enter "<< n <<" Elements :" << endl;
         for(int i=0; i<n; i++)
           {
              cin >> element;
              v.push_back(element);
            }

        //Size after adding values
      cout << "Size of Vector=" << v.size() << endl;

        //Display the contents of vector
        for(int i=0; i<v.size(); i++)
         {
            cout << v[i] << " ";
         }
     return 0;
     }
```

Output

```
Size of Vector=0
Enter No of elemnts to add :   3

Enter 3 Elements :
34
55
77
Size of Vector=3
```