Unit 5 Exception Handling and Templates

Exception Handling- Fundamentals, other error handling techniques, simple exception handling- Divide by Zero, Multiple catching, re-throwing an exception, exception specifications, user defined exceptions, processing unexpected exceptions, constructor, destructor and exception handling, exception and inheritance.

Templates- The Power of Templates, Function template, overloading Function templates, and class template, class template and Non-type parameters, template and friends Generic Functions, The type name and export keywords.

5.1 Exception Handling- Fundamentals

An exception is a problem that arises during the execution of a program. 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**.

1. **throw** − A program throws an exception when a problem shows up. This is done using a **throw** keyword.

2. **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.

3. **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.

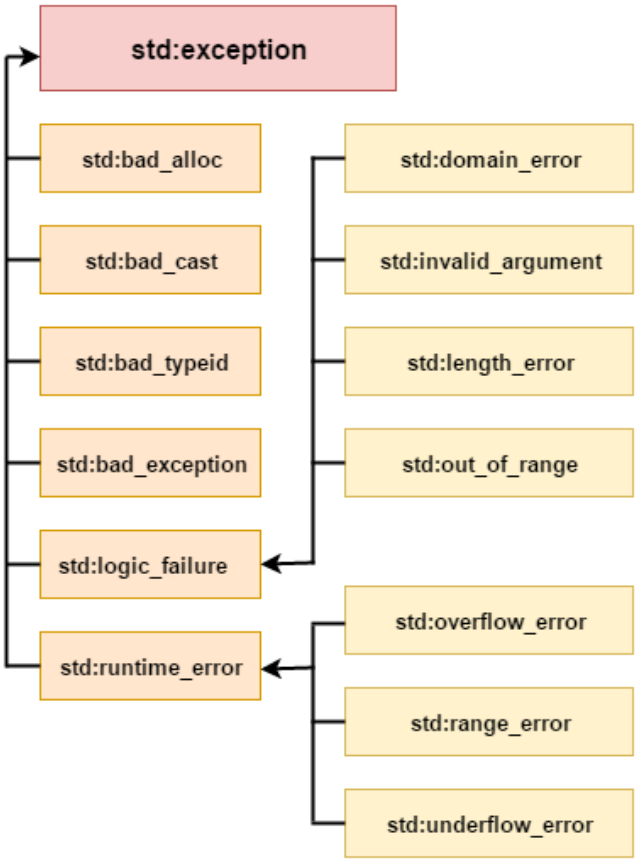
Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows −

```
try {
// protected code
} catch( ExceptionName e1 ) {
// catch block
} catch( ExceptionName e2 ) {
// catch block
} catch( ExceptionName eN ) {
// catch block
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

5.2 Other error handling techniques

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.**

5.3 Simple exception handling-Divide by Zero

Example

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

double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x = 50;
   int y = 0;
   double z = 0;

   try {
      z = division(x, y);
      cout << z << endl;
   } catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

Output

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block. If we compile and run code, this would produce the following result −

```
Division by zero condition!
```

5.4 Multiple Catching

Example

```
#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
```

5.5 Re-throwing an exception

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

Example

```
#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.

5.6 Exception Specifications

Exception Specification

Exception specification is a feature which specifies the compiler about possible exception to be thrown by a function.

Syntax

```
return_type FunctionName(arg_list) throw (type_list)
```

In the above syntax 'return_type' represents the type of the function. FunctionName represents the name of the function, arg_list indicates list of arguments passed to the function, throw is a keyword used to throw an exception by function and type_list indicates list of exception types. An empty exception specification can indicates that a function cannot throw any exception. if a function throws an exception which is not listed in exception specification then the unexpected function is called. This function terminates the program. In the function declaration, if the function does not have any exception specification then it can throw any exception.

Example

1. void verify(int p) throw(int)

In the above example, void is the return type, and verify() is the name of the function, int p is the argument. It is followed by exception specification i.e., throw(int), the function verify() is throws an exception of type 'int'.

2. void verify(int p) throw()

Here, the function can throw any exception because it has empty exception specification.

Example

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

void verify(int p) throw(int)
{
if(p==1)
throw p;
cout<<"\nEnd of verify().function( )";
}

int main()
{
try
{
cout<<p==1\n";
verify(1);
}
catch(int i)
{
cout<<"Interger exception is caught \n";
}
cout<<"\nEnd of main() function";

return 0;
}
```

Output

```
p==1
Interger exception is caught
```

5.7 User defined exceptions

You can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way.

Example

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

struct MyException : public exception {
   const char * what () const throw () {
```

```
      return "C++ Exception";
   }
};

int main() {
   try {
      throw MyException();
   } catch(MyException& e) {
      std::cout << "MyException caught" << std::endl;
      std::cout << e.what() << std::endl;
   } catch(std::exception& e) {
      //Other errors
   }
}
```

This would produce the following result −

```
MyException caught
C++ Exception
```

5.8 Processing unexpected exceptions

If an exception is thrown and no exception handler is found—that is, the exception is not caught—the program calls a termination function. You can specify your own termination function with set_terminate. If you do not specify a termination function, the terminate function is called. For example, the following code uses the my_terminate function to handle exceptions not caught by any handler.

```
void SetFieldValue(DF *dataField, int userValue) {
    if ((userValue < 0) || (userValue) > 10))
        throw EIntegerRange(0, 10, userValue);
    // ...
}

void my_terminate() {
    printf("Exception not caught");
    abort();
}

// Set my_terminate() as the termination function
set_terminate(my_terminate);

// Call SetFieldValue. This generates an exception because the user value is
greater
// than 10. Because the call is not in a try block, my_terminate is called.
SetFieldValue(DF, 11);
```

If a function specifies which exceptions it throws and it throws an unspecified exception, an unexpected function is called. You can specify your own unexpected function with set_unexpected. If you do not specify an unexpected function, the unexpected function is called.

5.9 constructor, destructor and exception handling

An exception is termed as an unwanted error that arises during the runtime of the program. The practice of separating the anomaly-causing program/code from the rest of the program/code is known as Exception Handling.

An object is termed as an instance of the class which has the same name as that of the class. A destructor is a member function of a class that has the same name as that of the class but is preceded by a '~' (tilde) sign, also it is automatically called after the scope of the code runs out. The practice of pulverizing or demolition of the existing object memory is termed object destruction. In other words, the class of the program never holds any kind of memory or storage, it is the object which holds the memory or storage and to deallocate/destroy the memory of created object we use destructors.

Example

```
// C++ Program to show the sequence of calling
// Constructors and destructors
#include <iostream>
using namespace std;

// Initialization of class
class Test {
public:
    // Constructor of class
    Test()
    {
        cout << "Constructing an object of class Test "
            << endl;
    }

    // Destructor of class
    ~Test()
    {
        cout << "Destructing the object of class Test "
            << endl;
    }
};
```

```
int main()
{
    try {
        // Calling the constructor
        Test t1;
        throw 10;

    } // Destructor is being called here
    // Before the 'catch' statement
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}
```

Output

```
Constructing an object of class Test
Destructing the object of class Test
Caught 10
```

When an exception is thrown, destructors of the objects (whose scope ends with the try block) are automatically called before the catch block gets executed. That is why the above program prints “Destructing an object of Test” before “Caught 10“.

5.10 Exception and inheritance.

Let’s discuss what is Exception Handling and how we catch base and derived classes as an exception in C++:

1. If both base and derived classes are caught as exceptions, then the catch block of the derived class must appear before the base class.

2. If we put the base class first then the derived class catch block will never be reached. For example, the following C++ code prints “Caught Base Exception “

Example

```
// C++ Program to demonstrate a
// Catching Base Exception
#include <iostream>
using namespace std;

class Base {
```

```
};
class Derived : public Base {
};
int main()
{
    Derived d;
    // Some other functionalities
    try {
        // Monitored code
        throw d;
    }
    catch (Base b) {
        cout << "Caught Base Exception";
    }
    catch (Derived d) {
        // This 'catch' block is NEVER executed
        cout << "Caught Derived Exception";
    }
    getchar();
    return 0;
}
```

Output

```
prog.cpp: In function ‘int main()’:
prog.cpp:20:5: warning: exception of type ‘Derived’ will be caught
    catch (Derived d) {
    ^
prog.cpp:17:5: warning:    by earlier handler for ‘Base’
    catch (Base b) {
```

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”

Example

```
// C++ Program to demonstrate a catching of
// Derived exception and printing it successfully
#include <iostream>
using namespace std;

class Base {};
class Derived : public Base {};
int main()
```

```
{
    Derived d;
    // Some other functionalities
    try {
        // Monitored code
        throw d;
    }
    catch (Derived d) {
        cout << "Caught Derived Exception";
    }
    catch (Base b) {
        cout << "Caught Base Exception";
    }
    getchar(); // To read the next character
    return 0;
}
```

Output

```
Caught Derived Exception
```

5. 11 Templates- The Power of Templates

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**

5.12 Function template

The general form of a template function definition is shown here −

```
template <class type> ret-type func-name(parameter list) {
// body of function
}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition. The following is the example of a function template that returns the maximum of two values

Example

```
#include <iostream>
#include <string>

using namespace std;

template <typename T>
inline T const& Max (T const& a, T const& b) {
   return a < b ? b:a;
}

int main () {
   int i = 39;
   int j = 20;
   cout << "Max(i, j): " << Max(i, j) << endl;

   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;

   string s1 = "Hello";
   string s2 = "World";
   cout << "Max(s1, s2): " << Max(s1, s2) << endl;

   return 0;
}
```

Output

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

5.13 Overloading Function templates

Function Template: The function template has the same syntax as a regular function, but it starts with a keyword template followed by template parameters enclosed inside angular brackets <>

```
template <class T>
T functionName(T arguments)
{
     // Function definition
     …. …. …. …
}
```

where, T is template argument accepting different arguments and class is a keyword.

Template Function Overloading:

1. The name of the function templates are the same but called with different arguments is known as function template overloading.
2. If the function template is with the ordinary template, the name of the function remains the same but the number of parameters differs.
3. When a function template is overloaded with a non-template function, the function name remains the same but the function's arguments are unlike.

Example

```
// C++ program to illustrate overloading of template function using an
// explicit function
#include <bits/stdc++.h>
using namespace std;

// Template declaration
template <class T>

// Template overloading of function
void display(T t1)
{
    cout << "Displaying Template: "
        << t1 << "\n";
}

// Template overloading of function
void display(int t1)
{
    cout << "Explicitly display: "
        << t1 << "\n";
}

// Driver Code
int main()
{
    // Function Call with a
    // different arguments
    display(200);
    display(12.40);
```

```
    display('G');

    return 0;
}
```

Output

```
Explicitly display: 200
Displaying Template: 12.4
Displaying Template: G
```

5.14 Class template

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here −

```
template <class type> class class-name {
.
.
.
}
```

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Example

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>
using namespace std;

template <class T>
class Stack {
   private:
      vector<T> elems;     // elements

   public:
      void push(T const&);  // push element
      void pop();               // pop element
      T top() const;            // return top element

      bool empty() const {      // return true if empty.
         return elems.empty();
```

```
      }
};

template <class T>
void Stack<T>::push (T const& elem) { // append copy of passed element
   elems.push_back(elem);
}

template <class T>
void Stack<T>::pop () {
   if (elems.empty()) {
      throw out_of_range("Stack<>::pop(): empty stack");
   }

    elems.pop_back();        // remove last element
}

template <class T>
T Stack<T>::top () const {
   if (elems.empty()) {
      throw out_of_range("Stack<>::top(): empty stack");
   }
      return elems.back();         // return copy of last element
}

int main() {
   try {
      Stack<int>         intStack;  // stack of ints
      Stack<string> stringStack;    // stack of strings

      // manipulate int stack
      intStack.push(7);
      cout << intStack.top() <<endl;

      // manipulate string stack
      stringStack.push("hello");
      cout << stringStack.top() << std::endl;
      stringStack.pop();
      stringStack.pop();
   } catch (exception const& ex) {
      cerr << "Exception: " << ex.what() <<endl;
      return -1;
   }
}
```

Output

```
7
hello
Exception: Stack<>::pop(): empty stack
```

5.15 class template and Non-type parameters

It is also possible to use non-type arguments (basic/derived data types) i.e., in addition to the type argument T, it can also use other arguments such as strings, function names, constant expressions, and built-in data types.

```
template <class T, int size>
class Array {
private:

    // Automatic array initialization
    T Arr[size]
    .....
    .....
};
```

In the above example, the template supplies the size of the array as an argument. This implies that the size of the array is known to the compiler at the compile time itself. The arguments must be specified whenever a template class is created.

Example

```
// C++ program to implement bubble sort
// by using Non-type as function parameters
#include <iostream>
using namespace std;

// Function to swap two numbers
template <class T>
void swap_(T* x, T* y)
{
    T temp = *x;
    *x = *y;
    *y = temp;
}

// Function to implement the Bubble Sort
template <class T, int size>
void bubble_sort(T arr[])
{
    for (int i = 0; i < size - 1; i++) {
```

```
        // Last i elements are already
        // in place
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {

                // Swap operation
                swap_(&arr[j], &arr[j + 1]);
            }
        }
    }
}

// Function to print an array
template <class T, int size>
void printArray(T arr[])
{
    int i;
    for (i = 0; i < size - 1; i++) {
        cout << arr[i] << ", ";
    }

    cout << arr[size - 1] << endl;
}

// Driver Code
int main()
{
    // Given array arr[]
    float arr[] = { 1.1, 1.2, 0.3, 4.55, 1.56, 0.6 };
    const int size_arr = sizeof(arr) / sizeof(arr[0]);

    // Size of the array passed as
    // an argument to the function
    bubble_sort<float, size_arr>(arr);

    cout << "Sorted Array is: ";
    printArray<float, size_arr>(arr);

    return 0;
}
```

Output

```
Sorted Array is: 0.3, 0.6, 1.1, 1.2, 1.56, 4.55
```

5.16 Template and friends Generic Functions

Generics is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces. For example, classes like an array, map, etc, which can be used using generics very efficiently. We can use them for any type.

The method of Generic Programming is implemented to increase the efficiency of the code. Generic Programming enables the programmer to write a general algorithm which will work with all data types. It eliminates the need to create different algorithms if the data type is an integer, string or a character.

The advantages of Generic Programming are

1. Code Reusability
2. Avoid Function Overloading
3. Once written it can be used for multiple times and cases.

Generics can be implemented in C++ using Templates. Template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

Example

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

// One function works for all data types.
// This would work even for user defined types, if operator '>' is overloaded
template <typename T>

T myMax(T x, T y)
{
    return (x > y) ? x : y;
}

int main()
{
    cout << myMax<int>(3, 7) << endl; // Call myMax for int
    // call myMax for double
    cout << myMax<double>(3.0, 7.0) << endl;

    // call myMax for char
    cout << myMax<char>('g', 'e') << endl;
```

```
    return 0;
}
```

Output

```
7
7
g
```

5.17 type name: Keyword

1. In the template parameter list of a template declaration, typename can be used as an alternative to class to declare type template parameters and template template parameters (since C++17).
2. Inside a declaration or a definition of a template, typename can be used to declare that a dependent qualified name is a type.
3. Inside a declaration or a definition of a template, (until C++11) typename can be used before a non-dependent qualified type name. It has no effect in this case.
4. Inside a requirements for type requirements (since C++20)

5.18 export: Keyword

Used to mark a template definition exported, which allows the same template to be declared, but not defined, in other translation units.	(until C++11)
The keyword is unused and reserved.	(since C++11) (until C++20)
Marks a declaration, a group of declarations, or another module as exported by the current module.	(since C++20)