MODULE 5

Templates and Exception Handling

SYLLABUS:
Templates–Uses, Generic classes, Class templates, Function templates, Advance templates.
Introduction to Exception - Benefits of Exception handling- Try and catch Block, Throw statement- pre-defined exceptions in C++.

1.1 INTRODUCTION

Templates in C++ allow the creation of **generic code** that can work with any data type. Instead of writing separate functions or classes for different data types, a single **template** can handle all types. This reduces code duplication and enhances maintainability.

- **Uses of Templates**: Templates are used for creating **generic functions** and **generic classes** that work with any data type. They are heavily used in libraries like the Standard Template Library (STL).
- **Generic Classes and Class Templates**: These allow defining classes where data types are specified as parameters. For example, a class template for a stack can work with int, float, or even user-defined types.
- **Function Templates**: These are used to create functions that can operate on different data types without rewriting code for each type.
- **Advanced Templates**: C++ also supports advanced features like **template specialization**, **template metaprogramming**, and **variadic templates** for more complex scenarios.

Exception handling in C++ is used to manage **run-time errors** gracefully, allowing the program to recover or terminate properly without crashing.

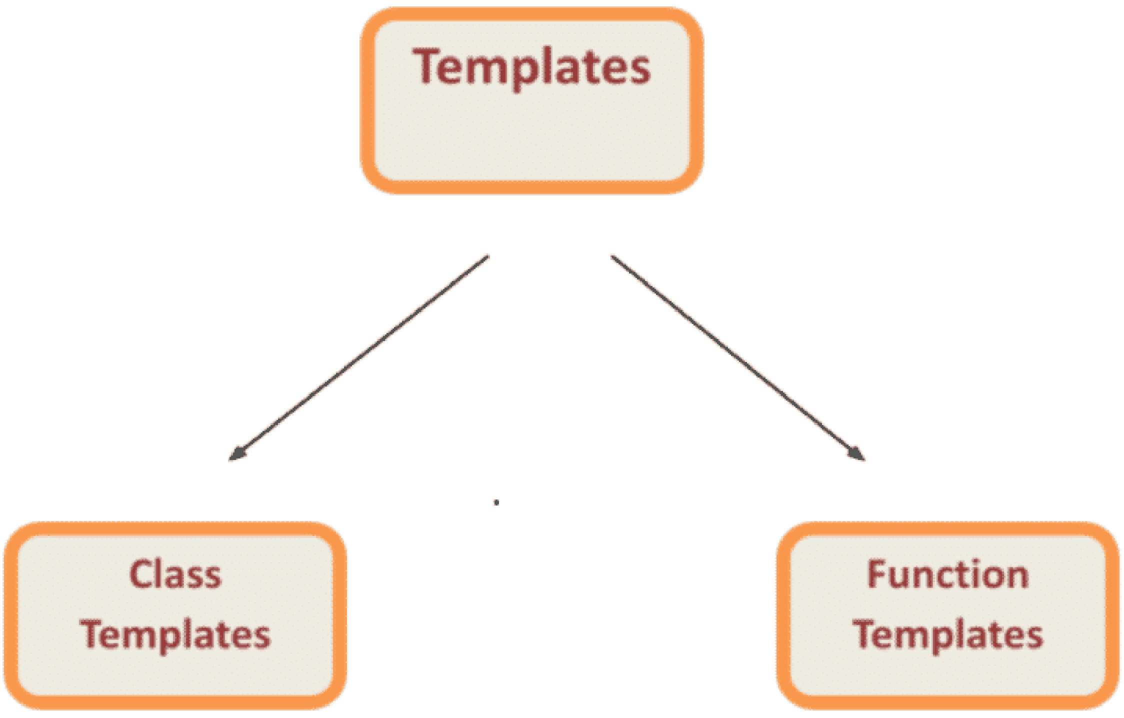
- **Introduction to Exception**: Exceptions are events that disrupt the normal flow of a program. C++ provides a mechanism to throw and catch exceptions, ensuring the program can handle errors appropriately.
- **Benefits of Exception Handling**: It improves program stability, separates error-handling code

from regular logic, and makes the code more readable and reliable.

- **Try and Catch Block**: The try block contains code that might throw an exception, and the catch block handles the exception.
- **Throw Statement**: The throw keyword is used to signal that an exception has occurred.
- **Pre-defined Exceptions in C++**: C++ provides built-in exception classes such as std::exception, std::runtime_error, and std::out_of_range, which help handle common error scenarios.

Together, **templates** and **exception handling** make C++ a flexible and powerful language for developing high-performance applications.

The templates in C++ are used to write the **data type independent** piece of code. The specified placeholder in the code gets replaced by the actual data type at the time of compilation, called the instantiation of code. The class or function written as the template is called Generics, and this entire concept is called Generic Programming in C++.



Introduction to Generic Programming in C++

The generic programming pattern generalizes the algorithm with the help of templates in C++ so that it can be used along with different data types. In templates, we specify a placeholder instead of the actual data type, and that placeholder gets replaced with the data type used during the compilation. So, if the template function were being called for integer, character, and float, the compiler would produce 3 copies of the function. This is possible because of the static-type nature of C++.

Use of Templates

To understand the use of templates, let's briefly discuss normal programming. In the C++ programming language, all data must be stored in a container. Technically, these are termed as data types like int, float, user-defined, etc.

As we know, the algorithm could be the same for different types of data, i.e., the procedure to find the distance between two coordinates will remain the same irrespective of whether the coordinates are given as integer or floating-point numbers. But in C++ Programming, we must write different functions for both data types because the int data type is incapable of storing float values, and using the float data type for int values is a waste of memory. So, **Templates** in C++ solve this problem by providing a generalized algorithm.

Real-Life Usage of Templates in C++

1. Standard Template Library (STL)

- STL containers like vector, map, stack, queue, and list are all built using templates.
- You can use them with **any data type** without rewriting the code.

```
std::vector<int> v1;     // vector of int
std::vector<std::string> v2; // vector of string
```

2. Generic Sorting Functions

- You can write a single sort() function that works for **integers, floats, or strings** using templates.

```
template <typename T>
void sortArray(T arr[], int n) {
```

```
// simple bubble sort
for (int i = 0; i < n-1; ++i)
for (int j = 0; j < n-i-1; ++j)
if (arr[j] > arr[j+1])
std::swap(arr[j], arr[j+1]);
}
```

3. Smart Pointers in Modern C++

- Templates are used to create smart pointers like std::shared_ptr<T>, std::unique_ptr<T>.
- These manage memory safely for any type.

```
std::shared_ptr<int> ptr = std::make_shared<int>(10);
```

4. Generic Stack or Queue Implementations

- You can create one Stack<T> class and use it for different types: Stack<int>, Stack<string>, etc.

```
template <class T>
class Stack {
T arr[100];
int top;
public:
Stack() { top = -1; }
void push(T val) { arr[++top] = val; }
T pop() { return arr[top--]; }
};
```

5. Algorithms Library

- Functions like std::max(), std::min(), std::swap() use templates internally so they work with any comparable types.

```
int a = 5, b = 10;
std::swap(a, b);
```

6. Type-Safe Code Reuse

- Templates allow writing **type-independent logic** (like search, sort, or compare) once and reusing it across many types.

7. Custom Generic Utilities

- You can write custom generic functions or classes — like Matrix<T>, Logger<T>, or ConfigLoader<T>.

8. Testing Code with Mock Data Types

- In unit testing, you can write generic mock functions or classes using templates to test different data types or modules.

1.2 Function Template

The function templates are used to write functions with generic types that can adapt functionality according to the data type used during the function call. This makes us easier to perform the same operation on different data types without code replication.

Example: Finding Maximum of two Numbers

```
#include <iostream>


// Template Function with a Type T
// During instantiation, this T will be replaced by the data type of argument.
template <class T>
T maxNum (T a, T b) {
return (a > b ? a : b);
}
```

```
int main()
{
int x = 5, y = 2;
float a = 4.5, b = 1.3;

std::cout << maxNum<int>(x, y) << "\n";
std::cout << maxNum<float>(a, b);
return 0;
}
```

Output:

```
5
4.5
```

Note: In the above example, we have specified the data type with the function call itself, but this could be skipped as the compiler automatically detects the values we give to the function.

Guidelines for Using Template Functions

This section will discuss Syntax, general guidelines, and a few important things related to template functions.

Generic Data Types

Generic types are classes or functions that are parameterized over a type. This is done with templates in C++. The template parameters are used to create a generic data type. This concept is similar to function parameters, we pass some template arguments, and the function receives it as a type value.

The **syntax** to specify the **template parameters** is,

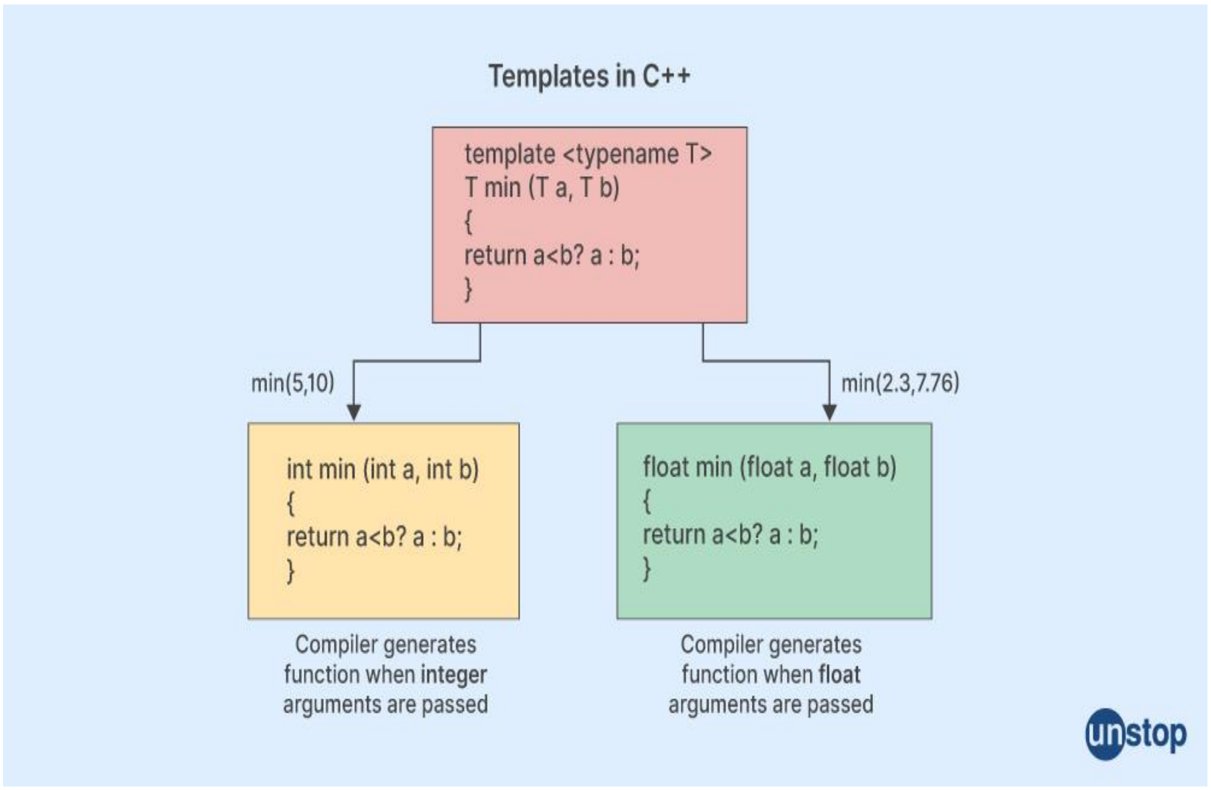
```
template <class identifier> function_declaration;


// or

```

```
template <typename identifier> function_declaration;
```

Either we use the class keyword or typename. Both approaches are the same.

Declaration and Definition Must Be in the Same File

The templates are not normal functions. They only get compiled when some instantiation with particular template arguments. The compiler generates the exact functionality with the provided arguments and template. Because templates are compiled when required, it is impossible to separate the definition and declaration of the template functions into two files.



Overloading Function Templates

The template functions can be overloaded. The compiler will search for the exact overloaded function definition if any particular function call is encountered inside the program. If found, then the overloaded function will execute. Otherwise, the matched template function will execute. Further, if no template function is associated with the function call, the compiler will throw an error.

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

```
template<class T>
void sum(T a, T b){
cout << "Inside Template " << a + b << endl;
}

void sum(int a, int b){
cout << "Inside Overload " << a + b << endl;
}

int main()
{
// Template Function will be called.
sum(4.5,9.8);

// Overload will be called.
sum(5,7);

return 0;
}
```

Output:

```
Inside Template 14.3
Inside Overload 12
```

Explanation:

- For the first function, call sum(4.5,9.8); the compiler will search for the exact function that doesn't exist and then search for the template function, as the template exists, so that it will be executed.

- For another function call, sum(5,7); there already exists an overload function for the exact data type. Hence template function will not be called.

Function template overloading means defining **multiple versions** of a **template function** with:

- different **number of parameters**
- or different **types of parameters**
- or a **non-template version** of the function

This allows more flexibility and control.

Example 1: Overloading by Number of Parameters

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

template <typename T>
void display(T a) {
cout << "One parameter: " << a << endl;
}

template <typename T>
void display(T a, T b) {
cout << "Two parameters: " << a << " and " << b << endl;
}

int main() {
display(10);          // Calls first version
display(10, 20);      // Calls second version
return 0;
}
```

Example 2: Template + Non-Template Overload

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

template <typename T>
void show(T a) {
cout << "Template version: " << a << endl;
}

void show(int a) {
cout << "Non-template version: " << a << " (int only)" << endl;
}

int main() {
show(5);      // Calls non-template version
show(5.5);    // Calls template version
return 0;
}
```

If an **exact match** is found (like int with a non-template), it takes priority over the template.

Example 3: Overloading with Different Data Types

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

template <typename T>
void print(T a) {
cout << "Single type: " << a << endl;
}

template <typename T, typename U>
```

```
void print(T a, U b) {
cout << "Mixed types: " << a << " and " << b << endl;
}

int main() {
print(100);            // Single type
print(100, "hello");     // Mixed types
return 0;
}
```

Recursive Template Functions

The recursion can be achieved with the template functions, and from the program execution's point of view, everything works the same as the normal recursive function.

Function Templates with User Defined Types

We are free to choose any type that we are going to pass inside the template parameters of the function template. The inference is that we can also create function templates with **user-defined** types.

```
#include <iostream>
using namespace std;
class Person
{
private:
int age;
string name;
public:
Person(string _name, int _age)
{
age = _age;
name = _name;
}
void toString()
{
```

```
cout << name << " is " << age << " years old."<< endl;
}
};

template < class T >
void printTheData(T &obj)
{
obj.toString();
}

int main()
{
Person p1 = Person("Tommy Vercetti", 21);

printTheData(p1);

return 0;
}
```

Output:

```
Tommy Vercetti is 21 years old.
```

Explanation:

- We have created a user-defined type as Person, which has a toString method and constructor along with some private entities.
- Subsequently, there is a function template to print the data by calling that function with the corresponding user-defined data type.
- When the compiler parses the call printTheData(p1); a copy of the template function with the specific data type will be created, and this call will be calling that copy of the template function where Person replaces T.

Class Template

Like function templates, we can also use templates with the class to make it compatible with more than one data type. In C++ programming, you might have used the vector to create a dynamic array, and you can notice that it works fine with every data type you pass inside the <>, ex- vector<int>. This is just because of the class template.

Syntax:

```
template <class T>
class className {
// Class Definition.
// We can use T as a type inside this class.
};
```

Class Template and Friend Functions

A non-member function that is defined outside the class and can access the private and protected members of the class is called a friend function. These are used to create a link between classes and functions. We can define our friend function as a template or a normal function in the class template.

A **class template** allows you to create a generic class that works with **any data type**.

Syntax:

```
template <class T>
class ClassName {
// members using type T
};
```

A **friend function** is a function that is **not a member** of the class but has **access to its private and protected members**.

Using Friend Functions with Class Templates

You can declare a friend function **inside a class template** so it can access private members of **templated objects**.

Example: Class Template with Friend Function

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

template <class T>
class Box {
T value;

public:
Box(T v) : value(v) {}

// Friend function declaration inside the template
friend void showValue(Box<T> b) {
cout << "Value: " << b.value << endl;
}
};

int main() {
Box<int> b1(100);
Box<double> b2(55.5);

showValue(b1);  // Output: Value: 100
showValue(b2);  // Output: Value: 55.5

return 0;
}
```

1.3 Class Templates and Static Variables

The classes in C++ can contain two types of variables, static and non-static(instance). Each object of the class consists of non-static variables. But the static variable remains the same for each object means it is shared among all the created objects. As we are discussing the templates in C++, a point worth noticing is that the static variable in template classes remains shared among all objects of the **same type**.

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

template < class T >
class Container {
private:
T data;
public:
static int count;
Container() {
count++;
}

static void displayStaticVariable() {
cout << count << endl;
}
};

template < class T >
int Container < T > ::count = 0;

int main() {
Container < int > obj1;
Container < float > obj2;
Container < int > obj3;
Container < int > ::displayStaticVariable();
Container < float > ::displayStaticVariable();
return 0;
}
```

Output:

```
2
1
```

Explanation:

- We have created a template class named Container which contains static member count.
- We could see from the above example that for different data types, static variables have different values, which shows every type has a separate copy of the static variable.

Class templates allow you to create a **generic class** that works with **any data type** (int, float, string, etc.).

```
template <class T>
class MyClass {
T data;
};
```

What Is a Static Variable in a Class?

A **static variable** in a class:

- Is **shared among all objects** of that class
- Is **initialized only once**
- Retains its value between function calls

Static Variables in Class Templates

In a class template, **each data type (T)** has its **own copy** of the static variable.

So:

- MyClass<int>::count is separate from
- MyClass<float>::count

Example: Class Template with Static Variable

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

template <class T>
class Counter {
private:
static int count;  // static member

public:
Counter() {
count++;
}

static void showCount() {
cout << "Count for type " << typeid(T).name() << ": " << count << endl;
}
};

// Static member definition (outside the class)
template <class T>
int Counter<T>::count = 0;

int main() {
Counter<int> c1, c2;
Counter<float> f1;

Counter<int>::showCount();    // Output: Count for type i: 2
Counter<float>::showCount();  // Output: Count for type f: 1

return 0;
}
```

Class Template and Inheritance

Concepts of inheritance work in a similar way for class templates also. There could be a few major scenarios in Class Template Inheritance which are listed below.

1. Base Class is not a Template class, but a Derived class is a Template class. We can derive from the non-template class and add template members to the derived class.

```
class Base {

};

template < class T >
class Derived: public Base {
//Use T inside the Derived class
};
```

2. Base Class is a Template class, but Derived class is not a Template class. We can derive from a template class. If we don't want our derived class to be generic, we can use the Base class by providing the template parameter type.

```
template<class T>
class Base {

};
```

```
//Inheriting from the Base<int>
class Derived : public Base<int>{


};
```

3. Base Class is a Template class, and the Derived class is also a Template class. If we want our derived class to be generic, then it should be a template that can pass the template parameter to the base class.

```
template<class T>
class Base {


};


template<class T>
class Derived : public Base<T>{
//Pass the T to Base class
};
```

Here Base Class and Derived Class are using the same Type.

4. Base Class is a Template Class, and derived class is a Template class with different Types.

We can also use more types in the derived class by including them in the template parameter of the derived class template.

```
template<class T>
class Base {


};


template<class U, class T>
class Derived : public Base<T>{
//Use U in Derived class and pass T to Base class.
//We can also use U and T in the Derived class.
};
```

Templates vs. Macros

The templates and macros are somewhat similar; the table below demonstrates the few differences between them.

In C++, **templates** and **macros** are both tools that help in writing reusable code, but they are **fundamentally different** in how they work, how they're used, and what they offer. Templates are a **modern, type-safe** feature of C++, while macros are a **preprocessor feature** that comes from the C language.

1. Definition and Purpose

- **Templates** are a C++ feature that allows functions or classes to operate with generic types. They are evaluated and type-checked at **compile time**.
- **Macros**, defined using #define, are handled by the **preprocessor** before compilation. They are used to create code shortcuts or constants, but they do not follow C++'s type rules.

2. Syntax Comparison

Template Example:

```
template <typename T>
T maxVal(T a, T b) {
return (a > b) ? a : b;
}
```

Macro Example:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

While both seem to do the same thing, the behavior is different.

3. Type Safety

- **Templates** are type-safe. The compiler checks types when a template is instantiated.
- **Macros** do not perform type checking and can lead to **unexpected behavior** or errors.

Example:

```
cout << maxVal(5, 2);  // Template: Safe
cout << MAX(5, 2);     // Macro: May work, but not type-checked
```

4. Debugging and Error Handling

- **Templates** generate meaningful compiler errors if used incorrectly.
- **Macros** can be hard to debug because errors are often **cryptic** and point to the **expanded macro code** instead of the original source.

5. Scope and Namespaces

- **Templates** respect **C++ scopes** and can be declared in classes or namespaces.
- **Macros** are **global** and do not respect scopes, which can lead to **name collisions**.

6. Code Expansion and Evaluation

- **Templates** are evaluated during **compilation**, which means you get **optimized and type-safe code**.
- **Macros** are simply **text replaced** before compilation, which can lead to **code bloat or bugs** if not carefully written.

7. Functionality

Templates support:

- Function overloading
- Default arguments
- Specialization
- Inheritance in classes

Macros cannot support any of these features.

8. Use Cases

- Use **templates** when working with **type-generic classes or functions**, such as in the Standard Template Library (STL).
- Use **macros** for **constant definitions**, header guards, or very simple code snippets — but **avoid using them for logic or functions**.

Templates	Macros
Templates are the special function or classes with generic types.	Macros are the segment of code that is replaced by the macro value. They are defined by #define directive.
Templates get instantiated by the compiler only if a function call exists.	The preprocessor resolves macros during the first phase of compilation, i.e., preprocessing.
Easy to debug, because at the end template is just a function	Little bit complex to debug because the preprocessor handles everything. Also, while working with macros, you can notice that the error for macros is being shown up at the line where macro is defined.
Template is an advanced form of substitution.	Macro is a primitive form of substitution.
Templates are nothing but function calls, hence less efficient as compared to macros.	The macros are efficient because of inline compilation
Templates go through type-checking during compilation.	There is no type checking in macros.

Advantages and Disadvantages of Templates

Advantages

1. Code Compaction, By using templates, we can define generalized functionality and eliminate the repetition of several algorithms for different data types.

2. Reusability of Code, We can define a template that can be used with multiple data types.

3. In-Demand Compilation, One of the amazing advantages of using the template is that when the compiler encounters any function call with associated data types, only the new function is instantiated from the template.

Disadvantages

1. Data Hiding issue, Since the declaration and definition of the template cannot be separated due to in-demand compilation, it is difficult to hide functionality written inside the template.

2. Lack of Portability, Not every compiler supports the templates; hence in rare cases, there might be some portability issues.

Bubble Sort Using Function Template

Below is the implementation of **bubble sort** with the function template.

```
#include <iostream>

using namespace std;

template < class Type >
void bubbleSort(Type arr[], int n) {
for (int i = 0; i < n - 1; i++) {
bool swapDone = false;
for (int j = 0; j < n - i - 1; j++) {
if (arr[j] > arr[j + 1]) {
```

```
Type temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
swapDone = true;
}
}
if (!swapDone) return;
}
}
int main() {
int n = 5;
int arr1[] = {
11,
4,
9,
2,
0
};
float arr2[] = {
3.67,
9.87,
1.22,
2.45,
4.32
};

bubbleSort(arr1, n);
bubbleSort(arr2, n);
cout << "Sorting of Integers\n";
for (int i = 0; i < n; i++) {
cout << arr1[i] << " ";
}
cout << "\nSorting of Floating Point Numbers\n";
for (int i = 0; i < n; i++) {
```

```
cout << arr2[i] << " ";
}
return 0;
}
```

Output:

```
Sorting of Integers
0 2 4 9 11
Sorting of Floating Point Numbers
1.22 2.45 3.67 4.32 9.87
```

Explanation:

- We have defined a function template for bubble sort, which has template parameter T.
- As soon as we call the bubbleSort function for arr1, the compiler will instantiate a bubbleSort function for the int data type and use it for that function call.
- Similarly, this template will work with every data type, i.e., char, int, float, etc.

Linked List Using Class Template

Below shown is the template for a linked list. There are two private members, as usual, and three public methods, namely,

- printList: To print the entire list
- pushFront: To push the data into the front of the linked list.
- deleteFront: To delete the data from the front of the linked list.

```
template < class T >
class List {
private:
T data;
List * next;

public:
void printList(List * head) {
```

```
while (head != NULL) {
cout << head -> data << " ";
head = head -> next;
}
cout << endl;
}

List * pushFront(List * head, T data) {
List * t = new List;
t -> data = data;
t -> next = head;
head = t;
return head;
}

List * deleteFront(List * head) {
head = head -> next;
return head;
}
};
```

Let's see how we can utilize it for different data types,

1. List of Strings

```
#include <iostream>

using namespace std;

int main() {
// Creating a head of the linked list.
List < string > * head = NULL;

// Creating a Linked List.
List < string > myList;

head = myList.pushFront(head, "A");
head = myList.pushFront(head, "B");
head = myList.pushFront(head, "C");
head = myList.pushFront(head, "D");
head = myList.deleteFront(head);

myList.printList(head);
return 0;
}
```

Output:

```
C B A
```

2. List of Integers

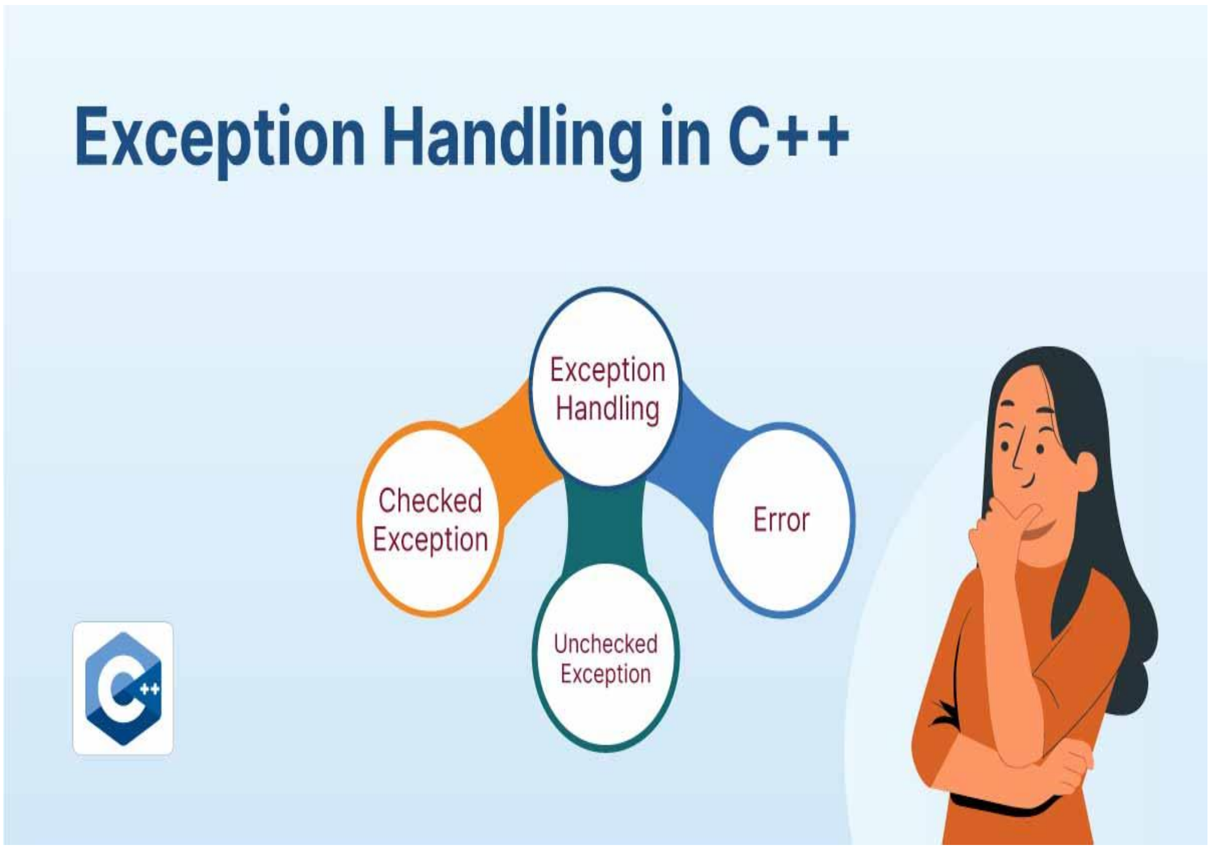
```
int main() {
// Creating a head of the linked list.
List < int > * head = NULL;

// Creating a Linked List.
List < int > myList;

head = myList.pushFront(head, 10);
head = myList.pushFront(head, 20);
head = myList.pushFront(head, 30);
head = myList.pushFront(head, 40);
myList.printList(head);

return 0;
}
```

Output:

```
40 30 20 10
```

As shown in the above example, we could use the same list class template for string data type, integer data type, and even for any other data type.

1.4 EXCEPTIONS



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**.

- **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** − 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.

1.5 Throwing Exceptions

In modern C++ programming, error handling is a critical aspect of writing robust and reliable code. While traditional methods like return codes and flags can be used, they often lead to confusing or error-prone code. C++ provides a much more powerful and structured approach through **exception handling** — and at the core of this mechanism is the **throw keyword**.

What is an Exception?

An **exception** is a problem or error that arises during the execution of a program. It could be due to invalid input, out-of-range array access, division by zero, or system-level issues like memory allocation failure.

Instead of letting the program crash or behave unpredictably, C++ allows you to detect and manage such errors using exceptions.

What is throw?

The **throw keyword** in C++ is used to **raise or signal an exception**. When something goes wrong, you throw an exception, and it is caught using a catch block.

Basic Syntax:

throw exception_value;

This transfers control from the current block to the nearest matching catch block.

Basic Exception Handling Structure

C++ uses three main keywords for exception handling:

1. try – Encapsulates code that might throw an exception
2. throw – Signals an exception
3. catch – Handles the exception

Example:

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

int main() {
try {
int a = 10, b = 0;
if (b == 0)
throw "Division by zero error!";
cout << a / b;
}
catch (const char* msg) {
cout << "Exception: " << msg << endl;
}
return 0;
}
```

Output:

```
Exception: Division by zero error!
```

How throw Works

When a throw statement is encountered:

1. Program control leaves the try block.
2. It looks for a matching catch block.
3. If a match is found, the control goes there.
4. If no match is found, the program terminates with an error (std::terminate() is called).

Types of Exceptions You Can Throw

You can throw:

- Built-in types: int, float, char*, etc.
- Standard exception classes: std::runtime_error, std::out_of_range, etc.
- User-defined classes

Example 1: Throwing Built-in Types

```
throw 404;
throw 3.14;
throw "Error occurred!";
```

Example 2: Throwing Standard Exception

```
#include <stdexcept>
throw std::runtime_error("File not found");
```

Example 3: Throwing a User-Defined Exception

```
class MyException {
public:
const char* what() const {
return "My custom exception!";
}
};

try {
throw MyException();
}
catch (MyException e) {
cout << e.what();
}
```

Matching catch Blocks

A thrown exception must match a catch block in type. You can have multiple catch blocks for different types.

```
try {
throw 100;
}
catch (int e) {
cout << "Caught integer: " << e;
}
catch (...) {
cout << "Caught some exception!";
}
```

The ... catch block is a **generic handler** that catches any exception not already caught above.

Throwing Exceptions from Functions

Functions can throw exceptions. These exceptions are caught by the calling function, if wrapped in try-catch.

Example:

```
void divide(int a, int b) {
if (b == 0)
throw "Cannot divide by zero!";
cout << a / b;
}

int main() {
try {
divide(10, 0);
}
catch (const char* msg) {
cout << "Exception: " << msg;
}
}
```

Re-throwing Exceptions

Sometimes a catch block cannot fully handle the exception. You can re-throw it using throw;

```
try {
try {
throw 5;
}
catch (int e) {
cout << "Caught in inner catch, rethrowing...\n";
throw;
}
}
catch (int e) {
cout << "Caught in outer catch: " << e;
}
```

Exception Specification (Legacy Feature)

In older C++ versions, you could declare which exceptions a function might throw using:

```
void func() throw(int, float);
```

This is **deprecated in C++11** and later. Modern C++ uses noexcept to specify functions that **do not** throw exceptions.

Custom Exception Class (Best Practice)

It's better to define your own exceptions by inheriting from std::exception.

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

class MyError : public exception {
public:
const char* what() const noexcept override {
return "My custom error!";
}
};

int main() {
try {
throw MyError();
}
catch (const exception& e) {
```

```
cout << "Caught: " << e.what();
}
}
```

This approach integrates well with the Standard Library.

Real-World Use Cases

1. **File I/O –** File not found, file corrupt
2. **Math operations –** Divide by zero, overflow
3. **Memory issues –** Allocation failure
4. **Containers –** Out-of-range access (std::vector::at)
5. **Custom APIs** – Invalid input, failed processing

Benefits of Using throw

Benefit	Explanation
Cleaner Code	No need to check errors after every line
Separation of Logic	Keeps business logic and error handling apart
Reusability	Can reuse catch blocks for multiple functions
Flexibility	Can throw any data type or object

1.6 Catching Exceptions

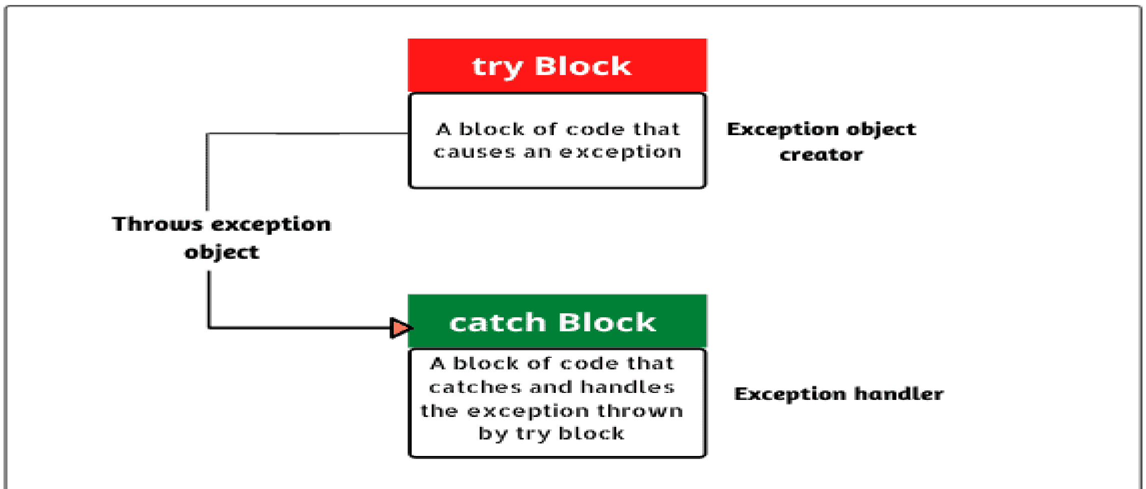
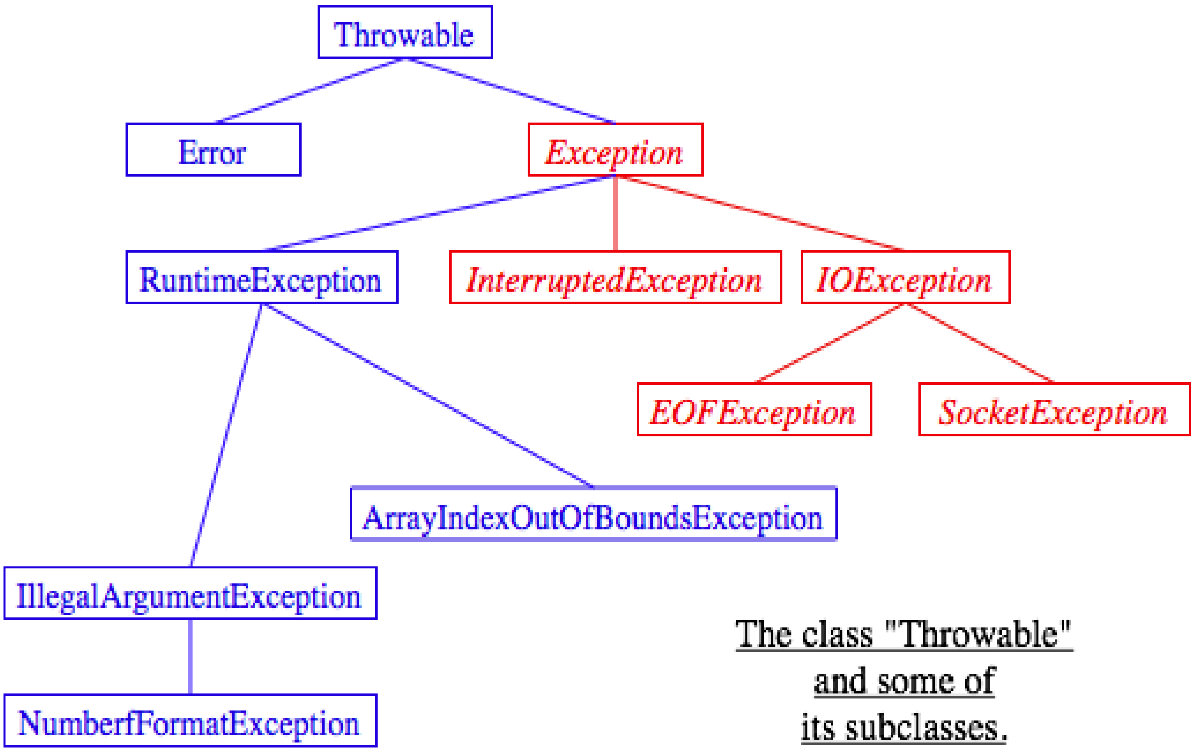
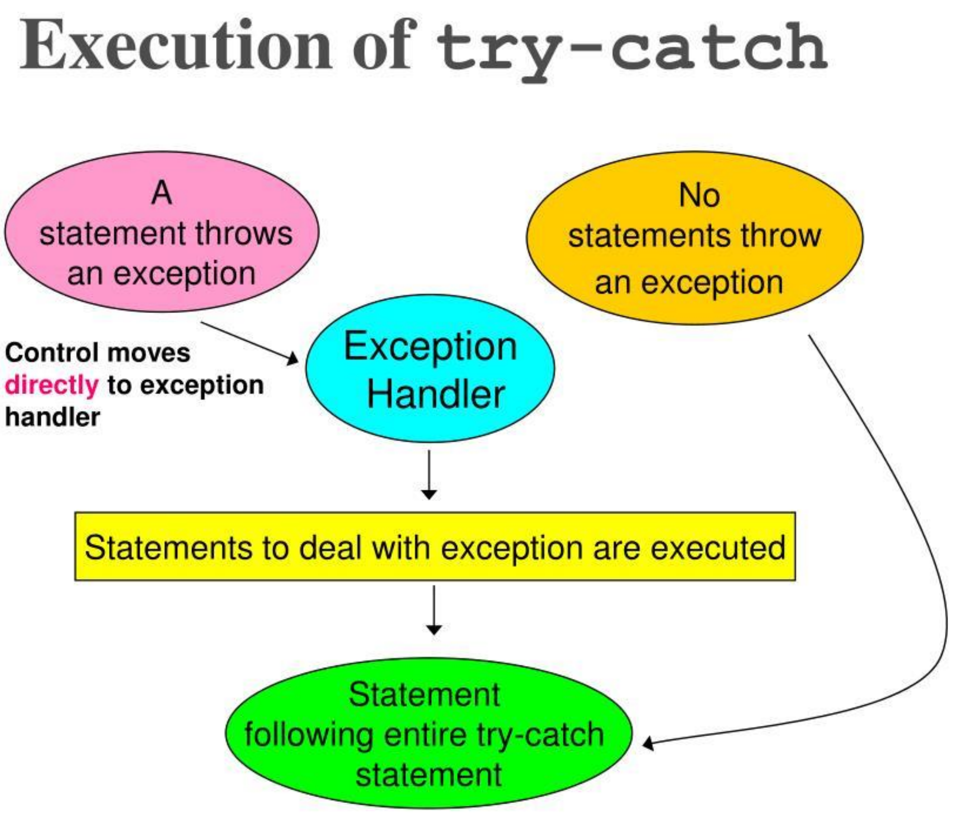

try Block
A block of code that causes an exception
Exception object creator
Throws exception object
catch Block
A block of code that catches and handles the exception thrown by try block
Exception handler

Fig: Exception handling mechanism



The class "Throwable" and some of its subclasses.


Execution of try-catch
A statement throws an exception
No statements throw an exception
Control moves directly to exception handler
Exception Handler
Statements to deal with exception are executed
Statement following entire try-catch statement

;

The try block contains code that may generate an exception. If an error is detected, the throw keyword is used to signal it. The catch block handles the exception, allowing the program to continue running or terminate cleanly.

Importance of Exception Handling

Traditional C-style error handling relies on returning error codes, which increases complexity and can be easily overlooked. Exception handling:

- Separates error-handling code from regular logic.
- Makes code cleaner and more understandable.
- Prevents unexpected program termination.
- Allows you to handle different types of errors appropriately.

Basic Syntax

```
try {
// Code that might cause an exception
throw exception; // Raise an exception
}
catch (type_of_exception var) {
// Code to handle the exception
}
```

Example: Divide by Zero

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

int main() {
int a = 10, b = 0;
try {
if (b == 0)
throw "Cannot divide by zero";
cout << "Result: " << a / b << endl;
}
catch (const char* msg) {
cout << "Exception caught: " << msg << endl;
}
return 0;
}
```

Output:

```
Exception caught: Cannot divide by zero
```

Explanation:

- The division is risky (b is 0), so it's placed inside a try block.
- When the condition is true, throw sends an exception message.
- The catch block catches and handles the exception.

Flow of Execution

1. The program enters the try block.
2. If no exception occurs, the catch block is skipped.
3. If an exception is thrown, the rest of the try block is skipped.
4. The control is passed to the matching catch block.
5. If no matching catch block is found, the program calls terminate() and stops.

Multiple Catch Blocks

C++ allows multiple catch blocks to handle different exception types.

```
try {
throw 3.14;
}
catch (int e) {
cout << "Caught int: " << e;
}
catch (double e) {
cout << "Caught double: " << e;
}
```

Output:

```
Caught double: 3.14
```

Each catch block handles a specific type. The first one that matches the thrown exception is executed.

Catch-All Handler

You can use catch(...) to catch any type of exception when the exact type is unknown.

```
try {
throw 5;
}

catch (...) {
cout << "Caught some exception.";
}
```

This is useful for debugging or as a fallback when no specific catch block is available.

Exception in Functions

Exceptions can be thrown from functions and caught in main() or any calling function.

```
void divide(int x, int y) {
if (y == 0)
throw "Division by zero error!";
cout << "Result: " << x / y << endl;
}

int main() {
try {
divide(10, 0);
}
catch (const char* msg) {
cout << "Caught exception: " << msg << endl;
}
}
```

Re-throwing Exceptions

Sometimes, a catch block may want to re-throw the exception to be handled elsewhere

```
try {
try {
throw "Error!";
}
catch (const char* e) {
cout << "Caught inside inner block: " << e << endl;
throw;  // Re-throwing the exception
}
}
catch (const char* e) {
cout << "Caught in outer block: " << e << endl;
}
```

Standard Exception Classes

C++ includes built-in exception types defined in the <exception> header. The base class is:

```
class exception {
public:
virtual const char* what() const noexcept;
};
```

Common derived classes:

- std::bad_alloc – thrown by new if memory allocation fails.
- std::out_of_range – for containers like vectors or strings.
- std::invalid_argument – for invalid arguments in functions.
- std::runtime_error – general-purpose runtime errors.

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

int main() {
try {
throw runtime_error("Runtime failure");
}
catch (exception& e) {
cout << "Caught: " << e.what();
}
}
```

Creating Custom Exceptions

You can define your own exception classes by inheriting from std::exception.

```
class MyException : public exception {
public:
const char* what() const noexcept override {
return "My custom exception!";
}
};

int main() {
try {
throw MyException();
}
catch (const exception& e) {
cout << e.what();
}
}
```

C++ 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 below –

1. C++ Standard Exceptions are predefined classes in the C++ Standard Library that represent various types of runtime errors.
2. **Base Class – std::exception**:

   All standard exception classes are derived from the base class std::exception, which provides a virtual what() function to describe the error.
3. **Header File**:

   To use standard exceptions, you must include the header:

   #include <stdexcept>
4. **Types of Standard Exceptions**:

   Some common types include:
   - std::out_of_range
   - std::invalid_argument
   - std::overflow_error
   - std::underflow_error
   - std::bad_alloc

     These represent different error categories.
5. **Use with STL Containers**:

   STL containers like vector, map, etc., throw standard exceptions (e.g., std::out_of_range) when invalid operations are performed.
6. **Polymorphism Support**:

   You can catch exceptions polymorphically using a base class reference or pointer (e.g., catch (const std::exception& e)).

7. **what() Function**:

   All standard exceptions override the what() method, which returns a C-style string describing the cause of the error.

8. **Memory Errors – std::bad_alloc**:

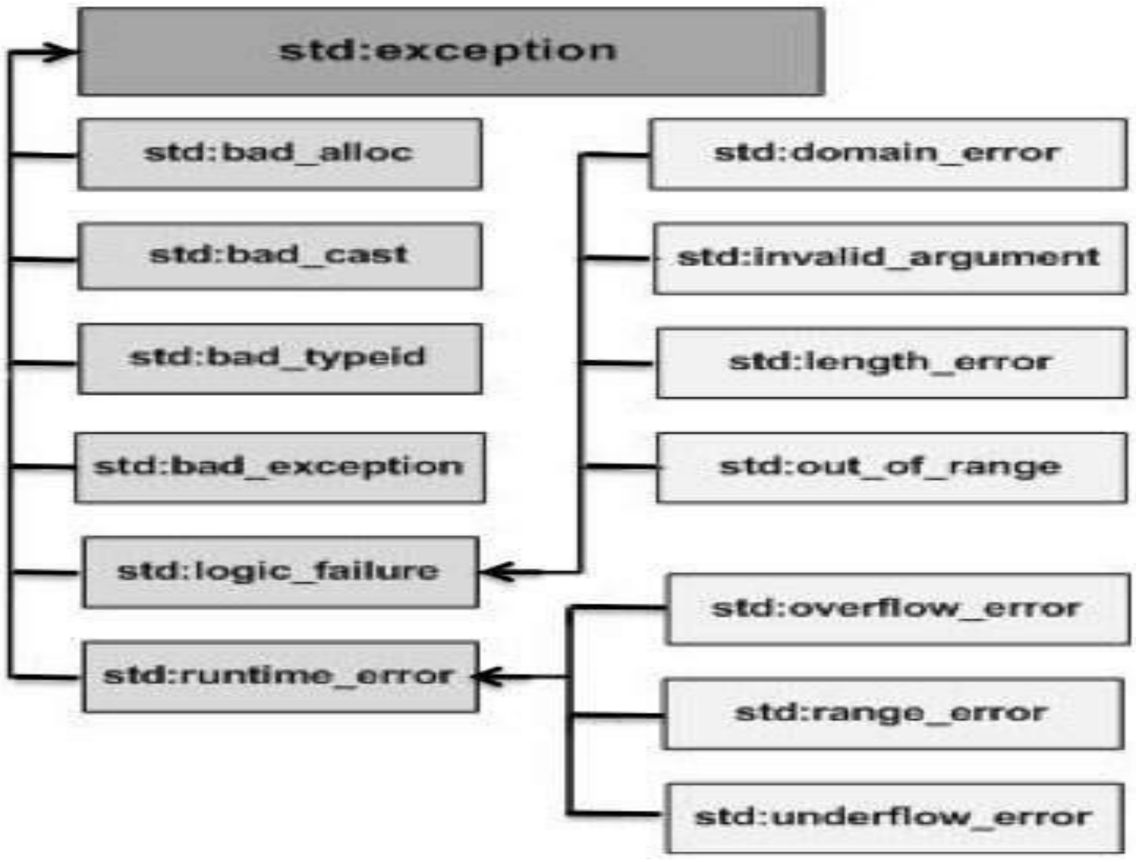
   Thrown when dynamic memory allocation (e.g., new) fails due to insufficient memory.

9. **Exception Safety**:

   Using standard exceptions helps write safer code by enabling proper error handling and avoiding crashes.

10. **Custom Exceptions**:

    You can define your own exception classes by inheriting from std::exception or its derived classes and overriding what().



Here is the small description of each exception mentioned in the above hierarchy −

Sr.No	Exception & Description
1	**std::exception** An exception and parent class of all the standard C++ exceptions.
2	**std::bad_alloc** This can be thrown by **new**.
3	**std::bad_cast** This can be thrown by **dynamic_cast**.
4	**std::bad_exception** This is useful device to handle unexpected exceptions in a C++ program.
5	**std::bad_typeid** This can be thrown by **typeid**.
6	**std::logic_error** An exception that theoretically can be detected by reading the code.
7	**std::domain_error** This is an exception thrown when a mathematically invalid domain is used.
8	**std::invalid_argument** This is thrown due to invalid arguments.
9	**std::length_error** This is thrown when a too big std::string is created.
10	**std::out_of_range** This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]().
11	**std::runtime_error**

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

12 **std::overflow_error**

This is thrown if a mathematical overflow occurs.

13 **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 for c++ standand exception

```
#include <iostream>

#include <vector>

#include <stdexcept>  // For standard exceptions

int main() {

std::vector<int> numbers = {10, 20, 30};

try {

// Try accessing out-of-bound index

std::cout << "Accessing element at index 5: " << numbers.at(5) << std::endl;

}

catch (const std::out_of_range& e) {

std::cerr << "Caught an out_of_range exception: " << e.what() << std::endl;

}
catch (const std::exception& e) {

std::cerr << "Caught a generic exception: " << e.what() << std::endl;

}

return 0; }
```

Define New Exceptions

You can define your own exceptions by inheriting and overriding **exception** class functionality.

Example

Following is the example, which shows how you can use std::exception class to implement your own exception in standard way −

```
Open Compiler
#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
```

Here, **what()** is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

CASE STUDY:

1. Function Template for Finding Minimum Value in an Array

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

// Function template to find the minimum value in an array
template <typename T>
T findMin(T arr[], int size) {
T min = arr[0];  // Assume first element is minimum
for (int i = 1; i < size; i++) {
if (arr[i] < min)
min = arr[i];
}
return min;
}

int main() {
// Integer array
int intArr[] = {5, 2, 8, 1, 4};
int intSize = sizeof(intArr) / sizeof(intArr[0]);
cout << "Minimum value in integer array: " << findMin(intArr, intSize) << endl;

// Floating-point array
float floatArr[] = {3.5, 2.1, 8.6, 1.4, 4.9};
int floatSize = sizeof(floatArr) / sizeof(floatArr[0]);
cout << "Minimum value in floating-point array: " << findMin(floatArr, floatSize) << endl;

return 0;
}
```

Expected Output:

```
Minimum value in integer array: 1
Minimum value in floating-point array: 1.4
```

2. *Exception Handling: Division by Zero*

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

// Function that performs division and throws an exception if divisor is zero
double divide(double numerator, double denominator) {
if (denominator == 0)
throw "Error: Division by zero!";
return numerator / denominator;
}

int main() {
double num, denom;
cout << "Enter numerator: ";
cin >> num;
cout << "Enter denominator: ";
cin >> denom;
try {
double result = divide(num, denom);
cout << "Result: " << result << endl;
} catch (const char* msg) {
cout << "Exception caught: " << msg << endl;
}

return 0;
}
```

Sample Run 1 (Valid Division):

```
Enter numerator: 10
Enter denominator: 2
Result: 5
```

Sample Run 2 (Division by Zero):

```
Enter numerator: 8
Enter denominator: 0
Exception caught: Error: Division by zero!
```

1.7 PRE DEFINED FUNCTIONS IN C++

C++ provides several **standard exception classes** in the **Standard Library (<exception> and <stdexcept>)** that are used to represent common runtime errors. These are called **predefined exceptions**, and they are derived from the base class std::exception.

Hierarchy of Predefined Exceptions:

```
std::exception
├── std::logic_error
│   ├── std::invalid_argument
│   ├── std::domain_error
│   ├── std::length_error
│   └── std::out_of_range
```

```
└── std::runtime_error
    ├── std::overflow_error
    ├── std::underflow_error
    └── std::range_error
```

Common Predefined Exceptions:

Exception Class	**Description**
std::exception	Base class for all standard exceptions.
std::logic_error	Errors in the internal logic of the program (e.g., invalid arguments).
std::runtime_error	Errors detected during runtime (e.g., arithmetic errors).
std::invalid_argument	Used when an invalid argument is passed to a function.
std::out_of_range	Thrown when accessing out-of-bound elements (like in vectors or arrays).
std::overflow_error	Indicates arithmetic overflow.
std::underflow_error	Indicates arithmetic underflow.

Example:

```
#include <iostream>
#include <stdexcept>

int main() {
   try {
      throw std::out_of_range("Index is out of range");
   } catch (const std::exception& e) {
      std::cout << "Caught exception: " << e.what() << std::endl;
   }
   return 0;
}
```

Output:

```
Caught exception: Index is out of range
```

These predefined exceptions make it easier to handle common error conditions in a structured and meaningful way.

IMPORTANT QUESTIONS

1. What are templates in C++? Why are they important?
2. Explain the syntax and working of function templates with an example.
3. Differentiate between function templates and class templates.
4. What are the advantages and disadvantages of using templates?
5. Write a function template to swap two values.
6. How do class templates help in generic programming? Explain with an example.
7. What are template parameters, and how do they work?
8. Explain template specialization with an example.
9. What is exception handling in C++? Why is it important?
10. Explain the try, catch, and throw statements with an example.
11. Write a program to handle division by zero using exception handling.
12. What are predefined exceptions in C++? Give examples.
13. Can we have multiple catch blocks for a single try block? Provide an example.
14. What is a catch-all exception handler? How is it used?
15. Explain how inheritance affects exception handling in C++.
16. How does template instantiation work in C++?
17. What are non-type template parameters? Provide an example.
18. What is partial specialization of templates? Explain with an example.
19. How can a template class be derived from another template class?
20. Write a class template for a simple queue implementation.
21. What is function template overloading? Explain with an example.
22. How do template friend functions work? Provide an example.
23. What are variadic templates, and how are they useful?
24. Explain SFINAE (Substitution Failure Is Not An Error) in C++.
25. Can a constructor throw an exception? How should it be handled?
26. What happens when an exception is thrown in a destructor?
27. What is rethrowing an exception, and how does it work?
28. Write a program to handle array index out of bound exceptions.
29. What is stack unwinding, and how does it relate to exception handling?
30. Can a catch block throw an exception? Explain with an example.
31. How do concepts like CRTP (Curiously Recurring Template Pattern) work with templates?

32. Explain enable_if and SFINAE for template metaprogramming.
33. What are advanced template techniques like type traits and type deduction?
34. How does the C++ standard library (STL) make use of templates?
35. Compare exception handling in C++ with Java/Python.

***************************END OF MODULE-5***************************