- **throw Statement:** The throw statement is followed by an expression, typically an object of an exception class. This expression is the exception that is being thrown.
- Throwing Syntax:

```
throw ExceptionType(arguments);
```

- Example:

```
throw runtime_error("Division by zero");
```

B. Catching Mechanism:

- The catching mechanism is used to handle exceptions thrown by the try block. It involves catch blocks.
- **catch Block:** A catch block is associated with a try block and specifies the type of exception it can handle. When an exception of that type is thrown, the corresponding catch block is executed.
- Example:

```
catch (const runtime_error& e) {
    cout << "Error: " << e.what() << endl;
}
```

- try-catch Syntax:

```
try {
    // Code that may throw an exception
} catch (ExceptionType1 e1) {
    // Code to handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Code to handle ExceptionType2
} catch (...) {
    // Catch-all block to handle any other exceptions
}
```

C. Rethrowing an Exception:

- Rethrowing an exception allows an exception caught in one catch block to be thrown again to be handled by another catch block.
- **throw Statement (inside catch block):** In a catch block, we can use the throw statement without any argument to rethrow the caught exception.
- Rethrow Syntax:

```
catch (ExceptionType e) {
    // Code to handle the exception
    throw; // Rethrow the exception
}
```

- Example:

```
catch (const runtime_error& e) {
    cout << "Caught error: " << e.what() << endl;
    throw; // Rethrow the exception
}
```

D. Specifying Exception :

- In older versions of C++, it was possible to specify the types of exceptions that a function could throw. This feature, known as exception specifications, is deprecated in C++11 and removed in C++17.
- Specifying exceptions allows to document the types of exceptions that a function may throw during its execution.
- Syntax:

```
returnType functionName(parameters) throw(ExceptionType1, ExceptionType2, ...)
{
    // Function body
}
```

- Instead of specifying exceptions, it's better to document the exceptions a function might throw in comments and handle them accordingly.

E. Standard Exception Classes:

In C++, the Standard Template Library (STL) provides a set of standard exception classes that can be used for common error scenarios. These classes are defined in the <stdexcept> header and serve as a hierarchy of exception types that cover a range of standard error conditions.

Common Standard Exception Classes:

1. std::exception:

- The base class for all standard C++ exceptions.
- Provides a virtual function what() that returns a descriptive string representing the exception.
- Developers can define their own custom exception types by deriving from std::exception or its subclasses.

2. std::runtime_error:

- Represents errors that occur during runtime and are typically not detectable before the program is executed.
- Derived from std::exception.
- Commonly used to report logical errors or exceptional conditions that arise during program execution.
- Example:

```
try {
    throw std::runtime_error("A runtime error occurred");
} catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

3. std::logic_error:

- Represents errors that occur due to logical errors in the program's design or implementation.
- Derived from std::exception.

- Examples include out-of-range errors, domain errors, and invalid argument errors.
- Example:

```
try {
    throw std::logic_error("A logic error occurred");
} catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

4. std::invalid_argument:

- Indicates that a function has received an invalid argument.
- Derived from std::logic_error.
- Used when a function is called with an argument that is not acceptable or within the expected range.
- Example:

```
try {
    throw std::invalid_argument("Invalid argument provided");
} catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

5. std::out_of_range:

- Indicates that an index or value is out of the valid range.
- Derived from std::logic_error.
- Commonly used in situations where accessing elements beyond the bounds of a container or array.
- Example:

```
try {
    throw std::out_of_range("Out of range exception");
} catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

6. std::bad_alloc:

- Represents errors that occur when memory allocation fails.
- Derived from std::exception.
- Typically thrown by the new operator when it fails to allocate memory.
- Example:

```
try {
    throw std::bad_alloc();
} catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

7. std::overflow_error:

- Indicates arithmetic overflow errors, where the result of an arithmetic operation exceeds the range of representable values.

- Derived from std::runtime_error.
- Commonly used in situations where arithmetic operations result in overflow.
- Example:

```
try {
    int result = std::numeric_limits<int>::max() + 1; // Overflow
} catch (const std::overflow_error& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

8. std::underflow_error:

- Indicates arithmetic underflow errors, where the result of an arithmetic operation is smaller than the minimum representable value.
- Derived from std::runtime_error.
- Less common than std::overflow_error but used in similar situations where arithmetic underflow occurs.
- Example:

```
try {
    float result = std::numeric_limits<float>::min() / 2; // Underflow
} catch (const std::underflow_error& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

Section 2 : Templates and Generic Programming

Templates and generic programming are powerful features of C++ that allow developers to write code that works with any data type. Templates provide a mechanism for creating generic classes and functions, allowing them to operate on multiple data types without the need for code duplication.

2.1 Template Concepts

- Definition: Templates are a feature of C++ that allows functions and classes to operate with generic types. They enable the creation of generic code that works with any data type.
- Benefits:
    - Code Reusability: Templates allow you to write code once and use it with different data types, promoting reuse.
    - Flexibility: Templates provide flexibility by allowing algorithms to work with various data types without sacrificing performance or type safety.
- Syntax: Template definitions begin with the template keyword followed by a list of template parameters enclosed in angle brackets < >.

2.2 Function Templates

- Definition: Function templates allow you to create a single function that can operate with different data types. They are instantiated to create specific functions for each data type when called.