- 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.

- They are defined using the template keyword followed by template parameters enclosed in angle brackets (<>).
- Template parameters can be type parameters or non-type parameters.
- Syntax:
```
template <typename T>
T functionName(T parameter) {
    // Function body
}
```
- Example:
```
#include <iostream>
using namespace std;

// Function template for adding two values of the same type
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    // Instantiation for int
    int result1 = add(5, 3);
    cout << "Result 1: " << result1 << endl;

    // Instantiation for double
    double result2 = add(3.5, 2.7);
    cout << "Result 2: " << result2 << endl;

    return 0;
}
```
In this example, the function template add can add two values of any data type (int, double, etc.).

2.3 Class Templates

- Definition: Class templates allow the creation of generic classes that can work with any data type. They are instantiated to create specific classes for each data type when used.
- They are defined similar to function templates, but instead of functions, entire classes are templated.
- Syntax:
```
template <typename T>
class ClassName {
    // Class members and methods
};
```
- Example:
```
#include <iostream>
```