3. Programming Language and Its Applications

3.1 Introduction to C Programming

- C Tokens, Operators
- Formatted/Unformatted Input/Output
- Control Statements, Looping
- User-defined Functions, Recursive Functions
- Array (1-D, 2-D, Multidimensional), and String Manipulations

---

3.2 Pointers, Structure, and Data Files in C Programming

- Pointer Arithmetic, Pointer and Array
- Passing Pointer to Function
- Structure vs Union, Array of Structure
- Passing Structure to Function, Structure and Pointer
- Input/Output Operations on Files
- Sequential and Random Access to File

3.3 C++ Language Constructs with Objects and Classes

- Namespace, Function Overloading
- Inline Functions, Default Argument
- Pass/Return by Reference
- Introduction to Class and Object, Access Specifiers
- Objects and the Member Access
- Defining Member Functions, Constructor and Its Type, Destructor
- Dynamic Memory Allocation for Objects and Object Array
- This Pointer, Static Data Member and Static Function
- Constant Member Functions and Constant Objects
- Friend Function and Friend Classes

3.4 Features of Object-Oriented Programming

- Operator Overloading (Unary, Binary), Data Conversion
- Inheritance (Single, Multiple, Multilevel, Hybrid, Multipath)
- Constructor/Destructor in Single/Multilevel Inheritances

3.5 Pure Virtual Function and File Handling

- Virtual Function, Dynamic Binding
- Defining, Opening, and Closing a File
- Input/Output Operations on Files
- Error Handling During Input/Output Operations

- Stream Class Hierarchy for Console Input/Output
- Unformatted Input/Output
- Formatted Input/Output with ios Member Functions and Flags
- Formatting with Manipulators

3.6 Generic Programming and Exception Handling

- Function Template, Overloading Function Template
- Class Template, Function Definition of Class Template
- Standard Template Library (Containers, Algorithms, Iterators)
- Exception Handling Constructs (try, catch, throw)
- Multiple Exception Handling, Rethrowing Exception
- Catching All Exceptions, Exception with Arguments
- Exceptions Specification for Function
- Handling Uncaught and Unexpected Exceptions

3.1 Introduction to C Programming

C is a general-purpose, **procedural**, and **middle-level** programming language used for developing **computer software**, **system programming**, **applications**, **games**, and more. Known for its simplicity and efficiency, C is an excellent choice for beginners as it provides a strong foundation in programming concepts. C was developed by Dennis M. Ritchie at Bell Laboratories in 1972.

1. C Tokens

- **C Tokens** are the building blocks of a C program. They are categorized into:
  - **Keywords**: Reserved words (e.g., `int`, `if`, `while`, `return`).
  - **Identifiers**: Names for variables, functions, and arrays (e.g., `main`, `num`).
  - **Constants**: Fixed values used in a program (e.g., `10`, `'A'`).
  - **Strings**: Sequence of characters enclosed in double quotes (e.g., `"Hello"`).
  - **Operators**: Symbols that perform operations (e.g., `+`, `-`, `*`, `/`).
  - **Special Characters**: Punctuation marks with specific functions (e.g., `{`, `}`, `;`).

---

2. Operators

- **Arithmetic Operators**: `+`, `-`, `*`, `/`, `%`.
- **Relational Operators**: `==`, `!=`, `<`, `>`, `<=`, `>=`.
- **Logical Operators**: `&&`, `||`, `!`.
- **Bitwise Operators**: `&`, `|`, `^`, `~`, `<<`, `>>`.
- **Assignment Operators**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`.
- **Increment/Decrement**: `++`, `--`.

---

3. Input and Output (Formatted and Unformatted)

- **Formatted I/O**:
  - Input: `scanf()` (e.g., `scanf("%d", &num);`)
  - Output: `printf()` (e.g., `printf("Value: %d", num);`)
- **Unformatted I/O**:
  - Input: `getchar()`, `gets()` (e.g., `char c = getchar();`)
  - Output: `putchar()`, `puts()` (e.g., `puts("Hello");`)

---

4. Control Statements

- **Decision-Making**:
  - `if`, `if-else`, `else if`, `switch`.
  - Example:
```
if (a > b) {
    printf("A is greater");
} else {
    printf("B is greater");
}
```
- **Looping**:
  - **For Loop**: Executes a block of code a fixed number of times.
```
for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```
  - **While Loop**: Executes as long as a condition is true.
```
codeint i = 0;
while (i < 10) {
    printf("%d\n", i);
    i++;
}
```
  - **Do-While Loop**: Executes at least once, then continues based on a condition.
```
int i = 0;
do {
    printf("%d\n", i);
    i++;
} while (i < 10);
```

---

5. Functions

- **User-Defined Functions**:
  - Functions created by the user for modularity and reusability.
  - Syntax:
```
return_type function_name(parameters) {
    // Function body
    return value;
}
```

- Example:
```
int add(int a, int b) {
    return a + b;
}
```

- **Recursive Functions**:
  - Functions that call themselves, typically used for tasks like calculating factorials or solving Fibonacci series.
  - Example:
```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

---

6. Arrays

- **1D Arrays**:
  - Syntax:
```
int arr[10];
```
  - Example:
```
int arr[5] = {1, 2, 3, 4, 5};
for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
}
```
- **2D Arrays**:
  - Syntax:
```
int matrix[3][3];
```
  - Example:
```
int matrix[2][2] = {{1, 2}, {3, 4}};
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
        printf("%d ", matrix[i][j]);
    }
}
```
- **Multi-Dimensional Arrays**:
  - Arrays with more than two dimensions (rarely used).

---

7. String Manipulations

- **String Functions** (from `<string.h>`):
  - `strlen()`: Finds the length of a string.
  - `strcpy()`: Copies one string to another.
  - `strcat()`: Concatenates two strings.
  - `strcmp()`: Compares two strings.
  - Example:
```
char str1[10] = "Hello";
```

```
char str2[10];
strcpy(str2, str1);
printf("%s", str2); // Output: Hello
```

Conclusion

C is a versatile and efficient language that lays the groundwork for programming concepts. Its features, such as tokens, operators, control statements, arrays, functions, and string manipulations, empower developers to write structured and robust code. C's simplicity and modularity make it an ideal choice for beginners and a powerful tool for developing a wide range of applications, from system software to games.

3.2 Pointers, Structures, and Data Files

1. Pointers

1. **What is a Pointer?**
   - A pointer is a variable that stores the *memory address* of another variable.
   - Syntax:
     ```
     int a = 10;
     int *p = &a;  // 'p' is a pointer to 'a', storing its address.
     printf("%d", *p); // Dereference 'p' to access the value of 'a': 10
     ```
2. **Pointer Arithmetic**
   - You can perform arithmetic operations on pointers to navigate through memory locations.
   - Valid operations:
     - **Increment (`p++`)**: Moves to the next memory location of the same type.
     - **Decrement (`p--`)**: Moves to the previous memory location.
     - **Addition/Subtraction (`p + n`, `p - n`)**: Adjusts the pointer by `n` locations.
   - Example:
     ```
     int arr[] = {10, 20, 30};
     int *p = arr;     // Points to the first element of 'arr'.
     printf("%d\n", *p);  // Output: 10
     p++;                 // Move pointer to the next element.
     printf("%d\n", *p);  // Output: 20
     ```
3. **Passing Pointers to Functions**
   - Pointers allow functions to modify the original value of variables.
   - Example:
     ```
     void increment(int *n) {
         (*n)++; // Increment the value pointed to by 'n'
     }

     int main() {
         int num = 5;
         increment(&num);  // Pass the address of 'num'.
         printf("%d", num);  // Output: 6
         return 0;
     }
     ```

---

2. Structures

1. **What is a Structure?**
   - A structure is a user-defined data type that groups variables of different types under a single name.
   - Syntax:
   ```
   struct Student {
       int id;
       char name[50];
   };
   ```
   - **Declaration**:
   ```
   struct Student s1;
   s1.id = 101;
   strcpy(s1.name, "Alice"); // Assign string to 'name'
   printf("ID: %d, Name: %s\n", s1.id, s1.name);  // Output: ID: 101, Name: Alice
   ```
2. **Pointer to a Structure**
   - Accessing structure members using a pointer:
   ```
   struct Student s1 = {101, "Alice"};
   struct Student *ptr = &s1;
   printf("ID: %d, Name: %s\n", ptr->id, ptr->name);  // Output: ID: 101, Name: Alice
   ```
3. **Array of Structures**
   - Example:
   ```
   struct Student {
       int id;
       char name[50];
   };

   struct Student students[2] = {{101, "Alice"}, {102, "Bob"}};
   printf("Student 1: %s\n", students[0].name);  // Output: Student 1: Alice
   printf("Student 2: %s\n", students[1].name);  // Output: Student 2: Bob
   ```

---

3. Unions

1. **What is a Union?**
   - A union is similar to a structure but uses shared memory for all its members. Only one member can hold a value at any given time.
   - Syntax:
   ```
   union Data {
       int i;
       float f;
   };

   union Data d;
   d.i = 5;
   printf("%d\n", d.i);  // Output: 5
   ```

```
d.f = 3.14;
printf("%f\n", d.f);  // Output: 3.140000
```

2. **Key Difference from Structures**
   - **Structure**: Allocates memory for all members.
   - **Union**: Allocates memory for the largest member only.

---

4. File Operations

1. **File Modes**
   - `"r"`: Open a file for reading.
   - `"w"`: Open a file for writing (overwrites if the file exists).
   - `"a"`: Open a file for appending (writes at the end without overwriting).
2. **Basic File Handling Functions**
   - **`fopen()`**: Opens a file.
   - **`fclose()`**: Closes a file.
   - **`fprintf()` and `fscanf()`**: Handles formatted input/output.
   - **`fread()` and `fwrite()`**: Handles binary data.
3. **Examples**
   - **Writing to a File**:

```
include <stdio.h>
int main() {
    FILE *file = fopen("data.txt", "w");  // Open file in write mode
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(file, "Hello, World!\n");  // Write to file
    fclose(file);  // Close file
    return 0;
}
```

   - **Reading from a File**:

```
#include <stdio.h>
int main() {
    FILE *file = fopen("data.txt", "r");  // Open file in read mode
    char str[100];
    if (file == NULL) {
        printf("File not found!\n");
        return 1;
    }
    while (fscanf(file, "%s", str) != EOF) {  // Read words until EOF
        printf("%s\n", str);  // Print each word
    }
    fclose(file);  // Close file
    return 0;
}
```

   - **Binary File Handling**:

```c
#include <stdio.h>
struct Student {
    int id;
    char name[50];
};

int main() {
    FILE *file = fopen("student.dat", "wb");  // Open binary file for writing
    struct Student s = {101, "Alice"};
    fwrite(&s, sizeof(s), 1, file);  // Write structure to file
    fclose(file);  // Close file

    file = fopen("student.dat", "rb");  // Open binary file for reading
    struct Student readStudent;
    fread(&readStudent, sizeof(readStudent), 1, file);  // Read structure
    printf("ID: %d, Name: %s\n", readStudent.id, readStudent.name);  // Output: ID: 1
    fclose(file);  // Close file
    return 0;
}
```

4. **Common File Handling Errors**
   - Forgetting to close a file.
   - Trying to read a file that doesn't exist.
   - Using the wrong file mode.

Conclusion

- Pointers, structures, unions, and file operations are essential concepts in C programming. Pointers provide direct memory manipulation and efficient data handling.
- Structures enable grouping of related data, while unions optimize memory usage by sharing storage among members.
- File operations facilitate data persistence and management through various modes and functions.

3.3 C++ Language Constructs with Objects and Classes

1. Namespace

- A **namespace** organizes code and prevents naming conflicts between variables, functions, or classes.
- Example: The standard C++ library is contained within the `std` namespace.

```cpp
#include <iostream>
// Using a namespace
namespace MyNamespace {
    int value = 100;
}

int main() {
    std::cout << "Value in MyNamespace: " << MyNamespace::value << std::endl;
```

```
    // Output: Value in MyNamespace: 100
    return 0;
}
```

---

2. Function Overloading

- Function overloading allows multiple functions with the same name but different parameter types or numbers of parameters.
- The compiler differentiates between them based on the argument list.

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

// Function Overloading Example
void display(int num) {
    cout << "Integer: " << num << endl; // Output: Integer: 5
}

void display(double num) {
    cout << "Double: " << num << endl; // Output: Double: 5.5
}

int main() {
    display(5);    // Calls the integer version
    display(5.5);  // Calls the double version
    return 0;
}
```

---

3. Inline Functions

- **Inline functions** reduce function call overhead by embedding the function code directly into the calling code during compilation.
- Use the `inline` keyword before the function definition.

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

inline int square(int x) {
    return x * x;  // Inline function logic
}

int main() {
    cout << "Square of 5: " << square(5) << endl; // Output: Square of 5: 25
    return 0;
}
```

---

4. Default Arguments

- Default arguments allow a function to be called with fewer arguments by specifying default values for some parameters.
- Default arguments must be specified from right to left in the parameter list.

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

// Function with Default Arguments
void greet(string name = "Guest") {
    cout << "Hello, " << name << "!" << endl;
}

int main() {
    greet();          // Output: Hello, Guest!
    greet("Alice");   // Output: Hello, Alice!
    return 0;
}
```

---

5. Classes and Objects

- **Classes** are user-defined data types that represent real-world entities, encapsulating data members and member functions.
- **Objects** are instances of classes.

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

class Rectangle {
public:
    int length, width;

    // Member function
    int area() {
        return length * width;
    }
};

int main() {
    Rectangle rect;       // Create an object of the class
    rect.length = 10;
    rect.width = 5;

    cout << "Area: " << rect.area() << endl; // Output: Area: 50
    return 0;
}
```

---

6. Constructors and Destructors

- **Constructors** initialize objects when they are created. They share the same name as the class and have no return type.
- **Destructors** clean up resources when the object is destroyed. They have the same name as the class but are preceded by a `~`.

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

class Demo {
public:
    Demo() { // Constructor
        cout << "Constructor called!" << endl;
    }
    ~Demo() { // Destructor
        cout << "Destructor called!" << endl;
    }
};

int main() {
    Demo obj; // Constructor is called here
    // Destructor is automatically called at the end of the program
    return 0;
}
```

---

7. Dynamic Memory Allocation

- C++ provides `new` and `delete` operators for dynamic memory allocation and deallocation.

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

int main() {
    // Allocate memory dynamically
    int* ptr = new int(42);
    cout << "Value: " << *ptr << endl; // Output: Value: 42

    delete ptr; // Deallocate memory
    return 0;
}
```

---

8. Friend Functions

- A **friend function** has access to private and protected members of a class. It is declared using the `friend` keyword inside the class.

Conclusion

C++ is a powerful object-oriented programming language that supports advanced features such as namespaces, function overloading, inline functions, default arguments, classes and objects, constructors and destructors, dynamic memory allocation, and friend functions. These features provide flexibility, reusability, and efficiency, enabling developers to create robust and scalable software. With its rich standard library and versatile constructs, C++ remains a top choice for applications ranging from system software to complex simulations and games.

3.4 Features of Object-Oriented Programming

Object-Oriented Programming (OOP) provides powerful features to enhance the modularity, reusability, and functionality of code. This section focuses on **operator overloading**, **data conversion**, and **inheritance** (single, multiple, multilevel, hybrid).

---

1. Operator Overloading

Operator overloading allows the redefinition of operators for user-defined types (e.g., classes). It helps to perform operations on objects intuitively, similar to primitive data types.

Example: Overloading the + operator for a class representing complex numbers.

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

class Complex {
    int real, imag;
public:
    Complex(int r = 0, int i = 0) : real(r), imag(i) {}

    // Overload the '+' operator
    Complex operator+(const Complex& obj) {
        return Complex(real + obj.real, imag + obj.imag);
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3, 4), c2(1, 2);
    Complex c3 = c1 + c2; // Using overloaded '+' operator
```

```
    c3.display(); // Output: 4 + 6i
    return 0;
}
```

---

2. Data Conversion

Data conversion refers to the process of converting one data type to another. This can involve user-defined types and is achieved by overloading type-casting operators.

Example: Converting an object into an integer.

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

class Distance {
    int meters;
public:
    Distance(int m) : meters(m) {}

    // Overload type-casting operator
    operator int() {
        return meters;
    }
};

int main() {
    Distance d(10);
    int meters = d; // Implicit conversion using overloaded operator
    cout << "Distance in meters: " << meters << endl; // Output: Distance in meters: 10
    return 0;
}
```

---

3. Inheritance

Inheritance allows one class to derive properties and methods from another, promoting **code reuse** and a hierarchical structure. There are several types of inheritance in C++.

---

a. Single Inheritance

One class inherits from another.

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

class Parent {
public:
    void show() {
```

```
        cout << "This is the Parent class." << endl;
    }
};

class Child : public Parent { // Single inheritance
};

int main() {
    Child obj;
    obj.show(); // Output: This is the Parent class.
    return 0;
}
```

---

b. Multiple Inheritance
A class inherits from more than one base class.

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

class ClassA {
public:
    void displayA() {
        cout << "Class A" << endl;
    }
};

class ClassB {
public:
    void displayB() {
        cout << "Class B" << endl;
    }
};

class ClassC : public ClassA, public ClassB { // Multiple inheritance
};

int main() {
    ClassC obj;
    obj.displayA(); // Output: Class A
    obj.displayB(); // Output: Class B
    return 0;
}
```

---

c. Multilevel Inheritance
A class inherits from another class, which in turn inherits from another class.

```
#include <iostream>
```

```cpp
using namespace std;

class Grandparent {
public:
    void displayGrandparent() {
        cout << "Grandparent class" << endl;
    }
};

class Parent : public Grandparent {
public:
    void displayParent() {
        cout << "Parent class" << endl;
    }
};

class Child : public Parent { // Multilevel inheritance
};

int main() {
    Child obj;
    obj.displayGrandparent(); // Output: Grandparent class
    obj.displayParent();      // Output: Parent class
    return 0;
}
```

---

d. Hybrid Inheritance

A combination of two or more types of inheritance (e.g., single and multiple). This often involves using a **virtual base class** to resolve ambiguity.

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

class Base {
public:
    void display() {
        cout << "Base class" << endl;
    }
};

class ClassA : virtual public Base {};
class ClassB : virtual public Base {};
class Derived : public ClassA, public ClassB {};

int main() {
    Derived obj;
    obj.display(); // Output: Base class
```

```
    return 0;
}
```

Conclusion

- **Operator Overloading**: Allows custom behavior for operators when used with objects.
- **Data Conversion**: Enables implicit or explicit conversion between objects and primitive types.
- **Inheritance**: Provides different mechanisms (single, multiple, multilevel, hybrid) to reuse and extend functionality.

3.5 Pure Virtual Functions and File Handling

This section introduces **pure virtual functions**, **dynamic binding**, and the fundamentals of **file input/output operations** using C++. The concept of **stream classes** is also explained to help manage file operations effectively.

---

1. Pure Virtual Functions

A **pure virtual function** is a function declared within a base class that has no definition relative to the base class. It enforces the concept of abstraction and makes a class abstract.

- A pure virtual function is declared using the syntax:

  ```
  virtual return_type function_name() = 0;
  ```

- Any class containing at least one pure virtual function is called an **abstract class**, and objects of such classes cannot be instantiated.

Example: Pure virtual function and dynamic binding.

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

// Abstract base class
class Shape {
public:
    virtual void draw() = 0; // Pure virtual function
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing a Circle" << endl;
    }
};
```

```cpp
class Rectangle : public Shape {
public:
    void draw() override {
        cout << "Drawing a Rectangle" << endl;
    }
};

int main() {
    Shape* shape1 = new Circle();
    Shape* shape2 = new Rectangle();

    shape1->draw(); // Output: Drawing a Circle (Dynamic Binding)
    shape2->draw(); // Output: Drawing a Rectangle (Dynamic Binding)

    delete shape1;
    delete shape2;
    return 0;
}
```

---

2. Dynamic Binding

Dynamic binding ensures that the correct method is called for a derived class object when the object is accessed through a pointer or reference to the base class. This is achieved using **virtual functions**.

- It allows **runtime polymorphism**.
- Ensures that the appropriate overridden function is called for an object.

---

3. File Input/Output Operations

C++ provides the **<fstream>** library for file handling. The key classes for file I/O are:

- `ifstream`: For input (reading from a file).
- `ofstream`: For output (writing to a file).
- `fstream`: For both input and output.

File Write Example

```cpp
#include <iostream>
#include <ofstream> // File stream library
using namespace std;

int main() {
    ofstream outFile("example.txt"); // Open file for writing
    if (!outFile) {
        cout << "Error opening file!" << endl;
        return 1;
```

```
    }

    outFile << "Hello, File Handling in C++!" << endl;
    outFile << "This is a test file." << endl;

    outFile.close(); // Close file after writing
    cout << "Data written to file successfully!" << endl;

    return 0;
}
```

The file contains:

```
Hello, File Handling in C++!
This is a test file.
```

File Read Example

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

int main() {
    ifstream inFile("example.txt"); // Open file for reading
    if (!inFile) {
        cout << "Error opening file!" << endl;
        return 1;
    }

    string line;
    while (getline(inFile, line)) { // Read line by line
        cout << line << endl;
    }

    inFile.close(); // Close file after reading
    return 0;
}
```

The file contains:

```
Hello, File Handling in C++!
This is a test file.
```

File Append Example

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

```cpp
int main() {
    ofstream outFile("example.txt", ios::app); // Open file in append mode
    if (!outFile) {
        cout << "Error opening file!" << endl;
        return 1;
    }

    outFile << "Appending more data to the file." << endl;

    outFile.close();
    cout << "Data appended successfully!" << endl;

    return 0;
}
```

The file now contains:

```
Hello, File Handling in C++!
This is a test file.
Appending more data to the file.
```

---

4. Stream Class Hierarchy

C++ provides a hierarchy of stream classes to manage input and output operations:

```
ios_base
   |
   +--> ios
         |
         +--> istream (input stream)
         |      |
         |      +--> ifstream (file input stream)
         |
         +--> ostream (output stream)
                |
                +--> ofstream (file output stream)
                |
                +--> iostream (input/output stream)
                       |
                       +--> fstream (file input/output stream)
```

Conclusion

- **Pure Virtual Functions**:
  - Declared with `= 0` in the base class.
  - Makes a class abstract and ensures derived classes implement specific functionality.
- **Dynamic Binding**: Ensures runtime polymorphism through virtual functions.

- **File Handling**:
  - **ifstream**, **ofstream**, and **fstream** are key classes for file I/O.
  - Files can be opened in modes such as `ios::in`, `ios::out`, or `ios::app`.
- **Stream Class Hierarchy**: Defines how different streams (standard and file-based) interact.

3.6 Generic Programming and Exception Handling

This section explores **generic programming** in C++, which allows writing flexible, reusable code using **templates**. We also discuss **exception handling**, which is used to handle errors gracefully using **try**, **catch**, and **throw** mechanisms.

---

1. Function Templates

A **function template** allows you to define a function that works with any data type. It acts as a blueprint for creating functions that can operate on different data types.

- **Syntax**:

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

Example: Using function templates.

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

template <typename T> // Function template for any type T
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << add(5, 10) << endl;        // Output: 15 (int)
    cout << add(3.5, 2.5) << endl;     // Output: 6.0 (double)
    cout << add('A', 'B') << endl;     // Output: 131 (char)
    return 0;
}
```

Output:

```
15
6
131
```

---

2. Class Templates

A **class template** enables you to define a class that can operate with any data type.

- **Syntax**:

```
template <typename T>
class MyClass {
    T data;
public:
    MyClass(T value) : data(value) {}
    T getData() { return data; }
};
```

Example: Using class templates.

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

template <typename T> // Class template
class MyClass {
    T data;
public:
    MyClass(T value) : data(value) {}
    T getData() { return data; }
};

int main() {
    MyClass<int> obj1(100);          // T is int
    MyClass<double> obj2(3.14);      // T is double

    cout << obj1.getData() << endl;  // Output: 100
    cout << obj2.getData() << endl;  // Output: 3.14

    return 0;
}
```

Output:

```
100
3.14
```

---

3. Standard Template Library (STL)

C++ provides the **Standard Template Library (STL)**, which is a collection of template classes and functions. It provides useful data structures and algorithms, making it easier to work with collections of data.

Containers

Containers are used to store collections of data. Some common types are:

- **Vector**: Dynamic array.
- **List**: Doubly linked list.
- **Queue**: FIFO structure.
- **Stack**: LIFO structure.
- **Map**: Key-value pair collection.

Example: Using a `vector` (dynamic array) container.

```
#include <iostream>
#include <vector>  // Include vector header
using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5}; // Initialize vector with values

    // Iterating through vector
    for (int i = 0; i < vec.size(); i++) {
        cout << vec[i] << " ";  // Output: 1 2 3 4 5
    }
    cout << endl;

    vec.push_back(6); // Adding element to the vector
    cout << "Last element: " << vec.back() << endl;  // Output: 6

    return 0;
}
```

Output:

```
1 2 3 4 5
Last element: 6
```

Algorithms

The STL provides useful algorithms for operations like sorting, searching, and manipulating data structures.

Example: Using the `sort` algorithm to sort a vector.

```
#include <iostream>
#include <vector>
#include <algorithm> // Include for sorting algorithm
using namespace std;

int main() {
    vector<int> vec = {5, 3, 8, 1, 2};

    // Sorting the vector
    sort(vec.begin(), vec.end());  // Sorting in ascending order

    // Display sorted vector
    for (int i = 0; i < vec.size(); i++) {
```

```
        cout << vec[i] << " ";  // Output: 1 2 3 5 8
    }
    cout << endl;

    return 0;
}
```

Output:

```
1 2 3 5 8
```

Iterators

Iterators are used to traverse through the elements of a container.

Example: Using iterators to traverse a vector.

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

int main() {
    vector<int> vec = {10, 20, 30, 40};

    // Using iterator to access vector elements
    for (auto it = vec.begin(); it != vec.end(); ++it) {
        cout << *it << " ";  // Output: 10 20 30 40
    }
    cout << endl;

    return 0;
}
```

Output:

```
10 20 30 40
```

---

4. Exception Handling

Exception handling in C++ helps to manage errors during program execution using the `try`, `catch`, and `throw` mechanisms.

- **try**: A block of code that might throw an exception.
- **catch**: A block of code that handles the exception.
- **throw**: Used to throw an exception.

Syntax:

```
try {
    // Code that may throw an exception
}
catch (exception_type e) {
```

```
    // Code to handle the exception
}
```

Example: Basic Exception Handling

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

int divide(int a, int b) {
    if (b == 0) {
        throw "Division by zero error!";  // Throwing an exception
    }
    return a / b;
}

int main() {
    try {
        int result = divide(10, 0);  // This will throw an exception
        cout << "Result: " << result << endl;
    }
    catch (const char* msg) {  // Catching the exception
        cout << "Error: " << msg << endl;  // Output: Error: Division by zero error!
    }

    return 0;
}
```

Output:

```
Error: Division by zero error!
```

Example: Multiple Exceptions

You can have multiple `catch` blocks to handle different exceptions.

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

int divide(int a, int b) {
    if (b == 0) {
        throw "Division by zero error!";
    }
    if (a < 0 || b < 0) {
        throw "Negative number error!";
    }
    return a / b;
}

int main() {
    try {
        int result = divide(-10, 0);  // This will throw a negative number error
```

```
        cout << "Result: " << result << endl;
    }
    catch (const char* msg) {
        cout << "Error: " << msg << endl;  // Output: Error: Negative number error!
    }

    return 0;
}
```

Output:

```
Error: Negative number error!
```

Conclusion

- **Function Templates**: Allow you to create generic functions that can work with any data type.
- **Class Templates**: Allow you to define classes that work with different data types, providing flexibility and reusability.
- **STL**:
  - **Containers**: Store collections of data (e.g., `vector`, `list`, `map`).
  - **Algorithms**: Predefined functions like `sort()`, `find()`, `reverse()` to operate on data in containers.
  - **Iterators**: Used to traverse containers in a generic way.
- **Exception Handling**: Handles errors using `try`, `catch`, and `throw`. It ensures that your program can recover gracefully from runtime errors, including handling multiple types of exceptions.