Modern Programming Principles & Practice

Feb-May 2025

sruthi.padathara@dorset.ie

Contents

- Template Definition
- Generic Function
- Generic Class
- Template function Overloading
- Streams Hierarchy
- Input Streams & Output Streams
- What is File?
- Implementing various file operations on basic data types(write, read, append and modify)
- Implementing various file operations on object data types(write, read, append and
- modify)
- Random Access Files(seekp, seekg, tellp, tellg)

Template

Templates allow writing **generic** and **reusable** code that works with any data type. They are mainly used for **functions** and **classes**.

What is a Template?

A **template** is a blueprint that allows the creation of functions or classes that work with any data type.

Why Use Templates?

Code reusability – Write one function/class for multiple data types.
Type safety – Ensures type correctness at compile time.
Efficiency – Reduces code duplication.

When to Use Templates?

When you need **a single function/class** to work with multiple data types.
When **reducing code duplication** improves maintainability.
When you need **compile-time type checking** instead of runtime polymorphism.

How to Use Templates?

Use `template<typename T>` before defining a function or class.
Replace specific data types with `T` (or any identifier).
The compiler automatically generates the required code based on the type used.

Generic Function (Function Template)

What is a Generic Function?

A **generic function** (or function template) is a function that can operate on **different data types** without rewriting the code.

Why Use Generic Functions?

Reduces **code duplication**
Improves **readability & maintainability**
Allows **type safety** while handling multiple types

When to Use Generic Functions?

When performing **similar operations** on different types (e.g., sorting, searching).
When you need a **common function** for multiple data types.

Example of a Generic Function

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

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

int main() {
    cout << add(5, 10) << endl;       // Works for int
    cout << add(3.5, 2.5) << endl;    // Works for double
    return 0;
}
```

Generic Class (Class Template)

What is a Generic Class?

A **generic class** (or class template) allows defining a class that can **store/process multiple types**.

Why Use a Generic Class?

Reusable data structures (like stack, queue, linked list)
Flexibility – Works with any data type
Reduces code duplication

When to Use a Generic Class?

When you need **a container or data structure** that works with different types.
When you want to **generalize a class** without specifying a fixed type.

How to Use a Generic Class?

Declare **template<typename T>** before class definition.
Use **T** as a placeholder for a data type.
Create objects with **specific data types**.

Example of a Generic Class

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

template <typename T>
class Box {
private:
    T value;
public:
    Box(T v) : value(v) {}
    void show() { cout << "Value: " << value << endl; }
};

int main() {
    Box<int> intBox(10);
    Box<double> doubleBox(5.5);

    intBox.show();
    doubleBox.show();

    return 0;
}
```

Template Function Overloading

What is Template Function Overloading?

Template function overloading means **defining multiple template functions** with different parameters.

Why Use Template Function Overloading?

Allows **different versions** of the function for different use cases.
Provides **specific implementations** along with generic ones.
Makes the program **more flexible**.

When to Use Template Function Overloading?

When a **specific data type** needs a different implementation than the generic one.
When you need **multiple versions** of a function template.

How to Use Template Function Overloading?

Define **multiple template functions** with different signatures.
Provide **specialized versions** for specific types.
The compiler selects the best-matching function.

Example of Template Function Overloading

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

template <typename T>
void print(T value) {  // Generic function
    cout << "Generic: " << value << endl;
}

// Overloaded function for int
void print(int value) {
    cout << "Integer: " << value << endl;
}

int main() {
    print(10);       // Calls overloaded function for int
    print(3.5);      // Calls generic function for double
    print("Hello");  // Calls generic function for string

    return 0;
}
```

Summary

Concept	What	Why	When	How
Template	Blueprint for generic functions/classes	Code reusability, type safety	When working with multiple types	template<typename T>
Generic Function	A function that works with any data type	Reduces duplication, type safety	When a function performs similar operations on different types	template<typename T> T functionName(T param);
Generic Class	A class that works with any data type	Reusable data structures, flexibility	When creating data structures like stack, queue, list	template<typename T> class ClassName { ... }
Template Overloading	Multiple template functions with different parameters	Flexibility, specialization	When different data types need different implementations	Define multiple template functions

Function Template Specialization

What is Function Template Specialization?

Function template specialization allows defining a **specific implementation** of a **generic function** for a particular data type.

- Normally, function templates are **generic** and work with **any data type**.
- However, sometimes **specific data types require special handling**.
- **Template specialization** provides a way to define **a custom implementation** for **specific types** while still keeping the generic version.

Why Use Function Template Specialization?

When a **specific data type** needs a different implementation than the generic template.
To **optimize performance** for a particular data type.
To **handle edge cases** that the generic template may not cover.

When to Use Function Template Specialization?

When **one or more data types** need a different implementation.
When a **specific operation** is more efficient for a particular type.
When a **special formatting** or behavior is required for a certain type.

How to Use Function Template Specialization?

Define a **generic template function** using `template<typename T>`.
Provide a **specialized version** using `template<>` with a **specific type**.
The compiler will **automatically select** the specialized version when the specific type is used.

Summary

1. **The generic template handles all types** unless a specialization exists.
2. **The specialized version overrides** the generic version **for the specified type**.
3. **We use template<> syntax** for specialization.
4. The compiler automatically chooses the **specialized function** when the specialized type is used

Example of Function Template Specialization

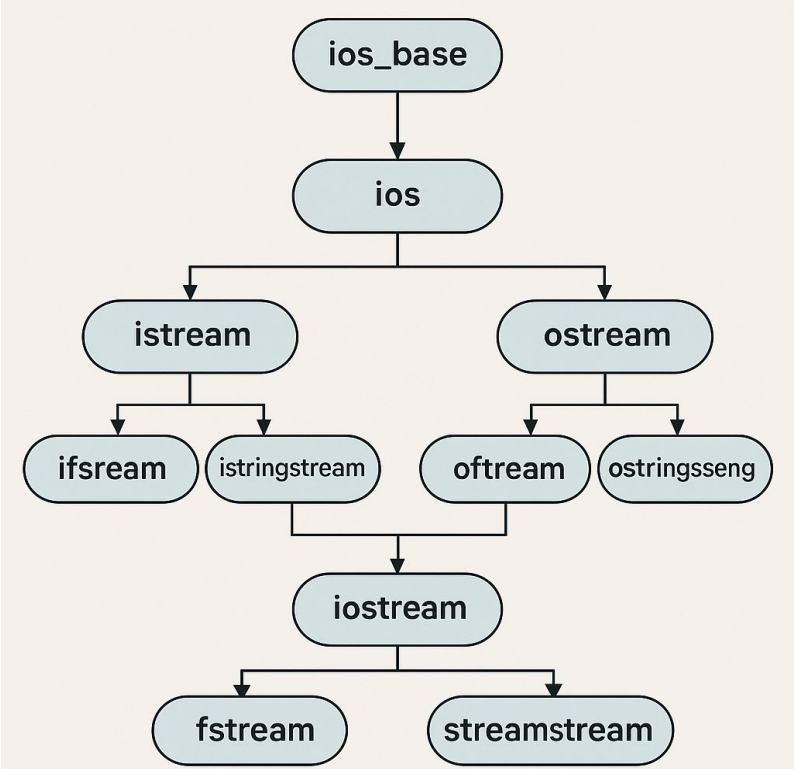
```
#include <iostream>
using namespace std;

// Generic template function
template <typename T>
void display(T value) {
    cout << "Generic display: " << value << endl;
}

// Specialization for string
template <>
void display<string>(string value) {
    cout << "Specialized display for string: " << value << endl;
}

int main() {
    display(10);         // Calls generic template function
    display(3.14);       // Calls generic template function
    display("Hello!");   // Calls specialized function for string

    return 0;}
```

I/O Streams & Files in C++

C++ provides a hierarchy of **input and output (I/O) streams** to manage different types of input and output operations. These streams are part of the **iostream** library.

Input Streams & Output Streams

- **Input Streams (`istream`)**: Used to read data from a source (e.g., keyboard, file).
- **Output Streams (`ostream`)**: Used to write data to a destination (e.g., console, file).



ios (Base Class)

- **ios is the base class for all input and output stream classes** in C++.
- It provides fundamental operations like formatting, error handling, and state control for all stream classes.
- Every stream class in C++ inherits from ios directly or indirectly.

istream (Input Stream)

- Handles **input operations** (reading data).
- It is used for reading data from **keyboard (cin)** or **files (ifstream)**.
- **Example: Reading from cin (standard input)**

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

int main() {
    int num;
    cout << "Enter a number: ";
    cin >> num;  // Using istream (cin)
    cout << "You entered: " << num <<
endl;
    return 0;
}
```

ifstream (File Input Stream)

- A subclass of istream used for **reading from files**.
- Used to **open, read, and close** a file.

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

int main() {
    ifstream inputFile("data.txt");  // Open file
    string line;
    if (inputFile.is_open()) {
        while (getline(inputFile, line)) {  // Read file line by line
            cout << line << endl;
        }
        inputFile.close();  // Close file
    } else {
        cout << "Error opening file!";
    }
    return 0;
}
```

istringstream (String Input Stream)

- A subclass of istream used for **reading data from a string** instead of an external source (e.g., keyboard or file).
- **Example: Extracting values from a string**

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

int main() {
    string data = "100 200 300";
    istringstream inputStream(data);  // Initialize string stream

    int num1, num2, num3;
    inputStream >> num1 >> num2 >> num3;  // Extract values from the string

    cout << "Extracted Numbers: " << num1 << ", " << num2 << ", " << num3 << endl;
    return 0;
}
```

ostream (Output Stream)

- Handles **output operations** (writing data).
- It is used for writing data to **console (cout)** or **files (ofstream)**.
- **Example: Writing to cout (standard output)**

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

int main() {
    cout << "Hello, World!" << endl;  // Using ostream (cout)
    return 0;
}
```

ostringstream (String Output Stream)

- A subclass of `ostream` used for **writing data to a string** instead of an external destination (e.g., console or file).
- **Example: Writing formatted data to a string**

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

int main() {
    ostringstream outputStream;  // Initialize string stream
    outputStream << "Age: " << 25 << ", Score: " << 90.5;

    string result = outputStream.str();  // Convert stream to string
    cout << "Formatted Output: " << result << endl;
    return 0;
}
```

iostream (Input-Output Stream)

- A combination of istream and ostream used for **both reading and writing**.
- Example classes that use iostream:
  - **cin** (standard input)
  - **cout** (standard output)
  - **fstream** (file input-output stream)
- **Example: Using cin and cout together**

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

int main() {
    string name;
    cout << "Enter your name: ";
    cin >> name;  // Using istream
    cout << "Hello, " << name << "!" << endl;  // Using ostream
    return 0;
}
```

fstream (File Input-Output Stream)

- A subclass of iostream used for **both reading from and writing to files**.
- It supports operations like **reading, writing, appending, modifying**.
- **Example: Reading and Writing to a File**

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

int main() {
    fstream file("example.txt", ios::in | ios::out | ios::trunc);  // Open file for input-output
    file << "Hello, File I/O!" << endl;  // Write data
    file.seekg(0);  // Move read pointer to the beginning

    string content;
    getline(file, content);  // Read data
    cout << "File Content: " << content << endl;

    file.close();
    return 0;
}
```

stringstream (String Input-Output Stream)

- A subclass of iostream that allows both **reading and writing to a string**.
- **Example: Using stringstream for formatted input-output**

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

int main() {
    stringstream ss;  // Create stringstream object
    ss << "C++ " << 2024;  // Writing to stringstream

    string text;
    int year;
    ss >> text >> year;  // Reading from stringstream

    cout << "Extracted Text: " << text << ", Year: " << year << endl;
    return 0;
}
```

What is a File?

- A **file** is a storage unit used to **store, retrieve, and manipulate** data permanently.
- In C++, we use the **fstream** library to handle file operations.

Implementing File Operations on Basic Data Types

- Write to a File
- Read from a File
- Append to a File
- Modify a File

Example: Writing & Reading Basic Data Types

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    // Writing data to a file
    ofstream outFile("data.txt");
    outFile << "Hello, File Handling in C++!" << endl;
    outFile.close();

    // Reading data from a file
    ifstream inFile("data.txt");
    string line;
    while (getline(inFile, line)) {
        cout << line << endl;
    }
    inFile.close();
    return 0;
}
```

Implementing File Operations on Object Data Types

- Writing objects to a file
- Reading objects from a file
- Appending objects to a file
- Modifying objects in a file

Example: Writing & Reading an Object

```
int main() {
    Student s;
    s.getData();
    // Writing object data to file
    ofstream outFile("student.dat", ios::binary);
    outFile.write((char*)&s, sizeof(s));
    outFile.close();
    // Reading object data from file
    Student s2;
    ifstream inFile("student.dat", ios::binary);
    inFile.read((char*)&s2, sizeof(s2));
    s2.displayData();
    inFile.close();
    return 0;
}
```

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

class Student {
public:
    string name;
    int age;

    void getData() {
        cout << "Enter Name: ";
        cin >> name;
        cout << "Enter Age: ";
        cin >> age;
    }

    void displayData() {
        cout << "Name: " << name << ", Age: " << age <<
endl;
    }
};
```

Random Access Files (seekp, seekg, tellp, tellg)

`seekp(pos)` – Move put pointer (write) to `pos` in file.
`seekg(pos)` – Move get pointer (read) to `pos` in file.
`tellp()` – Get current position of put pointer.
`tellg()` – Get current position of get pointer.

Example: Using Random Access Functions ->

Summary

- **Streams** provide a flexible way to handle input and output operations.
- **File handling** allows **storing and retrieving data** permanently.
- **Basic file operations** include writing, reading, appending, and modifying.
- **Objects can be stored in files** using binary file handling.
- **Random access functions** help in **manipulating specific positions** in a file.

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

int main() {
    fstream file("random.txt", ios::out | ios::in | ios::trunc);
    file << "Hello, this is C++ file handling!";

    // Move read pointer to position 7
    file.seekg(7);

    // Read and display from that position
    string str;
    file >> str;
    cout << "Read from position 7: " << str << endl;

    // Get the current write position
    cout << "Current write position: " << file.tellp() << endl;

    file.close();
    return 0;
}
```