UNIT IV – TEMPLATES AND FILE OPERATIONS

Templates, Function templates, Class Templates, Overloading of Template Functions, Member function Templates, **File** stream operations – ostream, istream, fstream, File Access Modes, File position pointer – seekg, seekp, tellg, tellp, Sequential Input and Output operations, Random Access, Error handling during file operation.

CO4: Make use of files and Templates to develop C++ programs. (K3)

4.1 Templates: Function templates, Class Templates

A **template** is a mechanism in C++ that allows:

- Writing **generic programs**
- Creating **functions or classes** that work with **any data type**

Templates support **generic programming**.

Definition: A template is a blueprint that enables a function or class to operate with different data types without rewriting the code.

Why Templates are Needed?

Templates help to:

- Avoid code duplication
- Improve code reusability
- Reduce program size
- Increase flexibility
- Support type-safe generic programming

Types of Templates

C++ supports two main types of templates:

1. Function Templates
2. Class Templates

Function Templates

A **function template**:

- Defines a **generic function**
- Works with different data types
- Compiler generates required functions automatically

Syntax

```
template <class T>
return_type function_name(T variable) {
    // function body
}
```

Example

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

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

int main() {
    cout << add(10, 20) << endl;       // int
    cout << add(2.5, 3.5) << endl;     // float
    return 0;
}
```

Advantages of Function Templates

- Eliminates function overloading
- Ensures type safety
- Reduces code repetition
- Easy to maintain

Class Templates

A class template:

- Defines a generic class

- Allows data members and member functions to work with different data types

Syntax

```
template <class T>
class ClassName {
    T data;
public:
    void setData(T value);
    T getData();
};
```

Example

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

template <class T>
class Sample {
    T data;
public:
    void setData(T value) {
        data = value;
    }

    T getData() {
        return data;
    }
};

int main() {
```

```
    Sample<int> s1;
    Sample<float> s2;

    s1.setData(10);
    s2.setData(25.5);

    cout << s1.getData() << endl;
    cout << s2.getData() << endl;

    return 0;
}
```

******

4.2 Overloading of Template Functions

Defining **more than one function with the same name**, where at least one of them is a **function template** and they differ in parameters.

The compiler selects the **most suitable function** based on the arguments.

Why Template Function Overloading is Needed?

- To handle special cases
- To give custom behavior for certain data types
- To combine generic functions and specific functions
- To improve program flexibility

Types of Template Function Overloading

Template functions can be overloaded with:

1. Another template function
2. A non-template (normal) function

Overloading Template Function with a Normal Function

Example Program

```
#include <iostream>
```

```
using namespace std;

template <class T>
void display(T x) {
    cout << "Template function: " << x << endl;
}

void display(int x) {
    cout << "Normal function: " << x << endl;
}

int main() {
    display(10);      // calls normal function
    display(10.5);    // calls template function
    return 0;
}
```

Overloading Template Functions with Another Template

Example Program

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

template <class T>
void show(T a) {
    cout << "One argument: " << a << endl;
}

template <class T>
void show(T a, T b) {
```

```
    cout << "Two arguments: " << a << " " << b << endl;
}

int main() {
    show(10);
    show(10, 20);
    return 0;
}
```

Function Selection Rule (Important)

When calling overloaded functions:

1. Exact match is preferred
2. Normal function has higher priority than template
3. Template function is selected if no exact normal match exists

******

4.3 Member function Templates

A **member function template** is:

- A template defined inside a class
- Allows a member function to work with different data types
- Used when the class is not generic, but the function is

A member function template is a template function that is defined as a member of a class.

Why Member Function Templates Are Needed?

Member function templates help to:

- Provide flexibility in class design
- Avoid making the entire class a template
- Reuse the same function logic for different data types
- Reduce code duplication

Syntax

```
class ClassName {
```

```
public:
    template <class T>
    return_type functionName(T value) {
        // function body
    }
};
```

Example

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

class Sample {
public:
    template <class T>
    void display(T value) {
        cout << "Value: " << value << endl;
    }
};

int main() {
    Sample obj;

    obj.display(10);       // int
    obj.display(25.5);     // float
    obj.display("Hello");  // string

    return 0;
}
```

Difference: Class Template vs Member Function Template

Feature	Class Template	Member Function Template
Template applied to	Entire class	Only a function
Class becomes generic	Yes	No
Syntax	`template<class T> class`	`template<class T> return_type func()`
Flexibility	Less	More

******

4.4 File stream operations – ostream, istream, fstream

File handling in C++ is used to:

- Store data permanently
- Read data from files
- Write data to files
- Process large amounts of data

C++ uses **stream-based file handling**.

A **stream** is a flow of data from:

- Program → file (output)
- File → program (input)

File Stream Classes

C++ provides **three main file stream classes**:

Stream Class	Purpose
`istream`	Input stream
`ostream`	Output stream
`fstream`	Input and Output stream

These classes are defined in the header file:

```
#include <fstream>
```

ostream – Output Stream

Purpose

- Used to **write data to a file**
- Supports output operations

Example: Writing to a File using ofstream (derived from ostream)

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

int main() {
    ofstream fout;
    fout.open("data.txt");

    fout << "Welcome to File Handling";
    fout.close();

    return 0;
}
```

istream – Input Stream

Purpose

- Used to **read data from a file**
- Supports input operations

Example: Reading from a File using ifstream (derived from istream)

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

int main() {
    ifstream fin;
    fin.open("data.txt");

    char ch;
```

```
    while (fin.get(ch)) {
        cout << ch;
    }

    fin.close();

    return 0;
}
```

fstream – Input and Output Stream

Purpose

- Used to read and write data using the same file object

Example: Using fstream

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

int main() {
    fstream file;
    file.open("sample.txt", ios::out);

    file << "File Stream Example";
    file.close();

    return 0;
}
```

******

4.5 File Access Modes

File access modes specify:

- Whether a file is opened for **reading**, **writing**, or **both**
- Whether data is **overwritten** or **appended**
- How the file pointer behaves

Access modes are provided using **ios flags**.

Syntax for Opening a File with Access Mode

```
file_object.open("filename", mode);
```

Example

```
file.open("data.txt", ios::out);
```

Common File Access Modes

Access Mode	Meaning
`ios::in`	Open file for reading
`ios::out`	Open file for writing
`ios::app`	Append data at end of file
`ios::ate`	Move file pointer to end
`ios::trunc`	Delete existing file content
`ios::binary`	Open file in binary mode

ios::in – Read Mode

Purpose

- Opens a file for **input operations**
- File must exist

Example

```
ifstream fin;
fin.open("data.txt", ios::in);
```

ios::out – Write Mode

Purpose

- Opens a file for **output operations**
- Creates a new file if it does not exist
- Deletes old content by default

Example

```
ofstream fout;
fout.open("data.txt", ios::out);
```

ios::app – Append Mode

Purpose

- Appends data at the **end of the file**
- Existing data is preserved

Example

```
ofstream fout;
fout.open("data.txt", ios::app);
```

ios::ate – At End Mode

Purpose

- File pointer moves to the **end initially**
- Reading and writing can happen anywhere

Example

```
fstream file;
file.open("data.txt", ios::ate);
```

ios::trunc – Truncate Mode

Purpose

- Deletes all existing content of the file
- File opens as empty

Example

```
ofstream fout;
fout.open("data.txt", ios::trunc);
```

ios::binary – Binary Mode

Purpose

- Used for **non-text files** (images, audio, etc.)
- Data is stored in binary format

Example

```
ofstream fout;
fout.open("data.bin", ios::binary);
```

Combining File Access Modes

Multiple modes can be combined using **bitwise OR (|)**.

Example

```
fstream file;
file.open("data.txt", ios::in | ios::out);
```

******

4.6 File Position Pointer - seekg, seekp, tellg, tellp

A **file position pointer**:

- Indicates the current location in a file
- Determines from where data is read or written
- Moves automatically or manually during file operations

C++ maintains separate pointers for:

- Reading
- Writing

Types of File Position Pointers

Pointer	Purpose
Get pointer	Used for reading
Put pointer	Used for writing

tellg() – Get Position (Input)

Purpose

- Returns the **current position of the get pointer**
- Used with input streams

Syntax

```
position = file.tellg();
```

Example

```
ifstream fin("data.txt");
cout << fin.tellg();
```

tellp() – Put Position (Output)

Purpose

- Returns the current position of the put pointer
- Used with output streams

Syntax

```
position = file.tellp();
```

Example

```
ofstream fout("data.txt");
cout << fout.tellp();
```

seekg() – Move Get Pointer

Purpose

- Moves the read pointer to a specified position
- Used in input operations

Syntax

```
file.seekg(offset, direction);
```

Direction Options

Direction	Meaning
`ios::beg`	Beginning of file
`ios::cur`	Current position
`ios::end`	End of file

Example

```
ifstream fin("data.txt");
fin.seekg(5, ios::beg);
```

seekp() – Move Put Pointer

Purpose

- Moves the write pointer to a specified position
- Used in output operations

Syntax

```
file.seekp(offset, direction);
```

Example

```
fstream file("data.txt", ios::in | ios::out);
file.seekp(10, ios::beg);
```

Simple Program Demonstrating All Functions

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

int main() {
    fstream file("sample.txt", ios::in | ios::out);

    file << "Hello World";

    cout << "Put pointer position: " << file.tellp() << endl;

    file.seekg(0, ios::beg);
    cout << "Get pointer position: " << file.tellg() << endl;

    char ch;
    file.get(ch);
    cout << "Character read: " << ch << endl;

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

******

4.7 Sequential Input and Output operations

Sequential file operations mean:

- Data is accessed **one after another**
- No jumping to random locations
- File pointer moves **automatically** after each read/write

This is the **simplest and most commonly used** file access method.

Characteristics of Sequential Access

- Data is processed in **fixed order**
- Suitable for **text files**
- Easy to implement
- Slower than random access for large files
- File pointer moves **forward only**

Sequential Output Operation (Writing to File)

Purpose

- Write data to a file from beginning to end
- Data is stored in the order it is written

Example: Sequential Writing

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

int main() {
    ofstream fout("data.txt");

    fout << "C++ Programming\n";
    fout << "File Handling\n";
    fout << "Sequential Output\n";
```

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

Sequential Input Operation (Reading from File)

Purpose

- Read data from a file **in sequence**
- Starts from the beginning of the file

Example: Sequential Reading

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

int main() {
    ifstream fin("data.txt");
    string line;

    while (getline(fin, line)) {
        cout << line << endl;
    }

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

Sequential I/O Using fstream

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

```
int main() {
    fstream file("sample.txt", ios::out);

    file << "Sequential I/O Example";
    file.close();

    file.open("sample.txt", ios::in);
    char ch;

    while (file.get(ch)) {
        cout << ch;
    }

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

Comparison: Sequential vs Random Access

Feature	Sequential Access	Random Access
Access order	One by one	Any position
Pointer movement	Automatic	Manual
Complexity	Simple	Complex
Speed	Slower for large files	Faster for specific records

******

4.8 Random Access

Random access means:

- Data can be accessed **directly at any location**
- File pointer is moved **manually**
- Useful when working with **large files or records**

Unlike sequential access, data need **not** be processed in order.

Characteristics of Random Access

- File pointer movement is controlled by programmer
- Faster access to specific records
- Uses file position pointer functions
- Suitable for database-like applications

Functions Used in Random Access

Random access is achieved using:

- seekg() → move read pointer
- seekp() → move write pointer
- tellg() → get read position
- tellp() → get write position

Random Access for Reading Data

Purpose

- Read data from a specific position
- Skip unnecessary data

Example: Random Read

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

int main() {
    ifstream fin("data.txt");

    fin.seekg(6, ios::beg);   // move pointer to 7th character
    char ch;
    fin.get(ch);
```

```
    cout << "Character read: " << ch << endl;

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

Random Access for Writing Data

Purpose

- Write data at a specific location
- Modify existing file content

Example: Random Write

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

int main() {
    fstream file("data.txt", ios::in | ios::out);

    file.seekp(5, ios::beg);   // move pointer to 6th position
    file << "XYZ";

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

Random Access Using Records (Concept)

- Each record has a fixed size
- Record number is used to calculate position
- Common in student or employee databases

Position = record_number × record_size

Sequential vs Random Access

Feature	Sequential Access	Random Access
Access order	One by one	Any position
Pointer movement	Automatic	Manual
Speed	Slower for search	Faster for search
Complexity	Simple	Moderate

Advantages of Random Access

- Faster retrieval of data
- Efficient for large files
- Easy update of specific records

Limitations of Random Access

- More complex than sequential access
- Requires careful pointer management
- Not suitable for variable-length records

******

4.9 Error handling during file operation

Error handling during file operations, which ensures that file operations are performed safely and correctly.

Why Error Handling is Needed?

Error handling is required to:

- Check whether a file is successfully opened
- Detect read/write failures
- Prevent program crashes
- Ensure reliable file processing

Common file errors:

- File not found
- Permission denied
- End-of-file reached

- Read/write failure

Checking File Open Status

Using is_open()

```
ifstream fin("data.txt");


if (!fin.is_open()) {
    cout << "File cannot be opened" << endl;
}
```

Stream State Flags for Error Handling

C++ stream classes provide **status flags** to detect errors.

Flag	Meaning
`good()`	No error
`eof()`	End of file reached
`fail()`	Logical error in operation
`bad()`	Serious I/O error

good() – Check Normal State

```
if (file.good()) {
    cout << "File is in good state" << endl;
}
```

eof() – End of File Detection

```
while (!fin.eof()) {
    // read operation
}
```

Indicates no more data to read.

fail() – Logical Error Detection

```
if (file.fail()) {
    cout << "File operation failed" << endl;
}
```

Occurs due to wrong data type or format.

bad() – Serious Error Detection

```
if (file.bad()) {
    cout << "Serious I/O error occurred" << endl;
}
```

Indicates corruption or hardware-related issues.

Clearing Error Flags

Once an error occurs, the stream enters an error state.

Using clear()

```
file.clear();
```

Resets error flags and allows further operations.

Complete Example: Error Handling in File Operations

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

int main() {
    ifstream fin("data.txt");

    if (!fin) {
        cout << "Error opening file" << endl;
        return 0;
    }

    char ch;
    while (fin.get(ch)) {
        cout << ch;
    }
```

```
    if (fin.eof()) {
        cout << "\nEnd of file reached" << endl;
    }

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

******

List of Practice Programs (Total: 28)

A. Templates

Templates & Function Templates

1. Write a C++ program to demonstrate a simple function template for finding the maximum of two values.
2. Write a C++ program using a function template to perform addition of two numbers of different data types.
3. Write a C++ program to demonstrate a function template with multiple data types.

Class Templates

1. Write a C++ program to demonstrate a class template for storing and displaying a single value.
2. Write a C++ program using a class template to perform basic operations on different data types.

Overloading of Template Functions

1. Write a C++ program to demonstrate overloading of a template function with a normal function.
2. Write a C++ program to demonstrate overloading of template functions with different parameters.

Member Function Templates

1. **Write a C++ program to demonstrate a member function template inside a normal** class.
2. Write a C++ program to show the difference between class template and member function template.

B. File Stream Operations

File Stream Classes (istream, ostream, fstream)

1. Write a C++ program to write data to a file using ofstream.
2. Write a C++ program to read data from a file using ifstream.
3. Write a C++ program to read and write data using fstream.

C. File Access Modes

1. Write a C++ program to open a file using ios::out and write data.
2. Write a C++ program to demonstrate append mode (ios::app).
3. Write a C++ program to demonstrate truncate mode (ios::trunc).
4. Write a C++ program to open a file using multiple access modes.

D. File Position Pointer

seekg(), seekp(), tellg(), tellp()

1. Write a C++ program to demonstrate tellg() and tellp().
2. Write a C++ program to demonstrate seekg() to read data from a specific position.
3. Write a C++ program to demonstrate seekp() to write data at a specific position.

E. Sequential Input and Output Operations

1. Write a C++ program to perform sequential writing of multiple records into a file.
2. Write a C++ program to perform sequential reading of data from a file.

F. Random Access

1. Write a C++ program to demonstrate random access reading using file position pointers.
2. Write a C++ program to demonstrate random access writing in a file.
3. Write a C++ program to access a specific record using record number.

G. Error Handling During File Operation

1. Write a C++ program to check file open failure using is_open().
2. Write a C++ program to demonstrate eof() condition.
3. Write a C++ program to demonstrate fail() and bad() states.
4. Write a C++ program to demonstrate clearing error flags using clear().

□**********□

UNIT IV - QUESTION BANK

Part A (2 Marks Questions)

1. **Define template. (K1)**

   A template is a mechanism in C++ that allows writing generic functions or classes that work with different data types.

2. **State the need for templates. (K1)**

   Templates are used to avoid code duplication and improve reusability by writing generic programs.

3. **List the types of templates supported in C++. (K1)**

   Function templates and class templates.

4. **Discuss generic programming. (K2)**

   Generic programming is a method of writing programs that work with different data types using templates.

5. **Define function template. (K1)**

   A function template is a generic function that works with different data types using template parameters.

6. **Identify the role of template parameter. (K2)**

   The template parameter represents a data type that is decided at compile time.

7. **What is class template? (K1)**

   A class template is a blueprint for creating classes that work with different data types.

8. **Discuss the advantage of class templates. (K2)**

   Class templates allow the same class definition to be reused for multiple data types.

9. **Outline overloading of template functions. (K2)**

   Overloading of template functions is the process of defining multiple functions with the same name where at least one function is a template.

10. **Why normal functions have priority over template functions? (K1)**

    Normal functions have priority because the compiler prefers exact matches over template functions.

11. Define member function template. (K1)

A member function template is a template function that is defined inside a class.

12. Differentiate class template and member function template. (K2)

In a class template, the entire class is generic, whereas in a member function template, only the function is generic.

13. List the file stream classes in C++. (K1)

istream, ostream, and fstream.

14. Illustrate the purpose of fstream. (K2)

fstream is used for both input and output operations on files.

15. Give any four file access modes. (K1)

ios::in, ios::out, ios::app, and ios::trunc.

16. Describe the use of ios::app. (K2)

ios::app is used to add data at the end of a file without deleting existing content.

17. Define file position pointer. (K1)

A file position pointer indicates the current position of reading or writing in a file.

18. Outline the function of tellg(). (K2)

tellg() returns the current position of the get pointer in a file.

19. Define sequential file access. (K1)

Sequential file access is a method where data is read or written in order from the beginning to the end of the file.

20. Summarize one advantage of sequential access. (K2)

Sequential access is simple to implement and suitable for processing text files.

21. Define random access. (K1)

Random access allows data to be read or written from any position in a file.

22. Why random access is faster for large files? (K1)

Random access allows direct access to required data without reading the entire file.

23. Mention any two stream state flags used in file error handling. (K1)

eof() and fail().

24. Show the purpose of eof(). (K2)

eof() indicates that the end of the file has been reached.

25. Define error handling in file operations. (K1)

Error handling in file operations is the process of detecting and managing errors during file access.

Part B (16 Marks Questions)

1. **Explain** templates in C++ and their importance in generic programming. (K2)
2. **Describe** function templates and class templates in C++ with suitable examples. (K2)
3. **Differentiate** overloading of template functions and member function templates. (K2)
4. **Illustrate** the role of file stream classes istream, ostream, and fstream in file handling. (K2)
5. **Outline** the file position pointer functions seekg, seekp, tellg, and tellp. (K2)
6. **Construct** a C++ program using function templates to perform operations on different data types. (K3)
7. **Develop** a C++ program to demonstrate overloading of template functions. (K3)
8. **Experiment** with a C++ program to perform random access file operations using file position pointers. (K3)
9. **Investigate** sequential input and output operations in C++ by creating and reading a file. (K3)
10. **Solve** file handling errors in a C++ program using appropriate error handling techniques. (K3)

Part C (16 Marks Questions)

1. **Develop** a C++ program for a marks processing system where the same function is used to calculate total marks for students of different data types using function templates. (K3)
2. **Construct** a C++ application for a simple data storage system that uses class templates to store and display integer and floating-point values. (K3)
3. **Experiment** with a C++ program to manage a text file where data is written sequentially and later accessed randomly using file position pointer functions. (K3)

4. **Solve** a real-time file handling problem in C++ by applying appropriate file access modes and error handling techniques while reading and writing data to a file. (K3)

For 4th Question in Part-C: Think about the Below Scenario

Application Scenario

Consider a **Student Registration System** in an educational institution.

The system should:

- Store student details (Roll No, Name, Marks) in a file
- Append new student records without deleting old data
- Read and display all stored records
- Handle errors such as file not found or file open failure

This requires:

- **File access modes**
- **Sequential file operations**
- **Error handling during file operations**

Course In-charge	Module Coordinator	HoD/IT
Mrs. P. Saraswathi, AP-II	Mrs. M. Prabha, AP-II	Dr. R. Kavitha