



National University of Computer & Emerging Sciences, Karachi
Computer Science Department
Spring 2023, Lab Manual 15



Course Code: CL1004	Course : Object Oriented Programming Lab
Instructor(s) :	Shazia Paras Shaikh

Contents

- Filing , I/O STREAM (formatted , unformatted)
- Binary write
- Files modes
- Templates , template function , template classes.

FILING

- Filing in C++ refers to the process of reading from or writing to files.
- Files are used to store data that can be accessed by a program.
- C++ provides a set of file stream classes to perform file operations.
- The ofstream class is used to create and write to files.
- The ifstream class is used to read from files.
- The fstream class can be used for both reading and writing to files.
- Before using a file stream object, you need to open a file using the open() function and close the file using the close() function when you are done.
- File I/O operations include reading from a file using the >> operator, writing to a file using the << operator, and using functions like getline() to read a line of text from a file.

I/O STREAM (FORMATTED , UNFORMATTED)

- In C++, input and output operations are performed using I/O streams.
- I/O streams are a way of abstracting input and output devices, such as the console or files, as streams of characters.
- C++ supports both formatted and unformatted I/O streams.
- Formatted I/O streams format the output data according to a specified format, such as decimal or hexadecimal.
- Formatted output is achieved using the insertion operator << to write to a stream and formatted input is achieved using the extraction operator >> to read from a stream.
- Examples of formatted I/O streams include cout and cin.

- Unformatted I/O streams do not format the output data in any way and simply write or read the data as-is.
- Unformatted output is achieved using the write() function to write to a stream, and unformatted input is achieved using the read() function to read from a stream.
- Examples of unformatted I/O streams include fstream and stringstream.
- It is important to properly handle errors that can occur during I/O operations, such as file not found or data corruption.

BINARY WRITE / OBJECT WRITE

Binary write, also known as object write, is a file I/O operation in C++ that allows you to write binary data, such as serialized objects or images, to a file. Unlike text mode, binary mode reads and writes data in its raw binary format, without any translation or formatting.

Here is an example of binary write in C++:

```
#include <fstream>
#include <iostream>
#include <string>
```

```
class Person {
public:
    std::string name;
    int age;
};
```

```
int main() {
    Person person;
    person.name = "John";
    person.age = 25;
```

```
    std::ofstream file("person.bin", std::ios::binary); //open the binary file
    file.write(reinterpret_cast<char*>(&person), sizeof(Person)); //reinterpret is used to cast the
    //person object to a char pointer
    file.close();
```

```
    std::cout << "File written successfully." << std::endl;
    return 0;
}
```

In this example, we define a Person class with a name and age field. We then create a Person object and set its name and age. Next, we create an ofstream object to open a binary file in write mode. We use the write() function to write the Person object to the file in its binary format. Finally, we close the file and print a message to the console.

Note that when writing binary data to a file, we need to use the reinterpret_cast operator to cast the Person object to a char pointer, which can be written to the file. We also need to specify the size of the Person object using the sizeof operator.

Binary write is a powerful tool for serializing objects and storing them in files, which can be loaded and deserialized at a later time. However, it is important to ensure that the data being written is in the correct binary format and that the data is being written to the correct location in the file.

FILE MODES

In C++, file modes **are used to specify the type of file operation** that you want to perform, such as **read, write, append, or binary mode**. Here are the different file modes in C++:

- ios::in: Read mode - allows reading from a file.
- ios::out: Write mode - allows writing to a file.
- ios::app: Append mode - allows writing to the end of a file.
- ios::trunc: Truncate mode - discards the contents of the file before opening it for writing.
- ios::binary: Binary mode - reads and writes data in binary format, without any translation or formatting.

```
#include <fstream>
#include <iostream>
#include <string>
```

```
int main() {
    std::ofstream myfile;
```

```
    // Read mode
    myfile.open("example.txt", std::ios::in);
```

```
    // Write mode
    myfile.open("example.txt", std::ios::out);
```

```
    // Append mode
    myfile.open("example.txt", std::ios::app);
```

```
// Truncate mode
myfile.open("example.txt", std::ios::trunc);

// Binary mode
myfile.open("example.bin", std::ios::binary);

myfile.close();

std::cout << "File operation successful." << std::endl;
return 0;
}
```

SEEK FUNCTION

In C++, **the seek function is used to move the read/write pointer to a specific location in a file.** The seek function is typically used in conjunction with file I/O operations, such as read and write, to navigate through the contents of a file.

The seek function is declared in the fstream library and has the following syntax:

```
streampos seekg(streampos pos); // infile.seekg(10); // set input pointer to position 10
streampos seekg(streamoff off, ios_base::seekdir way); // A positive value of off moves the
input position pointer forward, and a negative value moves it backward.
streampos seekp(streampos pos);
streampos seekp(streamoff off, ios_base::seekdir way);
```

Here is a brief summary of the parameters:

pos: the absolute position in the file to move the read/write pointer to.

off: the relative offset to move the read/write pointer.

way: the direction in which to move the read/write pointer, relative to the current position.

Here are the different seek directions:

ios_base::beg: move the read/write pointer relative to the beginning of the file.

ios_base::cur: move the read/write pointer relative to the current position.

ios_base::end: move the read/write pointer relative to the end of the file.

Here is an example of using the seek function in C++:

```
#include <iostream>
#include <fstream>
```

```
int main() {
```

```
std::fstream file("example.txt", std::ios::in | std::ios::out);
```

```
if (file.is_open()) {  
    file.seekp(10, std::ios_base::beg);  
    file << "Hello World!";  
    file.seekg(0, std::ios_base::beg);
```

```
    std::string line;  
    while (getline(file, line)) {  
        std::cout << line << std::endl;  
    }
```

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

```
return 0;  
}
```

In this example, we open a file in read/write mode using `std::fstream`. We then use the `seekp()` function to move the write pointer to the 10th byte in the file and write "Hello World!" to the file. We then use the `seekg()` function to move the read pointer to the beginning of the file. Finally, we read the contents of the file using the `getline()` function and print them to the console.

The seek function is a powerful tool for navigating through the contents of a file, allowing you to read, write, and modify data at specific locations in the file. It is important to use the appropriate seek direction and to properly close the file after performing the desired operations.

TELL FUNCTION

In C++, the `tellg()` and `tellp()` functions are used to return the current position of the read and write pointers, respectively, in a file. These functions are typically used in conjunction with the `seekg()` and `seekp()` functions to navigate through the contents of a file.

Here is a short example of using the `tellp()` function in C++:

```
#include <iostream>  
#include <fstream>
```

```
int main() {
```

```
std::ofstream file("example.txt", std::ios::out | std::ios::app);
```

```
if (file.is_open()) {  
    file << "Hello World!";  
    std::streampos position = file.tellp();  
    std::cout << "Current position: " << position << std::endl;  
    file.close();  
}  
  
return 0;  
}
```

GENERIC/ TEMPLATES

Generics is a concept that refers to the ability to write code that can be used with different data types. In C++, this is achieved through the use of templates, which are code structures that can be parameterized with one or more types.

In C++, **generics and templates are often used interchangeably because templates provide a way to write generic code. Templates allow you to write code that can work with different data types without having to write separate implementations for each data type.**

Here is an example of a combined use of generics and templates in C++:

```
#include <iostream>
```

```
#include <iostream>
```

```
template <typename T>  
void printValue(T value) {  
    std::cout << "Value: " << value << std::endl;  
}
```

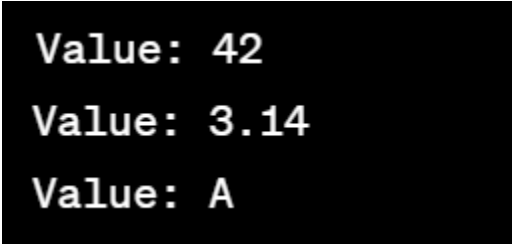
```
int main() {  
    int intValue = 42;  
    double doubleValue = 3.14;  
    char charValue = 'A';
```

```
    printValue<int>(intValue); // prints "Value: 42"  
    printValue<int>(doubleValue); // prints "Value: 3.14"  
    printValue<char>(charValue); // prints "Value: A"
```

```
    return 0;
}
```

In this example, we define a generic function called `printValue()` using templates. The typename `T` syntax declares a type parameter `T`, which can be any data type. The function takes a single parameter of type `T` and prints its value to the console.

In the `main()` function, we create three variables of different data types (`int`, `double`, and `char`) and pass each of them to the `printValue()` function. **Since `printValue()` is a template function, it can be used with any data type, and will print the value of the parameter passed to it.**



```
Value: 42
Value: 3.14
Value: A
```

TEMPLATE FUNCTION

A template function is a function that is defined using a template parameter, which can represent any data type. The template parameter is specified using the typename keyword, followed by a name that is used within the function definition to represent the actual data type. Here's the basic syntax **for defining a template function:**

```
template <typename T>
void functionName(T parameter) {
    // function body
}
```

In this example, `functionName` is the name of the function, and `T` is the template parameter. The typename keyword specifies that `T` represents a data type, and the function definition can use `T` to declare variables, parameters, and return values of that data type.

To call a template function, you specify the actual data type as the template argument when you call the function.

For example:

```
functionName<int>(42); // calls functionName with an int parameter
functionName<double>(3.14); // calls functionName with a double parameter
```

In this example, `int` and `double` are the template arguments that specify the actual data types for the `T` template parameter.

Template functions are useful when you need to write a function that can work with different data types, without having to write separate implementations for each data type. Templates allow you to write generic code that can be reused with different data types, which can **save time and reduce code duplication**.

A template function in C++ that finds the maximum element in an array of any data type:

```
#include <iostream>
```

```
template <typename T>
T maxElement(T* array, int size) {
    T max = array[0];
    for (int i = 1; i < size; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}
```

```
int main() {
    int intArray[] = { 4, 2, 5, 1, 3 };
    double doubleArray[] = { 3.14, 1.5, 2.7, 4.2 };
```

```
    int maxInt = maxElement(intArray, 5);
    double maxDouble = maxElement(doubleArray, 4);
```

```
    std::cout << "Max element in intArray: " << maxInt << std::endl;
    std::cout << "Max element in doubleArray: " << maxDouble << std::endl;
```

```
    return 0;
}
```

In this example, the `maxElement` template function takes an array of any data type and its size as input, and returns the maximum element in the array. The function uses a simple loop algorithm to traverse the array and find the maximum element. The main program calls the `maxElement` function with an array of integers and an array of doubles, and prints the maximum element for each array. This function can be used with any data type that supports the greater-than operator (`>`).

USE OF TEMPLATE FUNCTION (ANOTHER EXAMPLE)

Suppose you are writing a program for a fitness tracking application that needs to calculate the body mass index (BMI) of a person. The BMI is calculated using the following formula: $\text{weight} / (\text{height} * \text{height})$, where weight is in kilograms and height is in meters.

To calculate the BMI for a person, you need to perform the same calculation regardless of the person's weight and height data types. A template function can be used to perform this calculation for any data type.

Here's an example implementation:

```
#include
template<typename T>
T calculate_bmi(T weight, T height) {
    T bmi = weight / (height * height);
    return bmi;
}

int main() {
    double weight = 75.0; // kilograms
    double height = 1.8; // meters
    double bmi = calculate_bmi(weight, height);
    std::cout << "BMI: " << bmi << std::endl;
    return 0;
}
```

In this example, the `calculate_bmi` function is defined as a template function with a single type parameter `T`. This function takes two parameters `weight` and `height` of type `T` and returns the calculated BMI of type `T`.

In the `main` function, we use the `calculate_bmi` function to calculate the BMI for a person with weight 75.0 kg and height 1.8 meters. The result is then printed to the console.

Template class

In C++, a template class is a class that is defined with generic type parameters. These parameters can be used to define member functions, member variables, and the overall structure of the class.

Template classes are useful because they allow you to write code that is independent of the specific data types it operates on. For example, you could write a generic container class that can hold any type of object. This container class could then be used with any data type without needing to be re-implemented for each specific type.

```

template<class T>
class MyContainer {
private:
    T* data;
    int size;
public:
    MyContainer(int size) {
        this->size = size;
        data = new T[size];
    }
    ~MyContainer() {
        delete[] data;
    }
    T& operator[](int index) {
        return data[index];
    }
};

```

Another example

Suppose you are writing software for a weather station that can measure different types of weather data, such as temperature, humidity, and air pressure. You want to create a generic data structure to store this weather data that can work with any data type.

A template class can be used to create a generic weather data container that can store any data type for temperature, humidity, and air pressure. Here's an example implementation:

#include

```

template<typename T>
class WeatherData {
private:
    T temperature;
    T humidity;
    T air_pressure;
public:
    WeatherData(T temp, T hum, T press)
        : temperature(temp), humidity(hum), air_pressure(press) {}

    void print_data() {

```

```
std::cout << "Temperature: " << temperature << " C" << std::endl;
std::cout << "Humidity: " << humidity << "%" << std::endl;
std::cout << "Air Pressure: " << air_pressure << " kPa" << std::endl;
}
};
```

```
int main() {
WeatherData<double> data1(25.6, 70.2, 101.3);
data1.print_data();
```

```
WeatherData<int> data2(20, 65, 100);
data2.print_data();
```

```
return 0;
}
```

In this example, the WeatherData class is defined as a template class with a single type parameter T. This class has private member variables temperature, humidity, and air_pressure, all of type T.

The constructor of the class takes three parameters of type T, which correspond to the temperature, humidity, and air pressure measurements, respectively. The print_data function is a member function of the class that prints the stored data to the console.

In the main function, we create two instances of the WeatherData class, one with double data types and the other with integer data types. The print_data function is then called on both instances to display the stored weather data.

Using a template class in this way allows us to create a single data structure that can store any type of weather data without needing to define a separate class for each data type.

Tasks

Question 1 :

Suppose you are working on a library management system that needs to store information about books, such as their title, author, ISBN, and availability status, in a file. You need to implement a data structure to store this information and provide functionality to read and write the data to a file.

You decide to use a binary file to store the book information because it allows for faster read and write operations than a text file. You also want to use file modes to control the behavior of the file stream, seek and tell functions to manipulate the file pointer, and formatted and unformatted input-output operations to interact with the file.

Question 2 :

Suppose you are working on a database management system that needs to store information about various types of objects, such as books, movies, and music albums, in separate binary files. You want to create a generic file handling function that can read and write data to any of these files and provide functionality to seek to a specific position in the file and return the current file position.

To accomplish this, you decide to use a template function and class to handle the file operations and provide flexibility to work with different data types. You also want to use file modes to control the behavior of the file stream and seek and tell functions to manipulate the file pointer.

Your task is to implement the template function and class to handle the file operations and create a program that demonstrates the use of the function to read and write data to a binary file.

Here are the requirements for the program:

- The program should have a `main()` function that reads and writes data to a binary file using the template function.
- The template function should have the following signature: `template <typename T> void FileHandler(const std::string& filename, const T& data, std::ios_base::openmode mode, std::streampos position = 0)`.
- The `FileHandler` function should take four arguments: the filename of the binary file to read or write, the data to read or write, the file mode to use (such as `std::ios::in`, `std::ios::out`, or `std::ios::app`), and an optional position to seek to before performing the read or write operation.
- If the `FileHandler` function is called with the `std::ios::out` or `std::ios::app` mode, it should write the data to the end of the file. If called with the `std::ios::in` mode, it should read the data from the current file position.

- The FileHandler function should use the seekg and tellg functions to seek to the specified position in the file and return the current file position, respectively.
- The program should create a binary file for books and write two books to the file using the FileHandler function. Each book should have a title, author, and ISBN.
- The program should read the first book from the file using the FileHandler function and print the book information to the console.

Question 3

You are tasked with building a program that can manage an inventory of products using binary files in C++. The program should have the following functionalities:

1. Add a new product to the inventory
2. Display all products in the inventory
3. Search for a product by name or ID
4. Edit an existing product in the inventory
5. Delete an existing product from the inventory
6. Calculate the total value of the inventory

Each product should have the following attributes:

- Name (string)
- ID (integer)
- Price (double)
- Quantity (integer)

The program should be menu-driven, where the user can select the desired functionality by entering a corresponding number. The program should handle invalid user input and use exception handling to avoid program crashes.

The program should also use binary files to store and retrieve the inventory data. The program should support both formatted and unformatted input/output operations. Additionally, the program should allow the user to specify the file access mode, i.e., whether the file should be opened for reading, writing, or both.

Finally, the program should use seek and tell functions to navigate through the binary file and edit/delete specific products.

Your task is to implement the program using C++ and the following concepts:

- Binary file input/output with modes and seek/tell functions
- Template functions and classes

Question 4

Write a C++ program that implements a template class Person that represents a person with a name and an age. The class should have a constructor that takes a name and an age and sets the member variables accordingly. The class should also have member functions getName and getAge that return the name and age of the person, respectively.

Create two additional classes that inherit from Person:

- Student: represents a student with a major and a GPA. The class should have a constructor that takes a name, an age, a major, and a GPA, and sets the member variables accordingly. The class should also have member functions getMajor and getGPA that return the major and GPA of the student, respectively.
- Employee: represents an employee with a job title and a salary. The class should have a constructor that takes a name, an age, a job title, and a salary, and sets the member variables accordingly. The class should also have member functions getJobTitle and getSalary that return the job title and salary of the employee, respectively.

Create a template class Pair that represents a pair of two elements of different data types. The class should have a constructor that takes two arguments of different data types and sets the member variables accordingly. The class should also have member functions getFirst and getSecond that return the first and second elements of the pair, respectively.

Finally, create a function printInfo that takes a Person, a Student, or an Employee, and prints their name and age, as well as any additional information that is specific to the derived class (i.e., major and GPA for a Student, or job title and salary for an Employee).

Your program should demonstrate the use of the Pair template class and the printInfo function by creating instances of Person, Student, and Employee, creating instances of Pair that store different combinations of data types, and calling printInfo with each instance.