Object-Oriented Programming

Lecture 9: Struct and File Process

What is a Struct?

- Structures (also called structs) are a user-defined data type in C++ that groups related variables of different types under a single name.
- Think of it as a blueprint: Structs describe the layout of the data, but don't actually hold the data themselves.
- Like a custom template: Helps you create multiple instances (objects) with the same data structure.

Struct vs. Classes

Structs

- Representing data points: Representing geometric concepts like coordinates, colors, or physical quantities (e.g., velocity, temperature).
- Network & File I/O: Structs offer a convenient way to organize data into specific formats needed for sending data over a network or writing to files (think about binary file headers).
- Interacting with C APIs: Many libraries written in C use structs extensively for data structures, so C++ structs provide a natural interface.
- Optimization-critical scenarios: Where small memory footprints and predictable memory layouts are beneficial for performance.

Classes

- Strong Encapsulation: Classes offer robust access control (public, protected, private) for more organized code and safer data management.
- Inheritance and Polymorphism: Classes are fundamental to object-oriented programming, supporting hierarchical relationships and code reuse.
- Methods and Behavior: Classes can bundle associated functions (methods) within the data structure, promoting tighter coupling of data and logic.

Struct vs. Classes

```
struct Person {
    string name;  // public by default
};

class Employee {
    string name;  // private by default
};
```

Defining and Declaring Structs

```
struct Book {
    string title;
    string author;
    int year;
};
Book myBook;

struct {                 // Structure declaration
  int myNum;             // Member (int variable)
  string myString;       // Member (string variable)
} myStructure;           // Structure variable

struct {
  string brand;
  string model;
  int year;
} myCar1, myCar2;
```

Accessing Struct Members

```
myBook.title = "C++ Primer";
myBook.author = "Stanley B. Lippman";
myBook.year = 2012;

cout << "Book: " << myBook.title << ", Author: " << myBook.author << ", Year: " <<
myBook.year;
```

Initialization of Structs

```
// Traditional initialization
Person person1;
person1.name = "Alice";
person1.age = 30;
person1.city = "New York";

// List initialization (C++11 and later)
Person person2 = {"Bob", 25, "Los Angeles"};
```

Structs and Functions

```
void printPerson(Person p) {
    cout << p.name << ", " << p.age << ", " << p.city << endl;
}

Person createPerson(string name, int age, string city) {
    Person p = {name, age, city};
    return p;
}
```

I/O File Stream Objects and Methods

- To store and retrieve data outside a C++ program, two things are needed:
  - A file
  - A file stream object
- A file is a collection of data stored together under a common name, usually on disk, magnetic tape, USB drive, or CD
- Each file has a unique file name, referred to as file's external name
- Two basic types of files exist
  - Text files (also known as character-based files)
  - Binary files

I/O File Stream Objects and Methods

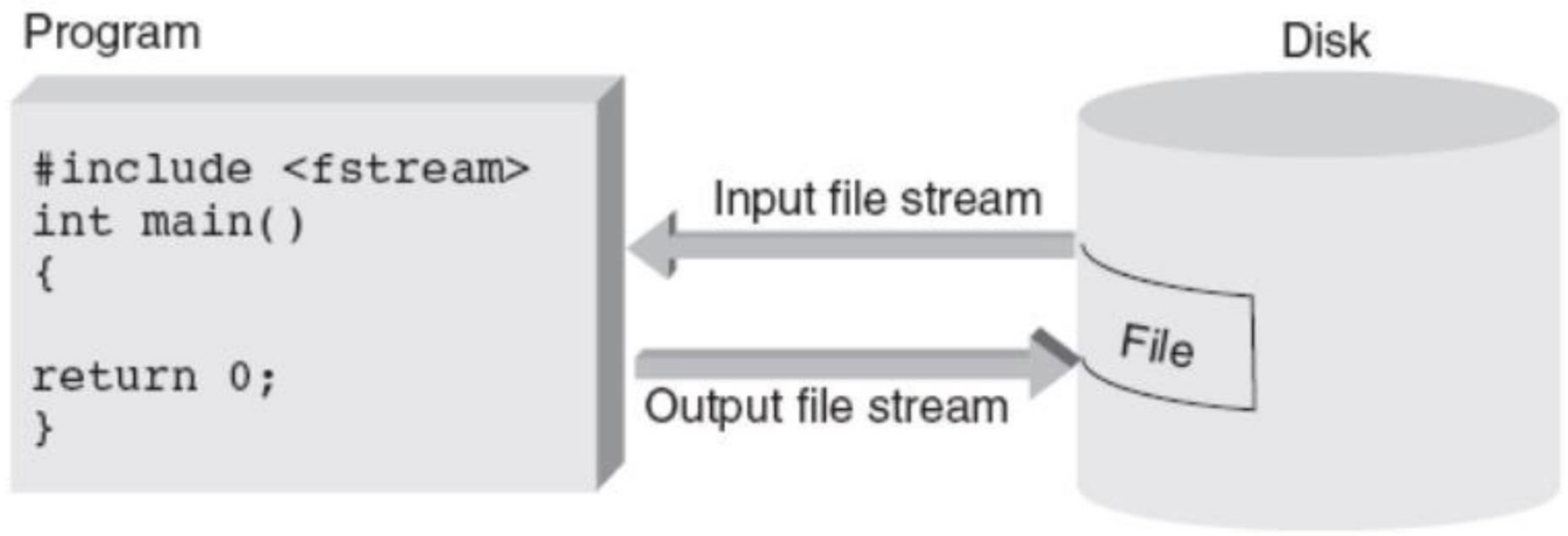
- File naming conventions:
  - Use descriptive names
  - Avoid long file names
    - They take more time to type and can result in typing errors
    - Manageable length for file name is 12 to 14 characters, with maximum of 25 characters
- Choose file names that indicate type of data in file and application for which it is used
  - Frequently, first eight characters describe data, and an extension describes application

I/O File Stream Objects and Methods

Mode	**Description**
ios::in	Open file for reading
ios::out	Open file for writing (erases content)
ios::app	Append data to file
ios::ate	Open and move pointer to end of file
ios::binary	Open in binary mode
ios::trunc	Truncate file if it exists

I/O File Stream Objects and Methods

- **File stream**: One-way transmission path that is used to connect file stored on physical device, such as disk or CD, to program
- **Mode (of file stream):** Determines whether path will move data from file into program or from program to file
- **Input file stream**: Receives or reads data from file into program
- **Output file stream**: Sends or writes data to file



```
#include <fstream>
int main()
{

return 0;
}
```

I/O File Stream Objects and Methods

- Distinct file stream object must be created for each file used, regardless of file's type
- For program to both read and write to file, both an input and output file stream object are required
  - Input file stream objects are declared to be of type ifstream
  - Output file streams are declared to be of type ofstream

File Stream Methods

- Each file stream object has access to methods defined for its respective ifstream , ofstream , fstream class, including:
  - Opening file: connecting stream object name to external file name
  - Determining whether a successful connection has been made
  - Closing file: closing connection
  - Getting next data item into program from input stream
  - Putting new data item from program onto output stream
  - Detecting when end of file has been reached
- Headers Required: #include <fstream>
- Classes:
  - ifstream: Input file stream.
  - ofstream: Output file stream.
  - fstream: Input/output file stream.

File Stream Methods

- open ( ) method:
  - Establishes physical connecting link between program and file
  - Connects file's external computer name to stream object name used internally by program
- Before a file can be opened, it must be declared as either ifstream or ofstream object
- File opened for input is said to be in **read mode**

Writing to a File

```
#include <fstream>

int main() {
    std::ofstream outFile("example.txt");
    if (outFile.is_open()) {
        outFile << "Hello, World! 1" << std::endl;
        outFile << "Hello, World! 2" << std::endl;
        outFile.close();
    }
    return 0;
}
```

Writing to a File

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

int main() {
    std::ifstream inFile("example.txt");
    std::string line;
    if (inFile.is_open()) {
        while (getline(inFile, line)) {
            std::cout << line << '\n';
        }
        inFile.close();
    }
    return 0;
}
```

Error Handling in File I/O

- fail() method: returns true value if file is unsuccessfully opened, false if open succeeded
  - Good programming practice is to check that connection is established before using file
- In addition to fail ()method, C++ provides three other methods, that can be used to detect file's status

```
Ifstream inFile;
inFile.open("prices.dat");
if (inFile.fail())
{
    cout <<"\nThe file was not successfully opened"<<endl;
    exit(1);
}
```

Error Handling in File I/O

- fail() method: returns true value if file is unsuccessfully opened, false if open succeeded
  - Good programming practice is to check that connection is established before using file
- In addition to fail ()method, C++ provides three other methods, that can be used to detect file's status

```
Ifstream inFile;
inFile.open("prices.dat");
if (inFile.fail())
{
    cout <<"\nThe file was not successfully opened"<<endl;
    exit(1);
}
```

Error Handling in File I/O

- fail(): Returns true if the last I/O operation failed because of an incorrect format or other I/O issue that does not involve reaching the end of the file.
- eof(): Indicates if the end of the file has been reached, which is a common situation when reading files.
- bad(): Signifies a serious error, such as loss of integrity of the stream, which may no longer be usable.

Error Handling in File I/O

```
ifstream file("example.txt");
if (!file) {
    cerr << "Error opening file!" << endl;
    return 1;
}

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

if (file.eof()) {
    cout << "\nEnd of file reached." << endl;
} else if (file.fail()) {
    cerr << "Error: Incorrect file format!" << endl;
} else if (file.bad()) {
    cerr << "Critical I/O error!" << endl;
}

file.close();
```

Closing File

- The close() method closes a file, breaking the connection between the file's external name and its file stream. This frees up the file stream to be used for a different file.
- Closing files no longer needed is important because of system limitations on the number of simultaneously open files. However, the operating system automatically closes any open files when a program ends normally.

get()

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

int main() {
    std::ifstream inFile("input.txt");
    char ch;

    if (!inFile) {
        std::cerr << "Unable to open file for reading." << std::endl;
        return 1;
    }

    while (inFile.get(ch)) {  // Reads one character at a time
        std::cout << ch;  // Output the character
    }

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

put()

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

int main() {
    std::ofstream outFile("output.txt");

    if (!outFile) {
        std::cerr << "Unable to open file for writing." << std::endl;
        return 1;
    }

    // Writing individual characters to the file
    outFile.put('H').put('e').put('l').put('l').put('o').put('\n');

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

Random File Access

- Ability to read from or write to any part of a file directly, without reading through the file sequentially.
- Seeking: Moving the file pointer to a specific location within the file using seekg() for input streams and seekp() for output streams.
- File Pointer Positioning: Positioning the file pointer at a desired byte offset from the beginning, current position, or end of the file.
- seekg(offset, dir) for input streams (ifstream): Sets the position of the next character to be extracted.
- seekp(offset, dir) for output streams (ofstream): Sets the position where the next character is to be inserted.

Random File Access

Function	Attributes and Overloads	Purpose
seekg(pos)	pos - A streampos or streamoff type indicating the absolute position.	Sets the position of the get pointer in the input stream to an absolute position.
seekg(offset, dir)	offset - A streamoff type indicating the offset relative to dir - std::ios_base::beg, std::ios_base::cur, or std::ios_base::end. Example Code: Randomly Reading from a File	Sets the get pointer position relative to dir, which specifies the reference point for offset.
seekp(pos)	pos - A streampos or streamoff type indicating the absolute position.	Sets the position of the put pointer in the output stream to an absolute position.
seekp(offset, dir)	offset - A streamoff type indicating the offset relative to dir. dir - One of std::ios_base::beg, std::ios_base::cur, or std::ios_base::end.	Sets the put pointer position relative to dir, which specifies the reference point for offset.
tellg()	None	Returns the current position of the get pointer in the input stream.
tellp()	None	Returns the current position of the put pointer in the output stream.

Random File Access

- Real-world applications of seekg() and seekp():
  - Reading a specific record in a file (e.g., search by ID in a database file).
  - Updating a single byte in a binary file (e.g., modifying a game save).
  - Jumping to a position in a log file.

```
fstream file("data.bin", ios::in | ios::out |
ios::binary);
file.seekp(10, ios::beg);  // Move to byte 10
file.put('X');  // Modify data
file.close();
```

Point for Positioning Operations

- std::ios_base::beg: Stands for "beginning." It is used to specify that the positioning operation should start from the beginning of the stream. When you use this constant with seekg or seekp, the offset is counted from the start of the file.
- std::ios_base::cur: Stands for "current." This constant is used to indicate that the positioning should be relative to the current position of the stream's pointer. Using this with seekg or seekp means that the offset will be applied starting from the current location of the get or put pointer in the stream.
- std::ios_base::end: Represents the "end" of the stream. This constant is used to specify that the positioning should be relative to the end of the stream. When used with seekg or seekp, the offset is counted backwards from the end of the file.

Example Code: Randomly Reading from a File

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

int main() {
    std::ifstream file("example.dat", std::ios::binary);
    // binary mode , no conversion occurs, and the data is read exactly as it is in the file.
    if (!file) {
        std::cerr << "Cannot open file." << std::endl;
        return 1;
    }

    // Move to 10th byte in file
    file.seekg(10, std::ios::beg);

    char ch;
    file.get(ch);  // Read a single character at the new position
    std::cout << "Character at position 10: " << ch << std::endl;

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

Example Code: Randomly Reading from a File

Example Code: Randomly Writing to a File

```
#include <fstream>

int main() {
    std::ffstream file("example.dat", std::ios::binary | std::ios::in | std::ios::out);
    // open a file for both reading and writing in binary mode
    if (!file) {
        std::cerr << "Cannot open file." << std::endl;
        return 1;
    }

    // Move to 5th byte in file
    file.seekp(5, std::ios::beg);

    file.put('X');  // Write a single character at the new position

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

Example Code: Randomly Reading from a File

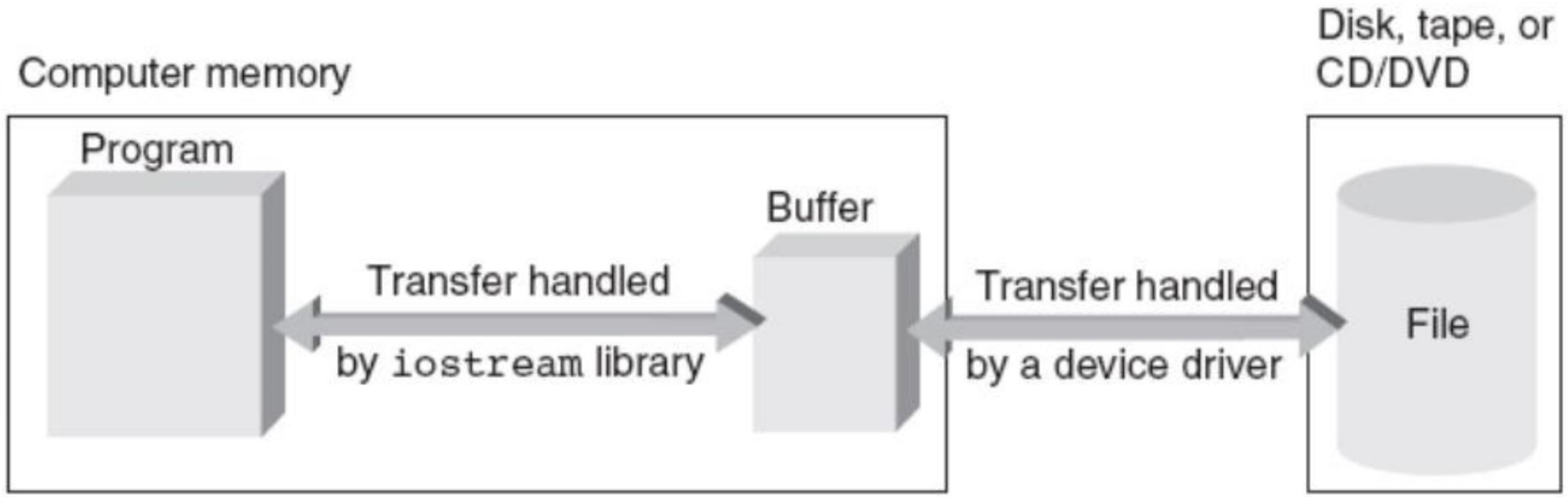
File Streams as Function Arguments

- Passing ifstream and ofstream as Arguments

```
void readFileContents(ifstream& fileStream) {
    string line;
    while (getline(fileStream, line)) {
        cout << line << endl;
    }
}

void writeFileContents(ofstream& fileStream, const string& content) {
    fileStream << content;
}
```

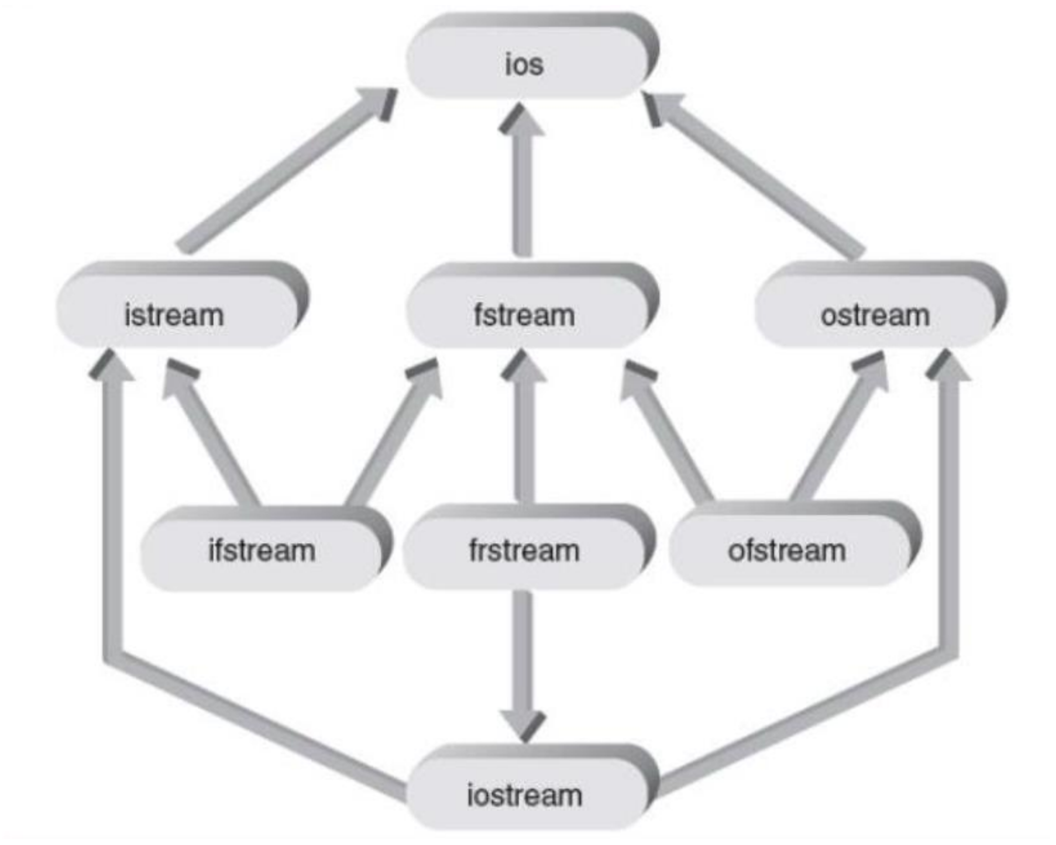
File Stream Transfer Mechanism


Computer memory
Program
Buffer
Transfer handled by iostream library
Disk, tape, or CD/DVD
Transfer handled by a device driver
File

iostream Class Library

- The iostream class library is a framework for handling input and output (I/O) operations.
- To provide developers with a set of tools that allow for reading from input sources (like the keyboard, files, or network connections) and writing to output destinations (such as the console, files, or over the network).
- iostream class library consists of two primary base classes
    - **streambuf** class provides the file buffer
    - **ios** class contains pointer to the file buffers provided by streambuf class and general routines for transferring text data

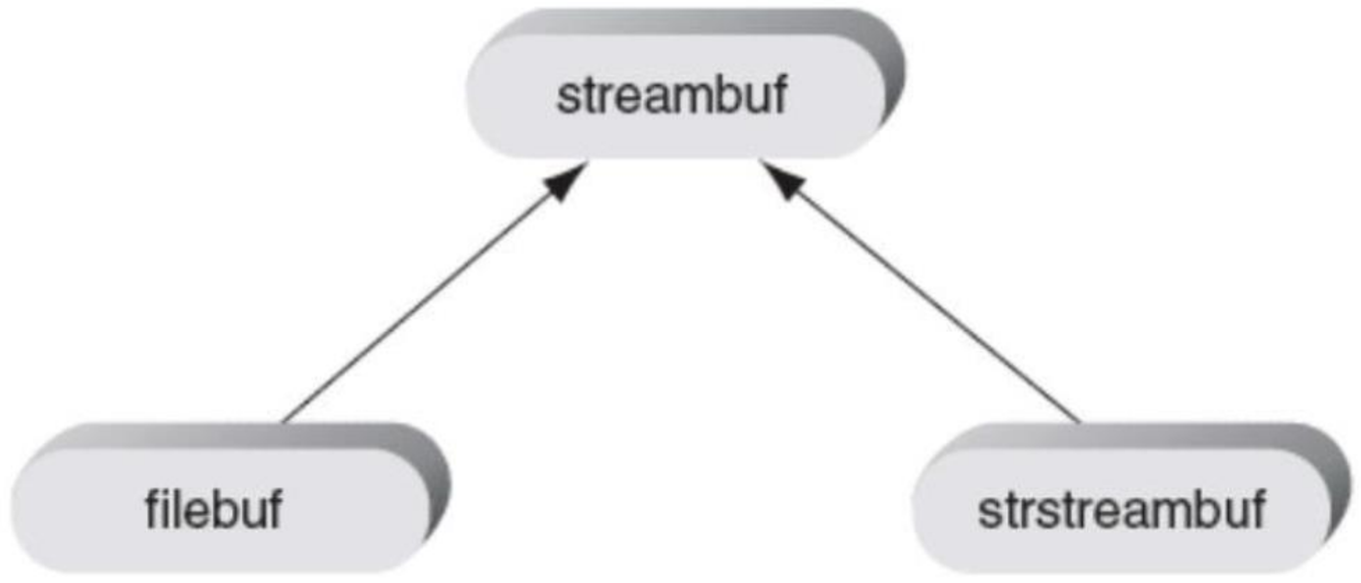
Hierarchy of Classes



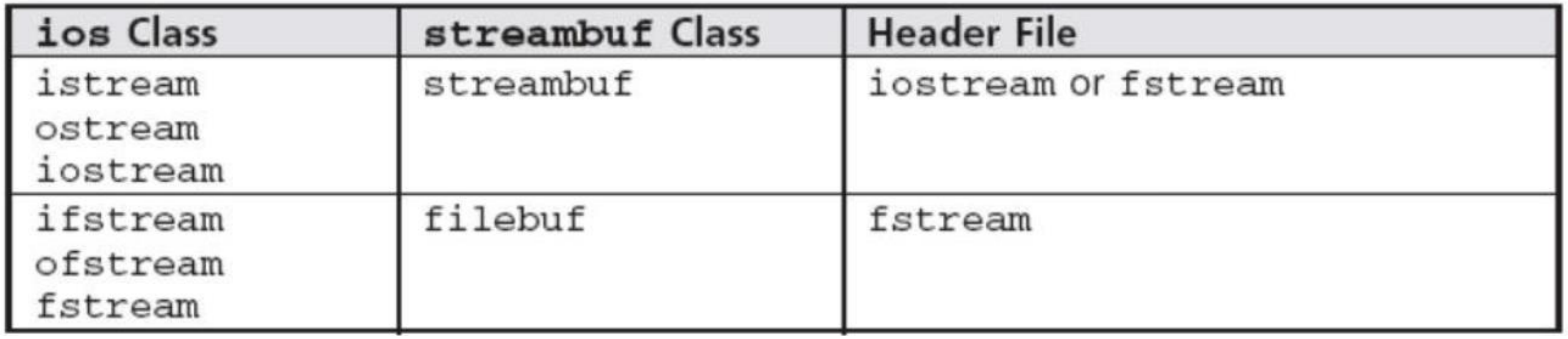
ios: Serves as a base class for all types of input and output streams. Provides basic I/O functionalities and manages the association with stream buffers **istream:** input streams. Provides functionalities for reading data from various sources.
ostream: output streams. Offers functionalities to write data to various destinations.
fstream: This class is derived from both istream and ostream, which are combined via the iostream class (not directly shown in the hierarchy). fstream is used for both input and output file streams. It inherits capabilities for both reading from and writing to files.
ifstream: A specialization of istream for file input operations. It inherits from istream and is used specifically for reading data from files.
ofstream: A specialization of ostream for file output operations. It inherits from ostream and is used for writing data to files.
iostream: derived from both istream and ostream. It is capable of handling both input and output operations, and it serves as the base for the fstream class.

The arrows indicate inheritance, with the direction of the arrow pointing from the derived class to the base class. This means that ifstream, ofstream, and fstream all eventually inherit from ios. The istream and ostream classes provide the fundamental operations for input and output, respectively, which are then specialized further in the ifstream, ofstream, and fstream classes for file operations.

Hierarchy of Classes



Components of the iostream Class Library

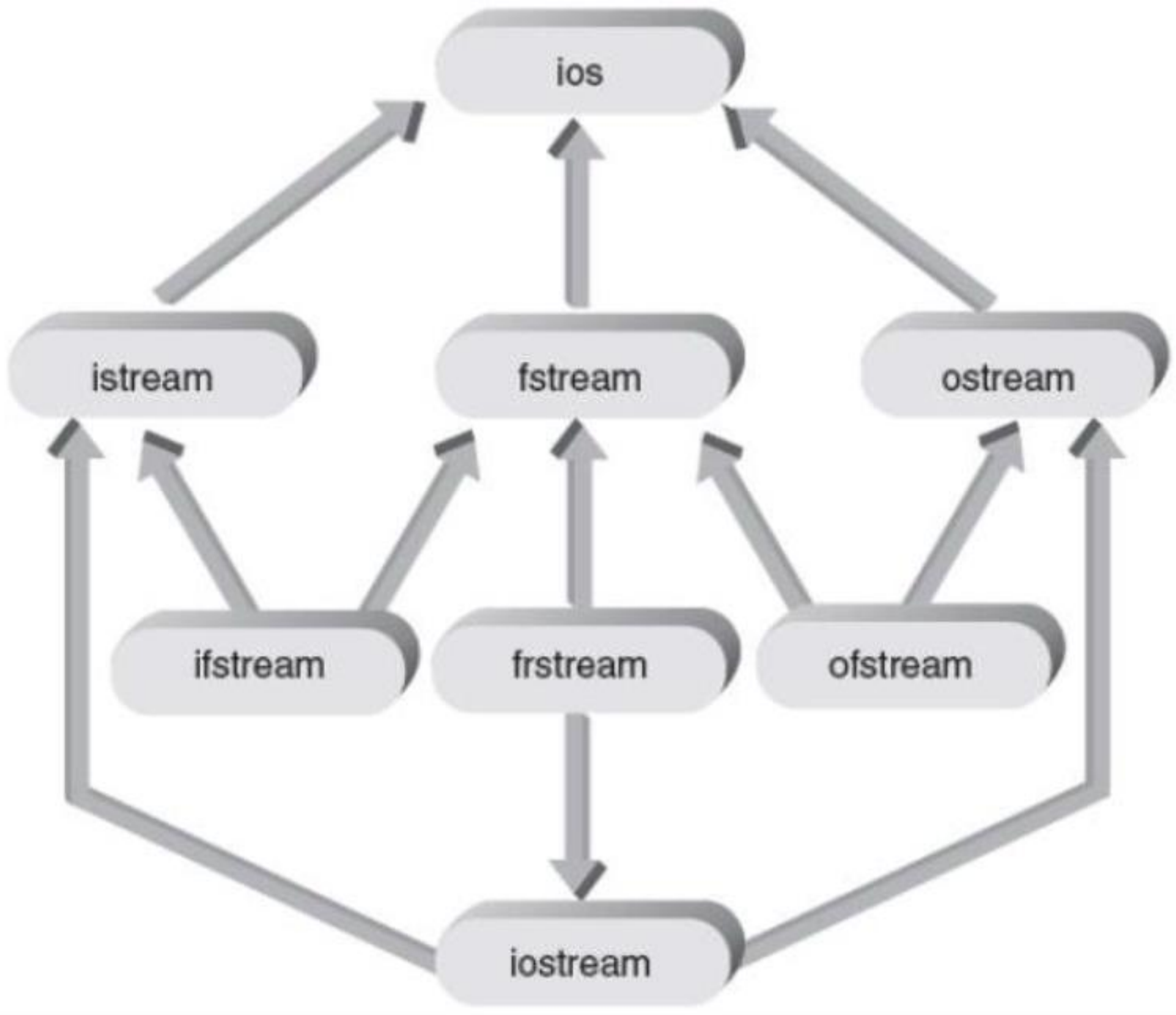


ios Class	streambuf Class	Header File
istream ostream iostream	streambuf	iostream or fstream
ifstream ofstream fstream	filebuf	fstream

Hierarchy of Classes

- **streambuf:** This base class is responsible for direct buffer management. It provides mechanisms for controlled access to the input and output sequences, effectively serving as the foundation for I/O operations.
- **ios_base:** This class serves as the base class for all I/O classes, containing definitions for types, members, and basic I/O functions that are common across all I/O streams. It defines several important aspects, such as format flags, I/O state flags, and control over the precision and width of I/O operations.
- **ios:** Derived from ios_base, the ios class introduces more functionality related to stream buffering and error state management. It contains a pointer to a streambuf object, linking high-level I/O functionalities with the low-level buffer control provided by streambuf.
- **istream and ostream:** These classes are derived from ios and are designed for input and output operations, respectively. istream is used for reading data, while ostream is used for writing data. Both classes provide a wide range of overloaded operators and functions for handling different data types, making I/O operations intuitive and easy to perform.
- **iostream:** This class inherits from both istream and ostream, combining their capabilities to create a stream that can handle both input and output operations. It's useful for streams that need to support reading and writing simultaneously, like network sockets.
- **fstream, ifstream, ofstream:** These classes are specialized versions of iostream, istream, and ostream, respectively, and are specifically designed for file I/O operations. They provide functionalities to open, read, write, and close files, extending the basic I/O capabilities to work efficiently with the filesystem.

Hierarchy of Classes



In-Memory Formatting with strstream

- strstream is part of the C++ I/O stream library.
- Derived from the ios class and provides capabilities for in-memory stream operations.
- "In-memory stream operations" refer to the processing of data within the program's memory. This is done through streams that operate on strings or memory buffers instead of files or console I/O.
- In C++, this functionality is typically provided by classes such as stringstream, istringstream, and ostringstream which are part of the Standard Template Library (STL). These classes allow to use strings as sources or destinations for input and output operations.
  - stringstream can be used for both input and output on a string.
  - istringstream is specialized for input operations on strings.
  - ostringstream is specialized for output operations to strings.

In-Memory Formatting with strstream

```
#include <sstream>
#include <string>
#include <iostream>

int main() {
   std::stringstream ss; // Create a stringstream object.

   // Output to the stream.
   ss << "This is an in-memory stream operation. ";
   ss << 42; // You can also insert numerical values.

   // Input from the stream.
   std::string line;
   int number;
   ss >> line >> number; // Extracts a string and then a number from the stream.

   std::cout << line << " " << number; // Outputs: "This 42"

   return 0;
}
```

Q & A