Lecture 49-50

Abstract classes, Virtual destructors & polymorphism, Order of execution of Virtual functions

1. Abstract Classes

An **Abstract Class** in object-oriented programming (OOP) is a class that cannot be instantiated directly, meaning you cannot create objects from it. Its primary purpose is to serve as a blueprint for other classes. Abstract classes are often used when you want to enforce a certain structure or behavior in the derived (child) classes, while leaving some implementation details to be defined in those subclasses.

Key Points:

1. **Cannot be Instantiated**: You cannot create an object from an abstract class.
2. **Can Contain Abstract Methods**: Abstract methods are methods declared without any implementation. These methods must be implemented in derived classes.
3. **May Contain Non-Abstract Methods**: Abstract classes can also have methods with full implementations that can be used by derived classes.
4. **Must be Inherited**: To use an abstract class, you must create a subclass that provides implementations for all of its abstract methods.

Code:

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

class Shape {
public:
    // Pure virtual function, making Shape an abstract class
    virtual void draw() = 0;
};

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

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

int main() {
    Shape* shape1 = new Circle();  // Create object of derived
class
    Shape* shape2 = new Square();  // Create object of another
derived class

    shape1->draw();  // Outputs: Drawing Circle
    shape2->draw();  // Outputs: Drawing Square

    delete shape1;
    delete shape2;

    return 0;
}
```

2. Virtual Destructors

A **virtual destructor** in object-oriented programming (particularly in languages like C++) ensures that the destructor of the most derived (child) class is called when an object is deleted through a pointer to a base class. Without a virtual destructor, only the base class destructor will be called, which can lead to resource leaks or undefined behavior if the derived class has its own resources that need to be cleaned up.

Code:

```
#include <iostream>

using namespace std;

class Base {

public:

    Base() { cout << "Base Constructor" << endl; }
    virtual ~Base() { cout << "Base Destructor" << endl; }  //
Virtual destructor

};

class Derived : public Base {

public:

    Derived() { cout << "Derived Constructor" << endl; }

    ~Derived() { cout << "Derived Destructor" << endl; }

};

int main() {

    Base* obj = new Derived();

    delete obj;  // Calls both Derived and Base destructors

    return 0;

}
```

3. Order of Execution of Virtual Functions

The **order of execution of virtual functions** in C++ is crucial when working with inheritance, especially when you have a base class pointer or reference pointing to an object of a derived class. Virtual functions ensure **runtime polymorphism**, meaning the version of the function executed is determined at runtime, based on the actual type of the object (not the type of the pointer or reference).

Code:

```
#include <iostream>

using namespace std;

class Base {

public:

    virtual void show() {

        cout << "Base class show function" << endl;

    }

    void print() {

        cout << "Base class print function" << endl;
```

```cpp
    }
};
class Derived : public Base {
public:
    void show() override {
        cout << "Derived class show function" << endl;
    }
    void print() {
        cout << "Derived class print function" << endl;
    }
};
int main() {
    Base* basePtr;
    Derived derivedObj;
    basePtr = &derivedObj;
    // Virtual function, resolved at runtime
    basePtr->show();  // Outputs: Derived class show function
    // Non-virtual function, resolved at compile time
    basePtr->print();  // Outputs: Base class print function
    return 0;
}
```

4. Polymorphism

Polymorphism is a key feature in object-oriented programming (OOP) that allows objects of different classes to be treated through the same interface. In simple terms, **polymorphism** means "many forms." It allows a function, object, or method to behave differently based on the object or context in which it is used.

Polymorphism in C++ can be classified into two types:

1. **Compile-Time Polymorphism** (Static Polymorphism)
2. **Run-Time Polymorphism** (Dynamic Polymorphism)

1. Compile-Time Polymorphism (Method Overloading and Operator Overloading)

Compile-time polymorphism is achieved through **function overloading** or **operator overloading**. This allows multiple functions or operators to have the same name but operate differently based on the parameters or types.

Code:

```
#include <iostream>

using namespace std;

class Math {

public:

    // Function to add two integers

    int add(int a, int b) {

        return a + b;

    }

    // Overloaded function to add two floating-point numbers

    float add(float a, float b) {

        return a + b;

    }

};

int main() {
    Math math;

    cout << math.add(3, 5) << endl;  // Calls the int version

    cout << math.add(3.2f, 5.4f) << endl;  // Calls the float
version

    return 0;

}
```

2. Run-Time Polymorphism (Using Virtual Functions)

Run-time polymorphism is achieved through **virtual functions**. This allows a base class pointer or reference to point to objects of derived classes, and the appropriate function of the derived class is called, even if the function call is made through a base class reference or pointer.

Key Mechanism:

- **Virtual Functions**: A virtual function is a function in a base class that you expect to override in a derived class. When a function is declared as `virtual` in the base class, the function in the derived class is called based on the type of the object pointed to, not the type of the pointer.

Code:

```
#include <iostream>

using namespace std;

// Base class

class Vehicle {

public:

    // Virtual function in base class

    virtual void startEngine() {

        cout << "Starting vehicle engine..." << endl;

    }

};
```

```
// Derived class 1

class Car : public Vehicle {

public:

    void startEngine() override {

        cout << "Starting car engine!" << endl;

    }

};


// Derived class 2

class Bike : public Vehicle {

public:

    void startEngine() override {

        cout << "Starting bike engine!" << endl;

    }

};

int main() {

    Vehicle* vehiclePtr;  // Base class pointer

    Car myCar;

    Bike myBike;

    // Point to Car object

    vehiclePtr = &myCar;

    vehiclePtr->startEngine();  // Outputs: Starting car
engine!

    // Point to Bike object

    vehiclePtr = &myBike;

    vehiclePtr->startEngine();  // Outputs: Starting bike
engine!

    return 0;
}
```

- We have a **base class** Vehicle with a virtual function startEngine().
- Two **derived classes** Car and Bike override the startEngine() function.
- In the main() function, we use a **base class pointer** (vehiclePtr) to point to objects of the derived classes (myCar and myBike).
- Even though the pointer type is Vehicle*, it calls the correct startEngine() function for the actual object (Car or Bike).

Lecture 51-52

Overriding member functions, Accessing base class functions

Overriding Member Functions in C++

In object-oriented programming, overriding occurs when a derived class provides a specific implementation of a member function that is already defined in its base class. The base class defines the function, and the derived class "overrides" it with its own version. To override a function in C++, the function signatures must be the same in both classes.

Key points:

- The function in the base class is usually marked as virtual to allow overriding.
- The overridden function in the derived class must have the same name, return type, and parameters.
- If you want to ensure the function is overridden (not just hidden), use the override keyword in C++11 and above.

Code:

```
#include <iostream>

using namespace std;

class Animal {

public:

     virtual void sound() {   // Base class function, marked
virtual for overriding

        cout << "Animal makes a sound" << endl;

    }

};

class Dog : public Animal {

public:

     void sound() override {   // Overriding the base class
function
```

```
        cout << "Dog barks" << endl;
    }
};
int main() {
    Animal a;
    Dog d;
    Animal* ptr = &a;
    ptr->sound();  // Calls Animal's sound function
    ptr = &d;
     ptr->sound();   // Calls Dog's sound function because of
overriding
    return 0;
}
```

Explanation:

- The Animal class has a sound() function.
- Dog class overrides the sound() function.
- When we use a pointer of type Animal*, the correct sound() function is called based on the actual object type (either Animal or Dog).

Accessing Base Class Functions in C++

Even when you override a function in the derived class, you can still call the base class version of that function using the scope resolution operator (::). This is useful if the derived class wants to extend the functionality of the base class function rather than completely replace it.

Code:

```
#include <iostream>

using namespace std;

class Animal {

public:

    virtual void sound() {

        cout << "Animal makes a sound" << endl;

    }

};

class Dog : public Animal {

public:

    void sound() override {

        Animal::sound();  // Calling the base class version of
sound()

        cout << "Dog barks" << endl;

    }

};

int main() {

    Dog d;

    d.sound();  // Calls Dog's sound, which also calls
Animal's sound

    return 0;

}
```

- In the Dog class, we call Animal::sound() to invoke the base class version of sound().
- After calling the base class version, we add additional functionality (printing "Dog barks").

Lecture 53-54

Order of execution of constructors and destructors

Order of Constructor Execution:

When an object is created, constructors are executed in a particular order:

1. **Base Class Constructor**: If inheritance is involved, the constructor of the base class is executed first.
2. **Derived Class Constructor**: After the base class, the derived class constructor is called.
3. **Members Initialized in Declaration Order**: Within the class itself, if there are any data members, they are initialized in the order of their declaration, regardless of the order of appearance in the initializer list.

Code:

```
#include<iostream>

using namespace std;

class Base {

public:

    Base() {

        cout << "Base class constructor called!" << endl;

    }

    ~Base() {

        cout << "Base class destructor called!" << endl;

    }

};

class Derived : public Base {

public:

    Derived() {

        cout << "Derived class constructor called!" << endl;

    }
    ~Derived() {

        cout << "Derived class destructor called!" << endl;

    }

};

int main() {

    Derived obj;

    return 0;

}
```

Explanation:

- The **Base class constructor** is called first, followed by the **Derived class constructor**.
- Constructors work from **base to derived**.

Order of Destructor Execution:

1. **Derived Class Destructor**: The destructor for the derived class is called first.
2. **Base Class Destructor**: After the derived class destructor finishes, the base class destructor is called.

Code:

```
#include<iostream>

using namespace std;

class A {

public:

    A() {

        cout << "Constructor of A" << endl;

    }

    ~A() {

        cout << "Destructor of A" << endl;

    }
};
class B : public A {
public:
    B() {
        cout << "Constructor of B" << endl;
    }
    ~B() {
        cout << "Destructor of B" << endl;
    }
};
int main() {
    B obj;
    return 0;
}
```

Lecture 55-56

Review of traditional error handling, Basics of exception handling

Error handling is a crucial aspect of any programming language, including C++, where it ensures that a program can deal with unexpected conditions or errors gracefully. In C++, two primary approaches are used for handling errors: **traditional error handling** (which involves error codes, flags, or special return values) and **exception handling** (which involves throwing and catching exceptions). Let's delve into both these topics in detail.

Traditional Error Handling in C++

Traditional error handling techniques in C++ rely on using return values, flags, or error codes to indicate whether a function has executed successfully or encountered an error. The basic idea is that functions or methods return a specific value (often a sentinel value or error code) when they fail to execute as intended.

Key Approaches in Traditional Error Handling:

1. Return Error Codes:
    - Functions often return a specific value to indicate success or failure. A return value of 0 might indicate success, while a non-zero value indicates an error.

Code:

```
#include <iostream>

using namespace std;

int divide(int a, int b, int& result) {

    if (b == 0) {

        return -1;  // Error: Division by zero

    }

    result = a / b;

    return 0;  // Success
}

int main() {

    int result;

    int error = divide(10, 0, result);  // Dividing by zero

    if (error == -1) {

        cout << "Error: Division by zero" << endl;

    } else {

        cout << "Result: " << result << endl;

    }

    return 0;

}
```

Explanation:

- The function divide returns -1 to indicate an error (division by zero). The calling code checks for this error and handles it accordingly.

Exception Handling in C++

Exception handling in C++ provides a more robust and organized way to handle errors, where errors are treated as **exceptions** that can be "thrown" when a problem occurs and "caught" by the code that knows how to handle them.

Key Concepts in Exception Handling:

1. **Throwing an Exception**: When an error occurs, an exception is "thrown" using the throw keyword. This interrupts the normal flow of the program, and the control is passed to the nearest handler (if present).
2. **Catching an Exception**: When an exception is thrown, the program looks for an appropriate "handler" to catch it, defined using the catch block. If no handler is found, the program terminates.
3. **try Block**: Code that might throw an exception is enclosed in a try block. The catch block follows the try block and catches the exception.

Basic Syntax:

```
try {

    // Code that may throw an exception
    if (errorCondition) {

        throw exceptionObject;  // Throw an exception

    }

} catch (exceptionType& e) {

    // Handle the exception

}
```

Example of Exception Handling:

```
#include <iostream>

using namespace std;

int divide(int a, int b) {

    if (b == 0) {

        throw runtime_error("Division by zero error");  // 
Throw exception

    }

    return a / b;

}

int main() {

    try {

        int result = divide(10, 0);  // Division by zero

        cout << "Result: " << result << endl;

    } catch (runtime_error& e) {

        cout << "Caught an exception: " << e.what() << endl;

    }

    return 0;

}
```

Explanation:

- When divide is called with b == 0, it throws a runtime_error exception.

- The exception is caught in the catch block, and the error message is printed using e.what().

Lecture 57-58

Exception handling mechanism, Throwing mechanism, Catching mechanism

The exception handling mechanism in C++ is a structured way to handle runtime errors and other exceptional circumstances. This mechanism ensures that the program does not terminate abruptly when an error occurs. Instead, errors (or exceptions) are "thrown" when a problem arises, and the program can "catch" these exceptions and take appropriate actions.

Basic Concepts in Exception Handling:

1. **Exception**: An exception is an error or unexpected condition that disrupts the normal flow of a program's execution. Examples include division by zero, memory allocation failure, or file-not-found errors.
2. **Throw**: When an exception occurs, it is "thrown" using the throw keyword.
3. **Try Block**: A block of code where exceptions may occur is enclosed in a try block.
4. **Catch Block**: When an exception is thrown, it is handled by a catch block. This block contains the code that should be executed if the exception is caught.

Throwing Mechanism in C++

The **throwing mechanism** in C++ allows you to signal an error or exceptional condition by "throwing" an exception. The throw keyword is used to indicate that an exception has occurred.

How Throwing Works:

1. When an error occurs or some condition is met that the program cannot handle, the code uses the throw keyword to throw an exception. The thrown exception is an object (or primitive type) that represents the error or condition.
2. Once an exception is thrown, the normal flow of the program is interrupted, and the program starts looking for an appropriate catch block to handle the exception.
3. If the exception is not caught, the program will terminate.

Example of Throwing an Exception:

Code:

```
#include <iostream>

using namespace std;

void checkDivision(int numerator, int denominator) {

    if (denominator == 0) {

        throw runtime_error("Division by zero!");  // Throw an
exception

    }

    cout << "Result: " << numerator / denominator << endl;

}

int main() {

    try {

        checkDivision(10, 0);  // This will throw an exception

    } catch (runtime_error& e) {  // Catching the exception

        cout << "Caught an exception: " << e.what() << endl;

    }

    return 0;

}
```

Explanation:

- When checkDivision(10, 0) is called, the function detects a division by zero and throws a runtime_error exception using the throw keyword.
- The exception interrupts the flow of the program and jumps to the catch block.

What Can Be Thrown?

C++ allows you to throw:

1. **Primitive types**: You can throw integers, floating-point numbers, characters, etc.

   ```
   throw 42;  // Throwing an integer
   ```

2. **Standard exception objects**: C++ provides a set of standard exception classes, such as std::runtime_error, std::invalid_argument, std::out_of_range, etc.

   ```
   throw std::runtime_error("An error occurred");
   ```

3. **Custom exception objects**: You can create your own exception classes (typically derived from std::exception).

   ```
   class MyException : public exception {

   public:

        const char* what() const noexcept override {

             return "My custom exception";

         }

   };
   ```

4. **Pointers to objects**: You can throw pointers to objects, though it's less common in practice.

Catching Mechanism in C++

The **catching mechanism** in C++ is responsible for handling exceptions that are thrown during program execution. The catch block defines how the program should respond to specific types of exceptions.

How Catching Works:

1. After an exception is thrown, the program looks for a matching catch block. The catch block must be of the same type (or compatible type) as the exception that was thrown.
2. If a match is found, the catch block is executed, and the program continues from there. If no matching catch block is found, the program terminates.

Basic Syntax

```
try {
    // Code that might throw an exception
    throw "An exception occurred!";
} catch (const char* msg) {  // Catch the exception
    cout << "Caught exception: " << msg << endl;
}
```

Multiple Catch Blocks:

- You can have multiple catch blocks to handle different types of exceptions.
- The order of catch blocks is important. More specific exceptions should be caught before more general ones (e.g., catch std::runtime_error before std::exception).

Code:

```
#include <iostream>
#include <stdexcept>
using namespace std;
int main() {
    try {

        throw runtime_error("Runtime error!");  // Throw
runtime_error

    } catch (runtime_error& e) {  // Catch runtime_error

        cout << "Caught runtime_error: " << e.what() << endl;

    } catch (exception& e) {  // Catch any other
std::exception

        cout << "Caught exception: " << e.what() << endl;

    }

    return 0;

}
```

Lecture 59-60

Re-throwing an exception, Specifying exceptions, Problems on recursion

Rethrowing Exceptions:

- Sometimes, after catching an exception, you may want to pass it up the call chain to be handled by another function. You can rethrow an exception using the throw; statement within a catch block.
- This will pass the same exception up to the next matching catch block.

Code:

```
#include <iostream>

using namespace std;

void func() {

    try {

        throw runtime_error("Error in func()");

    } catch (runtime_error& e) {

        cout << "Caught in func: " << e.what() << endl;

        throw;  // Rethrow the exception

    }

}

int main() {

    try {

        func();

    } catch (runtime_error& e) {

        cout << "Caught in main: " << e.what() << endl;

    }

    return 0;

}
```

Explanation:

- The runtime_error exception is first caught in func(), and then it is rethrown using throw;.
- The rethrown exception is caught in the main() function.

Lecture 61-62

Concept of streams, Input/ Output using Overloaded operators >> and << and Member functions of I/O stream classes.

Concept of Streams in C++

In C++, a stream is an abstraction that represents a flow of data. Streams are used for input (reading data) and output (writing data). The concept of streams allows us to perform input/output operations (I/O) in a consistent manner across different types of data sources and sinks, such as files, the console, or memory buffers.

C++ provides the I/O stream library for handling input and output operations. The most commonly used streams include:

- cin: Standard input stream, used to get input from the console.
- cout: Standard output stream, used to display output to the console.
- cerr: Standard error stream, used for error messages.
- fstream: File stream, used for file I/O operations.

Basic Structure of Streams in C++:

C++ streams are represented by classes, and input/output operations are performed by calling the methods of these classes. The important classes in C++ streams are:

1. **`istream`**: For input operations.
2. **`ostream`**: For output operations.
3. **`fstream`**: For file I/O operations.
4. **`iostream`**: Inherited from both `istream` and `ostream`, used for combined input/output operations.

Input/Output Using Overloaded Operators (>> and <<)

In C++, the **>>** and **<<** operators are overloaded to handle input and output operations on streams. These operators are known as **stream insertion (<<)** and **stream extraction (>>)** operators, respectively.

- **Stream Insertion Operator (<<)**: It is used to send data to an output stream, like `cout`, which writes the data to the console or another output destination.
- **Stream Extraction Operator (>>)**: It is used to extract (read) data from an input stream, like `cin`, which takes input from the console or another input source.

Syntax of >> and <<:

Input (Extraction):

```
cin >> variable;

int x;

cin >> x;  // Takes input from the user and stores it in 'x'
```

Output (Insertion):

```
cout << value;

cout << "Hello, World!";  // Displays the message on the
console
```

Member Functions of I/O Stream Classes

The I/O stream classes in C++ provide several member functions to perform input/output operations. Here are some of the key member functions:

1. ostream (Output Stream) Class Functions:

ostream is the class that defines output operations. The standard output stream (cout) is an instance of ostream.

- **put(char ch)**: This function writes a single character to the output stream.

```
cout.put('A');  // Output: A
```

- **write(const char* s, streamsize n)**: Writes a block of data (a string of n characters) to the output stream.

```
char str[] = "Hello";

cout.write(str, 5);  // Output: Hello
```

- **`flush()`**: Flushes the output buffer, forcing all output to be written to the destination immediately.

```
cout << "Data" << flush;
```

2. istream (Input Stream) Class Functions:

istream is the class that defines input operations. The standard input stream (cin) is an instance of istream.

- **get()**: This function extracts a single character from the input stream.

```
char ch;

cin.get(ch);
```

- **getline(char* s, streamsize n)**: Reads a line of text (up to n characters) from the input stream.

```
char line[50];

cin.getline(line, 50);  // Reads up to 50 characters or until
a newline is encountered
```

- **read(char* s, streamsize n)**: Reads n bytes of data into the buffer s from the input stream.

```
char buffer[10];

cin.read(buffer, 10);  // Reads 10 characters from the input
stream
```

3. iostream Class:

- The iostream class is derived from both istream and ostream and allows for both input and output operations. It is commonly used in file streams.

4. fstream (File Stream) Class Functions:

fstream supports both input and output operations for file handling.

- **`open()`**: Opens a file for input/output.

```
fstream file;

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

- **`close()`**: Closes the file.

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

- **eof()**: Checks if the end of the file has been reached.

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

File I/O Using Streams in C++

C++ provides the fstream class for file I/O operations. There are three classes used for file handling:

1. **ifstream**: Input file stream (read from a file).
2. **ofstream**: Output file stream (write to a file).
3. **fstream**: File stream (can be used for both reading and writing).

Code:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    // Writing to a file
    ofstream outFile("example.txt");
    if (outFile.is_open()) {
        outFile << "Hello, file!" << endl;
        outFile.close();
    } else {
        cout << "Unable to open file for writing" << endl;
    }
    // Reading from a file
    ifstream inFile("example.txt");
    if (inFile.is_open()) {
        string line;
        while (getline(inFile, line)) {
            cout << line << endl;
        }
        inFile.close();
    } else {
        cout << "Unable to open file for reading" << endl;
    }
    return 0;
}
```

Lecture 63-64

File streams, Hierarchy of file stream classes

File Streams in C++

File streams in C++ provide a mechanism to read from and write to files using the standard input/output library. Instead of interacting directly with raw data files (which can be complex), C++ abstracts file handling into streams, making file I/O similar to console I/O. C++ supports three primary file stream classes from the <fstream> header:

1. ifstream: Input file stream, used to read data from files.
2. ofstream: Output file stream, used to write data to files.
3. fstream: File stream, which can be used for both reading from and writing to files.

Concept of Streams

A stream is essentially a flow of data. In file I/O, the flow can be either:

- Input stream: Data is read from a source (a file, in this case).
- Output stream: Data is written to a destination (such as a file).

File streams abstract the complexities of reading and writing data from/to files by treating files as a sequence of bytes or characters, and they use familiar operators (<< and >>) for writing and reading, respectively.

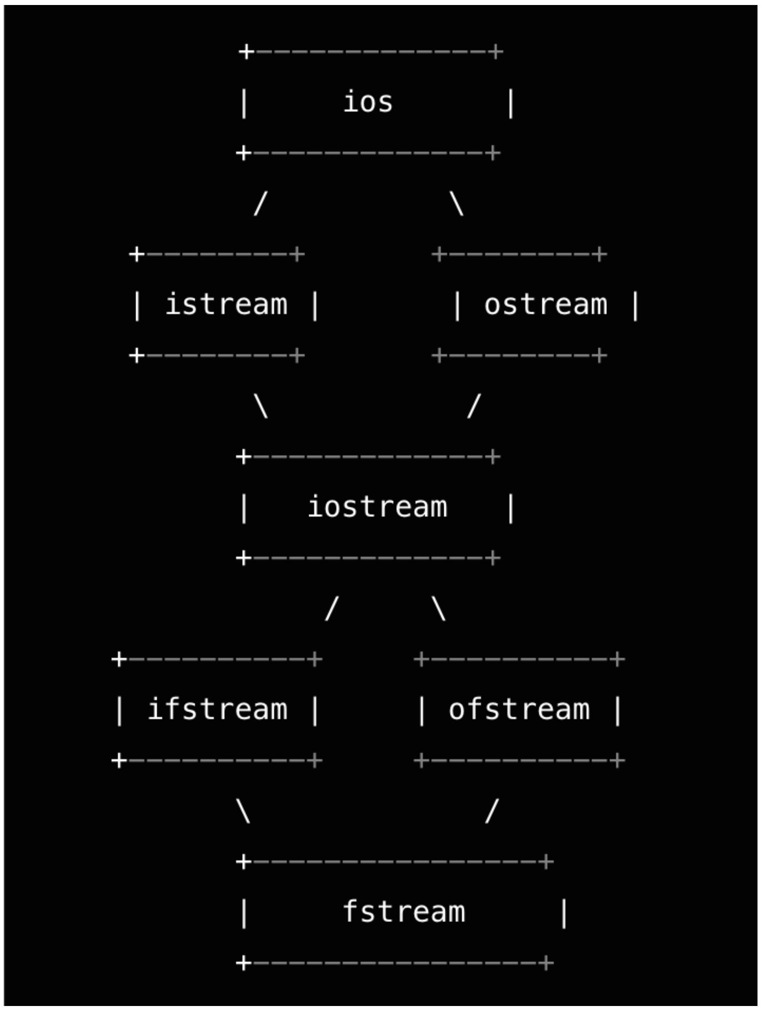
Opening and Closing Files

Before any file I/O operation, you must first open a file using the .open() method or by passing the filename as an argument to the stream constructor. After performing the necessary I/O operations, you should close the file using the .close() method.

- Opening Modes: While opening a file, you can specify modes like reading, writing, appending, and binary operations. These modes are defined using flags from the ios class, such as:
  - ios::in: Open the file for reading (default for ifstream).
  - ios::out: Open the file for writing (default for ofstream).
  - ios::app: Append data to the end of the file.
  - ios::binary: Open the file in binary mode (used for non-text files).
  - ios::trunc: Truncate the file to zero length if it already exists.

Hierarchy of File Stream Classes in C++

C++ organizes the file stream classes into a hierarchical structure. The core I/O functionality is defined in the ios class, and specialized input/output functionalities are built on top of this core. The file stream classes (ifstream, ofstream, and fstream) derive from standard I/O stream classes (istream, ostream), and all of them inherit common properties and methods from the base class ios.



```
             +-------------+
             |     ios     |
             +-------------+
              /          \
       +--------+        +---------+
       | istream |        | ostream |
       +--------+        +---------+
              \           /
             +-------------+
             |   iostream   |
             +-------------+
                   /     \
      +----------+      +----------+
      | ifstream |      | ofstream |
      +----------+      +----------+
             \             /
             +----------------+
             |     fstream     |
             +----------------+
```