Object Oriented Programming with C++

Dr. Dhamodharan G
M.C.A., Ph.D., SET., NET.,
Assistant Professor,
PG and Research Department of Computer Science,
Don Bosco College (Co-Ed), Yelagiri Hills.

Dr. Balasubramanian
Academic Advisor,
Controller of Examinations

Dr. Priya K
Head, PG and Research Department of Computer Science
MarudharKesari Jain College for Women (Autonomous)
Vaniyambadi, Tamil Nadu

Published by
Emperor Research Foundation
www.mayas.info

Published In First

Published In First

Published by Emperor Research Foundation
www.mayas.info

Disclaimer:

The findings/views/opinions expressed in this book are solely those of the authors and do not necessarily reflect the views of the publisher

Book Name: Python Programming
Author: Dr. Dhamodharan G, Dr. Balasubramanian, Dr. Priya K
Edition: First
ISBN: 978-81-19716-04-3
Price: Rs.170/-

CONTENTS

1. Introduction to Object Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data in the form of fields (attributes) and code in the form of procedures (methods). It focuses on organizing software around real-world entities, promoting modularity and reusability.

1.1 Basic Concepts of Object-Oriented Programming (OOP)

The main ideas behind Object-Oriented Programming (OOP) are:

1) Objects
2) Classes
3) Data Abstraction and Encapsulation
4) Inheritance
5) Polymorphism
6) Dynamic Binding
7) Message Passing

Objects

Objects are basic run time entities in an object-oriented system. They represent real-world entities like a person, place, or thing that a program needs to work with.

Classes

A class is a collection of objects. It defines the data and methods that an object can have. For example:

Fruit Mango;

This creates an object Mango from the class Fruit.

Data Abstraction and Encapsulation

Abstraction is about showing only the important details and hiding the unnecessary ones. For example, we can display a bank account's balance but hide how the data is stored or calculated.

Encapsulation is wrapping of data and methods into a single unit. The data is protected and can only be accessed through the methods in the class.

Inheritance

Inheritance is a process of deriving a sub class from base class. It allows one class to reuse the properties and methods of another class. Types of inheritance include:

1) Single Inheritance
2) Multilevel Inheritance
3) Multiple Inheritance
4) Hierarchical Inheritance
5) Hybrid Inheritance

Polymorphism

Polymorphism means "many forms." It allows the same operation to behave differently in different situations. For example:

- Compile-time polymorphism (e.g., function overloading)
- Run-time polymorphism (e.g., method overriding)

Dynamic Binding

Dynamic binding means that the code to execute for a method is decided while the program is running, not during compilation. This is often used with polymorphism and inheritance.

Message Passing

In OOP, objects communicate with each other by sending and receiving messages. The process involves:

- Defining classes and their behavior.
- Creating objects.
- Allowing objects to interact.

2. Basic Elements of C++

C++ is a powerful programming language built on the foundation of C, with additional features for object-oriented programming. The basic elements of C++ include: Tokens, Keywords, Identifiers, Variables, Basic Data Types in C++, and Operators.

2.1 Tokens

Tokens are the smallest building blocks of a C++ program. They are the basic units of a program's syntax and are used to create meaningful instructions. C++ has several types of tokens:

Keywords: Reserved words with special meanings, like int, return, class.

Identifiers: Names used for variables, functions, or classes, like age, sum, My Class.

Literals: Constant values, like 10, 3.14, 'A', "Hello".

Operators: Symbols used to perform operations, like +, -, *, /.

Punctuation or Special Symbols: Characters like {, }, ;, ().

Comments: Text ignored by the compiler, used to describe the code, e.g., // Single line comment or /* Multi-line comment */.

2.2 Keywords

Keywords are reserved words in C++ that have special meanings and cannot be used as identifiers (like variable names or function names). They are predefined by the language and are used to perform specific operations. C++ has a set of 95+ keywords like for, while, switch, class, new, delete, etc., that are essential for writing programs. They must follow specific rules, such as:

- An identifier must start with an alphabet (A-Z or a-z) or an underscore (_). It cannot start with a digit.
- Identifiers can include letters, numbers (0-9), and underscores but no special characters or spaces.
- Identifiers cannot use reserved keywords of C++ (e.g., int, float, if, return).

2.3 Variables

A variable in C++ is a named container used to store data that can change during the program's execution. It acts as a placeholder for values in memory, making it easy to manipulate data.

Each variable must have a specific data type that defines the type of data it can store, such as integers, floating-point numbers, or characters.

Syntax

Declaration: Declares a variable without assigning it a value.

data_type variable_name;
Initialization: Assigns an initial value to a variable during its declaration.
data_type variable_name = value;
Example
Declaration:

```
int age;          // Declares an integer variable
char grade;       // Declares a character variable
```

Initialization:

```
int age = 25;     // Declares and initializes an integer variable
char grade = 'A'; // Declares and initializes a character variable
```

2.4 Basic Data Types in C++

In C++, data types specify the type of data that a variable can hold. Basic data types are the fundamental building blocks used to store simple values like numbers, characters, and logical values. They include:
Integer (int): Stores whole numbers (e.g., -10, 0, 100).
Floating-point (float): Stores decimal numbers with single precision (e.g., 3.14).
Double (double): Stores decimal numbers with higher precision than float (e.g., 3.14159265359).
Character (char): Stores single characters enclosed in single quotes (e.g., 'A').
Boolean (bool): Stores logical values: true or false.
Void (void): Represents no data, usually used for functions that do not return any value.
Example Program:

```
#include <iostream>
using namespace std;
int main() {
   int age = 25;          // Integer
   float height = 5.8;    // Floating-point
   double salary = 75000.50; // Double
   char grade = 'A';      // Character
   bool isPassed = true;  // Boolean
   cout << "Age: " << age << endl;
   cout << "Height: " << height << endl;
   cout << "Salary: " << salary << endl;
   cout << "Grade: " << grade << endl;
   cout << "Passed: " << (isPassed ? "Yes" : "No") << endl;
   return 0;
}
```

Output:
Age: 25
Height: 5.8
Salary: 75000.5
Grade: A
Passed: Yes

2.5 Operators in C++

An operator in C++ is a symbol that performs a specific operation on one or more operands (variables or values). Operators are essential in programming as they allow developers to perform tasks such as arithmetic, comparison, and logical decision-making.

Types of Operators

i. Arithmetic Operators

These are used to perform mathematical calculations:

Addition (+), Subtraction (-), Multiplication (*), Division (/), Modulus (%).
Example: a + b, a * b

ii. Relational Operators

These compare two values and return a boolean result (true or false):

Equal to (==), Not equal to (!=), Greater than (>), Less than (<), Greater than or equal to (>=), Less than or equal to (<=).
Example: a > b, a == b

iii. Logical Operators

These are used for logical operations:

AND (&&), OR (||), NOT (!).
Example: (a > b) && (b > c)

iv. Assignment Operators

These assign values to variables. Some examples are:

Basic assignment (=), Add and assign (+=), Subtract and assign (-=), Multiply and assign (*=), Divide and assign (/=).
Example: a = 10, a += 5

v. Increment/Decrement Operators

These increase or decrease a variable's value by 1:

Increment (++), Decrement (--).

Example: a++, --b

vi. Bitwise Operators

These perform bit-level operations:

AND (&), OR (|), XOR (^), NOT (~), Left shift (<<), Right shift (>>).
Example: a & b, a << 2

vii. Ternary Operator

A compact way of writing conditional expressions:

Syntax: condition ? expression1 : expression2.
Example: a > b ? a : b

Simple Program Demonstrating Different Types of Operators

```
#include <iostream>
using namespace std;
int main() {
   int a = 10, b = 5, c;
   // Arithmetic Operators
   cout << "Arithmetic Operations:" << endl;
   cout << "a + b = " << a + b << endl;
   cout << "a - b = " << a - b << endl;
   cout << "a * b = " << a * b << endl;
   cout << "a / b = " << a / b << endl;
   cout << "a % b = " << a % b << endl;
   // Relational Operators
   cout << "\nRelational Operations:" << endl;
   cout << "a > b: " << (a > b) << endl;
   cout << "a == b: " << (a == b) << endl;
   // Logical Operators
   cout << "\nLogical Operations:" << endl;
   cout << "(a > b && b > 0): " << ((a > b) && (b > 0)) << endl;
   cout << "(a < b || b > 0): " << ((a < b) || (b > 0)) << endl;
   // Assignment Operators
   cout << "\nAssignment Operations:" << endl;
   c = a; // Assigning value of a to c
   cout << "c = a: " << c << endl;
   c += b; // c = c + b
   cout << "c += b: " << c << endl;
   // Increment/Decrement Operators
```

```
    cout << "\nIncrement/Decrement Operations:" << endl;
    cout << "a++: " << a++ << " (post-increment)" << endl;
    cout << "++b: " << ++b << " (pre-increment)" << endl;
    // Ternary Operator
    cout << "\nTernary Operation:" << endl;
    cout << "Max of a and b: " << (a > b ? a : b) << endl;
    return 0;
}
```

Output

```
Arithmetic Operations:
a + b = 15
a - b = 5
a * b = 50
a / b = 2
a % b = 0

Relational Operations:
a > b: 1
a == b: 0

Logical Operations:
(a > b && b > 0): 1
(a < b || b > 0): 1

Assignment Operations:
c = a: 10
c += b: 15

Increment/Decrement Operations:
a++: 10 (post-increment)
++b: 6 (pre-increment)

Ternary Operation:
Max of a and b: 11
```

3. Decision and Control Structures

In programming, Decision and Control Structures are fundamental mechanisms that allow a program to make decisions and control the flow of execution based on certain conditions. These structures enable dynamic behavior in a program by directing it to execute specific blocks of code under different circumstances.

Decision-Making Structures

These structures execute a block of code based on the evaluation of a condition:

a. **if Statement:** Executes a block of code if the condition is true.

b. **if-else Statement:** Provides two paths of execution, one for true and another for false conditions.

c. **switch Statement:** Selects one block of code to execute from multiple options based on the value of an expression.

Looping Structures

These structures repeat a block of code as long as a condition is true:

a. **for Loop:** Repeats a block of code for a specific number of iterations.

b. **while Loop:** Repeats a block of code as long as the condition remains true.

c. **do-while Loop:** Similar to the while loop but executes the block at least once before checking the condition.

3.1 if Statement

The if statement in C++ is a decision-making control structure used to execute a block of code only if a specified condition is true.

Syntax

```
if (condition)
{
   // Code to execute if the condition is true
}
```

- **condition:** An expression that evaluates to true (non-zero) or false (zero).
- If the condition is true, the code inside the braces {} is executed.
- If the condition is false, the code inside the braces is skipped.

Example:

```
#include <iostream>
using namespace std;
int main()
{
   int number;
   cout << "Enter a number: ";
   cin >> number;
   // Check if the number is positive
   if (number > 0)
{
cout << "The number is positive." ;
   }

   return 0;
}
```

Output:

Enter a number: 10
The number is positive.

Explanation:

- The if statement checks whether the entered number is greater than 0.
- If the condition number > 0 is true, the message "The number is positive." is printed.
- If the condition is false, the program simply ends without executing the if block.

3.2 if-else Statement

The if-else statement is a decision-making construct in C++ that allows the program to execute one block of code if a specific condition is true, and another block of code if the condition is false.

Syntax

```
if (condition) {
   // Code block to execute if condition is true
} else {
   // Code block to execute if condition is false
}
```

condition: A logical or relational expression that evaluates to either true or false.
If the condition is true, the code inside the if block is executed.
If the condition is false, the code inside the else block is executed.

Example:

```
#include <iostream>
using namespace std;
int main() {
   int number;
   cout << "Enter a number: ";
   cin >> number;
   if (number % 2 == 0) {
      // Executes if the number is even
      cout << number << " is even.";
   } else {
      // Executes if the number is odd
      cout << number << " is odd.";
   }

   return 0;
}
```

Output:

Case 1:
Enter a number: 6
8 is even.
Case 2:
Enter a number: 5
5 is odd.

Explanation:

The program takes an integer input from the user. The condition number % 2 == 0 checks if the number is divisible by 2 without any remainder (i.e., an even number).If the condition is true:
6 is even is displayed. If the condition is false: 5 is odd is displayed.

3.3 Switch Statement

The switch statement is a control statement that allows the execution of one out of several possible blocks of code based on the value of an expression. The switch statement evaluates an expression, and depending on the value of that expression, it jumps to the corresponding case. If no case matches the expression value, it will execute the default block (if provided).

Syntax:

```
switch(expression) {
   case value1:
      // block of code to be executed if expression == value1;
      break;
   case value2:
      // block of code to be executed if expression == value2;
      break;
   case value3:
      // block of code to be executed if expression == value3;
      break;
   default:
      // block of code to be executed if expression doesn't match any case;
}
```

- **expression:** This is the variable or value being evaluated.
- **case value N:** Each case defines a possible value for the expression. If the expression matches value N, the corresponding block of code is executed.
- **break:** This is used to terminate the switch block after a matching case has been executed. Without break, the code will "fall through" to the next case.
- **default:** This is an optional block that will execute if none of the case values match the expression.

Example:

```
#include <iostream>
using namespace std;
int main()
{
   int day = 5;
   switch(day)
{
      case 1:
         cout << "Monday";
         break;
      case 2:
         cout << "Tuesday";
         break;
      case 3:
    cout << "Wednesday";
         break;
```

```
        case 4:
      cout << "Thursday";
           break;
        case 5:
        cout << "Friday";
           break;
        case 6:
      cout << "Saturday";
           break;
        case 7:
        cout << "Sunday";
           break;
        default:
           cout<< "Invalid day number";
    }
    return 0;
}
```

Output:

Friday

Explanation:

- The day variable is set to 5.
- The switch statement checks the value of day:
- It matches with case 5, so it prints Friday.
- The break statement ensures that no further cases are checked once a match is found.

3.4 for Loop

A for loop in C++ is a control flow statement that allows you to execute a block of code repeatedly for a specific number of times or until a certain condition is met. It's especially useful when the number of iterations is known beforehand.

Syntax:

```
for (initialization; condition; increment/decrement)
{
   // Code to be executed repeatedly
}
```

Initialization: Sets up a loop counter. This step is executed once at the beginning of the loop.

Condition: Evaluated before each iteration. If it's true, the loop continues; if false, the loop terminates.

Increment/Decrement: Updates the loop counter after each iteration.

Example

Let's write a simple program that prints numbers from 1 to 5 using a for loop.

```
#include <iostream>
using namespace std;
int main()
{
   for (int i = 1; i <= 5; i++)
{
 cout << "Number: " << i << " \n";
   }
   return 0;
}
```

Explanation:

- A loop counter i is initialized to 1.
- Before each iteration, the condition i <= 5 is checked. If it's true, the loop body executes; otherwise, the loop terminates.
- After each iteration, the loop counter i is incremented by 1.

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

3.5 while loop

A while loop in C++ is a control flow statement that allows code to be executed repeatedly based on a given condition. It evaluates the condition before executing the loop body. If the condition is true, the loop body is executed; this process repeats until the condition becomes false.

Syntax:

```
while (condition)
{
   // Code to be executed repeatedly
}
```

condition: A boolean expression that is checked before each iteration. If it evaluates to true, the loop continues; otherwise, the loop stops.

The loop body can have one or more statements enclosed in curly braces {}. If the body has only one statement, the braces are optional.

How it Works

- The condition is checked.
- If the condition is true, the statements inside the loop are executed.
- After executing the loop body, the condition is checked again.
- This process continues until the condition becomes false.

Example:

```
#include <iostream>
using namespace std;
int main()
{
   int count = 1;
   while (count <= 5)
   {
      cout << "Count: " << count << "\n";
      count++;
   }
   return 0;
}
```

Explanation:

- The variable count is initialized to 1.
- The while loop checks if count <= 5. If true, it executes the cout statement.
- The count variable is incremented by 1 in each iteration.
- When count becomes greater than 5, the condition count <= 5 evaluates to false, and the loop terminates.

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

3.6 do-while loop

The do-while loop in C++ is a control flow statement that executes a block of code at least once, and then repeatedly executes it as long as the given condition is true. This loop is different from a while loop because the condition is checked after the code block is executed, ensuring that the code runs at least once.

Syntax:

```
do
{
   // Code to be executed
} while (condition);
```

- The code block inside the do section is executed first.
- The condition is evaluated after the execution of the code block.
- If the condition evaluates to true, the loop continues; otherwise, it stops.
- The loop guarantees at least one execution of the code block.

Example:

```
#include <iostream>
using namespace std;
int main()
{
   int count = 1;
   do
   {
      cout << "Count: " << count << "\n";
      count++;
   } while (count <= 5);
   return 0;
}
```

Explanation:

- The do block ensures that the statement cout << "Count: " << count << "\n"; executes at least once.
- The count variable starts at 1.
- After the code block is executed, the condition count <= 5 is checked.
- The loop continues until count becomes greater than 5.

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

4. Functions in C++

Functions are blocks of code designed to perform specific tasks. They allow developers to write modular, reusable code, improving readability and maintainability. A function is defined with a return type, a name, a set of parameters (optional), and a body that contains the code to execute.

- **Return Type:** Specifies the data type of the value the function returns. If no value is returned, the void keyword is used.
- **Function Name:** Identifier used to call the function.
- **Parameters:** Variables passed to the function as inputs, enclosed in parentheses.
- **Function Body:** Contains the code to execute, enclosed in curly braces {}.

Syntax:

```
return_type function_name(parameters)
{
   // Function body
   return value; // Optional
}
```

Example:

```
#include <iostream>
using namespace std;
int add(int a, int b)
{
   return a + b;
}
int main() {
   int x = 5, y = 10;
   cout << "Sum: " << add(x, y);
   return 0;
}
```

4.1 The Main Function

The main() function is the entry point of a C++ program. It is where program execution begins. Every C++ program must have exactly one main() function. The syntax of the main() function is as follows:

```
int main()
{
   // Code to execute
   return 0;
}
```

- **Return Type:** The return type is typically int, indicating that the function returns an integer to the operating system. A return value of 0 generally signifies successful execution.
- **Arguments (Optional):** The main() function can accept arguments for command-line inputs

```
int main(int argc, char* argv[])
{
   // argc: Argument count
   // argv: Array of argument strings
}
```

- **Execution:**The statements inside main() are executed sequentially.
- **Return Statement:**The return keyword is used to exit the program and optionally pass a status code. If omitted, the program implicitly returns 0.

```
#include <iostream>
using namespace std;
int main()
{
   cout << "Hello, World!";
   return 0;
}
```

This program outputs "Hello, World!" and then terminates, returning 0 to the operating system.

4.2 Function Prototyping

Function prototyping in C++ is the declaration of a function before its actual definition, specifying its name, return type, and parameter types. It informs the compiler about the function's interface, ensuring type checking during function calls. This helps detect errors such as mismatched arguments or incorrect return types early in the compilation process.

A typical function prototype has the following structure:

```
return_type function_name(parameter_type1, parameter_type2, ...);
```

Example:

```
int add(int, int); // Function prototype
int main()
{
   int result = add(5, 10); // Function call
   return 0;
```

```
}
int add(int a, int b)
{ // Function definition
   return a + b;
}
```

Benefits:

- Type Checking: Ensures that the arguments passed match the specified types.
- Flexibility: Allows functions to be defined after their usage in the program.
- Code Organization: Makes large programs easier to manage by separating function declarations, definitions, and usage.

4.3 Call by Reference

Call by Reference is a method of passing arguments to a function in C++ where the actual memory address of the variable is passed rather than its value. This allows the function to directly modify the original variable. Unlike Call by Value, which operates on a copy of the variable, Call by Reference ensures that any changes made to the parameter inside the function are reflected in the original variable. Call by Reference is implemented using reference variables with the & symbol.

Example:

```
#include <iostream>
using namespace std;
void swap(int *a, int *b)
{
   int temp = *a;
   *a = *b;
   *b = temp;
}


int main()
{
   int x = 5, y = 10;
   cout << "Before swapping: x = " << x << ", y = " << y << endl;
   swap(&x, &y);
   cout << "After swapping: x = " << x << ", y = " << y << endl;
   return 0;
```

```
}
```

Output:

Before swapping: x = 5, y = 10
After swapping: x = 10, y = 5

Advantages:

- Efficiency: Saves memory and execution time as no copy of the variable is created.
- Direct Modification: Useful when the function needs to update the original variable.
- Flexibility: Allows returning multiple values by modifying passed arguments.

4.4 Call by Value

In C++, Call by Value is a method of passing arguments to a function where a copy of the actual argument is made and passed to the called function. Changes made to the parameter inside the function do not affect the original argument.

Features:

- Isolation of Original Data: The original variable remains unchanged as the function operates on a separate copy of the data.
- No Side Effects: Since modifications are made on the copy, there is no unintended alteration of the original data.
- Memory Overhead: It may require more memory and time if the argument being passed is large, as a duplicate copy is created.

Example:

```
#include <iostream>
using namespace std;
void swap(int a, int b)
{
   int temp = a;
   a = b;
   b = temp;
}

int main()
{
   int x = 5, y = 10;
   cout << "Before swapping: x = " << x << ", y = " << y << endl;
   swap(x, y);
   cout << "After swapping: x = " << x << ", y = " << y << endl;
```

```
    return 0;
}
```

Output:

Before swapping: x = 5, y = 10
After swapping: x = 5, y = 10

As seen in the output, although the function swaps the values of a and b, the original values of x and y remain unchanged outside the function. This is because the function works on copies of the original values (Call by Value), not the actual variables.

4.5 Inline Function

An inline function in C++ is a function whose definition is directly inserted into the code at the point of function call. Instead of performing a regular function call, the compiler replaces the function call with the actual code of the function, which can improve performance by eliminating the overhead associated with function calls. To declare an inline function, you use the inline keyword before the function definition:

```
#include <iostream>
using namespace std;
inline int square(int x)
{
    return x * x;
}
int main()
{
    int num;
    cout << "Enter a number: ";
    cin >> num;
    int result = square(num);
    cout << "The square of " << num << " is " << result << endl;
    return 0;
}
```

Output:

Enter a number: 5
The square of 5 is 25

Advantages:

- Performance Improvement: Reduces the overhead of function calls, especially for small, frequently called functions.

• Faster Execution: By avoiding the function call overhead, inline functions can result in faster execution time for small functions.

4.6 Function Overloading

Function overloading is a feature in C++ that allows multiple functions to have the same name but with different parameters. The compiler differentiates between these functions based on the number and types of arguments passed to them. This helps to improve the readability and maintainability of the code, as the same function name can be used for performing similar operations on different types of data.

Example:

```
#include <iostream>
using namespace std;
int add(int a, int b)
{
   return a + b;
}
double add(double a, double b)
{
   return a + b;
}
int add(int a, int b, int c)
{
   return a + b + c;
}
int main()
{
   cout << "Sum of two integers: " << add(5, 10) << endl;
   cout << "Sum of two doubles: " << add(5.5, 7.2) << endl;
   cout << "Sum of three integers: " << add(1, 2, 3) << endl;
   return 0;
}
```

Output:

Sum of two integers: 15
Sum of two doubles: 12.7
Sum of three integers: 6

Explanation:

add(5, 10):

- This call matches the first overloaded add function, which takes two integer parameters (int add(int a, int b)).
- The integers 5 and 10 are passed as arguments.
- The function performs the addition: 5 + 10 = 15.
- The result, 15, is printed as the sum of two integers.

add(5.5, 7.2):

- This call matches the second overloaded add function, which takes two double parameters (double add(double a, double b)).
- The doubles 5.5 and 7.2 are passed as arguments.
- The function performs the addition: 5.5 + 7.2 = 12.7.
- The result, 12.7, is printed as the sum of two doubles.

add(1, 2, 3):

- This call matches the third overloaded add function, which takes three integer parameters (int add(int a, int b, int c)).
- The integers 1, 2, and 3 are passed as arguments.
- The function performs the addition: 1 + 2 + 3 = 6.
- The result, 6, is printed as the sum of three integers.

5. Classes and Objects

A class is a user-defined data type that acts as a blueprint for creating objects. It groups data members (variables) and member functions (methods) into a single unit, encapsulating both attributes and behaviors.

An object is an instance of a class that holds specific values and can invoke class methods. Objects occupy memory space and represent real-world entities with attributes and behaviors.

5.1 Specifying a Class

A class in C++ is defined using the class keyword, followed by the class name. Inside the class, you declare data members (variables) and member functions (methods). Access specifiers like public, private, and protected control access to these members. By default, class members are private if no access specifier is mentioned.

Syntax:

```
class ClassName
{
   // Access specifier
   public:
      data_type variable_name;          // Data member
      return_type function_name(parameters)
      {
         // Function definition
      }
};
```

- class Keyword: Declares a class.
- Class Name is a user-defined identifier (name of the class).
- Access Specifier: Controls the visibility of members (public, private, or protected).
- Public: Members are accessible from outside the class.
- Private: Members are only accessible within the class.
- Protected: Members are accessible within the class and derived classes.
- Data Members: Variables that store the class's data.
- Member Functions: Functions that operate on data members. Defined inside or outside the class.

Example:

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

```
class Rectangle
{
private:
   int length, width;
public:
  void setDimensions(int l, int w)
{
       length = l;
       width = w;
   }
   int calculateArea()
{
       return length * width;
   }
   void display()
{
       cout << "Length: " << length << ", Width: " << width << endl;
       cout << "Area: " << calculateArea() << endl;
   }
};
int main()
{
   Rectangle rect1;
   rect1.setDimensions(10, 5);
   rect1.display();
   return 0;
}
```

Explanation:

- Method Definitions Inside the Class:

The functions set Dimensions(), calculate Area(), and display() are fully defined within the class body, adhering to proper syntax.

- Object Creation and Method Calls:

The object rect1 uses the set Dimensions() method to assign values and display() to output details.

5.2 Defining Member Functions

In C++, member functions can be defined either inside or outside the class. The choice depends on the level of abstraction or clarity desired.

1. Defining Member Functions Inside the Class

In this method, the function's body is written directly within the class definition.

Syntax:

```
class Class Name {
public:
   void function Name(parameters) {
      // Function body
   }
};
```

Example:

```
class Rectangle {
public:
   int length, width;

   void setDimensions(int l, int w) {
      length = l;
      width = w;
   }
   int calculateArea() {
      return length * width;
   }
};
```

2. Defining Member Functions Outside the Class

To define functions outside the class, use the scope resolution operator (::) to specify that the function belongs to a particular class.

Syntax:

```
class Class Name {
public:
   void function Name(parameters);  // Declaration
};
// Function definition outside the class
void Class Name:: function Name(parameters) {
   // Function body
}
```

Example:

```
#include <io stream>
using namespace std;
```

```
class Rectangle {
public:
   int length, width;
   void set Dimensions(int l, int w);  // Declaration
   int calculate Area();               // Declaration
};
// Function definitions outside the class
void Rectangle::set Dimensions (int l, int w) {
   length = l;
   width = w;
}
int Rectangle::calculate Area() {
   return length * width;
}
int main() {
   Rectangle rect1;
   rect1.setDimensions(10, 5);
   cout << "Area: " << rect1.calculateArea() << endl;
   return 0;
}
```

Inside Class Definition:

- Inline functions (automatically treated as inline).
- Suitable for small, frequently used functions.

Outside Class Definition:

- Preferred for larger functions.
- Helps separate interface (class declaration) from implementation (function definition).

5.3 Nesting of Member Functions

Nesting of member functions means calling one member function from another member function within the same class. This allows for better code reuse and organization within the class.

Syntax:

```
class Class Name {
public:
   void function1() {
      // Function 1 logic
      function2();  // Calling function2 inside function1
```

```
    }
    void function2() {
        // Function 2 logic
    }
};
```

Example:

```
#include <iostream>
using namespace std;
class Rectangle {
private:
    int length, width;
public:
    void set Dimensions(int l, int w) {
        length = l;
        width = w;
        display Area();  // Nested function call
    }
    int calculate Area() {
        return length * width;
    }
     void display Area() {
        cout << "Area: " << calculate Area() << "\n";
    }
};
int main() {
    Rectangle rect;
    rect. set Dimensions(10, 5);
    return 0;
}
```

Output:

```
Area: 50
```

In a Rectangle class, the set Dimensions() function can call display Area(), which in turn calls calculate Area() to compute and display the area, reducing redundancy and maintaining clean logic.

5.4 Static Data Members and Static Member Functions

In C++, static data members and static member functions are shared across all objects of a class, meaning they belong to the class rather than individual objects.

1. Static Data Members:

A static data member is declared using the static keyword.

It is shared by all objects of the class and only one copy of it exists.

It must be defined outside the class for memory allocation.

Syntax:

```
class Class Name {
   static data_type variable_name;  // Declaration inside the class
};
// Definition outside the class
data_type Class Name:: variable_name = initial_value;
```

2. Static Member Functions:

A static member function is declared using the static keyword.

It can only access static data members and other static member functions.

It Can be called using the class name without creating an object.

Syntax:

```
class Class Name {
public:
   static return_type function_name(){
      // Function body
   }
};
```

Example:

```
#include <iostream>
using namespace std;
class Counter {
private:
   static int count;  // Static data member
public:
   static void increment() {  // Static member function
      count++;
   }
   static void display Count() {  // Static member function
      cout << "Count: " << count << '\n';
   }
};
```

```
// Definition and initialization of static data member
int Counter::count = 0;
int main() {
   Counter::display Count();  // Initial count
   Counter::increment();  // Increment count
   Counter::display Count();  // Display updated count
   Counter::increment();
   Counter::display Count();
   return 0;
}
```

Output:

Count: 0
Count: 1
Count: 2

Static Data Member (count):

- It is shared across all objects and initialized outside the class (int Counter::count = 0).

Static Member Functions (increment() and display Count()):

- They can be called using the class name (Counter::increment()), as they do not depend on any object.

5.5 Friend Function

A friend function is a function that is not a member of a class but is allowed to access the class's private and protected members. To declare a friend function, use the friend keyword inside the class. It helps in scenarios where a function needs to access private data from multiple classes.

Syntax:

```
class Class Name {
   private:
      data_type variable_name;
   public:
      friend return_type function_name(parameters);    // Friend function declaration
};
```

Example:

```
#include <iostream>
using namespace std;
class Sample {
private:
```

```
    int num1, num2;
public:
    Sample(int a, int b) {
        num1 = a;
        num2 = b;
    }
    // Friend function declaration
    friend float calculate Mean(Sample obj);
};
// Friend function definition
float calculate Mean(Sample obj) {
    return (obj.num1 + obj.num2) / 2.0;
}
int main() {
    Sample s(10, 20);
    cout << "Mean: " << calculate Mean(s) << "\n";
    return 0;
}
```

Output:

Mean: 15

Explanation:

Class Definition (Sample):

- Contains two private data members, num1 and num2.
- The friend function calculate Mean() is declared inside the class but defined outside.

Friend Function (calculate Mean):

- Takes an object of class Sample as an argument.
- Accesses the private data members and calculates their mean.

Object Creation:

- An object s is created with values 10 and 20.
- The friend function calculate Mean() computes and returns the mean.

6. Constructors and Destructors

6.1 Constructors

A constructor is a special member function that is automatically called when an object of a class is created. It has the same name as the class and has no return type. It is invoked automatically when an object is created. It can be overloaded to create multiple constructors with different parameters.

Types of Constructors:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

Destructors

A destructor is a special member function that is automatically called when an object goes out of scope or is explicitly deleted. It has the same name as the class, preceded by a tilde ~, and it has no return type or parameters.

- It is automatically called when an object is destroyed.
- Each class can have only one destructor, and it cannot be overloaded.
- It is useful for cleaning up resources like closing file handles or releasing dynamic memory.

1) Default Constructor

A Default Constructor is a constructor that takes no arguments and is automatically invoked when an object of a class is created. It is primarily used to initialize data members with default values. If no constructor is explicitly defined, C++ automatically provides a default constructor.

Syntax:

```
class Class Name {
public:
   Class Name() {
      // Constructor body (initialization code)
   }
};
```

Example:

```
#include <iostream>
using namespace std;
class Student {
public:
   int roll Number;
   // Default Constructor
   Student() {
```

```
        roll Number = 0;
        cout << "Default Constructor called\n";
    }
};
int main() {
    Student s1; // Custom default constructor is called automatically
    cout << "Roll Number: " << s1.rollNumber << "\n";
    return 0;
}
```

Output:

Default Constructor called
Roll Number: 0

Explanation:

- The constructor Student() initializes rollNumber to 0.
- When s1 is created, the default constructor is called, and "Default Constructor called" is displayed.
- The roll number is set to 0 as defined in the constructor.

2) Parameterized Constructor

A Parameterized Constructor is a constructor that accepts arguments to initialize the data members of a class when an object is created.

- Accepts one or more parameters to initialize object attributes.
- Called automatically when an object is created with arguments.
- Allows each object to have unique initial values.

Syntax:

```
class ClassName {
public:
    ClassName(dataType arg1, dataType arg2, ...) {
        // Constructor body to initialize data members
    }
};
```

Example:

```
#include <iostream>
using namespace std;
class Student {
public:
    int roll Number;
    string name;
    // Parameterized Constructor
```

```
    Student(int r, string n) {
        rollNumber = r;
        name = n;
    }
    void display() {
        cout << "Roll Number: " << rollNumber << ", Name: " << name << "\n";
    }
};
int main() {
    Student s1(101, "Alice");
    Student s2(102, "Bob");
    s1.display();
    s2.display();
    return 0;
}
```

Output:

Roll Number: 101, Name: Alice
Roll Number: 102, Name: Bob

Explanation:

The constructor Student(int r, string n) takes two arguments, r (roll number) and n (name).
When s1 and s2 objects are created, arguments are passed to initialize rollNumber and name for each object.

3) Copy Constructor

A copy constructor is a type of constructor that initializes an object using another object of the same class. It is called when:

- An object is initialized from another object of the same class.
- An object is passed by value to a function.
- An object is returned by value from a function.

Syntax:

```
Class Name (const Class Name &obj) {
    // Copy the values from obj to the new object
}
```

- Parameter: It takes a reference to a constant object of the same class (const Class Name & obj) to avoid unnecessary copying and to ensure the original object is not modified.
- Purpose: Copies the member variables of one object into another.

Example:

```
#include <iostream>
using namespace std;
class code {
   int id;
public:
   code() { }
   code(int a) { id = a; }
   code(code &x) {                 // copy constructor
      id = x.id;                 // copy in the value
   }
   void display(void) {
      cout << id;
   }
};
int main() {
   code A(100);
   code B(A);
   code C = A;
   code D;
   D = A;
   cout << "\n id of A: "; A.display();
   cout << "\n id of B: "; B.display();
   cout << "\n id of C: "; C.display();
   cout << "\n id of D: "; D.display();
   return 0;
}
```

Output:

```
id of A: 100
id of B: 100
id of C: 100
id of D: 100
```

Constructors:

- Default constructor: code().
- Parameterized constructor: code(int a) initializes the id variable with a value.
- Copy constructor: code(code &x) creates a copy of the object x.

Object Creation:

- A is created and initialized using the parameterized constructor.
- B and C are created using the copy constructor.
- D is created using the default constructor but later assigned the value of A (no copy constructor is called for assignment).

Output: The output will display the id values of all objects.

4) Constructor Overloading

Constructor Overloading in C++ means defining multiple constructors in the same class, each with a different number or type of parameters. This allows you to create objects with varying initial values, providing flexibility in object creation.

- Multiple Constructors: A class can have more than one constructor.
- Different Signatures: Each constructor must have a unique signature (different number or types of arguments).
- Automatic Invocation: The appropriate constructor is automatically called based on the arguments passed during object creation.
- Flexibility: It allows for object initialization in multiple ways.

Example:

```
#include <iostream>
using namespace std;
class Calculator {
public:
   Calculator(int a, int b) {
      cout << "Sum of two integers: " << (a + b) << "\n";
   }
   Calculator(float a, float b) {
      cout << "Sum of two floats: " << (a + b) << "\n";
   }
   Calculator(int a, int b, int c) {
      cout << "Sum of three integers: " << (a + b + c) << "\n";
   }
};
```

```
int main() {
    Calculator c1(10, 20); // Calls constructor with two integers
    Calculator c2(1.5f, 2.5f); // Calls constructor with two floats
    Calculator c3(5, 10, 15); // Calls constructor with three integers
    return 0;
}
```

Output:

```
Sum of two integers: 30
Sum of two floats: 4
Sum of three integers: 30
```

6.2 Destructor

A Destructor is a special member function of a class that destroys objects when they go out of scope or are explicitly deleted. It is used to release resources such as memory, file handles, or network connections that were acquired during the object's lifetime.

- Same Name as Class: The name of the destructor is the same as the class name, but it starts with a tilde ~ (e.g., ~Class Name()).
- No Arguments: Destructors cannot have parameters or a return type.
- Automatic Invocation: Called automatically when the object goes out of scope or is deleted.
- No Overloading: You can define only one destructor for a class, and it cannot be overloaded.
- Use Case: It is commonly used to release resources like dynamic memory (delete), close files, or disconnect from databases.

Syntax:

```
class Class Name {
public:
    ~Class Name() {
        // Code to release resources
    }
};
```

Example:

```
#include <iostream>
using namespace std;
class Student {
public:
    Student() {
```

```
        cout << "Constructor called\n";
    }
    // Destructor
    ~Student() {
        cout << "Destructor called\n";
    }
};
int main() {
    Student s1; // Constructor is called automatically
    cout << "Inside main function\n";
    return 0; // Destructor is called automatically when main() ends
}
```

Output:

Constructor called
Inside main function
Destructor called

Explanation:

- The constructor is called automatically when the object s1 is created.
- The destructor is called automatically when the program ends, and s1 goes out of scope.

7.Operator Overloading

Operator overloading is a technique that allows customizing the behavior of operators for user-defined data types. By overloading operators, you can define how they should operate on objects of your class.

7.1 Overloading Unary Operators

Overloading Unary Operators refers to customizing the behavior of operators that operate on a single operand.

Syntax:

```
class ClassName {
public:
   ReturnType operator symbol (Arguments) {
      // Custom definition for the operator
   }
};
```

Example:

```
#include <iostream>
using namespace std;
class Number {
public:
   int value;
   Number(int v) {
      value = v;
   }
   // Overloading unary '-' operator
   void operator - () {
      value = -value;
   }
   void display() {
      cout << "Value: " << value << "\n";
   }
};
int main() {
   Number num(10);
   cout << "Original: ";
   num.display();
   -num; // Calls the overloaded unary '-' operator
   cout << "After negation: ";
   num.display();
```

```
    return 0;
}
```

Output:

Original: Value: 10
After negation: Value: -10

The unary minus (-num) changes num.value from 10 to -10 as specified in the overloaded operator-() method.

7.2 Overloading Binary Operator

Binary Operator Overloading in C++ allows programmers to redefine the behavior of binary operators (operators that operate on two operands) for user-defined classes.

Syntax:

```
class ClassName {
public:
    ReturnType operator symbol (Arguments) {
        // Custom definition for the operator
    }
};
```

Example:

```
#include <iostream>
using namespace std;
class Complex {
public:
    int real, imag;
    Complex(int r = 0, int i = 0) {
        real = r;
        imag = i;
    }
    // Overloading the '+' operator
    Complex operator + (const Complex& obj) {
        Complex result;
        result.real = real + obj.real;
        result.imag = imag + obj.imag;
        return result;
    }
    void display() {
        cout << real << " + " << imag << "i\n";
    }
```

```
};
int main() {
    Complex c1(3, 4), c2(1, 2);
    Complex c3 = c1 + c2; // Calls overloaded '+' operator
    cout << "Sum of complex numbers: ";
    c3.display();
    return 0;
}
```

Output:

Sum of complex numbers: 4 + 6i

Here, c1 + c2 calls the operator+() function, and the real and imaginary parts of the complex numbers are added. The result is stored in c3.

8.Inheritance

Inheritance is a key concept of Object-Oriented Programming (OOP) that allows a class (derived class) to acquire the properties and behaviors (data members and member functions) of another class (base class). It promotes code reusability and establishes a hierarchical relationship between classes.

- Base Class: The class whose properties are inherited.
- Derived Class: The class that inherits the properties and methods of the base class.
- Access Specifiers: public, protected, and private control access to the base class members in the derived class.

Types of Inheritance in C++:

1. Single Inheritance: A derived class inherits from only one base class.
2. Multiple Inheritance: A derived class inherits from multiple base classes.
3. Multilevel Inheritance: A class is derived from a class, which is also derived from another class.
4. Hierarchical Inheritance: Multiple derived classes inherit from a single base class.
5. Hybrid Inheritance: A combination of two or more types of inheritance.

8.1 Defining a Derived Class

A derived class in C++ is a class that inherits properties and behaviors (data members and member functions) from another class, called the base class. This allows code reusability and establishes an "is-a" relationship between the derived and base classes.

Syntax:

```
class DerivedClass : access_specifier BaseClass {
   // Members (variables and methods) of DerivedClass
};
```

- DerivedClass: The new class that inherits from the BaseClass.
- BaseClass: The class from which properties and behaviors are inherited.
- access_specifier: Specifies how the members of the base class will be accessible in the derived class. It can be: public, protected and private.

8.2 Single Inheritance

Single inheritance is a type of inheritance in which a derived class inherits from only one base class. This allows the derived class to reuse the properties (data members) and behaviors (member functions) of the base class while adding its own unique features.

Syntax:

```
class BaseClass {
   // Members of the base class (data and methods)
};
class DerivedClass : access_specifier BaseClass {
   // Members of the derived class (additional data and methods)
};
```

Example:

```
#include <iostream>
using namespace std;
// Base Class
class InputNumbers {
public:
   int num1, num2;
   void getInput() {
      cout << "Enter the first number: ";
      cin >> num1;
      cout << "Enter the second number: ";
      cin >> num2;
   }
};
// Derived Class
class SumNumbers : public InputNumbers {
public:
   void calculateSum() {
      int sum = num1 + num2;
      cout << "The sum of " << num1 << " and " << num2 << " is: " << sum
<< endl;
   }
};
int main() {
   SumNumbers obj;
   obj.getInput();
   obj.calculateSum();
```

```
    return 0;
}
```

Output:

Enter the first number: 5
Enter the second number: 10
The sum of 5 and 10 is: 15

Base Class:

- InputNumbers has two member variables num1 and num2.
- The getInput() method takes input from the user for the two numbers.

Derived Class:

- SumNumbers inherits from InputNumbers using public inheritance.
- The calculateSum() method adds the two numbers from the base class and prints the result.

Main Function:

- An object obj of the SumNumbers class is created.
- The object calls getInput() to read two numbers.
- It then calls calculateSum() to compute and display the sum.

8.3 Multilevel Inheritance

Multilevel inheritance is a type of inheritance where a sub-class is derived from another sub-class.

Syntax:

```
class ClassA { // Base class
    // Members of ClassA
};
class ClassB : public ClassA { // Derived class B from ClassA
    // Members of ClassB
};
class ClassC : public ClassB { // Derived class C from ClassB
    // Members of ClassC
};
```

Example:

```
#include <iostream>
using namespace std;
class Input {
public:
    int num1, num2;
    void get Input() {
```

```
        cout << "Enter two numbers: ";
        cin >> num1 >> num2;
    }
};
class Addition : public Input {
public:
    int add() {
        return num1 + num2;
    }
};
class Average : public Addition {
public:
    float calculate Average() {
        return (num1 + num2) / 2.0;
    }
};
int main() {
    Average avg;
    avg. get Input();
     int sum = avg.add();
    cout << "Sum of the numbers: " << sum << endl;
       float average = avg. calculate Average();
    cout << "Average of the numbers: " << average << endl;
        return 0;
}
```

Output:

Enter two numbers: 10 20
Sum of the numbers: 30
Average of the numbers: 15

Input Class:

- This class has a method get Input() to take two numbers as input from the user.

Addition Class:

- This class inherits from the Input class and uses the numbers entered by the user.
- The add() method adds the two numbers together.

Average Class:

- This class inherits from the Addition class.
- The calculate Average() method calculates the average of the two numbers.

Main Function:

- An object of the Average class is created.
- First, get Input() is called to take user input.
- Then, the add() method is used to calculate the sum of the numbers.
- Finally, the calculate Average() method computes the average and displays it.

8.4 Multiple Inheritance

Multiple inheritance in C++ is a feature that allows a class to inherit properties and behaviors from more than one base class. This means that a derived class can inherit attributes and methods from two or more classes, allowing the derived class to combine the functionalities of all the base classes.

Syntax:

```
class BaseClass1 {
   // Members of BaseClass1
};
class BaseClass2 {
   // Members of BaseClass2
};
// Derived class inherits from two base classes
class Derived Class : public BaseClass1, public BaseClass2 {
   // Members of Derived Class
};
```

- BaseClass1 and BaseClass2 are base classes.
- Derived Class is the class that inherits from both BaseClass1 and BaseClass2.
- public inheritance means that the public members of the base classes become public members of the derived class.
- The derived class can access members from both base classes and can have its own additional members as well.

Example:

```
#include <io stream>
using namespace std;
// Base Class 1: Addition
class Addition {
public:
   int num1, num2;
   void get Input() {
      cout << "Enter two numbers: ";
      cin >> num1 >> num2;
   }
```

```
    int add() {
        return num1 + num2;
    }
};
// Base Class 2: Subtraction
class Subtraction {
public:
    int num1, num2;
    void get Input() {
        cout << "Enter two numbers: ";
        cin >> num1 >> num2;
    }
    int subtract() {
        return num1 - num2;
    }
};
class Calculator : public Addition, public Subtraction {
public:
    void display Results() {
        cout << "Addition of the numbers: " << add() << endl;
        cout << "Subtraction of the numbers: " << subtract() << endl;
    }
};
int main() {
    Calculator calc;
    calc.get Input();
    calc. display Results();
    return 0;
}
```

Output:

Enter two numbers: 15 5
Addition of the numbers: 20
Subtraction of the numbers: 10

Addition Class:

- This class has a method add() that adds the two numbers (num1 and num2).

Subtraction Class:

- This class has a method subtract() that subtracts num2 from num1.

Calculator Class:

- The Calculator class inherits from both the Addition and Subtraction classes.
- The display Results() method calls both add() and subtract() methods and displays the results.

Main Function:

- An object of Calculator is created.
- The get Input() method is called to take the numbers as input from the user.
- Finally, display Results() is called to display the results of both addition and subtraction.

8.5 Hierarchical Inheritance

In Hierarchical Inheritance, one base class is inherited by multiple derived classes. All the derived classes share the same base class, meaning they have access to the members of the base class.

```
Syntax:
class Base Class {   // Base class
        // Members of Base Class
};
class DerivedClass1 : public Base Class {   // Derived class 1 inherits Base Class
  // Members of DerivedClass1
};
class DerivedClass2 : public Base Class {   // Derived class 2 inherits Base Class
  // Members of DerivedClass2
};
```

- Base Class: The base class contains common members that are shared by all derived classes.
- DerivedClass1 and DerivedClass2: These derived classes inherit from Base Class and can access its public and protected members.

Example:

```
#include <iostream>
using namespace std;
class Numbers {
protected:
   int num1, num2;
public:
    void inputNumbers() {
      cout << "Enter two numbers: ";
      cin >> num1 >> num2;
   }
```

```
};
class Addition : public Numbers {
public:
     void add() {
       cout << "Sum: " << num1 + num2 << endl;
    }
};
class Subtraction : public Numbers {
public:
    void subtract() {
       cout << "Difference: " << num1 - num2 << endl;
    }
};
int main() {
    Addition add Obj;
    Subtraction sub Obj;
    add Obj. input Numbers();
    subObj.inputNumbers();
    addObj.add();
    subObj.subtract();
    return 0;
}
```

Output:

```
Enter two numbers: 10 5
Enter two numbers: 10 5
Sum: 15
Difference: 5
```

Base Class (Numbers):

- The Numbers class contains a method input Numbers() to take input for two numbers. These numbers are stored in the protected variables num1 and num2.

Drived Class for Addition (Addition):

- Inherits from Numbers and has a method add() to perform the addition of num1 and num2.

Derived Class for Subtraction (Subtraction):

- Inherits from Numbers and has a method subtract() to perform the subtraction of num1 and num2.

Main Function:

- Objects of Addition and Subtraction are created.
- The input Numbers() method is called for both objects to take the same input.
- The add() and subtract() methods are called to perform the respective operations.

8.6 Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.

Example:

```
#include <iostream>
using namespace std;
class Academic Marks {
protected:
   int subject1, subject2;
public:
   void input Marks() {
      cout << "Enter marks for subject 1: ";
      cin >> subject1;
      cout << "Enter marks for subject 2: ";
      cin >> subject2;
   }
};
class Sum : public Academic Marks {
public:
   int total Marks() {
      return subject1 + subject2;
   }
};
class Sports Marks {
protected:
   int sports Marks;
public:
   void input Sports Marks() {
      cout << "Enter sports marks: ";
      cin >> sports Marks;
   }
};
class Final Result : public Sum, public Sports Marks {
public:
```

```
    void input All Marks() {
        input Marks();
        input Sports Marks();
    }
    void calculate Grand Total() {
        int total = total Marks() + sports Marks;
        cout << "Total marks (subjects + sports): " << total << endl;
    }
};
int main() {
    Final Result result;
    result. input All Marks();
    result. calculate Grand Total();
    return 0;
}
```

Output:

```
Enter marks for subject 1: 85
Enter marks for subject 2: 90
Enter sports marks: 95
Total marks (subjects + sports): 270
```

In this example, Final Result inherits from both Sum (which inherits Academic Marks) and Sports Marks, thus combining multiple inheritance with multilevel inheritance.

9. Virtual Function

A virtual function is a member function in a base class that is declared using the virtual keyword. It is meant to be overridden in derived classes to achieve polymorphism. When you call a virtual function through a pointer or reference to the base class, the version of the function that gets executed depends on the type of the object pointed to, not the type of the pointer. This is called dynamic dispatch.

Example:

```
#include <iostream>
using namespace std;
class Base {
public:
   void display() {
      cout << "\nDisplay base";
   }
   virtual void show() { // virtual function
      cout << "\nShow base";
   }
};
class Derived : public Base {
public:
   void display() {
      cout << "\nDisplay derived";
   }
   void show() {
      cout << "\nShow derived";
   }
};
int main() {
   Base B;
   Derived D;
   Base* bptr;
   cout << "\n\nbptr points to Base\n";
   bptr = &B;
   bptr->display();  // calls Base version
   bptr->show();     // calls Base version
   cout << "\n\nbptr points to Derived\n";
   bptr = &D;
   bptr->display();  // calls Base version
```

```
    bptr->show();     // calls Derived version
    return 0;
}
```

Output:

```
bptr points to Base
Display base
Show base
bptr points to Derived
Display base
Show derived
```

1. Base Class: Contains a display() function (non-virtual) and a show() function (virtual).
2. Derived Class: Overrides the show() function but does not override the display() function.
3. Main Function:

- A pointer to Base (bptr) is used to point to both Base and Derived objects.
- When bptr->display() is called, it calls the display() function from the Base class.
- When bptr->show() is called, it calls the appropriate show() function based on the type of the object (Base or Derived) due to dynamic binding (virtual function mechanism).

10. Pure Virtual Function

A pure virtual function is a function that has no implementation in the base class and must be implemented by any derived class. It is used to make a class abstract, meaning it cannot be instantiated directly. The syntax for a pure virtual function is: virtual return_type function_name(parameters) = 0;

Example:

```
#include <iostream>
using namespace std;
class Shape {
public:
   // Pure virtual function
   virtual void draw() = 0; // No implementation in the base class
   virtual ~Shape() {}
};
class Circle : public Shape {
public:
   void draw() override {
      cout << "Drawing a Circle." << endl;
   }
};
class Square : public Shape {
public:
   void draw() override {
      cout << "Drawing a Square." << endl;
   }
};
int main() {
   Shape* shape1 = new Circle();
   Shape* shape2 = new Square();
   shape1->draw();
   shape2->draw();
   delete shape1;
   delete shape2;
   return 0;
}
```

Output:

Drawing a Circle.
Drawing a Square.

- Pure Virtual Function: It forces derived classes to provide an implementation of the function, thus ensuring that the derived class has its own version of the method.
- Abstract Class: A class with at least one pure virtual function is considered an abstract class, meaning it cannot be instantiated directly.

Rules for Pure Virtual Function:

1) The virtual functions must be members of some class.
2) They cannot be static members.
3) They are accessed by using object pointers.
4) A virtual function can be a friend of another class.
5) A virtual function in a base class must be defined, even though it may not be used.
6) The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7) We cannot have virtual constructors, but we can have virtual destructors.
8) While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
9) When a base pointer points to a derived class, incrementing or decrementing it will not make it point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
10) If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

11. Working with Files

In C++, file handling is essential for storing and retrieving data. It allows programs to interact with external files, making it possible to read data from a file or write data to a file. File handling in C++ is done through streams, which are objects used to read from or write to files.

11.1 Classes for File Stream Operations

C++ provides a set of built-in classes in the <fstream> header file to handle file operations. The main classes used for file handling are:

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): Used for both reading and writing to a file.

These classes allow the use of stream operators (<< for writing and >> for reading) with files, just like standard input/output streams.

11.2 Basic File Operations in C++

The following are the basic file operations that can be performed:

1. Opening a File: Before performing any operations on a file, the file must be opened using one of the above classes.
2. Reading from a File: Once the file is open, data can be read from it using the stream extraction operator (>>).
3. Writing to a File: Data can be written to a file using the stream insertion operator (<<).
4. Closing a File: After completing file operations, it is important to close the file to free up system resources.

File Modes

When opening a file, you can specify different file modes. These modes determine how the file should be opened. Some common file modes are:

- ios::in: Open the file for reading.
- ios::out: Open the file for writing.
- ios::app: Open the file for appending data to the end.
- ios::binary: Open the file in binary mode (used for non-text files).
- ios::ate: Open the file and move the pointer to the end of the file.
- ios::trunc: Truncate the file to zero length if it exists (deletes the content of the file before writing).

Example:

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

```
using namespace std;
int main() {
    ofstream outputFile("example.txt");
    if (!outputFile) {
        cout << "File could not be opened!" << endl;
        return 1;
    }
    outputFile << "Hello, this is a file handling example!" << endl;
    outputFile.close();
    ifstream inputFile("example.txt");
    if (!inputFile) {
        cout << "File could not be opened!" << endl;
        return 1;
    }
    string line;
    while (getline(inputFile, line)) {
        cout << line << endl;
    }
    inputFile.close();
    return 0;
}
```

- Writing to a file: The ofstream class is used to create a file (example.txt) and write some text into it.
- Reading from a file: The ifstream class is used to open the file and read its contents line by line.
- Closing the file: It is important to close the file after performing the read or write operations to ensure data is properly saved and resources are released.

11.3 Detecting End-of-File

When working with files, you often need to determine when you have reached the end of the file (EOF). The end-of-file condition occurs when there is no more data to read from the file.In C++, the ifstream (input file stream) class provides several ways to detect the end of a file:

- Using the eof() member function:The eof() function returns true when the end of the file is reached.
- Using while loops:Loops are typically used to process the file's content until no more data is available.

Example:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
   ifstream inputFile("example.txt");
   if (!inputFile.is_open()) {
      cout << "Error: Unable to open the file.\n";
      return 1;
   }
   cout << "Reading file content:\n";
   string line;
   while (!inputFile.eof()) {  // Continue reading until EOF
      getline(inputFile, line);
      if (!line.empty()) {  // To avoid printing empty lines at EOF
         cout << line << endl;
      }
   }
   inputFile.close();
   return 0;
}
```

Input File (example.txt):

```
Hello, World!
This is an example file.
It demonstrates end-of-file detection.
```

Output:

```
Reading file content:
Hello, World!
This is an example file.
It demonstrates end-of-file detection.
```

Explanation:

- inputFile.eof():
  - The eof() function returns true when the input pointer reaches the end of the file.
  - The while (!inputFile.eof()) loop ensures that the program reads lines until it hits the end of the file.
- getline():
  - Reads one line from the file into the string line.
- File Closure:

o Always close the file using inputFile.close() after reading or writing to release system resources.

11.4 Sequential Input and Output Operations

Sequential input and output operations involve reading data from a file or writing data to a file in a sequential manner, meaning data is processed in the order it appears.

Steps for Sequential Input and Output Operations

1. Open the File:

- Use ifstream for reading files.
- Use ofstream for writing files.

2. Perform Operations:

- Use input functions like >>, getline(), or output functions like << to read/write data.

3. Close the File:

- Always close the file after performing operations using the .close() method.

Example:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    // Writing to a file
    ofstream outputFile("data.txt");
    if (!outputFile) {
        cout << "Error: Could not create the file.\n";
        return 1;
    }
    cout << "Writing to the file...\n";
    outputFile << "Name: Dhamu\n";
    outputFile << "Age: 25\n";
    outputFile << "Department: Computer Science\n";
    outputFile.close(); // Close the file after writing
    cout << "Data written successfully.\n";
    // Reading from the file
    ifstream inputFile("data.txt");
    if (!inputFile) {
        cout << "Error: Could not open the file.\n";
        return 1;
```

```
    }
    cout << "\nReading from the file:\n";
    string line;
    while (getline(inputFile, line)) {
        cout << line << endl; // Display each line
    }
    inputFile.close(); // Close the file after reading
    return 0;
}
```

Output:

When Writing:
Writing to the file...
Data written successfully.

When Reading:
Reading from the file:
Name: Dhamu
Age: 25
Department: Computer Science

1. Writing to the File:
- ofstream outputFile("data.txt"); creates a new file named data.txt.
- The << operator writes data sequentially to the file.
- outputFile.close(); closes the file after writing.

2. Reading from the File:
- ifstream inputFile("data.txt"); opens the file for reading.
- getline(inputFile, line) reads one line at a time sequentially from the file.
- The while loop continues until the end of the file is reached.
- inputFile.close(); closes the file after reading.

12.Updating a File

Updating a file means modifying its contents after it has already been created. This could involve:

- Adding new data at the end of the file (appending).
- Modifying existing data in specific positions (random access).
- Overwriting the entire file with new data.

12.1 Random Access

Random access refers to the ability to read from or write to any position in a file without sequentially reading or writing all preceding data. This is particularly useful when you need to update specific parts of a file, rather than rewriting the entire file.

Key Functions for Random Access in C++:

1. seekg(): Moves the file pointer for input operations (reading).
2. seekp(): Moves the file pointer for output operations (writing).
3. tellg(): Returns the current position of the file pointer for input.
4. tellp(): Returns the current position of the file pointer for output.

Example: Updating a File with Random Access

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
   // Create and write initial data to the file
   fstream file("example.txt", ios::in | ios::out | ios::trunc); // Open file for both input and output
   if (!file) {
      cout << "Error: Could not open the file.\n";
      return 1;
   }
   // Writing initial data
   file << "Line 1: Hello\n";
   file << "Line 2: World\n";
   file << "Line 3: C++ File Handling\n";
   // Move the file pointer to update the second line
   file.seekp(8, ios::beg); // Move the pointer to the 8th byte from the beginning
   file << "Updated World";
   // Reading the updated file
   file.seekg(0, ios::beg); // Move the pointer back to the beginning for reading
   cout << "\nUpdated File Content:\n";
```

```
    string line;
    while (getline(file, line)) {
        cout << line << endl;
    }
    file.close(); // Close the file
    return 0;
}
```

Output:

Initial File Content:

Line 1: Hello
Line 2: World
Line 3: C++ File Handling

Updated File Content:

Line 1: Hello
Line 2: Updated World
Line 3: C++ File Handling

1. File Creation and Initialization:

- The file example.txt is created or opened using fstream with the ios::in, ios::out, and ios::trunc modes.
- Initial data is written using file <<.

2. Random Access Update:

- seekp(8, ios::beg) moves the write pointer to the 8th byte from the beginning of the file.
- The string "Updated World" overwrites the original data starting from this position.

3. Reading the Updated File:

- seekg(0, ios::beg) moves the read pointer to the beginning of the file.
- The updated content is read line by line using getline(file, line).

4. Closing the File:

- file.close() closes the file after all operations.

12.2 Error Handling during File Operations

Error handling is crucial during file operations to ensure the program can gracefully handle situations such as missing files, lack of permissions, or file corruption. In C++, file streams (fstream, ifstream, ofstream) provide built-in mechanisms to detect and handle such errors.

Common Causes of File Errors

1. File not found.
2. File cannot be opened due to permissions or being in use.
3. File reading or writing errors (e.g., disk full).
4. End-of-file (EOF) reached unexpectedly.
5. Incorrect file format.

Error Handling Mechanisms

C++ provides the following ways to check for errors:

1. fail():Returns true if an input or output operation fails.
2. Example: File not found or an invalid format.
3. eof():Returns true if the end-of-file is reached.
4. bad():Returns true if a serious error occurs (e.g., hardware failure).
5. good():Returns true if no errors have occurred.
6. exceptions():Throws exceptions for specific error states when enabled.

Example:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
    // 1. Attempt to open a non-existing file for reading
    ifstream inputFile("nonexistent.txt");
    if (inputFile.fail()) {
        cerr << "Error: File 'nonexistent.txt' could not be opened (fail).\n";
    } else {
        cout << "File 'nonexistent.txt' opened successfully.\n";
    }
    // 2. Writing to a file
    ofstream outputFile("example.txt");
    if (outputFile.good()) {
        outputFile << "Hello, this is a test file!\n";
        outputFile << "This file contains some random text.\n";
    }
    outputFile.close();
    // 3. Reading the file
    ifstream input("example.txt");
    if (!input) {
        cerr << "Error: Could not open the file 'example.txt' for reading (fail).\n";
```

```
        return 1;
    } else {
        cout << "File 'example.txt' opened successfully for reading.\n";
    }
    // 4. Reading and checking for file states
    string line;
    while (getline(input, line)) {
        if (input.bad()) {
            cerr << "Error: Serious error occurred while reading the file (bad).\n";
            break;
        } else if (input.fail()) {
            cerr << "Error: Non-serious failure occurred while reading the file
(fail).\n";
            break;
        }
        cout << line << endl;
    }
    // 5. Check for end of file
    if (input.eof()) {
        cout << "Reached the end of the file (eof).\n";
    }
    input.close();
    return 0;
}
```

Output:

Case 1: File Does Not Exist

Error: File 'nonexistent.txt' could not be opened (fail).
File 'example.txt' opened successfully for reading.
Hello, this is a test file!
This file contains some random text.
Reached the end of the file (eof).

Case 2: File Exists and is Read Successfully

File 'nonexistent.txt' opened successfully.
File 'example.txt' opened successfully for reading.
Hello, this is a test file!
This file contains some random text.
Reached the end of the file (eof).

1. File Opening Check (fail()):

- When attempting to open nonexistent.txt, if (inputFile.fail()) checks if the file was successfully opened.

2. File Write Operation (good()):

- if (outputFile.good()) ensures that the file is ready for writing.
- If the file could not be opened, it prints an error message.

3. File Read Operation (fail() and bad()):

- if (!input) verifies that example.txt is successfully opened for reading.
- The loop reads lines from the file.
- If bad() returns true, a critical error like disk failure is detected.
- If fail() returns true, a non-serious issue is flagged, such as incorrect input format.

4. End-of-File Check (eof()):

- Once the loop finishes, if (input.eof()) checks if the end of the file was reached.

12.3 Command Line Arguments

Command-line arguments are parameters passed to the main() function when a program is executed. These arguments allow users to provide input values directly from the command line.

Syntax:

int main(int argc, char *argv[])

- argc: Argument count — It stores the number of command-line arguments, including the name of the program.
- argv: Argument vector — It is an array of C-style strings (character pointers) representing the individual command-line arguments.
  - argv[0]: The name or path of the program being executed.
  - argv[1]: The first argument provided by the user.
  - argv[2]: The second argument, and so on.
  - argv[argc-1]: The last argument.

Example:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(int argc, char *argv[]) {
   if (argc != 3) {
      cerr << "Error: Please provide two numbers as command-line arguments.\n";
      return 1;
```

```
    }
    int num1 = atoi(argv[1]);
    int num2 = atoi(argv[2]);
    int sum = num1 + num2;
    cout << "The sum of " << num1 << " and " << num2 << " is " << sum <<
endl;
    return 0;
}
```

Output:

Input (Run Parameters):
10 20
Output (Console Window):
The sum of 10 and 20 is 30

1. Write a C++ program to demonstrate Class and Objects

```
#include <iostream>
using namespace std;
class Rectangle
{
private:
int length, width;
public:
void setDimensions(int l, int w)
{
length = l;
width = w;
}
int calculateArea()
{
return length * width;
}
void display()
{
cout << "Length: " << length << ", Width: " << width << endl;
cout << "Area: " << calculateArea() << endl;
}
};
int main()
{
```

```
Rectangle rect1;
rect1.setDimensions(10, 5);
rect1.display();
return 0;
}
```

2. Write a C++ program to demonstrate function overloading

```
#include <iostream>
using namespace std;
int add(int a, int b)
{
return a + b;
}
double add(double a, double b)
{
return a + b;
}
int add(int a, int b, int c)
{
return a + b + c;
}
int main()
{
cout << "Sum of two integers: " << add(5, 10) << endl;
cout << "Sum of two doubles: " << add(5.5, 7.2) << endl;
cout << "Sum of three integers: " << add(1, 2, 3) << endl;
return 0;
}
```

3. Write a C++ program to demonstrate the Friend Functions

```
#include <iostream>
using namespace std;
class Sample {
private:
int num1, num2;
public:
Sample(int a, int b) {
num1 = a;
num2 = b;
}
friend float calculateMean(Sample obj);
};
float calculateMean(Sample obj) {
return (obj.num1 + obj.num2) / 2.0;
}
int main() {
Sample s(10, 20);
cout << "Mean: " << calculateMean(s) << "\n";
return 0;
}
```

4. Write a C++ program to demonstrate Parameterized Constructor, Copy Constructor and Destructor

```
#include <iostream>
using namespace std;
class Student {
public:
   int rollNumber;
   string name;
   Student(int r, string n) {
      rollNumber = r;
      name = n;
   }
   void display() {
      cout << "Roll Number: " << rollNumber << ", Name: " << name << "\n";
   }
};
int main() {
   Student s1(101, "Alice");
```

```
    Student s2(102, "Bob");
    s1.display();
    s2.display();
    return 0;
}


#include <iostream>
using namespace std;
class code {
    int id;
public:
    code() { }
    code(int a) { id = a; }
    code(code &x) {
        id = x.id;
    }
    void display(void) {
        cout << id;
    }
};
int main() {
    code A(100);
    code B(A);
    code C = A;
    code D;
    D = A;
    cout << "\n id of A: "; A.display();
    cout << "\n id of B: "; B.display();
    cout << "\n id of C: "; C.display();
    cout << "\n id of D: "; D.display();
    return 0;
}

#include <iostream>
using namespace std;
class Student {
public:
    Student() {
```

```
        cout << "Constructor called\n";
    }
    ~Student() {
        cout << "Destructor called\n";
    }
};
int main() {
    Student s1;
    cout << "Inside main function\n";
    return 0;
}
```

5. Write a program to demonstrate operator overloading for Unary operator

```
#include <iostream>
using namespace std;
class Number {
public:
    int value;
    Number(int v) {
        value = v;
    }
    void operator - () {
        value = -value;
    }
    void display() {
        cout << "Value: " << value << "\n";
    }
};
int main() {
    Number num(10);
    cout << "Original: ";
    num.display();
    -num;
    cout << "After negation: ";
    num.display();
    return 0;
}
```

6. Write a program to demonstrate operator overloading for Binary operator

```
#include <iostream>
using namespace std;
class Complex {
public:
   int real, imag;
   Complex(int r = 0, int i = 0) {
      real = r;
      imag = i;
   }
      Complex operator + (const Complex& obj) {
      Complex result;
      result.real = real + obj.real;
      result.imag = imag + obj.imag;
      return result;
   }
   void display() {
      cout << real << " + " << imag << "i\n";
   }
};
int main() {
   Complex c1(3, 4), c2(1, 2);
   Complex c3 = c1 + c2;
   cout << "Sum of complex numbers: ";
   c3.display();
   return 0;
}
```

7. Write a C++ program to demonstrate: • Single Inheritance • Multilevel Inheritance • Multiple Inheritance • Hierarchical Inheritance

```
#include <iostream>
using namespace std;
class InputNumbers {
public:
   int num1, num2;
   void getInput() {
      cout << "Enter the first number: ";
      cin >> num1;
      cout << "Enter the second number: ";
```

```
        cin >> num2;
    }
};
class SumNumbers : public InputNumbers {
public:
    void calculateSum() {
        int sum = num1 + num2;
        cout << "The sum of " << num1 << " and " << num2 << " is: " << sum
<< endl;
    }
};
int main() {
    SumNumbers obj;
    obj.getInput();
    obj.calculateSum();
    return 0;
}
```

```
#include <iostream>
using namespace std;
class Input {
public:
    int num1, num2;
    void getInput() {
        cout << "Enter two numbers: ";
        cin >> num1 >> num2;
    }
};
class Addition : public Input {
public:
    int add() {
        return num1 + num2;
    }
};
class Average : public Addition {
public:
    float calculateAverage() {
        return (num1 + num2) / 2.0;
    }
```

```
};
int main() {
    Average avg;
    avg.getInput();
     int sum = avg.add();
    cout << "Sum of the numbers: " << sum << endl;
       float average = avg.calculateAverage();
    cout << "Average of the numbers: " << average << endl;
        return 0;
}

#include <iostream>
using namespace std;
class Addition {
public:
    int num1, num2;
    void getInput() {
        cout << "Enter two numbers: ";
        cin >> num1 >> num2;
    }
    int add() {
        return num1 + num2;
    }
};
class Subtraction {
public:
    int num1, num2;
    void getInput() {
        cout << "Enter two numbers: ";
        cin >> num1 >> num2;
    }
    int subtract() {
        return num1 - num2;
    }
};
class Calculator : public Addition, public Subtraction {
public:
    void displayResults() {
        cout << "Addition of the numbers: " << add() << endl;
```

```
        cout << "Subtraction of the numbers: " << subtract() << endl;
    }
};
int main() {
    Calculator calc;
    calc.getInput();
    calc.displayResults();
    return 0;
}
```

```
#include <iostream>
using namespace std;
class Numbers {
protected:
    int num1, num2;
public:
     void inputNumbers() {
        cout << "Enter two numbers: ";
        cin >> num1 >> num2;
    }
};
class Addition : public Numbers {
public:
      void add() {
        cout << "Sum: " << num1 + num2 << endl;
    }
};
class Subtraction : public Numbers {
public:
    void subtract() {
        cout << "Difference: " << num1 - num2 << endl;
    }
};
int main() {
    Addition addObj;
    Subtraction subObj;
    addObj.inputNumbers();
    subObj.inputNumbers();
    addObj.add();
```

```
    subObj.subtract();
    return 0;
}
```

8. Write a C++ program to demonstrate Virtual Functions

```
#include <iostream>
using namespace std;
class Base {
public:
    void display() {
        cout << "\nDisplay base";
    }
    virtual void show() {
        cout << "\nShow base";
    }
};
class Derived : public Base {
public:
    void display() {
        cout << "\nDisplay derived";
    }
    void show() {
        cout << "\nShow derived";
    }
};
int main() {
    Base B;
    Derived D;
    Base* bptr;
    cout << "\n\nbptr points to Base\n";
    bptr = &B;
    bptr->display();
    bptr->show();
    cout << "\n\nbptr points to Derived\n";
    bptr = &D;
    bptr->display();
    bptr->show();
    return 0;
}
```

9.Write a C++ program to perform Sequential I/O Operations on a file

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ofstream outputFile("data.txt");
    if (!outputFile) {
        cout << "Error: Could not create the file.\n";
        return 1;
    }
    cout << "Writing to the file...\n";
    outputFile << "Name: Dhamu\n";
    outputFile << "Age: 25\n";
    outputFile << "Department: Computer Science\n";
    outputFile.close();
    cout << "Data written successfully.\n";
    ifstream inputFile("data.txt");
    if (!inputFile) {
        cout << "Error: Could not open the file.\n";
        return 1;
    }
    cout << "\nReading from the file:\n";
    string line;
    while (getline(inputFile, line)) {
        cout << line << endl;
    }
    inputFile.close();
    return 0;
}
```

10. Write a C++ program to find the Biggest Number using Command Line Arguments

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(int argc, char *argv[]) {
   if (argc < 2) {
      cout << "Error: Please provide numbers as command-line arguments.\n";
      return 1;
   }
   int biggest = atoi(argv[1]);
   for (int i = 2; i < argc; i++) {
      int current = atoi(argv[i]);
      if (current > biggest) {
         biggest = current;
      }
   }
   cout << "The biggest number is: " << biggest << endl;
   return 0;
}
```

REFERENCE:

1. Balagurusamy, E. (2017). *Object-Oriented Programming with C++* (8th ed.). Tata McGraw-Hill Education.