Session 2025-26

Subject – Object Oriented Programming using C++

Subject Code – 2022471(022)

Prepared by – Manish Dongre, Lecturer (CSE)

Unit 1 Fundamentals of OO Programming

1. Evolution of Object-Oriented Programming (OOP)

The journey to OOP was driven by the increasing complexity of software systems. Traditional programming paradigms struggled to manage the scale and maintainability of large applications.

- **Early Days: Machine Language and Assembly Language:**
  - These were the earliest forms of programming, directly manipulating hardware instructions.
  - Low-level, difficult to understand, and machine-dependent.
- **Procedural Programming (POP):**
  - Introduced with languages like FORTRAN, COBOL, and C.
  - Programs are structured as a sequence of procedures or functions.
  - Focus on algorithms and step-by-step execution.
  - Data and functions are treated separately.
- **The Rise of Modularity:**
  - As software grew, the need for modularity became apparent.
  - Languages like Pascal and Modula-2 introduced better support for modular programming.
  - This improved code organization and reusability to some extent.
- **The Object-Oriented Revolution:**
  - Simula (Simulation Language) was one of the earliest OOP languages, introducing concepts like classes and objects.
  - Smalltalk further popularized OOP with its emphasis on objects and message passing.
  - C++ emerged as a powerful language that extended C with OOP features.
  - Java and C# solidified OOP's dominance in software development.
- **Key OOP Concepts:**
  - **Encapsulation:** Bundling data and methods that operate on that data into a single unit (class).
  - **Abstraction:** Hiding complex implementation details and exposing only essential features.
  - **Inheritance:** Creating new classes (derived classes) from existing classes (base classes), inheriting their properties and behaviors.
  - **Polymorphism:** The ability of objects of different classes to respond to the same message in different ways.

1.1 Comparison of Procedure-Oriented Programming (POP) and Object-Oriented Programming (OOP)

Feature	Procedure-Oriented Programming (POP)	Object-Oriented Programming (OOP)
Focus	Procedures/functions (algorithms)	Objects (data and methods)
Data/Functions	Data and functions are separate entities.	Data and functions are encapsulated within objects.
Approach	Top-down approach (breaking down a problem into smaller procedures)	Bottom-up approach (creating objects and building interactions)
Data Security	Less secure; global data can be accessed from anywhere.	More secure; data is hidden through encapsulation.
Real-World Modeling	Less effective at modeling real-world entities.	More effective at modeling real-world entities as objects.
Code Reusability	Limited code reusability.	High code reusability through inheritance and polymorphism.
Maintainability	Difficult to maintain and modify large programs.	Easier to maintain and modify due to modularity and encapsulation.

Complexity Management	Struggles with managing complex systems.	Better suited for managing complex systems.
Data Movement	data moves freely from one function to another.	data is tied to the object that contains it.
Examples	C, Pascal, FORTRAN	C++, Java, Python, C#

C++ and OOP

C++ is a powerful language that supports both procedural and object-oriented programming paradigms. Its ability to combine the efficiency of C with the flexibility of OOP has made it a popular choice for various applications, including:

- System software
- Game development
- Embedded systems
- High-performance computing

Key OOP Features in C++:

- **Classes and Objects:** C++ allows you to define classes, which are blueprints for creating objects.
- **Encapsulation:** Achieved through access specifiers (public, private, protected).
- **Inheritance:** Implemented using the `: public`, `: private`, and `: protected` keywords.
- **Polymorphism:** Supported through virtual functions and function overloading.
- **Abstraction:** Achieved through abstract classes and interfaces.

1.2 Features of Object-Oriented Programming (OOP) in C++

C++ has several key features that make it a powerful language for object-oriented programming (OOP):

1.2.1 Object

- An object is a runtime entity. That means it exists when the program is running.
- It represents a real-world entity or concept.
- Objects are instances of classes.
- They contain both data (attributes) and functions (methods) that operate on that data.
- Example: In a "Car" class, a specific car like "MyCar," with its own attributes (color, model, etc.), is an object.

1.2.2 Class

- A class is a blueprint or template for creating objects.
- It defines the attributes (data members) and methods (member functions) that objects of that class will have.
- It's a user-defined data type.
- Example:

  Code

```
class Car {
public:
    string color;
    string model;
    void startEngine() {
```

```
        // ...
    }
};
```

1.2.3 Data Hiding and Encapsulation

- **Encapsulation:** Bundling data and the methods that operate on that data into a single unit (class).
- **Data Hiding:** Restricting direct access to some of an object's components. This is achieved through access specifiers (public, private, protected).
- Private members can only be accessed from within the class, providing data security.
- Example:

  Code

  ```
  class BankAccount {
  private:
      double balance;
  public:
      void deposit(double amount) {
          balance += amount;
      }
      double getBalance() {
          return balance;
      }
  };
  ```

  In this example, `balance` is private and is only accessed by the public methods `deposit` and `getBalance`.

1.2.4 Dynamic Binding (Late Binding)

- Dynamic binding refers to the runtime association of a function call with its corresponding function body.
- Polymorphism is implemented using dynamic binding.
- Virtual functions are used to achieve dynamic binding.
- The actual function to be called is determined at runtime, based on the object's type.
- Example:

  Code

  ```
  class Shape {
  public:
      virtual void draw() {
          cout << "Drawing a shape" << endl;
      }
  };
  class Circle : public Shape {
  public:
      void draw() override {
          cout << "Drawing a circle" << endl;
      }
  };
  ```

  At runtime, if a pointer to a `Shape` points to a `Circle`, the `Circle`'s `draw()` function will be called.

1.2.5 Message Passing

- Objects communicate with each other by sending and receiving messages.
- A message is a request for an object to execute one of its methods.
- In C++, message passing is achieved by calling member functions of an object.
- Example:

Code

```
Car myCar;
myCar.startEngine(); // Sending a message to the myCar object
```

1.2.6 Inheritance

- Inheritance allows a new class (derived class or subclass) to inherit properties and behaviors from an existing class (base class or superclass).
- It promotes code reusability.
- Types of inheritance in C++:
  - Single inheritance
  - Multiple inheritance
  - Hierarchical inheritance
  - Multilevel inheritance
  - Hybrid inheritance
- Example:

Code

```
class Vehicle {
public:
    string brand;
};
class Car : public Vehicle {
public:
    int numberOfDoors;
};
```

`Car` inherits the `brand` property from `Vehicle`.

1.2.7 Polymorphism

- Polymorphism means "many forms."
- It allows objects of different classes to respond to the same message in different ways.
- Types of polymorphism in C++:
  - Compile-time (static) polymorphism: Function overloading, operator overloading.
  - Runtime (dynamic) polymorphism: Virtual functions, function overriding.
- Example (Function Overloading):

Code

```
class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
};
```

The `add` function has two forms, accepting different data types.

1.3 Benefits of Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) offers numerous advantages that make it a powerful and widely used paradigm. In C++, these benefits are particularly pronounced due to the language's robust support for OOP principles.

1. Code Reusability:

- **Inheritance:**
  - OOP allows you to create new classes (derived classes) that inherit properties and behaviors from existing classes (base classes).
  - This reduces code duplication and promotes the reuse of proven code.
  - For example, if you have a `Vehicle` class, you can create `Car`, `Truck`, and `Motorcycle` classes that inherit from `Vehicle`, inheriting common attributes and methods.
- **Class Libraries:**
  - OOP encourages the creation of reusable class libraries, which can be shared and used across multiple projects.
  - This significantly accelerates development time and reduces the effort required to build complex applications.

2. Modularity and Maintainability:

- **Encapsulation:**
  - OOP encapsulates data and methods within objects, creating self-contained modules.
  - This modularity makes it easier to understand, debug, and maintain code.
  - Changes to one part of the code are less likely to affect other parts, reducing the risk of unintended side effects.
- **Abstraction:**
  - OOP allows you to hide complex implementation details and expose only essential interfaces.
  - This simplifies code and makes it easier to modify without affecting other parts of the system.
  - This also allows for easier future expansion of code, as more detail can be added to the abstracted code without affecting the calling code.

3. Data Security:

- **Data Hiding (Information Hiding):**
  - OOP provides access control mechanisms (public, private, protected) to restrict direct access to object data.
  - This protects data from accidental or unauthorized modification, enhancing data integrity and security.
  - By hiding data, the programmer controls how that data is changed.

4. Real-World Modeling:

- **Object-Oriented Design:**
  - OOP allows you to model real-world entities and their interactions as objects.
  - This results in code that is more intuitive and easier to understand, as it mirrors real-world concepts.
  - For example, a bank system can be modeled using objects like `Account`, `Customer`, and `Transaction`.

5. Polymorphism:

- **Flexibility and Extensibility:**
  - Polymorphism allows objects of different classes to respond to the same message in different ways.

    - This makes code more flexible and extensible, as new classes can be added without modifying existing code.
    - This is especially useful when dealing with collections of objects that share a common interface.
- **Dynamic Binding:**
    - Runtime polymorphism (dynamic binding) allows the appropriate method to be called at runtime, based on the object's type.
    - This enhances flexibility and allows for dynamic behavior.

6. Improved Software Development Productivity:

- **Reduced Development Time:**
    - Code reusability and modularity reduce the amount of code that needs to be written from scratch.
    - This accelerates development time and allows developers to focus on higher-level design and functionality.
- **Simplified Maintenance:**
    - Modular code is easier to understand and modify, reducing the effort required for maintenance and updates.
    - This also reduces the chance of introducing new bugs to old code.

7. Scalability:

- **Handling Complex Systems:**
    - OOP is well-suited for developing large and complex systems.
    - The modularity and abstraction provided by OOP make it easier to manage the complexity of such systems.
    - As systems grow, they can be more easily expanded.

8. Collaboration and Team Development:

- **Standardized Interfaces:**
    - OOP promotes the use of standardized interfaces, making it easier for multiple developers to work on the same project.
    - Objects act as well-defined units of code, facilitating collaboration and code integration.
- The clear definition of classes and objects, makes it easier for multiple people to understand and work on the same code base.

1.4 Applications of Object-Oriented Programming (OOP)

C++'s robust support for OOP principles makes it a versatile language suitable for a wide range of applications. Here's a breakdown of key areas where OOP with C++ shines:

1. System Software:

- **Operating Systems:**
    - C++ is used extensively in the development of operating system kernels and device drivers.
    - OOP's modularity and abstraction enable developers to manage the complexity of operating system components.
    - Examples: Parts of Windows, macOS, and Linux kernels.
- **Device Drivers:**
    - OOP facilitates the creation of hardware-specific drivers by encapsulating device functionality into objects.
    - This makes driver development more organized and maintainable.

- **Utilities:**
    - System utilities, such as file management tools and network utilities, are often developed using C++.

2. Graphical User Interfaces (GUIs):

- **Desktop Applications:**
    - C++ is widely used for creating desktop applications with rich GUIs.
    - OOP frameworks like Qt and wxWidgets provide classes and objects for building user interface elements.
    - OOP enables the creation of reusable GUI components and event-driven architectures.
- **Game Development:**
    - C++ is a dominant language in game development due to its performance and OOP capabilities.
    - Game engines like Unreal Engine utilize C++ to create complex game logic, graphics, and physics simulations.
    - OOP helps to create game entities as objects, and manage complex interactions.

3. Embedded Systems:

- **Microcontroller Programming:**
    - C++ is used in embedded systems where performance and memory efficiency are crucial.
    - OOP allows developers to create modular and reusable code for controlling hardware devices.
    - Automotive systems, and industrial control systems are examples.
- **Real-Time Systems:**
    - OOP helps to manage the timing constraints of real-time systems by encapsulating real time tasks as objects.

4. Database Management Systems (DBMS):

- **Database Engines:**
    - C++ is used to develop high-performance database engines and server-side components.
    - OOP helps to model database objects and relationships.
    - Examples: Parts of MySQL and other DBMS.

5. High-Performance Computing (HPC):

- **Scientific Simulations:**
    - C++ is used in scientific simulations where performance is critical.
    - OOP allows developers to create modular and reusable code for complex simulations.
    - Financial simulations, and physics simulations.
- **Parallel Computing:**
    - OOP libraries and frameworks support parallel computing, enabling developers to leverage multi-core processors and distributed systems.

6. Client-Server Applications:

- **Network Programming:**
    - C++ is used to create robust and efficient client-server applications.
    - OOP facilitates the creation of network protocols and communication layers.
    - Web servers, and network based games.
- **Distributed Systems:**
    - OOP helps to design and implement distributed systems, where objects interact across network boundaries.

7. Artificial Intelligence (AI) and Machine Learning (ML):

- **AI Libraries:**
  - C++ is used to develop high-performance AI libraries and frameworks.
  - TensorFlow and other ML libraries use C++ for their backend performance.
- **Robotics:**
  - OOP enables the creation of modular and reusable code for robot control and perception.

8. Financial Applications:

- **Trading Systems:**
  - C++ is used to develop high-frequency trading systems where low latency is critical.
  - OOP helps to model financial instruments and trading strategies.
- **Risk Management Systems:**
  - OOP makes it easier to model complex financial risk calculations.

Why C++ for these Applications?

- **Performance:** C++ offers excellent performance, crucial for applications that require speed and efficiency.
- **Control:** C++ provides low-level control over hardware and memory, essential for system software and embedded systems.
- **Flexibility:** C++ supports both procedural and object-oriented programming, allowing developers to choose the best approach for their needs.
- **Mature Ecosystem:** C++ has a rich ecosystem of libraries and frameworks, making it suitable for a wide range of applications.

In essence, C++'s combination of performance, control, and OOP capabilities makes it a powerful language for developing a diverse array of software application

1.5 Keywords in C++

C++ keywords are reserved words that have special meanings to the compiler. They cannot be used as identifiers (variable names, function names, etc.). Here's a detailed look at some keywords listed:

1. `friend`

- **Purpose:** The `friend` keyword grants a non-member function or another class access to the private and protected members of a class.
- **Usage:** It's used within a class definition to declare a friend function or class.
- **Example:**

  Code

```
#include <iostream>

class Box {
private:
    double width;
public:
    Box(double w) : width(w) {}
    friend void printWidth(Box box); // Friend function
};

void printWidth(Box box) {
```

```
    std::cout << "Width of box: " << box.width << std::endl; // Accessing private
member
}

int main() {
    Box myBox(10.0);
    printWidth(myBox);
    return 0;
}
```

2. `const`

- **Purpose:** The `const` keyword is used to declare constants, which are variables whose values cannot be modified after initialization.
- **Usage:** It can be used with variables, function parameters, and member functions.
- **Example:**

  Code

```
#include <iostream>

int main() {
    const int myConst = 5;
    // myConst = 10; // Error: cannot modify a const variable

    return 0;
}
```

3. `auto`

- **Purpose:** The `auto` keyword is used for type inference. The compiler deduces the type of a variable from its initializer.
- **Usage:** It's used when the type of a variable is obvious or when you want to avoid explicitly specifying the type.
- **Example:**

  Code

```
#include <iostream>

int main() {
    auto x = 10;      // x is deduced as int
    auto y = 3.14;    // y is deduced as double
    auto z = "hello"; // z is deduced as const char*

    return 0;
}
```

4. `register`

- **Purpose:** The `register` keyword was used to suggest to the compiler that a variable should be stored in a CPU register for faster access.
- **Usage:** It's rarely used in modern C++, as compilers are generally very good at optimizing register usage.
- **Note:** In modern C++ (C++17 and later), the `register` keyword is deprecated and treated as a normal `auto` variable.
- **Example (Older C++):**

  Code

```
int main() {
```

```
    register int count = 0; // Suggestion to store in register
    // ...
    return 0;
}
```

5. `extern`

- **Purpose:** The `extern` keyword is used to declare a variable or function that is defined in another source file.
- **Usage:** It's used to tell the compiler that the variable or function exists elsewhere.
- **Example:**

  Code

```
// file1.cpp
int globalVar = 10;

// file2.cpp
#include <iostream>

extern int globalVar; // Declaration, not definition

int main() {
    std::cout << "Global variable: " << globalVar << std::endl;
    return 0;
}
```

6. `typedef`

- **Purpose:** The `typedef` keyword is used to create aliases (synonyms) for existing data types.
- **Usage:** It's used to make code more readable and maintainable.
- **Example:**

  Code

```
typedef unsigned int uint;

int main() {
    uint myUnsignedInt = 10;
    // ...
    return 0;
}
```

7. `static`

- **Purpose:** The `static` keyword has different meanings depending on its context:
  - **Static variables inside a function:** They retain their values between function calls.
  - **Static member variables of a class:** They belong to the class itself, not to individual objects.
  - **Static member functions of a class:** They can be called without creating an object of the class.
  - **Static variables at file scope:** They are local to the file.
- **Example:**

  Code

```
#include <iostream>

void myFunction() {
    static int count = 0;
    count++;
    std::cout << "Count: " << count << std::endl;
}
```

```
int main() {
    myFunction(); // Output: Count: 1
    myFunction(); // Output: Count: 2
    return 0;
}
```

8. `default`

- **Purpose:** Within a `switch` statement, the `default` keyword specifies the code to execute if none of the `case` labels match the switch expression.
- **Purpose 2:** Also, can be used to explicitly generate default constructors, assignment operators, or destructors.
- **Usage:** Used in switch statements, or within class definitions.
- **Example:**

  Code

```
#include <iostream>

int main() {
    int choice = 3;
    switch (choice) {
        case 1: std::cout << "Choice 1" << std::endl; break;
        case 2: std::cout << "Choice 2" << std::endl; break;
        default: std::cout << "Default choice" << std::endl; break;
    }
    return 0;
}
```

9. `return`

- **Purpose:** The `return` keyword is used to terminate a function and return a value (or nothing, if the function's return type is `void`).
- **Usage:** It's used within function definitions.
- **Example:**

  Code

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

int main() {
    int sum = add(5, 3);
    // ...
    return 0;
}
```

10. `void`

- **Purpose:** The `void` keyword has several uses:
  - It's used as the return type of a function that does not return a value.
  - It's used to declare a generic pointer (e.g., `void*`), which can point to any data type.
  - It's used to specify that a function takes no arguments.
- **Example:**

  Code

```
#include <iostream>

void printMessage() {
```

```
    std::cout << "Hello!" << std::endl;
}

int main() {
    printMessage();
    return 0;
}
```

11. `continue`

- **Purpose:** The `continue` keyword is used within loops (`for`, `while`, `do-while`) to skip the rest of the current iteration and proceed to the next iteration.
- **Usage:** It's used to bypass certain code blocks within a loop based on a condition.
- **Example:**

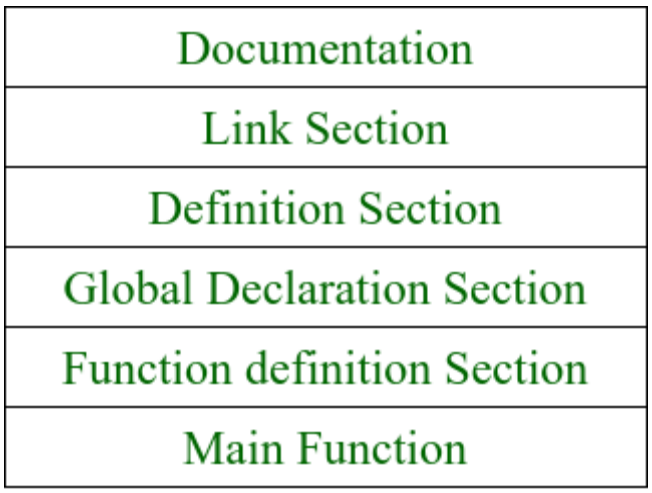
  Code

```
#include <iostream>

int main() {
    for (int i = 0; i < 5; i++) {
        if (i == 2) {
            continue; // Skip iteration when i is 2
        }
        std::cout << i << std::endl;
    }
    return 0;
}
```

1.6 Structure of C++ program

structure of the program written in C++ language is as follows:



Skeleton of C Program

Documentation Section:

- This section comes first and is used to document the logic of the program that the programmer going to code.
- It can be also used to write for purpose of the program.
- Whatever written in the documentation section is the comment and is not compiled by the compiler.

- Documentation Section is optional since the program can execute without them. Below is the snippet of the same:

```
/*     This is a C++ program to find the
        factorial of a number

    The basic requirement for writing this
    program is to have knowledge of loops

    To find the factorial of number
    iterate over range from number to one
*/
```

Linking Section:

The linking section contains two parts:

Header Files:

- Generally, a program includes various programming elements like built-in functions , classes, keywords, etc. that are already defined in the standard C++ library
- In order to use such pre-defined elements in a program, an appropriate header must be included in the program.
- Standard headers are specified in a program through the preprocessor directive #include. In Figure, the iostream header is used. When the compiler processes the instruction #include<iostream>, it includes the contents of the stream in the program. This enables the programmer to use standard input, output, and error facilities that are provided only through the standard streams defined in <iostream>. These standard streams process data as a stream of characters, that is, data is read and displayed in a continuous flow. The standard streams defined in <iostream> are listed here.

#include<iostream>

Namespaces:

- A namespace permits grouping of various entities like classes, objects and various C++ tokens , etc. under a single name.
- Any user can create separate namespaces of its own and can use them in any other program.
- In the below snippets, **namespace std** . contains declarations for cout ., cin, endl , etc. statements.

```
using namespace std;
```

- Namespaces can be accessed in multiple ways:
  - using namespace std;
  - using std :: cout;

Definition Section:

- It is used to declare some constants and assign them some value.
- In this section, anyone can define your own datatype using primitive data types.
- In **#define** is a compiler directive which tells the compiler whenever the message is found replace it with “Factorial\n” .
- **typedef int K;** this statement telling the compiler that whenever you will encounter K replace it by int and as you have declared k as datatype you cannot use it as an identifier.

Global Declaration Section:

- Here the variables and the class definitions which are going to be used in the program are declared to make them global.
- The scope of the variable declared in this section lasts until the entire program terminates.
- These variables are accessible within the user-defined functions.

Function Declaration Section:

- It contains all the functions which our main functions need.
- Usually, this section contains the User-defined functions.
- This part of the program can be written after the main function but for this, write the function prototype in this section for the function which for you are going to write code after the main function.

Code

```
// Function to implement the
// factorial of number num
int factorial(k& num)
{
    // Iterate over the loop from
    // num to one
    for (k i = 1; i <= num; i++) {
        fact *= i;
    }

    // Return the factorial calculated
    return fact;
}
```

Main Function:

- The main function tells the compiler where to start the execution of the program. The execution of the program starts with the main function.
- All the statements that are to be executed are written in the main function.
- The compiler executes all the instructions which are written in the curly braces **{}** which encloses the body of the main function.
- Once all instructions from the main function are executed. control comes out of the main function and the program terminates and no further execution occur.

Below is the program to illustrate this:

- Code

```
// Documentation Section
/*  This is a C++ program to find the
    factorial of a number
    The basic requirement for writing this
    program is to have knowledge of loops
    To find the factorial of a number
    iterate over the range from number to 1
*/

// Linking Section
#include <iostream>
using namespace std;

// Defination Section
#define msg "FACTORIAL\n"
typedef int k;
```

```
// Global Declaration Section
k num = 0, fact = 1, storeFactorial = 0;

// Function Section
k factorial(k& num)
{
    // Iterate over the loop from
    // num to one
    for (k i = 1; i <= num; i++) {
        fact *= i;
    }

    // Return the factorial
    return fact;
}

// Main Function
int main()
{
    // Given number Num
    k Num = 5;

    // Function Call
    storeFactorial = factorial(Num);
    cout << msg;

    // Print the factorial
    cout << Num << "! = "
         << storeFactorial << endl;

    return 0;
}
```

Output

```
FACTORIAL
5! = 120
```

1.7 Basic Data types and User Data types

1. Basic (Primitive) Data Types

C++ provides a set of fundamental data types that are built into the language. These are often called primitive data types.

- **Integer Types:**
  - These types represent whole numbers (numbers without a fractional part).
  - Key types:
    - `int`: The most common integer type. Size is typically 4 bytes (32 bits), but it can vary depending on the compiler and system.
    - `short int` (or `short`): Typically smaller than `int`, often 2 bytes (16 bits).
    - `long int` (or `long`): Typically larger than or equal to `int`, often 4 or 8 bytes.
    - `long long int` (or `long long`): Guaranteed to be at least 8 bytes (64 bits).
    - `unsigned int`, `unsigned short int`, `unsigned long int`, `unsigned long long int`: These versions store only non-negative values, effectively doubling the positive range.
  - Example:

Code

```
#include <iostream>
```

```
int main() {
    int myInt = 10;
    short myShort = -5;
    long myLong = 1000000L; // 'L' suffix indicates a long literal
    unsigned int myUnsigned = 20;

    std::cout << "int: " << myInt << std::endl;
    std::cout << "short: " << myShort << std::endl;
    std::cout << "long: " << myLong << std::endl;
    std::cout << "unsigned int: " << myUnsigned << std::endl;

    return 0;
}
```

- **Floating-Point Types:**
  - These types represent numbers with fractional parts.
  - Key types:
    - `float`: Single-precision floating-point, typically 4 bytes.
    - `double`: Double-precision floating-point, typically 8 bytes (offers greater precision than `float`).
    - `long double`: Extended-precision floating-point, often 10 or 12 bytes, or 16 bytes.
  - Example:

Code

```
#include <iostream>

int main() {
    float myFloat = 3.14f; // 'f' suffix indicates a float literal
    double myDouble = 3.14159265359;
    long double myLongDouble = 2.718281828459045L; // 'L' suffix indicates a long
double literal

    std::cout << "float: " << myFloat << std::endl;
    std::cout << "double: " << myDouble << std::endl;
    std::cout << "long double: " << myLongDouble << std::endl;

    return 0;
}
```

- **Character Type:**
  - `char`: Represents a single character, typically 1 byte.
  - Example:

Code

```
#include <iostream>

int main() {
    char myChar = 'A';
    std::cout << "char: " << myChar << std::endl;
    return 0;
}
```

- **Boolean Type:**
  - `bool`: Represents a truth value, either `true` or `false`.
  - Example:

Code

```
#include <iostream>
```

```
int main() {
    bool myBool = true;
    std::cout << "bool: " << myBool << std::endl;
    return 0;
}
```

- **Void Type:**
  - `void`: Represents the absence of a type. It's used in several contexts, such as:
    - Function return types when a function doesn't return a value.
    - Pointers to memory locations of unknown types.

2. User-Defined Data Types

C++ allows you to create your own data types, extending the basic types.

- **Structures (`struct`):**
  - A structure is a collection of variables (members) of different data types grouped under a single name.
  - Example:

Code

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

struct Person {
    std::string name;
    int age;
    double height;
};

int main() {
    Person person1;
    person1.name = "Alice";
    person1.age = 30;
    person1.height = 1.65;

    std::cout << "Name: " << person1.name << std::endl;
    std::cout << "Age: " << person1.age << std::endl;
    std::cout << "Height: " << person1.height << std::endl;

    return 0;
}
```

- **Classes (`class`):**
  - Classes are similar to structures, but they also allow you to define functions (methods) within them and have access control (public, private, protected).
  - Classes are the core of object oriented programming.
  - Example:

Code

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

class Car {
public:
    std::string brand;
    std::string model;
    int year;

    void displayInfo() {
```

```
        std::cout << "Brand: " << brand << ", Model: " << model << ", Year: " <<
year << std::endl;
    }
};

int main() {
    Car car1;
    car1.brand = "Toyota";
    car1.model = "Camry";
    car1.year = 2022;

    car1.displayInfo();

    return 0;
}
```

- **Enumerations (`enum`):**
  - Enumerations allow you to define a set of named integer constants.
  - Example:

Code

```
#include <iostream>

enum Day {
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
};

int main() {
    Day today = WEDNESDAY;
    std::cout << "Today is day: " << today << std::endl; // Output will be the
integer value of WEDNESDAY (2)

    if (today == WEDNESDAY){
        std::cout << "Wednesday is a mid-week day." << std::endl;
    }
    return 0;
}
```

- **Unions (`union`):**
  - Unions are similar to structures, but all members share the same memory location. The size of a union is the size of its largest member.
  - Unions are used to save memory when only one of the members needs to be used at a time.
  - Example:

Code

```
#include <iostream>

union Data {
    int i;
    double d;
    char c;
};

int main(){
    Data data;
    data.i = 10;
    std::cout << data.i << std::endl;
```

```
    data.d = 12.5;
    std::cout << data.d << std::endl;
    data.c = 'a';
    std::cout << data.c << std::endl;
    return 0;
}
```

- **Typedef and Type Aliases (using):**
  - `typedef` and `using` (C++11 and later) allow you to create aliases for existing data types.
  - Example:

Code

```
typedef unsigned int uint; // Using typedef
using myDouble = double; // Using type alias (C++11)

int main() {
    uint myUnsignedInt = 100;
    myDouble myDoubleValue = 3.14159;

    std::cout << "uint: " << myUnsignedInt << std::endl;
    std::cout << "myDouble: " << myDoubleValue << std::endl;

    return 0;
}
```

These basic and user-defined data types are fundamental to C++ programming, allowing you to represent and manipulate various forms of data.

1.8 Various operators in C++

1. Arithmetic Operators

These operators are used to perform mathematical calculations.

- **Addition (+):**
  - Adds two operands.
  - Example: `int sum = 5 + 3;` (sum will be 8)
- **Subtraction (-):**
  - Subtracts the second operand from the first.
  - Example: `int difference = 10 - 4;` (difference will be 6)
- **Multiplication (*):**
  - Multiplies two operands.
  - Example: `int product = 6 * 7;` (product will be 42)
- **Division (/):**
  - Divides the first operand by the second.
  - **Important:** Integer division truncates the decimal part.
  - Example: `int quotient = 15 / 4;` (quotient will be 3)
  - Example: `double quotientDouble = 15.0 / 4.0;` (quotientDouble will be 3.75)
- **Modulo (%):**
  - Returns the remainder of a division.
  - Can only be used with integer operands.
  - Example: `int remainder = 17 % 5;` (remainder will be 2)
- **Increment (++):**
  - Increases the operand by 1.
  - **Prefix (++x)**: Increments before using the value.
  - **Postfix (x++)**: Increments after using the value.

- Example:

  Code

```
int x = 5;
int y = ++x; // x is now 6, y is 6
int z = x++; // z is 6, x is now 7.
```

- **Decrement (`--`):**
  - Decreases the operand by 1.
  - **Prefix (`--x`)**: Decrements before using the value.
  - **Postfix (`x--`)**: Decrements after using the value.
  - Example:

    Code

```
int x = 5;
int y = --x; // x is now 4, y is 4
int z = x--; // z is 4, x is now 3.
```

2. Relational Operators

These operators are used to compare two operands and return a boolean value (`true` or `false`).

- **Equal to (`==`):**
  - Checks if two operands are equal.
  - Example: `bool isEqual = (5 == 5);` (isEqual will be `true`)
- **Not equal to (`!=`):**
  - Checks if two operands are not equal.
  - Example: `bool isNotEqual = (5 != 6);` (isNotEqual will be `true`)
- **Greater than (`>`):**
  - Checks if the first operand is greater than the second.
  - Example: `bool isGreater = (10 > 7);` (isGreater will be `true`)
- **Less than (`<`):**
  - Checks if the first operand is less than the second.
  - Example: `bool isLess = (3 < 8);` (isLess will be `true`)
- **Greater than or equal to (`>=`):**
  - Checks if the first operand is greater than or equal to the second.
  - Example: `bool isGreaterOrEqual = (10 >= 10);` (isGreaterOrEqual will be `true`)
- **Less than or equal to (`<=`):**
  - Checks if the first operand is less than or equal to the second.
  - Example: `bool isLessOrEqual = (4 <= 5);` (isLessOrEqual will be `true`)

3. Logical Operators

These operators are used to combine or negate boolean expressions.

- **Logical AND (`&&`):**
  - Returns `true` if both operands are `true`.
  - Example: `bool result = (true && true);` (result will be `true`)
  - Example: `bool result2 = (true && false);` (result2 will be `false`)
- **Logical OR (`||`):**
  - Returns `true` if at least one operand is `true`.
  - Example: `bool result = (true || false);` (result will be `true`)
  - Example: `bool result2 = (false || false);` (result2 will be `false`)
- **Logical NOT (`!`):**
  - Negates the operand.

- Example: `bool result = !true;` (result will be `false`)
- Example: `bool result2 = !false;` (result2 will be `true`)

Example Combining Operators

Code

```
#include <iostream>

int main() {
    int x = 10;
    int y = 5;

    bool condition1 = (x > y) && (x % 2 == 0); // true && true
    bool condition2 = (x < y) || (y == 5); // false || true

    std::cout << "Condition 1: " << condition1 << std::endl; // Output: 1 (true)
    std::cout << "Condition 2: " << condition2 << std::endl; // Output: 1 (true)

    return 0;
}
```

These operators form the building blocks for many C++ programs, allowing for calculations, comparisons, and complex logical evaluations.

4. Scope Resolution Operator (`::`)

- **Purpose:**
  - Used to access members of a namespace or class when they're hidden by another variable or function with the same name in the current scope.
  - Used to access static members of a class.
  - Used to access global variables when a local variable with the same name exists.
- **Examples:**
  - Namespace access:

    Code

    ```
    #include <iostream>

    namespace MyNamespace {
        int value = 10;
    }

    int main() {
        std::cout << MyNamespace::value << std::endl; // Accessing value from
    MyNamespace
        return 0;
    }
    ```

  - Class static member access:

    Code

    ```
    #include <iostream>

    class MyClass {
    public:
        static int staticValue;
    };

    int MyClass::staticValue = 20; // Initializing static member

    int main() {
        std::cout << MyClass::staticValue << std::endl;
    ```

```
    return 0;
}
```

- Global scope access:

Code

```
#include <iostream>

int globalVar = 30;

int main() {
    int globalVar = 40; // Local variable hides global one
    std::cout << ::globalVar << std::endl; // Accessing the global variable
    return 0;
}
```

5. Member Dereferencing Operators (`.`, `->`, `.*`, `->*`)

- **`.` (Dot operator):**
  - Used to access members of a class or struct object directly.
  - Example:

Code

```
struct Point {
    int x, y;
};

int main() {
    Point p;
    p.x = 5;
    p.y = 10;
    std::cout << p.x << std::endl;
    return 0;
}
```

- **`->` (Arrow operator):**
  - Used to access members of a class or struct object through a pointer.
  - Example:

Code

```
struct Point {
    int x, y;
};

int main() {
    Point* p = new Point;
    p->x = 5;
    p->y = 10;
    std::cout << p->y << std::endl;
    delete p;
    return 0;
}
```

- **`.*` and `->*` (Pointer-to-member operators):**
  - Used to access members through pointers to members. These are more advanced and used less frequently.
  - `.*` is used with objects, and `->*` is used with object pointers.

6. Memory Management Operators (`new`, `delete`, `new[]`, `delete[]`)

- **`new`:**
  - Allocates dynamic memory for a single object.
  - Returns a pointer to the allocated memory.
  - Example:

  Code

  ```
  int* ptr = new int;
  *ptr = 10;
  ```

- **`delete`:**
  - Deallocates memory that was allocated with `new`.
  - Example:

  Code

  ```
  delete ptr;
  ```

- **`new[]`:**
  - Allocates dynamic memory for an array of objects.
  - Example:

  Code

  ```
  int* arr = new int[5];
  arr[0] = 1;
  ```

- **`delete[]`:**
  - Deallocates memory that was allocated with `new[]`.
  - Example:

  Code

  ```
  delete[] arr;
  ```

7. Type Cast Operators

- **Purpose:**
  - Converts a value from one data type to another.
- **Types:**
  - **`static_cast`:**
    - For conversions that are valid at compile time.
    - Example: `static_cast<double>(intVar)`
  - **`dynamic_cast`:**
    - For safe downcasting in inheritance hierarchies.
    - Performs runtime type checking.
  - **`reinterpret_cast`:**
    - For low-level type conversions, potentially unsafe.
    - Example: `reinterpret_cast<int*>(charPtr)`
  - **`const_cast`:**
    - For adding or removing `const` or `volatile` qualifiers.
    - Example: `const_cast<int*>(constIntPtr)`
  - Implicit Casting:
    - C++ will sometimes implicitly cast datatypes. For example, int to double.
    - Example:

Code

```
int a = 5;
double b = a;
```

8. Operator Precedence

- **Purpose:**
  - Determines the order in which operators are evaluated in an expression.
- **Key points:**
  - Operators with higher precedence are evaluated first.
  - Parentheses `()` can be used to override precedence.
  - Common precedence levels from highest to lowest include:
    - Parentheses `()`
    - Unary operators (`++`, `--`, `!`, `-`)
    - Multiplicative operators (`*`, `/`, `%`)
    - Additive operators (`+`, `-`)
    - Relational operators (`<`, `>`, `<=`, `>=`)
    - Equality operators (`==`, `!=`)
    - Logical AND (`&&`)
    - Logical OR (`||`)
    - Assignment operators [1] (`=`, `+=`, `-=`, etc.)
- Example:
  - `int result = 5 + 3 * 2;` (result is 11, because `*` has higher precedence than `+`)
  - `int result2 = (5 + 3) * 2;` (result2 is 16, because parentheses override precedence)

1.9 C++ Manipulators:

Introduction

- Manipulators are functions specifically designed to modify the state of input/output streams.
- They provide a convenient way to control formatting aspects like:
  - Width of output fields.
  - Precision of floating-point numbers.
  - Base of number representation (decimal, hexadecimal, octal).
  - Alignment of output.
- Manipulators are used with the insertion (`<<`) and extraction (`>>`) operators.
- To use most manipulators, you need to include the `<iomanip>` header.

Key Manipulators

- **`<iostream>` Manipulators:**
  - **`endl`:**
    - Inserts a newline character (`\n`) and flushes the output stream.
    - Example: `std::cout << "Hello" << std::endl;`
  - **`flush`:**
    - Flushes the output stream, forcing any buffered output to be written immediately.
    - Example: `std::cout << "This will be written now" << std::flush;`
  - **`ws`:**
    - Extracts and discards leading whitespace characters from an input stream.
    - Example: `std::cin >> std::ws >> myString;`
- **`<iomanip>` Manipulators:**
  - **`setw(int width)`:**
    - Sets the width of the next output field.

        - The output is right-aligned by default.
        - Example: `std::cout << std::setw(10) << 123;`
    - **`setprecision(int precision)`:**
        - Sets the precision (number of digits) for floating-point output.
        - Example: `std::cout << std::setprecision(3) << 3.14159;`
    - **`fixed`:**
        - Displays floating-point numbers in fixed-point notation (e.g., 3.14).
        - Example: `std::cout << std::fixed << 3.14159;`
    - **`scientific`:**
        - Displays floating-point numbers in scientific notation (e.g., 3.14e+00).
        - Example: `std::cout << std::scientific << 1234.56;`
    - **`setfill(char fill)`:**
        - Sets the fill character for padding output fields.
        - Example: `std::cout << std::setw(10) << std::setfill('*') << 123;`
    - **`left`:**
        - Left-aligns output within the field width.
        - Example: `std::cout << std::left << std::setw(10) << 123;`
    - **`right`:**
        - Right-aligns output within the field width (default).
        - Example: `std::cout << std::right << std::setw(10) << 123;`
    - **`setbase(int base)`:**
        - Sets the base for integer output (8 for octal, 10 for decimal, 16 for hexadecimal).
        - Example: `std::cout << std::setbase(16) << 255;`
    - **`hex`, `dec`, `oct`:**
        - These are used to set the base of integer output.
        - Example: `std::cout << std::hex << 255;`
    - **`boolalpha`:**
        - Displays boolean values as "true" or "false" instead of 1 or 0.
        - Example: `std::cout << std::boolalpha << true;`
    - **`noboolalpha`:**
        - Displays boolean values as 1 or 0.
        - Example: `std::cout << std::noboolalpha << true;`
    - **`showpos`:**
        - Displays a plus sign (+) for positive numeric values.
        - Example: `std::cout << std::showpos << 10;`
    - **`noshowpos`:**
        - Does not display a plus sign (+) for positive numeric values (default).
        - Example: `std::cout << std::noshowpos << 10;`
    - **`uppercase`:**
        - displays hexadecimal values in uppercase letters.
        - Example: `std::cout << std::uppercase << std::hex << 255;`
    - **`nouppercase`:**
        - displays hexadecimal values in lowercase letters (default).
        - Example: `std::cout << std::nouppercase << std::hex << 255;`
    - **`showpoint`:**
        - Forces the decimal point to be displayed for floating-point numbers, even if the fractional part is zero.
        - Example: `std::cout << std::showpoint << 10.0;`
    - **`noshowpoint`:**
        - Does not display the decimal point for floating-point numbers if the fractional part is zero (default).
        - Example: `std::cout << std::noshowpoint << 10.0;`

3. Example Usage

Code

```
#include <iostream>
#include <iomanip>

int main() {
    int num = 123;
    double pi = 3.14159;

    std::cout << "Formatted output:" << std::endl;
    std::cout << std::setw(10) << std::setfill('*') << num << std::endl;
    std::cout << std::fixed << std::setprecision(2) << pi << std::endl;
    std::cout << std::hex << num << std::endl;
    std::cout << std::boolalpha << true << std::endl;

    return 0;
}
```

Unit 2: Functions and Classes in C++

2.1 Functions in C++

2.1.1 Introduction to Functions, Function Prototyping, Call by Reference

Introduction to Functions

- **Definition:** A function is a block of organized, reusable code that performs a specific task. Functions break down large programs into smaller, manageable parts, making them easier to understand, modify, and debug.
- **Benefits:**
  - **Modularity:** Dividing a program into functions promotes modularity.
  - **Reusability:** Functions can be called multiple times, reducing code duplication.
  - **Abstraction:** Functions hide implementation details, providing a clear interface.
  - **Readability:** Functions improve code readability and maintainability.
- **Structure:** A function generally consists of:
  - **Return type:** Specifies the data type of the value returned by the function. If the function doesn't return a value, the return type is `void`.
  - **Function name:** A unique identifier for the function.
  - **Parameters (optional):** Input values passed to the function.
  - **Function body:** The code that performs the function's task.

Function Prototyping

- **Definition:** A function prototype declares a function before its definition. It informs the compiler about the function's return type, name, and parameters.
- **Syntax:**

```
return_type function_name(parameter_list);
```

  - `return_type`: The data type of the value returned by the function.
  - `function_name`: The name of the function.
  - `parameter_list`: A comma-separated list of parameter types (and optionally, parameter names).
- **Purpose:**
  - Allows the compiler to check if function calls are valid (i.e., if the correct number and types of arguments are passed).
  - Enables functions to be defined after they are called.
- **Example:**

Code

```
int add(int a, int b); // Function prototype

int main() {
    int result = add(5, 3); // Function call
    // ...
}

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

Call by Reference

- **Definition:** Call by reference passes the memory address of a variable to a function. Any changes made to the parameter within the function affect the original variable.
- **Syntax:** Use the ampersand (`&`) symbol in the function parameter list to indicate a reference parameter.

```
void modify(int &x);
```

- **Mechanism:**
  - Instead of creating a copy of the variable, the function works directly with the original variable's memory location.
  - This allows functions to modify the values of variables passed as arguments.
- **Example:**

  Code

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

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10, y = 20;
    cout << "Before swap: x = " << x << ", y = " << y << endl;
    swap(x, y);
    cout << "After swap: x = " << x << ", y = " << y << endl;
    return 0;
}
```

  - In this example, the `swap` function modifies the original values of `x` and `y`.

2.1.2 Nesting Member Functions

- **Definition:** Nesting member functions refers to the ability of a member function within a class to call other member functions of the same class.
- **Purpose:**
  - Promotes code reuse within a class.
  - Allows complex operations to be broken down into smaller, more manageable member functions.
  - Improves code organization and readability.
- **Mechanism:**
  - Within a member function, other member functions can be called directly by their names.
  - The `this` pointer is implicitly used to access other member functions and data members of the same object.
- **Example:**

  Code

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

class Calculator {
private:
    int num1, num2;

public:
    void setNumbers(int a, int b) {
```

```
        num1 = a;
        num2 = b;
    }

    int add() {
        return num1 + num2;
    }

    void displaySum() {
        int sum = add(); // Nesting member function 'add()'
        cout << "Sum: " << sum << endl;
    }
};

int main() {
    Calculator calc;
    calc.setNumbers(5, 10);
    calc.displaySum();
    return 0;
}
```

- In this example, the `displaySum()` member function calls the `add()` member function.

2.1.3 Recursive Functions and Function Overloading

Recursive Functions

- **Definition:** A recursive function is a function that calls itself, either directly or indirectly, within its own definition.
- **Mechanism:**
  - A recursive function breaks down a problem into smaller, self-similar subproblems.
  - It continues to call itself until it reaches a base case, which is a condition that stops the recursion.
  - Once the base case is reached, the function returns, and the results of the subproblems are combined to produce the final result.
- **Key Components:**
  - **Base Case:** The condition that terminates the recursion. Without a base case, the function would call itself indefinitely, leading to a stack overflow.
  - **Recursive Call:** The call to the function itself, typically with modified arguments that move closer to the base case.
- **Example: Factorial Calculation**

  Code

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

int factorial(int n) {
    if (n == 0) { // Base case
        return 1;
    } else {
        return n * factorial(n - 1); // Recursive call
    }
}

int main() {
    int num = 5;
    cout << "Factorial of " << num << " is " << factorial(num) << endl;
    return 0;
}
```

  - In this example, the `factorial()` function calculates the factorial of a number by calling itself with decreasing values of `n` until `n` reaches 0.

- **Advantages:**
  - Recursive solutions can be elegant and concise for certain problems, particularly those involving self-similar structures (e.g., tree traversals, fractal generation).
- **Disadvantages:**
  - Recursive functions can be less efficient than iterative solutions due to the overhead of function calls.
  - Deep recursion can lead to stack overflow if the base case is not reached or if the recursion depth is too large.

Function Overloading

- **Definition:** Function overloading allows you to define multiple functions with the same name but different parameters within the same scope.
- **Mechanism:**
  - The compiler distinguishes between overloaded functions based on the number, types, and order of their parameters.
  - When a function is called, the compiler selects the appropriate overloaded function based on the arguments passed.
- **Rules:**
  - Overloaded functions must have different parameter lists.
  - The return type alone cannot be used to overload functions.
- **Example: Overloading the `add()` Function**

  Code

```
#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 (int, int): " << add(5, 10) << endl;
    cout << "Sum (double, double): " << add(3.5, 2.7) << endl;
    cout << "Sum (int, int, int): " << add(1, 2, 3) << endl;
    return 0;
}
```

  - In this example, the `add()` function is overloaded to handle different data types and parameter counts.
- **Advantages:**
  - Function overloading enhances code readability and flexibility by allowing you to use the same function name for operations that are conceptually similar but operate on different data types or parameter lists.
  - Reduces the amount of function names that you have to remember.
- **Disadvantages:**
  - If overloading is overused, it can lead to ambigious code, where the compiler can not figure out which function should be called.
  - It can make the code harder to read if the overloaded functions are not close together in the code.

2.2 Classes and Objects in C++

2.2.1 Introduction to Classes, Types of Classes, and Friend Classes

Introduction to Classes

- **Definition:** A class is a blueprint or template for creating objects. It defines the data members (attributes) and member functions (methods) that objects of that class will possess.
- **Encapsulation:** Classes encapsulate data and functions into a single unit, providing data hiding and abstraction.
- **Syntax:**

```
class ClassName {
private: // Access specifier
    // Data members (attributes)
public: // Access specifier
    // Member functions (methods)
protected: // Access specifier
    // protected members
}; // Don't forget the semicolon!
```

- **Objects:** An object is an instance of a class. It occupies memory and has its own set of data members.
- **Creating Objects:**

Code

```
ClassName objectName; // Static allocation
ClassName* objectPtr = new ClassName; // Dynamic allocation
```

- **Accessing Members:**
  - Use the dot (`.`) operator to access members of an object.
  - Use the arrow (`->`) operator to access members of an object pointer.

Code

```
objectName.memberFunction();
objectPtr->dataMember = value;
```

Types of Classes

1. **Base Class (Parent Class or Superclass):**
   - A class that serves as the foundation for other classes.
   - Derived classes inherit properties and behaviors from the base class.
   - It is used for code reusability via inheritance.
   - Example: A "Vehicle" class can be a base class for "Car" and "Truck" classes.
2. **Derived Class (Child Class or Subclass):**
   - A class that inherits from a base class.
   - It can add new data members and member functions or override inherited ones.
   - Inheritance establishes an "is-a" relationship (e.g., a Car "is-a" Vehicle).
   - **Syntax:**

```
class DerivedClassName : accessSpecifier BaseClassName {
    // ...
};
```

     - `accessSpecifier` can be `public`, `private`, or `protected`, controlling the accessibility of inherited members.

3. **Virtual Class:**
    - Used to solve the "diamond problem" in multiple inheritance.
    - Ensures that a base class is inherited only once when multiple inheritance paths exist.
    - Syntax: virtual keyword is used when inheriting.

      Code

      ```
      class Base { /* ... */ };
      class Derived1 : virtual public Base { /* ... */ };
      class Derived2 : virtual public Base { /* ... */ };
      class FinalDerived : public Derived1, public Derived2 { /* ... */ };
      ```

    - Using virtual inheritance avoids duplicate base class subobjects.
4. **Abstract Class:**
    - A class that cannot be instantiated (i.e., you cannot create objects of it).
    - It serves as a base class for derived classes.
    - Typically contains one or more pure virtual functions.
    - Pure virtual function : a virtual function that is declared in the base class, but is not defined. The derived class must define it.
    - **Syntax:**

      ```
      class AbstractClassName {
      public:
          virtual void pureVirtualFunction() = 0; // Pure virtual function
      };
      ```

    - Abstract classes define interfaces that derived classes must implement.
5. **Friend Class:**
    - A class that is granted access to the private and protected members of another class.
    - It breaks encapsulation to allow specific classes to work closely together.
    - **Syntax:** The `friend` keyword is used within the class granting access.

      ```
      class ClassA {
          friend class ClassB;
      private:
          int privateData;
      };

      class ClassB {
      public:
          void accessPrivateData(ClassA& obj) {
              obj.privateData = 10; // Accessing private member
          }
      };
      ```

    - Friend classes should be used sparingly as they can weaken encapsulation.

2.2.2 Specifying Classes, Defining Data Members, and Member Functions in C++

1. Specifying a Class

- **Declaration:**
    - A class declaration introduces a new data type to the compiler. It specifies the name of the class and its members (data members and member functions).
    - The `class` keyword is used to declare a class.

    - The class declaration typically resides in a header file (`.h` or `.hpp`) to allow for separate compilation.
- **Syntax:**

```
class ClassName {
// Access specifiers: private, public, protected
private:
    // Data members (attributes)
public:
    // Member functions (methods)
protected:
    // Protected members
}; // Don't forget the semicolon!
```

- **Access Specifiers:**
    - `private`: Members are accessible only within the class itself.
    - `public`: Members are accessible from anywhere outside the class.
    - `protected`: Members are accessible within the class and its derived classes.
- **Example:**

Code

```
// In a header file (e.g., "Rectangle.h")
class Rectangle {
private:
    double length;
    double width;
public:
    void setDimensions(double l, double w);
    double calculateArea();
};
```

2. Defining Data Members

- **Data Members (Attributes):**
    - Data members are variables that represent the state of an object.
    - They are declared within the class declaration.
    - They can be of any valid C++ data type (e.g., `int`, `double`, `char`, `string`, or user-defined types).
    - Access specifiers determine their accessibility.
- **Initialization:**
    - Data members can be initialized within the class declaration using initializer lists or within constructor functions.
    - Example:

Code

```
class Example {
private:
    int value = 10; //In class initialization(c++11 and above)
    const double pi;
public:
    Example(double p): pi(p){}
};
```

- **Example (Continuing from Rectangle class):**

Code

```
class Rectangle {
private:
```

```
    double length; // Data member for length
    double width;  // Data member for width
public:
    // ...
};
```

3. Defining Member Functions

- **Member Functions (Methods):**
  - Member functions are functions that operate on the data members of an object.
  - They define the behavior of the class.
  - They can be declared within the class declaration and defined either inside or outside the class.
- **Defining Inside the Class:**
  - When a member function is defined inside the class declaration, it is implicitly an inline function.
- **Defining Outside the Class:**
  - When a member function is defined outside the class declaration, the scope resolution operator (`::`) is used to specify the class to which the function belongs.
  - Example:

    Code

    ```
    //in Rectangle.cpp file.
    #include "Rectangle.h"
    #include <iostream>

    void Rectangle::setDimensions(double l, double w) {
        length = l;
        width = w;
    }

    double Rectangle::calculateArea() {
        return length * width;
    }
    ```

- **`this` Pointer:**
  - Within a member function, the `this` pointer refers to the object for which the function is called.
  - It is implicitly passed as the first argument to non-static member functions.
  - It is used to access the object's data members and member functions.
- **Constant Member Functions:**
  - Member functions can be declared as `const` to indicate that they do not modify the object's data members.
  - The `const` keyword is placed after the function's parameter list.
  - Example:

    Code

    ```
    double Rectangle::calculateArea() const {
        return length * width;
    }
    ```

- **Example (Continuing from Rectangle class):**

  Code

  ```
  // In "Rectangle.cpp"
  #include "Rectangle.h"

  void Rectangle::setDimensions(double l, double w) {
      length = l;
      width = w;
  ```

```
}

double Rectangle::calculateArea() {
    return length * width;
}
```

2.3 Object Creation and Memory Allocation for Objects in C++

1. Object Creation

- **Definition:** Object creation (also known as instantiation) is the process of creating an instance of a class. An object is a concrete realization of the class's blueprint.
- **Syntax:**
  - **Static Allocation (Stack Allocation):**

    ```
    ClassName objectName; // Creates an object named 'objectName'
    ```

  - **Dynamic Allocation (Heap Allocation):**

    ```
    ClassName* objectPtr = new ClassName; // Creates an object on the heap
    ```

- **Static Allocation:**
  - Objects are created on the stack.
  - Memory is automatically allocated when the object is declared and automatically deallocated when the object goes out of scope.
  - Fast allocation and deallocation.
  - Limited to the stack's size.
  - Objects have a lifetime tied to the scope in which they are created.
- **Dynamic Allocation:**
  - Objects are created on the heap (free store).
  - Memory is explicitly allocated using the `new` operator and deallocated using the `delete` operator.
  - Provides more flexibility in terms of object lifetime.
  - Slower allocation and deallocation compared to the stack.
  - Requires manual memory management to prevent memory leaks.
  - Objects have a lifetime that is controlled by the user.
- **Constructors:**
  - Constructors are special member functions that are automatically called when an object is created.
  - They are used to initialize the object's data members.
  - If you don't define a constructor, the compiler provides a default constructor (which does nothing).
  - Constructors can be overloaded to provide multiple ways to initialize an object.
  - Example:

    Code

    ```
    class MyClass {
    public:
        int value;
        MyClass(int val) { // Constructor
            value = val;
        }
    };

    int main() {
        MyClass obj1(10); // Static allocation, constructor called
        MyClass* obj2 = new MyClass(20); // Dynamic allocation, constructor
    called.

        delete obj2;
        return 0;
    }
    ```

2. Memory Allocation for Objects

- **Memory Layout:**
  - When an object is created, the compiler allocates memory for its data members.
  - The size of the object is determined by the sum of the sizes of its data members.
  - Member functions are not stored within each object; instead, a single copy of each member function is shared among all objects of the class.
  - The `this` pointer is implicitly passed to member functions, allowing them to access the specific object's data.
- **Static Allocation Details:**
  - Memory is allocated on the stack frame of the function where the object is created.
  - The stack grows and shrinks as functions are called and return.
  - Automatic memory management prevents memory leaks.
- **Dynamic Allocation Details:**
  - The `new` operator allocates memory from the heap.
  - The `new` operator returns a pointer to the allocated memory.
  - The `delete` operator deallocates the memory pointed to by the pointer.
  - Failure to use `delete` leads to memory leaks, where memory is allocated but never released.
  - Example of memory leak prevention:

    Code

    ```
    MyClass* obj = new MyClass(25);
    // ... use obj ...
    delete obj; // Deallocate memory
    obj = nullptr; // good practice.
    ```

- **Array of Objects:**
  - Objects can be created in arrays using both static and dynamic allocation.
  - Static array of objects.

    Code

    ```
    MyClass myArray[5]; // Static array of 5 MyClass objects.
    ```

  - Dynamic array of objects.

    Code

    ```
    MyClass* myDynamicArray = new MyClass[10]; // Dynamic array of 10 MyClass
    objects.
    delete[] myDynamicArray; // Deallocate array memory
    myDynamicArray = nullptr;
    ```

- **Memory Fragmentation:**
  - Repeated dynamic allocation and deallocation can lead to memory fragmentation, where free memory becomes scattered and difficult to allocate in contiguous blocks.
- **Smart Pointers:**
  - To automate dynamic memory management and prevent memory leaks, C++ provides smart pointers (e.g., `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`).
  - Smart pointers automatically deallocate memory when the pointer goes out of scope or is no longer needed.

2.4 Arrays as Class Members and Arrays of Objects in C++

1. Arrays as Class Members

- **Definition:** An array can be used as a data member within a class to store a collection of elements of the same data type.
- **Declaration:**
  - The array is declared within the class's private or protected section, depending on the required access level.
  - The array's size must be known at compile time for statically allocated arrays.
- **Syntax:**

```
class MyClass {
private:
    int myArray[10]; // Array of 10 integers as a data member
    double data[5];  // Array of 5 doubles.
public:
    // Member functions to manipulate the array
    void setArrayValue(int index, int value);
    int getArrayValue(int index) const;
    // ...
};
```

- **Initialization:**
  - Arrays within classes can be initialized using member initialization lists in constructors.
  - It is crucial to handle array boundaries to prevent buffer overflows.
  - Example:

    Code

```
#include <iostream>
class ArrayHolder {
private:
    int numbers[3];
public:
    ArrayHolder(int a, int b, int c) : numbers{a,b,c} {}
    void printArray() const {
        for(int i = 0; i < 3; i++){
            std::cout << numbers[i] << " ";
        }
        std::cout << std::endl;
    }
};

int main(){
    ArrayHolder arr(1,2,3);
    arr.printArray();
    return 0;
}
```

- **Accessing Elements:**
  - Array elements are accessed using the index operator (`[]`).
  - Member functions are typically provided to encapsulate array access and enforce boundary checks.
- **Dynamic Allocation:**
  - For arrays whose size is not known at compile time, dynamic allocation using the `new` operator is required.
  - Remember to use `delete[]` to deallocate the dynamically allocated array.
  - Example:

Code

```
class DynamicArray {
private:
    int* array;
    int size;
public:
    DynamicArray(int size) : size(size) {
        array = new int[size];
    }
    ~DynamicArray() {
        delete[] array;
    }
    int& operator[](int index){ //overloaded operator to access array.
        return array[index];
    }
};

int main(){
    DynamicArray arr(5);
    arr[0] = 10;
    std::cout << arr[0] << std::endl;
    return 0;
}
```

2. Arrays of Objects

- **Definition:** An array of objects is an array where each element is an object of a specific class.
- **Declaration:**
  - Arrays of objects can be declared using both static and dynamic allocation.
- **Static Allocation:**
  - The array is created on the stack.
  - The compiler must know the size of the array at compile time.
  - Example:

Code

```
class Point {
public:
    int x, y;
};

int main() {
    Point points[3]; // Array of 3 Point objects
    points[0].x = 10;
    points[0].y = 20;
    // ...
    return 0;
}
```

- **Dynamic Allocation:**
  - The array is created on the heap.
  - The `new` operator is used to allocate memory for the array.
  - The `delete[]` operator is used to deallocate the memory.
  - Example:

Code

```
class Point {
public:
    int x, y;
};
```

```
int main() {
    Point* points = new Point[5]; // Array of 5 Point objects on the heap
    points[0].x = 5;
    points[0].y = 6;
    delete[] points;
    return 0;
}
```

- **Initialization:**
  - When creating an array of objects, the default constructor of the class is called for each element in the array.
  - If the class has constructors with parameters, you can use initializer lists for static arrays or call the parameterized constructors in a loop for dynamic arrays.
  - Example of initialization of static array:

    Code

```
class Point {
public:
    int x, y;
    Point(int xVal = 0, int yVal = 0) : x(xVal), y(yVal) {}
};
int main(){
    Point pointArray[2] = {Point(1,2), Point(3,4)};
    return 0;
}
```

- **Accessing Members:**
  - Object members are accessed using the index operator (`[]`) followed by the dot (`.`) or arrow (`->`) operator, depending on whether the array is static or dynamic.
  - Example:

    Code

```
points[0].x = 10; // Static array
points[1]->y = 20; // Dynamic array
```

2.5 Passing Objects as Function Arguments in C++

Passing objects as function arguments is a fundamental concept in C++, allowing functions to work with and manipulate objects. There are three primary ways to pass objects: by value, by reference, and by pointer.

1. Passing Objects by Value

- **Mechanism:**
  - When an object is passed by value, a copy of the object is created and passed to the function.
  - Any changes made to the object within the function do not affect the original object in the calling scope.
  - The copy of the object is destroyed when the function returns.
- **Syntax:**

```
void myFunction(ClassName objectName); // Object passed by value
```

- **Example:**

  Code

```
#include <iostream>
```

```
using namespace std;

class Point {
public:
    int x, y;
    Point(int xVal = 0, int yVal = 0) : x(xVal), y(yVal) {}
    void print() const {
        cout << "x: " << x << ", y: " << y << endl;
    }
};

void modifyPoint(Point p) { // Passing by value
    p.x = 100;
    p.y = 200;
    cout << "Inside modifyPoint: ";
    p.print();
}

int main() {
    Point myPoint(1, 2);
    cout << "Before modifyPoint: ";
    myPoint.print();

    modifyPoint(myPoint);

    cout << "After modifyPoint: ";
    myPoint.print(); // Original object remains unchanged
    return 0;
}
```

- **Considerations:**
  - Passing by value can be expensive for large objects due to the overhead of copying.
  - It is suitable when the function needs to work with a copy and avoid modifying the original object.

2. Passing Objects by Reference

- **Mechanism:**
  - When an object is passed by reference, the function receives a reference (alias) to the original object.
  - Any changes made to the object within the function directly affect the original object in the calling scope.
  - No copy of the object is created.
- **Syntax:**

```
void myFunction(ClassName& objectName); // Object passed by reference
```

- **Example:**

  Code

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

class Point {
public:
    int x, y;
    Point(int xVal = 0, int yVal = 0) : x(xVal), y(yVal) {}
    void print() const {
        cout << "x: " << x << ", y: " << y << endl;
    }

};
```

```
void modifyPoint(Point& p) { // Passing by reference
    p.x = 100;
    p.y = 200;
    cout << "Inside modifyPoint: ";
    p.print();
}

int main() {
    Point myPoint(1, 2);
    cout << "Before modifyPoint: ";
    myPoint.print();

    modifyPoint(myPoint);

    cout << "After modifyPoint: ";
    myPoint.print(); // Original object is modified
    return 0;
}
```

- **Considerations:**
  - Passing by reference is more efficient than passing by value, especially for large objects.
  - It is suitable when the function needs to modify the original object.
  - To prevent the function from modifying the original object, you can pass a constant reference.
    `void myFunction(const ClassName& objectName);`

3. Passing Objects by Pointer

- **Mechanism:**
  - When an object is passed by pointer, the function receives the memory address of the original object.
  - Any changes made to the object through the pointer directly affect the original object in the calling scope.
  - No copy of the object is created.
- **Syntax:**

```
void myFunction(ClassName* objectPtr); // Object passed by pointer
```

- **Example:**

  Code

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

class Point {
public:
    int x, y;
    Point(int xVal = 0, int yVal = 0) : x(xVal), y(yVal) {}
    void print() const {
        cout << "x: " << x << ", y: " << y << endl;
    }
};

void modifyPoint(Point* p) { // Passing by pointer
    p->x = 100;
    p->y = 200;
    cout << "Inside modifyPoint: ";
    p->print();
}

int main() {
    Point myPoint(1, 2);
    cout << "Before modifyPoint: ";
    myPoint.print();
```

```
    modifyPoint(&myPoint); // Pass the address of the object

    cout << "After modifyPoint: ";
    myPoint.print(); // Original object is modified
    return 0;
}
```

- **Considerations:**
  - Passing by pointer is also efficient, similar to passing by reference.
  - It is suitable when the function needs to modify the original object or when dealing with dynamically allocated objects.
  - Pointers can be null, so it's essential to check for null pointers before accessing the object.

2.6 Static Data Members and Member Functions in C++

1. Static Data Members

- **Definition:**
  - A static data member is a class member that belongs to the class itself, rather than to any specific object of the class.
  - There is only one copy of a static data member, shared by all objects of the class.
  - Static data members exist even if no objects of the class have been created.
- **Declaration:**
  - Static data members are declared within the class using the `static` keyword.
  - They are typically declared in the class's private or protected section, depending on the required access level.
- **Initialization:**
  - Static data members must be defined (initialized) outside the class declaration, usually in the source file (`.cpp`).
  - The scope resolution operator (`::`) is used to specify the class to which the static data member belongs.
  - Example:

    Code

    ```
    class MyClass {
    public:
        static int count; // Declaration
    };

    int MyClass::count = 0; // Definition and initialization
    ```

- **Accessing Static Data Members:**
  - Static data members can be accessed using the class name and the scope resolution operator (`::`).
  - They can also be accessed through objects of the class, but this is generally discouraged for clarity.
  - Example:

    Code

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

    class MyClass {
    public:
        static int count;
        MyClass(){count++;}
    ```

```
};

int MyClass::count = 0;

int main() {
    MyClass obj1, obj2, obj3;
    cout << "Count: " << MyClass::count << endl; // Access using class name
    return 0;
}
```

- **Use Cases:**
  - Counting the number of objects created.
  - Storing class-wide constants or configurations.
  - Implementing shared resources.

2. Static Member Functions

- **Definition:**
  - A static member function is a class member function that belongs to the class itself, rather than to any specific object of the class.
  - Static member functions can only access static data members and other static member functions.
  - They do not have access to the `this` pointer because they are not associated with any particular object.
- **Declaration:**
  - Static member functions are declared within the class using the `static` keyword.
- **Definition:**
  - Static member functions can be defined inside or outside of the class. If defined outside, the scope resolution operator `::` must be used.
- **Accessing Static Member Functions:**
  - Static member functions are called using the class name and the scope resolution operator (`::`).
  - They can also be called through objects of the class, but this is generally discouraged for clarity.
  - Example:

    Code

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

class MyClass {
private:
    static int count;

public:
    MyClass() { count++; }
    static int getCount() {
        return count;
    }
};

int MyClass::count = 0;

int main() {
    MyClass obj1, obj2, obj3;
    cout << "Count: " << MyClass::getCount() << endl; // Access using class
name
    return 0;
}
```

- **Use Cases:**
  - Providing utility functions that operate on static data.
  - Creating factory functions that return objects of the class.
  - Implementing class-level operations.

Key Differences:

- **Static Data Members:**
  - Belong to the class, not objects.
  - Shared by all objects.
  - Must be defined outside the class.
- **Static Member Functions:**
  - Belong to the class, not objects.
  - Can only access static members.
  - Do not have access to the `this` pointer.
  - Accessed via the class name using the scope resolution operator.

2.7 Access Specifiers in C++ and Their Scope

Access specifiers in C++ are keywords that control the accessibility of class members (data members and member functions). They define the scope within which these members can be accessed. C++ provides three access specifiers: `public`, `private`, and `protected`.

1. Public Access Specifier

- **Keyword:** `public`
- **Scope:**
  - Members declared as `public` are accessible from anywhere in the program.
  - This includes access from within the class itself, from objects of the class, and from functions outside the class.
- **Use Cases:**
  - Public members define the interface of the class, allowing external code to interact with the class's objects.
  - They are used for members that need to be accessed and modified by other parts of the program.
- **Example:**

  Code

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

class MyClass {
public:
    int publicData;
    void publicFunction() {
        cout << "Public function called." << endl;
    }
};

int main() {
    MyClass obj;
    obj.publicData = 10; // Accessing public data member
    obj.publicFunction(); // Calling public member function
    return 0;
}
```

2. Private Access Specifier

- **Keyword:** `private`
- **Scope:**
  - Members declared as `private` are accessible only within the class itself.

    - They cannot be accessed from outside the class, including from objects of the class or from functions outside the class.
    - If no access specifier is used, private is the default.
- **Use Cases:**
    - Private members are used to implement data hiding and encapsulation.
    - They are used for members that represent the internal state or implementation details of the class.
    - They prevent direct access to sensitive data, ensuring data integrity.
- **Example:**

  Code

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

class MyClass {
private:
    int privateData;
    void privateFunction() {
        cout << "Private function called." << endl;
    }

public:
    void accessPrivate() {
        privateData = 20; // Accessing private data within the class
        privateFunction(); // Calling private function within the class
    }
};

int main() {
    MyClass obj;
    obj.accessPrivate();
    // obj.privateData = 30; // Error: privateData is inaccessible
    // obj.privateFunction(); // Error: privateFunction is inaccessible
    return 0;
}
```

3. Protected Access Specifier

- **Keyword:** `protected`
- **Scope:**
    - Members declared as `protected` are accessible within the class itself and within its derived classes (classes that inherit from the base class).
    - They are not accessible from outside the class or from objects of the class, unless that object is of a derived class.
- **Use Cases:**
    - Protected members are used to provide access to members that should be shared among the base class and its derived classes, but not exposed to the general public.
    - They facilitate inheritance and code reuse.
- **Example:**

  Code

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

class BaseClass {
protected:
    int protectedData;
};

class DerivedClass : public BaseClass {
```

```
public:
    void accessProtected() {
        protectedData = 40; // Accessing protected data in derived class
        cout << "Protected data: " << protectedData << endl;
    }
};

int main() {
    DerivedClass obj;
    obj.accessProtected();
    // BaseClass baseObj;
    // baseObj.protectedData = 50; // Error: protectedData is inaccessible
    return 0;
}
```

Summary of Access Specifiers and Their Scope:

Access Specifier	Scope	Use Cases
`Public`	Accessible from anywhere.	Class interface, members that need to be accessed from outside.
`Private`	Accessible only within the class.	Data hiding, encapsulation, internal implementation details.
`Protected`	Accessible within the class and derived classes.	Inheritance, members that should be shared among the base class and derived classes, but not public.

2.8 C++ Streams and C++ Stream Classes

C++ streams provide a powerful and flexible way to perform input/output (I/O) operations. They abstract the underlying I/O devices, allowing you to work with files, console input/output, and other sources in a consistent manner.

1. C++ Streams

- **Definition:**
  - A stream is an abstraction that represents a sequence of bytes flowing from a source (input stream) or to a destination (output stream).
  - C++ streams work with various data sources, including files, console input/output, and strings.
- **Key Concepts:**
  - **Input Stream:** Used to read data from a source.
  - **Output Stream:** Used to write data to a destination.
  - **Buffering:** Streams often use buffers to improve I/O efficiency. Data is temporarily stored in the buffer before being transferred to or from the I/O device.
  - **Formatting:** Streams provide mechanisms to format data (e.g., setting precision, width, and base).
- **Standard Streams:**
  - **`std::cin`:** Standard input stream (usually connected to the keyboard).
  - **`std::cout`:** Standard output stream (usually connected to the console).
  - **`std::cerr`:** Standard error stream (usually connected to the console, used for error messages).
  - **`std::clog`:** Standard log stream (buffered version of `cerr`).

2. C++ Stream Classes

The C++ Standard Library provides a hierarchy of stream classes in the `<iostream>` and `<fstream>` headers. These classes provide the functionality for I/O operations.

- **Base Classes:**
  - **`std::ios_base`:** The base class for all stream classes. It defines formatting flags, error states, and other common stream properties.
  - **`std::ios`:** Derived from `ios_base`, it adds buffering and formatting capabilities.
  - **`std::istream`:** Base class for input streams. It provides functions for reading data (e.g., `operator>>`, `getline`).
  - **`std::ostream`:** Base class for output streams. It provides functions for writing data (e.g., `operator<<`, `put`).
  - **`std::iostream`:** Derived from `istream` and `ostream`, it supports both input and output operations.
- **File Stream Classes:**
  - **`std::ifstream`:** Input file stream. Used for reading data from files.
  - **`std::ofstream`:** Output file stream. Used for writing data to files.
  - **`std::fstream`:** File stream. Used for both reading and writing data to files.
- **String Stream Classes:**
  - **`std::istringstream`:** Input string stream. Used for reading data from strings.
  - **`std::ostringstream`:** Output string stream. Used for writing data to strings.
  - **`std::stringstream`:** String stream. Used for both reading and writing data to strings.

3. Common Stream Operations

- **Input Operations:**
  - **`operator>>` (Extraction Operator):** Reads formatted data from an input stream.
  - **`getline()`:** Reads a line of text from an input stream.
  - **`get()`:** Reads a single character from an input stream.
  - **`read()`:** Reads a block of raw bytes from an input stream.
- **Output Operations:**
  - **`operator<<` (Insertion Operator):** Writes formatted data to an output stream.
  - **`put()`:** Writes a single character to an output stream.
  - **`write()`:** Writes a block of raw bytes to an output stream.
- **File Operations:**
  - **`open()`:** Opens a file for I/O operations.
  - **`close()`:** Closes a file.
  - **`is_open()`:** Checks if a file is open.
- **Stream State:**
  - **`good()`:** Checks if the stream is in a good state (no errors).
  - **`eof()`:** Checks if the end of the file has been reached.
  - **`fail()`:** Checks if a non-fatal error occurred.
  - **`bad()`:** Checks if a fatal error occurred.
  - **`clear()`:** Clears the error state of the stream.
- **Formatting:**
  - **`setprecision()`:** Sets the precision for floating-point numbers.
  - **`setw()`:** Sets the width of the output field.
  - **`setfill()`:** Sets the fill character for padding.
  - **`setbase()`:** Sets the base for integer output (e.g., decimal, hexadecimal).

Example (Console I/O):

Code

```
#include <iostream>
#include <iomanip> // For formatting

int main() {
    int num;
    double pi = 3.14159;

    std::cout << "Enter a number: ";
    std::cin >> num;

    std::cout << "Number: " << num << std::endl;
    std::cout << std::fixed << std::setprecision(2) << "Pi: " << pi << std::endl;

    return 0;
}
```

Example (File I/O):

Code

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

int main() {
    std::ofstream outFile("output.txt");
    if (outFile.is_open()) {
        outFile << "Hello, file I/O!" << std::endl;
        outFile.close();
    }

    std::ifstream inFile("output.txt");
    if (inFile.is_open()) {
        std::string line;
        while (std::getline(inFile, line)) {
            std::cout << line << std::endl;
        }
        inFile.close();
    }

    return 0;
}
```

2.9 Introduction to Class Templates in C++

Class templates are a powerful feature in C++ that allow you to define generic classes. These templates enable you to create classes that can work with different data types without having to rewrite the class definition for each type. This promotes code reuse and flexibility.

1. What are Class Templates?

- **Definition:**
  - A class template is a blueprint for creating generic classes.
  - It allows you to define a class that can work with various data types without specifying the exact type at the time of class definition.
  - The actual data type is specified when an object of the template class is created.
- **Purpose:**
  - Code reusability: Avoids writing the same class definition for different data types.
  - Flexibility: Allows classes to work with a wide range of data types.
  - Generic programming: Enables the creation of generic algorithms and data structures.

2. Syntax of Class Templates

- **Template Declaration:**
  - The `template` keyword is used to declare a class template.
  - The template parameters are specified within angle brackets (`<...>`).
  - The `typename` or `class` keyword is used to specify template parameters.
- **Syntax:**

```
template <typename T>
class ClassName {
    // Class members using the template parameter T
};
```

  - `template <typename T>`: Declares that this is a template, and `T` is a template parameter.
  - `T`: Represents a placeholder for a data type. You can use any valid identifier as the template parameter name.

3. Creating Objects of Class Templates

- **Instantiation:**
  - To create an object of a class template, you must specify the actual data type for the template parameter.
  - This process is called instantiation.
- **Syntax:**

```
ClassName<DataType> objectName;
```

  - `ClassName<DataType>`: Specifies the class template name and the actual data type.
  - `objectName`: The name of the object.
- **Example:**

  Code

```
template <typename T>
class MyPair {
public:
    T first, second;
    MyPair(T a, T b) : first(a), second(b) {}
    T getMax() {
        return (first > second) ? first : second;
    }
};

int main() {
    MyPair<int> intPair(10, 20);
    std::cout << "Max (int): " << intPair.getMax() << std::endl;

    MyPair<double> doublePair(3.14, 2.71);
    std::cout << "Max (double): " << doublePair.getMax() << std::endl;

    return 0;
}
```

4. Member Functions of Class Templates

- **Defining Member Functions:**
  - Member functions of class templates can be defined inside or outside the class definition.
  - If defined outside, you must use the template syntax and the scope resolution operator (`::`).
- **Syntax (Outside Class):**

```
template <typename T>
ReturnType ClassName<T>::functionName(parameters) {
    // Function body
}
```

- **Example:**

  Code

```
template <typename T>
class MyArray {
private:
    T* array;
    int size;
public:
    MyArray(T arr[], int s);
    T getElement(int index);
};

template <typename T>
MyArray<T>::MyArray(T arr[], int s) {
    size = s;
    array = new T[size];
    for (int i = 0; i < size; i++) {
        array[i] = arr[i];
    }
}

template <typename T>
T MyArray<T>::getElement(int index) {
    if (index >= 0 && index < size) {
        return array[index];
    }
    return T(); // Default value
}

int main() {
    int intArray[] = {1, 2, 3, 4, 5};
    MyArray<int> arr(intArray, 5);
    std::cout << "Element at index 2: " << arr.getElement(2) << std::endl;
    return 0;
}
```

5. Non-Type Template Parameters

- **Definition:**
  - Class templates can also have non-type template parameters, such as integers or characters.
  - These parameters must be constant expressions.
- **Syntax:**

```
template <typename T, int size>
class MyFixedArray {
    T array[size];
public:
    // ...
};
```

- **Example:**

  Code

```
template <typename T, int size>
class MyFixedArray {
private:
    T array[size];
```

```
public:
    T& getElement(int index) {
        return array[index];
    }
};

int main() {
    MyFixedArray<int, 10> arr;
    arr.getElement(5) = 100;
    std::cout << "Element at index 5: " << arr.getElement(5) << std::endl;
    return 0;
}
```

Unit 3.0: Constructors and Destructors

3.1 Constructors

Constructors are special member functions of a class that are automatically called when an object of that class is created. Their primary purpose is to initialize the object's data members.

3.1.1 Concept of Initialization using Constructor

- **Automatic Invocation:** Unlike regular member functions that need to be explicitly called, constructors are invoked automatically by the compiler whenever a new object of the class is instantiated. This ensures that objects are properly initialized upon creation.
- **Same Name as Class:** A constructor has the same name as the class it belongs to.
- **No Return Type:** Constructors do not have a return type, not even `void`. Their job is to initialize the object, not to return a value.
- **Multiple Constructors (Constructor Overloading):** A class can have multiple constructors with different parameter lists. This allows objects to be initialized in various ways depending on the arguments provided during object creation. The compiler determines which constructor to call based on the number and types of arguments passed.
- **Default Constructor:** If a class does not explicitly define any constructors, the compiler automatically provides a default constructor. This default constructor performs no explicit initialization of the object's data members (they will have indeterminate values for primitive types and will call the default constructors of member objects). However, if you define any constructor in your class (even with parameters), the compiler will *not* generate a default constructor. If you need a constructor that takes no arguments in such a case, you must define it explicitly.
- **Purpose of Constructors:**
    - **Initialize Data Members:** The most common use is to set initial values to the member variables of the object.
    - **Resource Allocation:** Constructors can allocate resources needed by the object, such as dynamic memory using `new`, opening files, or establishing database connections.
    - **Perform Setup Operations:** Any setup or configuration required for the object to be in a valid state can be done within the constructor.

Example demonstrating a simple constructor:

C++

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

class Rectangle {
private:
    int length;
    int width;

public:
    // Constructor
    Rectangle(int len, int wid) {
        std::cout << "Rectangle constructor called." << std::endl;
        length = len;
        width = wid;
    }

    int area() const {
        return length * width;
    }

    void display() const {
        std::cout << "Length: " << length << ", Width: " << width << std::endl;
    }
};
```

```
int main() {
    Rectangle rect1(10, 5); // Constructor is automatically called here
    rect1.display();
    std::cout << "Area: " << rect1.area() << std::endl;

    return 0;
}
```

Example demonstrating constructor overloading:

C++

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

class Point {
private:
    int x;
    int y;

public:
    // Default Constructor
    Point() {
        std::cout << "Default Point constructor called." << std::endl;
        x = 0;
        y = 0;
    }

    // Parameterized Constructor 1
    Point(int val) {
        std::cout << "Point constructor with one parameter called." << std::endl;
        x = val;
        y = val;
    }

    // Parameterized Constructor 2
    Point(int xVal, int yVal) {
        std::cout << "Point constructor with two parameters called." << std::endl;
        x = xVal;
        y = yVal;
    }

    void display() const {
        std::cout << "X: " << x << ", Y: " << y << std::endl;
    }
};

int main() {
    Point p1;       // Calls the default constructor
    Point p2(5);    // Calls the constructor with one parameter
    Point p3(10, 20); // Calls the constructor with two parameters

    p1.display();
    p2.display();
    p3.display();

    return 0;
}
```

Key Takeaways about Constructors:

- Ensure proper initialization of objects.
- Have the same name as the class.
- Have no return type.
- Can be overloaded to provide flexible initialization options.
- The compiler provides a default constructor if no constructors are explicitly defined (but this behavior changes if you define any constructor).

- Can perform resource allocation and other setup tasks during object creation.

3.1.2 Multiple Constructors in a Class

As briefly introduced earlier, C++ allows a single class to have multiple constructors. This feature, known as **constructor overloading**, provides flexibility in how objects of that class can be initialized.

Why Use Multiple Constructors?

- **Different Initialization Needs:** Objects of a class might need to be initialized with varying sets of data. For example, a `Date` object could be created with just a year, or with a year, month, and day. Multiple constructors allow you to handle these different initialization scenarios.
- **Providing Default Values:** You can have a constructor with fewer parameters that uses default values for the missing arguments. This offers a convenient way to create objects with common default settings.
- **Initialization from Different Data Types:** A class might need to be initialized from different types of input. For instance, a `ComplexNumber` class could be initialized from two `double` values (real and imaginary parts) or from a single `std::string` representing the complex number in a specific format.

Rules for Constructor Overloading:

- **Same Name:** All constructors in a class must have the same name as the class itself.
- **Different Parameter Lists:** The compiler distinguishes between overloaded constructors based on their parameter lists. The parameter lists must differ in at least one of the following:
    - **Number of parameters:** Constructors with a different number of arguments are considered distinct.
    - **Types of parameters:** If the number of parameters is the same, their types must differ in at least one position.
    - **Order of parameters:** If the number and types of parameters are the same, their order must differ.

Example Demonstrating Multiple Constructors:

C++
```
#include <iostream>
#include <string>

class Employee {
private:
    int id;
    std::string name;
    double salary;

public:
    // Constructor 1: No parameters (default initialization)
    Employee() {
        std::cout << "Employee default constructor called." << std::endl;
        id = 0;
        name = "N/A";
        salary = 0.0;
    }

    // Constructor 2: With ID only
    Employee(int empId) {
        std::cout << "Employee constructor with ID called." << std::endl;
        id = empId;
        name = "Unknown";
        salary = 0.0;
    }
```

```
    // Constructor 3: With ID and name
    Employee(int empId, const std::string& empName) {
        std::cout << "Employee constructor with ID and name called." << std::endl;
        id = empId;
        name = empName;
        salary = 0.0;
    }

    // Constructor 4: With ID, name, and salary
    Employee(int empId, const std::string& empName, double empSalary) {
        std::cout << "Employee constructor with ID, name, and salary called." <<
std::endl;
        id = empId;
        name = empName;
        salary = empSalary;
    }

    void display() const {
        std::cout << "ID: " << id << ", Name: " << name << ", Salary: " << salary <<
std::endl;
    }
};

int main() {
    Employee emp1;           // Calls Constructor 1
    Employee emp2(101);      // Calls Constructor 2
    Employee emp3(102, "Alice"); // Calls Constructor 3
    Employee emp4(103, "Bob", 50000.0); // Calls Constructor 4

    emp1.display();
    emp2.display();
    emp3.display();
    emp4.display();

    return 0;
}
```

Constructor Selection by the Compiler:

When you create an object, the compiler examines the arguments you provide and matches them with the parameter list of the available constructors. The constructor whose parameter list best matches the arguments is the one that gets called. If no constructor matches the provided arguments, or if there is an ambiguous match (more than one constructor could be called), the compiler will generate an error.

Using Default Arguments in Constructors:

Instead of providing multiple constructors with similar functionality, you can often use default arguments to achieve a similar effect with fewer constructors.

Example using default arguments:

C++

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

class Rectangle {
private:
    int length;
    int width;

public:
    Rectangle(int len = 1, int wid = 1) { // Constructor with default arguments
        std::cout << "Rectangle constructor called (with default arguments)." <<
std::endl;
        length = len;
```

```
        width = wid;
    }

    int area() const {
        return length * width;
    }

    void display() const {
        std::cout << "Length: " << length << ", Width: " << width << std::endl;
    }
};

int main() {
    Rectangle rect1;      // Uses default length=1, width=1
    Rectangle rect2(5);   // Uses length=5, default width=1
    Rectangle rect3(10, 20); // Uses length=10, width=20

    rect1.display();
    rect2.display();
    rect3.display();

    return 0;
}
```

Considerations when using Multiple Constructors:

- **Code Clarity:** While multiple constructors offer flexibility, having too many can sometimes make the class interface confusing. Aim for a reasonable number of constructors that clearly represent the common ways to initialize objects.
- **Code Duplication:** If multiple constructors have similar initialization logic, consider using constructor delegation (available in C++11 and later) to reduce code duplication.
- **Default Arguments vs. Overloading:** Decide whether default arguments or separate overloaded constructors are more appropriate based on the specific initialization scenarios and code readability.

3.1.3 Types of Constructors

C++ provides several types of constructors to handle different object initialization scenarios. Let's explore each of them in detail:

1. Default Constructor

- **Definition:** A default constructor is a constructor that takes no arguments.
- **Invocation:** It is called when an object is declared without any explicit initialization arguments.
- **Compiler-Generated:** If you don't define any constructors in your class, the compiler automatically provides a default constructor. This compiler-generated default constructor performs member-wise default initialization:
    - For built-in types (like `int`, `float`, `bool`), members are left uninitialized (they contain garbage values).
    - For class objects, the default constructor of the member object is called (if it exists).
- **User-Defined:** You can also explicitly define a default constructor in your class. This is often done to provide specific default initialization values for your members.
- **Importance:** If you define any constructor (even a parameterized one), the compiler will *not* generate a default constructor. If you need to be able to create objects without any arguments in such a case, you must explicitly define a default constructor.

Example of Default Constructor:

C++

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

class MyClass {
```

```
private:
    int value;
    std::string message;

public:
    // User-defined default constructor
    MyClass() {
        std::cout << "Default constructor called." << std::endl;
        value = 0;
        message = "Default message";
    }

    void display() const {
        std::cout << "Value: " << value << ", Message: " << message << std::endl;
    }
};

int main() {
    MyClass obj1; // Calls the default constructor
    obj1.display();
    return 0;
}
```

2. Parameterized Constructor

- **Definition:** A parameterized constructor is a constructor that accepts one or more arguments.
- **Invocation:** It is called when an object is declared with initial values provided as arguments.
- **Purpose:** Parameterized constructors allow you to initialize the object's data members with specific values at the time of object creation.

Example of Parameterized Constructor:

C++

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

class Car {
private:
    std::string brand;
    std::string model;
    int year;

public:
    // Parameterized constructor
    Car(const std::string& b, const std::string& m, int y) {
        std::cout << "Parameterized Car constructor called." << std::endl;
        brand = b;
        model = m;
        year = y;
    }

    void display() const {
        std::cout << "Brand: " << brand << ", Model: " << model << ", Year: " << year <<
std::endl;
    }
};

int main() {
    Car myCar("Toyota", "Camry", 2023); // Calls the parameterized constructor
    myCar.display();
    return 0;
}
```

3. Copy Constructor

- **Definition:** A copy constructor is a special constructor that creates a new object as a copy of an existing object of the same class.
- **Signature:** It typically has the following signature: `ClassName(const ClassName& other);`
    - It takes a reference to an existing object of the same class as its parameter (usually a `const` reference to prevent accidental modification of the original object).
- **Invocation:** The copy constructor is called in several situations:
    - When an object is initialized with another object of the same class (e.g., `ClassName obj2 = obj1;` or `ClassName obj2(obj1);`).
    - When an object of the class is passed by value to a function.
    - When an object of the class is returned by value from a function.
- **Default Copy Constructor:** If you don't explicitly define a copy constructor, the compiler provides a default copy constructor. This default constructor performs a **shallow copy**, meaning it copies the values of the data members from the source object to the new object. For primitive types, this is usually fine. However, if the class contains pointers to dynamically allocated memory, a shallow copy will cause both objects to point to the same memory location. This can lead to issues like double deletion and unintended data sharing.
- **Deep Copy:** If your class manages dynamically allocated resources, you often need to define your own copy constructor that performs a **deep copy**. A deep copy involves allocating new memory for the copied object and then copying the actual data from the source object to this new memory. This ensures that the copied object has its own independent set of resources.

Example of Copy Constructor (demonstrating potential shallow copy issue):

C++
```
#include <iostream>

class MyArray {
private:
    int size;
    int* data;

public:
    MyArray(int sz) : size(sz) {
        data = new int[size];
        for (int i = 0; i < size; ++i) {
            data[i] = i;
        }
        std::cout << "MyArray constructor called for size " << size << std::endl;
    }

    ~MyArray() {
        delete[] data;
        std::cout << "MyArray destructor called for size " << size << std::endl;
    }

    void display() const {
        std::cout << "Data: ";
        for (int i = 0; i < size; ++i) {
            std::cout << data[i] << " ";
        }
        std::cout << std::endl;
    }

    // Without a user-defined copy constructor, a shallow copy will occur
};

int main() {
    MyArray arr1(5);
    arr1.display();

    MyArray arr2 = arr1; // Copy constructor (compiler-generated shallow copy) called
    arr2.display();

    arr1.data[0] = 100; // Modifying arr1's data will also affect arr2's data!
```

```
    arr1.display();
    arr2.display();

    // When arr1 and arr2 go out of scope, the same memory will be deleted twice,
leading to an error.
    return 0;
}
```

Example of Copy Constructor (performing deep copy):

C++

```
#include <iostream>

class MyArray {
private:
    int size;
    int* data;

public:
    MyArray(int sz) : size(sz) {
        data = new int[size];
        for (int i = 0; i < size; ++i) {
            data[i] = i;
        }
        std::cout << "MyArray constructor called for size " << size << std::endl;
    }

    // User-defined copy constructor (deep copy)
    MyArray(const MyArray& other) : size(other.size) {
        std::cout << "MyArray copy constructor (deep copy) called for size " << size <<
std::endl;
        data = new int[size];
        for (int i = 0; i < size; ++i) {
            data[i] = other.data[i];
        }
    }

    ~MyArray() {
        delete[] data;
        std::cout << "MyArray destructor called for size " << size << std::endl;
    }

    void display() const {
        std::cout << "Data: ";
        for (int i = 0; i < size; ++i) {
            std::cout << data[i] << " ";
        }
        std::cout << std::endl;
    }
};

int main() {
    MyArray arr1(5);
    arr1.display();

    MyArray arr2 = arr1; // User-defined copy constructor (deep copy) called
    arr2.display();

    arr1.data[0] = 100; // Modifying arr1's data will NOT affect arr2's data
    arr1.display();
    arr2.display();

    return 0;
}
```

4. Dynamic Constructor (Less Common Term)

The term "dynamic constructor" is not a standard C++ terminology for a specific type of constructor. However, it is sometimes used informally to refer to constructors that:

- **Allocate Dynamic Memory:** Constructors that use the `new` operator to allocate memory on the heap for the object's members. The `MyArray` examples above have constructors that could be considered "dynamic" in this sense.
- **Perform Operations at Runtime:** Constructors whose behavior might depend on runtime conditions or input. However, this is a characteristic of many constructors, especially parameterized ones.

It's generally better to use more specific terms like "default constructor," "parameterized constructor," and "copy constructor" for clarity. When discussing constructors that allocate dynamic memory, it's best to describe them by their primary function (e.g., "constructor that allocates dynamic memory for the `data` member").

5. Constructor with Default Arguments

- **Definition:** A constructor where some or all of its parameters have default values.
- **Invocation:** When an object is created using such a constructor, you can omit the arguments that have default values. The compiler will use the default values for the missing arguments.
- **Benefits:** Reduces the need for multiple overloaded constructors with similar functionality.

Example of Constructor with Default Arguments:

C++
```
#include <iostream>
#include <string>

class Rectangle {
private:
    int length;
    int width;

public:
    Rectangle(int len = 1, int wid = 1) {
        std::cout << "Rectangle constructor with default arguments called." <<
std::endl;
        length = len;
        width = wid;
    }

    int area() const {
        return length * width;
    }

    void display() const {
        std::cout << "Length: " << length << ", Width: " << width << std::endl;
    }
};

int main() {
    Rectangle rect1;      // Uses default length=1, width=1
    Rectangle rect2(5);   // Uses length=5, default width=1
    Rectangle rect3(10, 20); // Uses length=10, width=20

    rect1.display();
    rect2.display();
    rect3.display();

    return 0;
}
```

Key Takeaways about Types of Constructors:

- **Default Constructor:** No arguments, essential for creating objects without explicit initialization.
- **Parameterized Constructor:** Accepts arguments to initialize members with specific values.
- **Copy Constructor:** Creates a new object as a copy of an existing one; crucial for handling objects with dynamically allocated resources (deep copy vs. shallow copy).
- **Dynamic Constructor (Informal):** Often refers to constructors that allocate dynamic memory.
- **Constructor with Default Arguments:** Provides flexibility in object creation by allowing omission of arguments with predefined default values.

3.2 Destructors

Destructors are special member functions of a class that are automatically called when an object of that class is destroyed (goes out of scope or is explicitly deleted). Their primary purpose is to perform cleanup operations, such as releasing resources that were acquired by the object during its lifetime (often in the constructor).

Key Characteristics of Destructors:

- **Same Name as Class with a Tilde (~):** A destructor has the same name as the class it belongs to, but it is prefixed with a tilde (`~`). For example, for a class named `MyClass`, the destructor would be named `~MyClass()`.
- **No Arguments:** Destructors do not take any arguments. Therefore, a class can have only one destructor.
- **No Return Type:** Destructors do not have a return type, not even `void`. Their job is to perform cleanup, not to return a value.
- **Automatic Invocation:** Destructors are invoked automatically by the compiler in the following scenarios:
    - When a local object goes out of scope (e.g., at the end of a function block).
    - When a dynamically allocated object is explicitly deleted using the `delete` operator.
    - When a temporary object's lifetime ends.
    - During stack unwinding in exception handling.
- **Purpose of Destructors:**
    - **Resource Deallocation:** The most common use is to release resources acquired by the object, such as:
        - Deallocating dynamic memory allocated using `new` or `new[]` (using `delete` or `delete[]` respectively).
        - Closing open files.
        - Releasing network connections.
        - Releasing locks or other system resources.
    - **Performing Final Cleanup:** Any other necessary cleanup operations before the object is destroyed can be performed in the destructor.

Example Demonstrating a Destructor:

C++
```
#include <iostream>

class MyClass {
public:
    MyClass() {
        std::cout << "MyClass constructor called." << std::endl;
        // Acquire resources if needed
    }

    ~MyClass() {
        std::cout << "MyClass destructor called." << std::endl;
        // Release resources if needed
    }
```

```
};

void someFunction() {
    std::cout << "Entering someFunction" << std::endl;
    MyClass localObj; // Constructor called when localObj is created
    std::cout << "Leaving someFunction" << std::endl;
    // Destructor called automatically when localObj goes out of scope
}

int main() {
    std::cout << "Entering main" << std::endl;
    MyClass mainObj; // Constructor called
    someFunction();
    std::cout << "Leaving main" << std::endl;
    // Destructor called automatically when mainObj goes out of scope
    return 0;
}
```

Example Demonstrating Resource Deallocation in a Destructor:

C++

```
#include <iostream>

class DynamicArray {
private:
    int* data;
    int size;

public:
    DynamicArray(int sz) : size(sz) {
        data = new int[size];
        std::cout << "DynamicArray constructor called, allocated memory for " << size <<
" integers." << std::endl;
    }

    ~DynamicArray() {
        delete[] data;
        std::cout << "DynamicArray destructor called, deallocated memory." << std::endl;
    }

    void fill() {
        for (int i = 0; i < size; ++i) {
            data[i] = i * 2;
        }
    }

    void display() const {
        std::cout << "Array elements: ";
        for (int i = 0; i < size; ++i) {
            std::cout << data[i] << " ";
        }
        std::cout << std::endl;
    }
};

int main() {
    DynamicArray arr(5); // Constructor allocates memory
    arr.fill();
    arr.display();
    // Destructor will be called automatically when 'arr' goes out of scope, releasing
the allocated memory.
    return 0;
}
```

Importance of Destructors:

- **Preventing Memory Leaks:** When a class allocates dynamic memory, it's crucial to deallocate that memory in the destructor to prevent memory leaks. If you forget to do this, the memory will remain allocated even after the object is no longer needed, potentially leading to program instability or crashes over time.
- **Resource Management:** Destructors ensure that other acquired resources (like file handles, network connections, etc.) are properly released when the object is no longer in use. This is essential for maintaining system integrity and preventing resource exhaustion.
- **RAII (Resource Acquisition Is Initialization):** Constructors and destructors are fundamental to the RAII programming idiom in C++. RAII dictates that resources should be acquired during object initialization (in the constructor) and released during object destruction (in the destructor. This automatic management of resources through object lifecycle helps write safer and more robust code.

When to Define a Destructor:

You typically need to define a destructor for a class if it:

- **Allocates dynamic memory using `new` or `new[]`.**
- **Opens files.**
- **Establishes network connections.**
- **Acquires locks or other system resources.**
- **Needs to perform any specific cleanup actions before the object is destroyed.**

If your class does not manage any such resources, the compiler-provided default destructor (which does nothing beyond calling the destructors of its member objects) is usually sufficient.

Virtual Destructors and Inheritance:

In inheritance hierarchies, especially when dealing with polymorphism (using base class pointers to point to derived class objects), it's crucial to declare the base class's destructor as `virtual`. This ensures that when a derived class object is deleted through a base class pointer, the derived class's destructor is also called, preventing resource leaks and ensuring proper cleanup of the derived class's specific resources. This is a more advanced topic covered in detail in the context of inheritance and polymorphism.

Unit 4.0: Operator Overloading & Inheritance

4.1 Defining Operator Overloading

Operator overloading is a mechanism in C++ that allows you to redefine the meaning of standard C++ operators (like `+`, `-`, `*`, `/`, `==`, `!=`, etc.) so that they can operate on objects of user-defined types (classes). This enables you to use these operators with your custom classes in a way that is natural and consistent with their behavior on built-in types.

Why Overload Operators?

- **Enhanced Readability and Intuition:** Operator overloading allows you to write code that is more expressive and easier to understand. For example, if you have a `Vector` class, overloading the `+` operator to perform vector addition makes the code `vector3 = vector1 + vector2;` much more intuitive than a function call like `vector3 = vector1.add(vector2);`.
- **Code Consistency:** By overloading operators, you can make your user-defined types behave more like built-in types, leading to more consistent and predictable code.
- **Facilitating Library Design:** Operator overloading is heavily used in many C++ libraries (like the Standard Template Library - STL) to provide a natural and efficient way to work with custom objects.

Operators that Can Be Overloaded:

Most C++ operators can be overloaded, including:

- **Arithmetic Operators:** `+`, `-`, `*`, `/`, `%`, `++`, `--`
- **Relational Operators:** `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Bitwise Operators:** `&`, `|`, `^`, `~`, `<<`, `>>`
- **Assignment Operators:** `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
- **Subscript Operator:** `[]`
- **Function Call Operator:** `()`
- **Member Access Operators:** `.` (non-static member access - can only be overloaded in a very specific way), `->` (smart pointers often overload this)
- **Memory Management Operators:** `new`, `delete`, `new[]`, `delete[]` (these are overloaded as static member functions)
- **Insertion (<<) and Extraction (>>) Operators:** Used for input/output streams.
- **Comma Operator:** `,`

Operators that Cannot Be Overloaded:

A few operators cannot be overloaded in C++:

- **Scope Resolution Operator:** `::`
- **Member Access Operator (for direct member access):** `.`
- **Pointer-to-Member Operators:** `.*`, `->*`
- **Ternary Conditional Operator:** `?:`
- **`sizeof` Operator**
- **`typeid` Operator**
- **`alignof` Operator**
- **`noexcept` Operator**
- **`static_cast`, `dynamic_cast`, `reinterpret_cast`, `const_cast`**

Defining Operator Overloading:

Operator overloading is achieved by defining special member functions (or sometimes non-member functions or friend functions) that have the keyword `operator` followed by the symbol of the operator you

want to overload. The name of the operator function is `operator` followed by the operator symbol (e.g., `operator+`, `operator==`, `operator<<`).

Syntax for Overloading a Binary Operator (as a member function):

C++
```
class MyClass {
public:
    // ... other members ...

    ReturnType operator op (const MyClass& other) {
        // Implementation to perform the operation 'op'
        // using the current object (*this) and the 'other' object.
        // Return the result of the operation.
    }
};
```

Here:

- `ReturnType` is the type of the value returned by the operation.
- `operator op` is the name of the operator function (e.g., `operator+`).
- `op` is the binary operator being overloaded (requires two operands).
- `const MyClass& other` is the parameter representing the right-hand operand. The `const` keyword indicates that the `other` object will not be modified, and the `&` indicates that it's passed by reference (for efficiency).

Syntax for Overloading a Unary Operator (as a member function):

- **Prefix (`++obj`, `--obj`, `!obj`, `-obj`, `+obj`):**

C++
```
class MyClass {
public:
    // ... other members ...

    MyClass& operator op () {
        // Implementation to perform the unary operation 'op'
        // on the current object (*this).
        // Return a reference to the modified object (*this).
    }
};
```

- **Postfix (`obj++`, `obj--`):** Postfix operators are typically overloaded with an extra dummy `int` parameter to distinguish them from the prefix versions.

C++
```
class MyClass {
public:
    // ... other members ...

    MyClass operator op (int) {
        // Implementation to perform the postfix unary operation 'op'
        // on the current object (*this).
        // Return the original value of the object before modification.
        MyClass temp = *this; // Save the current state
        // Modify *this
        return temp; // Return the saved original state
    }
};
```

Syntax for Overloading a Binary Operator (as a non-member function or friend function):

C++
```
// Non-member function
```

```
ReturnType operator op (const MyClass& obj1, const MyClass& obj2) {
    // Implementation to perform the operation 'op'
    // using obj1 and obj2.
    // Return the result.
}

// Friend function (declared inside the class definition)
class MyClass {
    // ... other members ...
    friend ReturnType operator op (const MyClass& obj1, const MyClass& obj2);
};

// Definition of the friend function (outside the class)
ReturnType operator op (const MyClass& obj1, const MyClass& obj2) {
    // Implementation (can access private members of MyClass)
    // using obj1 and obj2.
    // Return the result.
}
```

Choosing Between Member and Non-Member (Friend) Functions:

- **Member Functions:** Are usually preferred for operators that modify the object's state (like `+=`, `++`, `=`) or that logically act upon a single object (though binary operators are also commonly overloaded as members). The left-hand operand becomes the implicit `*this` object.
- **Non-Member (Friend) Functions:** Are often used for binary operators where symmetry might be desired (e.g., `a + b` should work the same way as `b + a` if `a` and `b` are of different but compatible types, which might require non-member access for one of them). They are also commonly used for overloading I/O stream operators (`<<` and `>>`). Friend functions can access the private members of the class.

Example: Overloading the `+` Operator for a `Point` Class (as a member function):

C++

```
#include <iostream>

class Point {
private:
    int x;
    int y;

public:
    Point(int xCoord = 0, int yCoord = 0) : x(xCoord), y(yCoord) {}

    void display() const {
        std::cout << "(" << x << ", " << y << ")";
    }

    // Overloading the + operator
    Point operator+(const Point& other) const {
        Point sum;
        sum.x = x + other.x;
        sum.y = y + other.y;
        return sum;
    }
};

int main() {
    Point p1(1, 2);
    Point p2(3, 4);
    Point p3;

    p3 = p1 + p2; // Using the overloaded + operator
    std::cout << "p1: "; p1.display(); std::cout << std::endl;
    std::cout << "p2: "; p2.display(); std::cout << std::endl;
    std::cout << "p1 + p2 = p3: "; p3.display(); std::cout << std::endl;

    return 0;
}
```

Example: Overloading the << Operator for a `Point` Class (as a friend function):

C++
```
#include <iostream>

class Point {
private:
    int x;
    int y;

public:
    Point(int xCoord = 0, int yCoord = 0) : x(xCoord), y(yCoord) {}

    // Friend function to overload the << operator
    friend std::ostream& operator<<(std::ostream& out, const Point& p);
};

std::ostream& operator<<(std::ostream& out, const Point& p) {
    out << "(" << p.x << ", " << p.y << ")";
    return out;
}

int main() {
    Point p1(1, 2);
    std::cout << "Point p1: " << p1 << std::endl; // Using the overloaded << operator

    return 0;
}
```

Important Considerations for Operator Overloading:

- **Maintain Intuitive Meaning:** The overloaded operator should behave in a way that is consistent with its standard meaning for built-in types. Avoid overloading operators in a way that is surprising or counter-intuitive. For example, overloading `+` for subtraction would be bad practice.
- **Return Types:** Choose appropriate return types for your overloaded operators. For arithmetic operators, returning a new object (by value) is common. For operators that modify the object (like `++` prefix), returning a reference to the modified object (`MyClass&`) is typical.
- **Const Correctness:** Make your operator functions `const` if they do not modify the object on which they are called (for member functions) or the operand objects (for non-member functions passed by reference). Also, ensure that you are working with `const` references where appropriate to avoid unnecessary copying.
- **Symmetry for Binary Operators:** If an operation is conceptually symmetric (e.g., addition), consider overloading it as a non-member friend function to allow for implicit type conversions on both operands. For example, if you have a `Complex` class and want `complex + double` and `double + complex` to work, a friend function might be more suitable.
- **Assignment Operator (`=`):** This is a special operator and requires careful implementation, especially when dealing with dynamic memory management (to avoid self-assignment issues and ensure proper deep copying).
- **Subscript Operator (`[]`):** Overloading this allows you to provide array-like access to the elements of your custom container classes.
- **Function Call Operator (`()`):** Overloading this allows objects of your class to be "called" like functions (function objects or functors).

4.2 Rules for Operator Overloading

While operator overloading provides a powerful way to customize the behavior of operators for user-defined types, it's essential to follow certain rules and guidelines to ensure that the overloaded operators are well-behaved, intuitive, and maintainable. Here are the key rules for operator overloading in C++:

1. Preserve the Arity of the Operator:

- **Arity** refers to the number of operands an operator takes.
- You cannot change the arity of an operator when you overload it.
    - Unary operators (`++`, `--`, `!`, `-`) must remain unary.
    - Binary operators (`+`, `-`, `*`, `/`, `==`, `!=`) must remain binary.
    - The ternary operator (`?:`) cannot be overloaded at all.

2. Preserve the Precedence and Associativity of the Operator:

- **Precedence** determines the order in which operators are evaluated in an expression (e.g., `*` and `/` have higher precedence than `+` and `-`).
- **Associativity** determines the direction of evaluation for operators with the same precedence (e.g., left-to-right for `+` and `-`, right-to-left for assignment operators).
- Operator overloading does **not** allow you to change the precedence or associativity of the standard C++ operators. Your overloaded operator will have the same precedence and associativity as its built-in counterpart.

3. At Least One Operand Must Be a User-Defined Type:

- When you overload an operator, at least one of the operands involved in the operation must be an object of the class for which you are defining the overloaded operator (or a reference to that object).
- You cannot overload operators to work exclusively with built-in types in a new way. For example, you cannot redefine the behavior of `int + int`.

4. Certain Operators Cannot Be Overloaded:

As mentioned in the previous section, some operators cannot be overloaded. These include:

- Scope Resolution Operator (`::`)
- Member Access Operator (`.`)
- Pointer-to-Member Operators (`.*`, `->*`)
- Ternary Conditional Operator (`?:`)
- `sizeof` Operator
- `typeid` Operator
- `alignof` Operator
- `noexcept` Operator
- Type casting operators (`static_cast`, `dynamic_cast`, `reinterpret_cast`, `const_cast`)

5. Overloading as Member Functions vs. Non-Member (Friend) Functions:

- **Member Functions:**
    - For a binary operator overloaded as a member function, the left-hand operand is implicitly the `*this` object, and the right-hand operand is passed as an argument.
    - For a unary operator overloaded as a member function (without the dummy `int` for postfix), it operates on the `*this` object.
    - The member function has access to all members (public, private, protected) of the class.
- **Non-Member (Friend) Functions:**
    - For a binary operator overloaded as a non-member function, both the left-hand and right-hand operands must be passed as arguments.
    - For a unary operator overloaded as a non-member function, the single operand is passed as an argument.

- Friend functions can be granted access to the private and protected members of the class.
- Non-member functions are often preferred for symmetric binary operators or when the left-hand operand might be of a different type (for which the overloaded operator is not a member). They are also essential for overloading I/O stream operators (`<<`, `>>`).

6. Overloading Assignment Operator (`=`):

- The assignment operator (`=`) must be overloaded as a **non-static member function**.
- It's crucial to handle self-assignment (`obj = obj;`) correctly to avoid unintended consequences.
- When the class manages dynamic memory, the overloaded assignment operator should perform a **deep copy** of the data.
- It should typically return a reference to the left-hand object (`ClassName& operator=(const ClassName& other);`) to allow for chaining of assignments (`a = b = c;`).

7. Overloading Subscript Operator (`[]`):

- The subscript operator (`[]`) is typically overloaded as a member function to provide array-like access to the elements of a container class.
- It's common to provide two versions: a non-`const` version that returns a reference to the element (allowing modification) and a `const` version that returns a `const` reference to the element (for accessing elements of `const` objects).

8. Overloading Function Call Operator (`()`):

- The function call operator (`()`) must be overloaded as a **non-static member function**.
- Overloading this operator allows objects of your class to be called like functions (function objects or functors). It can take any number of arguments of any type.

9. Overloading Increment (`++`) and Decrement (`--`) Operators:

- When overloading the prefix (`++obj`) and postfix (`obj++`) versions, they must be distinguished. This is typically done by having the postfix version take a dummy `int` argument (which is not used).
- The prefix version should usually return a reference to the modified object (`ClassName& operator++();`).
- The postfix version should usually return a copy of the original object (before modification) by value (`ClassName operator++(int);`).

10. Maintain Intuitive Behavior:

- The most important rule is to overload operators in a way that is consistent with their expected behavior. The overloaded operator should perform an operation that is analogous to its built-in meaning.
- Avoid overloading operators to perform completely unrelated or surprising actions. This can lead to code that is difficult to understand and debug.
- For example, if you have a `ComplexNumber` class, overloading `+` for addition and `-` for subtraction makes sense. Overloading `*` to perform some other unrelated operation would be confusing.

Example Illustrating Some Rules:

C++

```
#include <iostream>

class Counter {
private:
    int count;

public:
    Counter(int c = 0) : count(c) {}
```

```
    // Overloading prefix increment (member function, returns reference)
    Counter& operator++() {
        ++count;
        return *this;
    }

    // Overloading postfix increment (member function, returns copy)
    Counter operator++(int) {
        Counter temp = *this;
        ++count;
        return temp;
    }

    // Overloading addition (member function, returns new object)
    Counter operator+(const Counter& other) const {
        return Counter(count + other.count);
    }

    // Friend function for output stream (non-member)
    friend std::ostream& operator<<(std::ostream& out, const Counter& c);

    int getCount() const {
        return count;
    }
};

std::ostream& operator<<(std::ostream& out, const Counter& c) {
    out << c.count;
    return out;
}

int main() {
    Counter c1(5);
    Counter c2;

    std::cout << "c1: " << c1 << std::endl; // Uses overloaded <<

    Counter c3 = ++c1; // Uses overloaded prefix ++
    std::cout << "++c1: " << c3 << ", c1: " << c1 << std::endl;

    Counter c4 = c2++; // Uses overloaded postfix ++
    std::cout << "c2++: " << c4 << ", c2: " << c2 << std::endl;

    Counter c5 = c1 + c2; // Uses overloaded +
    std::cout << "c1 + c2: " << c5 << std::endl;

    return 0;
}
```

4.3 Overloading Unary Operators

Unary operators are those that operate on a single operand. In C++, common unary operators include:

- **Arithmetic:** + (unary plus), – (unary minus), ++ (increment), –– (decrement)
- **Logical:** ! (logical NOT)
- **Bitwise:** ~ (bitwise NOT)

You can overload these unary operators for your user-defined classes using either member functions or friend functions.

4.3.1 Overloading Unary Operators using Member Functions

When you overload a unary operator using a member function, the object on which the operator is applied becomes the implicit operand (`*this`). The member function for overloading a unary operator typically takes no explicit arguments (for prefix operators) or a dummy `int` argument (for postfix increment/decrement).

Syntax for Overloading Prefix Unary Operators (e.g., `++obj`, `--obj`, `!obj`, `-obj`, `+obj`):

C++

```
class MyClass {
public:
    // ... other members ...

    ReturnType operator op () {
        // Implementation of the unary operation 'op'
        // on the current object (*this).
        // Modify the object if necessary.
        // Return the result (often a reference to the modified object).
    }
};
```

Here, `op` represents the unary operator being overloaded. The function takes no arguments because it operates on the object that calls it.

Syntax for Overloading Postfix Increment/Decrement Operators (e.g., `obj++`, `obj--`):

To distinguish postfix from prefix, a dummy `int` argument is used. This argument is never actually used in the implementation; it's just a convention for the compiler. The postfix operator should typically return a copy of the object *before* the increment/decrement occurs.

C++

```
class MyClass {
public:
    // ... other members ...

    ReturnType operator op (int) {
        // Implementation of the postfix unary operation 'op'
        // on the current object (*this).
        // Save the current state of the object.
        // Modify the object.
        // Return the saved original state (often by value).
    }
};
```

Examples using Member Functions:

1. Overloading Unary Minus (-)

C++

```
#include <iostream>

class Number {
private:
    int value;

public:
    Number(int val = 0) : value(val) {}

    void display() const {
        std::cout << "Value: " << value << std::endl;
    }

    // Overloading unary minus
    Number operator-() const {
        return Number(-value); // Returns a new Number object with the negated value
    }
};

int main() {
    Number n1(10);
```

```
    std::cout << "n1: "; n1.display();

    Number n2 = -n1; // Calls the overloaded unary minus operator
    std::cout << "-n1 (n2): "; n2.display();

    return 0;
}
```

2. Overloading Prefix Increment (++)

C++

```
#include <iostream>

class Counter {
private:
    int count;

public:
    Counter(int c = 0) : count(c) {}

    void display() const {
        std::cout << "Count: " << count << std::endl;
    }

    // Overloading prefix increment
    Counter& operator++() {
        ++count;
        return *this; // Returns a reference to the modified object
    }
};

int main() {
    Counter c1(5);
    std::cout << "c1: "; c1.display();

    Counter& c2 = ++c1; // Calls the overloaded prefix ++
    std::cout << "++c1 (c2): "; c2.display();
    std::cout << "c1 after ++: "; c1.display(); // c1 is also modified

    return 0;
}
```

3. Overloading Postfix Increment (++)

C++

```
#include <iostream>

class Counter {
private:
    int count;

public:
    Counter(int c = 0) : count(c) {}

    void display() const {
        std::cout << "Count: " << count << std::endl;
    }

    // Overloading postfix increment
    Counter operator++(int) {
        Counter temp = *this; // Save the original value
        ++count;              // Increment the current object
        return temp;          // Return the saved original value
    }
};

int main() {
    Counter c1(5);
```

```
    std::cout << "c1: "; c1.display();

    Counter c2 = c1++; // Calls the overloaded postfix ++
    std::cout << "c1++ (c2): "; c2.display(); // c2 has the original value
    std::cout << "c1 after ++: "; c1.display(); // c1 is incremented

    return 0;
}
```

4.3.2 Overloading Unary Operators using Friend Functions

You can also overload unary operators using friend functions. When a unary operator is overloaded as a friend function, it takes one argument, which is the object on which the operator is applied.

Syntax for Overloading Prefix Unary Operators (e.g., `!obj`, `-obj`, `+obj`) as Friend Functions:

C++

```
class MyClass {
    // ... other members ...
    friend ReturnType operator op (const MyClass& obj);
};

// Definition of the friend function outside the class
ReturnType operator op (const MyClass& obj) {
    // Implementation of the unary operation 'op'
    // on the 'obj' object.
    // Return the result.
}
```

Syntax for Overloading Prefix Increment/Decrement Operators (e.g., `++obj`, `--obj`) as Friend Functions:

C++

```
class MyClass {
    // ... other members ...
    friend ReturnType operator op (MyClass& obj);
};

// Definition of the friend function outside the class
ReturnType operator op (MyClass& obj) {
    // Implementation of the unary operation 'op'
    // on the 'obj' object.
    // Modify the 'obj' object.
    // Return the result (often a reference to the modified object).
}
```

Syntax for Overloading Postfix Increment/Decrement Operators (e.g., `obj++`, `obj--`) as Friend Functions:

Similar to member functions, a dummy `int` argument is used to distinguish postfix friend operators.

C++

```
class MyClass {
    // ... other members ...
    friend ReturnType operator op (MyClass& obj, int);
};

// Definition of the friend function outside the class
ReturnType operator op (MyClass& obj, int) {
    // Implementation of the postfix unary operation 'op'
    // on the 'obj' object.
    // Save the original state.
    // Modify 'obj'.
    // Return the saved original state (often by value).
}
```

Examples using Friend Functions:

1. Overloading Logical NOT (!)

C++

```
#include <iostream>

class Flag {
private:
    bool status;

public:
    Flag(bool s = false) : status(s) {}

    void display() const {
        std::cout << "Status: " << (status ? "true" : "false") << std::endl;
    }

    friend Flag operator!(const Flag& f);
};

// Friend function for overloading !
Flag operator!(const Flag& f) {
    return Flag(!f.status);
}

int main() {
    Flag f1(true);
    std::cout << "f1: "; f1.display();

    Flag f2 = !f1; // Calls the overloaded ! operator
    std::cout << "!f1 (f2): "; f2.display();

    return 0;
}
```

2. Overloading Prefix Increment (++) as a Friend Function

C++

```
#include <iostream>

class Counter {
private:
    int count;

public:
    Counter(int c = 0) : count(c) {}

    void display() const {
        std::cout << "Count: " << count << std::endl;
    }

    friend Counter& operator++(Counter& c);
};

// Friend function for overloading prefix ++
Counter& operator++(Counter& c) {
    ++c.count;
    return c;
}

int main() {
    Counter c1(5);
    std::cout << "c1: "; c1.display();

    Counter& c2 = ++c1; // Calls the overloaded prefix ++
    std::cout << "++c1 (c2): "; c2.display();
    std::cout << "c1 after ++: "; c1.display();
```

```
    return 0;
}
```

Choosing Between Member and Friend Functions for Unary Operators:

- **Member Functions:** Often considered more natural for unary operators that modify the object's state (like increment and decrement) or that conceptually belong tightly to the class. They have direct access to the object's members.
- **Friend Functions:** Can be useful when the unary operation doesn't inherently modify the object or when you need symmetry or want to provide the operator as a free function in the namespace of the class. For operators like logical NOT or unary minus that typically return a new object, either approach can be suitable. Friend functions might be preferred if you want the operand to be implicitly convertible to the class type in some scenarios (though this is less common for unary operators).

4.4 Overloading Binary Operators

Binary operators are those that operate on two operands. In C++, common binary operators include:

- **Arithmetic:** `+`, `-`, `*`, `/`, `%`
- **Relational:** `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Bitwise:** `&`, `|`, `^`, `<<`, `>>`
- **Assignment (compound):** `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
- **Subscript:** `[]` (can also be used in a way that appears unary, but conceptually involves the object and the index)
- **Function Call:** `()` (can take multiple arguments, making it technically n-ary, but often discussed in the context of binary-like usage with the object and parameters)

You can overload binary operators for your user-defined classes using either member functions or friend functions.

4.4.1 Overloading Binary Operators using Member Functions

When you overload a binary operator using a member function, the left-hand operand of the operator becomes the implicit operand (`*this`), and the right-hand operand is passed as an explicit argument to the member function.

Syntax for Overloading Binary Operators (e.g., `op` can be `+`, `-`, `*`, `/`, `==`, etc.):

C++

```
class MyClass {
public:
    // ... other members ...

    ReturnType operator op (const MyClass& rhs) const {
        // Implementation of the binary operation 'op'
        // using the current object (*this) and the right-hand side operand 'rhs'.
        // The 'const' at the end indicates that this operation does not modify
        // the state of the current object.
        // Return the result of the operation (often a new object).
    }
};
```

Here, `rhs` (right-hand side) is a common name for the parameter representing the second operand. The `const` keyword at the end of the function signature is often used if the operation does not modify the left-hand operand (`*this`). The parameter `rhs` is often passed as a `const` reference for efficiency and to prevent accidental modification.

Examples using Member Functions:

1. Overloading the Addition Operator (+)

C++

```
#include <iostream>

class Point {
private:
    int x;
    int y;

public:
    Point(int xCoord = 0, int yCoord = 0) : x(xCoord), y(yCoord) {}

    void display() const {
        std::cout << "(" << x << ", " << y << ")";
    }

    // Overloading the + operator as a member function
    Point operator+(const Point& other) const {
        return Point(x + other.x, y + other.y); // Returns a new Point object
    }
};

int main() {
    Point p1(1, 2);
    Point p2(3, 4);
    Point p3;

    p3 = p1 + p2; // Calls the overloaded + operator (p1 is *this, p2 is other)
    std::cout << "p1: "; p1.display(); std::cout << std::endl;
    std::cout << "p2: "; p2.display(); std::cout << std::endl;
    std::cout << "p1 + p2 = p3: "; p3.display(); std::cout << std::endl;

    return 0;
}
```

2. Overloading the Equality Operator (==)

C++

```
#include <iostream>

class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

    bool operator==(const Complex& other) const {
        return (real == other.real && imag == other.imag);
    }

    void display() const {
        std::cout << real << " + " << imag << "i";
    }
};

int main() {
    Complex c1(1.0, 2.0);
    Complex c2(1.0, 2.0);
    Complex c3(3.0, 4.0);

    if (c1 == c2) { // Calls the overloaded == operator
        std::cout << "c1 and c2 are equal." << std::endl;
    } else {
```

```
        std::cout << "c1 and c2 are not equal." << std::endl;
    }

    if (c1 == c3) {
        std::cout << "c1 and c3 are equal." << std::endl;
    } else {
        std::cout << "c1 and c3 are not equal." << std::endl;
    }

    return 0;
}
```

4.4.2 Overloading Binary Operators using Friend Functions

You can also overload binary operators using friend functions. When a binary operator is overloaded as a friend function, it takes two arguments, representing the left-hand and right-hand operands.

Syntax for Overloading Binary Operators (e.g., `op` can be +, -, *, /, ==, etc.) as Friend Functions:

C++

```
class MyClass {
    // ... other members ...
    friend ReturnType operator op (const MyClass& lhs, const MyClass& rhs);
};

// Definition of the friend function outside the class
ReturnType operator op (const MyClass& lhs, const MyClass& rhs) {
    // Implementation of the binary operation 'op'
    // using the left-hand side operand 'lhs' and the right-hand side operand 'rhs'.
    // Return the result of the operation (often a new object).
}
```

Here, `lhs` (left-hand side) and `rhs` (right-hand side) are common names for the parameters representing the two operands. They are often passed as `const` references for efficiency and to prevent accidental modification.

Examples using Friend Functions:

1. Overloading the Addition Operator (+) as a Friend Function

C++

```
#include <iostream>

class Vector {
private:
    int x;
    int y;

public:
    Vector(int xComp = 0, int yComp = 0) : x(xComp), y(yComp) {}

    void display() const {
        std::cout << "[" << x << ", " << y << "]";
    }

    friend Vector operator+(const Vector& v1, const Vector& v2);
};

// Friend function for overloading +
Vector operator+(const Vector& v1, const Vector& v2) {
    return Vector(v1.x + v2.x, v1.y + v2.y);
}

int main() {
    Vector vec1(1, 2);
    Vector vec2(3, 4);
```

```
    Vector vec3;

    vec3 = vec1 + vec2; // Calls the overloaded + operator (vec1 is lhs, vec2 is rhs)
    std::cout << "vec1: "; vec1.display(); std::cout << std::endl;
    std::cout << "vec2: "; vec2.display(); std::cout << std::endl;
    std::cout << "vec1 + vec2 = vec3: "; vec3.display(); std::cout << std::endl;

    return 0;
}
```

2. Overloading the Output Stream Operator (<<)

Overloading the output stream operator (`<<`) is typically done as a friend function because the left-hand operand (`std::ostream&`) is not an object of your class.

C++

```
#include <iostream>

class Point {
private:
    int x;
    int y;

public:
    Point(int xCoord = 0, int yCoord = 0) : x(xCoord), y(yCoord) {}

    friend std::ostream& operator<<(std::ostream& out, const Point& p);
};

// Friend function for overloading <<
std::ostream& operator<<(std::ostream& out, const Point& p) {
    out << "(" << p.x << ", " << p.y << ")";
    return out; // Returns the ostream object to allow chaining
}

int main() {
    Point p1(5, 10);
    std::cout << "Point p1: " << p1 << std::endl; // Uses the overloaded << operator

    return 0;
}
```

Choosing Between Member and Friend Functions for Binary Operators:

- **Member Functions:** Often preferred when the left-hand operand is always an object of the class for which the operator is being overloaded, and the operation might modify the left-hand operand (although many binary operators in their mathematical sense do not modify operands and return a new value). They feel more integrated with the class.
- **Friend Functions:** Are particularly useful in the following scenarios:
    - **Symmetry:** When you want the operation to be symmetric with respect to the types of the operands. For example, if you want `myObject + someOtherType` and `someOtherType + myObject` to both work (assuming an appropriate conversion exists or another overload is provided). A friend function allows for implicit conversion on both operands.
    - **Left-hand operand is not the class object:** As seen with the output stream operator (`<<`), if the left-hand operand is of a different type (like `std::ostream`), the operator must be overloaded as a non-member function (which can be a friend if it needs access to private members).
    - **Encapsulation:** Some argue that using non-member functions (even friends) can sometimes lead to better encapsulation by limiting the number of member functions that can directly access the private parts of the class. However, this is a matter of design philosophy.

4.5 Type Conversions

Type conversion, also known as type casting, is the process of converting a value of one data type into a value of another data type. In C++, type conversions can occur implicitly (automatically by the compiler) or explicitly (by the programmer using cast operators). When dealing with user-defined types (classes), you can also define custom type conversions to allow objects of your class to be converted to other types, and for other types to be converted to objects of your class.

Why Define Custom Type Conversions?

- **Integration with Existing Types:** Allows your class objects to interact seamlessly with built-in types and objects of other classes.
- **Flexibility and Expressiveness:** Makes your class more versatile and can lead to more natural and readable code.
- **Avoiding Explicit Casts:** By defining appropriate conversions, you can reduce the need for explicit casts in your code, making it cleaner.

Two Main Ways to Define Custom Type Conversions for Classes:

1. **Conversion Constructors (Converting from another type to your class type):** These are single-argument constructors that are not declared with the `explicit` keyword. They implicitly define a conversion from the type of the argument to the class type.
2. **Conversion Functions (Converting from your class type to another type):** These are member functions of your class that have the following characteristics:
    - They have the name `operator` followed by the target type.
    - They have no return type specified (the return type is implied by the name of the operator).
    - They take no arguments.

1. Conversion Constructors

A constructor that can be called with a single argument acts as a conversion constructor if it is not declared with the `explicit` keyword. This allows the compiler to implicitly convert a value of the argument type to an object of the class type.

Syntax:

C++
```
class MyClass {
public:
    // Conversion constructor from 'OtherType' to 'MyClass'
    MyClass(OtherType arg) {
        // Initialization of MyClass object using 'arg'
    }

    // ... other members ...
};
```

Example:

C++
```
#include <iostream>
#include <string>

class MyString {
private:
    std::string str;

public:
    MyString(const std::string& s) : str(s) {
        std::cout << "MyString(const std::string&) called." << std::endl;
    }
```

```
    MyString(const char* s) : str(s) {
        std::cout << "MyString(const char*) called." << std::endl;
    }

    void display() const {
        std::cout << "String: " << str << std::endl;
    }
};

void printMyString(const MyString& ms) {
    ms.display();
}

int main() {
    MyString s1("Hello"); // Direct initialization using the constructor
    MyString s2 = "World"; // Implicit conversion from const char* to MyString

    printMyString(s1);
    printMyString("C++"); // Implicit conversion from const char* to MyString

    return 0;
}
```

In this example, the constructor `MyString(const char* s)` acts as a conversion constructor, allowing the implicit conversion of a C-style string literal to a `MyString` object.

The `explicit` Keyword:

You can use the `explicit` keyword before a single-argument constructor to prevent it from being used for implicit type conversions. This is often done to avoid unintended or surprising conversions.

Example with `explicit`:

C++

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

class MyString {
private:
    std::string str;

public:
    explicit MyString(const char* s) : str(s) {
        std::cout << "explicit MyString(const char*) called." << std::endl;
    }

    void display() const {
        std::cout << "String: " << str << std::endl;
    }
};

void printMyString(const MyString& ms) {
    ms.display();
}

int main() {
    MyString s1("Hello"); // Direct initialization is still allowed
    // MyString s2 = "World"; // Error: implicit conversion is not allowed

    printMyString(s1);
    // printMyString("C++"); // Error: implicit conversion is not allowed

    printMyString(MyString("C++")); // Explicit conversion is required

    return 0;
}
```

With `explicit`, the constructor can only be used for direct initialization or when an explicit cast to `MyString` is performed.

2. Conversion Functions (Operator Overloading for Type Conversion)

A conversion function is a member function of a class that defines how an object of that class can be converted to another type. It has the form `operator TargetType()`.

Syntax:

C++
```
class MyClass {
public:
    // Conversion function to 'TargetType'
    operator TargetType() const {
        // Code to perform the conversion
        // Return a value of 'TargetType'
    }

    // ... other members ...
};
```

- `operator TargetType()`: This is the name of the conversion function. `TargetType` is the type you want to convert your class object to.
- No return type is specified in the function declaration because the return type is implicitly `TargetType`.
- The function takes no arguments because it operates on the `*this` object.
- The `const` keyword at the end is often used if the conversion does not modify the object's state.

Example:

C++
```
#include <iostream>

class Feet {
private:
    int feet;

public:
    Feet(int f = 0) : feet(f) {}

    int getFeet() const {
        return feet;
    }

    // Conversion function to int (representing total inches)
    operator int() const {
        std::cout << "Feet::operator int() called." << std::endl;
        return feet * 12;
    }

    // Conversion function to double (representing meters)
    operator double() const {
        std::cout << "Feet::operator double() called." << std::endl;
        return feet * 0.3048;
    }
};

void printInches(int inches) {
    std::cout << "Inches: " << inches << std::endl;
}

void printMeters(double meters) {
    std::cout << "Meters: " << meters << std::endl;
}
```

```
int main() {
    Feet distance(10);

    int totalInches = distance; // Implicit conversion to int
    printInches(totalInches);

    double meters = distance; // Implicit conversion to double
    printMeters(meters);

    std::cout << "Distance in feet: " << distance.getFeet() << std::endl;
    std::cout << "Distance in inches (explicit cast): " << (int)distance << std::endl;
    std::cout << "Distance in meters (explicit cast): " << (double)distance <<
std::endl;

    return 0;
}
```

In this example, `Feet` objects can be implicitly converted to `int` (representing inches) and `double` (representing meters) using the defined conversion functions.

Potential Issues with Implicit Conversions:

While implicit conversions can make code more concise, they can also lead to unexpected behavior if not used carefully. The compiler might perform conversions that you didn't intend, potentially causing logical errors.

- **Unintended Constructor Conversions:** A single-argument constructor (without `explicit`) can lead to implicit conversions that you might not expect. Using `explicit` helps prevent this.
- **Ambiguous Conversions:** If a class provides conversions to multiple types, or if there are multiple ways to convert between types (e.g., through conversion constructors in another class), the compiler might face ambiguity and issue an error.

Best Practices for Type Conversions:

- **Use `explicit` for single-argument constructors unless implicit conversion is clearly intended and beneficial.**
- **Be mindful of potential ambiguities when defining multiple conversion constructors or conversion functions.**
- **Design your classes to have intuitive and well-defined conversions.**
- **Consider providing explicit conversion functions (with a clear name, not just `operator TargetType()`) if the conversion is not always safe or obvious.**
- **Overload operators appropriately to work with your class and other types, which can sometimes reduce the need for type conversions.**

4.6 Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class (the **derived class** or **child class**) to inherit properties and behaviors (data members and member functions) from an existing class (the **base class** or **parent class**). This promotes code reusability, establishes an "is-a" relationship between classes, and facilitates the creation of class hierarchies.

Key Benefits of Inheritance:

- **Code Reusability:** Derived classes can reuse the code (data members and member functions) of the base class, reducing redundancy and the effort required to write new code.
- **Extensibility:** Derived classes can extend the functionality of the base class by adding new data members and member functions or by overriding the inherited ones.
- **Maintainability:** Changes made to the base class are automatically reflected in all its derived classes (unless overridden), making maintenance easier.

- **Polymorphism:** Inheritance forms the basis for polymorphism, allowing objects of different classes in the same hierarchy to be treated uniformly through base class pointers or references.
- **"Is-a" Relationship:** Inheritance models an "is-a" relationship. For example, a `Dog` *is a* `Animal`, a `Car` *is a* `Vehicle`.

4.6.1 Defining Derived Classes

To define a derived class in C++, you use the following syntax:

C++

```
class DerivedClassName : access-specifier BaseClassName {
    // Data members and member functions specific to the derived class
    // or overrides of base class members
};
```

Let's break down the components of this syntax:

- **`class DerivedClassName`:** This declares a new class named `DerivedClassName`.
- **`:`:** The colon indicates that `DerivedClassName` is inheriting from another class.
- **`access-specifier`:** This specifies the access control for the inherited members of the `BaseClassName` within the `DerivedClassName`. The common access specifiers used in inheritance are:
    - **`public`:** Public members of the base class remain public in the derived class. Protected members of the base class become public members of the derived class. Private members of the base class are inherited but remain inaccessible directly in the derived class.
    - **`protected`:** Public and protected members of the base class become protected members in the derived class. Private members of the base class are inherited but remain inaccessible directly in the derived class.
    - **`private`:** Public and protected members of the base class become private members in the derived class. Private members of the base class are inherited but remain inaccessible directly in the derived class. This is the default if no access specifier is provided.
- **`BaseClassName`:** This is the name of the class from which `DerivedClassName` is inheriting. A class can inherit from only one direct base class in C++ (single inheritance). However, through a hierarchy of inheritance, a class can indirectly inherit from multiple base classes.
- **`{ ... }`:** This block contains the data members and member functions that are specific to the `DerivedClassName` or are overrides of members inherited from the `BaseClassName`.

Access Control and Inheritance:

The access specifier used during inheritance (`public`, `protected`, or `private`) controls the accessibility of the inherited members within the derived class and for objects of the derived class. Here's a summary:

Base Class Member	`public` Inheritance	`protected` Inheritance	`private` Inheritance
`Public`	`public` in Derived	`protected` in Derived	`private` in Derived
`Protected`	`protected` in Derived	`protected` in Derived	`private` in Derived
`Private`	Inaccessible in Derived	Inaccessible in Derived	Inaccessible in Derived

Important Points about Inheritance and Access:

- **Private members of the base class are always inherited but are not directly accessible in the derived class.** They can only be accessed indirectly through public or protected member functions of the base class (if they exist).
- The choice of access specifier during inheritance depends on the desired relationship between the base and derived classes and how you want the inherited members to be accessible.
- **`public` inheritance** is the most common and is used to model the "is-a" relationship where the derived class is a specialized version of the base class and should inherit the public interface of the base class.

- **`protected` inheritance** is less common and is used when the base class's public interface should become part of the derived class's protected interface (accessible to further derived classes but not directly to objects of the derived class).
- **`private` inheritance** is the least common and is used to implement a "has-a" or "implemented-in-terms-of" relationship, where the base class's implementation details are being reused, but its public interface is not meant to be part of the derived class's public interface.

Example of Defining Derived Classes with `public` Inheritance:

C++

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

// Base class: Animal
class Animal {
public:
    Animal(std::string name) : name_(name) {
        std::cout << "Animal constructor called for " << name_ << std::endl;
    }

    virtual ~Animal() {
        std::cout << "Animal destructor called for " << name_ << std::endl;
    }

    void eat() const {
        std::cout << name_ << " is eating." << std::endl;
    }

    virtual void makeSound() const {
        std::cout << "Animal makes a generic sound." << std::endl;
    }

protected:
    std::string name_;
};

// Derived class: Dog (inherits publicly from Animal)
class Dog : public Animal {
public:
    Dog(std::string name, std::string breed) : Animal(name), breed_(breed) {
        std::cout << "Dog constructor called for " << name_ << " (breed: " << breed_ <<
")" << std::endl;
    }

    ~Dog() override {
        std::cout << "Dog destructor called for " << name_ << std::endl;
    }

    void bark() const {
        std::cout << name_ << " (a " << breed_ << ") is barking: Woof! Woof!" <<
std::endl;
    }

    void makeSound() const override {
        bark(); // Dog's specific sound
    }

private:
    std::string breed_;
};

// Derived class: Cat (inherits publicly from Animal)
class Cat : public Animal {
public:
    Cat(std::string name, std::string color) : Animal(name), color_(color) {
        std::cout << "Cat constructor called for " << name_ << " (color: " << color_ <<
")" << std::endl;
    }
```

```
    ~Cat() override {
        std::cout << "Cat destructor called for " << name_ << std::endl;
    }

    void meow() const {
        std::cout << name_ << " (a " << color_ << " cat) is meowing: Meow!" <<
std::endl;
    }

    void makeSound() const override {
        meow(); // Cat's specific sound
    }

private:
    std::string color_;
};

int main() {
    Dog myDog("Buddy", "Golden Retriever");
    Cat myCat("Whiskers", "Gray");

    myDog.eat();       // Inherited from Animal
    myDog.bark();      // Specific to Dog
    myDog.makeSound(); // Overrides Animal's makeSound

    myCat.eat();       // Inherited from Animal
    myCat.meow();      // Specific to Cat
    myCat.makeSound(); // Overrides Animal's makeSound

    Animal* animalPtr1 = &myDog;
    Animal* animalPtr2 = &myCat;

    animalPtr1->makeSound(); // Polymorphism in action
    animalPtr2->makeSound(); // Polymorphism in action

    return 0;
}
```

In this example:

- `Animal` is the base class with common properties and behaviors of animals.
- `Dog` and `Cat` are derived classes that inherit publicly from `Animal`.
- `Dog` and `Cat` have their own specific data members (`breed_`, `color_`) and member functions (`bark()`, `meow()`).
- They also override the `makeSound()` function to provide their specific sounds.
- The `main()` function demonstrates how objects of the derived classes can use the inherited members and their own members. It also shows a basic example of polymorphism using base class pointers.

4.6.2 Types of Inheritance in C++

C++ supports several types of inheritance, allowing you to structure your class hierarchies in different ways to achieve code reuse and model complex relationships. The main types of inheritance in C++ are:

1. Single Inheritance:

- In single inheritance, a derived class inherits from only one direct base class. This is the most common and straightforward form of inheritance.
- It establishes a clear "is-a" relationship. For example, a `Car` *is a* `Vehicle`.

Syntax:

C++

```
class DerivedClass : access-specifier BaseClass {
    // ... members ...
};
```

Example:

C++
```
#include <iostream>
#include <string>

class Vehicle {
public:
    Vehicle(std::string model) : model_(model) {
        std::cout << "Vehicle created: " << model_ << std::endl;
    }

    void startEngine() const {
        std::cout << "Vehicle engine started." << std::endl;
    }

protected:
    std::string model_;
};

class Car : public Vehicle {
public:
    Car(std::string model, int numDoors) : Vehicle(model), numDoors_(numDoors) {
        std::cout << "Car created: " << model_ << " with " << numDoors_ << " doors." <<
std::endl;
    }

    void drive() const {
        std::cout << "Car is driving." << std::endl;
    }

private:
    int numDoors_;
};

int main() {
    Car myCar("Sedan", 4);
    myCar.startEngine(); // Inherited from Vehicle
    myCar.drive();       // Specific to Car
    return 0;
}
```

2. Multiple Inheritance:

- In multiple inheritance, a derived class inherits from more than one direct base class.
- This can be useful for combining the features of different classes into a single derived class.
- However, it can also lead to complexities like the "diamond problem" (ambiguity when a derived class inherits the same member from two or more base classes through different paths). C++ provides mechanisms like the scope resolution operator (`::`) and virtual inheritance to resolve such ambiguities.

Syntax:

C++
```
class DerivedClass : access-specifier1 BaseClass1, access-specifier2 BaseClass2, ... {
    // ... members ...
};
```

Example:

C++
```
#include <iostream>
```

```
class Engine {
public:
    void start() const {
        std::cout << "Engine started." << std::endl;
    }
};

class Body {
public:
    void assemble() const {
        std::cout << "Body assembled." << std::endl;
    }
};

class Car : public Engine, public Body {
public:
    void drive() const {
        std::cout << "Car is moving." << std::endl;
    }
};

int main() {
    Car myCar;
    myCar.start();    // Inherited from Engine
    myCar.assemble(); // Inherited from Body
    myCar.drive();
    return 0;
}
```

3. Multilevel Inheritance:

- In multilevel inheritance, a derived class inherits from a base class, and then another derived class inherits from that first derived class, forming a chain of inheritance.
- It represents an "is-a-kind-of" relationship extending through multiple levels. For example, `Animal -> Mammal -> Dog`.

Syntax:

C++
```
class BaseClass { /* ... */ };
class DerivedClass1 : public BaseClass { /* ... */ };
class DerivedClass2 : public DerivedClass1 { /* ... */ };
```

Example:

C++
```
#include <iostream>
#include <string>

class Animal {
public:
    Animal(std::string name) : name_(name) {
        std::cout << "Animal created: " << name_ << std::endl;
    }

    void eat() const {
        std::cout << name_ << " is eating." << std::endl;
    }

protected:
    std::string name_;
};

class Mammal : public Animal {
public:
    Mammal(std::string name) : Animal(name) {
        std::cout << "Mammal created: " << name_ << std::endl;
    }
```

```
    void breathe() const {
        std::cout << name_ << " is breathing." << std::endl;
    }
};

class Dog : public Mammal {
public:
    Dog(std::string name, std::string breed) : Mammal(name), breed_(breed) {
        std::cout << "Dog created: " << name_ << " (breed: " << breed_ << ")" <<
std::endl;
    }

    void bark() const {
        std::cout << name_ << " (a " << breed_ << ") is barking." << std::endl;
    }

private:
    std::string breed_;
};

int main() {
    Dog myDog("Buddy", "Labrador");
    myDog.eat();     // Inherited from Animal
    myDog.breathe(); // Inherited from Mammal
    myDog.bark();    // Specific to Dog
    return 0;
}
```

4. Hierarchical Inheritance:

- In hierarchical inheritance, multiple derived classes inherit from a single base class.
- It represents a "is-a" relationship where several classes are specialized versions of a common base class. For example, `Animal` can be the base class for `Dog`, `Cat`, `Bird`, etc.

Syntax:

C++
```
class BaseClass { /* ... */ };
class DerivedClass1 : public BaseClass { /* ... */ };
class DerivedClass2 : public BaseClass { /* ... */ };
// ... and so on
```

Example:

C++
```
#include <iostream>
#include <string>

class Shape {
public:
    Shape(std::string color) : color_(color) {
        std::cout << "Shape created with color: " << color_ << std::endl;
    }

    virtual double area() const {
        return 0.0;
    }

protected:
    std::string color_;
};

class Rectangle : public Shape {
public:
    Rectangle(std::string color, double length, double width)
        : Shape(color), length_(length), width_(width) {
        std::cout << "Rectangle created." << std::endl;
```

```
    }

    double area() const override {
        return length_ * width_;
    }

private:
    double length_;
    double width_;
};

class Circle : public Shape {
public:
    Circle(std::string color, double radius)
        : Shape(color), radius_(radius) {
        std::cout << "Circle created." << std::endl;
    }

    double area() const override {
        return 3.14159 * radius_ * radius_;
    }

private:
    double radius_;
};

int main() {
    Rectangle rect("Red", 5.0, 3.0);
    Circle circ("Blue", 2.5);

    std::cout << "Rectangle area: " << rect.area() << std::endl;
    std::cout << "Circle area: " << circ.area() << std::endl;

    return 0;
}
```

5. Hybrid Inheritance:

- Hybrid inheritance is a combination of two or more of the above types of inheritance. For example, a class might inherit from two base classes (multiple inheritance), and one of those base classes might be part of a multilevel inheritance hierarchy.
- Hybrid inheritance can lead to complex class structures and potential ambiguities, so it should be used carefully and with a clear understanding of the relationships being modeled.

Example (combining Hierarchical and Multiple Inheritance - illustrating the Diamond Problem):

C++

```
#include <iostream>

class Grandparent {
public:
    void display() const {
        std::cout << "Grandparent's display." << std::endl;
    }
};

class Parent1 : public Grandparent {
public:
    void show1() const {
        std::cout << "Parent1's show." << std::endl;
    }
};

class Parent2 : public Grandparent {
public:
    void show2() const {
        std::cout << "Parent2's show." << std::endl;
    }
};
```

```
class Child : public Parent1, public Parent2 {
public:
    void showChild() const {
        std::cout << "Child's show." << std::endl;
    }
};

int main() {
    Child baby;
    baby.show1();
    baby.show2();
    baby.showChild();
    baby.Grandparent::display(); // Resolving ambiguity using scope resolution
    return 0;
}
```

In the hybrid inheritance example above, `Child` inherits `display()` from `Grandparent` through both `Parent1` and `Parent2`, leading to ambiguity. We use the scope resolution operator (`::`) to specify which `Grandparent::display()` we want to call.

The Diamond Problem and Virtual Inheritance:

The "diamond problem" arises in multiple inheritance when a class inherits from two classes that have a common ancestor. This can lead to the derived class inheriting multiple copies of the grandparent's members, causing ambiguity and increased memory usage.

C++ provides **virtual inheritance** to solve the diamond problem. By declaring the inheritance from the common ancestor as `virtual`, only one shared instance of the grandparent's members is inherited by the grandchild.

Example using Virtual Inheritance to solve the Diamond Problem:

C++

```
#include <iostream>

class Grandparent {
public:
    Grandparent() { std::cout << "Grandparent constructor." << std::endl; }
    void display() const {
        std::cout << "Grandparent's display." << std::endl;
    }
};

class Parent1 : virtual public Grandparent {
public:
    Parent1() { std::cout << "Parent1 constructor." << std::endl; }
    void show1() const {
        std::cout << "Parent1's show." << std::endl;
    }
};

class Parent2 : virtual public Grandparent {
public:
    Parent2() { std::cout << "Parent2 constructor." << std::endl; }
    void show2() const {
        std::cout << "Parent2's show." << std::endl;
    }
};

class Child : public Parent1, public Parent2 {
public:
    Child() { std::cout << "Child constructor." << std::endl; }
    void showChild() const {
        std::cout << "Child's show." << std::endl;
    }
};
```

```
int main() {
    Child baby;
    baby.show1();
    baby.show2();
    baby.showChild();
    baby.display(); // Only one copy of Grandparent's members
    return 0;
}
```

By using `virtual public Grandparent` in the definitions of `Parent1` and `Parent2`, the `Child` class inherits only a single instance of `Grandparent`, resolving the ambiguity.

4.6.3 Making Private Members Inheritable (Indirectly)

In C++, private members of a base class are **always inherited** by derived classes, but they are **not directly accessible** within the derived class. This is a fundamental aspect of encapsulation – private members are meant to be the internal implementation details of the base class and should not be directly manipulated by derived classes.

However, there are indirect ways in which derived classes can interact with or be influenced by the private members of their base class:

1. Through Public and Protected Member Functions of the Base Class:

- The base class can provide public or protected member functions that access or manipulate its private data members.
- Derived classes can call these base class functions to indirectly work with the private data.

Example:

C++

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

class Base {
private:
    std::string secretInfo;

protected:
    Base(std::string secret) : secretInfo(secret) {}

public:
    std::string getSecret() const {
        return secretInfo;
    }

    void revealPartialSecret(int length) const {
        if (length <= secretInfo.length()) {
            std::cout << "Partial Secret: " << secretInfo.substr(0, length) <<
std::endl;
        } else {
            std::cout << "Requested length exceeds secret length." << std::endl;
        }
    }
};

class Derived : public Base {
public:
    Derived(std::string secret) : Base(secret) {}

    void accessSecretIndirectly() const {
        std::cout << "Derived class accessing secret: " << getSecret() << std::endl;
        revealPartialSecret(3);
```

```
    }
};

int main() {
    Derived d("TopSecret");
    d.accessSecretIndirectly();
    // std::cout << d.secretInfo; // Error: secretInfo is private in Base
    return 0;
}
```

In this example, `secretInfo` is private in `Base`. `Derived` cannot directly access it. However, `Derived` can use the public member functions `getSecret()` and `revealPartialSecret()` of `Base` to indirectly interact with `secretInfo`.

2. Through Protected Member Variables of the Base Class:

- If the base class declares certain data members as `protected` instead of `private`, these members are directly accessible within the derived classes.
- `protected` access provides a level of visibility between the base class and its derived classes, allowing for more direct manipulation while still restricting access from outside the inheritance hierarchy.

Example:

C++

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

class Base {
protected:
    std::string semiSecretInfo;

public:
    Base(std::string semiSecret) : semiSecretInfo(semiSecret) {}

    void displaySemiSecret() const {
        std::cout << "Base class semi-secret: " << semiSecretInfo << std::endl;
    }
};

class Derived : public Base {
public:
    Derived(std::string semiSecret, std::string extraInfo) : Base(semiSecret),
extraInfo_(extraInfo) {}

    void manipulateSemiSecret() {
        semiSecretInfo += " (Modified by Derived)";
    }

    void displayAllInfo() const {
        std::cout << "Semi-secret in Derived: " << semiSecretInfo << std::endl;
        std::cout << "Extra info in Derived: " << extraInfo_ << std::endl;
    }

private:
    std::string extraInfo_;
};

int main() {
    Derived d("Confidential", "Additional Details");
    d.displaySemiSecret();
    d.manipulateSemiSecret();
    d.displayAllInfo();
    return 0;
}
```

Here, `semiSecretInfo` is protected in `Base`, so `Derived` can directly access and modify it.

Why Private Members Remain Inaccessible Directly:

- **Encapsulation and Data Hiding:** The primary reason for making members private is to encapsulate the internal state of the class and hide its implementation details from the outside world, including derived classes. This allows the base class to control how its data is accessed and modified, ensuring data integrity and allowing for changes to the internal implementation without breaking derived classes (as long as the public and protected interfaces remain consistent).
- **Maintaining Invariants:** The base class might have certain invariants (rules about its data) that are maintained by its own member functions. Allowing direct access from derived classes could potentially violate these invariants.

Alternatives to Direct Access:

While derived classes cannot directly access private members, consider if they *should* need to. Often, the base class should provide the necessary interface (public or protected functions) for derived classes to interact appropriately. If a derived class needs very tight coupling with the base class's internal data, it might suggest a closer relationship (perhaps composition instead of inheritance, or a redesign of the base class's protected interface).

Friend Classes:

One way to grant a specific derived class (or any other class) direct access to the private and protected members of a base class is by declaring the derived class as a `friend` within the base class.

Example using `friend` class:

C++
```
#include <iostream>
#include <string>

class Base {
private:
    std::string verySecret;

public:
    Base(std::string secret) : verySecret(secret) {}

    friend class FriendDerived; // Granting FriendDerived access

    void revealSecretToFriend() const {
        std::cout << "Base revealing secret to friend." << std::endl;
    }
};

class FriendDerived : public Base {
public:
    FriendDerived(std::string secret, std::string identifier) : Base(secret),
id(identifier) {}

    void accessSecretDirectly() const {
        std::cout << "FriendDerived accessing secret directly: " << verySecret <<
std::endl;
    }

private:
    std::string id;
};

int main() {
    FriendDerived fd("SuperDuperSecret", "FD123");
    fd.accessSecretDirectly();
    fd.revealSecretToFriend(); // Can still access public members
    return 0;
}
```

In this case, `FriendDerived` is declared as a friend of `Base`, so it can directly access the private member `verySecret`. However, overuse of `friend` can weaken encapsulation and should be used judiciously when a very tight coupling is genuinely necessary.

4.6.4 Constructors in Derived Classes

When you create an object of a derived class, the constructors of both the base class(es) and the derived class are called. The order of execution is crucial for proper initialization of the object.

Order of Constructor Execution:

1. **Base Class Constructor(s):** If the derived class directly inherits from one or more base classes, their constructors are called first. In the case of single inheritance, the base class constructor is called. In multiple inheritance, the base class constructors are called in the order they appear in the derived class's inheritance list.
2. **Member Object Constructors:** If the derived class has member objects (objects of other classes as members), their constructors are called after the base class constructor(s) and before the derived class's own constructor. The order of member object constructor calls is determined by the order of their declaration in the derived class.
3. **Derived Class Constructor:** Finally, the constructor of the derived class itself is executed. This constructor typically initializes the data members that are specific to the derived class and may perform any other necessary setup.

Why this Order?

This order ensures that the base class parts of the derived object are properly initialized before the derived class's own initialization takes place. The derived class might rely on the base class's members being in a valid state. Similarly, member objects need to be initialized before the derived class can use them.

Calling Base Class Constructors:

- **Implicit Call (Default Constructor):** If the derived class's constructor does not explicitly call a base class constructor, the default (no-argument) constructor of the base class is called automatically before the derived class's constructor executes.
- **Explicit Call (Parameterized Constructor):** If the base class has a constructor that requires arguments (i.e., no default constructor exists or you want to call a specific parameterized constructor), the derived class's constructor **must explicitly** call the desired base class constructor in its **initializer list**.

Syntax for Explicitly Calling a Base Class Constructor:

C++
```
DerivedClass::DerivedClass(arguments for derived constructor) : BaseClass(arguments for base constructor), memberObject1(arguments), memberObject2(arguments) {
    // Initialization of derived class members
    // Other derived class constructor code
}
```

- The base class constructor call `BaseClass(arguments for base constructor)` appears after the colon (`:`) and before the opening brace `{` of the derived class constructor's body.
- If there are multiple base classes (in multiple inheritance), their constructors are called in the order they are listed after the colon, separated by commas.
- Member object constructor calls also appear in the initializer list.

Examples:

1. Single Inheritance with Default Base Class Constructor:

C++
```
#include <iostream>

class Base {
public:
    Base() {
        std::cout << "Base default constructor called." << std::endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        std::cout << "Derived constructor called." << std::endl;
    }
};

int main() {
    Derived d; // Calls Base() then Derived()
    return 0;
}
```

2. Single Inheritance with Parameterized Base Class Constructor:

C++
```
#include <iostream>
#include <string>

class Base {
    std::string name;
public:
    Base(std::string n) : name(n) {
        std::cout << "Base parameterized constructor called for " << name << std::endl;
    }

    std::string getName() const { return name; }
};

class Derived : public Base {
    int id;
public:
    Derived(std::string n, int i) : Base(n), id(i) {
        std::cout << "Derived constructor called for ID " << id << " (Name: " <<
getName() << ")" << std::endl;
    }
};

int main() {
    Derived d("Example", 123); // Calls Base("Example") then Derived("Example", 123)
    return 0;
}
```

3. Multiple Inheritance:

C++
```
#include <iostream>

class Base1 {
public:
    Base1() {
        std::cout << "Base1 constructor called." << std::endl;
    }
};
```

```
class Base2 {
public:
    Base2(int x) {
        std::cout << "Base2 constructor called with x = " << x << std::endl;
    }
};

class Derived : public Base1, public Base2 {
public:
    Derived() : Base2(10) { // Base1's default constructor is called implicitly
        std::cout << "Derived constructor called." << std::endl;
    }
};

int main() {
    Derived d; // Calls Base1(), Base2(10), then Derived()
    return 0;
}
```

4. Inheritance with Member Objects:

C++

```
#include <iostream>

class Member {
public:
    Member(int val) : value(val) {
        std::cout << "Member constructor called with value " << value << std::endl;
    }
private:
    int value;
};

class Base {
public:
    Base() {
        std::cout << "Base constructor called." << std::endl;
    }
};

class Derived : public Base {
    Member m1;
    Member m2;
public:
    Derived() : m2(20), m1(10) { // Member constructors in order of declaration
        std::cout << "Derived constructor called." << std::endl;
    }
};

int main() {
    Derived d; // Calls Base(), Member(10), Member(20), then Derived()
    return 0;
}
```

Important Considerations:

- **Base Class Must Have an Accessible Constructor:** If the base class only has private constructors, a derived class cannot inherit from it unless the derived class is a friend of the base class.
- **Initializer List is Mandatory for Parameterized Base Class Constructors:** If the base class requires arguments in its constructor, the derived class's constructor *must* provide these arguments by explicitly calling the base class constructor in its initializer list. Failing to do so will result in a compilation error.
- **Order of Initialization:** The order in the initializer list does not determine the order of constructor calls. Base class constructors are always called first (in the order of inheritance), then member object constructors (in the order of declaration), and finally the derived class constructor. The initializer list initializes the members in the order they are declared in the class.

- **Virtual Base Classes:** In the case of virtual inheritance (to solve the diamond problem), the constructors of virtual base classes are called before any non-virtual base class constructors and before the constructor of the most derived class. The most derived class is responsible for initializing the virtual base class.

Unit 5.0: Polymorphism and Pointers

5.1 Introduction to Polymorphism

Polymorphism, derived from the Greek words "poly" (many) and "morph" (form), is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to respond to the same method call in their own specific way. In simpler terms, it's the ability of an object to take on many forms.

Polymorphism enables you to write more flexible and extensible code by treating objects of different classes uniformly through a common interface (usually a base class pointer or reference). The actual behavior executed at runtime depends on the actual type of the object being pointed to or referenced.

Key Benefits of Polymorphism:

- **Code Reusability:** You can write generic code that works with objects of a base class, and this code will automatically work with any derived class objects.
- **Extensibility:** Adding new derived classes doesn't require modifying existing code that uses the base class interface. The new classes will seamlessly integrate.
- **Flexibility:** You can easily switch between different implementations of a behavior at runtime by simply changing the type of object being referenced.
- **Abstraction:** Polymorphism allows you to focus on the common interface of objects, hiding the specific implementation details of each class.

How Polymorphism is Achieved in C++:

C++ primarily achieves polymorphism through:

1. **Function Overloading (Compile-time Polymorphism):** This allows multiple functions in the same scope to have the same name but different parameter lists (number, types, or order of parameters). The compiler resolves which function to call at compile time based on the arguments provided in the function call.
2. **Operator Overloading (Compile-time Polymorphism):** Similar to function overloading, operator overloading allows you to redefine the behavior of built-in operators (like +, -, ==, etc.) for user-defined types. The compiler determines which operator overload to use based on the types of the operands.
3. **Virtual Functions (Runtime Polymorphism):** This is the most powerful form of polymorphism in the context of inheritance. Virtual functions allow a derived class to override a function in its base class, and the correct overridden function to be called at runtime based on the actual type of the object being pointed to or referenced (even if accessed through a base class pointer or reference).

Types of Polymorphism:

Based on when the binding between the function call and the function definition occurs, polymorphism in C++ can be broadly categorized into two main types:

1. Compile-time Polymorphism (Static Polymorphism or Early Binding):

- The decision of which function to execute is made by the compiler at compile time.
- This is achieved through function overloading and operator overloading.
- The compiler knows the exact type of the object or the arguments at the point of the function call or operator use, so it can determine the correct function or operator overload to invoke.
- It leads to faster execution because the function call is resolved before the program runs. However, it offers less flexibility at runtime.

Examples of Compile-time Polymorphism:

a) Function Overloading:

C++
```
#include <iostream>

class Calculator {
public:
    int add(int a, int b) {
        std::cout << "Adding two integers." << std::endl;
        return a + b;
    }

    double add(double a, double b) {
        std::cout << "Adding two doubles." << std::endl;
        return a + b;
    }

    int add(int a, int b, int c) {
        std::cout << "Adding three integers." << std::endl;
        return a + b + c;
    }
};

int main() {
    Calculator calc;
    std::cout << "Sum (int, int): " << calc.add(5, 3) << std::endl;
    std::cout << "Sum (double, double): " << calc.add(2.5, 1.7) << std::endl;
    std::cout << "Sum (int, int, int): " << calc.add(1, 2, 3) << std::endl;
    return 0;
}
```

In this example, the `add` function is overloaded. The compiler determines which `add` function to call based on the types and number of arguments passed.

b) Operator Overloading:

C++
```
#include <iostream>

class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag + other.imag);
    }

    void display() const {
        std::cout << real << " + " << imag << "i" << std::endl;
    }
};

int main() {
    Complex c1(1.0, 2.0);
    Complex c2(3.0, 4.0);
    Complex sum = c1 + c2; // Calls the overloaded + operator
    std::cout << "Sum: ";
    sum.display();
    return 0;
}
```

Here, the `+` operator is overloaded for the `Complex` class. The compiler chooses the `Complex` addition based on the operands being `Complex` objects.

2. Runtime Polymorphism (Dynamic Polymorphism or Late Binding):

- The decision of which function to execute is made at runtime based on the actual type of the object being pointed to or referenced.
- This is achieved through **virtual functions** and **pointers or references to base class types**.
- The compiler doesn't know the exact type of the object at compile time (it only knows the type of the pointer or reference). The runtime system determines the actual object type and calls the appropriate overridden function.
- It provides more flexibility and allows for more dynamic behavior but might have a slight performance overhead compared to compile-time polymorphism due to the runtime type resolution.

Example of Runtime Polymorphism (using Virtual Functions):

C++
```
#include <iostream>

class Shape {
public:
    virtual void draw() const {
        std::cout << "Drawing a generic shape." << std::endl;
    }
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a rectangle." << std::endl;
    }
};

void drawShape(const Shape& s) {
    s.draw(); // The actual draw() called depends on the runtime type of s
}

int main() {
    Shape* shapePtr;
    Circle circleObj;
    Rectangle rectObj;

    shapePtr = &circleObj;
    shapePtr->draw(); // Calls Circle::draw() at runtime

    shapePtr = &rectObj;
    shapePtr->draw(); // Calls Rectangle::draw() at runtime

    drawShape(circleObj); // Calls Circle::draw() at runtime (through reference)
    drawShape(rectObj);  // Calls Rectangle::draw() at runtime (through reference)

    return 0;
}
```

In this example:

- The `draw()` function in the `Shape` base class is declared as `virtual`.
- The `Circle` and `Rectangle` derived classes override the `draw()` function.

- When `draw()` is called through a `Shape` pointer or reference, the actual `draw()` function of the object being pointed to (either a `Circle` or a `Rectangle`) is executed at runtime. This is runtime polymorphism in action.

5.2 Pointers in C++

Pointers are a fundamental feature of C++ that allow you to work directly with memory addresses. A pointer is a variable that holds the memory address of another variable. They are crucial for dynamic memory allocation, passing arguments by reference, and, as we'll see, for achieving runtime polymorphism.

Key Concepts of Pointers:

- **Address-of Operator (`&`):** The `&` (ampersand) operator, when placed before a variable name, returns the memory address of that variable.
- **Dereference Operator (`*`):** The `*` (asterisk) operator, when placed before a pointer variable, accesses the value stored at the memory address held by the pointer. This is called dereferencing the pointer.
- **Pointer Declaration:** You declare a pointer variable by using an asterisk (`*`) after the data type of the variable it will point to. The pointer's type must match the data type of the variable whose address it will store (or be a `void` pointer, which can hold the address of any data type).
- **Pointer Initialization:** Pointers should be initialized before use. You can initialize a pointer with the address of an existing variable using the `&` operator, with `nullptr` (or `NULL` in older C++), or with the address of dynamically allocated memory.
- **Pointer Arithmetic:** You can perform certain arithmetic operations on pointers (increment, decrement, addition, subtraction). Pointer arithmetic is scaled by the size of the data type the pointer points to.
- **`void` Pointers:** A `void` pointer can hold the address of any data type. However, you cannot directly dereference a `void` pointer; you must first cast it to a specific pointer type.
- **Null Pointers:** A null pointer (represented by `nullptr` in modern C++) is a pointer that does not point to any valid memory location. It's used to indicate that a pointer is currently not pointing to anything.

Basic Pointer Operations:

C++

```
#include <iostream>

int main() {
    int number = 10;
    int* ptr; // Declare an integer pointer

    ptr = &number; // Assign the address of 'number' to 'ptr'

    std::cout << "Value of number: " << number << std::endl;
    std::cout << "Address of number: " << &number << std::endl;
    std::cout << "Value of ptr (address of number): " << ptr << std::endl;
    std::cout << "Address of ptr: " << &ptr << std::endl;
    std::cout << "Value pointed to by ptr (*ptr): " << *ptr << std::endl;

    *ptr = 20; // Modify the value at the address pointed to by ptr (which is 'number')
    std::cout << "Value of number after modification: " << number << std::endl;

    return 0;
}
```

5.2.1 Pointer to Objects

You can also declare pointers that hold the memory addresses of objects (instances of classes). Pointers to objects are essential for dynamic object creation and, crucially, for achieving runtime polymorphism through base class pointers pointing to derived class objects.

Declaring a Pointer to an Object:

The syntax for declaring a pointer to an object is similar to declaring a pointer to a built-in data type:

C++
```
ClassName* pointerName;
```

Here, `ClassName` is the type of the object the pointer will point to, `*` indicates that `pointerName` is a pointer, and `pointerName` is the name of the pointer variable.

Assigning the Address of an Object to a Pointer:

You use the address-of operator (`&`) to get the memory address of an object and assign it to a pointer of the appropriate class type:

C++
```
ClassName objectName;
ClassName* pointerName = &objectName;
```

Accessing Members of an Object Through a Pointer:

To access the members (data members and member functions) of an object through a pointer, you use the **arrow operator (`->`)**. This operator combines dereferencing the pointer and accessing a member in a single step.

Alternatively, you can first dereference the pointer using the `*` operator (which gives you the object itself) and then use the dot operator (`.`) to access the member, but you need to enclose the dereferenced pointer in parentheses due to operator precedence: `(*pointerName).memberName`. The arrow operator (`->`) is generally preferred for its clarity and conciseness.

Examples of Pointers to Objects:

C++
```
#include <iostream>
#include <string>

class Dog {
public:
    Dog(std::string n, int a) : name(n), age(a) {
        std::cout << "Dog " << name << " created." << std::endl;
    }

    void bark() const {
        std::cout << name << " says Woof!" << std::endl;
    }

    void displayAge() const {
        std::cout << name << " is " << age << " years old." << std::endl;
    }

private:
    std::string name;
    int age;
};

int main() {
    Dog myDog("Buddy", 3);
    Dog* dogPtr = &myDog; // Pointer to a Dog object

    std::cout << "Address of myDog: " << &myDog << std::endl;
    std::cout << "Value of dogPtr: " << dogPtr << std::endl;

    // Accessing members using the arrow operator
```

```
    dogPtr->bark();
    dogPtr->displayAge();

    // Accessing members using dereference and dot operator (less common)
    (*dogPtr).bark();
    (*dogPtr).displayAge();

    return 0;
}
```

Pointers to Base Class Objects:

You can have a pointer to a base class object. This is straightforward:

C++
```
class Animal {
public:
    virtual void makeSound() const {
        std::cout << "Animal makes a sound." << std::endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() const override {
        std::cout << "Cat meows." << std::endl;
    }
};

int main() {
    Animal genericAnimal;
    Animal* animalPtr = &genericAnimal;
    animalPtr->makeSound(); // Calls Animal::makeSound()

    Cat myCat;
    Animal* animalPtrToCat = &myCat; // Base class pointer pointing to a derived class
object
    animalPtrToCat->makeSound(); // Calls Animal::makeSound() (without virtual)
    return 0;
}
```

Pointers to Derived Class Objects Through Base Class Pointers (Crucial for Runtime Polymorphism):

The power of pointers to objects becomes particularly evident in the context of inheritance and polymorphism. You can have a pointer of the base class type that points to an object of a derived class. This is allowed because a derived class object *is a* base class object (due to the "is-a" relationship established by public inheritance).

However, there's a crucial point regarding which version of a virtual function gets called in this scenario. If the base class function is **not** declared as `virtual`, then calling that function through a base class pointer will always invoke the base class's version of the function, regardless of the actual type of the object being pointed to.

If the base class function **is** declared as `virtual`, then when you call that function through a base class pointer that points to a derived class object, the **runtime type** of the object is examined, and the **derived class's overridden version** of the function is executed. This is the essence of runtime polymorphism in C++.

Example Demonstrating Runtime Polymorphism with Base Class Pointers:

C++
```
#include <iostream>

class Shape {
public:
```

```
    virtual void draw() const {
        std::cout << "Drawing a generic shape." << std::endl;
    }
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a rectangle." << std::endl;
    }
};

int main() {
    Shape* shapePtr;
    Circle circleObj;
    Rectangle rectObj;

    shapePtr = &circleObj;
    shapePtr->draw(); // Calls Circle::draw() because draw() is virtual

    shapePtr = &rectObj;
    shapePtr->draw(); // Calls Rectangle::draw() because draw() is virtual

    return 0;
}
```

In this example, because `Shape::draw()` is virtual, the correct `draw()` function (either `Circle::draw()` or `Rectangle::draw()`) is called at runtime based on the object that `shapePtr` is currently pointing to.

5.2.2 The `this` Pointer and Pointer to Derived Classes

The `this` Pointer:

- Within a non-static member function of a class, `this` is a special pointer that holds the memory address of the current object on which the function is being called.
- It is implicitly passed as a hidden argument to every non-static member function.
- The type of the `this` pointer is `ClassName* const` (a constant pointer to an object of `ClassName`), meaning you can modify the object that `this` points to, but you cannot make `this` point to a different object. If the member function is declared `const`, then the type of `this` becomes `const ClassName* const` (a constant pointer to a constant object), meaning you cannot modify the object through `this` either.

Uses of the `this` Pointer:

1. **Distinguishing Member Variables from Local Variables:** When a local variable or a function parameter has the same name as a member variable, you can use `this->` to explicitly refer to the member variable.

   C++

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

   class MyClass {
   private:
   ```

```
    std::string name;

public:
    MyClass(std::string name) {
        // Using this-> to differentiate
        this->name = name;
        std::cout << "Object created with name: " << this->name << std::endl;
    }

    void printName(std::string name) {
        std::cout << "Local name: " << name << std::endl;
        std::cout << "Member name (this->name): " << this->name << std::endl;
    }
};

int main() {
    MyClass obj("My Object");
    obj.printName("Function Parameter");
    return 0;
}
```

2. **Returning the Current Object:** Member functions can return a pointer or a reference to the current object using `*this` (for the object itself) or `this` (for the pointer). This is often used in operator overloading (especially assignment operators) and in method chaining.

   C++

```
#include <iostream>

class Counter {
private:
    int count;

public:
    Counter(int c = 0) : count(c) {}

    Counter& increment() {
        count++;
        return *this; // Return a reference to the current object
    }

    void display() const {
        std::cout << "Count: " << count << std::endl;
    }
};

int main() {
    Counter c1(5);
    c1.increment().increment().display(); // Method chaining
    return 0;
}
```

3. **Passing the Current Object as an Argument:** A member function might need to pass the current object to another function. `this` can be used for this purpose.

   C++

```
#include <iostream>

class Processor {
public:
    void processData(int value) {
        std::cout << "Processing: " << value << std::endl;
    }
};

class DataObject {
```

```
private:
    int data;

public:
    DataObject(int d) : data(d) {}

    void process(Processor* p) {
        p->processData(this->data); // Passing data member
    }

    DataObject* getDataObjectPtr() {
        return this; // Returning a pointer to the current object
    }
};

int main() {
    Processor proc;
    DataObject obj(100);
    obj.process(&proc);

    DataObject* ptr = obj.getDataObjectPtr();
    std::cout << "Address of obj: " << &obj << std::endl;
    std::cout << "Pointer to obj: " << ptr << std::endl;

    return 0;
}
```

Pointer to Derived Classes:

- Just as you can have a pointer to a base class, you can also have a pointer to a derived class.
- A pointer of a derived class type can hold the address of an object of that derived class.

C++

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

class Animal {
public:
    virtual void makeSound() const {
        std::cout << "Animal sound." << std::endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() const override {
        std::cout << "Woof!" << std::endl;
    }

    void fetch() const {
        std::cout << "Dog is fetching." << std::endl;
    }
};

int main() {
    Dog myDog;
    Dog* dogPtr = &myDog; // Pointer to a Dog object

    dogPtr->makeSound(); // Calls Dog::makeSound()
    dogPtr->fetch();     // Calls Dog::fetch()

    return 0;
}
```

Base Class Pointer Pointing to a Derived Class Object (Upcasting):

- A pointer of a base class type can point to an object of a publicly derived class. This is called **upcasting**. It's safe because a derived class object *is a* base class object.
- However, when you access members through a base class pointer that points to a derived class object:
    - You can only directly access the members that are inherited from the base class (and are accessible according to the inheritance access specifier).
    - If you call a **non-virtual** function that is also defined in the derived class, the **base class's version** of the function will be executed (static binding).
    - If you call a **virtual** function that is overridden in the derived class, the **derived class's version** of the function will be executed (dynamic binding/runtime polymorphism).
    - You cannot directly access members that are specific to the derived class through a base class pointer without explicit casting (which should be done with caution).

Derived Class Pointer Pointing to a Base Class Object (Downcasting):

- Assigning the address of a base class object to a pointer of a derived class type without an explicit cast is generally **not allowed** and will result in a compile-time error. This is because a base class object is not necessarily a derived class object.
- **Downcasting** (converting a base class pointer to a derived class pointer) can be done using explicit type casting (`static_cast` or `dynamic_cast`). However, it is **unsafe** if the base class pointer does not actually point to an object of the derived class.
- `dynamic_cast` is safer for downcasting in polymorphic hierarchies (where the base class has at least one virtual function). It performs a runtime check and returns `nullptr` if the cast is not valid. `static_cast` does not perform this runtime check and should only be used when you are certain of the object's type.

Example Demonstrating Upcasting and Downcasting:

C++
```
#include <iostream>

class Base {
public:
    virtual void show() const {
        std::cout << "Base show." << std::endl;
    }
};

class Derived : public Base {
public:
    void show() const override {
        std::cout << "Derived show." << std::endl;
    }

    void derivedSpecific() const {
        std::cout << "Derived specific function." << std::endl;
    }
};

int main() {
    Derived dObj;
    Base* bPtr = &dObj; // Upcasting (safe)

    bPtr->show(); // Calls Derived::show() because show() is virtual
    // bPtr->derivedSpecific(); // Error: Base pointer cannot directly access derived-
specific members

    // Downcasting (potentially unsafe)
    Derived* dPtr1 = static_cast<Derived*>(bPtr); // Assuming bPtr actually points to a
Derived object
    dPtr1->derivedSpecific(); // Now it's accessible
```

```
    Base bObj;
    Base* bPtr2 = &bObj;

    // Derived* dPtr2 = static_cast<Derived*>(bPtr2); // Might cause runtime issues if
used incorrectly
    // dPtr2->derivedSpecific();

    Derived* dPtr3 = dynamic_cast<Derived*>(bPtr2); // Safe downcasting (returns nullptr
if invalid)
    if (dPtr3 != nullptr) {
        dPtr3->derivedSpecific();
    } else {
        std::cout << "Invalid downcast." << std::endl;
    }

    return 0;
}
```

5.3 Virtual Function, Pure Virtual Function, Virtual Constructor and Destructor

Virtual functions are a cornerstone of runtime polymorphism in C++. They enable derived classes to provide their own implementations of functions declared in the base class, and the correct version is called at runtime based on the actual type of the object being pointed to or referenced.

5.3.1 Virtual Functions:

- A virtual function is a member function declared in the base class using the keyword `virtual`.
- When a virtual function is called through a base class pointer or reference that points to a derived class object, the runtime system determines the actual type of the object and invokes the overridden function in the derived class. This is known as **dynamic binding** or **late binding**.
- If a base class declares a function as virtual, any derived class that overrides this function (with the same signature: name, parameter types, and const-ness) will have its version called when accessed through a base class pointer or reference.
- If a derived class does not override a virtual function, the base class's implementation is used.

Syntax:

C++

```
class Base {
public:
    virtual void someFunction() {
        // Base class implementation
    }
};

class Derived : public Base {
public:
    void someFunction() override {
        // Derived class implementation (override keyword is optional but recommended in
C++11 and later)
    }
};

int main() {
    Base* basePtr;
    Base baseObj;
    Derived derivedObj;

    basePtr = &baseObj;
    basePtr->someFunction(); // Calls Base::someFunction()

    basePtr = &derivedObj;
```

```
    basePtr->someFunction(); // Calls Derived::someFunction() due to virtual keyword

    return 0;
}
```

Key Points about Virtual Functions:

- The `virtual` keyword is only needed in the base class declaration. Derived classes that override the function are implicitly virtual.
- The return type of the overriding function in the derived class must be the same as the base class's virtual function, with a few exceptions related to covariant return types (where a derived class can return a pointer or reference to its own type if the base class returns a pointer or reference to the base class).
- Virtual functions enable polymorphism, allowing you to treat objects of different classes uniformly through a base class interface.

5.3.2 Pure Virtual Functions:

- A pure virtual function is a virtual function that has no definition within the base class. It is declared by assigning `0` to the virtual function in the base class declaration.
- A class that contains at least one pure virtual function is called an **abstract class**.
- You cannot create objects of an abstract class. Abstract classes serve as interfaces, defining a contract that derived classes must fulfill by providing concrete implementations for all pure virtual functions.
- Derived classes that inherit from an abstract class must override all pure virtual functions to become concrete (non-abstract) classes, from which objects can be created.

Syntax:

C++
```
class AbstractBase {
public:
    virtual void mustImplement() = 0; // Pure virtual function
    virtual void mayImplement() {
        // Optional virtual function with a default implementation
    }
};

class ConcreteDerived : public AbstractBase {
public:
    void mustImplement() override {
        // Concrete implementation
    }
    void mayImplement() override {
        // Optional override
    }
};

int main() {
    // AbstractBase absObj; // Error: Cannot create objects of an abstract class
    ConcreteDerived concreteObj; // OK: ConcreteDerived is not abstract
    AbstractBase* basePtr = &concreteObj;
    basePtr->mustImplement(); // Calls ConcreteDerived::mustImplement()
    return 0;
}
```

Purpose of Pure Virtual Functions and Abstract Classes:

- To define interfaces that all derived classes must adhere to.
- To enforce a certain structure and behavior in the inheritance hierarchy.
- To provide a base class without committing to a specific implementation for certain methods.

5.3.3 Virtual Destructor:

- When you delete a derived class object through a base class pointer, if the base class destructor is not virtual, only the base class's destructor is called. This can lead to **resource leaks** if the derived class has allocated memory or holds other resources that are cleaned up in its destructor.
- Declaring the base class destructor as `virtual` ensures that when a derived class object is deleted through a base class pointer, the derived class's destructor is called first, followed by the base class's destructor. This ensures proper cleanup of resources allocated by both the base and derived classes.

Syntax:

C++
```
class Base {
public:
    virtual ~Base() {
        std::cout << "Base destructor called." << std::endl;
    }
};

class Derived : public Base {
private:
    int* data;

public:
    Derived() {
        data = new int[10];
        std::cout << "Derived constructor called, allocated memory." << std::endl;
    }

    ~Derived() override {
        delete[] data;
        std::cout << "Derived destructor called, freed memory." << std::endl;
    }
};

int main() {
    Base* basePtr = new Derived();
    delete basePtr; // Calls Derived destructor then Base destructor (if ~Base() is
virtual)
    return 0;
}
```

Importance of Virtual Destructors:

- Essential for proper resource management in inheritance hierarchies where derived class objects might be deleted through base class pointers.
- Prevents memory leaks and ensures that destructors of all relevant classes in the inheritance chain are executed.
- **Rule of Thumb:** If a class has virtual functions, it should also have a virtual destructor. This is because the presence of virtual functions suggests that the class is intended to be used as a base class in a polymorphic hierarchy.

5.3.4 Virtual Constructor:

- C++ **does not directly support virtual constructors** in the same way it supports virtual functions and destructors. Constructors are responsible for creating an object of a specific type, and the type to be constructed is usually known at compile time.
- However, you can achieve similar behavior using design patterns like the **Factory Pattern** or the **Prototype Pattern**. These patterns involve a separate function or class that is responsible for creating objects of the appropriate derived type based on some criteria (e.g., a type identifier). This function can then return a pointer to the base class, effectively providing a form of "virtual construction."

Example using a Simple Factory Function:

C++

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

class Shape {
public:
    virtual ~Shape() {}
    virtual void draw() const = 0;
    static Shape* createShape(const std::string& type);
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a rectangle." << std::endl;
    }
};

Shape* Shape::createShape(const std::string& type) {
    if (type == "circle") {
        return new Circle();
    } else if (type == "rectangle") {
        return new Rectangle();
    } else {
        return nullptr;
    }
}

int main() {
    Shape* circlePtr = Shape::createShape("circle");
    if (circlePtr) {
        circlePtr->draw();
        delete circlePtr;
    }

    Shape* rectPtr = Shape::createShape("rectangle");
    if (rectPtr) {
        rectPtr->draw();
        delete rectPtr;
    }

    Shape* invalidPtr = Shape::createShape("triangle");
    if (!invalidPtr) {
        std::cout << "Unknown shape type." << std::endl;
    }

    return 0;
}
```

In this example, `createShape` acts as a virtual constructor by returning a pointer to the appropriate derived `Shape` object based on the input type.