```
    // Method to receive a message and process data
    int receiveMessage(int data) {
        cout << "Received data from sender: " << data << endl;
        // Processing data and returning result
        return data * 2;
    }
};

int main() {
    Sender sender;
    Receiver receiver;

    // Sender object sends a message to the receiver object
    sender.sendMessage(5, receiver);

    return 0;
}
```

2.2 Access Specifiers

Access specifiers in C++ are keywords used to define the accessibility of class members (attributes and methods) from outside the class. They play a crucial role in enforcing encapsulation, which is one of the fundamental principles of object-oriented programming (OOP). In C++, there are three main access specifiers: public, private, and protected.

1. Public Access Specifier:

- Members declared as public are accessible from outside the class through objects of that class.
- Public members form the interface of the class, providing access to its functionality.
- Public members can be accessed and modified freely from any part of the program.

Example:

```
class MyClass {
public:
    int publicVar;          // Public attribute
    void publicMethod();    // Public method
};
```

2. Private Access Specifier:

- Members declared as private are accessible only within the class itself.
- They cannot be accessed directly from outside the class or by derived classes.
- Private members are used to hide implementation details and enforce data encapsulation.

Example:

```
class MyClass {
```

```
private:
    int privateVar;          // Private attribute
    void privateMethod();    // Private method
};
```

3. Protected Access Specifier:

- Members declared as protected are accessible within the class itself and by derived classes.
- They are not accessible from outside the class hierarchy.
- Protected members enable inheritance and code reuse by allowing derived classes to access certain aspects of the base class's implementation.

Example:

```
class MyBaseClass {
protected:
    int protectedVar;        // Protected attribute
    void protectedMethod(); // Protected method
};
```

Access Specifier Usage:

- Access specifiers define the level of encapsulation and control the visibility of class members.
- Public members provide an external interface to the class, allowing users to interact with its functionality.
- Private members are hidden from external access, ensuring data integrity and preventing unauthorized modification.
- Protected members enable inheritance and allow derived classes to access base class functionality.

Example:

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

// Class declaration
class MyClass {
public:
    int publicVar;    // Public attribute
    void publicMethod() {
        cout << "Public method called" << endl;
    }

private:
    int privateVar;   // Private attribute
    void privateMethod() {
        cout << "Private method called" << endl;
    }
```

```
protected:
    int protectedVar; // Protected attribute
    void protectedMethod() {
        cout << "Protected method called" << endl;
    }

public:
    // Constructor to initialize attributes
    MyClass() {
        publicVar = 0;
        privateVar = 0;
        protectedVar = 0;
    }
};

int main() {
    // Create an object of MyClass
    MyClass obj;

    // Accessing public members
    obj.publicVar = 10;
    obj.publicMethod();

    // Accessing private members (will cause compilation error)
    // obj.privateVar = 20;
    // obj.privateMethod();

    // Accessing protected members (will cause compilation error)
    // obj.protectedVar = 30;
    // obj.protectedMethod();

    return 0;
}
```

2.3 Getter and Setter

Getter and setter methods, also known as accessor and mutator methods, are class member functions used to access and modify the private data members of a class. They play a crucial role in implementing encapsulation and data abstraction in object-oriented programming. Getter methods allow us to retrieve the values of private data members, while setter methods enable us to set or modify the values of private data members.

Getter Methods:

- Getter methods are used to retrieve the values of private data members of a class.
- They are defined as public member functions within the class.

- Getter methods provide controlled access to the private data members, ensuring encapsulation and information hiding.
- They typically do not modify the state of the object and are declared as const member functions to indicate that they do not alter the object's internal state.
- Getter methods typically have a return type corresponding to the type of the data member being accessed.
- They provide a read-only view of the data, allowing users to retrieve information without directly accessing the private data members.

```
class MyClass {
private:
    int privateVar;

public:
    // Getter method to retrieve the value of privateVar
    int getPrivateVar() const {
        return privateVar;
    }
};
```

Setter Methods:

- Setter methods are used to set or modify the values of private data members of a class.
- They are defined as public member functions within the class.
- Setter methods provide controlled access to modify the private data members, ensuring encapsulation and data integrity.
- They typically take parameters corresponding to the new values to be assigned to the private data members.
- Setter methods may include validation checks or additional logic to ensure the validity of the new values being assigned.
- They are responsible for maintaining the internal consistency of the object's state.

```
class MyClass {
private:
    int privateVar;

public:
    // Setter method to set the value of privateVar
    void setPrivateVar(int newValue) {
        privateVar = newValue;
    }
};
```

Usage:

- Getter and setter methods allow controlled access to private data members, preventing direct manipulation from outside the class.

- They provide an interface for interacting with the class's internal state, hiding implementation details and promoting abstraction.
- Getter and setter methods facilitate encapsulation by encapsulating the internal representation of data and exposing only essential functionality to the user.

```
MyClass obj;
obj.setPrivateVar(10);       // Using setter method to set privateVar
int value = obj.getPrivateVar(); // Using getter method to retrieve
privateVar
```

Benefits:

- Encapsulation: Getter and setter methods encapsulate the access to private data members, preventing direct modification and ensuring data integrity.
- Abstraction: They provide a simplified interface for interacting with class objects, hiding implementation details and promoting abstraction.
- Data Validation: Setter methods enable validation checks to ensure the validity of new values being assigned to private data members.
- Flexibility: Getter and setter methods provide flexibility to modify the internal representation of data without affecting external code that interacts with the class.

2.4 Static Members (Static Keyword)

The static keyword in C++ is used to define class-level members that are shared among all instances of the class. These class-level members can include static variables and static methods. Static members belong to the class itself rather than individual objects, and they are shared among all instances of the class.
The static keyword can also be used in local variables and functions, but here we'll focus on its usage with class members.

1. Static Variables (Static Data Members):

- Static variables are shared among all instances of the class, rather than each instance having its own copy.
- They are declared with the static keyword inside the class definition.
- Static variables are initialized once, before any object of the class is created, and they retain their values throughout the program's execution.
- They are typically used for maintaining class-wide data or for counting the number of objects created from the class.

Example:

```
class MyClass {
public:
    static int staticVar; // Declaration of static variable
};

int MyClass::staticVar = 0; // Initialization of static variable

// Usage
MyClass obj1;
```

```
MyClass obj2;
obj1.staticVar = 10; // Accessing static variable using object
cout << obj2.staticVar; // Output: 10
```

2. Static Methods (Static Member Functions):

- Static methods belong to the class rather than to any specific object.
- They can be called using the class name without needing an object.
- Static methods cannot access non-static members of the class directly, as they do not have access to any specific object's state.
- They are typically used for performing operations that do not depend on the state of any particular object.

Example:

```
class MyClass {
public:
    static void staticMethod() {
        cout << "Static method called" << endl;
    }
};

// Usage
MyClass::staticMethod(); // Calling static method using class name
```

Benefits of Static Members:

- Memory Efficiency: Static variables are shared among all instances of the class, reducing memory usage as compared to non-static variables that have a separate copy for each instance.
- Global Access: Static methods can be called using the class name without needing an instance of the class, allowing for global access to certain functionalities.
- Class-wide Operations: Static members are useful for operations that are common to all instances of the class, such as maintaining global state or performing utility functions.
- Initialization: Static variables are initialized once before any object of the class is created, providing predictable behavior and avoiding initialization issues.

Scope Resolution Operator:

- The scope resolution operator is used to access static members (static variables and static methods) of a class. Static members belong to the class itself rather than individual objects, and they are shared among all instances of the class.

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

class MyClass {
public:
    static int count; // Static variable declaration
    static void displayCount() { // Static method definition
```

```
        cout << "Count: " << count << endl;
    }
};

int MyClass::count = 0; // Static variable definition

int main() {
    MyClass::count = 5; // Accessing and modifying static variable
    MyClass::displayCount(); // Accessing static method
    return 0;
}
```

- The scope resolution operator can be used to access global variables and functions from within a local scope. This is particularly useful when there's a local variable or function with the same name as a global one.

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

int x = 10; // Global variable

int main() {
    int x = 20; // Local variable
    cout << "Local variable x: " << x << endl;        // Output: Local
variable x: 20
    cout << "Global variable x: " << ::x << endl;     // Output: Global
variable x: 10
    return 0;
}
```

2.5 Constant Members (Const Keyword)

The const keyword in C++ is used to declare constants or to specify that an object or function is immutable, meaning its value cannot be changed after initialization.
When applied to member variables and member functions of a class, const ensures that these members cannot be modified after initialization. This feature enhances code safety, readability, and maintainability.

1. Constant Data Members:

- When applied to member variables of a class, const makes them constant data members.
- Constant data members must be initialized during object creation and cannot be modified thereafter.
- They provide a way to enforce immutability and prevent unintentional modification of critical data within a class.

2. Constant Member Functions:

- When applied to member functions of a class, const indicates that these functions do not modify the state of the object.
- Constant member functions can be called on const and non-const objects, but they cannot modify non-mutable data members.
- They ensure that the object's state remains unchanged when invoking certain operations on const objects.

Example:

```
class Circle {
public:
    const double PI = 3.141592653589793; // Constant data member
    double getArea() const { // Constant member function
        return PI * radius * radius;
    }
private:
    double radius;
};

// Usage
const Circle circle; // Constant object declaration
double area = circle.getArea(); // Invoking constant member function
on const object
```

Benefits:

- Safety: Constant members help prevent accidental modification of critical data and ensure code safety.
- Readability: Clearly defining constant members and constant member functions enhances code readability and understandability.
- Maintainability: Immutable data members and non-modifying member functions simplify code maintenance by reducing the risk of unintended changes and side effects.

2.6 Friends of a Class

In C++, the concept of "friendship" allows a class to grant access to its private and protected members to another class or function. When a class or function is declared as a friend of another class, it can access the private and protected members of that class as if it were a member of that class itself. This feature is often used to allow certain external functions or classes to operate on private data members of a class without violating encapsulation.

1. Declaring Friend Functions:

- In C++, a friend function is a function that is not a member of a class but has the ability to access the class's private and protected members.

- It is declared within the class using the friend keyword.
- Friend functions are typically used when an external function needs to access the internal state of a class in a controlled manner.

```
class MyClass {
private:
    int privateVar;

public:
    friend void friendFunction(MyClass obj); // Declaration of friend function
};
```

2. Declaring Friend Classes:

- A friend class is declared using the friend keyword within the class definition.
- Friend classes have access to the private and protected members of the class to which they are declared as friends.
- They are granted access to the member functions and variables of the class as if they were part of the class itself.

```
class MyClass {
private:
    int privateVar;

public:
    friend class FriendClass; // Declaration of friend class
};
```

3. Accessing Private Members:

- Friend functions and friend classes can access private and protected members of the class to which they are declared as friends.
- They are allowed to read and modify the private members of the class without using member functions or access specifiers.

```
class MyClass {
private:
    int privateVar;

public:
    friend void friendFunction(MyClass obj) {
        obj.privateVar = 10; // Accessing private member
    }
};
```

Example:

Here's an example demonstrating the usage of friend function to access private members of a class:

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

```
class MyClass {
private:
    int privateVar;

public:
    MyClass(int value) {
        privateVar = value;
    }

    friend void friendFunction(MyClass obj);
};

void friendFunction(MyClass obj) {
    cout << "Value of privateVar: " << obj.privateVar << endl;
}

int main() {
    MyClass obj(5);
    friendFunction(obj); // Output: Value of privateVar: 5
    return 0;
}
```

Merits of Friend Functions:

1. Access to Private Members: Friend functions can access private and protected members of the class, which allows for greater flexibility in function implementation.
2. Separation of Concerns: Friend functions can be used to separate the functionality that requires access to the internals of a class without making those functions members of the class.
3. Operator Overloading: Friend functions are often used for operator overloading where the left-hand operand is not an object of the class.

Demerits of Friend Functions:

1. Breaks Encapsulation: Friend functions violate the encapsulation principle by allowing external functions to access private and protected members of a class.
2. Maintenance Complexity: With increased access to a class's internals, the use of friend functions can lead to more complex and harder-to-maintain code.
3. Tight Coupling: Friend functions can create tight coupling between the class and the external function, making changes to the class more impactful on external code.

2.7 Empty Classes

- An empty class is defined as a class that contains no data members or member functions, or it contains only inline or default-constructed members.

- Size: An empty class has a size of at least one byte in C++ to ensure that distinct objects of the class have unique addresses in memory.
- **Syntax:**

```
class EmptyClass {
    // No member variables or member functions
};
```

Example:

```
// Empty class declaration
class Marker {};

int main() {
    Marker marker; // Creating an object of the empty class
    return 0;
}
```

Purpose and Usage:

- Marker Classes: Empty classes are often used as marker classes to convey specific information about the type or purpose of objects or functions.
- Compile-Time Checks: They can be utilized to trigger certain behaviors or compile-time checks when used as template parameters or function arguments.
- Polymorphic Behavior: Empty classes can be used as base classes to enable polymorphic behavior through inheritance, allowing derived classes to override member functions.

2.8 Nested Classes

In C++, a class can be defined within another class, known as a nested class. Nested classes have access to the private members of the enclosing class and can be used to encapsulate related functionality within the scope of the enclosing class.

Example:

```
class Outer {
private:
    int privateVar;

public:
    // Nested class declaration
    class Inner {
    public:
        void innerMethod() {
            // Accessing private member of the outer class
            cout << "Accessing privateVar from inner class: " <<
privateVar << endl;
        }
```

```
    };
};
```

Purpose and Usage:

- Encapsulation: Nested classes help in encapsulating related functionality within the scope of the enclosing class, reducing namespace pollution and improving code organization.
- Access to Enclosing Class Members: Nested classes have access to the private members of the enclosing class, allowing them to manipulate the state of the enclosing object.
- Information Hiding: They can be used to hide implementation details and provide a cleaner interface to the users of the enclosing class.

2.9 Local Classes

Local classes in C++ are classes that are defined within the scope of a function or a block. Local classes have limited local scope and are accessible only within the function or block in which they are defined. Local classes can be used to encapsulate functionality that is only relevant within a specific scope.

Example:

```
void myFunction() {
    // Local class definition
    class LocalClass {
    public:
        void localMethod() {
            cout << "Inside local method" << endl;
        }
    };

    LocalClass obj;
    obj.localMethod(); // Accessing method of local class
}
```

Purpose and Usage:

- Limited Scope: Local classes have a limited scope and are only accessible within the block or function where they are defined, reducing namespace pollution.
- Encapsulation: They can encapsulate functionality that is only relevant within a specific scope, improving code organization and modularity.
- Information Hiding: Local classes can hide implementation details and provide a cleaner interface to the surrounding code.