

# C++ Inheritance: Comprehensive Notes Part II

## Table of Contents

|                                                                                                                                                                                                                                     |                                                                                |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| 1. <u>Introduction to Inheritance</u>                                                                                                                                                                                               | <b><u>C++ Inheritance Notes Part I</u></b><br><br><b><u>Shared Earlier</u></b> |
| 2. <u>Basic Inheritance Syntax</u>                                                                                                                                                                                                  |                                                                                |
| 3. <u>Access Specifiers in Inheritance</u>                                                                                                                                                                                          |                                                                                |
| 4. <u>Types of Inheritance</u> <ul style="list-style-type: none"><li>• Single Inheritance</li><li>• Multiple Inheritance</li><li>• Multilevel Inheritance</li><li>• Hierarchical Inheritance</li><li>• Hybrid Inheritance</li></ul> |                                                                                |
| 5. <u>Function Overriding</u>                                                                                                                                                                                                       |                                                                                |
| 6. <u>Constructor and Destructor in Inheritance</u>                                                                                                                                                                                 |                                                                                |

|                                                       |                                             |
|-------------------------------------------------------|---------------------------------------------|
| 7. <u>Virtual Functions</u>                           | <b><u>C++ Inheritance Notes Part II</u></b> |
| 8. <u>Abstract Classes and Pure Virtual Functions</u> |                                             |
| 9. <u>Virtual Inheritance and Diamond Problem</u>     |                                             |
| 10. <u>Object Slicing</u>                             |                                             |
| 11. <u>Protected Members</u>                          |                                             |
| 12. <u>Final Specifier</u>                            |                                             |
| 13. <u>Override Specifier</u>                         |                                             |
| 14. <u>Best Practices for C++ Inheritance</u>         |                                             |
| 15. <u>Real-world Example: Shape Hierarchy</u>        |                                             |
| 16. <u>Friend Functions in C++</u>                    |                                             |

# Virtual Functions

Virtual functions enable polymorphism, allowing a derived class's method to be called through a base class pointer or reference.

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

class Base {
public:
    virtual void display() {
        cout << "Display method of Base class." << endl;
    }
};

class Derived : public Base {
public:
    // Override the virtual function
    void display() override {
        cout << "Display method of Derived class." << endl;
    }
};

int main() {
    Base* basePtr;
    Derived derivedObj;

    basePtr = &derivedObj;

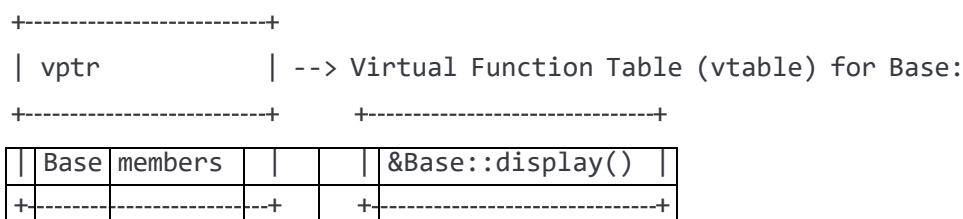
    // Without virtual, this would call Base::display()
    // With virtual, it calls Derived::display()
    basePtr->display(); // Display method of Derived class.

    return 0;
}
```

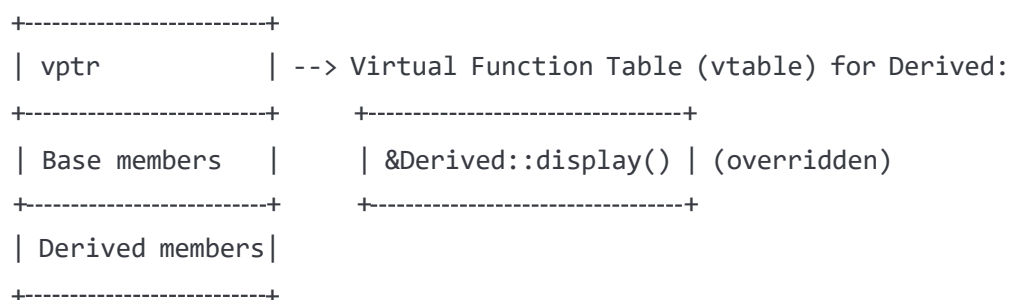
## Virtual Function Mechanism Diagram:

## Memory Layout of an Object with Virtual Functions

Base Class Object:



Derived Class Object:



## Rules for Virtual Functions

1. Virtual functions in C++ cannot be static.
2. A virtual function in C++ can also be a friend function of another class.
3. Virtual functions in C++ should be accessed using a base class pointer or base class reference variable to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as the derived class.
5. In C++, a class may have a virtual destructor but it cannot have a virtual constructor.

## Limitations of Virtual Functions:

1. **Slower:** The function call takes a slightly longer time due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at runtime.
2. **Difficult to Debug:** In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.

# Abstract Classes and Pure Virtual Functions

Abstract classes in C++ are special classes that cannot be instantiated directly and are designed to be used as base classes. They serve as templates or blueprints for derived classes, enforcing that certain methods must be implemented by any concrete (non-abstract) derived class.

## Key Characteristics :

- **Pure Virtual Functions:** An abstract class contains at least one pure virtual function, declared using the `= 0` syntax.
- **Cannot Be Instantiated:** You cannot create objects of an abstract class.
- **Inheritance:** Abstract classes are meant to be inherited by derived classes.
- **Implementation Requirement:** Derived classes must implement all pure virtual functions to become concrete classes.
- **Partial Implementation:** Abstract classes can provide implementation for some methods while leaving others as pure virtual.

## Implementation

To create an abstract class in C++:

1. Declare at least one member function as pure virtual by appending `= 0` to its declaration.
2. The derived class must override and implement all pure virtual functions.

```
class AbstractClass {
public:
    // Pure virtual function
    virtual void pureVirtualFunction() = 0;
    // Regular virtual function with implementation
    virtual void regularVirtualFunction() {
        // Default implementation
    }
    // Non-virtual function
    void normalFunction()
    { // Implementation
    };
};
```

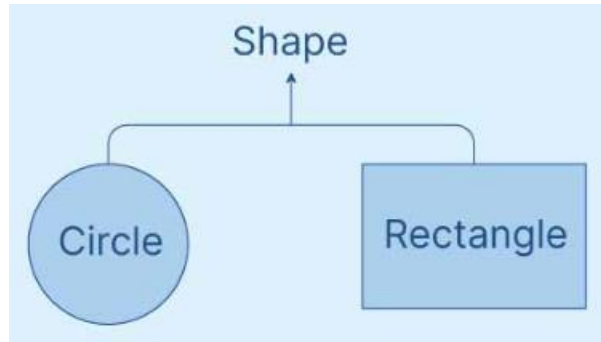
## Code Example 1

```
#include <iostream>
using namespace std;
// Abstract class
class Shape {
protected:
    string color;
public:
    // Pure virtual function
    virtual double area()= 0;
    // Normal virtual function
    virtual void display(){
        cout << "This is a shape with area: " << area() << endl;
    }
    // Non-virtual function
    void setColor(string color) {
        this->color = color;
    }
};

class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    // Implementation of pure virtual function
    double area() override { return
        3.14159 * radius * radius;
    }
    // Optional: override display
    void display() override {
        cout << "Circle with radius " << radius << " and area " << area() << endl;
    }
};

class Rectangle : public Shape {
private:
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}

    // Implementation of pure virtual function
    double area() override { return
        width * height;
    }
};
```



```

int main() {
    // Shape shape; // Error: Cannot instantiate abstract class

    Circle circle(5.0);
    Rectangle rectangle(4.0, 6.0);

    Shape* shapes[2];
    shapes[0] = &circle;
    shapes[1] = &rectangle;

    for (int i = 0; i < 2; i++) {
        shapes[i]->display(); // Polymorphic call
    }

    return 0;
}

```

```

● cd "c:\Users\shivanshi\Downloads\" ; if ($?) { g++ trial.cpp -o trial } ;
al }
Circle with radius 5 and area 78.5397
This is a shape with area: 24

```

## Code Example 2

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

// Abstract class
class Shape {
public:
    // Pure virtual function - makes this class abstract
    virtual double calculateArea() = 0;

    // Pure virtual function
    virtual double calculatePerimeter() = 0;

    // Regular virtual function with implementation
    virtual void displayProperties() {
        cout << "Area: " << calculateArea() << endl;
        cout << "Perimeter: " << calculatePerimeter() << endl;
    }

    // Regular function
    void setColor(string c) {
        color = c;
    }

    string getColor() {
        return color;
    }

protected:
    string color;
};

// Concrete derived class
class Rectangle : public Shape {
private:
    double length;
    double width;

public:
    Rectangle(double l, double w) : length(l), width(w) {
        color = "No color";
    }

    // Implementation of first pure virtual function
    double calculateArea() override {
        return length * width;
    }
}
```

```

// Implementation of second pure virtual function
double calculatePerimeter() override {
    return 2 * (length + width);
}

// Override of regular virtual function
void displayProperties() override {
    cout << "Rectangle Properties:" << endl;
    cout << "Length: " << length << endl;
    cout << "Width: " << width << endl;
    cout << "Color: " << getColor() << endl;
    Shape::displayProperties(); // Calling base class implementation
}
};

// Another concrete derived class
class Circle : public Shape {
private:
    double radius;
    const double PI = 3.14159;

public:
    Circle(double r) : radius(r) {
        color = "No color";
    }

// Implementation of first pure virtual function
double calculateArea() override {
    return PI * radius * radius;
}

// Implementation of second pure virtual function
double calculatePerimeter() override {
    return 2 * PI * radius;
}
};

int main() {
    // Shape s; // Error: Cannot instantiate an abstract class

    Rectangle rect(5.0, 3.0);
    rect.setColor("Blue");

    Circle circle(4.0);
    circle.setColor("Red");

```



```

cout << "=== Rectangle ===" << endl;
rect.displayProperties();

cout << "\n=== Circle ===" << endl;
circle.displayProperties();

// Using abstract class pointer
cout << "\n=== Using Abstract Pointers ===" << endl;
Shape* shapes[2];
shapes[0] = &rect;
shapes[1] = &circle;

for (int i = 0; i < 2; i++) {
    cout << "Shape " << i+1 << " - ";
    cout << "Area: " << shapes[i]->calculateArea() << ", ";
    cout << "Perimeter: " << shapes[i]->calculatePerimeter() << ", ";
    cout << "Color: " << shapes[i]->getColor() << endl;
}

return 0;
}

```

## Output:

```

=== Rectangle ===
Rectangle Properties:
Length: 5
Width: 3
Color:
Blue
Area: 15
Perimeter: 16

```

```

=== Circle
=== Area:
50.2654
Perimeter: 25.1327

```

```

=== Using Abstract Pointers ===
Shape 1 - Area: 15, Perimeter: 16, Color: Blue
Shape 2 - Area: 50.2654, Perimeter: 25.1327, Color: Red

```

## Best Practices

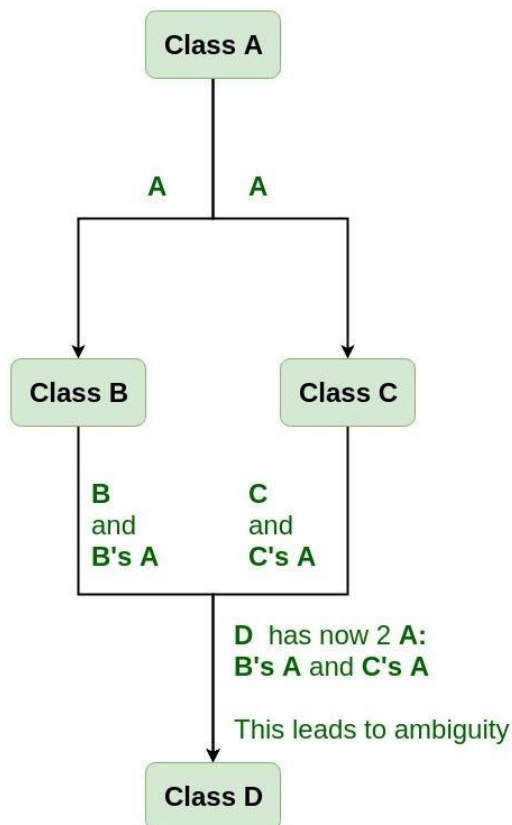
1. **Use abstract classes to define interfaces** that derived classes must implement.
2. **Keep pure virtual functions focused and simple**, defining only what must be implemented.
3. **Provide default implementations for non-pure virtual functions** when it makes sense.
4. **Use protected members** for data that derived classes need to access.
5. **Consider using abstract classes for framework design** where you want to enforce certain behavior.

| S.No. | Virtual Function                                                                                        | Pure Virtual Function                                                                                                                    |
|-------|---------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| 1.    | A virtual function is a member function in a parent class that can be further defined in a child class. | A pure virtual function is a member function in a parent class, and declaration is given in a parent class and defined in a child class. |
| 2.    | The classes that contain virtual functions are not abstract.                                            | The classes that contain pure virtual functions are abstract.                                                                            |
| 3.    | In the child classes, they may or may not redefine the virtual function.                                | The child classes must define the pure virtual function.                                                                                 |
| 4.    | Instantiation can be done from the parent class with a virtual function.                                | It Cannot be instantiated as it becomes an abstract class.                                                                               |
| 5.    | Definition of function is provided in the parent class.                                                 | Definition of function is not provided in the parent class.                                                                              |

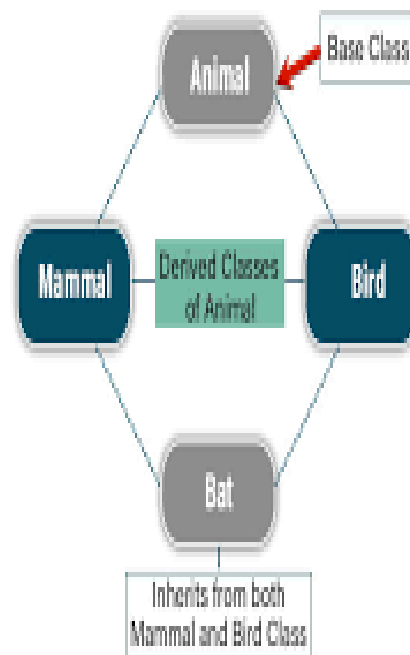
## Virtual Inheritance and Diamond Problem

The diamond problem occurs in multiple inheritance when a class derives from two classes that both inherit from a common base class. Virtual inheritance solves this problem by ensuring only one instance of the base class exists.

### Diamond Problem Diagram:



### Diamond Problem in C++



Virtual inheritance allows you to create a shared base class so that only one instance of the base class exists in the object hierarchy, regardless of how many derived classes inherit from it. This way, the diamond problem is resolved, and there is no ambiguity when accessing members of the shared base class through the derived classes.

Consider the following class hierarchy:

```
class Animal {
public:
    void makeSound() {
        cout << "Animal sound\n";
    }
};

class Mammal : public Animal {
public:
    void move() {
        cout << "Mammal moves\n";
    }
};

class Bird : public Animal {
public:
    void move() {
        cout << "Bird moves\n";
    }
};

class Bat : public Mammal, public Bird {
public:
    // ...
};
```

**In this example, we have an Animal class, and two derived classes Mammal and Bird. Both Mammal and Bird inherit from the Animal class.**

Now, we have a Bat class that is derived from both Mammal and Bird . The Bat class is the cause of the diamond problem since it indirectly inherits Animal twice through Mammal and Bird.

To resolve the diamond problem, we can use virtual inheritance.

### Implementation

To use virtual inheritance, the `virtual` keyword is added to the inheritance declaration:

```
class B : virtual public A { ... };
class C : virtual public A { ... };
class D : public B, public C { ... };
```

**This ensures that only one instance of A exists in an object of type D.**

The Animal base class should be virtually inherited in Mammals and Bird like this:

```
class Animal {
public:
    void makeSound() {
        cout << "Animal sound\n";
    }
};

class Mammal : virtual public Animal { // Virtual inheritance
public:
    void move() {
        cout << "Mammal moves\n";
    }
};

class Bird : virtual public Animal { // Virtual inheritance
public:
    void move() {
        cout << "Bird moves\n";
    }
};

class Bat : public Mammal, public Bird {
public:
    // ...
};
```

**With virtual inheritance, both Mammal and Bird virtually inherit from Animal, which means there is only one shared instance of the Animal class in the Bat object**

### Code Example

```
#include <iostream>
using namespace std;
class A {
public:
    int a;
    A() : a(10) {
        cout << "A constructor called" << endl;
    }
};

// Without virtual inheritance
class B1 : public A {
public:
    B1() {
        cout << "B1 constructor called" << endl;
    }
};
```

```

class C1 : public A {
public:
    C1() {
        cout << "C1 constructor called" << endl;
    }
};

class D1 : public B1, public C1 {
public:
    D1() {
        cout << "D1 constructor called" << endl;
    }

    // Problem: D1 has two copies of A
    void display() {
        // cout << "a = " << a; // Error: ambiguous
        cout << "B1's a = " << B1::a << endl;
        cout << "C1's a = " << C1::a << endl;
    }
};

// With virtual inheritance
class B2 : virtual public A {
public:
    B2() {
        cout << "B2 constructor called" << endl;
    }
};

class C2 : virtual public A {
public:
    C2() {
        cout << "C2 constructor called" << endl;
    }
};

class D2 : public B2, public C2 {
public:
    D2() {
        cout << "D2 constructor called" << endl;
        a = 30; // Can access 'a' directly now
    }

    void display() {
        cout << "a = " << a << endl; // Now unambiguous
    }
};

```

```

int main() {
    cout << "Without virtual inheritance:" << endl;
    D1 d1;
    d1.display();

    cout << "\nWith virtual inheritance:" << endl;
    D2 d2;
    d2.display();

    return 0;
}

```

## Output:

Without virtual inheritance:

```

A constructor called
B1 constructor called
A constructor called
C1 constructor called
D1 constructor called
B1's a = 10
C1's a = 10

```

With virtual inheritance:

```

A constructor called
B2 constructor called
C2 constructor called
D2 constructor called
a = 30

```

## Constructor Calling Sequence

With virtual inheritance, the constructor calling sequence is different:

1. The constructor of the virtual base class (Animal) is called first
2. The constructors of the intermediate classes (Mammal, WingedAnimal) are called
3. The constructor of the most derived class (Bat) is called last

Additionally, it's the most derived class's responsibility to initialize the virtual base class, not the intermediate classes.

## Object Slicing - Addon Concept

Object slicing occurs when a derived class object is assigned to a base class object, resulting in the loss of derived class-specific members.

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

class Base {
protected:
    string name;

public:
    Base(string n) : name(n) {}

    virtual void display() {
        cout << "Base class: " << name << endl;
    }
};

class Derived : public Base {
private:
    int extraData;

public:
    Derived(string n, int d) : Base(n), extraData(d) {}

    void display() override {
        cout << "Derived class: " << name << ", Extra data: " << extraData << endl;
    }
};

int main() {
    Derived d("Object", 100);
    d.display(); // Displays name and extraData

    // Object slicing: extraData is lost
    Base b = d;
    b.display(); // Displays only name

    // No slicing with pointers or references
    Base* bp = &d;
    bp->display(); // Uses Derived::display()

    Base& br = d;
    br.display(); // Uses Derived::display()

    return 0;}
```





## Final specifier

The **"final"** specifier can be used to prevent classes from being inherited or virtual methods from being overridden.

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

class Base {
public:
    virtual void method1() {
        cout << "Base::method1()" << endl;
    }

    // Cannot be overridden in derived classes
    virtual void method2() final {
        cout << "Base::method2()" << endl;
    }
};

class Derived : public Base {
public:
    // Override method1
    void method1() override {
        cout << "Derived::method1()" << endl;
    }

    // Error: cannot override final method
    // void method2() override {
    //     cout << "Derived::method2()" << endl;
    // }
};

// Final class - cannot be inherited from
class FinalClass final: public Base {
public:
    void method1() override {
        cout << "FinalClass::method1()" << endl;
    }
};

// Error: cannot inherit from final class
// class AnotherDerived : public FinalClass {
// };
```

```

int main() {
    Base baseObj;
    Derived derivedObj;
    FinalClass finalObj;

    Base* b1 = &baseObj;
    Base* b2 = &derivedObj;
    Base* b3 = &finalObj;

    b1->method1(); // Base::method1()
    b2->method1(); // Derived::method1()
    b3->method1(); // FinalClass::method1()

    b1->method2(); // Base::method2()
    b2->method2(); // Base::method2() (cannot be overridden)
    b3->method2(); // Base::method2() (cannot be overridden)

    return 0;
}

```

## Override Specifier

The `override` specifier ensures that a function is overriding a virtual function from a base class. This helps catch errors where you might think you're overriding a function but actually aren't.

```

#include <iostream>
using namespace std;

class Base {
public:
    virtual void method1() {
        cout << "Base::method1()" << endl;
    }

    virtual void method2() const {
        cout << "Base::method2()" << endl;
    }
};

class Derived : public Base {
public:
    // Correct override
    void method1() override {
        cout << "Derived::method1()" << endl;
    }

    // Error: method2 in Base is const, this is not
    // void method2() override {
    //     cout << "Derived::method2()" << endl;
    // }
}

```

```
// }

// Correct override
void method2() const override {
    cout << "Derived::method2()" << endl;
}

// Error: no method3 in Base to override
// void method3() override {
//     cout << "Derived::method3()" << endl;
// }

};

int main() {
    Base* b = new Derived();
    b->method1(); // Derived::method1()
    b->method2(); // Derived::method2()
    delete b;
    return 0;
}
```

# Best Practices for C++ Inheritance

## 1. Use public inheritance for "is-a" relationships

- Use private or protected inheritance for "implemented-in-terms-of" relationships

## 2. Make base class destructor virtual if class has virtual functions

```
class Base {  
public:  
    virtual ~Base() { /* cleanup */ }  
};
```

## 3. Prefer virtual functions to dynamic\_cast

- Virtual functions provide better performance and cleaner design

## 4. Don't override non-virtual functions

- It can lead to confusion and unexpected behavior

## 5. Use override and final specifiers

- Helps catch errors and clarifies intentions

## 6. Apply the Liskov Substitution Principle

- A derived class should be substitutable for its base class without altering the program's correctness

## 7. Prefer composition over inheritance where appropriate

- Inheritance creates tight coupling; composition is often more flexible

## 8. Keep inheritance hierarchies shallow

- Deep hierarchies are difficult to understand and maintain

## 9. Use abstract base classes to define interfaces

- Promotes programming to an interface, not an implementation

## Summary

These three C++ concepts are powerful tools for designing complex object-oriented systems:

1. **Abstract Classes** provide a way to define interfaces and ensure derived classes implement required functionality.
2. **Scope Resolution Operator** helps resolve ambiguities in multiple inheritance and name hiding scenarios.
3. **Virtual Base Classes** solve the diamond inheritance problem by ensuring only one instance of a base class exists in the inheritance hierarchy.

When used properly, these features enable you to create clean, maintainable, and extensible code. However, they should be applied judiciously, as overuse can lead to complex and hard-to-maintain code structures.

# Friend Functions in C++

## What Are Friend Functions?

Friend functions in C++ are special functions that can access the private and protected members of a class, even though they are not member functions of that class.

## Basic Concept

- A friend function is declared inside a class with the friend keyword
- It is defined outside the class like a normal function
- It has access to all private and protected members of the class

## Why Use Friend Functions?

- To allow specific non-member functions to access private data
- For operator overloading when the left operand is not an object of the class
- To improve efficiency by accessing data directly
- When two classes need to closely interact with each other's private members

## Simple Example

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

class Box { private:
    double length;
    double width;
    double height;

public:
    // Constructor
    Box(double l = 0, double w = 0, double h = 0) { length =
        l;
        width = w; height =
        h;
    }

    // Friend function declaration
    friend double calculateVolume(Box box);
};

// Friend function definition
double calculateVolume(Box box) {
    // Direct access to private members
    return box.length * box.width * box.height;
}

int main() {
```

```

// Create Box object
Box myBox(3.0, 4.0, 5.0);

// Use friend function
cout << "Volume of the box: " << calculateVolume(myBox) << endl;

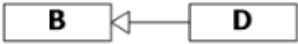



return 0;
}

```

## Key Points to Remember

1. Friend functions are not member functions of the class
2. They can access all private and protected members of the class
3. They are declared inside the class using the `friend` keyword
4. They are defined outside the class like normal functions
5. Friend functions cannot access class members directly by name - they need an object
6. Friendship is not inherited, transitive, or reciprocal

## Relationships between C++ Classes

| Relationship                                       | Diagram                                                                             | Code                                                                               | Explanation                                                                                                                                                              |
|----------------------------------------------------|-------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Inheritance<br>D "is-a" B                          |  | <pre>class D: public B { ... };</pre>                                              | Derived class D is a specialization of the Base class B. D inherits all the members of B                                                                                 |
| Composition<br>Ownership, P is "part-of" C         |  | <pre>class C { ... Private:     P p; };</pre>                                      | Composite class C owns, or contains, a part class P. P is created and destroyed with C. The interface of P is visible only to C, not its clients.                        |
| Aggregation<br>Ownership, P is "part-of" A         |  | <pre>class A { ... Void fun() { P* ptrP = new P(); ... } };</pre>                  | The Aggregator class A owns a part class P. P is created by a member function of A, and so its lifetime is strictly less than that of A. A is expected to destroy P.     |
| Using<br>Referral:<br>U uses R through a reference |  | <pre>public class U { ... public void register(R&amp; r) {     // use r } };</pre> | A class U uses instance of class R, to which it holds a reference. R is created by some other entity and a reference to it is passed to some member function of class U. |

## Class Relationships

## Real-world Example: Shape Hierarchy : EXTRA CODE

Here's a comprehensive example demonstrating many inheritance concepts with a shape hierarchy:

```
#include <iostream>
#include <vector>
#include <memory>
#include <cmath>
using namespace std;

// Abstract base class
class Shape {
protected:
    string color;

public:
    Shape(string color) : color(color) {}

    // Pure virtual functions (abstract methods)
    virtual double area() const = 0;
    virtual double perimeter() const = 0;

    // Virtual function with implementation
    virtual void display() const {
        cout << "This is a " << color << " shape." << endl;
    }

    // Non-virtual function
    string getColor() const {
        return color;
    }

    // Virtual destructor
    virtual ~Shape() {
        cout << "Shape destructor called" << endl;
    }
};

// Derived class: Circle
class Circle : public Shape {
private:
    double radius;
public:
    Circle(string color, double radius)
        : Shape(color), radius(radius) {}

    double area() const override {
        return M_PI * radius * radius;
    }
}
```

```

double perimeter() const override {
    return 2 * M_PI * radius;
}

void display() const override {
    cout << "This is a " << getColor() << " circle with radius " << radius << endl;
    cout << "Area: " << area() << ", Perimeter: " << perimeter() << endl;
}

~Circle() {
    cout << "Circle destructor called" << endl;
}
};

```

*// Derived class: Rectangle*

```

class Rectangle : public Shape {
private:
    double width;
    double height;

public:
    Rectangle(string color, double width, double height)
        : Shape(color), width(width), height(height) {}

    double area() const override {
        return width * height;
    }

    double perimeter() const override {
        return 2 * (width + height);
    }

    void display() const override {
        cout << "This is a " << getColor() << " rectangle with dimensions "
            << width << "x" << height << endl;
        cout << "Area: " << area() << ", Perimeter: " << perimeter() << endl;
    }

    ~Rectangle() {
        cout << "Rectangle destructor called" << endl;
    }
};

```

*// Further derived class: Square inherits from Rectangle*

```

class Square : public Rectangle {
public:

```



```

Square(string color, double side)
    : Rectangle(color, side, side) {}

void display() const override {
    cout << "This is a " << getColor() << " square" << endl;
    cout << "Area: " << area() << ", Perimeter: " << perimeter() << endl;
}

~Square() {
    cout << "Square destructor called" << endl;
}
};

```

*// Another derived class: Triangle*

```

class Triangle : public Shape {
private:
    double a, b, c; // Three sides

public:
    Triangle(string color, double a, double b, double c)
        : Shape(color), a(a), b(b), c(c) {
        // Check if triangle is valid
        if (a + b <= c || a + c <= b || b + c <= a) {
            throw invalid_argument("Invalid triangle sides");
        }
    }
}

```