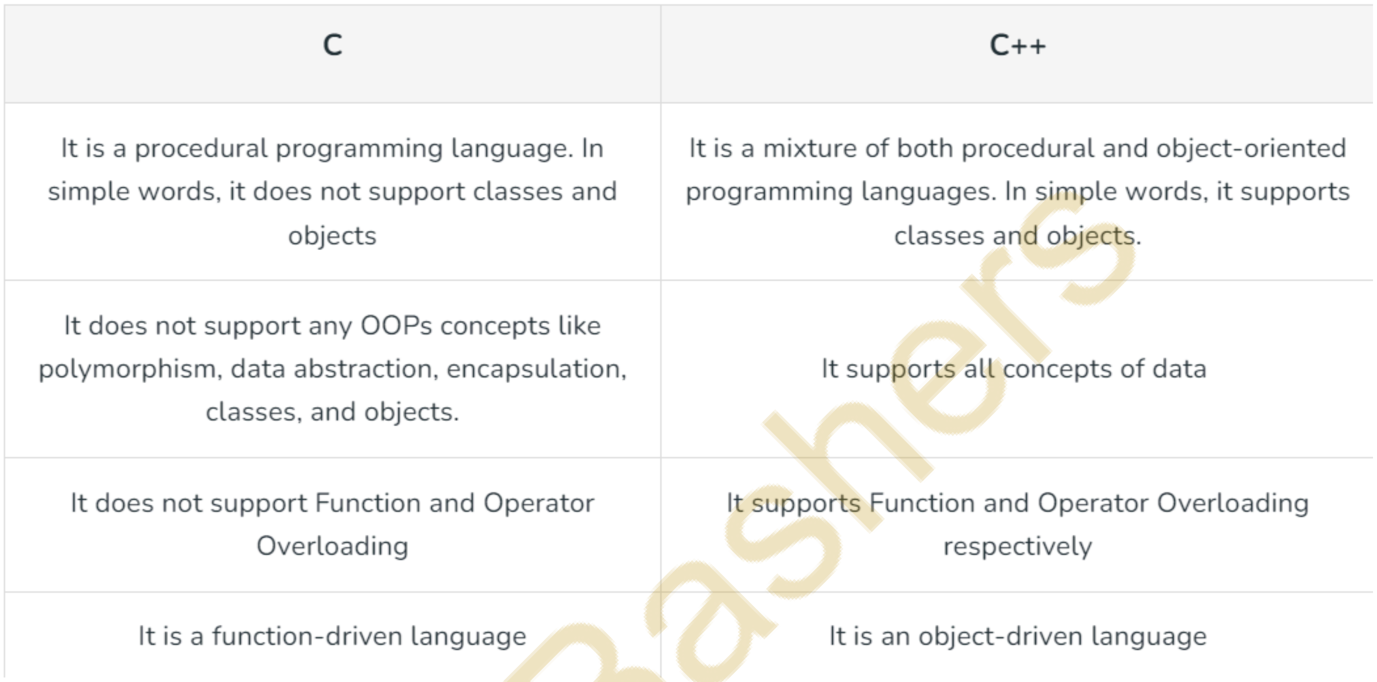
1. What is Difference Bw C and C++ ?



C	C++
It is a procedural programming language. In simple words, it does not support classes and objects	It is a mixture of both procedural and object-oriented programming languages. In simple words, it supports classes and objects.
It does not support any OOPs concepts like polymorphism, data abstraction, encapsulation, classes, and objects.	It supports all concepts of data
It does not support Function and Operator Overloading	It supports Function and Operator Overloading respectively
It is a function-driven language	It is an object-driven language

2. Explain OOPS Concepts In C++

Encapsulation : Encapsulation means binding of data (attributes) and methods (behaviors) that operate on the data into a single unit, called a class. This helps keep the implementation details hidden from the outside world, allowing for better organization and security.

By encapsulation we can keep data related to two different Department separate. For example in a company there are two department Sales and Accounts. So here sales data and people are separate from Accounts data and people.

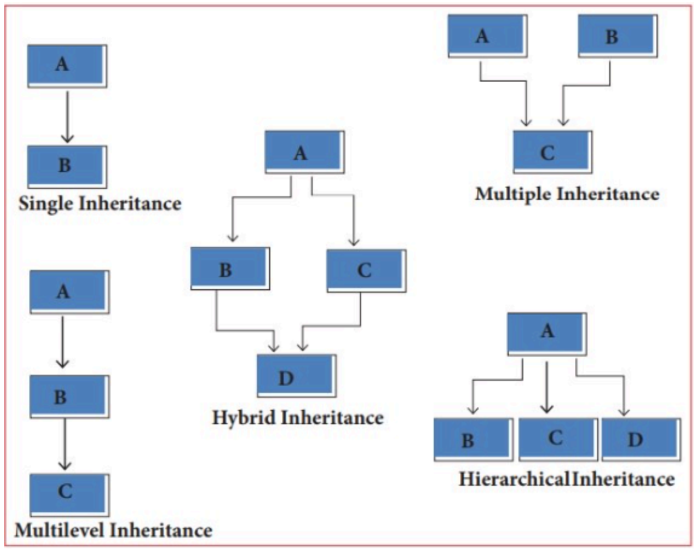
Abstraction : Abstraction means showing only the necessary details to user and hiding the implementation of it.

For example : We use ATM , we just press some buttons and cash comes out , but we dont know what actually is happening behind machine.

Polymorphism : It means existing in various forms . Two important types of Polymorphism is discussed later in this pdf.

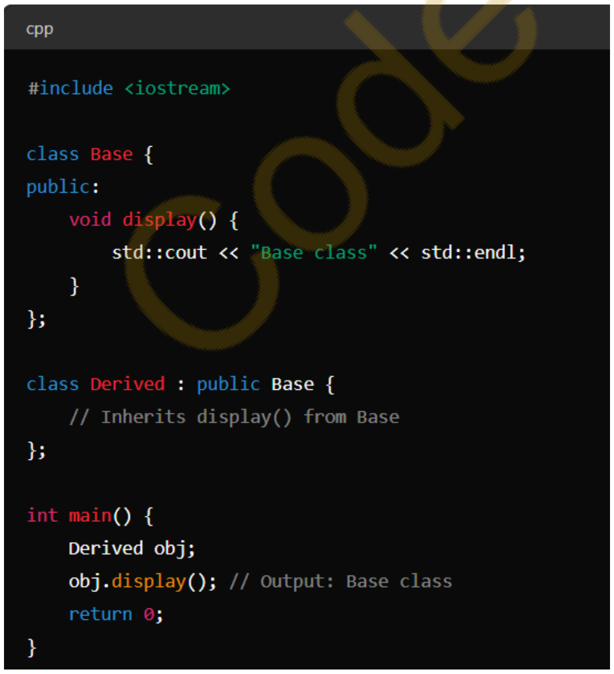
Inheritance : Inheritance is a way to create new classes based on existing ones. It allows a new class (called a derived class or subclass) to inherit properties and behaviors from an existing class (called a base class or superclass). This promotes code reuse and helps establish a hierarchy of classes.

3. What are Types of Inheritance In C++;



Single Inheritance:

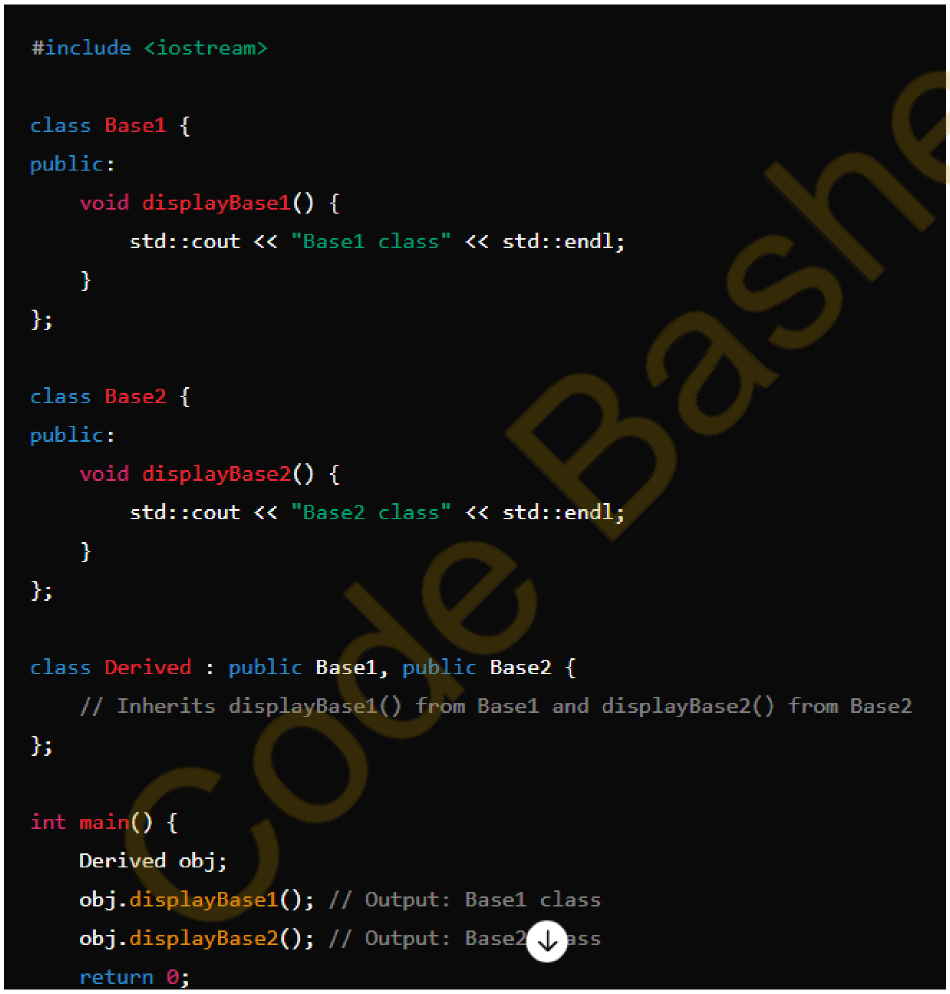
In single inheritance, a derived class inherits from only one base class. This is the simplest form of inheritance and represents a straightforward parent-child relationship.



```cpp
#include <iostream>

class Base {
public:
    void display() {
        std::cout << "Base class" << std::endl;
    }
};

class Derived : public Base {
    // Inherits display() from Base
};

int main() {
    Derived obj;
    obj.display(); // Output: Base class
    return 0;
}
```

Multiple Inheritance:

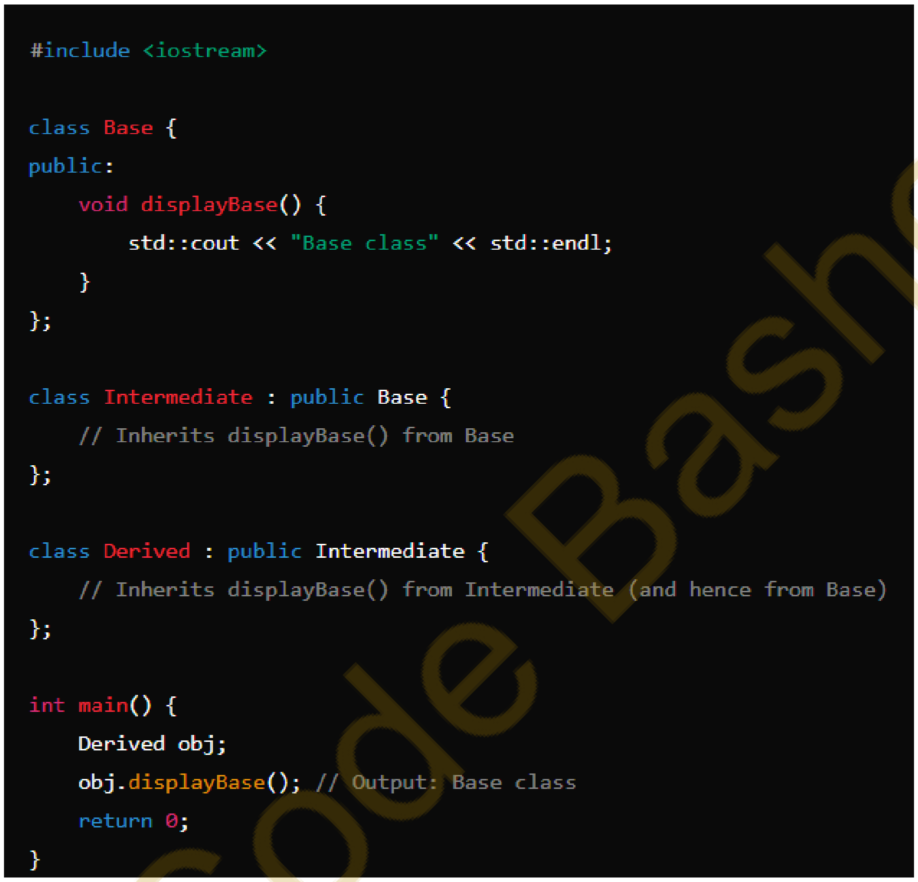
Multiple inheritance allows a derived class to inherit from more than one base class. This means that the derived class has access to the properties and behaviors of all its base classes. However, it can lead to complexity and ambiguity, especially when base classes have members with the same name.



```cpp
#include <iostream>

class Base1 {
public:
    void displayBase1() {
        std::cout << "Base1 class" << std::endl;
    }
};

class Base2 {
public:
    void displayBase2() {
        std::cout << "Base2 class" << std::endl;
    }
};

class Derived : public Base1, public Base2 {
    // Inherits displayBase1() from Base1 and displayBase2() from Base2
};

int main() {
    Derived obj;
    obj.displayBase1(); // Output: Base1 class
    obj.displayBase2(); // Output: Base2 class
    return 0;
```

Multilevel Inheritance:

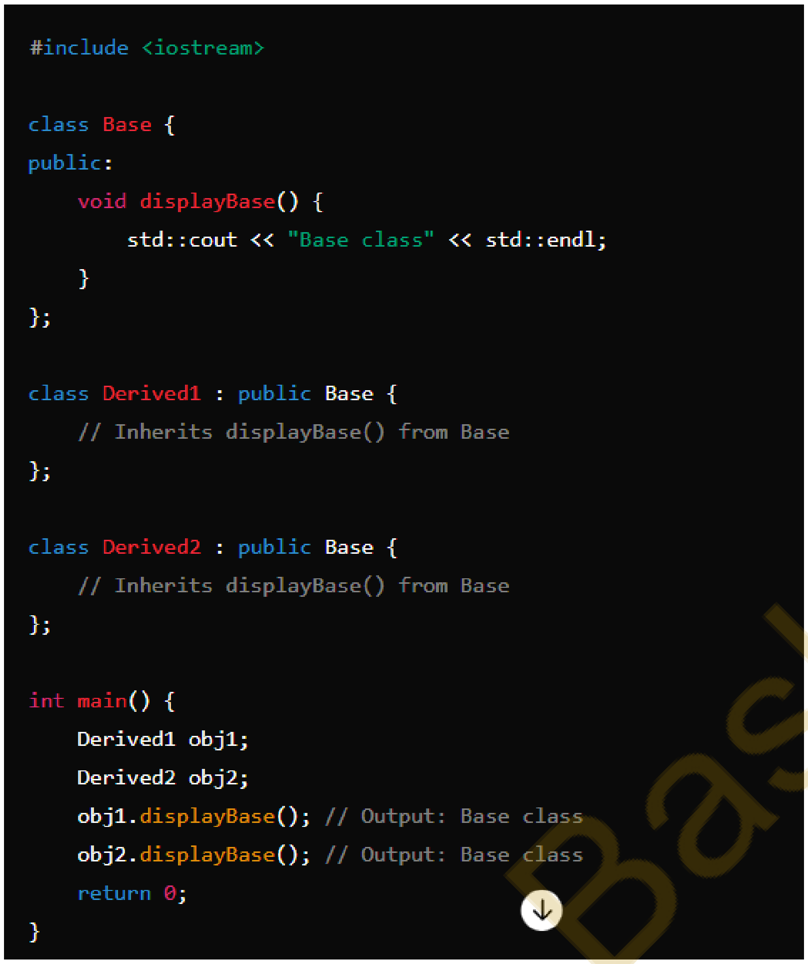
Multilevel inheritance is a type of inheritance where a class is derived from another derived class, creating a chain of inheritance. In this scenario, a class inherits from a base class, and another class inherits from this derived class, and so on.



```cpp
#include <iostream>

class Base {
public:
    void displayBase() {
        std::cout << "Base class" << std::endl;
    }
};

class Intermediate : public Base {
    // Inherits displayBase() from Base
};

class Derived : public Intermediate {
    // Inherits displayBase() from Intermediate (and hence from Base)
};

int main() {
    Derived obj;
    obj.displayBase(); // Output: Base class
    return 0;
}
```

Hierarchical Inheritance:

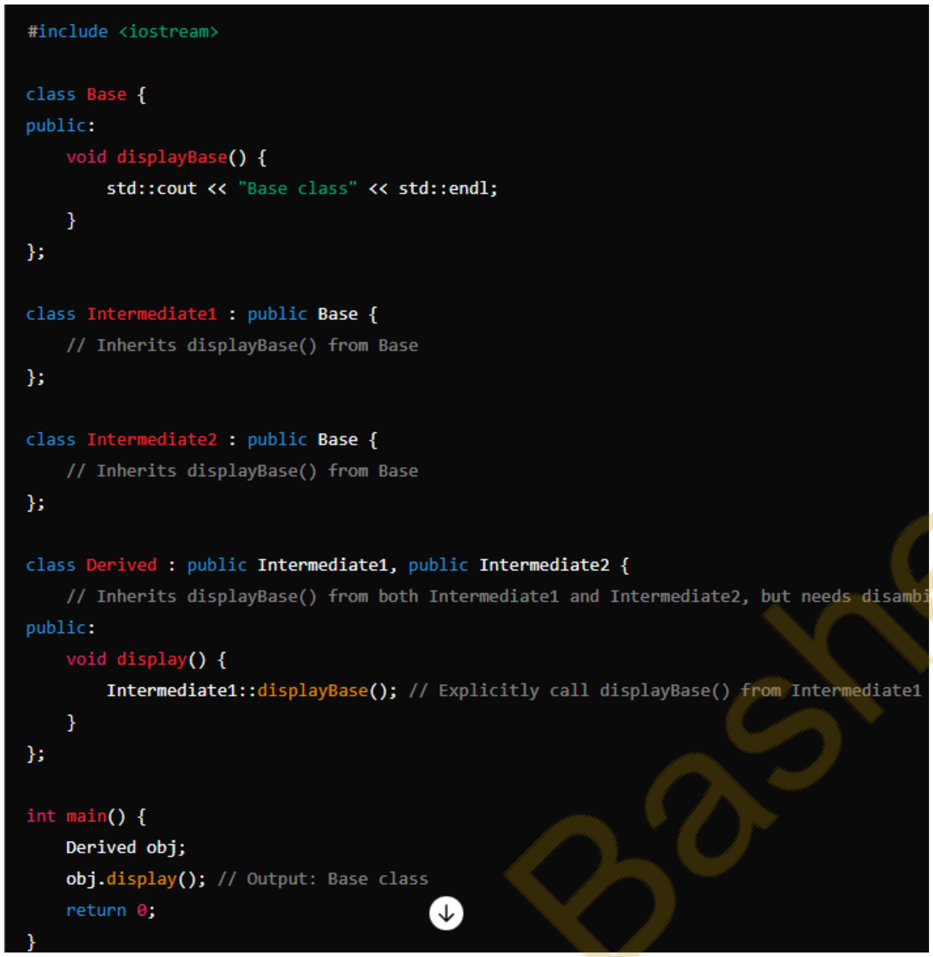
In hierarchical inheritance, multiple classes are derived from a single base class. This form of inheritance is useful when multiple derived classes share common characteristics or behaviors from the same base class.



```cpp
#include <iostream>

class Base {
public:
    void displayBase() {
        std::cout << "Base class" << std::endl;
    }
};

class Derived1 : public Base {
    // Inherits displayBase() from Base
};

class Derived2 : public Base {
    // Inherits displayBase() from Base
};

int main() {
    Derived1 obj1;
    Derived2 obj2;
    obj1.displayBase(); // Output: Base class
    obj2.displayBase(); // Output: Base class
    return 0;
}
```

Hybrid Inheritance:

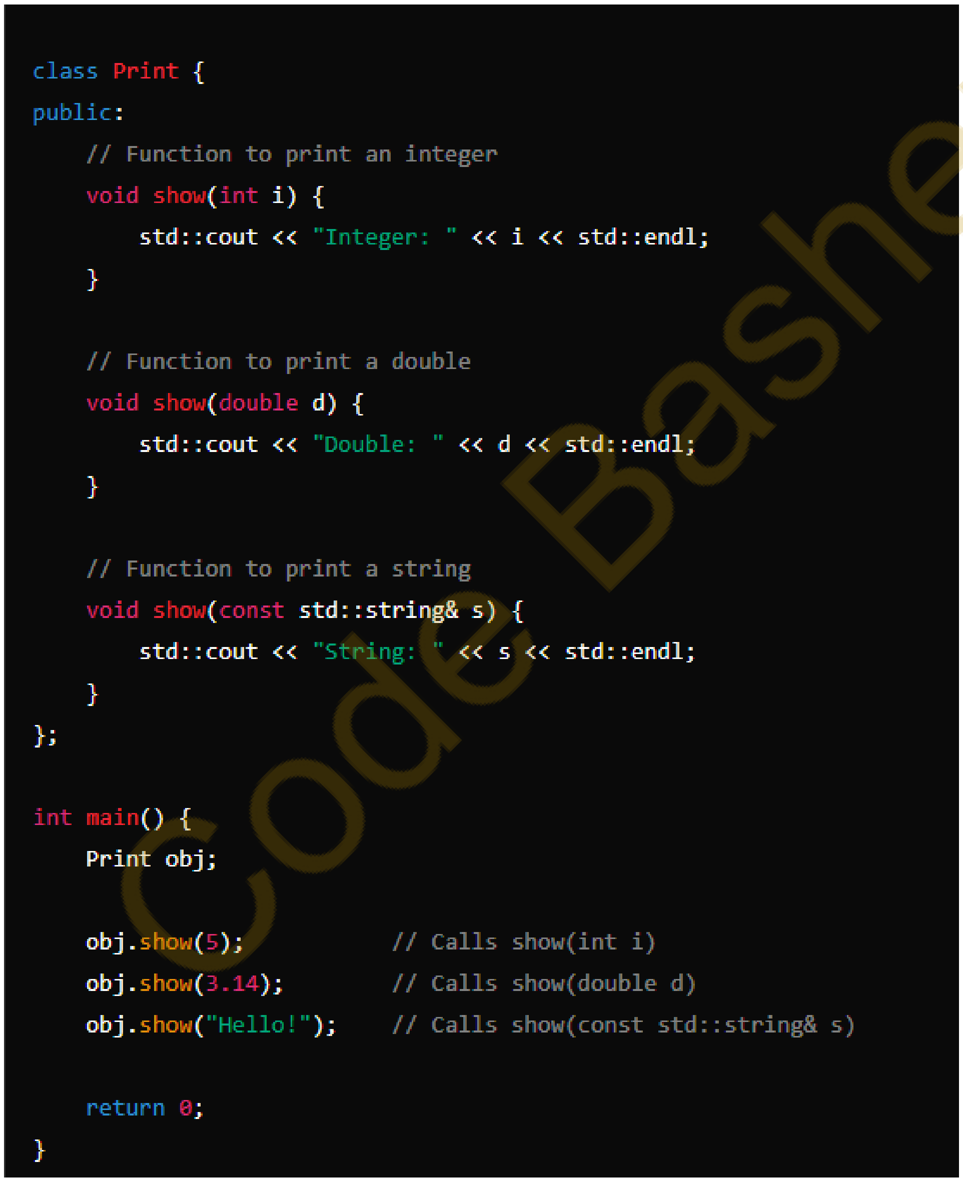
Hybrid inheritance is a combination of two or more types of inheritance. It typically involves a mix of single, multiple, multilevel, and hierarchical inheritance patterns. Hybrid inheritance allows for more complex relationships and can be used to design flexible and scalable systems.



```cpp
#include <iostream>

class Base {
public:
    void displayBase() {
        std::cout << "Base class" << std::endl;
    }
};

class Intermediate1 : public Base {
    // Inherits displayBase() from Base
};

class Intermediate2 : public Base {
    // Inherits displayBase() from Base
};

class Derived : public Intermediate1, public Intermediate2 {
    // Inherits displayBase() from both Intermediate1 and Intermediate2, but needs disambi
public:
    void display() {
        Intermediate1::displayBase(); // Explicitly call displayBase() from Intermediate1
    }
};

int main() {
    Derived obj;
    obj.display(); // Output: Base class
    return 0;
}
```

4. Explain Different Types of Polymorphism in C++

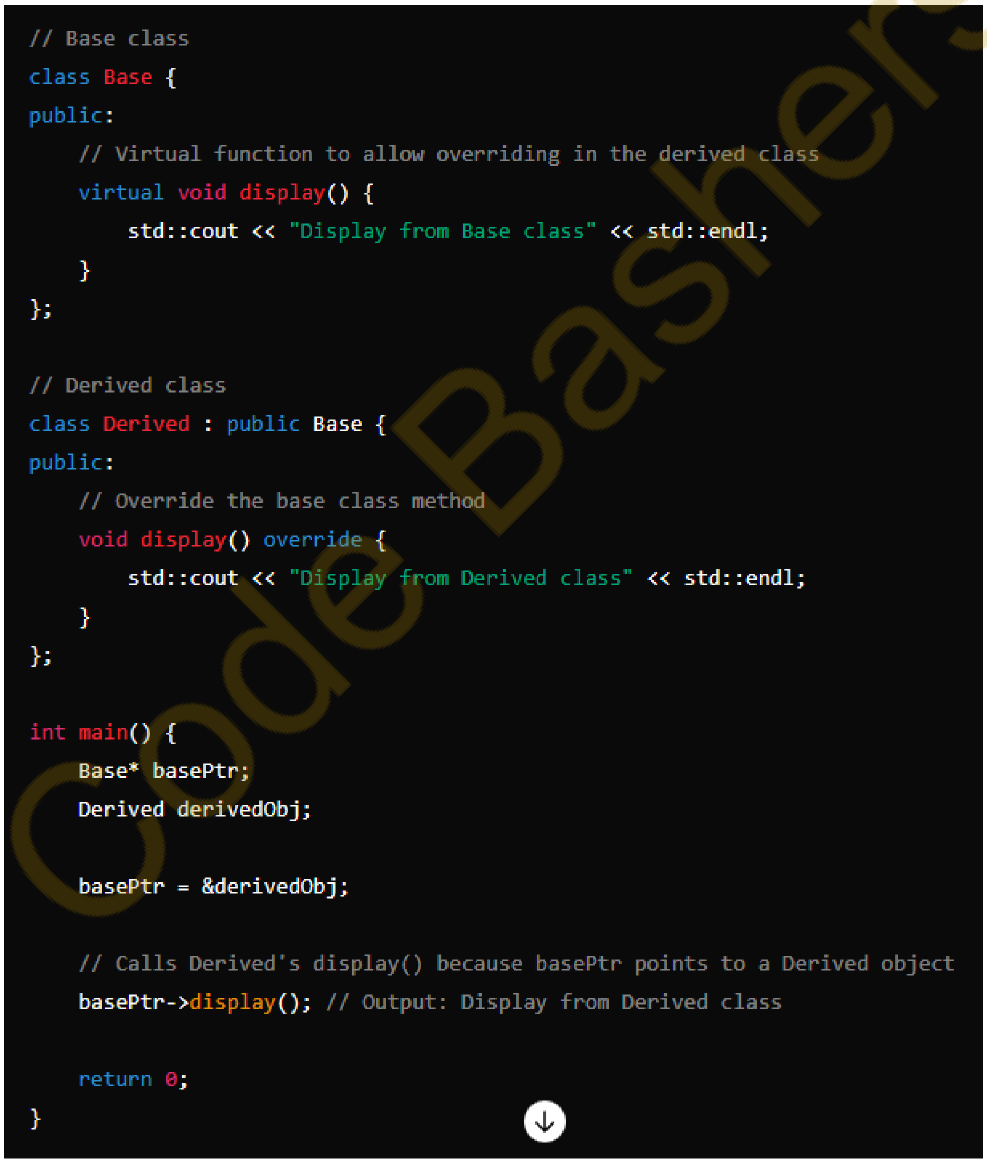
Function Overloading (compile-time-polymorphism):
Function overloading is a feature in C++ that allows multiple functions to have the same name but different parameters (either in number, type, or both). This is resolved at compile time, allowing the correct function to be called based on the arguments passed.



```cpp
class Print {
public:
    // Function to print an integer
    void show(int i) {
        std::cout << "Integer: " << i << std::endl;
    }

    // Function to print a double
    void show(double d) {
        std::cout << "Double: " << d << std::endl;
    }

    // Function to print a string
    void show(const std::string& s) {
        std::cout << "String: " << s << std::endl;
    }
};

int main() {
    Print obj;

    obj.show(5);            // Calls show(int i)
    obj.show(3.14);         // Calls show(double d)
    obj.show("Hello!");     // Calls show(const std::string& s)

    return 0;
}
```

Function Overriding (runtime-polymorphism):

Function overriding is a feature in C++ that allows a derived class to provide a specific implementation of a method that is already defined in its base class. This enables polymorphism, where a call to an overridden method will invoke the method defined in the derived class, even if the call is made using a base class pointer or reference.



```cpp
// Base class
class Base {
public:
    // Virtual function to allow overriding in the derived class
    virtual void display() {
        std::cout << "Display from Base class" << std::endl;
    }
};

// Derived class
class Derived : public Base {
public:
    // Override the base class method
    void display() override {
        std::cout << "Display from Derived class" << std::endl;
    }
};

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

    basePtr = &derivedObj;

    // Calls Derived's display() because basePtr points to a Derived object
    basePtr->display(); // Output: Display from Derived class

    return 0;
}
```

5. What are constructors and Types of Constructors

A constructor is a special member function in a class that is automatically called when an object of that class is created. The primary purpose of a constructor is to initialize the object's data members and allocate resources if necessary.

Key Points

Same Name as Class: The constructor has the same name as the class.

No Return Type: Constructors do not have a return type, not even void

Automatic Call: Constructors are automatically invoked when an object is created.

Initialization: Used to initialize objects of the class.

1. Default Constructor

- **What it is:** A constructor that doesn't take any arguments.
- **Purpose:** Initializes an object with default values.
- **Example:**

```cpp
class Person {
public:
    Person() {
        // Default values
    }
};
```

2. Parameterized Constructor

- **What it is:** A constructor that takes one or more arguments.
- **Purpose:** Initializes an object with specific values provided during creation.
- **Example:**

```cpp
class Person {
public:
    Person(std::string name, int age) {
        // Initialize with provided values
    }
};
```

3. Copy Constructor

- **What it is:** A constructor that takes a reference to an object of the same class.
- **Purpose:** Creates a new object as a copy of an existing object.
- **Example:**

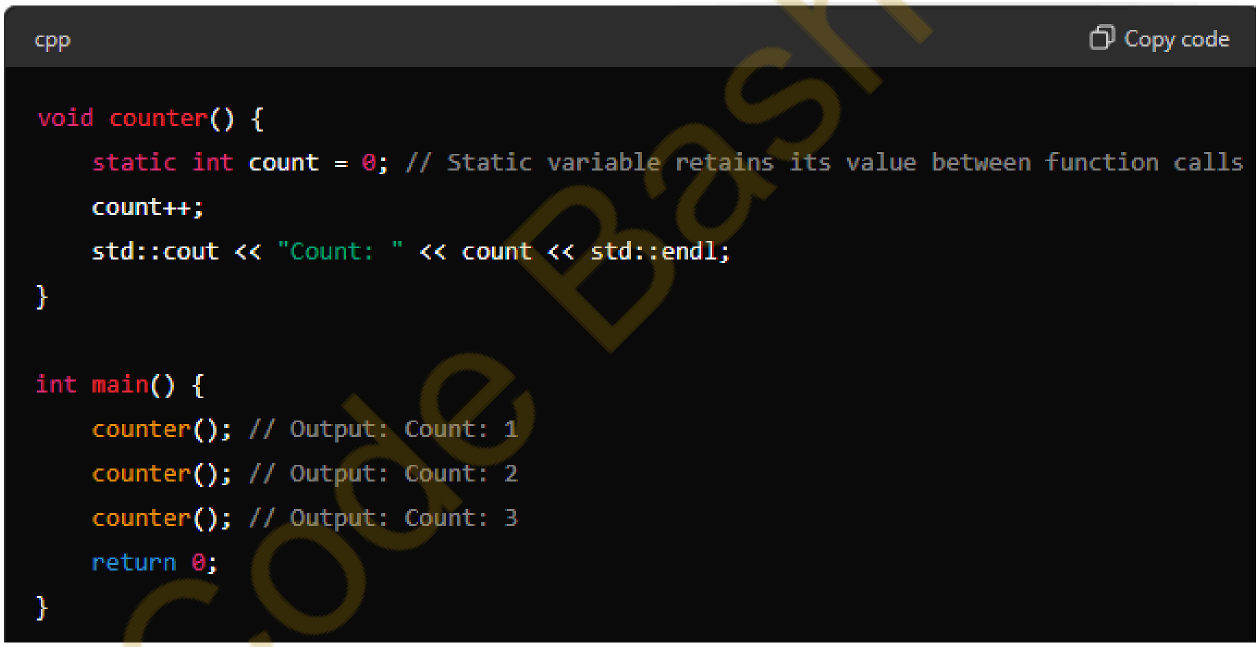
```cpp
class Person {
public:
    Person(const Person& other) {
        // Copy values from other object
    }
};
```

6. Explain Static Keyword and its use in C++;

Static Variables

Definition: When you declare a variable as static inside a function or a class, it means the variable is shared among all instances of that class or function.

Scope: For variables inside a function, they keep their value between function calls. For variables inside a class, they're shared by all objects of that class.

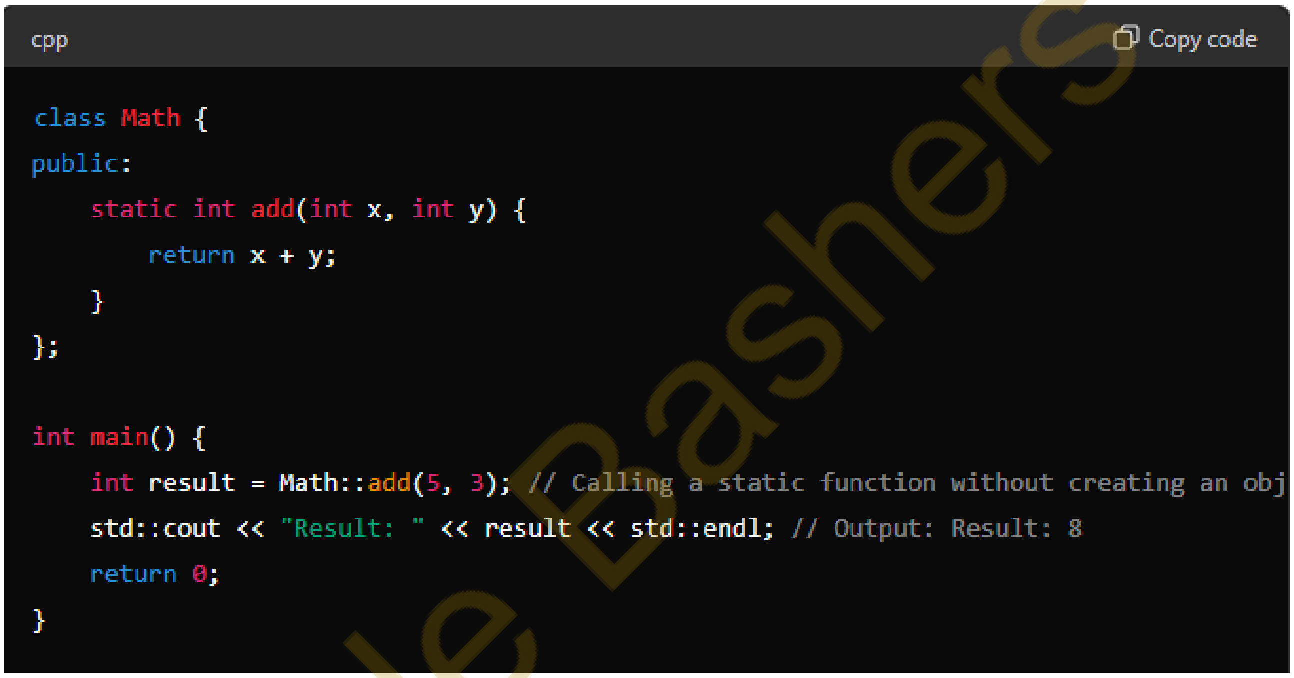


```cpp
void counter() {
    static int count = 0; // Static variable retains its value between function calls
    count++;
    std::cout << "Count: " << count << std::endl;
}

int main() {
    counter(); // Output: Count: 1
    counter(); // Output: Count: 2
    counter(); // Output: Count: 3
    return 0;
}
```

Static Functions (Member Functions)

Definition: When you declare a member function as static inside a class, it means the function can be called without an object of that class. It operates on class-level data and doesn't have access to object-specific data.

Usage: Often used for utility functions that don't need access to object-specific data.



```cpp
class Math {
public:
    static int add(int x, int y) {
        return x + y;
    }
};

int main() {
    int result = Math::add(5, 3); // Calling a static function without creating an obj
    std::cout << "Result: " << result << std::endl; // Output: Result: 8
    return 0;
}
```

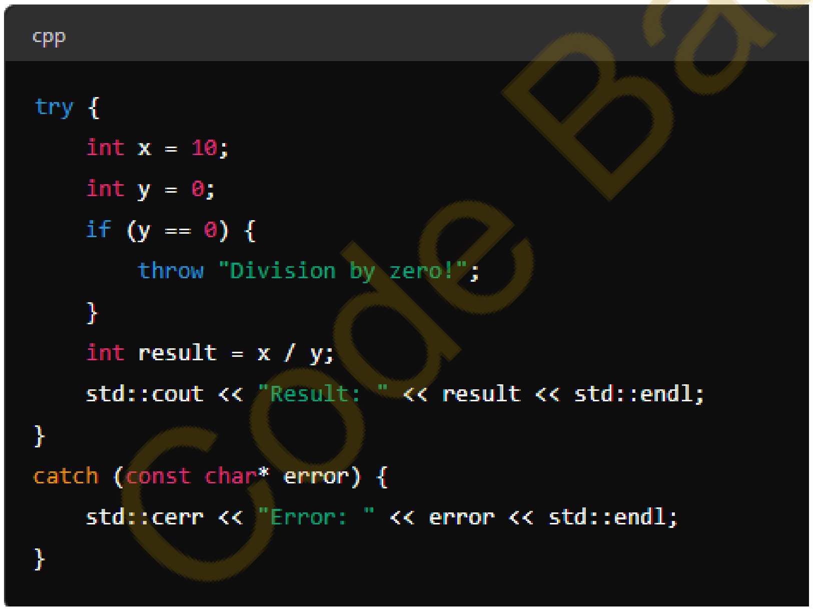
7. Explain Exceptional Handling in C++

What is Exception Handling?

When something unexpected happens in your program (like dividing by zero or trying to read a file that doesn't exist), exception handling helps you deal with it gracefully instead of crashing your program.

How does it work?

1. **Try-Catch Blocks**: You put the risky code inside a try block.
2. **Throwing Exceptions**: If something goes wrong inside the try block, you can "throw" an exception.
3. **Catching Exceptions**: You have a catch block to catch the thrown exception and handle it.



```cpp
try {
    int x = 10;
    int y = 0;
    if (y == 0) {
        throw "Division by zero!";
    }
    int result = x / y;
    std::cout << "Result: " << result << std::endl;
}
catch (const char* error) {
    std::cerr << "Error: " << error << std::endl;
}
```

Why is it useful?

- **Prevents Crashes**: Instead of crashing your program, it handles errors gracefully.
- **Makes Code Safer**: Helps you write safer code by handling unexpected situations.

- **Improves Readability**: Keeps error-handling code separate, making your code easier to read and understand.

8. What are Access Specifiers in C++

Access specifiers in C++ are keywords that control the visibility of class members (variables and functions) from outside the class. They determine who can access and modify these members. Here's a simple explanation:

Public

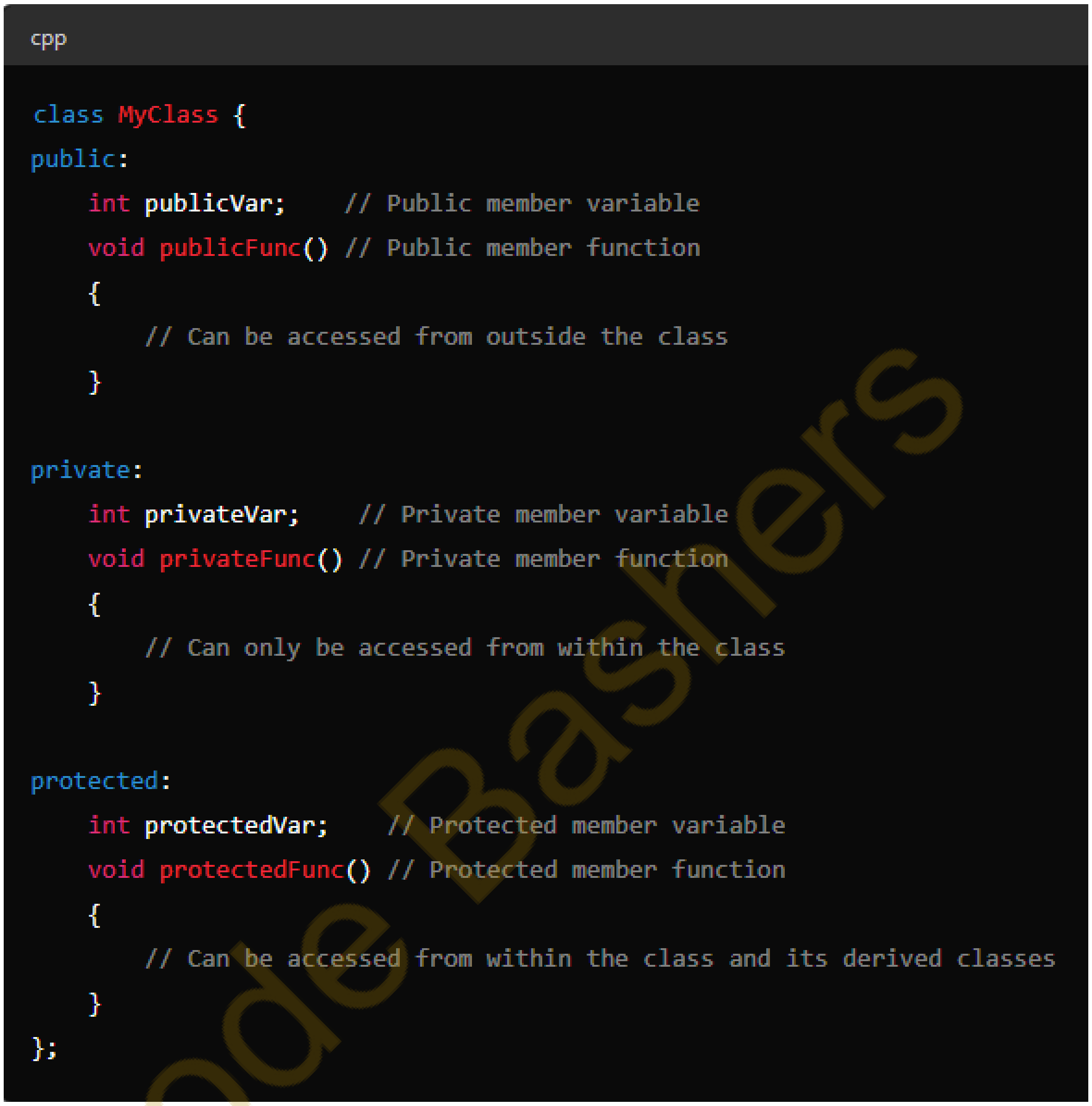
- **What it does**: Members declared as `public` are accessible from anywhere outside the class.
- **Example**: If a member is declared as `public`, it's like putting it on a public billboard – everyone can see it and use it.

Private

- **What it does**: Members declared as `private` are only accessible from within the same class.
- **Example**: If a member is declared as `private`, it's like keeping it in a locked room – only members of the class can access it.

Protected

- **What it does**: Members declared as `protected` are accessible from within the same class and its derived classes.
- **Example**: If a member is declared as `protected`, it's like putting it in a room with a special key – only certain classes (and their children) can access it.

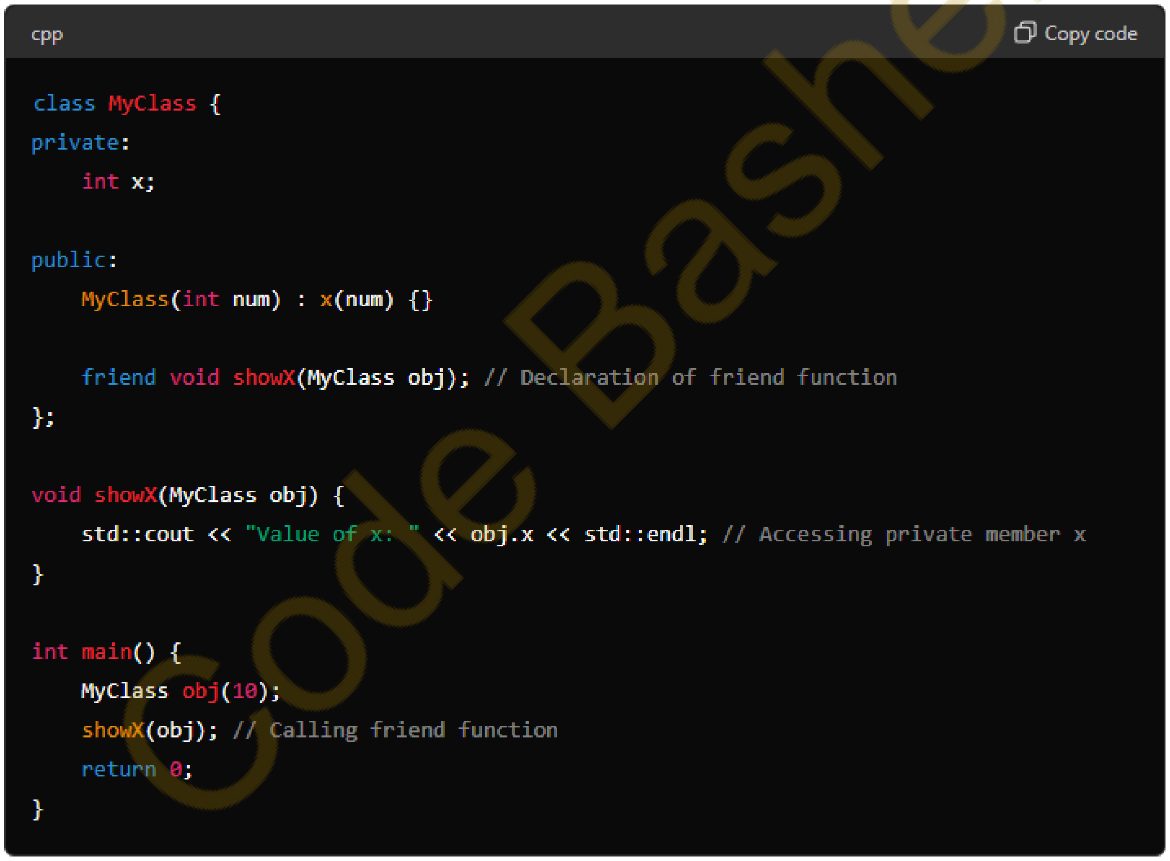

cpp

```cpp
class MyClass {
public:
    int publicVar;    // Public member variable
    void publicFunc() // Public member function
    {
        // Can be accessed from outside the class
    }

private:
    int privateVar;    // Private member variable
    void privateFunc() // Private member function
    {
        // Can only be accessed from within the class
    }

protected:
    int protectedVar;    // Protected member variable
    void protectedFunc() // Protected member function
    {
        // Can be accessed from within the class and its derived classes
    }
};
```

9. What is Friend Function and Friend Class;

Friend Function:

- **What it does**: Allows a function to access private and protected members of a class as if it were a member of that class.
- **Usage**: Useful when you have a function that needs access to private or protected members of a class but isn't actually a member function of that class.
- **Declaration**: You declare a friend function inside the class declaration using the `friend` keyword.



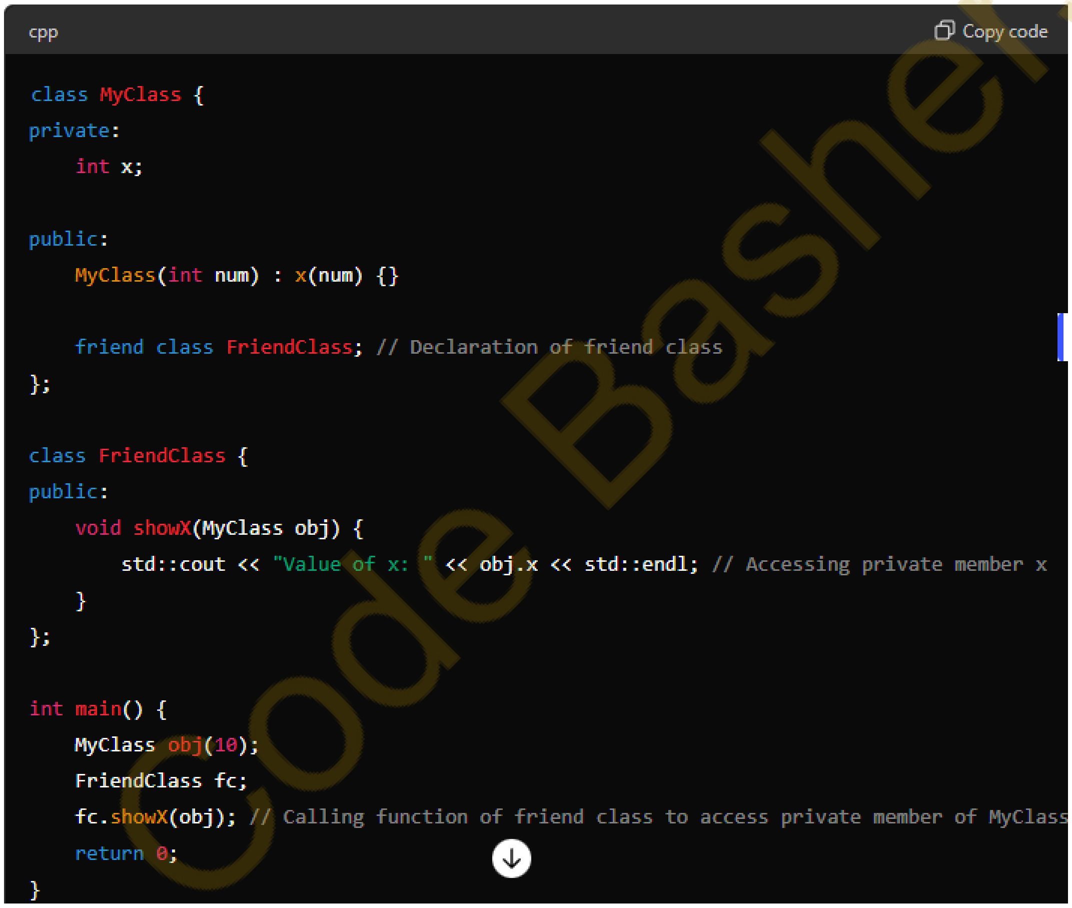
```cpp
class MyClass {
private:
    int x;

public:
    MyClass(int num) : x(num) {}

    friend void showX(MyClass obj); // Declaration of friend function
};

void showX(MyClass obj) {
    std::cout << "Value of x: " << obj.x << std::endl; // Accessing private member x
}

int main() {
    MyClass obj(10);
    showX(obj); // Calling friend function
    return 0;
}
```

Friend Class:

What it does: Allows a class to access private and protected members of another class.

Usage: Useful when you want to grant access to all members of one class to another class.

Declaration: You declare a friend class inside the class declaration using the friend keyword.



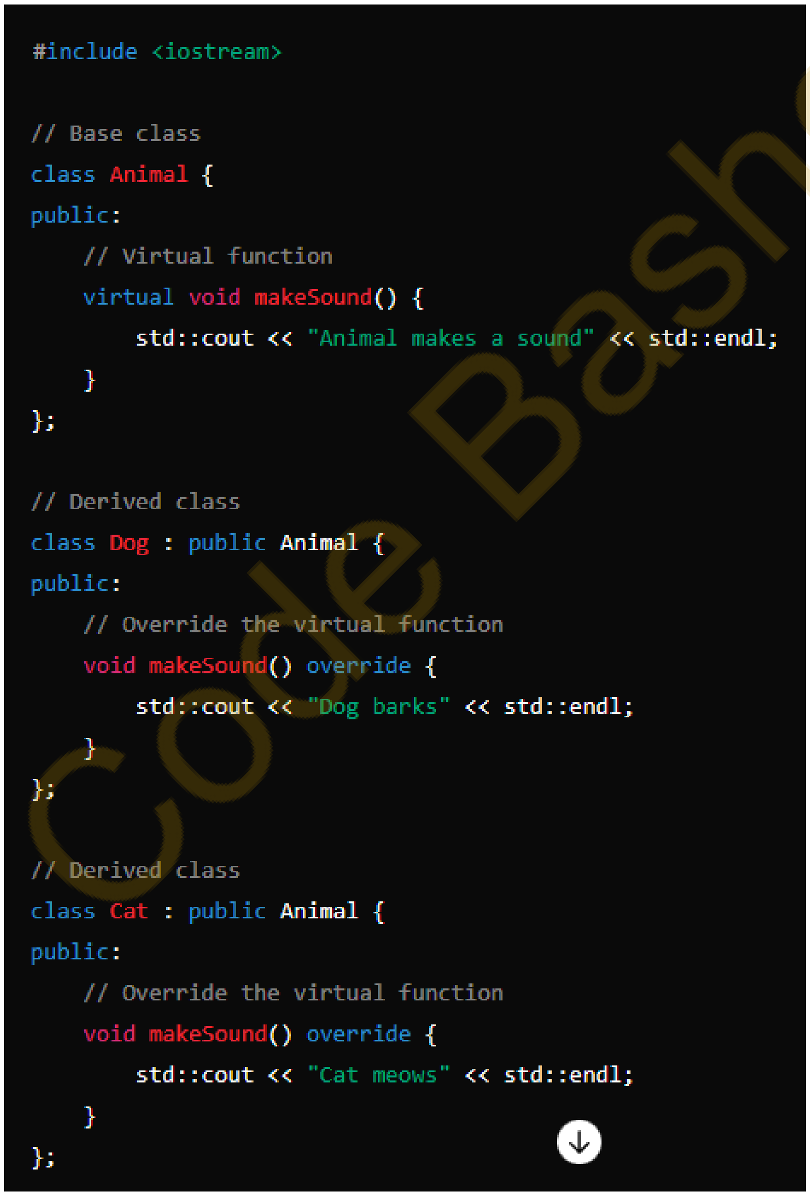
cpp

Copy code

```cpp
class MyClass {
private:
    int x;

public:
    MyClass(int num) : x(num) {}

    friend class FriendClass; // Declaration of friend class
};

class FriendClass {
public:
    void showX(MyClass obj) {
        std::cout << "Value of x: " << obj.x << std::endl; // Accessing private member x
    }
};

int main() {
    MyClass obj(10);
    FriendClass fc;
    fc.showX(obj); // Calling function of friend class to access private member of MyClass
    return 0;
}
```

10. Explain Virtual Keyword In C++;

The `virtual` keyword in C++ is used to enable polymorphic behavior in classes. When a function is declared as `virtual` in a base class and is overridden in derived classes, it allows the correct version of the function to be called based on the actual object type during runtime. This helps achieve dynamic binding and enables more flexible and extensible code, especially when working with inheritance hierarchies.

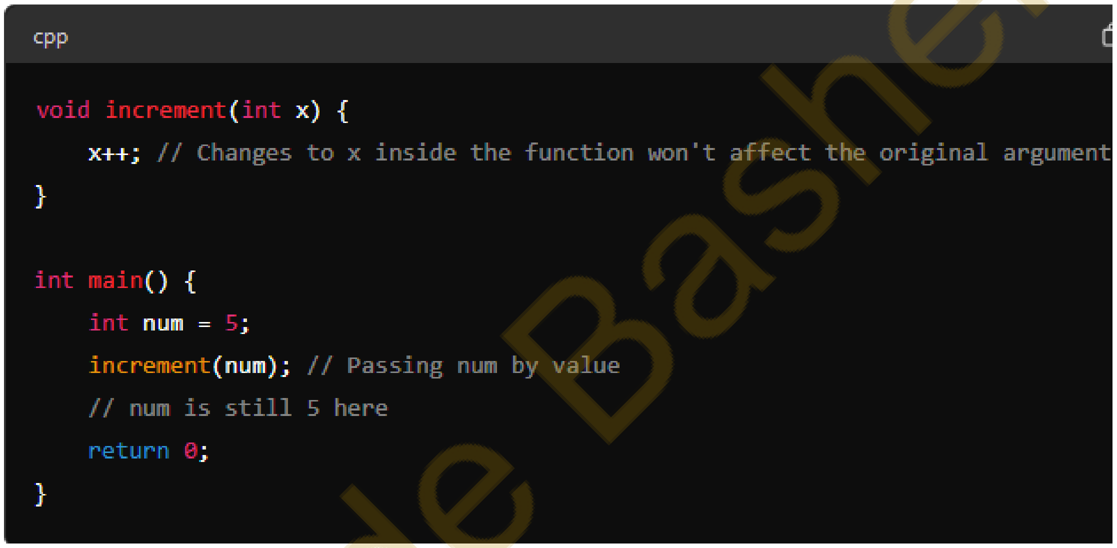


```cpp
#include <iostream>

// Base class
class Animal {
public:
    // Virtual function
    virtual void makeSound() {
        std::cout << "Animal makes a sound" << std::endl;
    }
};

// Derived class
class Dog : public Animal {
public:
    // Override the virtual function
    void makeSound() override {
        std::cout << "Dog barks" << std::endl;
    }
};

// Derived class
class Cat : public Animal {
public:
    // Override the virtual function
    void makeSound() override {
        std::cout << "Cat meows" << std::endl;
    }
};
```

11. What is call By value and call By reference

Call by Value:

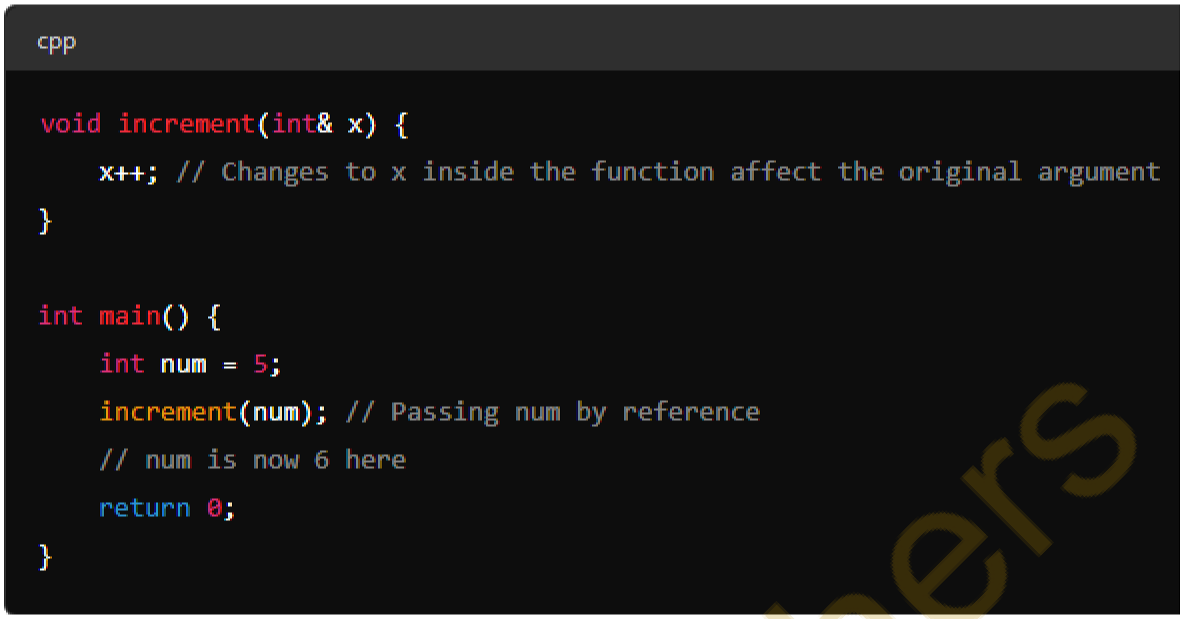
- **What it means**: In "call by value," a copy of the actual parameter's value is passed to the function parameter.
- **Behavior**: Any changes made to the parameter inside the function do not affect the original argument.
- **Usage**: It's useful when you don't want the function to modify the original value of the argument.


cpp

```cpp
void increment(int x) {
    x++; // Changes to x inside the function won't affect the original argument
}

int main() {
    int num = 5;
    increment(num); // Passing num by value
    // num is still 5 here
    return 0;
}
```

Call by Reference:

- **What it means**: In "call by reference," the memory address of the actual parameter is passed to the function parameter.
- **Behavior**: Changes made to the parameter inside the function directly affect the original argument.
- **Usage**: It's useful when you want the function to modify the original value of the argument.


cpp

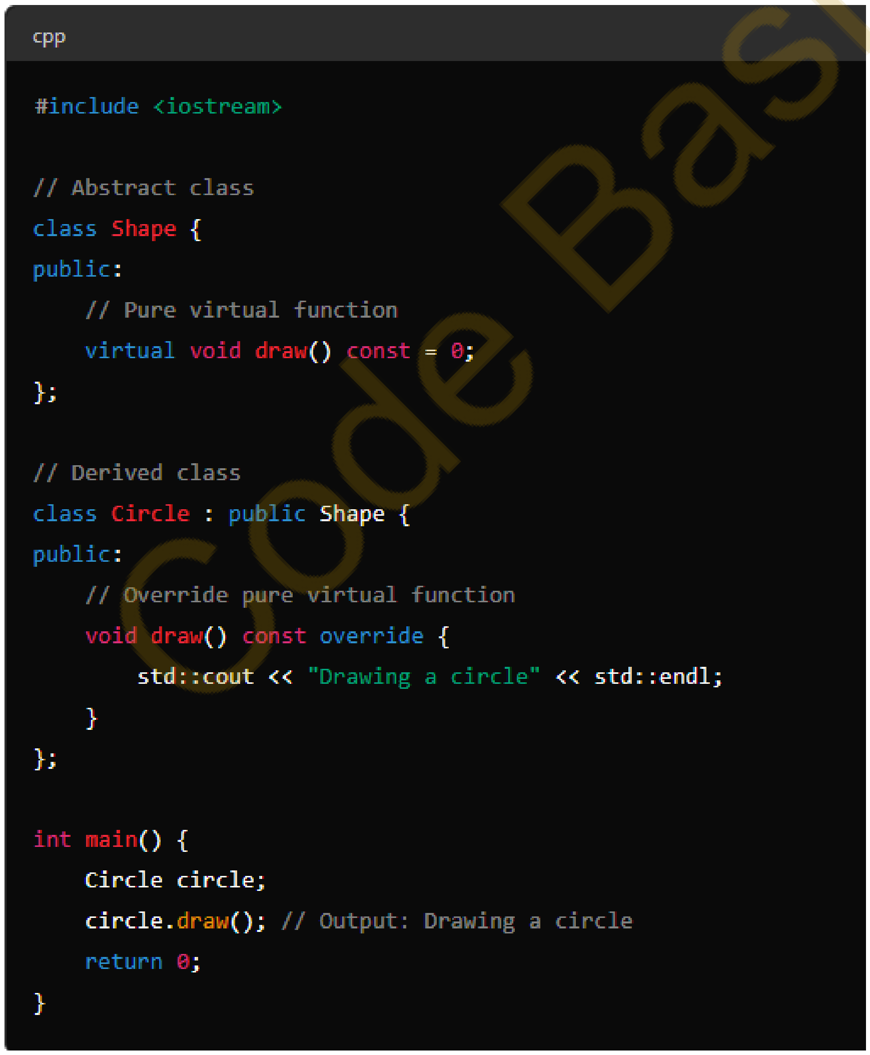
```cpp
void increment(int& x) {
    x++; // Changes to x inside the function affect the original argument
}

int main() {
    int num = 5;
    increment(num); // Passing num by reference
    // num is now 6 here
    return 0;
}
```

12. What is Abstract Class in C++;

An abstract class in C++ is like a blueprint for other classes. It's a class that can't be instantiated on its own, meaning you can't create objects directly from it. Instead, it's meant to serve as a base for other classes to inherit from.

Key Points:

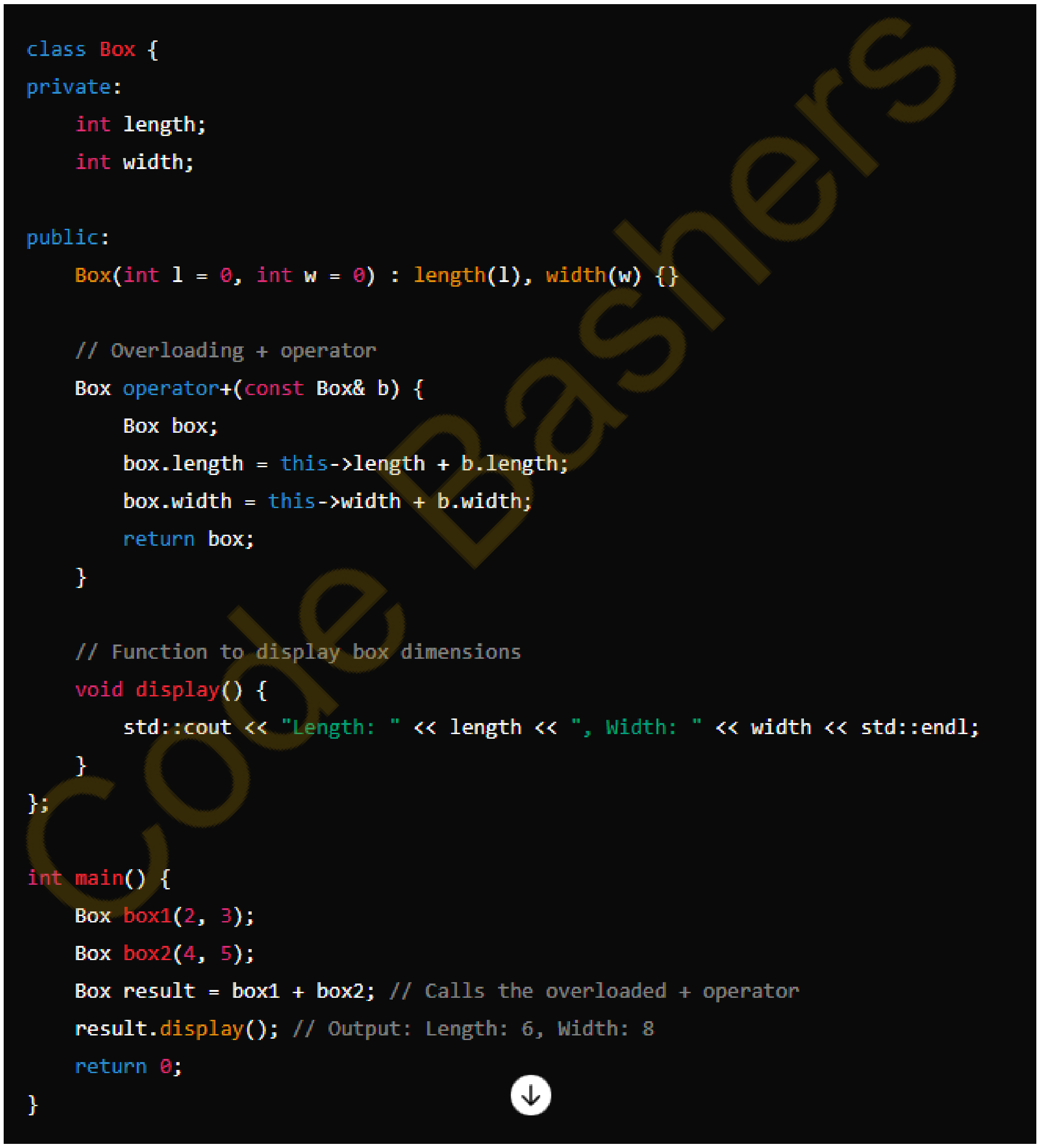
1. **Cannot Be Instantiated**: You cannot create objects of an abstract class directly.
2. **Contains Pure Virtual Functions**: An abstract class often contains at least one pure virtual function, which means the function has no implementation and must be overridden by derived classes.
3. **Intended for Inheritance**: It's designed to be inherited by other classes, which will provide implementations for its pure virtual functions.



```cpp
#include <iostream>

// Abstract class
class Shape {
public:
    // Pure virtual function
    virtual void draw() const = 0;
};

// Derived class
class Circle : public Shape {
public:
    // Override pure virtual function
    void draw() const override {
        std::cout << "Drawing a circle" << std::endl;
    }
};

int main() {
    Circle circle;
    circle.draw(); // Output: Drawing a circle
    return 0;
}
```

13. What is Operator Overloading?

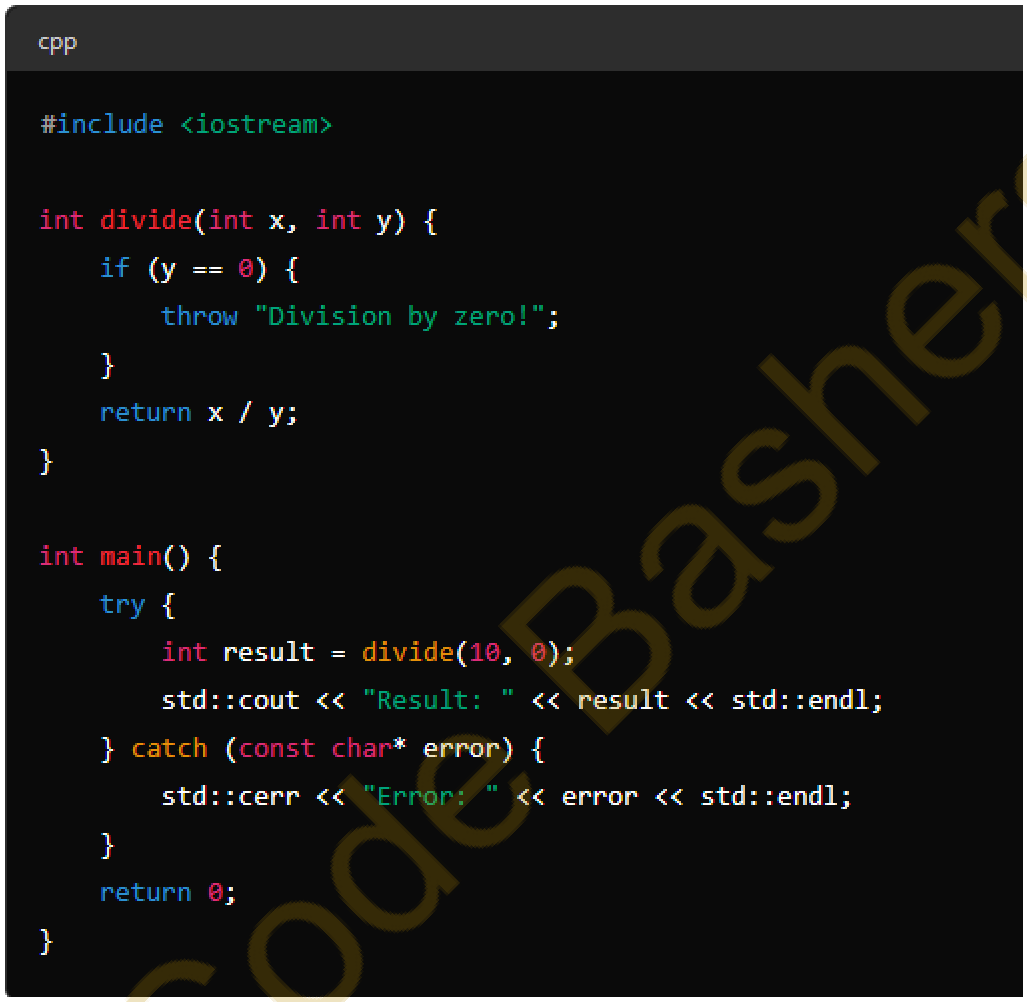
- **Redefining Operator Behavior**: Allows you to redefine what an operator does when applied to objects of your class.
- **Syntax**: You define a special member function inside your class for each operator you want to overload.



```
class Box {
private:
    int length;
    int width;

public:
    Box(int l = 0, int w = 0) : length(l), width(w) {}

    // Overloading + operator
    Box operator+(const Box& b) {
        Box box;
        box.length = this->length + b.length;
        box.width = this->width + b.width;
        return box;
    }

    // Function to display box dimensions
    void display() {
        std::cout << "Length: " << length << ", Width: " << width << std::endl;
    }
};

int main() {
    Box box1(2, 3);
    Box box2(4, 5);
    Box result = box1 + box2; // Calls the overloaded + operator
    result.display(); // Output: Length: 6, Width: 8
    return 0;
}
```

14. Explain Throw keyword in C++;

Throw keyword is used to throw exceptions explicitly , we can throw inbuilt and custom exceptions using throw keyword

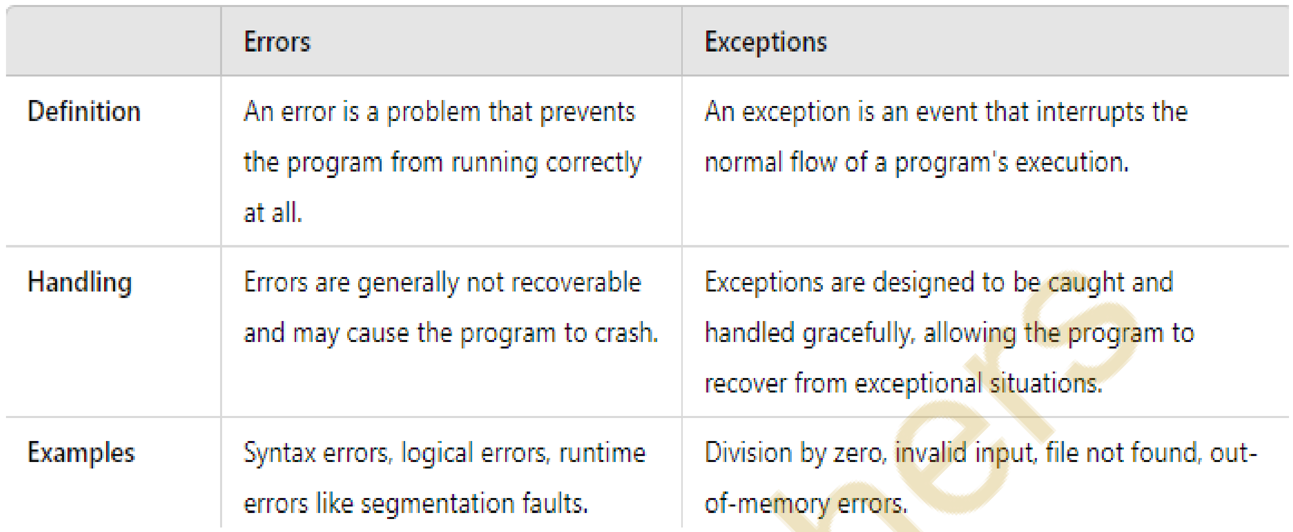


cpp

```cpp
#include <iostream>

int divide(int x, int y) {
    if (y == 0) {
        throw "Division by zero!";
    }
    return x / y;
}

int main() {
    try {
        int result = divide(10, 0);
        std::cout << "Result: " << result << std::endl;
    } catch (const char* error) {
        std::cerr << "Error: " << error << std::endl;
    }
    return 0;
}
```

\

15. Difference Bw Error and Exception



	Errors	Exceptions
Definition	An error is a problem that prevents the program from running correctly at all.	An exception is an event that interrupts the normal flow of a program's execution.
Handling	Errors are generally not recoverable and may cause the program to crash.	Exceptions are designed to be caught and handled gracefully, allowing the program to recover from exceptional situations.
Examples	Syntax errors, logical errors, runtime errors like segmentation faults.	Division by zero, invalid input, file not found, out-of-memory errors.

16. Explain Concept of Multithreading.

What is Multithreading?

- **Doing Many Things at Once**: Multithreading allows a program to execute multiple tasks simultaneously.
- **Independent Workers**: Threads are like independent workers in your program that can perform tasks concurrently.
- **Parallel Execution**: Multithreading enables parts of your program to run in parallel, making your program faster and more efficient.

Why Use Multithreading?

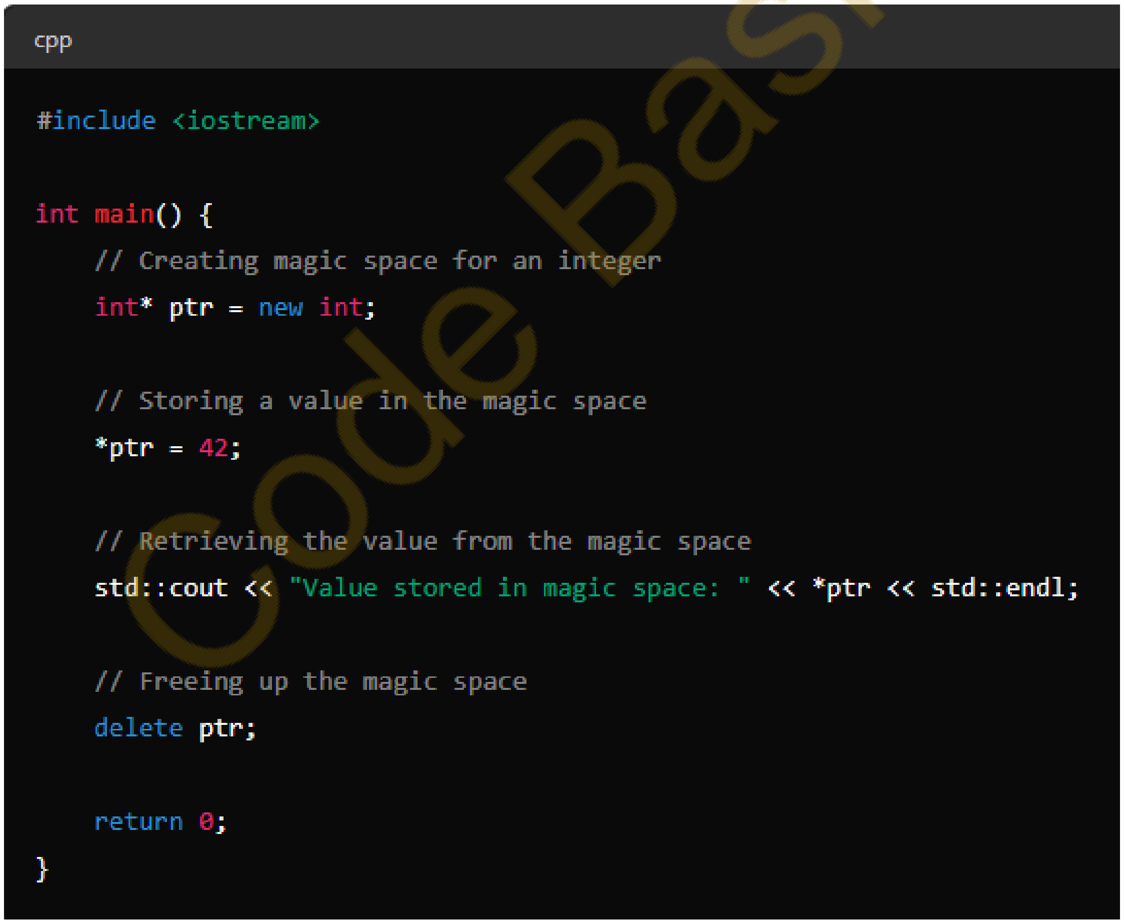
- **Concurrency**: Enables your program to perform multiple tasks concurrently, improving performance and responsiveness.
- **Utilize CPU Cores**: Allows your program to make use of multiple CPU cores effectively, maximizing hardware resources.
- **Asynchronous Operations**: Facilitates asynchronous operations such as handling user input while performing background tasks.

How Does it Work?

- **Create Threads**: You can create threads in your program using libraries like `<thread>` in C++.
- **Assign Tasks**: Each thread is assigned a task to execute concurrently

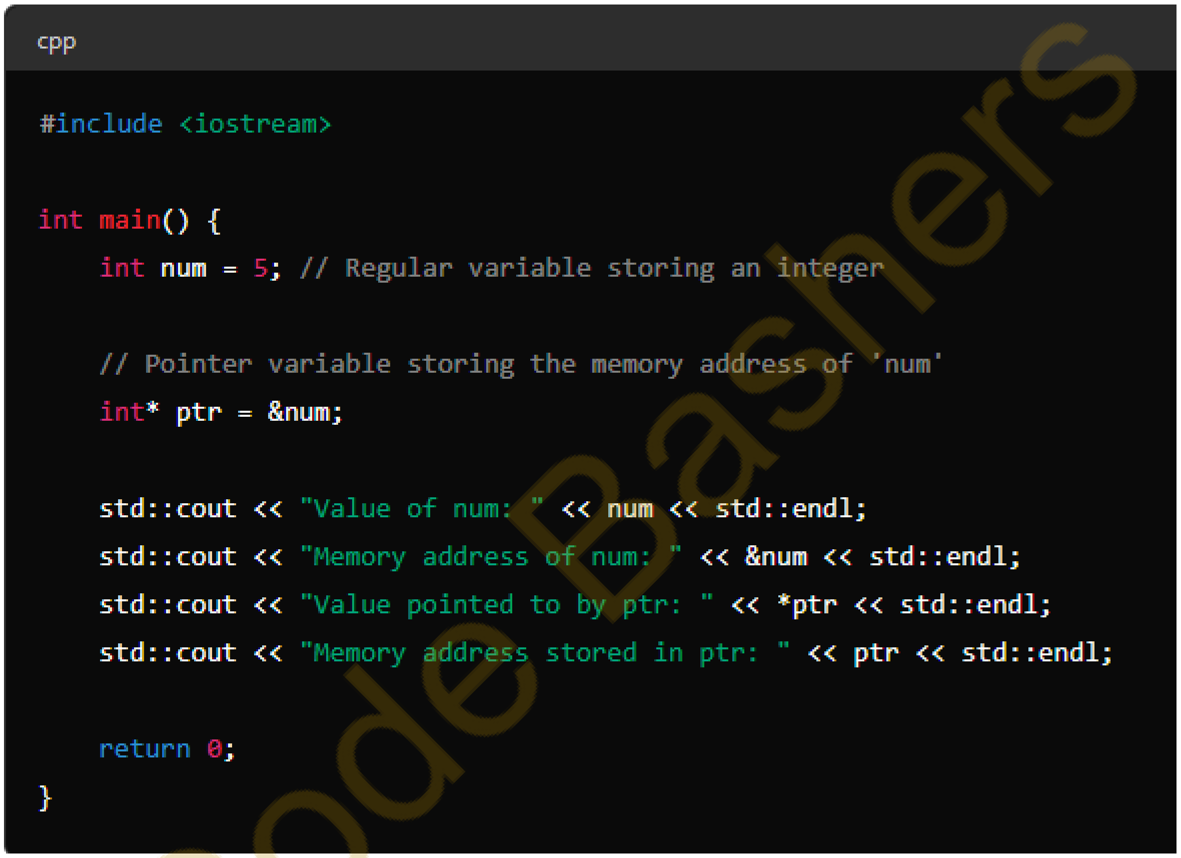
17. Explain The role of new and delete operators

`new`: Creates space (memory) dynamically.
`delete`: Makes space disappear, preventing memory leaks.
Together, they allow you to manage memory dynamically, creating and destroying objects as needed during program execution.


```cpp
#include <iostream>

int main() {
    // Creating magic space for an integer
    int* ptr = new int;

    // Storing a value in the magic space
    *ptr = 42;

    // Retrieving the value from the magic space
    std::cout << "Value stored in magic space: " << *ptr << std::endl;

    // Freeing up the magic space
    delete ptr;

    return 0;
}
```

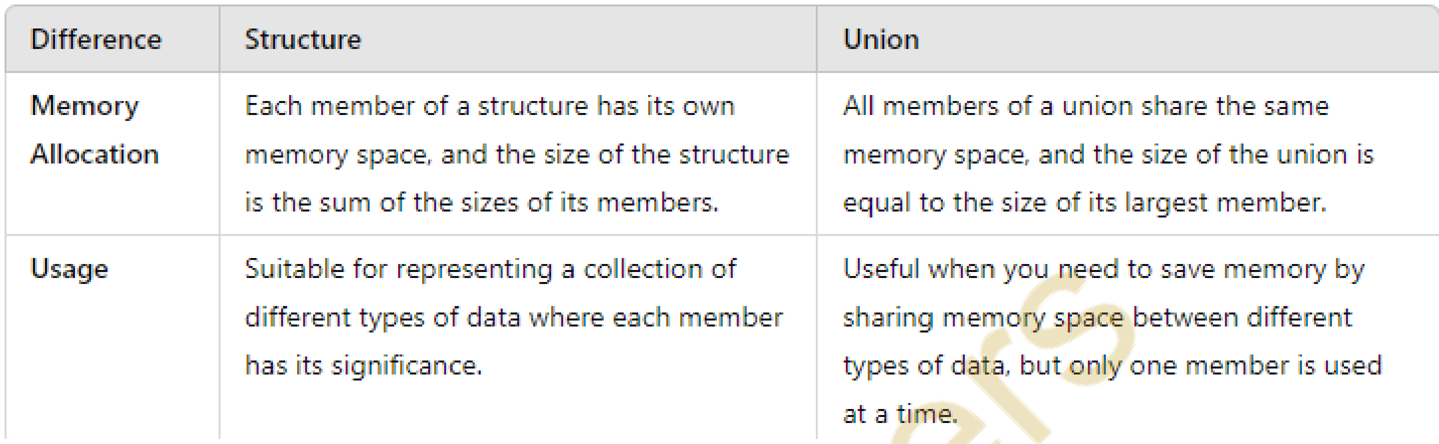
18. What are pointers in C++ ?

Pointers in C++ are like arrows pointing to specific memory locations. They allow you to indirectly access and manipulate data stored in memory. Here's a simplified explanation with a basic example:



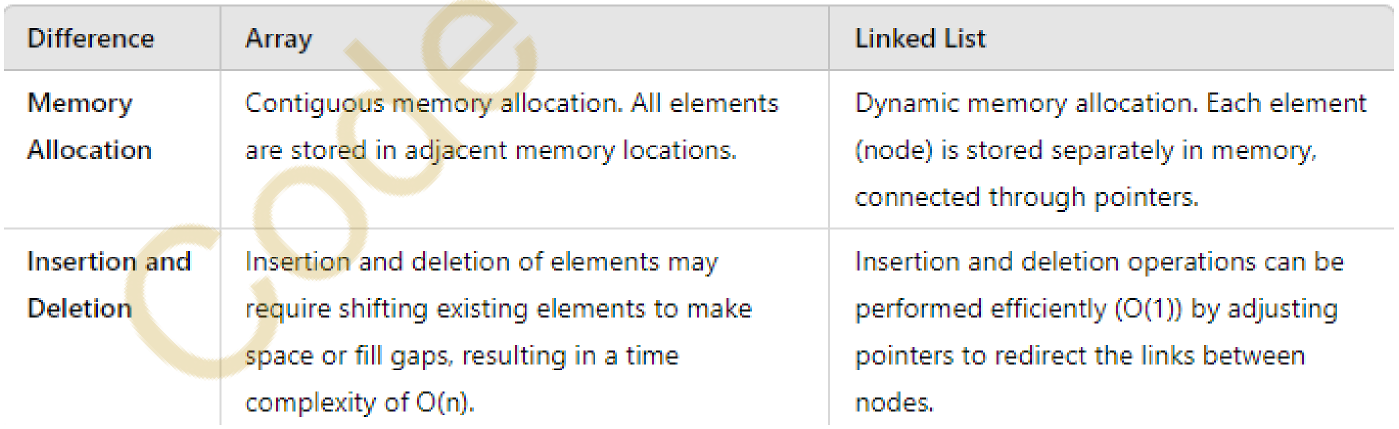
```cpp
#include <iostream>

int main() {
    int num = 5; // Regular variable storing an integer

    // Pointer variable storing the memory address of 'num'
    int* ptr = &num;

    std::cout << "Value of num: " << num << std::endl;
    std::cout << "Memory address of num: " << &num << std::endl;
    std::cout << "Value pointed to by ptr: " << *ptr << std::endl;
    std::cout << "Memory address stored in ptr: " << ptr << std::endl;

    return 0;
}
```

19. Explain Diff bw Structures and Union?



Difference	Structure	Union
Memory Allocation	Each member of a structure has its own memory space, and the size of the structure is the sum of the sizes of its members.	All members of a union share the same memory space, and the size of the union is equal to the size of its largest member.
Usage	Suitable for representing a collection of different types of data where each member has its significance.	Useful when you need to save memory by sharing memory space between different types of data, but only one member is used at a time.

20. Whats Difference Bw Array and Linked List



Difference	Array	Linked List
Memory Allocation	Contiguous memory allocation. All elements are stored in adjacent memory locations.	Dynamic memory allocation. Each element (node) is stored separately in memory, connected through pointers.
Insertion and Deletion	Insertion and deletion of elements may require shifting existing elements to make space or fill gaps, resulting in a time complexity of O(n).	Insertion and deletion operations can be performed efficiently (O(1)) by adjusting pointers to redirect the links between nodes.