Data Structures Lab

(BBM203 Software Practicum I)

Week 2: Java to C++ Transition Tutorial

https://web.cs.hacettepe.edu.tr/~bbm201/
https://piazza.com/hacettepe.edu.tr/fall2023/bbm203

Topics

- ➔ Introduction to Pointers
- ➔ Classes

Introduction to Pointers

What is a pointer?

A pointer is a variable that **stores a memory address** to another variable.

Pointers in C++ are declared using an * (asterisk) character.

```
int* my_integer_pointer;
```

When you declare a pointer, it points to nothing, or worse yet, it often points to a random slot.

What is a pointer?

Pointers can have multiple dimensions. We can have pointers pointing to other pointers.

```
int** my_integer_pointer;
```

We use the **&** (ampersand) operator to **get the address** of a variable.

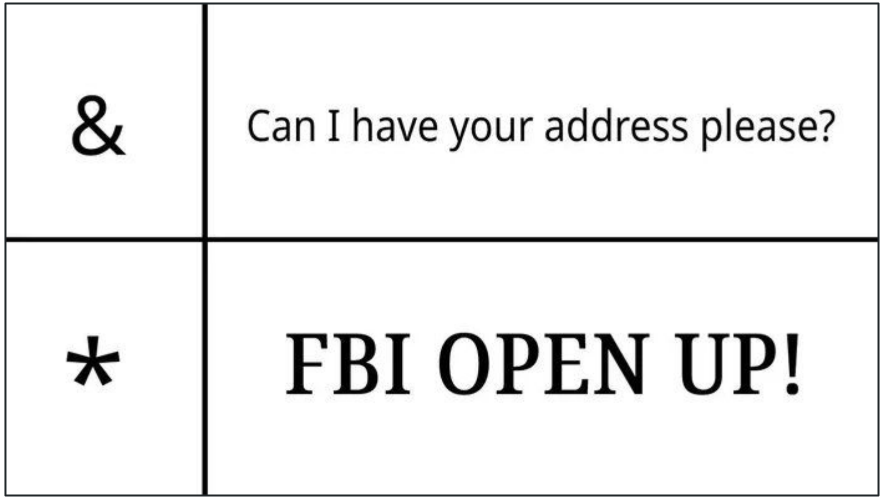
```
int* my_integer_pointer;
int my_integer = 1000;
my_integer_pointer = &my_integer;
```

What is a pointer?

To acquire the value the pointer is pointing to, we **dereference** the pointer by again using the * (asterisk) character.

```
int* my_integer_pointer;
int my_integer = 1000;
my_integer_pointer = &my_integer;
cout << *my_integer_pointer << endl;
```

```
> 1000
```



&	Can I have your address please?
*	FBI OPEN UP!

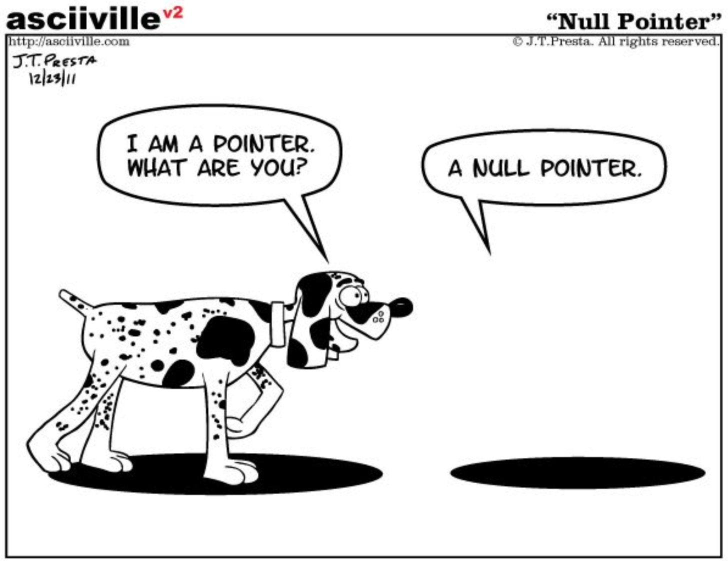
What is a pointer?

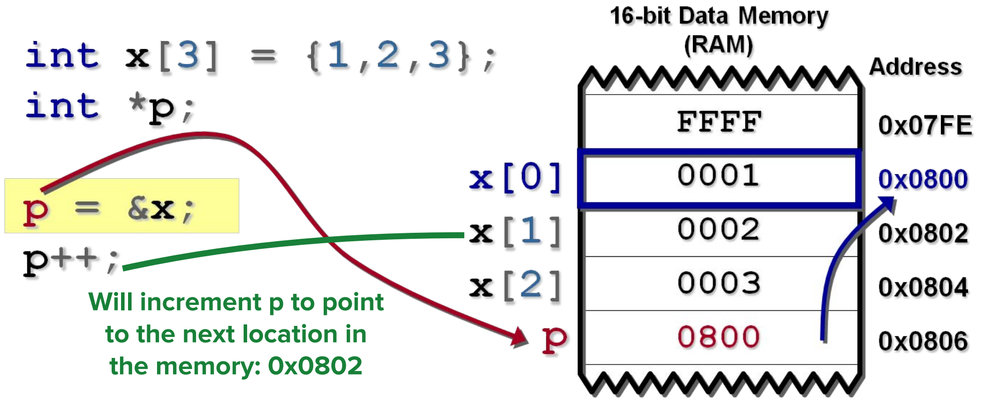
A pointer that doesn't point to any memory location can be set to a null pointer using the `nullptr` keyword.

```
int* pointer = nullptr;
```

Pointers can be used for arithmetic operations, moving through memory locations.

```
int arr[5] = {12, 23, 34, 41, 54};
int* arr_ptr = arr;
cout << "First: " << *arr_ptr << endl;
arr_ptr++;
cout << "Next: " << *arr_ptr << endl;
```




```
int x[3] = {1,2,3};
int *p;
p = &x;
p++;
```

Why pointers?

Pointers are used to dynamically allocate memory during program execution (on heap).

```
int* int_ptr = new int;
*int_ptr = 10;
delete int_ptr;
```

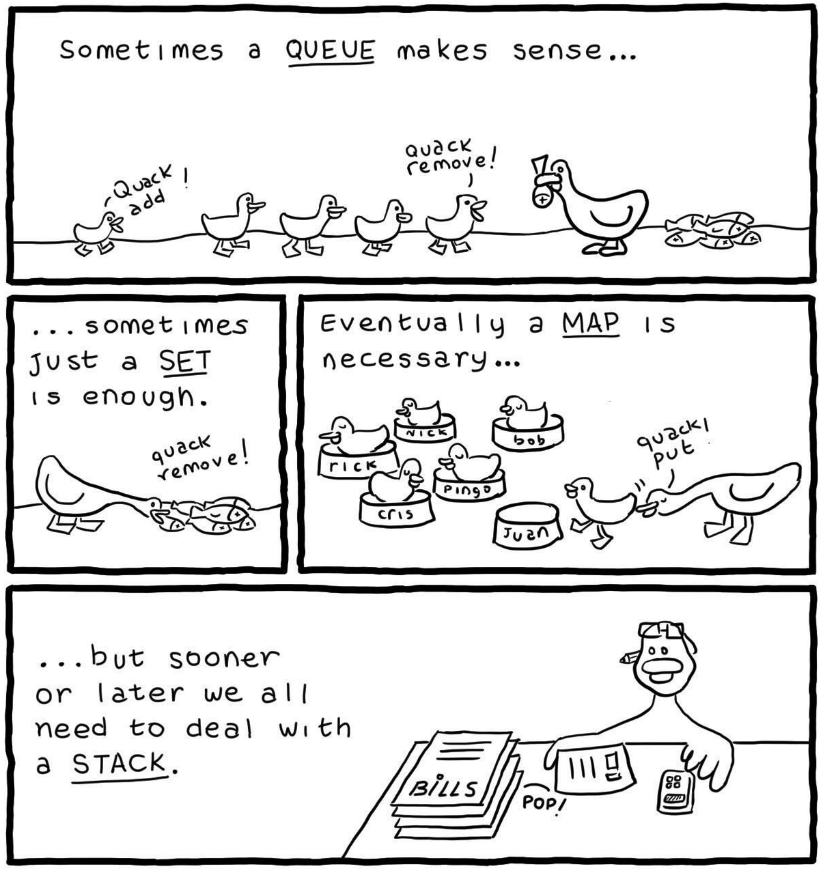
This is useful when you need memory that persists beyond a function's scope or when you want to allocate memory for an array or an object at runtime.

Ordered, on top of eachother!
No particular order!
Stack
Heap

Why pointers?

Pointers and dynamic memory allocation are especially useful for defining data structures such as linked lists, stacks, queues, etc.

```
class Node {
    public:
        int data;
        Node* next;
        Node(int val) {
            data = val;
            next = nullptr;
        }
};
```


Sometimes a QUEUE makes sense...
Quack add!
quack remove!
...sometimes just a SET is enough.
quack remove!
Eventually a MAP is necessary...
Nick
bob
rick
pingo
cris
Juan
quack! put
...but sooner or later we all need to deal with a STACK.
BILLS
POP!

Why pointers?

Pointers can be used to pass parameters by reference to functions, allowing the function to modify the original data.

```
void modify_value(int* value) {
    *value = 12;
}
int main() {
    int num = 10;
    modify_value(&num);
    cout << "Modified value: " << num << endl;
    return 0;
}
```

Why pointers?

There are two ways to pass arguments by reference in C++. The first one is by using pointers (on the left). The second one is again by using argument address, without declaring or dealing with pointers explicitly (on the right).

```
void modify_value(int* value) {
    *value = 12;
}
int main() {
    int num = 10;
    modify_value(&num);
    cout << "Modified value: " << num << endl;
    return 0;
}

> Modified value: 12
```

```
void modify_value(int& value) {
    value = 12;
}

int main() {
    int num = 10;
    modify_value(num);
    cout << "Modified value: " << num << endl;
    return 0;

> Modified value: 12
```

Why pointers?

The default behaviour in C++ is to pass by value.

```cpp
void modify_value(int value) {
    value = 12;
}

int main() {
    int num = 10;
    modify_value(num);
    cout << "Unmodified value: " << num << endl;
    return 0;
```

```
> Unmodified value: 10
```

Exercise: Visualize Stack Frame and Heap Memory

Let’s visualize static vs. dynamic memory allocation and pointers by inspecting stack frame and heap memory during program execution:

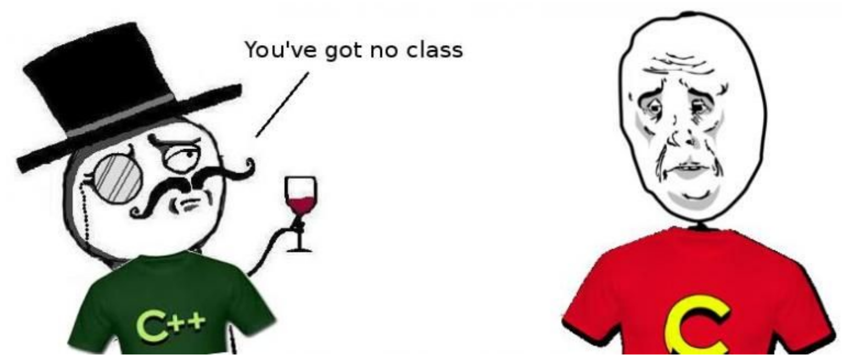
https://pythontutor.com/visualize.html

```cpp
#include <iostream>

using namespace std;

int main() {
    int x = 1;
    int* pointer = nullptr;
    pointer = &x;

    int arr[5] = {12, 23, 34, 41, 54};
    int* arr_ptr = arr;
    cout << "First: " << *arr_ptr << endl;
    arr_ptr++;
    cout << "Next: " << *arr_ptr << endl;

    int* heap_ptr = new int[5];

    heap_ptr[0] = 2;
    heap_ptr[1] = 34;
    heap_ptr[2] = 16;

    delete [] heap_ptr;

    return 0;
}
```

Classes

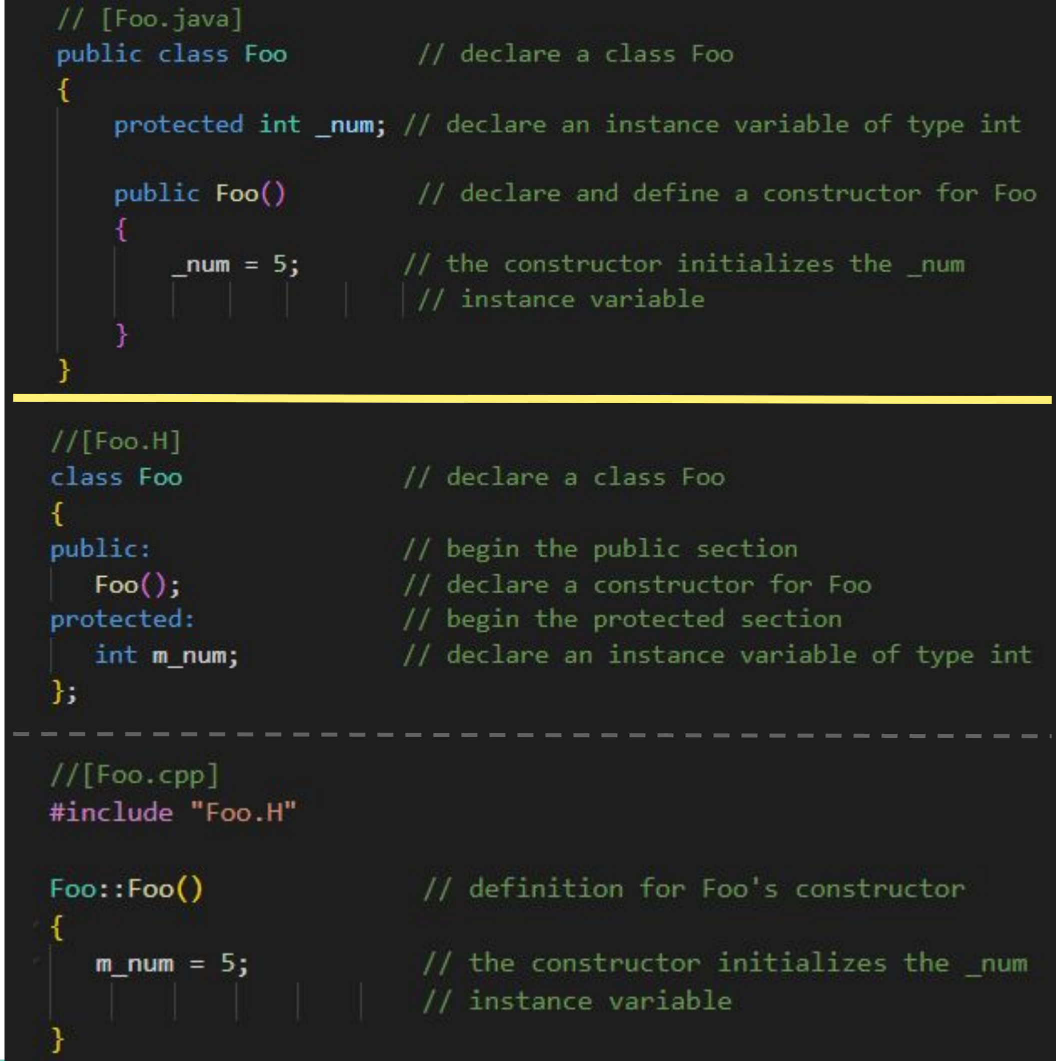
Classes

- A class is a **user-defined type or data structure** that has data and functions as its members whose access is governed by the three access specifiers private, protected or public.
- Defining a C++ class is similar to its Java equivalent, but there are a few differences.
- In C++, the class definition is separated into two files,
  - the **interface (also called the header or .h file)** and
  - the **implementation (the .cpp file)**.



Classes

- The interface **header** file defines the class "skeleton", listing the data members and the member functions including types and access modifiers.
- The **implementation** file contains the code for the functions.



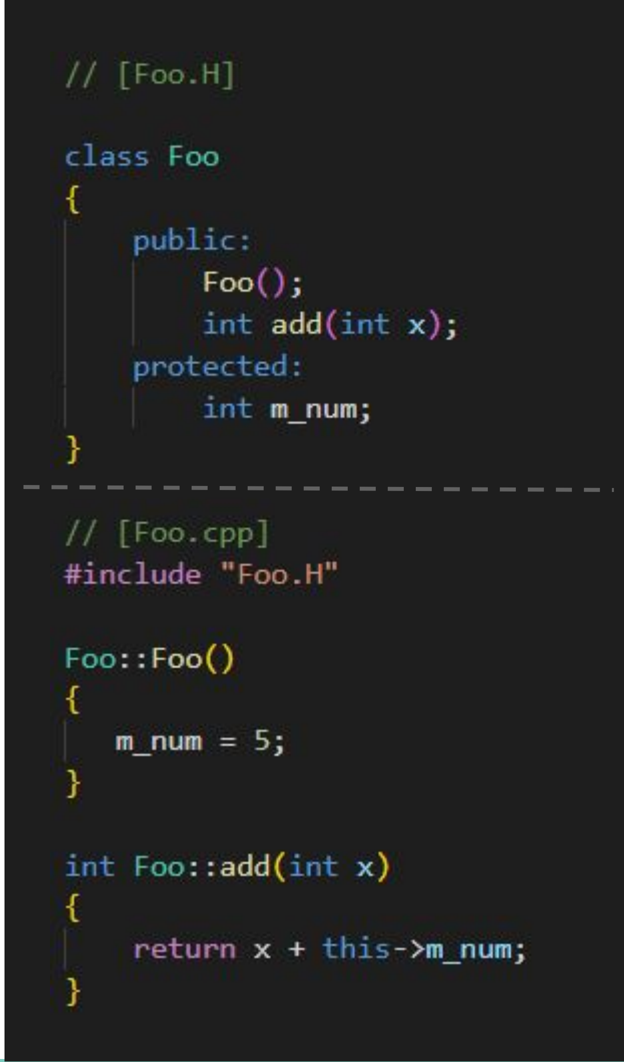
```java
// [Foo.java]
public class Foo          // declare a class Foo
{
    protected int _num; // declare an instance variable of type int

    public Foo()          // declare and define a constructor for Foo
    {
        _num = 5;         // the constructor initializes the _num
                          // instance variable
    }
}
```

```cpp
//[Foo.H]
class Foo                 // declare a class Foo
{
public:                   // begin the public section
    Foo();                // declare a constructor for Foo
protected:                // begin the protected section
    int m_num;            // declare an instance variable of type int
};
```

```cpp
//[Foo.cpp]
#include "Foo.H"

Foo::Foo()                 // definition for Foo's constructor
{
    m_num = 5;             // the constructor initializes the _num
                           // instance variable
}
```

Classes

- Note that each member function is prefixed by **`Classname::`**.
- The double colon is the **C++ scope resolution operator** and it tells the compiler that you are providing the implementation for the function named **`add`** within the class named **Foo**. If you forget the **`Foo::`**, the compiler will think you are defining a global function named **`add`**, which is not what we want in this code.
- This mistake will produce error messages about undeclared variable **`n_num`** since global functions have no "**`this`**" reference and there are no such instance variables in the scope.



```
// [Foo.H]

class Foo
{
    public:
        Foo();
        int add(int x);
    protected:
        int m_num;
}
```

```
// [Foo.cpp]
#include "Foo.H"

Foo::Foo()
{
   m_num = 5;
}

int Foo::add(int x)
{
    return x + this->m_num;
}
```

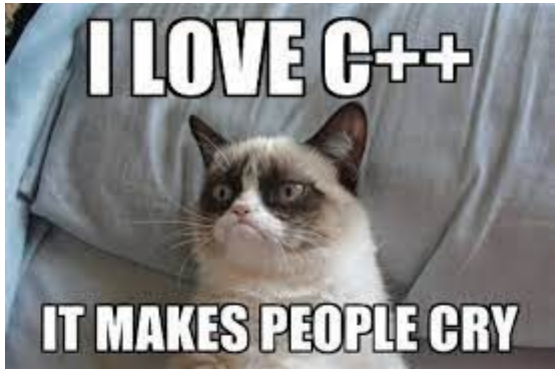
Classes

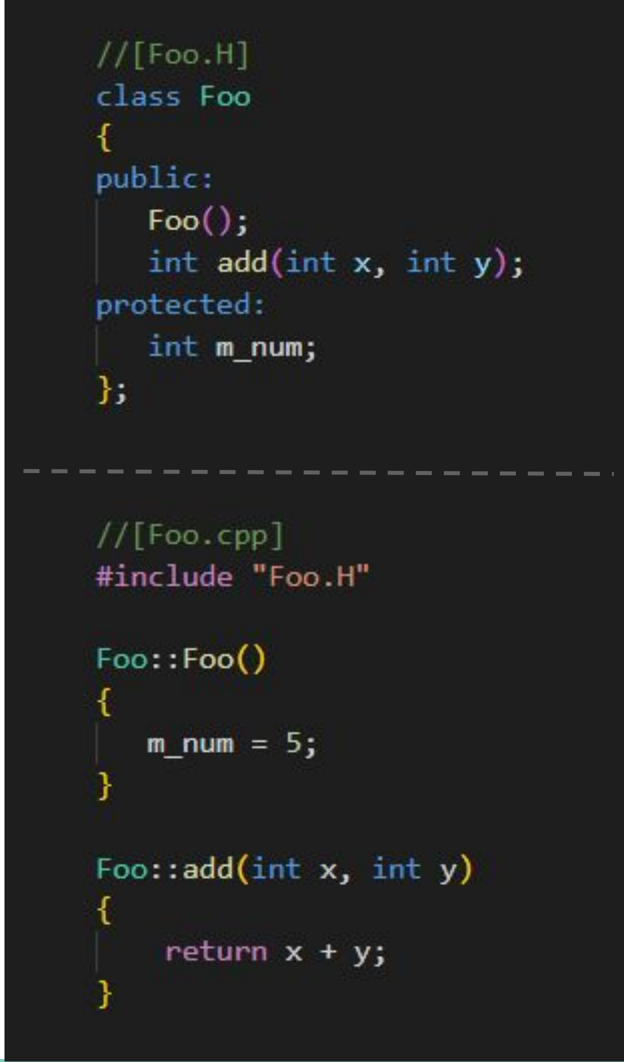
The `this` keyword in both C++ and Java is used to refer to the current instance of a class. However, there are differences:

- **Pointer vs. Reference**: In C++, this is a pointer, allowing for pointer-like operations. In Java, this is a reference, and you cannot perform pointer arithmetic or access memory addresses.
- **Usage with Member Access**: In both languages, this is used to access class members. However, in C++, you use `->` to access members, whereas in Java, you use `.`
- **Java Doesn't Allow Pointer-Like Operations**: Java doesn't allow the manipulation of pointers or memory addresses directly, making this more restricted in its use compared to C++.

Classes

Can you think of any advantages of separating the program declaration (header file) and definition (source file) into two files?

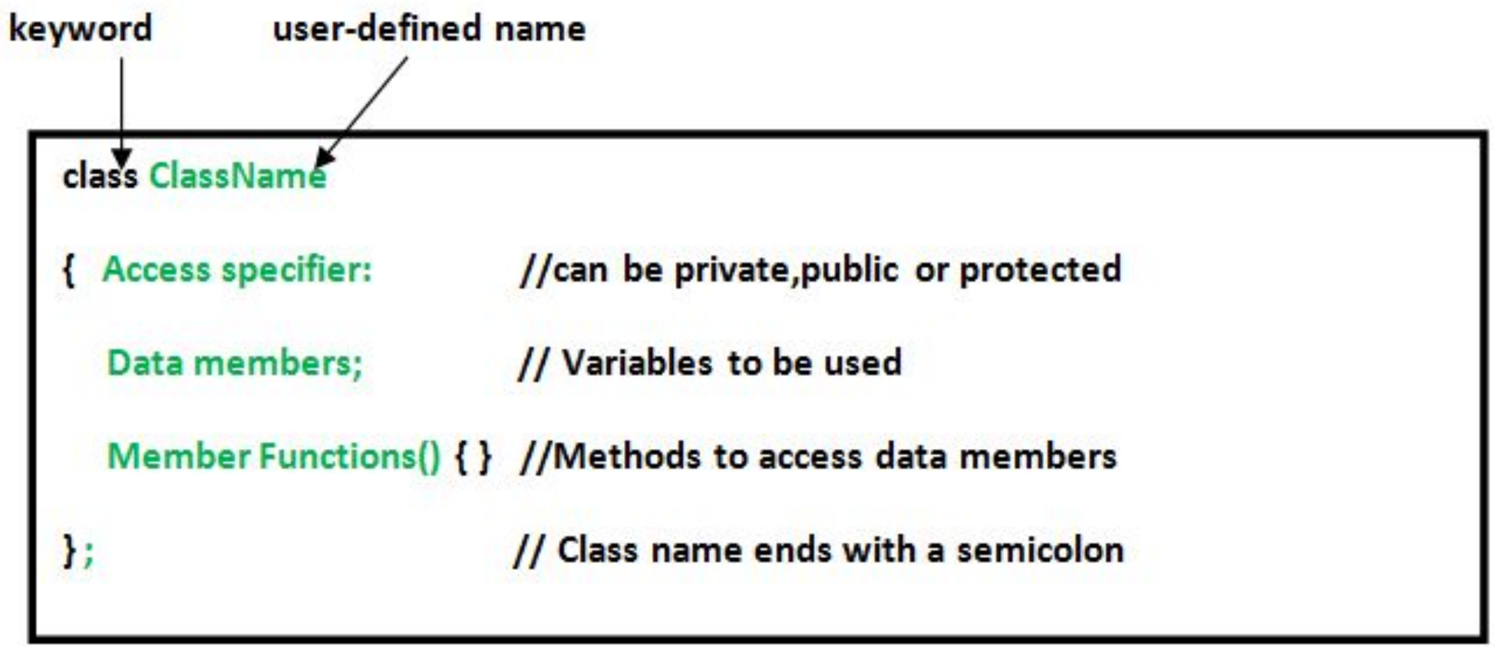




```
//[Foo.H]
class Foo
{
public:
    Foo();
    int add(int x, int y);
protected:
    int m_num;
};
```

```
//[Foo.cpp]
#include "Foo.H"

Foo::Foo()
{
    m_num = 5;
}

Foo::add(int x, int y)
{
     return x + y;
}
```

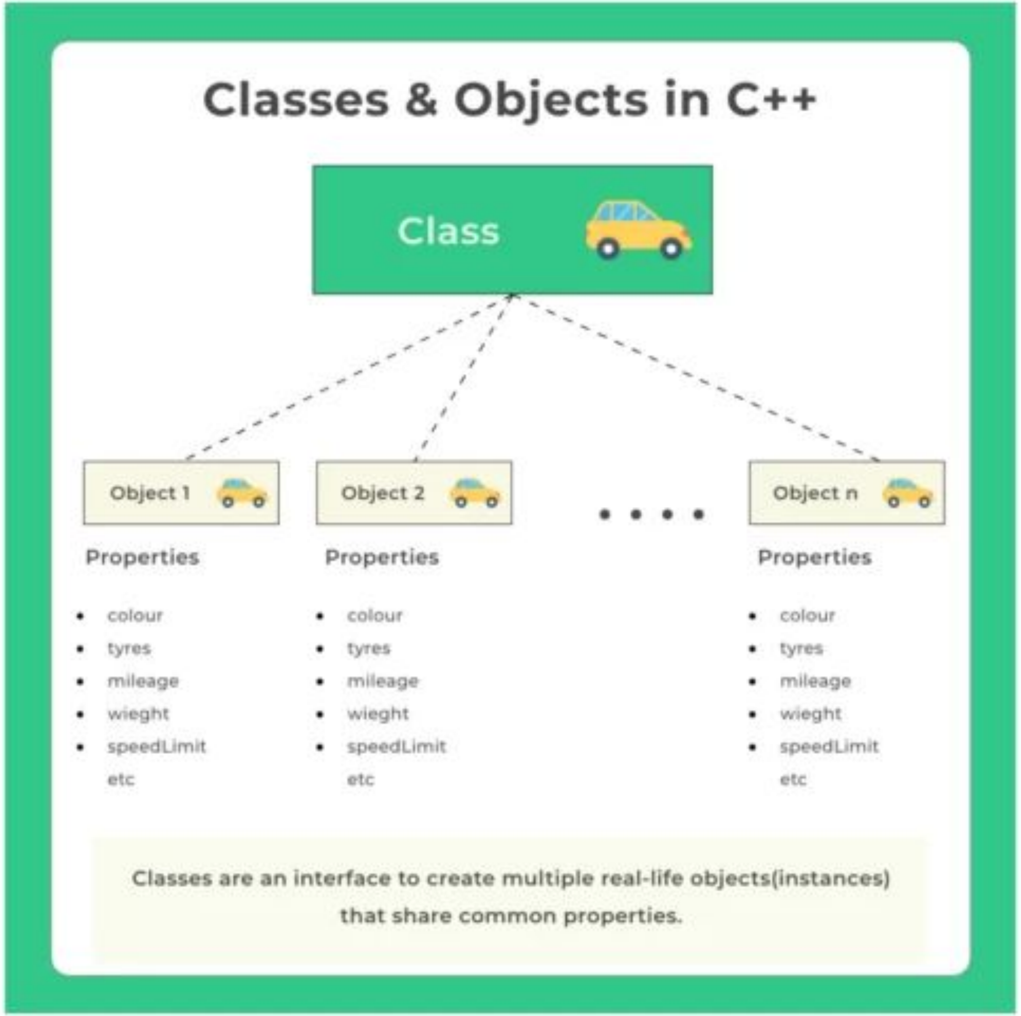
Classes



```
class ClassName

{  Access specifier:             //can be private,public or protected

   Data members;                 // Variables to be used

   Member Functions() { }        //Methods to access data members

};                               // Class name ends with a semicolon
```

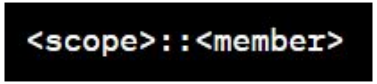
Classes

- An **object** is an instantiation of a class (same as in Java).


Classes & Objects in C++
Class
Object 1
Object 2
Object n
Properties
colour
tyres
mileage
wieght
speedLimit
etc
Classes are an interface to create multiple real-life objects(instances) that share common properties.

Classes: Scope Resolution Operator (::)

- The **Scope Resolution Operator (::)** in C++ allows access to a member (variable or function) that is part of a class, namespace, or global scope.

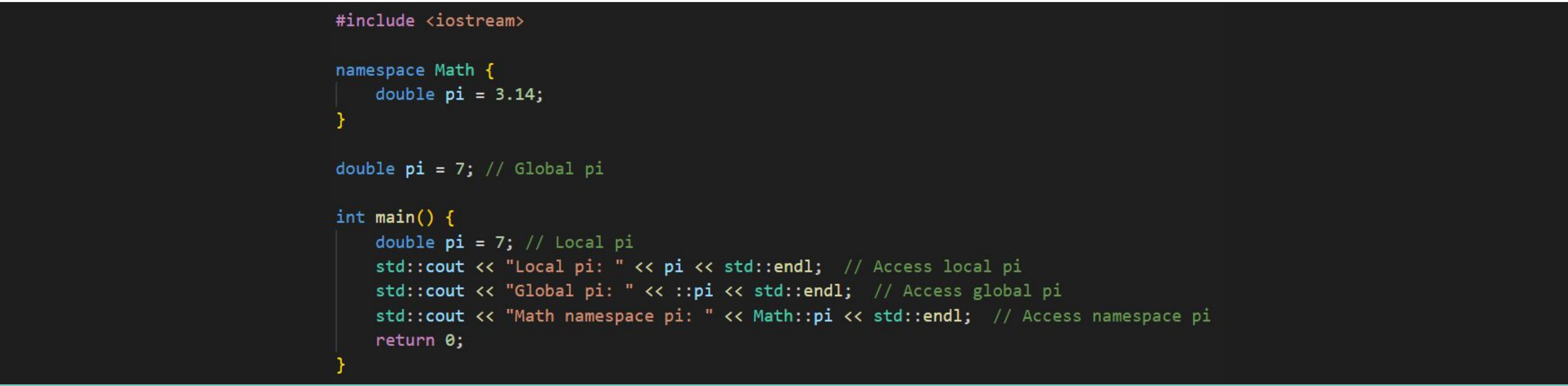


```
<scope>::<member>
```

- **Accessing Class Members**:
  - Allows access to class members outside the class definition: `ClassName::member`
- **Accessing Namespace Members**:
  - Provides access to members within a specific namespace: `namespace::member`
- **Accessing Global Variables**:
  - Accesses global variables defined outside of any class or namespace: `::globalVariable`

Classes: Scope Resolution Operator (::)

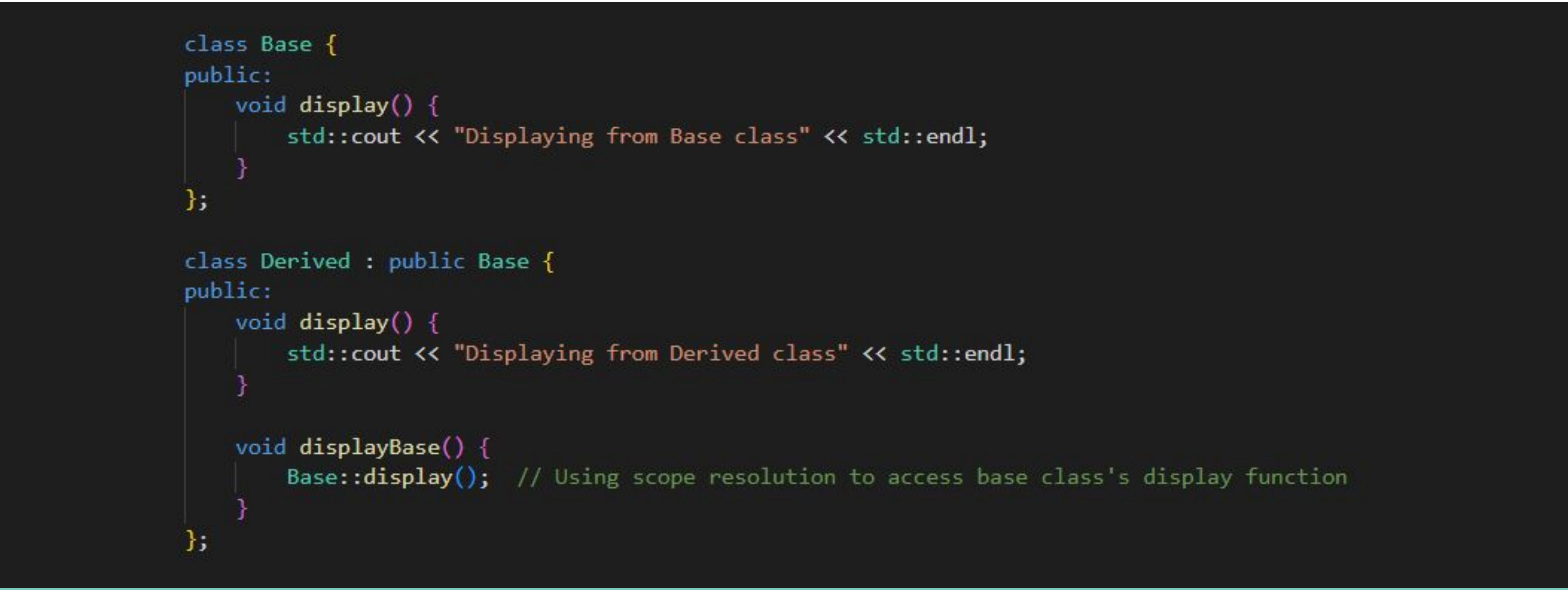
- **Key points**:
  - Enables explicit referencing of members in different scopes.
  - Resolves ambiguity when there are similar names in different scopes.
  - Enhances code readability and avoids naming conflicts.



```cpp
#include <iostream>

namespace Math {
    double pi = 3.14;
}

double pi = 7; // Global pi

int main() {
    double pi = 7; // Local pi
    std::cout << "Local pi: " << pi << std::endl;  // Access local pi
    std::cout << "Global pi: " << ::pi << std::endl;  // Access global pi
    std::cout << "Math namespace pi: " << Math::pi << std::endl;  // Access namespace pi
    return 0;
}
```

Classes: Scope Resolution Operator (::)

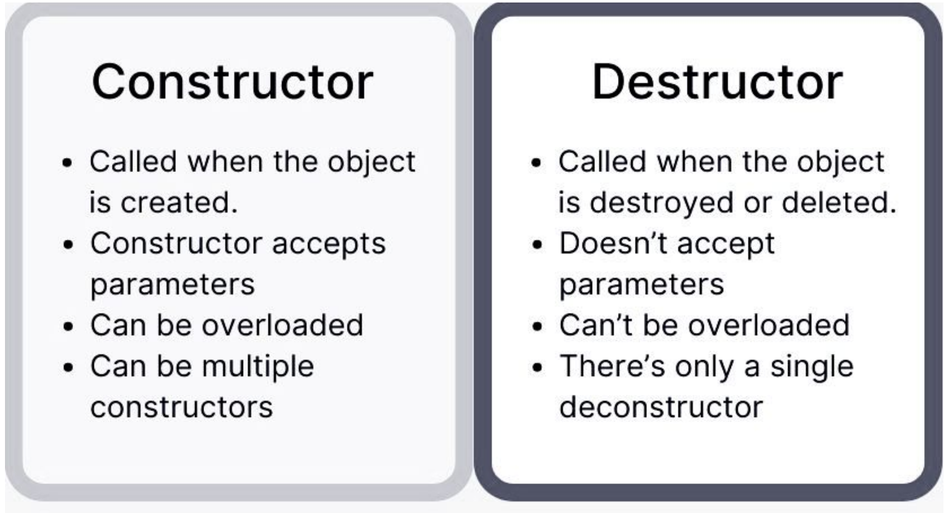
- Specifying scope in derived classes when overriding the superclass function:



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

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

    void displayBase() {
        Base::display();  // Using scope resolution to access base class's display function
    }
};
```

Classes: Understanding Essentials in C++

- **Class Constructors**: creating class objects
- **Operator Overloading**: intuitive use of operators with user-defined types
- **Destructors**: memory cleanup



Constructor

- Called when the object is created.
- Constructor accepts parameters
- Can be overloaded
- Can be multiple constructors

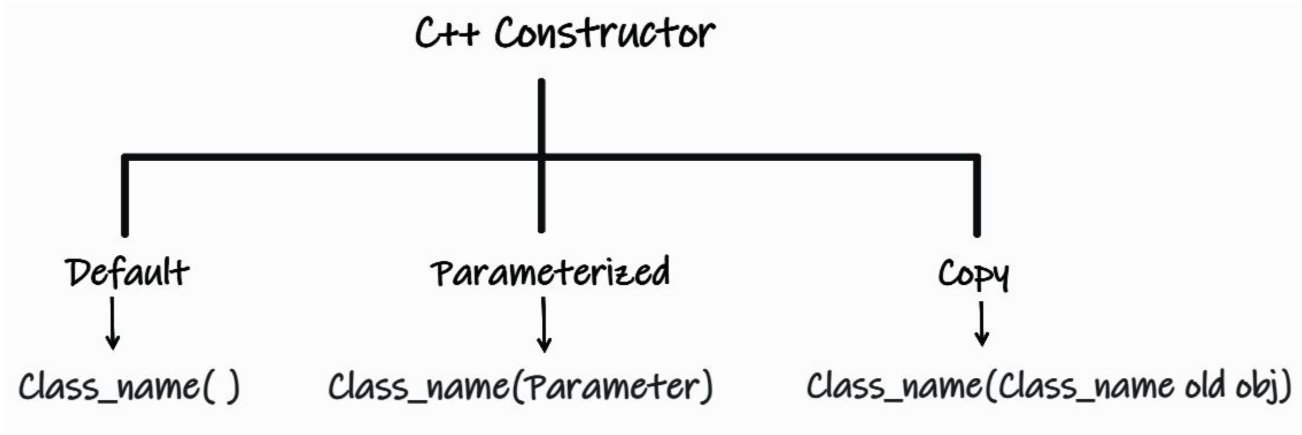
Destructor

- Called when the object is destroyed or deleted.
- Doesn't accept parameters
- Can't be overloaded
- There's only a single deconstructor

Classes: Constructor

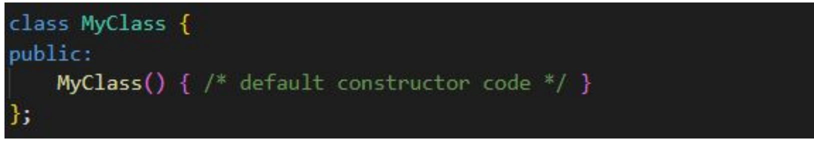
Class Constructor: a special member function that is automatically invoked when an object of the class is created, with the purpose of initialize class data members, allocating resources, and setting up the object's initial state.

- Types: Default, parameterized, copy constructor.



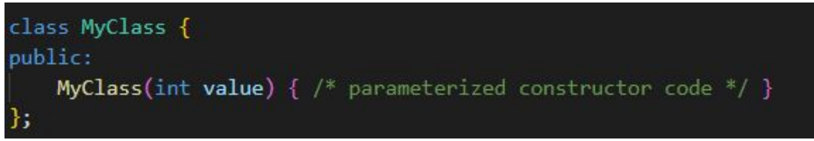
Classes: Constructor

- **Default Constructor**: automatically called when an object is created with no arguments. Sets up a default object state.



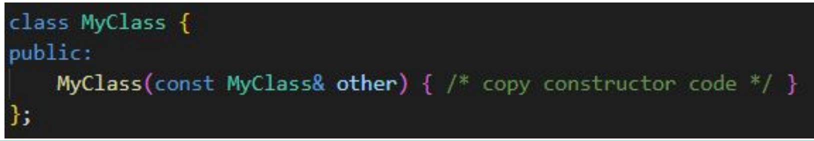
```
class MyClass {
public:
    MyClass() { /* default constructor code */ }
};
```

- **Parameterized Constructor**: accepts parameters to initialize class members during object creation based on the provided values.



```
class MyClass {
public:
    MyClass(int value) { /* parameterized constructor code */ }
};
```

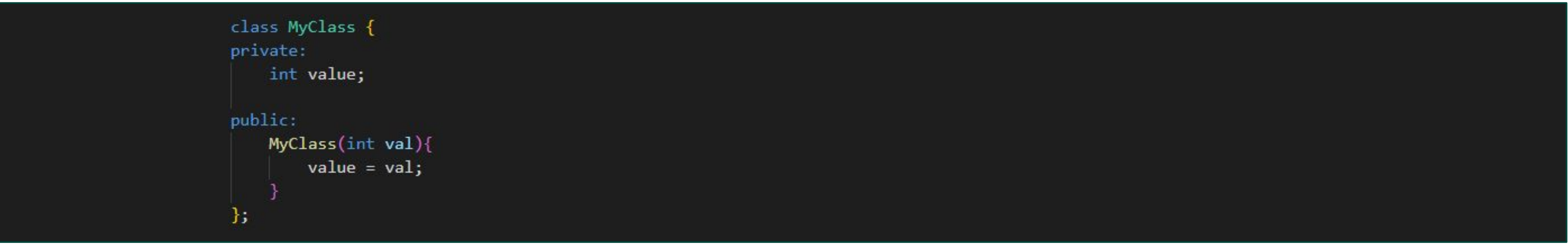
- **Copy Constructor**: creates a new object as a copy of an existing object of the same class (same state as an existing object).



```
class MyClass {
public:
    MyClass(const MyClass& other) { /* copy constructor code */ }
};
```

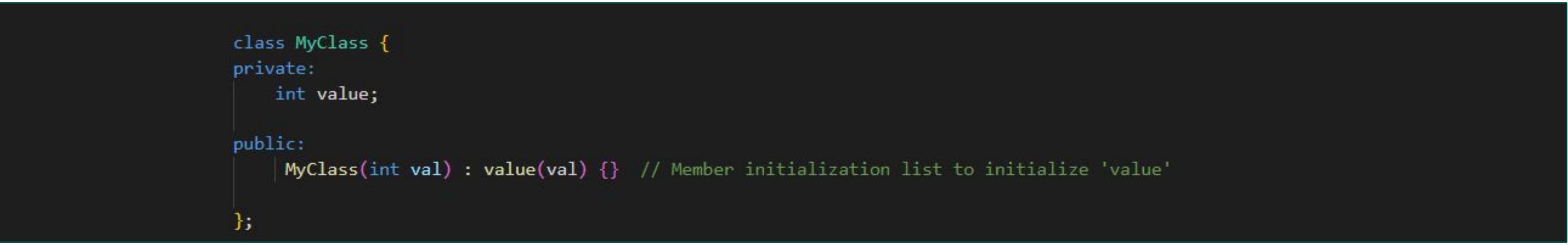
Classes: Constructor

- **Constructors**: different ways to initialize class members
  - **In the constructor body:**



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

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

  - **Using member initialization list:**



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

public:
    MyClass(int val) : value(val) {}  // Member initialization list to initialize 'value'

};
```

Classes: Constructor

- **Constructors**: different ways to initialize class members
  - **In the constructor body:**
  - **Using member initialization list:**

```cpp
class Book {
    private:
        string title;
        string ISBN;
        double price;
    public:

        Book(string title, string ISBN, double price) {
            this->title = title;
            this->ISBN = ISBN;
            this->price = price;
        }
};
```

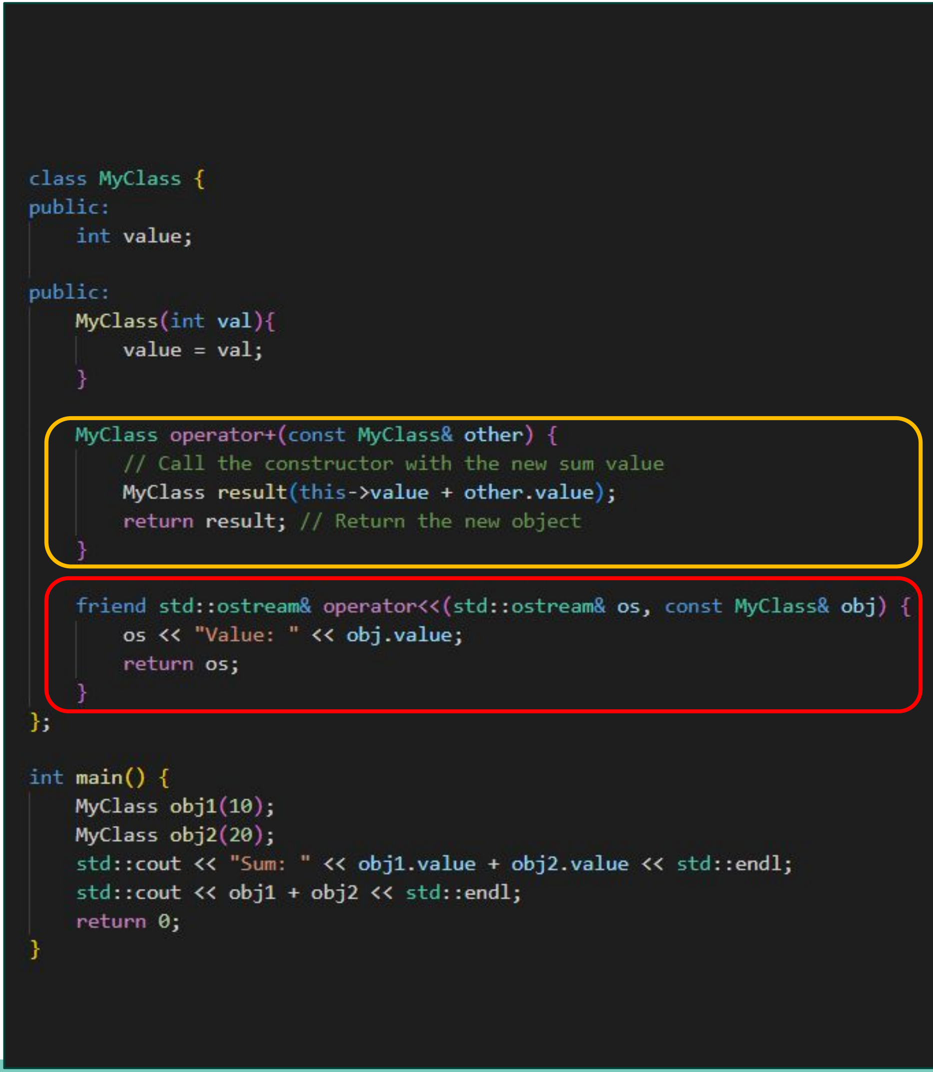
```cpp
class Book {
    private:
        string title;
        string ISBN;
        double price;
    public:

        Book(string title, string ISBN, double price) :
        title(title), ISBN(ISBN), price(price) {}
};
```

Classes: Operator Overloading

Operator Overloading: allows defining custom behaviors for operators based on the operands' types with the purpose of enabling intuitive use of operators with user-defined types.

- Examples: Overloading **+ for addition** or **<< for custom output** of class objects.

`friend` keyword is used to grant non-member functions or external classes access to the private and protected members of a class. When overloading the << operator for a class, we typically want to access the private or protected members of the class (e.g., data members) to print their values. In this example it is not necessary.



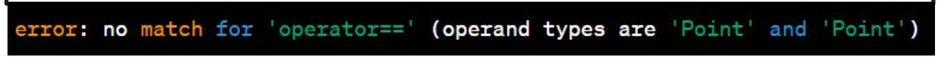
```cpp
class MyClass {
public:
    int value;

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

    MyClass operator+(const MyClass& other) {
        // Call the constructor with the new sum value
        MyClass result(this->value + other.value);
        return result; // Return the new object
    }

    friend std::ostream& operator<<(std::ostream& os, const MyClass& obj) {
        os << "Value: " << obj.value;
        return os;
    }
};

int main() {
    MyClass obj1(10);
    MyClass obj2(20);
    std::cout << "Sum: " << obj1.value + obj2.value << std::endl;
    std::cout << obj1 + obj2 << std::endl;
    return 0;
}
```

Classes: Operator Overloading

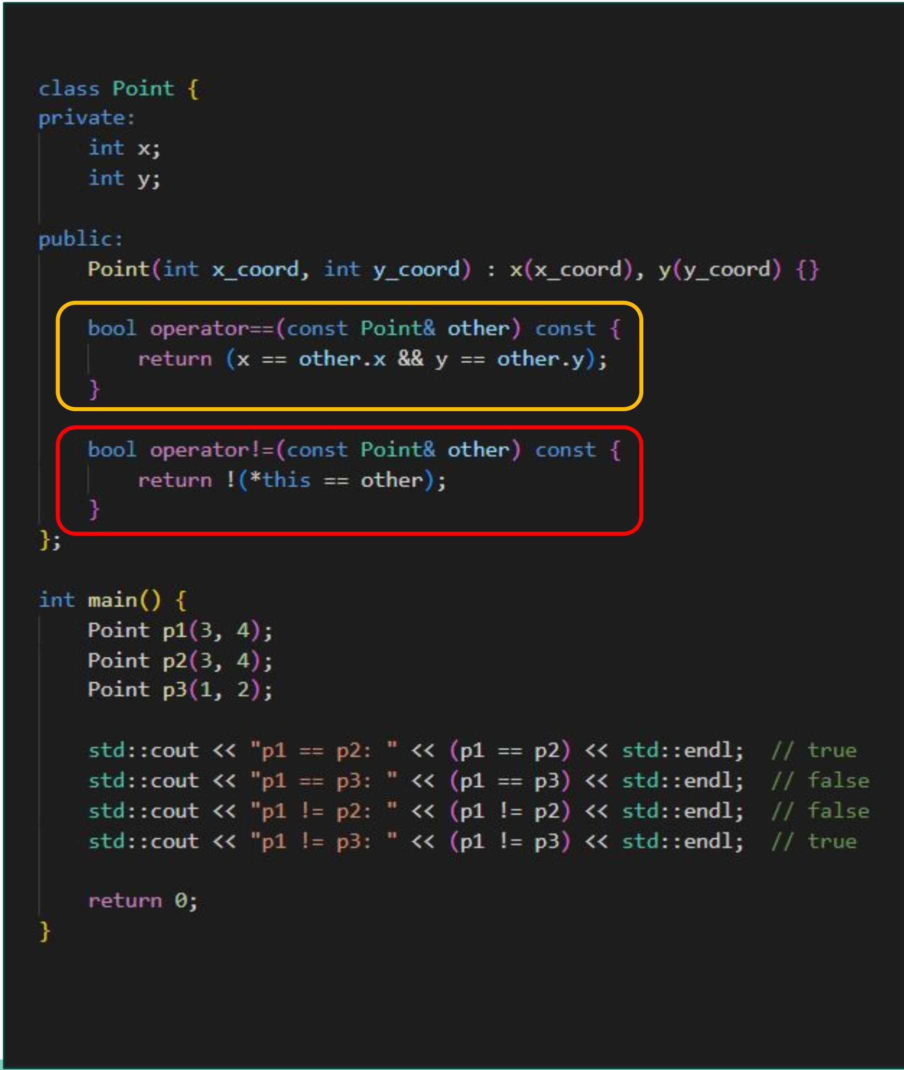
Operator Overloading another example:

- overloading the comparison operators (**==**, **!=**) for a simple coordinate point class:

If you do not overload the operators for your custom class, attempting to use them on objects of that class will result in a **compilation error**. The compiler will not know how to perform the operation if the standard operators are not defined for your class.



```
error: no match for 'operator==' (operand types are 'Point' and 'Point')
```

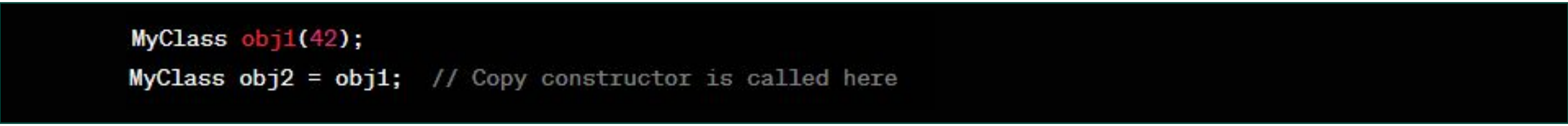


```cpp
class Point {
private:
    int x;
    int y;

public:
    Point(int x_coord, int y_coord) : x(x_coord), y(y_coord) {}

    bool operator==(const Point& other) const {
        return (x == other.x && y == other.y);
    }

    bool operator!=(const Point& other) const {
        return !(*this == other);
    }
};

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

    std::cout << "p1 == p2: " << (p1 == p2) << std::endl;  // true
    std::cout << "p1 == p3: " << (p1 == p3) << std::endl;  // false
    std::cout << "p1 != p2: " << (p1 != p2) << std::endl;  // false
    std::cout << "p1 != p3: " << (p1 != p3) << std::endl;  // true

    return 0;
}
```

C++ Copy Constructor vs. Assignment Operator

Copy Constructor

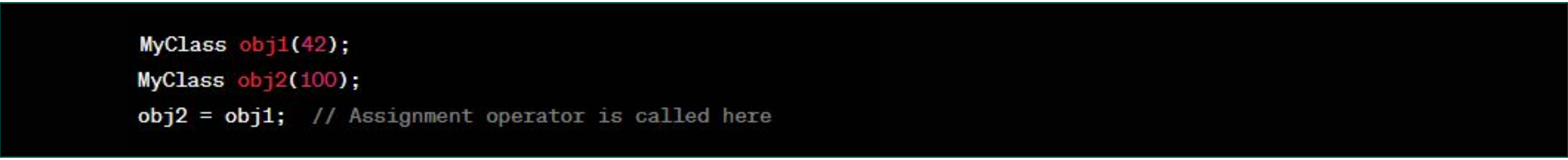
- Creates a new object as a copy of an existing object during initialization.
- Invoked when object is created and initialized with an existing object.



```
MyClass obj1(42);
MyClass obj2 = obj1;  // Copy constructor is called here
```

Assignment Operator (operator=)

- Assigns the value of an existing object to another existing object.
- Invoked when an already initialized object is assigned a new value.



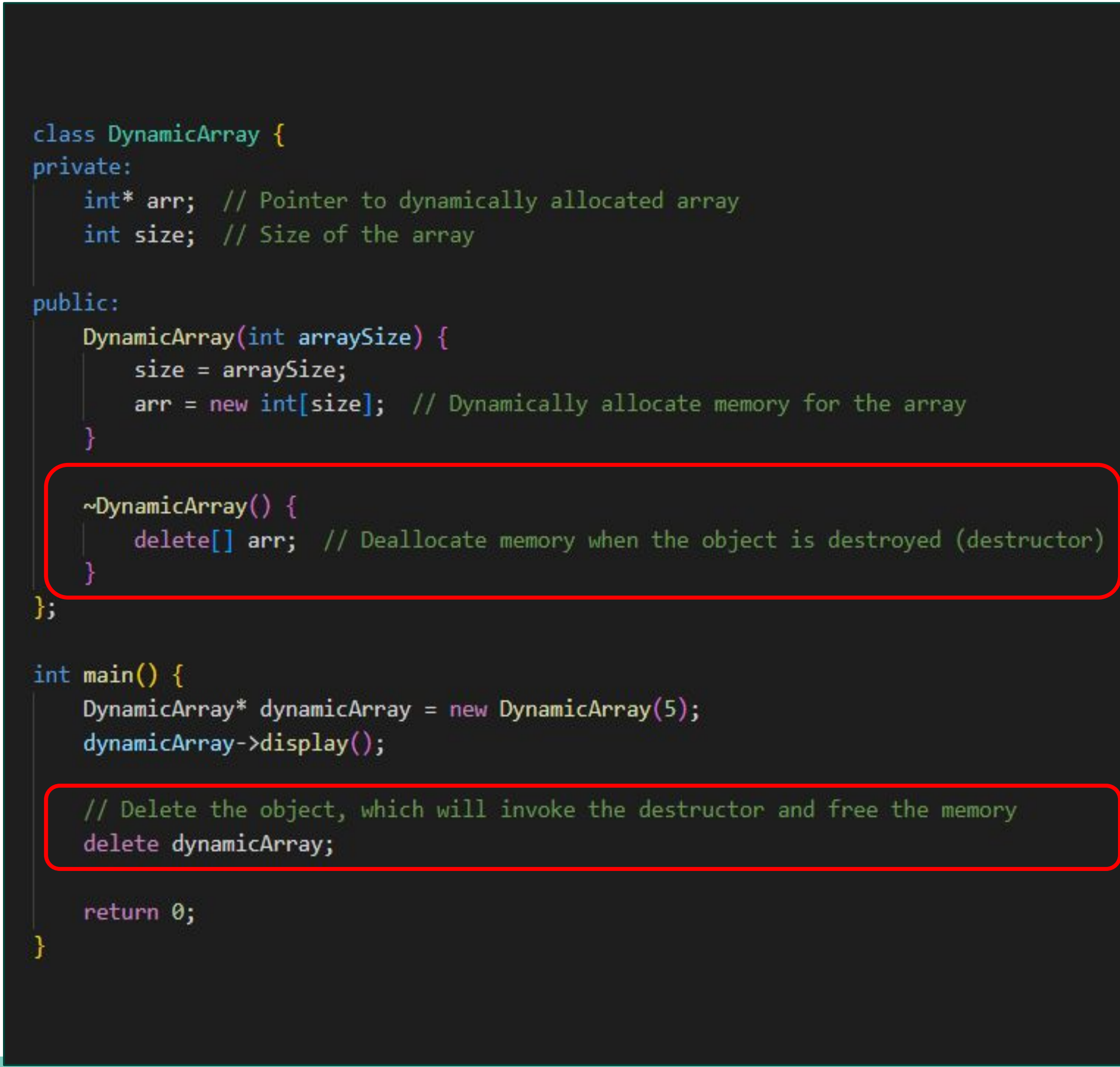
```
MyClass obj1(42);
MyClass obj2(100);
obj2 = obj1;  // Assignment operator is called here
```

Key Differences:

- Copy constructor is used for object creation and initialization.
- Assignment operator modifies an already existing object's value.
- Different use cases and invocation scenarios.

Classes: Destructor

Destructor: a special member function that is automatically invoked when an object goes out of scope or is explicitly deleted. It releases resources, cleans up memory, and performs any other necessary cleanup for the object.



```
class DynamicArray {
private:
    int* arr;   // Pointer to dynamically allocated array
    int size;   // Size of the array

public:
    DynamicArray(int arraySize) {
        size = arraySize;
        arr = new int[size];  // Dynamically allocate memory for the array
    }

    ~DynamicArray() {
        delete[] arr;  // Deallocate memory when the object is destroyed (destructor)
    }
};

int main() {
    DynamicArray* dynamicArray = new DynamicArray(5);
    dynamicArray->display();

    // Delete the object, which will invoke the destructor and free the memory
    delete dynamicArray;

    return 0;
}
```

Classes: Essentials Practice Example

Create a C++ class called `Counter` that keeps track of a count and implements the following features:

Implement three types of constructors:

- Default constructor (initialize count to 0)
- Parameterized constructor (initialize count to a given value)
- Copy constructor (create a copy of an existing Counter object)

Operator Overloading:

- Overload the post-increment ++ operator (count++) to increment the count by 1.
- Overload the << operator to display the count.

Destructor:

- Implement a destructor to free the used heap memory and display a message indicating the destruction of an object. Ensure proper memory management and handle any potential issues related to object destruction.

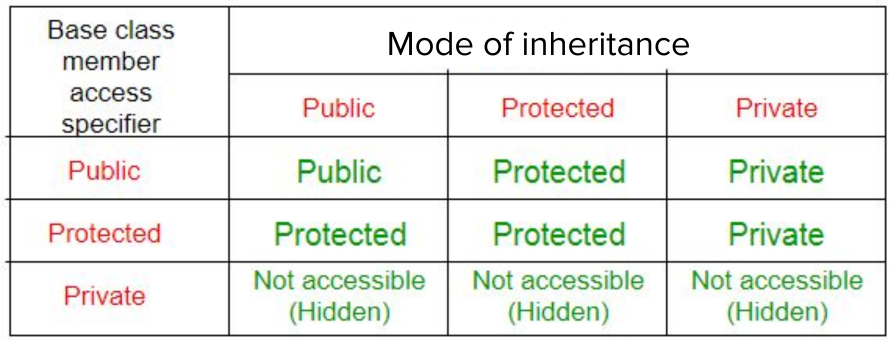
Testing: Create multiple `Counter` objects, perform increment operations, and display the count.

Create both the header (Counter.H) and implementation files (Counter.cpp). Test in main.cpp

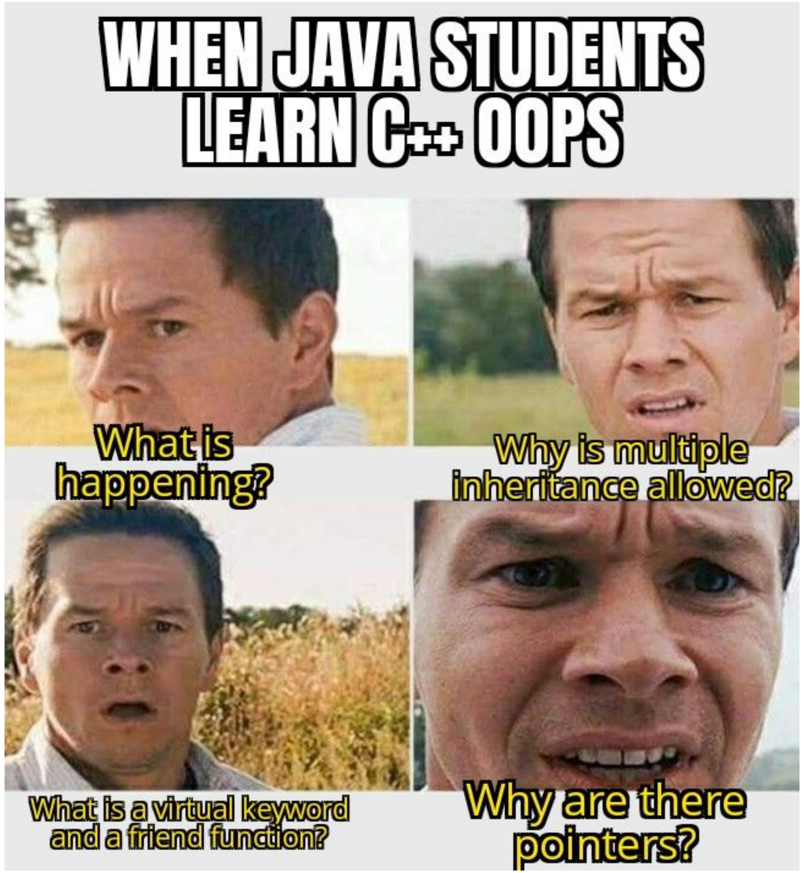
Classes: Inheritance

C++ supports multiple **modes** and multiple **types** of inheritance.

```
class subclass_name : access_mode base_class_name
{
    //body of subclass
};
```



Base class member access specifier	Mode of inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)


WHEN JAVA STUDENTS LEARN C++ OOPS
What is happening?
Why is multiple inheritance allowed?
What is a virtual keyword and a friend function?
Why are there pointers?

Modes of Inheritance: Public Inheritance

Public inheritance makes **public members** of the base class **public in the derived class**, and the **protected members** of the base class remain **protected** in the derived class.

Private members of the base class are **never** visible in the derived class.

```
class Base {
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};

class PublicDerived : public Base {
    // x is public
    // y is protected
    // z is not accessible
    // (private to PublicDerived)
};
```

Modes of Inheritance: Private Inheritance

Private inheritance makes the **public** and **protected** members of the base class **private** in the derived class.

```
class Base {
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};

class PublicDerived : private Base {
    // x is private
    // y is private
    // z is not accessible
    // (private to PublicDerived)
};
```

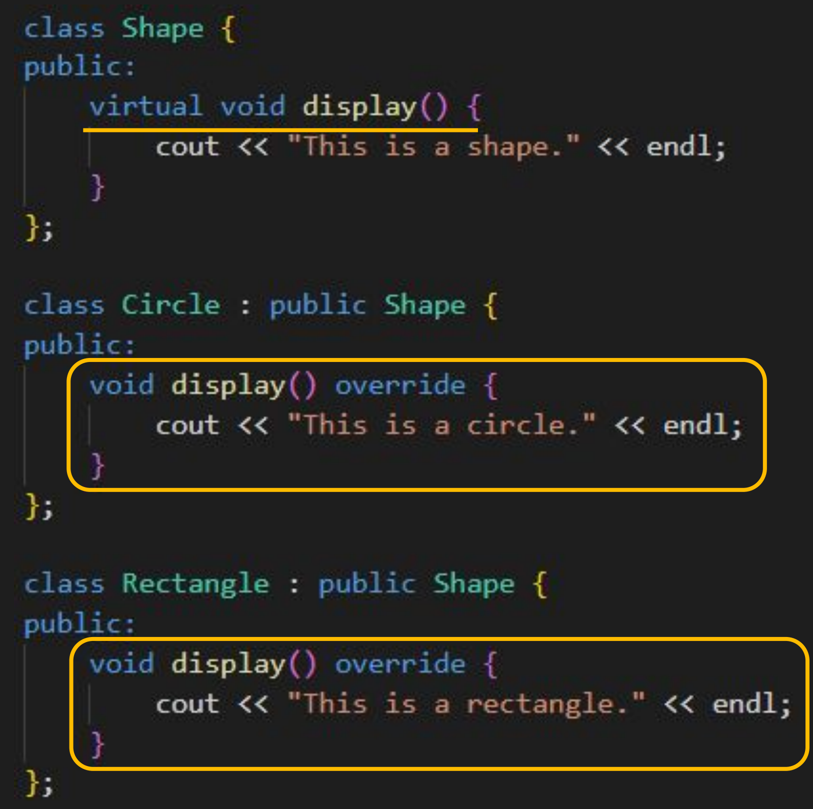
Modes of Inheritance: Protected Inheritance

Protected inheritance makes the **public** and **protected** members of the base class **protected** in the derived class.

```
class Base {
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};

class PublicDerived : protected Base {
    // x is protected
    // y is protected
    // z is not accessible
    // (private to PublicDerived)
};
```

Classes: Functions Overriding and Scope

- Function overriding occurs when a derived class defines a method with the same signature as a method in its base class.
- **The derived class provides a specific implementation, effectively replacing the base class's behavior.**



```cpp
class Shape {
public:
    virtual void display() {
        cout << "This is a shape." << endl;
    }
};

class Circle : public Shape {
public:
    void display() override {
        cout << "This is a circle." << endl;
    }
};

class Rectangle : public Shape {
public:
    void display() override {
        cout << "This is a rectangle." << endl;
    }
};
```

Classes: Virtual Functions

- ***Virtual functions*** are **late-bound** (runtime-bound) functions that are used to achieve **polymorphism**.
- Function overriding is only possible with virtual functions.
- To declare a function as virtual, the **virtual** keyword is used.
- **Dynamic initialization** must be used to utilize virtual functions as they are **bound in the runtime.**

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

class Base {
    public:
        virtual void print() {
            cout << "print base class\n";
        }
};

class Derived : public Base {
    public:
        void print() {
            cout << "print derived class\n";
        }
        void print_base() {
            Base::print();
        }
};

int main() {
    Base *b = new Base();
    Derived *d = new Derived();

    b->print();
    d->print();
    d->print_base();

    delete b;
    delete d;
}
```

Console output:

```
>print base class
>print derived class
>print base class
```

Classes: Virtual Functions

- The first part demonstrates **static binding**, where the objects are treated as their static (declared) types and lose their derived class properties when stored in an array of the base class.
- The second part demonstrates **dynamic binding** and **polymorphism**, where objects are created **dynamically** and their **virtual functions are called through pointers**, resulting in the correct function being executed based on the actual object's type.

```
int main() {

    // This does NOT work!!

    Base b_static;
    Derived d_static;

    Base * array = new Base[2];
    array[0] = b_static;
    array[1] = d_static;

    array[0].print();
    array[1].print();
```

```
Console output:
------------------------------------
>print base class
>print base class
```

```
    // Correct polymorphism example

    Base * b_dynamic = new Base();
    Derived * d_dynamic = new Derived();

    Base ** array_dynamic = new Base*[2];
    array_dynamic[0] = b_dynamic;
    array_dynamic[1] = d_dynamic;

    array_dynamic[0]->print();
    array_dynamic[1]->print();
}
```

```
Console output:
------------------------------------
>print base class
>print derived class
```

Classes: Inheritance Practice Example

Create a hierarchy of shapes using inheritance and implement function overriding and virtual functions. Create a base class called **Shape** with a **virtual** function **`calculateArea()`** to calculate the area of a shape. Derive three shapes from the base class:

- **`Rectangle`**: Implement **`calculateArea()`** that calculates the area of a rectangle.
- **`Circle`**: Implement **`calculateArea()`** that calculates the area of a circle.
- **`Triangle`**: Implement **`calculateArea()`** that calculates the area of a triangle.

Demonstrate function overriding by implementing versions of the **`calculateArea()`** function in each derived class that accept different parameters (e.g., for rectangles: **length** and **breadth**; for circles: **radius**; for triangles: **base** and **height**).

Testing: In the **`main.cpp`**, create objects of each shape and demonstrate the calculation of their areas using the appropriate **`calculateArea()`** functions.

ITS OVER
WE DID IT

More Useful Resources For Practice:

- https://www.w3resource.com/cpp-exercises/basic/index.php
- https://www.w3schools.com/cpp/cpp_exercises.asp
- https://www.hackerrank.com/domains/cpp
- https://algoleague.com/

Questions?