Object Oriented Programming with C++

Tutorial 1

Introduction to Object-Oriented Programming

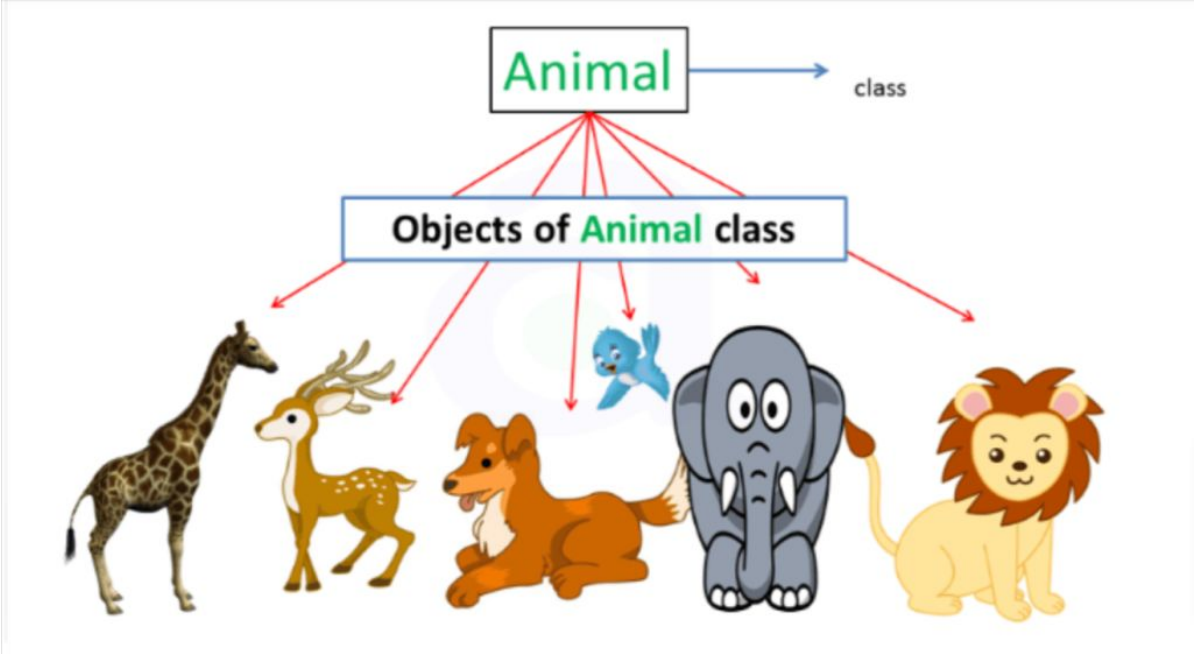
Why Learn OOP?

- **Models real-world problems** using objects and classes
- **Encourages code reusability** through inheritance
- **Protects data** with encapsulation and access control
- **Makes code easier to maintain** and extend
- **Frequently asked in interviews** for both internships and jobs

Introduction to Object-Oriented Programming

What is OOP?

Object-Oriented Programming is a programming paradigm based on the concept of "objects" that contain attributes (data) and behaviours (functions).



Introduction to Object-Oriented Programming

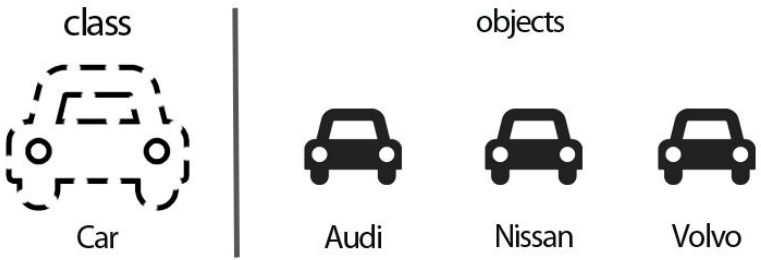
What is OOP?

Object-Oriented Programming is a programming paradigm based on the concept of "objects" that contain data and functions.

Organizes software design around objects rather than functions and logic

Enables modular, reusable, and maintainable code

Represents real-world entities as software objects



Key OOP Concepts

- **Encapsulation:** Bundling data and methods that operate on that data
- **Inheritance:** A new class of object derives properties and characteristics from another class
- **Polymorphism:** Objects can take different forms depending on context
- **Abstraction :** Hiding complex implementation details

Data Types in C++

Fundamental Data Types

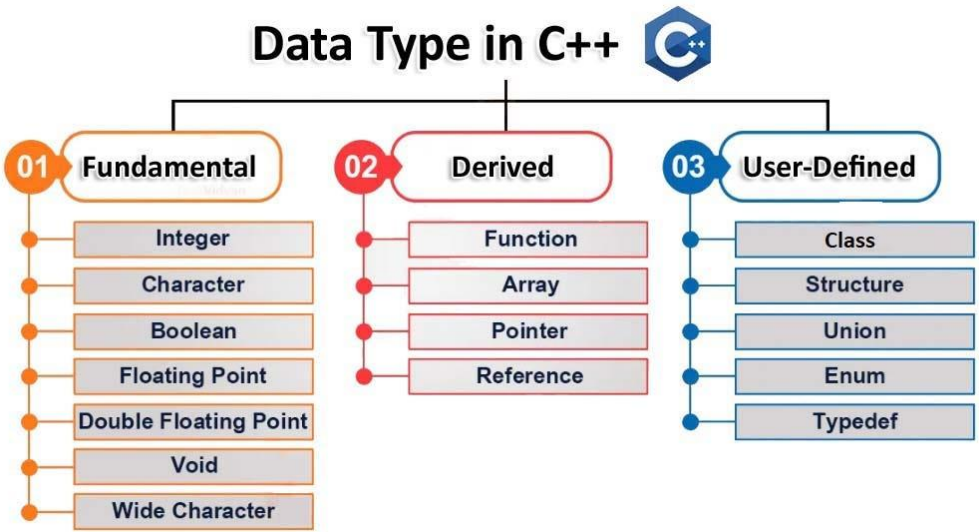
- **Integer Types:** int, short, long, long long
- **Floating-Point Types:** float, double, long double
- **Character Types:** char, wchar_t, char16_t, char32_t
- **Boolean Type:** bool
- **Void Type:** void

Derived Data Types

- **Arrays:** Collection of similar data types
- **Pointers:** Store memory addresses
- **References:** Alias to another variable

User-Defined Types

- **User-Defined Types:** struct, class, union, enum



Structures vs Classes in C++

Key Differences

Default Access:

struct: public by default

class: private by default

Default Inheritance:

struct: public inheritance by default

class: private inheritance by default

Usage Convention:

struct: typically for passive data containers

class: for objects with behavior and data

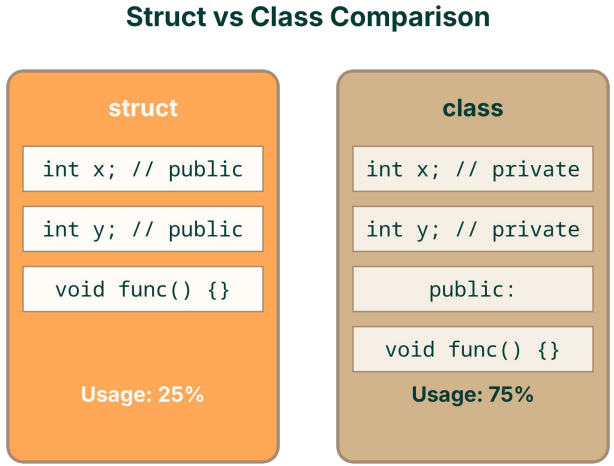
When to Use Each

Use struct when:

Creating simple data containers with public access

Use class when:

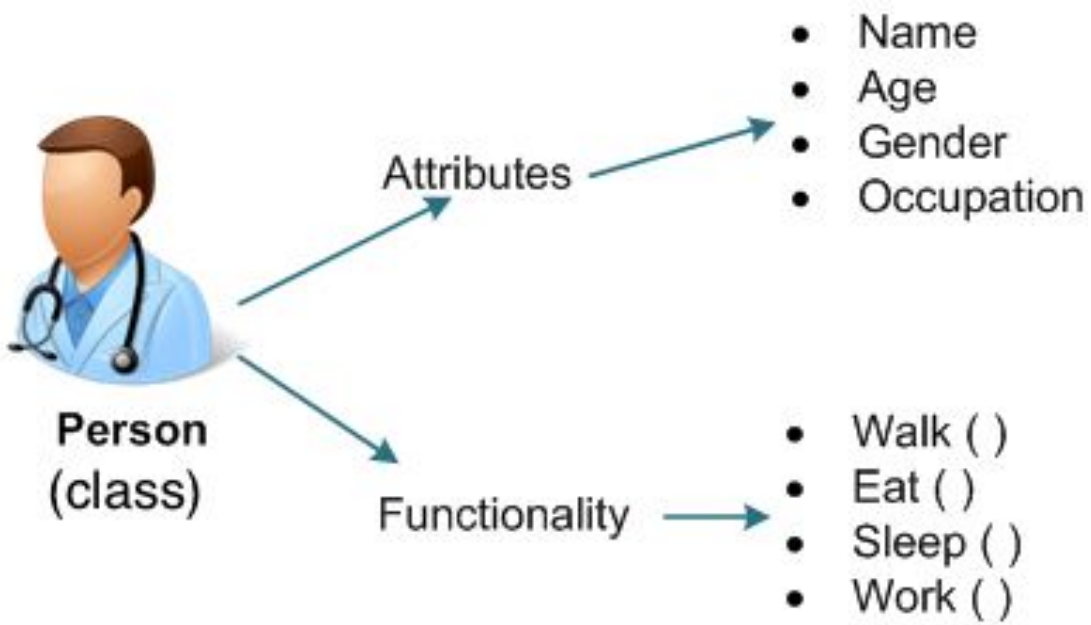
Implementing encapsulation with private data and public interface



Usage statistics based on modern C++ codebases

Working with Classes : Class and Objects

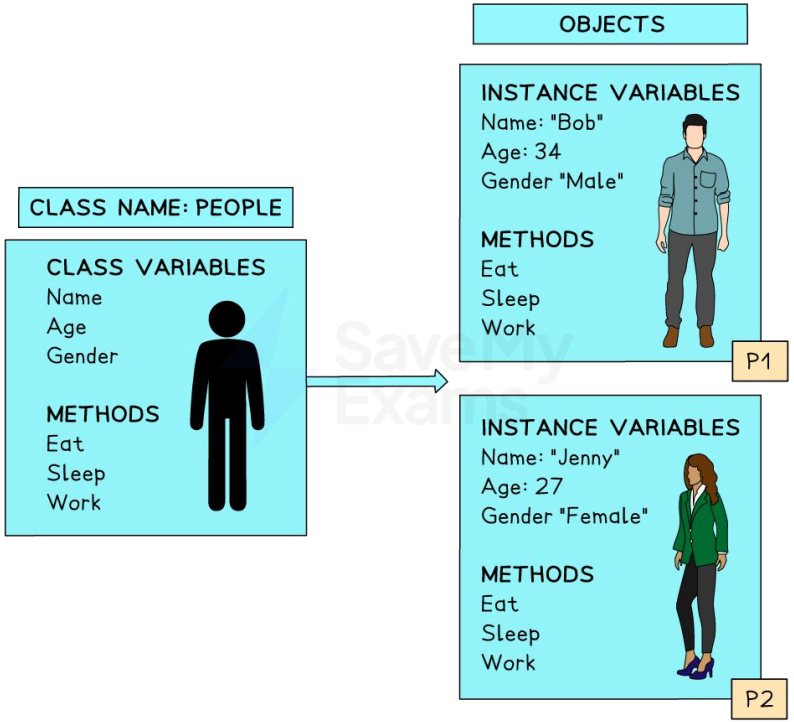
Class and Objects



};

Working with Classes : Class and Objects

Class and Objects




```
};
```

Working with Classes : Class Syntax

Definition of a Class

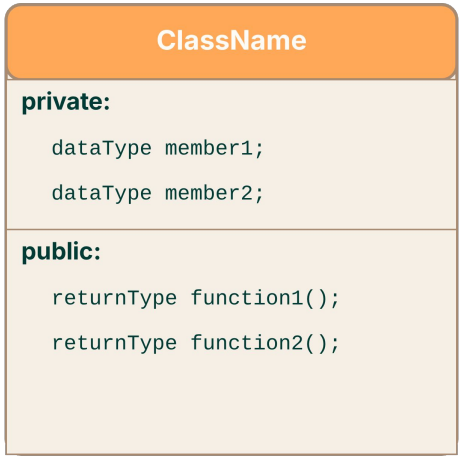
A class is a user-defined data type that serves as a blueprint for creating objects.

It defines:

- Data members (attributes)
- Member functions (methods)
- Access specifiers (public, private, protected)

```
class ClassName {
  // Access specifier
  private:
    // Data members
    dataType member1;
    dataType member2;

  public:
    // Member functions
    returnType function1();
    returnType function2();
};
```

Class Structure Visualization



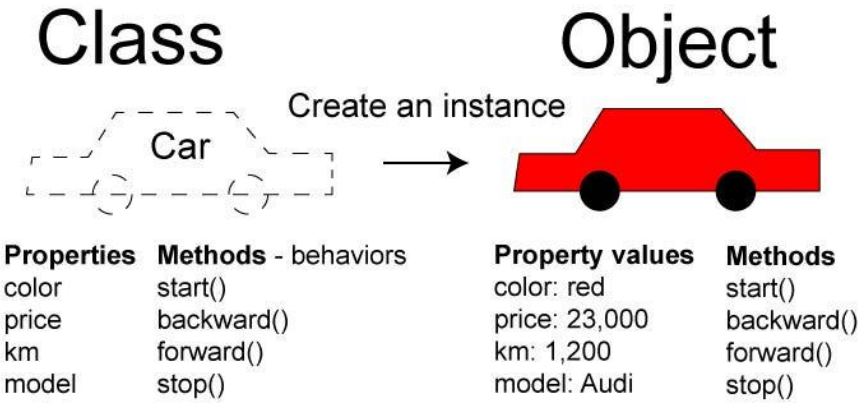
Working with Classes : Member Functions and Data Members

Data Members (Attributes)

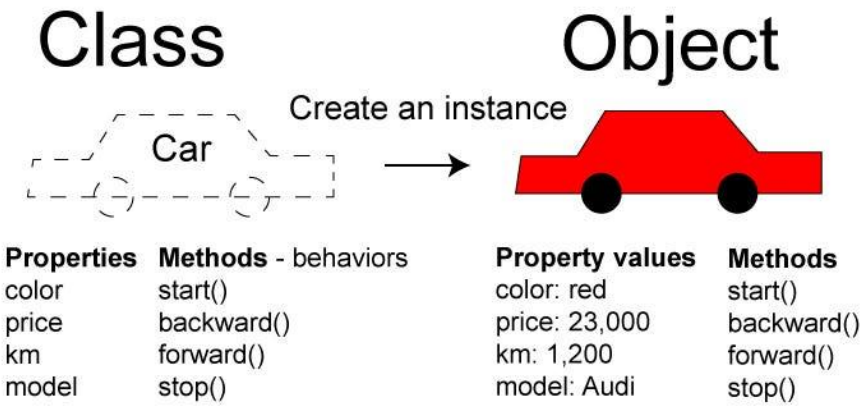
- Variables declared within a class
- Represent the state/properties of an object
- Can have different access levels (public, private, protected)

Member Functions (Methods)

- Functions declared within a class
- Define the behavior of objects
- Can access and modify data members
- Can be declared inside the class and defined outside

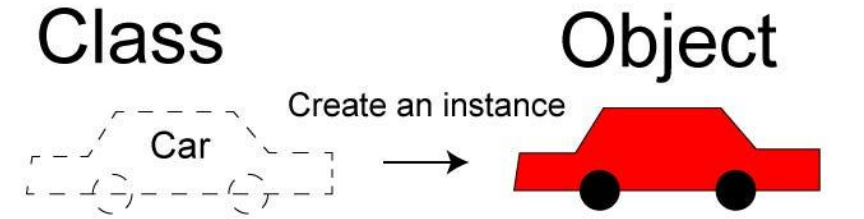


Working with Classes : Example Class




```
1   #include <iostream>
2
3   using namespace std;
4
5   class Car {
6       public:
7           int price;
8           int km;
9           string color;
10          string model;
11
12          void start();
13          int forward();
14          int backward();
15          void stop();
16  };
17
```

Working with Classes : Object Creation



Properties	**Methods** - behaviors
color	start()
price	backward()
km	forward()
model	stop()

Property values	**Methods**
color: red	start()
price: 23,000	backward()
km: 1,200	forward()
model: Audi	stop()

```
18  int main() {
19      Car audi;
20
21      audi.price = 2000;
22      audi.km = 500;
23      audi.color = "black";
24      audi.model = "b200";
25
26      cout<<"Color:" <<audi.color<<endl;
27  }
```

Working with Classes : Dot Operator (.)

Dot Operator

Using dot operator (.) to access members

```
18  int main() {
19      Car audi;
20
21      audi.price = 2000;
22      audi.km = 500;
23      audi.color = "black";
24      audi.model = "b200";
25
26      String color = audi.color; // Accessing Data members
27      audi.start(); // Access Function Members
28  }
29
```

Working with Classes : Access Operator (->)

Pointer Access

Using arrow operator (->) for pointer objects.

```
22  int main() {
23      Car bmw;
24
25      bmw.price = 4500;
26      bmw.km = 800;
27      bmw.color = "White";
28      bmw.model = "X7";
29
30      Car* ptrBmw = &bmw;
31
32      // Access members using the pointer and arrow operator (->)
33      string color = ptrBmw->color;
34      ptrBmw->start();
35      ptrBmw->stop();
36  }
```

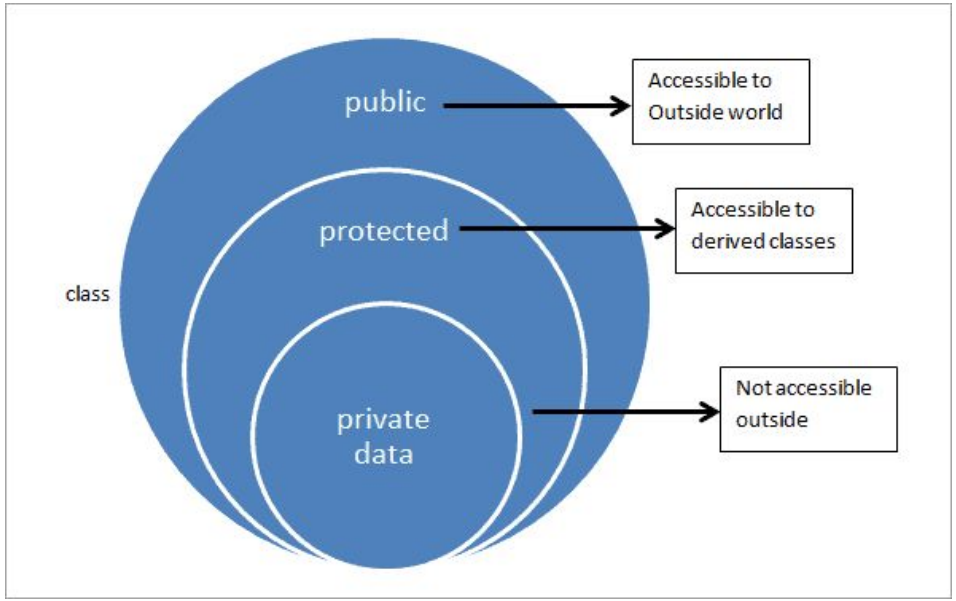
Working with Classes : Access Specifiers

Access Specifiers in C++

Access specifiers define the accessibility of class members:

- **public:** Accessible from anywhere
- **private:** Accessible only within the class
- **protected:** Accessible within the class and derived classes

Access specifiers are fundamental to encapsulation, one of the core principles of OOP.



Best Practices

- Keep data members private
- Provide public methods to access and modify private data
- Use protected for members that derived classes need access to

Working with Classes : Access Specifiers

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

using namespace std;

class Car {
    private:
        int price;
        int km;

    public:
        string model;
        string color;

        void start();
        int forward();
};

int main() {
    Car bmw;

    // Set public members
    bmw.model = "BMW M3";
    bmw.color = "Black";

    // Set or access private members
    bmw.price = 1000; // Gives error [Error] 'int Car::price' is private

    bmw.km = 220; // Gives error  [Error] 'int Car::km' is private

}
```

Working with Classes : Access Specifiers

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

using namespace std;

class Car {
private:
    int price;
    int km;

public:
    string model;
    string color;

    // Setter methods
    void setPrice(int p) {
        price = p;
    }

    void setKm(int k) {
        km = k;
    }

    // Getter methods
    int getPrice() {
        return price;
    }

    int getKm() {
        return km;
    }
};
```

```cpp
int main() {
    Car bmw;

    // Set public members
    bmw.model = "BMW M3";
    bmw.color = "Black";

    // Set private members via setters
    bmw.setPrice(1000);
    bmw.setKm(220);

    // Get private members via getters
    int bmwPrice = bmw.getPrice();
    int bmwKm = bmw.getKm();
}
```

Working with Classes : Defining Member functions in classes

Inside class definition

Within the class definition itself we can define member functions

```cpp
#include <iostream>

using namespace std;

class Car {
    private:
        int price;
        int km;

    public:
        string model;
        string color;

        void start() { cout << "Car started!" << endl;} // Function defined inside the class
};

int main() {
    Car bmw;
    bmw.model = "BMW M3";
    bmw.color = "Black";

    bmw.start();
}
```

Working with Classes : Defining Member functions in classes

Outside class definition

Use the scope resolution :: operator along with class name and function name.

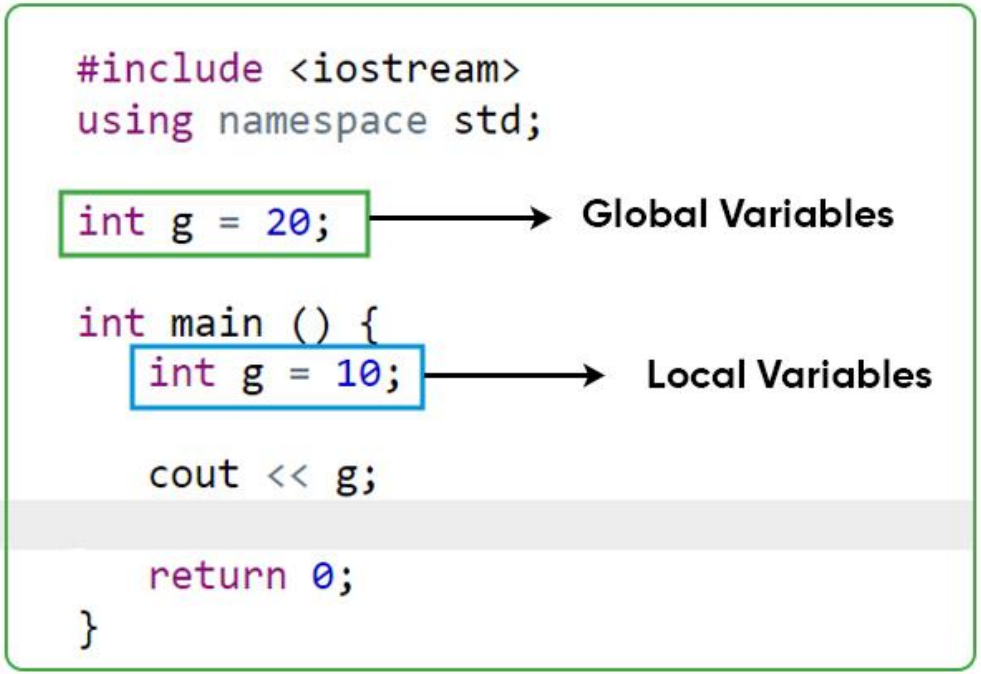
```cpp
#include <iostream>

using namespace std;

class Car {
    private:
        int price;
        int km;

    public:
        string model;
        string color;

        void start(); // declaration only
};

void Car::start() {
    cout << "Car started!" << endl;
}

int main() {
    Car bmw;
    bmw.model = "BMW M3";
    bmw.color = "Black";

    bmw.start();
}
```

Scope of Variables

Variable Scope

Local Variables: Accessible only within the function/block

Global Variables: Accessible throughout the program



Scope of Variables

Within the scope

- You **cannot declare two variables with the same name in the same scope**
- You **can reuse the same name in different scopes**
- A **local variable takes precedence** over a global variable with the same name

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

int a = 100; // Global variable

int main() {
    int a = 200; // Local variable

    cout << "Value of a = " << a << endl; // will print 200

    int a = 300; // gives error [Error] redeclaration of 'int a'
}

```

Scope Resolution Operator

Access global variables

The scope resolution operator(**::**) should be used if there is a local variable and a global variable with the same name.

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

int a = 100; // Global variable

int main() {
    int a = 200; // Local variable

    cout << "Value of local a = " << a << endl; // will print 200

    cout << "Value of global a = " << ::a << endl; // will print 100

}
```

Scope resolution operator

Scope Resolution Operator

Namespace

Can define Functions, classes, variables having the same name in different libraries.

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

// First namespace
namespace car_brand {
    void show() {
        cout << "This is a car from Car Brand namespace." << endl;
    }
}

// Second namespace
namespace bike_brand {
    void show() {
        cout << "This is a bike from Bike Brand namespace." << endl;
    }
}

int main() {
    // Calling functions from each namespace

    car_brand::show(); // Output : "This is a car from Car Brand namespace."

    bike_brand::show(); // Output : "This is a bike from Bike Brand namespace."

    return 0;
}
```

Scope Resolution Operator

Referencing a Nested Class

When you define a class inside another class (called a **nested class**), you can refer to it from outside the outer class using the **scope resolution operator (::)**.

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

class Outer {
    private:
        int x;
        int y;

    public:
        string z;

        // Nested class
        class Inner {
            public:
                void display() {
                    cout << "Inside Inner class (nested in Outer)" << endl;
                }
        };
};

int main() {
    // Referencing the nested class using scope resolution operator
    Outer::Inner obj;

    obj.display();

    return 0;
}
```

Dynamic Memory Allocation

Why is it?

- Memory allocated **at runtime** using pointers.
- Useful when the size of data is **not known in advance**.
- Memory must be **manually deallocated** to avoid memory leaks.

Operators

- new → allocates memory from the heap
- delete → frees the allocated memory

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5
6       int *numPtr = new int;    // allocate memory
7
8       delete numPtr;            // deallocate memory
9   }
10
```

Dynamic Memory Allocation

Why is it?

- Memory allocated **at runtime** using pointers.
- Useful when the size of data is **not known in advance**.
- Memory must be **manually deallocated** to avoid memory leaks.

Operators

- new → allocates memory from the heap
- delete → frees the allocated memory

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

int main() {

    int *arr = new int[5]; // allocate memory for 5 integers

    delete[] arr;  // deallocate memory
}

```