OOPS in C++

In **C++**, objects can be created in two primary ways:

1. **Simple way (Stack Allocation)**
2. **Using new keyword (Heap Allocation)**

1. Creating an Object in a Simple Way (Stack Allocation)

In this method, the object is created **on the stack**. When the function or block ends, the object is automatically destroyed.

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

class Person {
public:
    string name;
    int age;

    void greet() {
        cout << "Hello, " << name << "!" << endl;
    }
};

int main() {
    Person p; // Object created on the stack
    p.name = "John";
    p.age = 30;
    p.greet(); // Output: Hello, John!

    return 0;
}
```

Characteristics of Stack Allocation

- **Faster memory allocation** because the stack is managed automatically.
- Objects are **automatically destroyed** when they go out of scope.
- **No need to manually delete the object**.
- Best suited for objects with a short lifespan.

2. Creating an Object Using the new Keyword (Heap Allocation)

In this method, the object is **created on the heap** using the new keyword. The object remains in memory until it is explicitly deleted using delete. We get **Object pointer variable here (access using ->)**

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

class Person {
public:
    string name;
    int age;

    void greet() {
        cout << "Hello, " << name << "!" << endl;
    }
};

int main() {
    Person* p = new Person(); // Object created on the heap
    p->name = "Alice";
    p->age = 25;
    p->greet(); // Output: Hello, Alice!

    delete p; // Manually delete the object to free memory

    return 0;
}
```

Characteristics of Heap Allocation

- Objects **persist in memory until explicitly deleted**.
- Memory must be **manually freed using delete**, otherwise, it causes a memory leak.
- More **flexible** for dynamically allocated objects (e.g., when object size is unknown at compile-time).
- Slightly **slower than stack allocation** due to heap memory management.

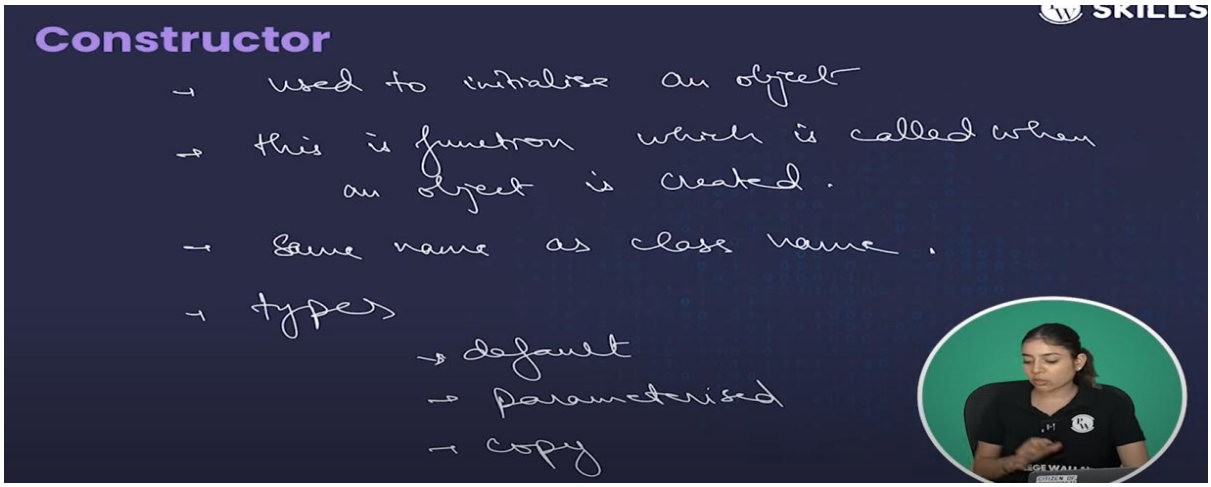
Key Differences

Feature	Stack Allocation (Simple Way)	Heap Allocation (new Keyword)
Memory Location	Stack	Heap
Lifetime	Automatic (destroyed when out of scope)	Manual (must use delete)
Performance	Faster	Slightly slower
Syntax	ClassName obj;	ClassName* obj = new ClassName();
Memory Management	Handled by compiler	Must be manually managed

When to Use What?

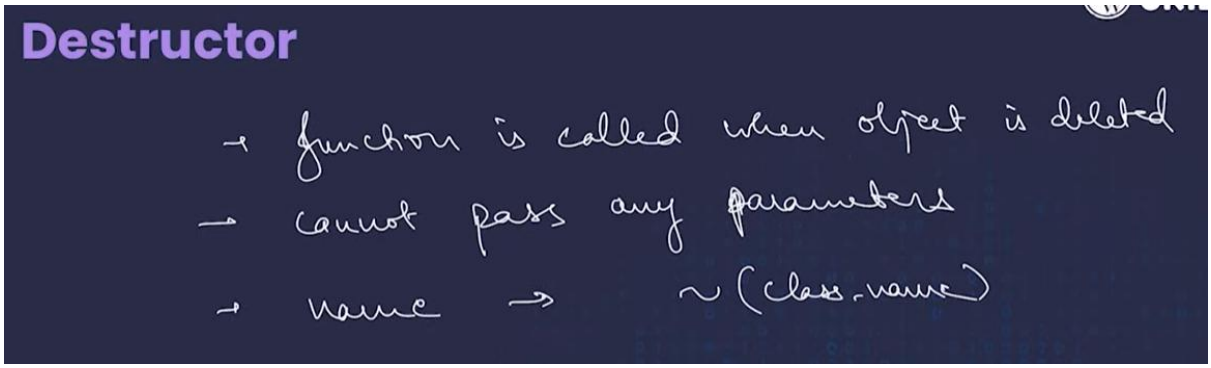
- Use **stack allocation** when objects are small, temporary, and don't need to persist outside the function.
- Use **heap allocation** when objects are large, need to persist across multiple function calls, or when the size is unknown at compile time.

Constructor -

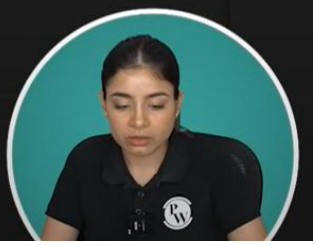


```
int main(){

    Rectangle r1;
    cout<<r1.l<<" "<<r1.b<<endl;

    Rectangle r2(3,4);
    cout<<r2.l<<" "<<r2.b<<endl;

    Rectangle r3 = r2;
    cout<<r3.l<<" "<<r3.b<<endl;


    return 0;
}
```

Destructor –

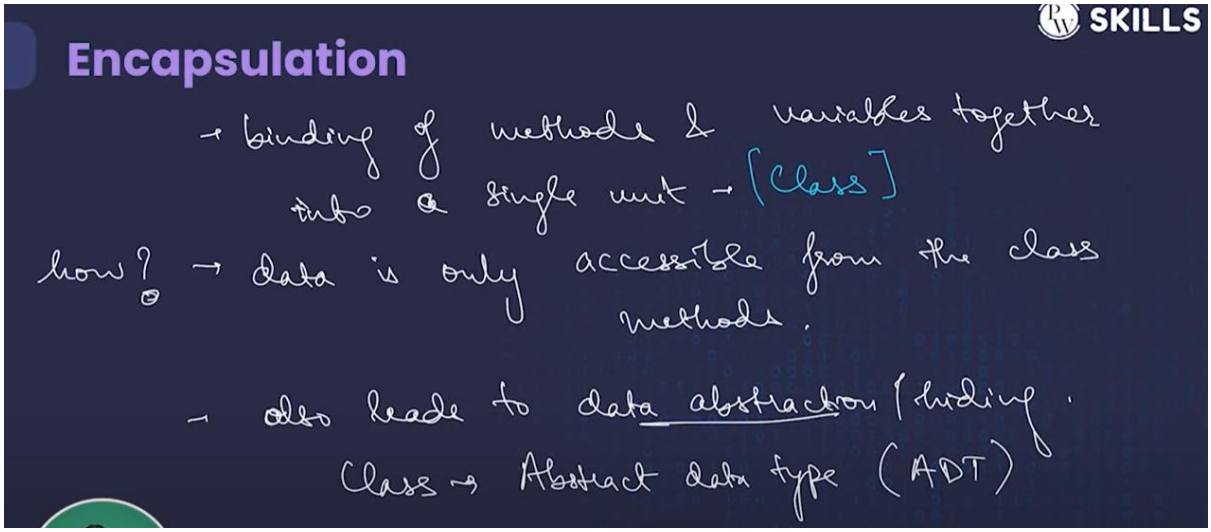



```
    Rectangle(){ //default constructor - no args passed
        l = 0;
        b = 0;
    }

    Rectangle(int x, int y){ //parameterised constructor - args pass
        l=x;
        b=y;
    }

    Rectangle(Rectangle& r){ //copy constructor - initialise an obj by another existing o
        l = r.l;
        b = r.b;
    }

    ~Rectangle(){ //
        cout<<"Destructor is called";
    }

};
```



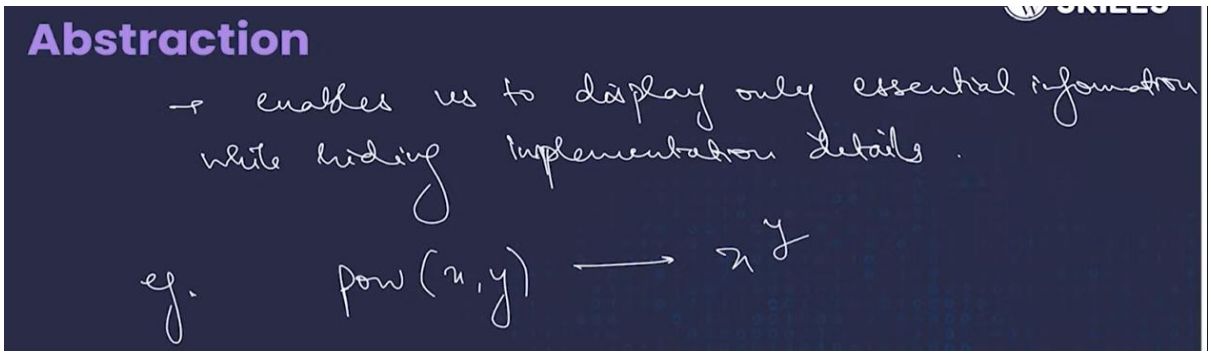
```
int main(){

    Rectangle* r1 = new Rectangle();
    cout<<r1->l<<" "<<r1->b<<endl;
    delete r1;
```

Encapsulation –



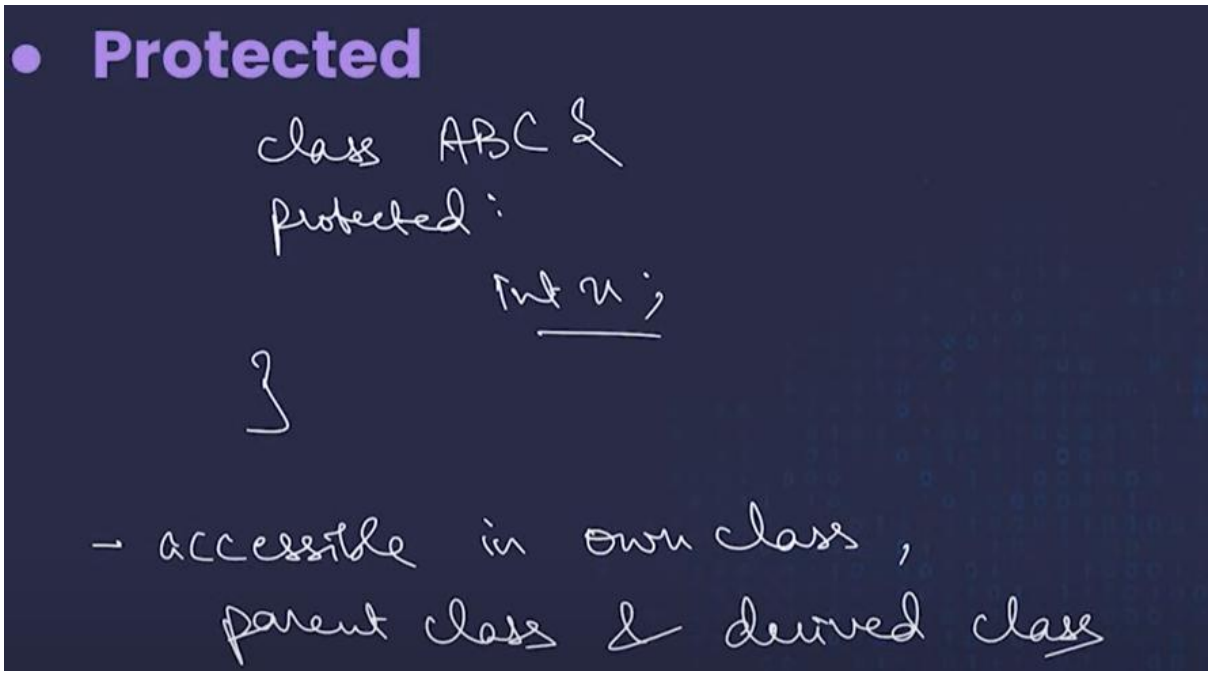

```
#include<iostream>
using namespace std;

class ABC{
    int x;

    public:

    void set(int n){
        x = n;
    }

    int get(){
        return x;
    }

};

int main(){

    ABC obj1;
    obj1.set(3);
    cout<<obj1.get()
```

Abstraction -



Access Specifier & mode of Inheritence – Pub, Priv, Protected.

Public – data & Methods will be accessed anywhere in the code.



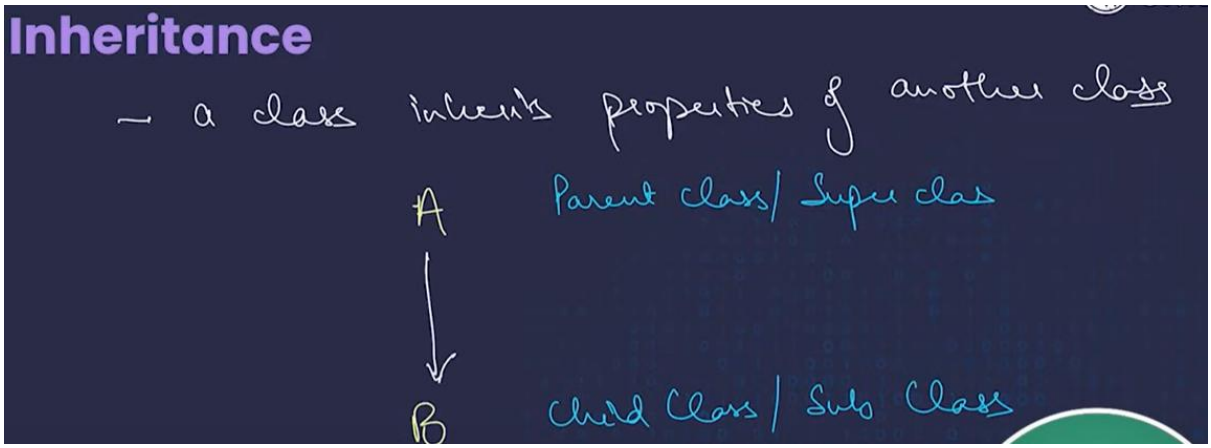
Private – accessible only in own class. (by default all member are private if don't written Privately)

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

class Parent{

public:
    int x;

protected:
    int y;

private:
    int z;

};
```

Inherit Class in 3 ways Then -

```
class Child1: public Parent{
    //x will remain public
    //y will remain protected
    //z will not be accessible
};

class Child2: private Parent{
    //x will be private
    //y will be private
    //z will be inaccessible
};

class Child3: protected Parent{
    //x will be protected
    //y will be protected
    //z will be inaccessible
}
```

Inheritance -



Types of Inheritance in C++

1. **Single Inheritance - One class inherits from another.**
2. **Multiple Inheritance - A class inherits from multiple base classes.**
3. **Multilevel Inheritance - A derived class inherits from another derived class.**
4. **Hierarchical Inheritance - Multiple classes inherit from a single base class.**
5. **Hybrid Inheritance - Combination of two or more types of inheritance.**

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

// Base class
class Animal {
public:
    void eat() {
        cout << "Animal is eating." << endl;
    }
};

// Derived class (inherits from Animal)
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog is barking." << endl;
    }
};

int main() {
    Dog d;
    d.eat();  // Inherited from Animal
    d.bark(); // Own method
    return 0;
}
```

Output:

```
Animal is eating.
Dog is barking.
```

2 Multiple –

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

// First base class
class Father {
public:
    void height() {
        cout << "Height from Father." << endl;
    }
};

// Second base class
class Mother {
public:
    void eyes() {
        cout << "Eyes from Mother." << endl;
    }
};

// Derived class inheriting from both Father and Mother
class Child : public Father, public Mother {
public:
    void traits() {
        cout << "Child inherits traits." << endl;
    }
};

int main() {
    Child c;
    c.height(); // From Father
    c.eyes();   // From Mother
    c.traits(); // Own method
    return 0;
}
```

```
Height from Father.
Eyes from Mother.
Child inherits traits.
```

3. Multilevel Inheritance (Grandparent → Parent → Child)

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

// Grandparent class
class Grandfather {
public:
    void wisdom() {
        cout << "Wisdom from Grandfather." << endl;
    }
};

// Parent class inheriting from Grandfather
class Father : public Grandfather {
public:
    void skills() {
        cout << "Skills from Father." << endl;
    }
};

// Child class inheriting from Father
class Child : public Father {
public:
    void play() {
        cout << "Child loves to play." << endl;
    }
};

int main() {
    Child c;
    c.wisdom(); // Inherited from Grandfather
    c.skills(); // Inherited from Father
    c.play();   // Own method
    return 0;
}
```

```
Wisdom from Grandfather.
Skills from Father.
Child loves to play.
```

4 Hierarchical Inheritance (One Parent, Multiple Children)

Copy Edit

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

// Base class
class Parent {
public:
    void house() {
        cout << "Parent has a house." << endl;
    }
};

// First derived class
class Son : public Parent {
public:
    void bike() {
        cout << "Son has a bike." << endl;
    }
};

// Second derived class
class Daughter : public Parent {
public:
    void car() {
        cout << "Daughter has a car." << endl;
    }
};

int main() {
    Son s;
    s.house(); // Inherited from Parent
    s.bike();  // Own method

    Daughter d;
    d.house(); // Inherited from Parent
    d.car();   // Own method

    return 0;
}
```

```
Parent has a house.
Son has a bike.
Parent has a house.
Daughter has a car.
```

5. Hybrid Inheritance (Combination of Multiple Types)

cpp

Copy Edit

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

// Base class
class A {
public:
    void showA() {
        cout << "Class A" << endl;
    }
};

// Class B inherits from A (Single Inheritance)
class B : public A {
public:
    void showB() {
        cout << "Class B" << endl;
    }
};

// Class C inherits from A (Hierarchical Inheritance)
class C : public A {
public:
    void showC() {
        cout << "Class C" << endl;
    }
};

// Class D inherits from both B and C (Multiple Inheritance)
class D : public B, public C {
public:
    void showD() {
        cout << "Class D" << endl;
    }
};

int main() {
    D d;
    d.showB(); // From B
    d.showC(); // From C
    d.showD(); // Own method

    // d.showA(); ❌ ERROR: Ambiguity (A is inherited twice)

    return 0;
}
```

```
Class B
Class C
Class D
```

Q. What is the Diamond Problem in C++?

The **diamond problem** occurs in **multiple inheritance** when a class **inherits from two classes that share a common base class**. This leads to **ambiguity** because the derived class receives **two copies of the common base class**.

```cpp
// Base class
class A {
public:
    void show() {
        cout << "Class A" << endl;
    }
};

// Derived class B inherits from A
class B : public A { };

// Derived class C inherits from A
class C : public A { };

// Derived class D inherits from both B and C
class D : public B, public C { };

int main() {
    D d;
    d.show();  // ❌ ERROR: Ambiguity
    return 0;
}
```

Output - error: request for member 'show' is ambiguous

- D inherits show() from **both B and C**.
- **Compiler doesn't know which one to use** (from B or C?).
- This is called the **diamond problem** because the inheritance structure looks like a **diamond**:

🚀 **The Diamond Problem** happens due to **multiple inheritance with a shared base class**, causing **duplicate copies** and **ambiguity**.
🛠️ **Solution:** Use **virtual inheritance** to ensure **only one copy** of the base class exists.

```cpp
// Base class
class A {
public:
    void show() {
        cout << "Class A" << endl;
    }
};

// Virtual inheritance
class B : virtual public A { };
class C : virtual public A { };

// Derived class D
class D : public B, public C { };

int main() {
    D d;
    d.show();  // ✅ Works fine, no ambiguity
    return 0;
}
```

Copy Edit

Output:

```
Class A
```

Copy Edit

How Virtual Inheritance Works?

✅ `B` and `C` virtually inherit `A`, so only one copy of `A` exists in `D`.

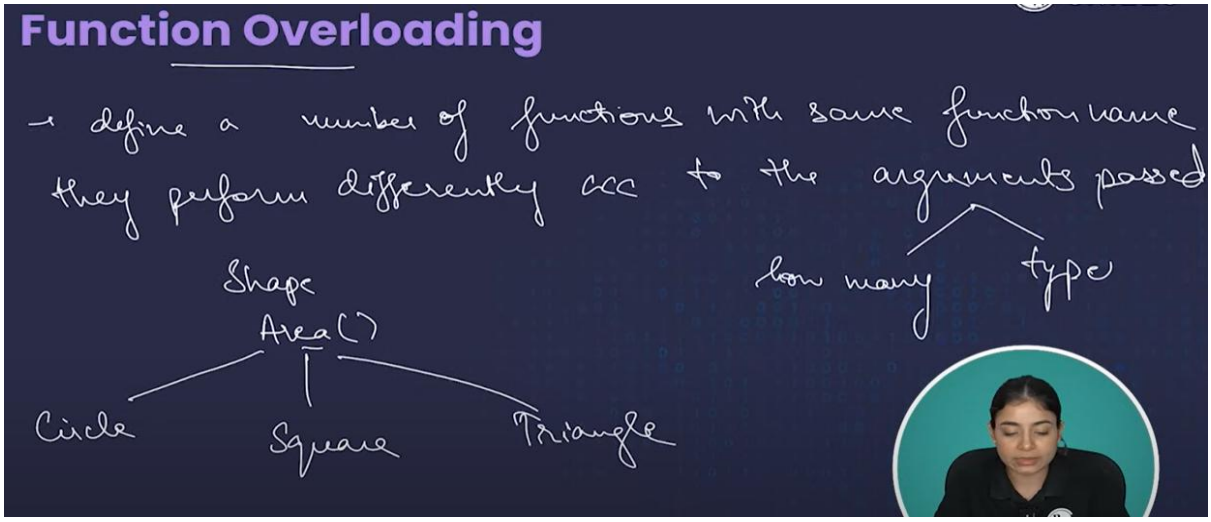
✅ No ambiguity when calling `show()` in `D`.

Polymorphism

Polymorphism in C++ is the **ability of a function, object, or operator to take multiple forms**. It allows the **same interface** to be used for **different types of behavior**, improving **code reusability and flexibility**.

It is classified into:

1. **Compile-time Polymorphism (Static Binding)** – Achieved through **Function Overloading** and **Operator Overloading**. That Shape / form is decided at compile time.
2. **Run-time Polymorphism (Dynamic Binding)** – Achieved using **Function Overriding and Virtual Functions**. Decided at runtime




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

// Function to add two integers
int add(int a, int b) {
    return a + b;
}

// Function to add two doubles
double add(double a, double b) {
    return a + b;
}

// Function to add three integers
int add(int a, int b, int c) {
    return a + b + c;
}

int main() {
    cout << add(10, 20) << endl;       // Calls add(int, int) → Output: 30
    cout << add(5.5, 2.5) << endl;     // Calls add(double, double) → Output: 8.0
    cout << add(1, 2, 3) << endl;      // Calls add(int, int, int) → Output: 6
    return 0;
}
```

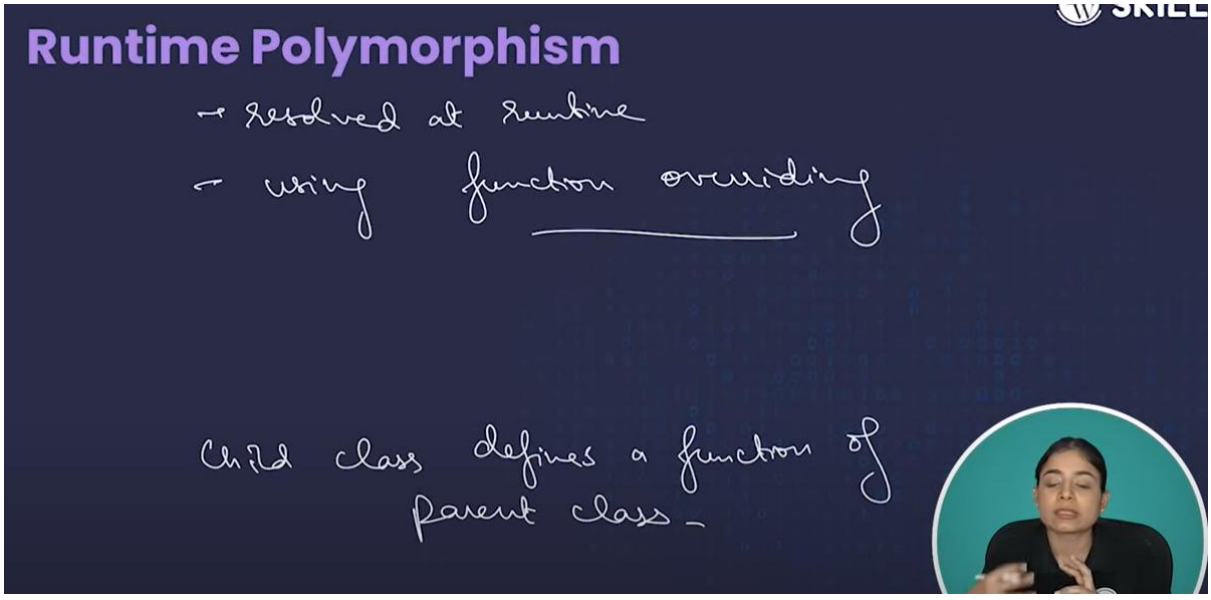
Operator Overloading for Adding Complex Numbers in C++

Operator overloading allows us to **overload operators** to work with user-defined types like Complex numbers.

✅ **Example: Overloading + Operator for Complex Numbers**

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

// Complex Number Class
class Complex {
public:
    int real, imag;

    // Constructor
    Complex(int r = 0, int i = 0) {
        real = r;
        imag = i;
    }

    // Overloading + operator to add two complex numbers
    Complex operator+(const Complex& c) {
        Complex ans(0,0);
        ans.real = real + c.real;
        ans.imag = imag + c.imag;
        return ans;
        // return Complex(real + c.real, imag + c.imag);
    }
};

int main() {
    Complex c1(3, 4);  // 3 + 4i
    Complex c2(2, 5);  // 2 + 5i

    Complex c3 = c1 + c2;  // Calls operator+ function
    cout << c3.real << " + " << c3.imag << "i" << endl;

    return 0;
```

5 + 9i

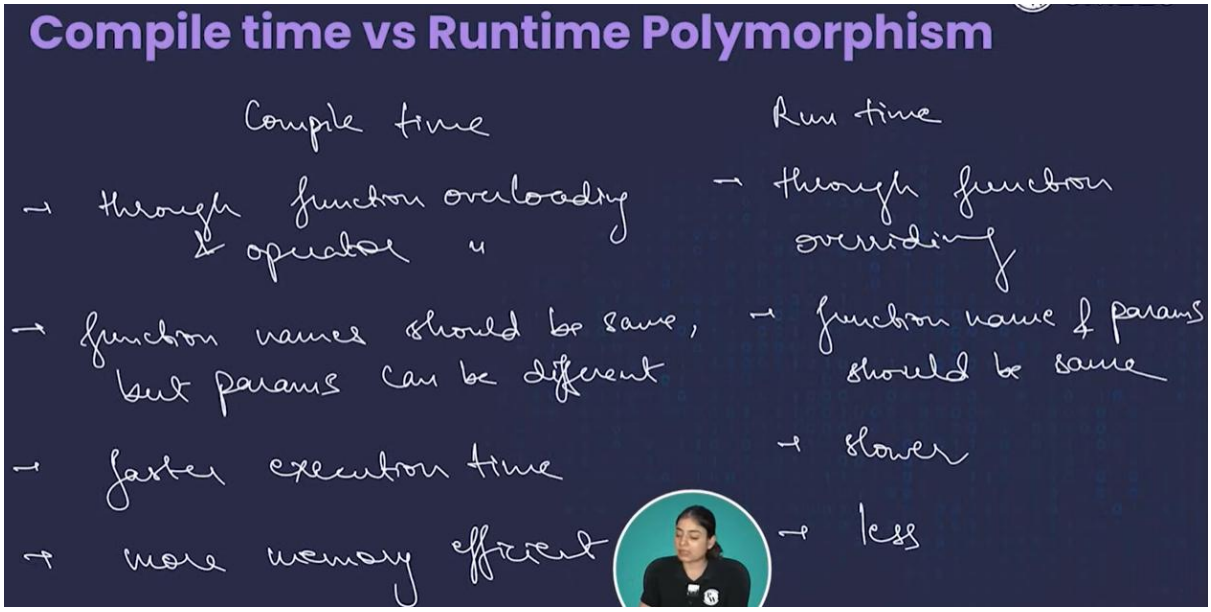


Method Overriding occurs when a **derived class** provides a **specific implementation** of a function that is already defined in the **base class**.

👉 Used in **Run-time Polymorphism** (with virtual functions).
👉 The function in the base class **must be virtual** for dynamic binding.

```cpp
class Parent {
public:
    virtual void show() {  // Virtual function
        cout << "Parent class function" << endl;
    }

    void print(){
        cout <<"Parent" <<endl;
    }
};

class Child : public Parent {
public:
    void show() {  // Overriding base class function
        cout << "Child class function" << endl;
    }
    void print(){
        cout <<"Child" <<endl;
    }
};

int main() {
    Parent* p;
    Child c;
    Parent p1;

    p = &c;  // Base class pointer points to derived class object
    p->show();  // Calls Child's show() function (Run-time Polymorphism)
    p->print();

    cout << "------"<<endl;
    p = &p1;  // Base class pointer points to base class object
    p->show();  // Calls Parent's show() function (Run-time Polymorphism)
    p->print();

    return 0;
```

```
PS E:\SDE\CPP Project> cd "e:\SDE\CPP Project\" ; if ($?)
Child class function
Parent
------
Parent class function
Parent
PS E:\SDE\CPP Project>
```



Friend Function in C++

A **friend function** is a function that is **not a member of a class** but has **access to its private and protected members**. It is declared **inside the class** but defined **outside** using the friend keyword.

```cpp
class Parent {
private:
    int num;

public:
    Parent(int n) {
        num=n;
    }

    // Friend function declaration
    friend void display(Parent& s);
};

// Friend function definition (outside class)
void display(Parent& s) {
    cout << "The number is: " << s.num << endl; // Accessing private member
}

int main() {
    Parent obj(10);
    display(obj); // Calls friend function

    return 0;
}
```

The number is: 10

Friend Function vs. Member Function

Feature	Friend Function	Member Function
Access	Can access private & protected members.	Can only access own class members.
Definition	Defined **outside** the class.	Defined **inside** the class.
Invocation	Called like a normal function.	Called using an object (`obj.method()`).
Use Case	Helpful in **operator overloading** and **external functions**.	Used for **regular class operations**.

Q. Why Use const in C++?

The const keyword ensures that a **variable or function cannot be modified after declaration**.

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

int main() {
    const int x = 10;
    // x = 20;  // ❌ ERROR: Cannot modify a const variable
    cout << "x = " << x << endl;
    return 0;
}
```

🔹 **Output:**

```
ini                                                    Copy  Edit

x = 10
```

2️⃣ Why Use `num(n)`? (Initializer List)

The syntax `num(n)` in a **constructor** is called an **initializer list**. It **directly initializes** class members **before** the constructor body executes.

✅ Example: Initializer List (`num(n)`)

cpp Copy Edit

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

class Sample {
private:
    int num;

public:
    // Constructor using initializer list
    Sample(int n) : num(n) {
        cout << "Constructor Called. num = " << num << endl;
    }
};

int main() {
    Sample obj(10);  // Calls constructor
    return 0;
}
```

🔹 **Output:**

nginx Copy Edit

```nginx
Constructor Called. num = 10
```