Composition

- Composition is a “has-a” relationship in OOP.
- A class contains objects of other classes.
- Promotes code reuse and modularity.

Composition

```
class Engine {
        public:
        void start() { cout << "Engine starts!"; }
        };
class Car {
        Engine e; // Composition
        public:
                void drive() {
                e.start();
                cout << "Car is driving!";
        }
};
```

this pointer

- Every non-static member function has a hidden pointer called “this”.
- It points to the current object.
- Syntax: this->member or (*this).member

this pointer

- A **static member function** belongs to the **class itself**, not to any particular object.
- It means:
  - We can call a static function even **without creating an object**.
  - Since there is no object, there is no this pointer.

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

class MyClass {
public:
    static void show() {
        cout << "Static function\n";
        // cout << this;  ERROR: no 'this' inside static
    }
};

int main() {
    MyClass::show();   // works without object
    MyClass obj;
    obj.show();        // also works, but still no 'this'
}
```

```
Static function
Static function
```

this pointer

- this is a pointer to the current object.
- Used to:
  - Differentiate between member variables and parameters when they have the same name.
  - Return the current object (useful in method chaining).
- It’s automatically passed to all non-static member functions.

Example: 1

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

class Student {
    int age;
    string name;
public:
    Student(int age, string name)
    {
        age=age;
        name=name;
    }
    void printValue()
    {
        cout<<"Age:"<<age<<endl;
        cout<<"Name:"<<name<<endl;
    }

};

int main() {
    Student s(45, "Rohit");
    s.printValue();
    return 0;
}
```

```
Age:1651076199
Name:
```

Example: 2

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

class Student {
    int age;
    string name;
public:
    Student(int age, string name)
    {
        this->age=age;
        this->name=name;
    }
    void printValue()
    {
        cout<<"Age:"<<age<<endl;
        cout<<"Name:"<<name<<endl;
    }

};

int main() {
    Student s(45, "Rohit");
    s.printValue();
    return 0;
}
```

```
Age:45
Name:Rohit
```

this pointer

- this is a local object pointer (instance member function).
- this contains address of calling object.
- this refers to the caller object.
- this address can’t be modified.

this as local object pointer

- this is a local object pointer (instance member function).
  - The this pointer **exists only inside non-static member functions** of a class.
  - It's automatically passed as a hidden argument when you call a function on an object.
  - It's **local** to that function, meaning each call has its own this pointer.

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

class A {
public:
    void show() {
        cout << "this is local to this function: " << this << endl;
    }
};

int main() {
    A obj1, obj2;
    obj1.show();
    obj2.show();
}

```

```
this is local to this function: 0x7ffea55a71d6
this is local to this function: 0x7ffea55a71d7
```

this as local object pointer (cont.)

- When you call a **non-static member function** (like obj.show()), the compiler secretly adds an extra parameter:

  show(&obj); // &obj is passed as `this`

- Inside the function, this is just like a **local variable** holding the address of the calling object.
- When you call the same function from a **different object**, another this pointer is created, pointing to the new object.
  - So, **every call** gets its own this pointing to **that specific object**.

this contains address of calling object

- this contains the address of the calling object.
  - The pointer's value is the **memory address** of the object used to invoke the method.

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

class A {
public:
    void display() {
        cout << "Object address: " << this << endl;
    }
};

int main() {
    A a;
    cout << "Actual address of a: " << &a << endl;
    a.display();
}
```

```
Actual address of a: 0x7ffcc9fd5a17
Object address: 0x7ffcc9fd5a17
```

- So, this is literally the object's address.

this refers to the caller object.

- this refers to the caller object.
  - When you call obj.method(), inside that method, this refers to obj.
  - This is why this->member accesses the calling object's member variables.

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

class Student {
    string name;
public:
    void setName(string n) {
        this->name = n;  // 'this' points to the object calling setName
    }
    void show() { cout << "Name: " << name << endl; }
};

int main() {
    Student s1, s2;
    s1.setName("Raj");
    s2.setName("Priya");
    s1.show();  // this points to s1
    s2.show();  // this points to s2
}
```

```
Name: Raj
Name: Priya
```

this address can’t be modified.

- this address can’t be modified.
  - The this pointer is const.
  - You cannot change what this points to (cannot make it point to another object).
  - You can use it, dereference it, and access data through it.
  - But you **cannot make it point elsewhere**.

this pointer

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

class A {
public:
    void test() {
        A another;  // create another object

        // ❌ ERROR: You cannot modify 'this' pointer
        // this = &another;
        // Uncommenting the line above will cause a compilation error

        cout << "Address of current object (this): " << this << endl;
        cout << "Address of another object: " << &another << endl;
    }
};

int main() {
    A obj;        // create an object
    obj.test();   // call the test() method
    return 0;
}
```

```
Address of current object (this): 0x7ffefaa16237
Address of another object: 0x7ffefaa16217
```

To return reference to the calling object

```
// Reference to the calling object can be returned
Test& Test::func () {
// Some processing
return *this;
}
```

- When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

To return reference to the calling object

```cpp
#include<iostream>
using namespace std;
class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test &setX(int a) { x = a; return *this; }
  Test &setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};
int main()
{
  Test obj1(5, 5);
  // Chained function calls.  All calls modify the same object
  // as the same object is returned by reference
  obj1.setX(10).setY(20);
  obj1.print();
  return 0;
}
```

```
x = 10 y = 20
```

Question 1

```
1  #include<iostream>
2  using namespace std;
3
4  class Test
5  {
6  private:
7    int x;
8  public:
9    Test(int x = 0) { this->x = x; }
10   void change(Test *t) { this = t; }
11   void print() { cout << "x = " << x << endl; }
12 };
13
14 int main()
15 {
16   Test obj(5);
17   Test *ptr = new Test (10);
18   obj.change(ptr);
19   obj.print();
20   return 0;
21 }
```

- If you want to **copy data from another object**, you should **copy its members** instead of reassigning this.
- Corrected change() function:
  - void change(Test *t) { this->x = t->x; }
- this is a **constant pointer**; you cannot reassign it.
- To "change" an object's values from another object, **copy the data**, not the pointer itself.

Question 2

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

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  static void fun1() { cout << "Inside fun1()"; }
  static void fun2() { cout << "Inside fun2()"; this->fun1(); }
};

int main()
{
  Test obj;
  obj.fun2();
  return 0;
}
```

- fun2() is a **static member function**.
- In **static functions**, there is **no this pointer** because static functions:
  - Belong to the class, **not any specific object**.
  - Can be called without creating an object.
- So, trying to use this inside a static method causes a compile-time error.

Question 2

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

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  void fun1() { cout << "Inside fun1()"; }
  void fun2() { cout << "Inside fun2()"; this->fun1(); }
};

int main()
{
  Test obj;
  obj.fun2();
  return 0;
}
```

```
Inside fun2()Inside fun1()
```

Question 3

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

class Test
{
private:
  int x;
  int y;
public:
  Test (int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test setX(int a) { x = a; return *this; }
  Test setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj1;
  obj1.setX(10).setY(20);
  obj1.print();
  return 0;
}
```

```
x = 10 y = 0
```

Question 4

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

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  void setX(int a) { x = a; }
  void setY(int b) { y = b; }
  void destroy()  { delete this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj;
  obj.destroy();
  obj.print();
  return 0;
}
```

Question 4

- Test obj;
  - obj is a **stack-allocated object** (created on the stack, not with new).
- obj.destroy();
  - Inside destroy(), delete this; is called.
  - **this points to obj**, which lives on the stack.
  - delete should **only be used for heap-allocated objects** (created with new).
  - Calling delete this; on a stack object is **undefined behavior**.
- obj.print();
  - After destroy(), the memory for obj is considered **invalid**.
  - Any further access (like calling print()) leads to **undefined behavior** (could crash, print garbage, or even appear to work "sometimes").
- delete this; is only valid if:
  - The object was created with a new.
  - You are absolutely sure no other code will access it afterward.
- Here, obj is stack-allocated, so deleting it is **illegal**.

```
int main() {
  Test* obj = new Test;  // allocate on heap
  obj->destroy();       // ok: delete this; deletes heap object
  return 0;
}
```

Cascading Call Example :

```cpp
class Box {
    int length, breadth;
public:
    Box() : length(0), breadth(0)  {}

    // setter functions returning *this for cascading
    Box& setLength(int l) {
        length = l;
        return *this;   // return current object
    }

    Box& setBreadth(int b) {
        breadth = b;
        return *this;
    }

    void display() {
        cout << "Length=" << length
             << " Breadth=" << breadth << endl;
    }
};

int main() {
    Box b;
    // Cascading (method chaining)
    b.setLength(10).setBreadth(20).display();
    return 0;
}
```

Dynamic Memory Allocation

```
int *arr = new int[5];

for(int i=0; i<5; i++)
        arr[i] = i;

// free memory

delete[] arr;
```

Normal Constructor call

```
#include<iostream>
using namespace std;
class GPS
{
        int x,y,z;
        public:
        GPS()
        {
                x=0;
                y=0;
                z=0;
        }
        GPS(int z)
        {
                x=0;
                y=0;
                this->z=z;
        }
        void show()
        {
                cout<<x<<" "<<y<<" "<<z<<endl;
        }
};
int main()
{
        GPS obj(3);
        obj.show();  // displays 0 0 3
        return 0;
}
```

Delegating Constructor call

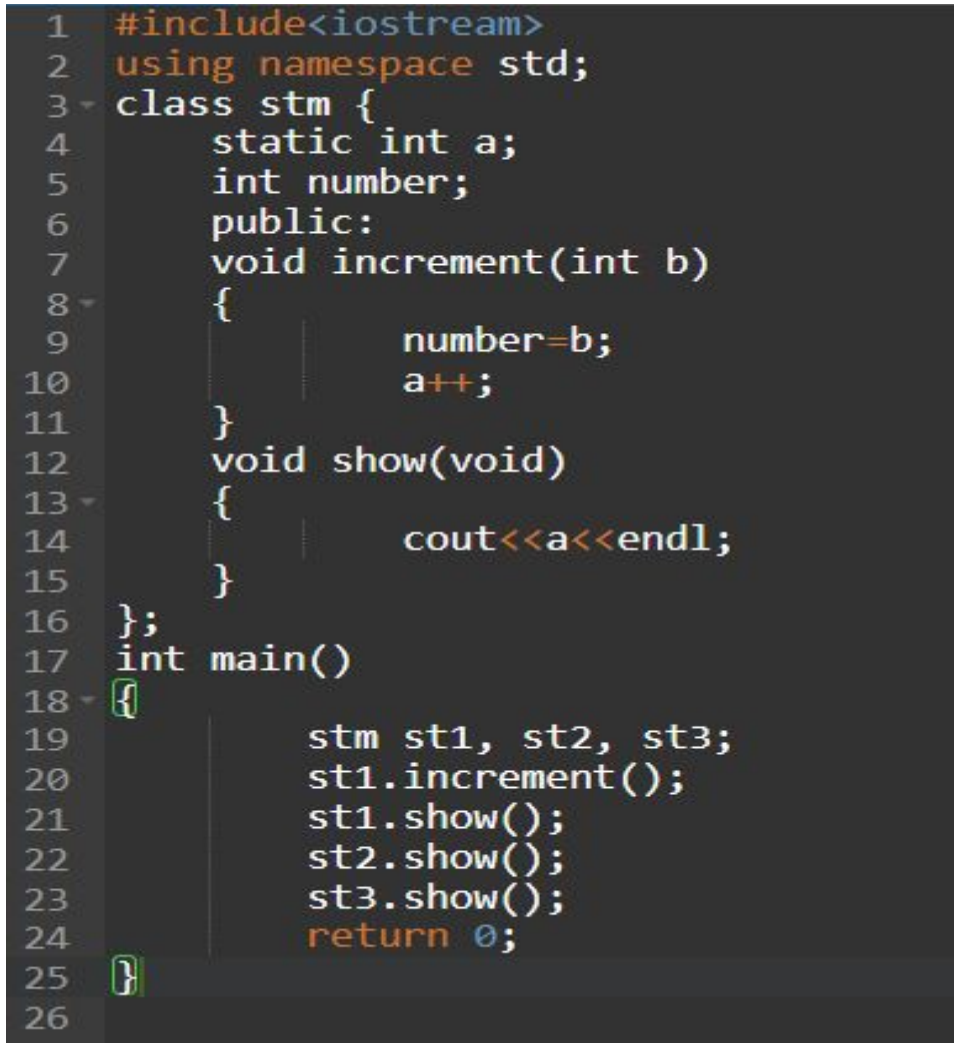
```
#include<iostream>
using namespace std;
class GPS
{
        int x,y,z;
        public:
        GPS()
        {
                x=0;
                y=0;
                z=0;
        }
        GPS(int z) : GPS()
        {
                this->z=z;
        }
        void show()
        {
                cout<<x<<" "<<y<<" "<<z<<endl;
        }
};
int main()
{
        GPS obj(3);
        obj.show(); //displays  0 0 3
        return 0;
}
```

Debugging Exercise 1

```
#include<iostream>
using namespace std;
class addition {
        int p,q,r,s;
        void input(void)
        {
                cout<<"Enter the three numbers \n";
                cin >>p>>q>>r;
        }
        void show(void)
        {
                cout<<"The addition of three numbers is :"<< add() << endl;
        }
        int add(void)
        {
                s=p+q+r;
                return(s);
        }
};
int main()
{
        addition x;
        x.input();
        x.show();
        return 0;
}
```

1. input, show, and add are declared private here.
2. main function cannot access.

Debugging Exercise 2



```cpp
#include<iostream>
using namespace std;
class stm {
    static int a;
    int number;
    public:
    void increment(int b)
    {
        number=b;
        a++;
    }
    void show(void)
    {
        cout<<a<<endl;
    }
};
int main()
{
    stm st1, st2, st3;
    st1.increment();
    st1.show();
    st2.show();
    st3.show();
    return 0;
}
```

No matching for increment function call

Debugging Exercise 3

```cpp
#include<iostream>
using namespace std;
class weight
{
        int letter, ML;
        public:
        void getdata();
        void putdata();
        void sum_of_weight(weight,weight);
};
void weight::getdata()
{
    cout<<"in letter:";
    cin>>letter;
    cout<<"in ML";
    cin>>ML;

}
void weight::putdata()
{
        cout<<letter<<"Letter and"<<ML<<"ML";
}
void weight :: sum_of_weight(weight w1, weight w2)
{
        ML=w1.ML+w2.ML;
        letter = ML/1000;
        ML = ML%1000;
        letter+=w1.letter+w2.letter;
}
int main()
{
        weight w1,w2,w3;
        cout<<"Enter weight in Letter and ML\n";
        cout<<"Enter weight 1";
        w1.getdata();
        cout<<"Enter Weight 2";
        w2.getdata();
        sum_of_weight(w1,w2);
        cout<<"Total weight = ";
        w3.putdata();
        return 0;
}
```

Sum of weight was not declared in the main scope

Debugging Exercise 4

```
#include<iostream>
using namespace std;
class age
{
        private:

        public:
        int child_age;
        age():child_age(20){}
        int father_age(age);
};
int father_age(age d)
{
        d.child_age+=15;
        return d.child_age;
}
int main()
{
        age D;
        cout<<"Father's age"<<father_age(D);
        return 0;
}
```

Fathers age: 35

Note: if child_age is private data member, then father_age cannot access the child_age inside its body.