# Clue Card of Unit-1: OOP with Memory Concepts

## 1. Structure vs Class

- **struct:** public by default (light blue)
- **class:** private by default (light green)

**Example Code snippet**:

```
struct Point { int x, y; };
class Person { string name; int age; public: Person(string n,int a):name(n),age(a){} };
```

## 2. Classes & Objects

- Stack vs Heap (highlight memory types with colors)
- **Code snippet:**

```
MyClass obj;          // Static memory from Stack
MyClass *p = new MyClass(); // Dynamic from Heap
obj.method(); p->method();
```

---

| **3. Namespaces**: Avoid name conflicts | **Code snippet:** `namespace A { void f(); } A::f(); using namespace A; f()` |
|---|---|

---

## 6. Types of Function Calling

| Method | Code Snippet | Effect / Difference | Effect on Memory | Remark |
|---|---|---|---|---|
| **Call by Value** | `void fun(int x){ x = 20; } int main(){ int a = 10; fun(a); cout << a; // 10 }` | Copy of variable is passed, Changes **don't affect** original, Safe but inefficient for large objects. | Caller → Copy → Function (no effect back) | Copy, safe, unchanged |
| **Call by Pointer** | `cpp void fun(int *p){ *p = 20; } int main(){ int a = 10; fun(&a); cout << a; // 20 }` | Address of variable passed, Changes **affect** original, Risk of invalid/null pointers, Common for arrays & dynamic memory. | Caller ↔ Address ↔ Function (* modifies caller) | Uses address, original changes, needs * and &. |
| **Call by Reference** | `cpp void fun(int &r){ r = 20;} int main(){int a = 10; fun(a); cout << a; // 20}` | Variable passed as **alias**, Changes **affect** original, Safer than pointers, more efficient than value, Preferred in modern C++. | Caller ↔ Alias ↔ Function (direct effect) | Alias, original changes, clean syntax. |

---

## 7. Passing/Returning Objects

- **Pass by reference:** `void f(MyClass &o)`
- **Return by value:** `MyClass f(){ return obj; }`

## 8. Array of Objects

```
MyClass arr[5];
arr[0].method();
```

---

## 9. Static Keyword

| Concept | Explanation | Code Snippet | Key Difference / Note |
|---|---|---|---|
| **Definition** | static is a storage class specifier. Variables exist for the entire program, not destroyed when scope ends. | NOT APPLICABLE | Lifetime = whole program. Scope depends on declaration. |
| **Local Static Variable** | Declared inside a function, retains value between calls. | `void fun(){ static int x=0; x++; cout<<x; } int main(){ fun(); fun(); } // Output: 1 2` | Unlike normal locals, not re-initialized each call. |
| **Static Data Member (Class)** | Shared by all objects of a class. Only one copy exists. | `class Test{ static int count; }; int Test::count=0;` | Memory shared, not per object. |
| **Static Member Function** | Belongs to class, not objects. Can access only static members. | `class Test{ static void show(); }; Test::show();` | Called with ClassName::func(). |
| **Global Static Variable** | Declared outside functions, visible only within the file. | `static int g=10; // file-scope only` | Provides **internal linkage** (not accessible in other files). |
| **Local vs Local Static** | Local → recreated each call. Static local → persists across calls. Useful for counters, caching. | | |
| **Instance Var vs Static Member** | Instance → one copy per object. Static → one copy for class. Saves memory, common values. | | |
| **Global vs Global Static** | Global → accessible across files. Global static → restricted to current file. Prevents name conflicts. | | |

## 10. Friend Keyword

| Aspect | Explanation | Code Snippet | Key Notes |
|---|---|---|---|
| **Definition** | friend keyword allows non-member function or class to access private/protected members of another class. | class A { int x; friend void show(A&); // friend function }; | Friendship gives special access. |
| **Friend Function** | Non-member function with access to class's private members. | void show(A &a){ cout<<a.x; } | Declared inside class with friend. |
| **Friend Class** | One class can declare another class as friend, giving access to its private/protected members. | class B; class A { friend class B; }; | Friendship is **not mutual** unless declared both ways. |

**Access:** Can access private & protected data of the class. Called like normal function, not with object.
**Use cases:** Operator overloading, sharing data between classes, helper functions. Common for <<, >> operators.
**Limitations:** Breaks encapsulation, should be used sparingly. One-way relationship, not inherited.

---

## 10. Pointers & this

- Pointer: p->member
- this pointer: this->member
- Diagram: object -> pointer flow

## 11. Dynamic Initialization & Memory

```
MyClass *p = new MyClass(args);
delete p;
delete[] arr;
```

Note: Always free heap memory

**Dr. Suresh Raikwar**

# Clue Card of Unit-2: Constructor and Destructor

---

**Constructors & Its Types**

- Special member function with the same name as the class.
- Automatically called when an object is created.

**Types:**

1. Default Constructor → No parameters.
2. Parameterized Constructor → With parameters.
3. Copy Constructor → Copies another object.

**Constructor Overloading:** Multiple constructors with different parameters. Allows objects to be initialized in different ways.

- **Rule-1:** Different number of parameters.
- **Rule-2:** Different types of parameters.

***Note: if both rules violated, then overloading is not possible.***

**Example:**
```
class MyClass{ public: MyClass() { /* default */ }
               MyClass(int a) { /* parameterized */ }
               MyClass(int a, int b) { /* two params */ }};
```

**Constructors in Array of Objects:**

- When an array of objects is created, the constructor is called for each element.
- Default constructor is used if no arguments are provided.

**Example:**
```
class Example {int i; float j;
public:
   Example() { cout << "Constructor called\n"; }
   Example(int i, float j) { this→i=i; this→j=j; cout << "Constructor called\n"; }};


Example arr[3];  // Constructor called 3 times
Example arr[2]={Example(), Example(2,5.3)}; //zero and parameterized constructor will be called.
```

**Constructors with Default Arguments**

- Parameters can have default values.
- Acts like both default and parameterized constructor.

**Example:**
```
class Example { public: Example(int a = 0, int b = 0) { /* use a, b */ } };

Example obj1;       // a=0, b=0
Example obj2(5);    // a=5, b=0
Example obj3(3,4);  // a=3, b=4
```

**Dynamic Constructor**

- Objects created at runtime using new.
- Constructor is called when the object is allocated dynamically.
- Dynamic constructor may be zero-argument, parameterized or copy type.

**Example:**
```
class Example {  int *p;
public: Example(int val) { p=new int; *p=val; cout << "Value: " << val << endl; } };

Example e(5); // dynamic constructor will be called.
```

**Destructor**

- Special member function with ~ before class name.
- Called automatically when an object is destroyed.
- Used to free resources like memory or files.

**Example:**
```
class Example {
public:
   ~Example() { cout << "Destructor called\n"; } };
```

**const Keyword:** Ensures data integrity and prevents modification.

- **const with Data Member:** Data can't be modified after initialization.
  **Example:** `class Example { const int x; public:  Example(int val) : x(val) { } };`
- **const with Member Function:** Function doesn't modify object's state.
  **Example:** `class Example { public:  int getValue() const { return 10; } };`
- **const with Object:** Entire object is constant → only const functions can be called.
  **Example:**
  ```
  const Example obj;
  obj.getValue(); // allowed only if getValue() is const
  ```

**Dynamic and Static memory Allocation:**
```
class Dynamic_Memory{   int i;
  public: Dynamic_Memory(){i=10;cout<<"Constructor"<<endl;}
  void display(){cout<<"i="<<i<<endl; }
  ~Dynamic_Memory(){cout<<"Destructor"<<endl;}
};
int main() {  Dynamic_Memory o1; //static memory allocation
   o1.display();
   Dynamic_Memory *o2=nullptr; //This does not call default constructor.
   o2= new Dynamic_Memory(); //dynamic memory allocation;
   o2→display();  delete o2; return 0;}
```

# Clue Card of Unit-3: Inheritance

1. **Forms of Inheritance:** Inheritance allows a class to acquire properties (data members and member functions) from another class. Below are its forms and access modes:

| Form of Inheritance | Description | Access Modes | Code Example |
|---|---|---|---|
| Single Inheritance | A derived class inherits from a single base class. | Public, Private, Protected | class Base { public: int a;}; class Derived : public Base {}; |
| Multiple Inheritance | A derived class inherits from more than one base class. | Public, Private, Protected | class Base1 { public: int a; }; class Base2 { public: int b; }; class Derived : public Base1, public Base2 {}; |
| Multilevel Inheritance | A derived class inherits from a base class, and another derived class inherits from it. | Public, Private, Protected | class Base { public: int a;}; class Derived1 : public Base {}; class Derived2 : public Derived1 {}; |
| Hierarchical Inheritance | Multiple derived classes inherit from a single base class. | Public, Private, Protected | class Base { public: int a;}; class Derived1 : public Base {}; class Derived2 : public Base {}; |
| Hybrid Inheritance | Combination of two or more types of inheritance. | Public, Private, Protected | class Base { public: int a; }; class Derived1 : public Base {}; class Derived2 : public Derived1 {}; class Derived3 : public Base {}; |

2. **Inheritance with Constructor and Destructor:** Constructors and destructor are called in a specific order in inheritance.
   - Base class constructors are invoked first, followed by derived class constructors.
   - Destructors are called in reverse order.

**Example:**

```
class Base { public: Base(){cout << "Base Constructor called" << endl;}
    ~Base() { cout << "Base Destructor called" << endl; } };
class Derived : public Base { public: Derived() { cout << "Derived
Constructor called" << endl; }
    ~Derived() { cout << "Derived Destructor called" << endl; }
};
int main() {
    Derived obj;
    return 0;
}
```

3. **Benefits and Limitations of Inheritance**

**Benefits:**
- Code reusability to reduce redundancy
- Extensibility allows new functionalities to be added easily.
- Data Hiding can be achieved using access specifiers (private/protected).
- Method Overriding through polymorphism.

**Limitations:**
- Increased complexity in multi-level and multiple inheritance.
- Dependency on base class: Any changes in base class affect derived classes.
- Diamond problem: Ambiguity arises in multiple inheritance.
- Tight coupling between classes may reduce flexibility.
- Debugging can be harder due to complex relationships.

# Special Clue Card: this Pointer

**What is `this` Pointer?**

- It is an implicit pointer available inside all non-static member functions.
- Points to the current object invoking the function.
- Used to access the calling object's members.

**Characteristics**

- Available in all non-static member functions
- Holds the address of the object that invoked the function
- Used to resolve naming conflicts between parameters and class members

**Common Uses of `this` Pointer**

**1. Access members of the current object:**

```
class Example {
   int x;
public:
void setX(int x) {
     this->x = x;  // distinguishes between member and parameter
}};
```

**2. Return object from a member function:**

```
Example& setX(int x) {
   this->x = x;
   return *this;
}
```

**3. Chain member function calls:** obj.setX(10).setY(20);

4**. Compare objects:**

```
bool isSame(Example& other) {
   return this == &other;
}
```

## Important Notes

- **`this` pointer** is automatically passed to non-static functions
- **Cannot be used in static member functions (as static functions are not tied to a particular object)**
- Useful in operator overloading, fluent interfaces, and avoiding shadowing between member variables and function parameters

**Example Code**

```
class Example {
   int x;
public:
   Example(int x) { this->x = x; }

   Example& setX(int x) {
      this->x = x;
      return *this;
   }

   void show() {
      cout << "x = " << this->x << endl;
   }
};

int main() {
   Example obj(5);
   obj.setX(10).setX(20);
   obj.show();
   return 0;
}
```