C++ Concepts: Constructors, Destructors, Pointers and this Pointer

1. Class and Constructors

A class is a blueprint for creating objects. It contains data members (variables) and member functions (methods).

A constructor is a special function that initializes an object when it is created. It has the same name as the class and no return type.

Types of constructors:

1. Default Constructor – No parameters.
2. Parameterized Constructor – Takes arguments.
3. Copy Constructor – Initializes one object as a copy of another.
4. Constructor with Default Arguments – Allows default parameter values.

Syntax:

```
class ClassName {
public:
  // Default constructor
  ClassName() { ... }

  // Parameterized constructor
  ClassName(int x, int y) { ... }

  // Constructor with default arguments
  ClassName(int x = 0, int y = 0) { ... }
};
```

2. Destructor

A destructor is a special function that is automatically invoked when an object goes out of scope or is deleted. It is used to free resources or perform clean-up operations.

Key Properties of Destructor:

1. Name starts with a tilde (~) followed by the class name.
2. There can only be one destructor in a class.
3. Destructor does not take parameters.
4. Destructor has no return type.
5. Compiler automatically generates a default destructor if none is defined.

Syntax:

```
class ClassName {
public:
  ~ClassName() {
    // cleanup code here
  }
};
```

3. Pointer to Objects

We can create a pointer that stores the address of an object. To access members of the object through the pointer, we use the arrow operator (->).

Syntax:

```
ClassName obj;
ClassName *ptr = &obj;
ptr->memberFunction();
```

4. this Pointer

The 'this' pointer is an implicit pointer available in all non-static member functions. It points to the current object of the class.

Uses of 'this' pointer:

1. To resolve ambiguity when local variables shadow class data members.
2. To return the current object from a function.
3. Used in operator overloading and method chaining.

Syntax:

```
class ClassName {
    int x;
public:
    void setX(int x) {
        this->x = x;   // 'this' pointer resolves ambiguity
    }
};
```

5. Explanation of new Operator

1. **Normal variables** are created on the **stack** (temporary memory).
   They are destroyed automatically when the function ends.
2. Sometimes we need memory that should **exist until we decide to delete it**.
   For this, we use the **heap** (permanent runtime memory).
3. The **new operator** creates memory on the **heap** and gives us a **pointer** to it.
4. When we create an **object using new**, its **constructor is automatically called**.
5. Memory created with new is **not freed automatically** → we must use delete.

6. Friend Function

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 with the keyword 'friend'.

Uses:
- To allow external functions to access private data.
- Useful for operations involving multiple classes (e.g., swapping private values, adding objects).

Syntax:

```
class ClassName {
   int data;
   friend void functionName(ClassName &obj);
};

void functionName(ClassName &obj) {
   // can access obj.data
}
```

Example:

```
class ClassName {
private:
   int data;                // private data
public:
   ClassName(int d) { data = d; }
   friend void showData(ClassName c);  // declaration of friend function
};
// Definition of friend function
void showData(ClassName c) {
   cout << "Data = " << c.data;  // can access private member directly
}
```

7. Friend Class

A friend class is a class whose member functions have access to the private and protected members of another class. Declared using the keyword 'friend class'.

Uses:
- When two classes are closely related and need to share data.
- Simplifies operations involving multiple classes.

Syntax:

```
class A {
   int value;
   friend class B; // B is a friend of A
};
class B {
   void show(A &obj) {
      cout << obj.value; // Allowed
   }
};
```

8. Dynamic Memory Allocation (new & delete)

Dynamic memory allocation means allocating memory during runtime instead of compile time. In C++, this is done using 'new' and 'delete'.

- 'new' keyword allocates memory from heap.
- 'delete' keyword frees the allocated memory to avoid memory leaks.

Syntax:

```
int *p = new int;       // allocates memory for one integer
delete p;               // frees memory

int *arr = new int[5];   // allocates memory for array of 5 integers
delete[] arr;           // frees memory
```