Notes

Constructors And Destructors In C++

1. Object Must Be in a Deterministic State

- An object should have a defined state when one of its data members changes.
- There must be:
  - An **initial state (Initialize function: Constructor)** when an object is created.
  - A **finish state** (**destroy function: Destructor)** to properly clean up resources when an object is destroyed.

2. Constructors: Creation and Initialization

Definition

A **constructor** is a special function in a class that is automatically called when an object is created. Its primary purpose is to initialize data members.

Properties of a Constructor

- **Same name as the class**.
- **No return type** (not even void).
- **Automatically called** when an object is created.

```
class shoplist {
public:
   shoplist() {
      count = 0;
   }
};
```

- The constructor shoplist() is automatically called when an object is created.
- It initializes count to 0.

Parameterized Constructor

A constructor can take arguments to initialize an object with specific values.

```
class A {
public:
    string c;

    A(string e)
{
        c = e;
    }
};
```

- A(string e) is a constructor with a **parameter**.
- It initializes c with the value passed as e.

```
A obj("hello c++");
```

- The object obj is created, and its c member is initialized with "hello c++".

Stack Allocation

When an object is created **without new**, it is allocated on the **stack**.

```
A a1;
A a2;
```

- Here, a1 and a2 are created **on the stack**.
- Memory is automatically managed; when the function ends, the objects are destroyed.

Heap Allocation

When an object is created using new, it is allocated on the **heap**.

```
A *c = new A("hello");
```

- The object is allocated **dynamically**.
- It remains in memory until delete is explicitly called.

4. Destructors

A **destructor** is a special function that is automatically called when an object is destroyed.

Properties of a Destructor

- **Same name as the class**, but preceded by a tilde (~).
- **No return type**.
- **No parameters**.
- **Called automatically** when an object is destroyed.

Example Destructor

```
class A {
public:
   ~A() {
      cout << "Destructor called";
   }
};
```

- This destructor will print a message when an object is destroyed.

Destructor Call in Stack vs Heap

1.Stack Objects

```
int main() {
    A a1;
    A b1;
}
```

- The destructors for a1 and b1 are automatically called when main() ends.

2.Heap Objects

```
int main() {
    A *c = new A("hello");
    delete c; // Destructor called here
}
```

Since c is created on the **heap**, it must be explicitly deleted.

- If delete c; is omitted, there will be a **memory leak**.

When is a Constructor Called in C++?

A **constructor** is automatically called when an object of a class is **created**.
The constructor is responsible for initializing the object.

Different Ways of Calling Constructors

- Implicit Call
- Explicit Call
- Copy Initialization
- Direct Initialization

Scenarios When a Constructor is Called

1. When a Local (Stack) Object is Created

- If an object is created **without new**, the constructor is called immediately.

```
class A {
public:
   A() { cout << "Constructor called" << endl; }
};

int main() {
   A obj;  // Constructor is called automatically
}
```

- Output: Constructor called

2. When a Heap Object is Created Using new

- If an object is created dynamically using new, the constructor is called.

```
A* obj = new A(); // Constructor is called
```

- The object remains in memory until explicitly deleted using delete.

3. When an Object is Created with a Parameterized Constructor

- If a class has a constructor with parameters, it is called when an object is initialized with arguments.

```
class A {
public:
   A(string msg) { cout << "Constructor called with: " << msg << endl; }
};

int main() {
   A obj("Hello");  // Parameterized constructor is called
}
```

- Output: Constructor called with: Hello

4. When an Array of Objects is Created

- If an array of objects is created, the constructor is called for each object.

```
class A {
public:
   A() { cout << "Constructor called" << endl; }
};

int main() {
   A arr[3]; // Constructor is called 3 times
}
```

- Output:

```
Constructor called
Constructor called
Constructor called
```

5. When an Object is Passed by Value

- When an object is passed **by value**, the **copy constructor** is called.

```
class A {
public:
   A() { cout << "Default Constructor called" << endl; }
   A(const A& obj) { cout << "Copy Constructor called" << endl; }
};

void func(A obj) { } // Copy constructor is called

int main() {
   A a1;   // Default constructor is called
   func(a1); // Copy constructor is called
}
```

- Output:

```
Default Constructor called
Copy Constructor called
```

6. When an Object is Returned by Value

- When a function returns an object by value, the **copy constructor** is called.

```
A func() {
   A temp;
   return temp;
}
int main() {
   A obj = func(); // Copy constructor is called
}
```

7. When an Object is Created Using Another Object

- If an object is initialized using another object, the **copy constructor** is called.

```
A obj1;
A obj2 = obj1;  // Copy constructor is called
```

- **Constructors** initialize objects automatically when they are created.
- **Parameterized constructors** allow passing values at creation.
- **Objects created on the stack** are automatically managed.
- **Objects created on the heap** must be manually deleted.
- **Destructors clean up resources** when an object is destroyed.
- If heap memory is not freed using delete, **memory leaks** occur.
- **Constructor is called when an object is created.**
- **For stack objects**, it is called immediately.
- **For heap objects (new)**, it is called during allocation.
- **For parameterized constructors**, values are passed at creation.
- **Copy constructor is called** when passing or returning objects by value.
- **For arrays, the constructor is called once per object** in the array.

When is a Constructor *Not* Called in C++?

A **constructor** is **not** called in the following scenarios:

When Using Pointers Without new

If you declare a pointer to an object **without using new**, the constructor is **not** called.

```
class A {
public:
   A() { cout << "Constructor called" << endl; }
};

int main() {
   A* obj;  // Constructor is NOT called (only a pointer is declared)
}
```

Why?

- obj is just a pointer to an object, but no actual object is created.
- The constructor is only called when an object is instantiated, not just when a pointer is declared.

Types of constructors in C++:

Default Constructor

- A constructor that takes no parameters and initializes the object with default values.

```
class Point {
public:
    int x, y;
    Point() { x = 0; y = 0; } // Default constructor
};
```

Parameterized Constructor

- A constructor that takes arguments to initialize the object with specific values.

```
class Point {
public:
    int x, y;
    Point(int a, int b) { x = a; y = b; } // Parameterized constructor
};
```

Copy Constructor

- A constructor that creates a new object by copying an existing object.

```
class Point {
public:
    int x, y;
    Point(int a, int b) { x = a; y = b; }
    Point(const Point &p) { x = p.x; y = p.y; } // Copy constructor
};
```

Calling Default Constructor

- The default constructor is automatically called when an object is created without passing any arguments.

```
class Point {
public:
   int x, y;

   // Default Constructor
   Point() {
      x = 0;
      y = 0;
      cout << "Default Constructor Called" << endl;
   }
};

int main() {
   Point p1;  // Calls Default Constructor
   return 0;
}
```

Calling Parameterized Constructor

- The parameterized constructor is called when an object is created with arguments.

```
class Point {
public:
   int x, y;

   // Parameterized Constructor
   Point(int a, int b) {
      x = a;
      y = b;
      cout << "Parameterized Constructor Called" << endl;
   }
};

int main() {
   Point p2(10, 20); // Calls Parameterized Constructor
   return 0;
}
```

Calling Copy Constructor

- The copy constructor is called when a new object is initialized from an existing object.

```
class Point {
public:
   int x, y;

   // Parameterized Constructor
   Point(int a, int b) {
      x = a;
      y = b;
   }

   // Copy Constructor
   Point(const Point &p) {
      x = p.x;
      y = p.y;
      cout << "Copy Constructor Called" << endl;
   }
};

int main() {
   Point p3(5, 15);  // Calls Parameterized Constructor
   Point p4 = p3;    // Calls Copy Constructor
   return 0;
}
```

Destructors in C++

A **destructor** is a special member function of a class that is automatically invoked when an object **goes out of scope** or is **explicitly deleted**.

Properties of Destructors:

- **Same name as class**, prefixed with ~ (tilde).
- **No arguments allowed** (i.e., destructors **cannot be overloaded**).
- **Automatically called when an object is destroyed**.
- **Mainly used for cleanup**, such as releasing dynamically allocated memory.

```
class A {
public:
    A() {
        cout << "Constructor A\n";
    }
    ~A() {
        cout << "Destructor A\n";
    }
};

int main() {
    A obj;  // Constructor called
    return 0;  // Destructor called when obj goes out of scope
}
```

Output:

```
Constructor A
Destructor A
```

Heap Allocation and Destructor

If an object is **dynamically allocated using new**, its destructor is **not called automatically**.
You must explicitly use delete.

```
class A {
public:

   A() {
      cout << "Constructor A\n";
   }

   ~A() {
      cout << "Destructor A\n";
   }
};

int main() {

   A* p = new A(); // Constructor called

   delete p;      // Destructor called explicitly

   return 0;
}
```

Output:

Constructor A
Destructor A

If delete p; is omitted, the destructor will not be called, causing a memory leak.

Feature	Constructor	Destructor
Naming	Same as class	Same as class, prefixed with ~
Parameters	Can have parameters	Cannot have parameters
Return Type	No return type	No return type
Call Time	Called when an object is created	Called when an object is destroyed
Overloading	Can be overloaded	Cannot be overloaded
Inheritance	Should be virtual for polymorphic behavior	Should be virtual if used in a base class

```
class A {
public:
    string n;

    // Default Constructor
    A() {
        cout << "A Constructor H\n";
    }

    // Parameterized Constructor
    A(string s) {
        n = s;
        cout << "A Constructor " << n <<
"\n";
    }

    // Destructor
    ~A() {
        cout << "A Destructor " << n << "\n";
    }
};
```

```
int main() {
    A q;                // Default constructor

    A* q1 = new A("Q1");    // Dynamically allocated object
    A b("B");           // Local object with parameterized constructor

    A* p;               // Pointer (not initialized)

    {   // Inner scope
        A c("C");       // Local object with parameterized constructor
        p = new A("P");     // Dynamically allocated object
        delete q1;      // Explicitly deleting dynamically allocated
object
    }   // 'c' goes out of scope here, so its destructor is called

    A e("E");           // Another local object

    delete p;           // Explicitly deleting dynamically allocated
object

    return 0;
}
```

Code Execution Flow

1. A q;
   - Calls the **default constructor** → Prints "A Constructor H"
2. A* q1 = new A("Q1");
   - Dynamically allocates an object with "Q1"
   - Calls the **parameterized constructor** → Prints "A Constructor Q1"
3. A b("B");
   - Creates a **local object** "B"
   - Calls the **parameterized constructor** → Prints "A Constructor B"
4. A* p;
   - Creates a **pointer to A**, but **does not allocate memory** (so no constructor call)
5. Inner Scope {}
   - A c("C");
     - Creates a **local object** "C"
     - Calls the **parameterized constructor** → Prints "A Constructor C"
   - p = new A("P");
     - Dynamically allocates an object "P"
     - Calls the **parameterized constructor** → Prints "A Constructor P"
   - delete q1;
     - **Deletes "Q1"**
     - Calls the **destructor** → Prints "A Destructor Q1"
   - Inner scope ends
     - "c" goes **out of scope**, so its **destructor is called** → Prints "A Destructor C"
6. A e("E");
   - Creates a **local object** "E"
   - Calls the **parameterized constructor** → Prints "A Constructor E"
7. delete p;
   - **Deletes "P"**
   - Calls the **destructor** → Prints "A Destructor P"
8. Function main() ends
   - Local objects (b, e, q) **go out of scope**
   - **Destructors are called in reverse order of creation**:
     - "E" → "B" → "H" (default constructor object)

Output:

A Constructor H
A Constructor Q1
A Constructor B
A Constructor C
A Constructor P
A Destructor Q1
A Destructor C
A Constructor E
A Destructor P
A Destructor E
A Destructor B
A Destructor

- Objects created with new **must be deleted** using delete.
- Local (stack) objects are **automatically destroyed** when they go out of scope.
- If delete is missing for a new object, **memory leaks occur**.
- **Default Constructor** is called for q (since no argument is passed).
- **Parameterized Constructor** is used for "Q1", "B", "C", "P", and "E".
- **Dynamically allocated objects (q1 and p) must be explicitly deleted using delete.**
- **Local objects (c, b, e, q) are automatically destroyed when they go out of scope.**
- **Objects are destroyed in reverse order of their creation (Last In, First Out - LIFO).**

Difference Between Structure and Class in C++

Feature	Structure (struct)	Class (class)
Default Access Modifier	Public	Private
Usage	Used for simple data grouping	Used for complex data with encapsulation
Encapsulation	Less strict (all members public by default)	More strict (data hiding by default)
Inheritance	Inherits publicly by default	Inherits privately by default
Member Functions	Allowed	Allowed
Constructor & Destructor	Supported	Supported
Real-world Use	Mostly for Plain Data Structures (like C structs)	Used for Object-Oriented Programming (OOP)

```
struct MyStruct {
    int x;  // Public by default
};

class MyClass {
    int x;  // Private by default
};
```

- Use **struct** for simple **data containers** (like C-style structs).
- Use **class** for **full-fledged objects** with encapsulation.

Function Overloading and **Constructor Overloading** in C++

Function Overloading

Function overloading allows multiple functions to have the same name but different **parameter lists** (different number or types of parameters).

Same function name but different parameter lists.

Can be overloaded by:

- **Number of parameters**
- **Type of parameters**

Return type cannot be used to differentiate functions

Function Overloading

```
int add(int a, int b) {
    return a + b;
}

float add(float a, float b) {
    return a + b;
}

char add(char a, char b) {
    return a + b;
}

int main() {
    cout << add(2, 3) << endl;   // Calls int version
    cout << add(2.5f, 3.2f) << endl; // Calls float version
    cout << add('A', 'B') << endl;  // Calls char version
    return 0;
}
```

Constructor Overloading

Just like functions, constructors can also be overloaded with different parameter lists.

- **Same constructor name but different parameter lists**
- Used to provide **multiple ways** to initialize an object
- If no constructor is defined, a **default constructor** is automatically created
- If a constructor with parameters is defined but **no default constructor exists**, then **objects cannot be created without arguments**

```
class Point {
  int x, y;

public:
  Point() {    // Default Constructor
    x = 0;
    y = 0;
  }

  Point(int a) {  // Constructor with one parameter
    x = a;
    y = 0;
  }

  Point(int a, int b) { // Constructor with two parameters
    x = a;
    y = b;
  }

  void display() {
    cout << "(" << x << ", " << y << ")" << endl;
  }
};
```

```
int main() {
  Point p1;      // Calls default constructor
  Point p2(5);   // Calls constructor with one parameter
  Point p3(4, 9); // Calls constructor with two parameters

  p1.display();
  p2.display();
  p3.display();

  return 0;
}
```

If a **default constructor** is missing, then creating an object **without arguments** will cause a compilation error.

Example:

If we remove Default Constructor in previous program

```
Point() {    // Default Constructor
      x = 0;
      y = 0;
   }
```

Creating object will give error.

Point p; // Error if no default constructor exists

Feature	Function Overloading	Constructor Overloading
Purpose	Provides multiple implementations of a function with the same name	Provides multiple ways to initialize an object
Differentiation	By number or type of parameters	By number or type of parameters
Return Type	Cannot be used to differentiate	Not applicable (constructors have no return type)
Default Handling	No function is automatically created	Default constructor is automatically provided if no constructor is defined

Encapsulation in C++

Encapsulation is a fundamental principle of Object-Oriented Programming (OOP) that:

1. **Restricts direct access** to data members.
2. **Exposes only necessary functionalities** using methods.

Example: Mobile Phone

- **Encapsulation hides complexity**: A user only interacts with the main features (e.g., calling, messaging) without seeing the internal circuits.

- **Two principles**:
  - **Restrict direct access** to sensitive data.
  - **Provide necessary functionalities** while hiding the implementation.

Access Specifiers

Specifier	Accessibility
Public	Accessible from **anywhere**.
Private	Accessible **only within the class**.
Protected	Accessible **within the class and its derived classes**.

Encapsulation using struct and class

- **Issue in struct (Everything is Public by Default)**

```
struct Point {
    int x;  // Public by default
    int y;

    // Constructor
    Point(int a = 0, int b = 0) {
        if (a >= 0 && b >= 0) {
            x = a;
            y = b;
        } else {
            cout << "Invalid input!\n";
        }
    }
};
```

```
int main() {
    Point j;   // Default constructor
    j.x = -5;  // Allowed (since struct members are public)
    j.y = -10; // Allows negative values, leading to errors

    return 0;
}
```

Problem:

- **Since struct members are public by default, we can directly modify x and y, allowing incorrect values.**

- **Fixing the Issue with class (Encapsulation using Private Members)**

```
class Point {
private:
   int x;
   int y;

public:
   // Constructor with validation
   Point(int a = 0, int b = 0) {
      if (a >= 0 && b >= 0) {
         x = a;
         y = b;
      } else {
         cout << "Invalid input!\n";
         x = 0;
         y = 0;
      }
   }

   // Getter Methods
      int getX() { return x; }
      int getY() { return y; }

   // Setter Methods (with validation)
      void setX(int a) {
         if (a >= 0) x = a;
         else cout << "Invalid X
value!\n";
      }

      void setY(int b) {
         if (b >= 0) y = b;
         else cout << "Invalid Y
value!\n";
      }
};
int main() {
      Point j;   // Default constructor

      // j.x = -5;  // Error: x is private (Encapsulation working)
      j.setX(-5); // Output: Invalid X value!
      j.setY(10); // Correct

      cout << "X: " << j.getX();
      cout<< ", Y: " << j.getY() << "\n";
      return 0;
   }
```

- Encapsulation prevents direct modification of data.
- class is private by default, ensuring safety.
- Getters and Setters allow controlled access to private data.
- Validation inside constructors prevents invalid object creation.
- If a **constructor is private**, you **cannot create objects** directly.
- **Auxiliary functions** (helper functions) assist the main program but are not exposed externally.
- **Structures vs. Classes:**
  - **Structure (struct)**: Everything is **public by default**.
  - **Class (class)**: Everything is **private by default**.
- **Encapsulation ensures data security** by restricting direct access to class members.
- **Private data members** can only be accessed through **public getters and setters**.
- **Constructors validate inputs**, preventing invalid object states.
- **Encapsulation supports abstraction**, hiding implementation details.

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

class Point {
private:
    int x;
    int y;

public:
    // Constructor with input validation
    Point(int a = 0, int b = 0) {
        if (a >= 0 && b >= 0) {
            x = a;
            y = b;
        } else {
            cout << "Invalid input!\n";
            x = 0;
            y = 0;
        }
    }

    // Getter functions
    int getX() { return x; }
    int getY() { return y; }

    // Setter functions with validation
    void setX(const int a) {
        if (a >= 0) x = a;
        else cout << "Invalid X value!\n";
    }

    void setY(const int b) {
        if (b >= 0) y = b;
        else cout << "Invalid Y value!\n";
    }

    // Function to display coordinates
    void display() {
        cout << "(" << x << ", " << y << ")\n";
    }

// Function to calculate distance from another point
    double distance(const Point &p) {
        return sqrt(pow(x - p.x, 2) + pow(y - p.y, 2));
    }
};

int main() {
    Point p1(3, 4);
    Point p2(6, 8);

    p1.display();
    p2.display();

    cout << "Distance between points: " << p1.distance(p2) << endl;

    return 0;
}
```

Separating Class Definition (.h) and Implementation (.cpp)

- Header File (Point.h)
- Implementation File (Point.cpp)
- Main File (main.cpp)

1. Header File (Point.h)

```
#ifndef POINT_H
#define POINT_H

#include <iostream>
#include <cmath>

class Point {
private:
   int x;
   int y;

public:
   // Constructor
   Point(int a = 0, int b = 0);

   // Getter functions
   int getX() const;
   int getY() const;

   // Setter functions
   void setX(const int a);
   void setY(const int b);

   // Display function
   void display() const;

   // Function to calculate distance
   double distance(const Point &p) const;
};

#endif // POINT_H
```

2. Implementation File (Point.cpp)

```
#include "Point.h"
using namespace std;

// Constructor definition
Point::Point(int a, int b) {
   if (a >= 0 && b >= 0) {
      x = a;
      y = b;
   } else {
      cout << "Invalid input!\n";
      x = 0;
      y = 0;
   }
}

// Getter function definitions
int Point::getX() const { return x; }
int Point::getY() const { return y; }

// Setter function definitions
void Point::setX(const int a) {
   if (a >= 0) x = a;
   else cout << "Invalid X value!\n";
}

void Point::setY(const int b) {
   if (b >= 0) y = b;
   else cout << "Invalid Y value!\n";
}

// Display function definition
void Point::display() const {
   cout << "(" << x << ", " << y << ")\n";
}

// Distance function definition
double Point::distance(const Point &p) const {
   return sqrt(pow(x - p.x, 2) + pow(y - p.y, 2));
}
```

3. Main File (main.cpp)

```cpp
#include "Point.h"

int main() {
    Point p1(3, 4);
    Point p2(6, 8);

    p1.display();
    p2.display();

    std::cout << "Distance between points: " << p1.distance(p2) << std::endl;

    return 0;
}
```

Inline Functions in C++

- By default, all functions defined inside a class are **inline**.
- **Advantages**:
  - Eliminates function call overhead.
  - Helps in optimizing small, frequently used functions.
- **Important Note**: The compiler **decides** whether to inline a function; the inline keyword is a suggestion.

```
class Point {
public:
   inline void print() {
      cout << "X: " << x << ", Y: " << y;
   }
};
```

What are Inline Functions?

An **inline function** in C++ is a function for which the compiler replaces the function call with the actual function code during compilation. This avoids the overhead of a function call, improving performance for small, frequently used functions.

Why Use Inline Functions?

1. Function calls introduce overhead **Pushing arguments** onto the stack.
2. **Jumping to the function definition** and executing it.
3. **Returning control** back to the caller.

- due to:
- For small functions, this overhead can be more expensive than the function itself.
- **Inlining** eliminates this by inserting the function code directly at the call site.

Syntax of Inline Functions

To declare a function as inline, use the inline keyword before the function definition:

```
inline int add(int a, int b) {
    return a + b;
}
```

When add(x, y) is called, the compiler **replaces** it with x + y, instead of performing a function call.

Example of an Inline Function

```
class Math {
public:
    inline int square(int x) {
        return x * x;
    }
};

int main() {
    Math obj;
    cout << "Square of 5: " << obj.square(5) << endl;
    return 0;
}
```

How it Works?

- Instead of calling square(5), the compiler replaces it with 5 * 5 directly in main().
- This reduces execution time by eliminating function call overhead.

Inline Functions Inside a Class

- Functions defined **inside a class** are **implicitly inline** (even without the inline keyword).
- Example:

```
class Point {
public:
   int x, y;
   void show() {  // Implicitly inline
      cout << "X: " << x << ", Y: " << y << endl;
   }
};
```

- The compiler automatically treats show() as an inline function.

When to Use Inline Functions?

Use inline for:

- **Small, frequently used functions** (1-3 lines).
- **Getter functions** that return private members.
- **Operator overloading functions** for simple operations.

Avoid inline for:

- Functions with **loops, recursion, or complex logic**.
- Functions that use **static variables** (as multiple copies may cause issues).
- Functions that are **too large** (increases code size, leading to slower execution).

Advantages of Inline Functions

- **Reduces function call overhead** (especially for small functions).
- **Speeds up execution** for simple functions.
- **Improves performance** when used correctly.

Disadvantages of Inline Functions

- **Increases executable file size** (code duplication).
- **May lead to cache inefficiency** if used excessively.
- **Compiler ignores inline for complex functions** (loops, recursion).

- The **compiler decides** whether to inline a function (even if marked inline).
- Functions defined inside **a class are automatically inline**.
- Large functions should **not be inlined** (increases code size, reducing efficiency).
- **Inlining does not work with virtual functions**, as their calls are resolved at runtime.

What is the (this Pointer)?

- The (**this pointer)** in C++ is an implicit pointer available inside **non-static** member functions of a class. It points to the **current object** that invokes the function.

Properties of this Pointer

- It is an **implicit** pointer (automatically provided by the compiler).
- It stores the **memory address** of the calling object.
- It can be used to **differentiate instance variables from parameters** when they have the same name.
- The (this pointer) is **not available in static functions** because they belong to the class, not any specific object.

Understanding this Pointer

```
class Example {
private:
   int x;

public:
   void setX(int x) {
      this->x = x;  // Using 'this' to differentiate between instance variable and parameter
   }

   void display() {
      cout << "Value of x: " << this->x << endl;
   }
};

int main() {
   Example obj;
   obj.setX(10);
   obj.display();

   return 0;
}
```

Explanation

- setX(int x) has a **local variable** x that shadows the **instance variable** x.
- this->x refers to the **instance variable**, while x (without this) refers to the **parameter**.
- this ensures that we are assigning the parameter value to the **instance variable**.

Returning this Pointer

- The (this pointer) can be used to return the **current object**.

```
class Example {
private:
   int x;

public:
   Example& setX(int x) {
      this->x = x;
      return *this;  // Returning the current object
   }

   void display() {
      cout << "Value: " << x << endl;
   }
};

int main() {
   Example obj;
   obj.setX(20).display();  // Chained function calls
   return 0;
}
```

Explanation

- setX(int x) assigns a value and **returns *this** (the current object).
- This allows **method chaining**, where we call display() directly after setX(20).

(this Pointer) in Copy Constructor

- The (this pointer) is useful in **copy constructors** to avoid self-assignment issues.

```
class Example {
private:
   int x;

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

   Example(Example &obj) {  // Copy constructor
      this->x = obj.x;
   }

   void display() {
      cout << "Value: " << x << endl;
   }
};

int main() {
   Example obj1(50);
   Example obj2 = obj1;  // Calls copy constructor

   obj1.display();
   obj2.display();

   return 0;
}
```

Explanation

- The copy constructor Example(Example &obj) assigns values using this->x = obj.x.
- This ensures the new object gets the same value as obj1.

(this Pointer)

- The (this pointer) points to the **current instance** of the class.
- It is **automatically passed** to all non-static member functions.

```
class Point {
   int x, y;
public:
   void setX(int x) {
      this->x = x;
   }
};
```

Advantages of this Pointer

- Helps differentiate **instance variables** from local variables.
- Enables **method chaining** by returning the current object.
- Prevents **self-assignment** in operator overloading.
- Used in **copy constructors** to assign values correctly.