

HIMALAYA COLLEGE OF ENGINEERING

**Advanced C++ Programming Lab Report**

Lab 4: CLASS, OBJECTS, POINTER, STATIC, CONST AND FRIEND FUNCTION

**Prepared By**: Pranik Baral

**Subject:** Object-Oriented Programming(OOP)

**Program**: Bachelors of Electronics Engineering

**Institution**: Himalaya College of Engineering

**Date**: JUNE 6,2025

**Objectives:**

• To understand the use of objects as function arguments and return values.

• To implement and manipulate arrays and pointers to objects.

• To apply dynamic memory allocation for object management.

• To understand the use of friend functions and friend classes.

• To explore static and constant members in classes.

**Tools and Libraries Used**

• Programming Language: C++

• IDE: Code::blocks

• Libraries: #include <iostream>, #include <string>

**THEORY**

**OBJECT AS FUNCTION ARGUMENT**

In C++, objects can be passed to functions by value, reference, or pointer. This allows efficient data sharing and manipulation between functions and objects. The purpose of this is to pass data between functions, access object members and perform different operations like comparison, copying or modification.

It can be done in following ways:

1. PASS BY VALUE

In pass-by-value, a copy of the object is passed to the function, so any changes made inside the function do not affect the original object. This method invokes the copy constructor and can be inefficient for large objects due to the overhead of copying.

Syntax

void functionName(ClassName obj);

1. PASS BY REFRENCE

In pass-by-reference, the original object is passed using a reference (indicated by &), so changes made inside the function directly affect the original object. This method avoids copying, making it more efficient, especially for large objects.

Syntax

void functionName(ClassName &obj);

1. PASS BY POINTER

In pass-by-pointer, a pointer to the object is passed to the function, allowing indirect access to the original object. This method is useful for dynamic memory management and performing null checks.

Syntax

void functionName(ClassName \*obj);

**RETURNING OBJECTS FROM FUNCTIONS**

In C++, functions can return objects just like basic data types. This allows a function to perform operations and return a new object as the result. It supports features like method chaining, avoids modifying the original object, and helps in object-oriented programming. Syntax

class ClassName {

public:

ClassName functionName() {

ClassName obj;

// initialize or process obj

return obj; // return object

}

};

**ARRAY OF OBJECTS**

An array of objects is a collection of multiple objects of the same class stored in contiguous memory. It allows managing and processing many similar objects (like students or employees) efficiently using a single array and loops. Syntax

ClassName objectArray[arraySize];

**POINTER OF OBJECTS**

In C++, pointers can store the address of class objects, known as pointers to objects. You can access the object's members using the -> operator through the pointer. Objects can be created normally or dynamically using “new” operator.

Syntax

ClassName \*ptr;

ptr = &object;

ptr->memberFunction();

**DYNAMIC MEMORY ALLOCATION FOR OBJECTS**

Dynamic memory allocation assigns memory to objects at runtime using the ‘new’ keyword, and ‘delete’ is used to free that memory. It’s useful when the number of objects isn’t known beforehand, to save memory by creating objects only when needed, and for managing large or dynamic data structures.

Syntax

ClassName \*ptr = new ClassName();

ptr->memberFunction();

delete ptr;

**DYNAMIC MEMORY ALLOCATION FOR OBJECT ARRAYS**

Dynamic memory allocation for object arrays lets you create arrays of objects at runtime using pointers and the new operator. This allows flexibility in determining the array size based on user input or conditions, improving memory efficiency and management.

Syntax

ClassName\* ptr = new ClassName[size];

ptr[i].memberFunction();

delete[] ptr;

**DYNAMIC CONSTRUCTORS**

A dynamic constructor allocates memory at runtime using the new keyword to initialize objects whose size isn’t known at compile time. It improves flexibility and memory efficiency and is usually paired with a destructor to free the allocated memory.

SYNTAX

class ClassName {

datatype\* pointer;

public:

ClassName(parameters) {

pointer = new datatype[size]; // dynamic memory allocation

}

~ClassName() {

delete[] pointer; // release memory

}

};

**THIS POINTER**

The this pointer is a special, compiler-provided pointer available inside non-static member functions that points to the current object invoking the function. It helps differentiate between data members and function parameters when they share the same name. It allows access to the object's members using this->member.

Syntax

this->dataMember;

**STATIC DATA MEMBERS**

A static data member belongs to the class itself rather than any individual object, meaning only one shared copy exists for all objects. Declared with the static keyword, it must be defined outside the class. Static members are initialized before any object is created and persist for the program’s lifetime. Access to static members is controlled by access specifiers.

Syntax

class ClassName {

static data\_type variable\_name; // Declaration

...

};

// Definition outside the class

data\_type ClassName::variable\_name = value;

**STATIC MEMBER FUNCTIONS**

A static member function is declared with the static keyword and operates independently of any class object. It can be called using the class name without creating an object. Static functions can only access static data members and other static functions, as they lack a this pointer. They’re often used for tasks like tracking the number of objects created.

Syntax

class ClassName {

public:

static return\_type functionName(parameters);

};

return\_type ClassName::functionName(parameters) {

// function body

}

**CONSTANT MEMBER FUNCTIONS**

A constant member function, declared by adding const after its declaration and definition, guarantees it does not modify any data members of the class. This ensures data safety, especially when working with constant objects. Const functions can be called by both const and non-const objects, and making functions const helps prevent accidental changes to the object's data, improving code reliability.

Syntax

class ClassName {

public:

void functionName() const; // Declaration

};

void ClassName::functionName() const { // Definition

// Read-only function body

}

**CONSTANT OBJECTS**

A constant object is declared with the const keyword and its data members cannot be modified after creation. It can only call const member functions and must be initialized at declaration, typically via a constructor. Calling non-const functions on a constant object causes a compile-time error.

Syntax

const ClassName objectName(arguments);

**FRIEND FUNCTION**

In object-oriented programming, data hiding restricts access to private and protected members. However, friend functions are non-member functions declared with the friend keyword inside a class, granting them special access to the class’s private and protected members. They are invoked like normal functions and cannot be called using a class object.

Characteristics of it

 Not class members but can access private/protected data.

 Declared with ‘friend’ inside the class.

 Can be global or member of another class.

 Called like regular functions, not via objects.

Syntax

class ClassName {

private:

data\_type data;

public:

friend return\_type functionName(ClassName obj); // Friend function declaration

};

return\_type functionName(ClassName obj) {

// Access private/protected members of ClassName

}

**FRIEND CLASS**

A friend class is granted access to the private and protected members of another class that declares it as a friend. When Class A declares Class B as a friend, all members of Class B can access Class A’s private and protected data. Friendship is not mutual by default—Class B must explicitly declare Class A as a friend to get similar access. Friend classes enable closely related classes to share data without exposing members publicly.

Syntax

class ClassB; // Forward declaration of ClassB

class ClassA {

friend class ClassB; // Declare ClassB as a friend of ClassA

private:

int dataA; // Private data member

public:

ClassA(int x) : dataA(x) {} // Constructor

};

class ClassB {

public:

void showData(ClassA &a) {

// Access private member of ClassA because ClassB is a friend

std::cout << "Data from ClassA: " << a.dataA << std::endl;

}};