Object Oriented Programming – C++

Chapter 5: Pointers in C++

Engineering Semester Notes

5.1 Concepts of Pointer

A pointer is a variable that stores the memory address of another variable, rather than a data value directly. Pointers are one of the most powerful features of C++ because they allow a program to access and manipulate memory directly, pass large data structures efficiently between functions, build dynamic data structures such as linked lists and trees, and manage memory that is allocated at run time. Every variable in a program occupies a location in memory, and that location has an address; a pointer simply stores this address so that the data at that location can be accessed indirectly.

Pointer Declaration

A pointer must be declared with the data type of the variable it is going to point to, followed by an asterisk (*) and the pointer name. The data type tells the compiler how many bytes to read from the address stored in the pointer, and how to interpret that data.

```
int *p;      // pointer to an int
float *fp;   // pointer to a float
char *cp;    // pointer to a char
int *a, *b;  // both a and b are pointers to int
```

It is important to note that the * in a declaration is part of the pointer's type, not a multiplication. A pointer variable itself occupies a fixed number of bytes (commonly 4 or 8, depending on the system architecture), regardless of the data type it points to, because it only stores an address.

Pointer Operator (*) and Address Operator (&)

C++ provides two special unary operators that work together with pointers:

- Address-of operator (&): When placed before a variable name, it returns the memory address of that variable.
- Indirection / dereference operator (*): When placed before a pointer variable, it accesses (reads or writes) the value stored at the address the pointer holds. This is called dereferencing the pointer.

```
int x = 10;
int *p = &x;      // p now holds the address of x
cout << *p;       // prints 10 (value at address stored in p)
*p = 20;          // changes x to 20, through the pointer
```

Note the dual role of *: in a declaration (int *p) it means "p is a pointer to int"; in an expression (*p) it means "the value at the address held by p".

Pointer Expressions

Pointers can be used in expressions in several meaningful ways:

- Assignment: one pointer can be assigned the value of another pointer of the same type (p2 = p1;), so both point to the same location.
- Comparison: two pointers can be compared using ==, !=, <, > to test whether they point to the same location or to determine relative ordering within an array.
- Null pointer: a pointer that does not point to any valid location is assigned nullptr (or NULL in older code) and should always be checked before dereferencing.
- Pointer to pointer: C++ allows a pointer to store the address of another pointer, declared as int **pp;, which is useful in multi-level dynamic structures.

```
int x = 5;
int *p = &x;
int **pp = &p;      // pointer to pointer
cout << **pp;       // dereference twice to get x, prints 5
```

Pointer Arithmetic

Unlike ordinary variables, pointers support a restricted form of arithmetic that is closely tied to the size of the data type they point to. When a pointer is incremented or decremented, it does not move by one byte; it moves by the size of the type it points to, so that it correctly steps to the next element of an array.

- p++ or p + 1 moves the pointer forward by sizeof(type) bytes.
- p-- or p - 1 moves the pointer backward by sizeof(type) bytes.
- Subtracting two pointers of the same type gives the number of elements between them, not the number of bytes.
- Only addition and subtraction with integers are meaningful; multiplying or dividing two pointers is not allowed.

```
int arr[5] = {10, 20, 30, 40, 50};
int *p = arr;          // p points to arr[0]
cout << *p;            // 10
p++;                   // p now points to arr[1]
cout << *p;            // 20
cout << *(p + 2);      // 40  (arr[3])
```

This close relationship between pointers and arrays exists because an array name, when used in an expression, decays into a pointer to its first element. This is the reason array elements can be accessed either through subscript notation (arr[i]) or through pointer notation (*(arr + i)).

Pointers and Functions

When arguments are passed to a function, C++ offers different mechanisms that determine whether the function works on a copy of the data or on the original data itself. The two classical mechanisms relevant to pointers are call by value and call by reference.

Call by Value

In call by value, a copy of the actual argument's value is passed to the function's parameter. The function works entirely on this copy, so any modification made inside the function has no effect on the original variable in the calling program. This is the default parameter-passing mechanism in C++.

```
void addTen(int x) {
    x = x + 10;      // modifies only the local copy
}
int main() {
    int a = 5;
    addTen(a);
    cout << a;       // still prints 5
}
```

Call by Reference (using Pointers)

In call by reference implemented through pointers, the address of the actual argument is passed instead of its value. The function parameter is declared as a pointer, and the function dereferences it to access and modify the original variable directly. Any change made inside the function is reflected back in the calling program.

```
void addTen(int *x) {
    *x = *x + 10;    // modifies the original variable
}
int main() {
    int a = 5;
    addTen(&a);      // pass address of a
    cout << a;       // prints 15
}
```

C++ also provides reference variables (int &r = a;) as a more convenient, syntactically simpler alternative to pointer-based call by reference, but the underlying idea — allowing a function to operate on the caller's original data — is the same. Call by reference is essential when a function must modify multiple values, return more than one result, or avoid the overhead of copying large objects such as arrays and structures.

5.2 Pointers and Objects

Just as pointers can hold the address of ordinary variables, they can also hold the address of objects created from a class. Object pointers are widely used in C++ for dynamic object creation, for building linked data structures of objects, and for achieving runtime polymorphism through base-class pointers.

Pointers to Objects

A pointer to an object is declared using the class name followed by *, in the same way as any other pointer declaration. Once such a pointer is assigned the address of an object, its members are accessed using the arrow operator (->) instead of the dot operator (.), because the arrow operator automatically dereferences the pointer before accessing the member.

```
class Student {
public:
    int roll;
    void display() { cout << roll; }
};

Student s1;            // ordinary object
Student *p = &s1;      // pointer to object

p->roll = 101;         // same as (*p).roll = 101;
p->display();          // same as (*p).display();
```

Expressions p->roll and (*p).roll are exactly equivalent; the arrow operator is simply a more convenient and more commonly used shorthand. Object pointers are especially important when objects are created dynamically with new, since a dynamically created object has no name and can only be accessed through the pointer returned by new.

The this Pointer

Every non-static member function of a class receives, implicitly and automatically, a hidden pointer named this, which holds the address of the object through which the member function was invoked. Programmers do not declare this; it is generated automatically by the compiler for every call to a non-static member function.

- this allows a member function to refer to the calling object's own address.
- this is commonly used to distinguish a class member from a parameter that has the same name (e.g., this->x = x;).
- this is used to return the current object by reference from a member function, which enables method chaining (return *this;).
- this is used to compare the addresses of two objects, or to link an object to itself in certain data structures.

```
class Box {
    int side;
public:
    Box& setSide(int side) {
        this->side = side;   // 'this->side' is the member,
```

```
                            // 'side' is the parameter
        return *this;       // returns the current object
    }
};
```

Because this is only defined inside non-static member functions (a static member function has no calling object, so it has no this), it cannot be used there.

Pointer to Derived Classes

In C++, a pointer of the base-class type is allowed to point to an object of a derived class, because a derived-class object contains a complete base-class sub-object within it. This rule is fundamental to achieving runtime polymorphism.

- A base-class pointer can point to a derived-class object directly, without any explicit cast.
- Through a base-class pointer, only the members declared in the base class are normally accessible, unless those members are declared virtual, in which case the derived class's overridden version is called instead (dynamic binding).
- A derived-class pointer cannot directly point to a base-class object without an explicit downcast, since a base object may not contain the additional members added by the derived class.

```
class Animal {
public:
    virtual void sound() { cout << "Some sound"; }
};
class Dog : public Animal {
public:
    void sound() override { cout << "Bark"; }
};

Animal *a = new Dog();   // base pointer to derived object
a->sound();              // prints "Bark" due to virtual dispatch
```

This mechanism — a base-class pointer bound at run time to the correct derived-class function — is the foundation of virtual functions and polymorphism in object-oriented programming, and it depends entirely on the ability of a base pointer to reference a derived object.

5.3 Memory Management through Pointers

Memory used by a program is broadly divided into stack memory, which is allocated automatically for local variables and released when they go out of scope, and heap (free store) memory, which must be requested and released explicitly by the programmer. Pointers are the only means of accessing heap memory, since heap memory has no variable name of its own. C++ provides both its own operators (new and delete) and a set of C-style library functions (malloc(), calloc(), free()) for this purpose.

The new Operator

The new operator allocates memory from the heap at run time and returns a pointer to the allocated memory of the correct type, so no explicit type cast is required. It also automatically calls the constructor when used to create an object. If memory allocation fails, new throws a bad_alloc exception (unlike malloc(), which returns NULL).

```
int *p = new int;           // allocate single int
int *arr = new int[10];     // allocate array of 10 ints
Student *s = new Student(); // allocate object, constructor runs
```

The delete Operator

Memory obtained with new must be released explicitly with delete once it is no longer needed; otherwise the memory remains reserved for the lifetime of the program, causing a memory leak. delete also automatically calls the destructor when used on an object pointer.

```
delete p;        // free memory allocated for a single int
delete[] arr;    // free memory allocated for an array (note the [])
delete s;        // destructor of Student runs, then memory is freed
```

A very common and serious error is to use delete on memory allocated with new[], or delete[] on memory allocated with plain new — the array form ([]) must always match the allocation form. After deleting a pointer, it is good practice to set it to nullptr, since using a pointer after its memory is released (a dangling pointer) leads to undefined behaviour.

malloc(), calloc(), and free() (C-style Memory Management)

C++ also supports the older C library functions, declared in <cstdlib>, which manage memory in raw bytes rather than typed objects:

- malloc(size): allocates size bytes of uninitialised memory from the heap and returns a void* pointer, which must be explicitly cast to the required type. It does not initialise memory and does not call any constructor.
- calloc(n, size): allocates memory for n elements of size bytes each (total n * size bytes) and initialises every byte to zero, unlike malloc().
- free(ptr): releases memory previously allocated with malloc() or calloc(). It must never be used on memory allocated with new, just as delete must never be used on memory allocated with malloc()/calloc().

```
int *p = (int*) malloc(5 * sizeof(int));   // 5 ints, uninitialised
int *q = (int*) calloc(5, sizeof(int));    // 5 ints, all zero
free(p);
free(q);
```

Because malloc()/calloc()/free() do not know about constructors and destructors, they are considered unsuitable for allocating class objects in C++; new and delete are strongly preferred in modern C++ programs, and the two families of functions should never be mixed on the same block of memory.

Aspect	**new / delete**	**malloc / calloc / free**
Return type	Correct pointer type (no cast needed)	void* (explicit cast needed)
Constructor / Destructor	Called automatically	Not called
On failure	Throws bad_alloc	Returns NULL
Operator/Function	Operators (can be overloaded)	Library functions

Member Dereferencing Operators

C++ provides a set of operators specifically for accessing members of a class through an object or a pointer:

- Dot operator (.): accesses a member directly through an object (obj.member).
- Arrow operator (->): accesses a member through a pointer to an object (ptr->member); it is equivalent to (*ptr).member.
- Pointer-to-member operator (.*): used with an object (or reference) together with a pointer to a member, to access that member through the object (obj.*ptrToMember).
- Pointer-to-member operator (->*): used with a pointer to an object together with a pointer to a member, to access that member through the pointer (objPtr->*ptrToMember).

```
class Test {
public:
    int val = 10;
};

int Test::*pm = &Test::val;   // pointer to member 'val'
Test t;
Test *pt = &t;

cout << t.*pm;      // access via object  -> 10
cout << pt->*pm;    // access via pointer  -> 10
```

The pointer-to-member operators (.* and ->*) are used far less often than the ordinary dot and arrow operators, but they are important in advanced designs where a member (data or function) needs to be selected dynamically at run time rather than being fixed in the source code.

Key Points to Remember

- A pointer stores an address; & gets an address, * dereferences it.
- Pointer arithmetic scales automatically by sizeof(type).

- Call by reference through pointers lets a function modify the caller's original data.
- Objects are accessed through pointers using ->; this-> distinguishes members from parameters.
- A base-class pointer can point to a derived-class object — the basis of runtime polymorphism.
- Always match new with delete, and new[] with delete[]; never mix with malloc()/free().
- malloc() leaves memory uninitialised; calloc() zero-initialises it; neither calls constructors.

Multiple Choice Questions (MCQs)

Answer the following questions. Each question carries one mark. The answer key is provided at the end of this section.

1. What does a pointer variable store?

(a) The value of another variable
(b) The memory address of another variable
(c) The data type of a variable
(d) The name of a variable

2. Which operator is used to obtain the address of a variable?

(a) *
(b) &
(c) ->
(d) ::

3. What is the correct declaration of a pointer to an int?

(a) int p*;
(b) int *p;
(c) pointer int p;
(d) *int p;

4. If p is a pointer to an int, what does *p represent?

(a) The address stored in p
(b) The size of the int
(c) The value stored at the address held by p
(d) A new pointer

5. If int *p = arr; (arr is an int array) and p is incremented by 1, by how many bytes does the underlying address actually advance?

(a) 1 byte
(b) sizeof(int) bytes
(c) sizeof(pointer) bytes
(d) It does not advance

6. In call by value, changes made to a parameter inside a function:

(a) Affect the original argument
(b) Do not affect the original argument
(c) Cause a compile error
(d) Are stored permanently

7. Call by reference using pointers is achieved by passing:

(a) A copy of the value
(b) The address of the variable
(c) The name of the function
(d) Nothing

8. Which operator is used to access a member of an object through a pointer?

(a) .
(b) ->
(c) &
(d) ::

9. The 'this' pointer inside a member function holds:

(a) The address of the class
(b) The address of the calling object
(c) The address of the first object created
(d) Nothing until explicitly assigned

10. Which of the following statements about 'this' is TRUE?

(a) 'this' is available in static member functions
(b) 'this' must be declared by the programmer
(c) 'this' is available only in non-static member functions
(d) 'this' stores a value, not an address

11. A base-class pointer pointing to a derived-class object is primarily used to achieve:

(a) Encapsulation
(b) Runtime polymorphism
(c) Data hiding
(d) Operator overloading

12. Which operator is used to allocate memory dynamically in C++ and also calls the constructor for an object?

(a) malloc()
(b) calloc()
(c) new
(d) alloc()

13. What happens if memory allocation with 'new' fails?

(a) It returns NULL
(b) It returns -1
(c) It throws a bad_alloc exception
(d) The program continues silently

14. Which function allocates memory and also initialises it to zero?

(a) malloc()
(b) calloc()
(c) free()
(d) new

15. Which of the following pairs is correctly matched for releasing dynamically allocated memory?

(a) malloc() with delete
(b) new with free()
(c) new[] with delete[]
(d) calloc() with delete

Answer Key

1-(b) 2-(b) 3-(b) 4-(c) 5-(b) 6-(b) 7-(b) 8-(b) 9-(b) 10-(c) 11-(b) 12-(c) 13-(c) 14-(b) 15-(c)

Broad / Descriptive Questions

Answer any of the following questions in detail, with programs and diagrams wherever applicable.

1. Explain the concept of a pointer with a suitable example. How is a pointer declared and initialised in C++?
2. Differentiate between the address operator (&) and the dereference operator (*), with the help of a program.
3. What is pointer arithmetic? Explain how incrementing and decrementing a pointer behaves differently for different data types.
4. Explain call by value and call by reference with the help of programs. What are the advantages of call by reference?
5. What is an object pointer? How are members of a class accessed through a pointer to an object? Illustrate with an example.
6. Explain the 'this' pointer in detail. Why is it needed, and in which situations is it commonly used?
7. Explain, with an example, how a base-class pointer can point to a derived-class object, and how this supports runtime polymorphism.
8. Compare the new and delete operators with the malloc(), calloc(), and free() functions of C, highlighting at least four points of difference.
9. What are dangling pointers and memory leaks? Explain how improper use of new/delete or malloc/free can lead to these problems.
10. Explain the member dereferencing operators (., ->, .*, ->*) in C++ with suitable examples for each.