1.5. POINTERS

Definition

A pointer is a variable that stores the memory address of another variable. Instead of storing a value directly, it stores the *location* where the value is kept.

Syntax:

```
data_type *pointer_name;
```

Example:

```
int *ptr;
float *fptr;
char *cptr;
```

Initialization of Pointer

```
int a = 20;
int *p = &a;   // p stores the address of a
```

Features of Pointers

- A pointer stores the address of another variable.
- The * operator is used to access the value stored at the address (dereferencing).
- The & operator is used to get the address of a variable.
- Pointers allow dynamic memory handling.
- They support pointer arithmetic (e.g., p++).

Dereferencing a Pointer

Dereferencing means accessing the value stored at the address held by the pointer.

```
                cout << *p;  // prints 20
```

Example:

```
#include <iostream>
class PointerDemo
{
public:
    int a;                        // normal variable
    int *p;                        // pointer variable
    void assign( )
  {
        a = 10;
        p = &a;                                        // pointer stores address of a
    }
    void display( )
{
        cout << "Value of a = " << a << endl;
        cout << "Address of a = " << p << endl;
        cout << "Value using pointer = " << *p << endl;
    }
};
void main()
{
    PointerDemo O;                                        // object
    O.assign( );
    O.display( );
}
```

Pointer Arithmetic

Pointers can be incremented or decremented.

- p++ → moves to the next memory location of the same data type

- p-- → moves to the previous memory location

Example:

```
int *p;
p++;              // moves by 4 bytes (size of int)
```

Null Pointer

A null pointer is a pointer that points to nothing. A **null pointer** is a pointer that does not point to any valid memory location. nullptr ensures the pointer is not pointing to garbage memory.

```
int *p = NULL;
```

Void Pointer

A void pointer is a general-purpose pointer that can store the address of any data type.

```
syntax: void *ptr;
```

It must be type-casted before dereferencing.

Example:

```
int x = 10;

float y = 5.5;

void *ptr;

ptr = &x;

cout << "Integer value: " << *(int*)ptr << endl;

ptr = &y;
```

```
        cout << "Float value: " << *(float*)ptr << endl;
```

Output:

```
        Integer value: 10

        Float value: 5.5
```

Dangling Pointer

A pointer that points to memory location that has been freed or deleted is called a dangling pointer.

```
        int *p = new int(10);

        delete p;                        // memory freed

        p = NULL;                  // pointer reset (avoids dangling pointer)
```

Pointers and Arrays

An array name itself acts like a pointer to its first element.

```
            int a[3] = {10, 20, 30};
            int *p = a;            // same as &a[0]
            cout << *(p+1);              // prints 20
```

Pointers to Functions

Pointers can store addresses of functions.

```
            int add(int x, int y);
            int (*fp)(int, int) = add;
```

Advantages of Pointers

- Useful for dynamic memory allocation.

- Improve efficiency by passing large data structures by reference.
- Enable implementation of data structures like linked lists, trees, and graphs.
- Help in accessing array elements efficiently.

Disadvantages of Pointers

- Incorrect pointer handling can cause crashes.
- Dangling pointers may lead to unpredictable behavior.
- Pointer misuse can cause memory leaks.
- Complex to understand for beginners.