Unit-2

Pointers, References and Dynamic Memory Allocation

Pointers in C++

Pointers are a fundamental concept in C++ that enable direct memory access. They allow you to efficiently manage and manipulate data. This document explores pointers, their syntax, usage, and important concepts.

---

1. Introduction to Pointers

A pointer is a variable that stores the memory address of another variable. Each pointer has a specific type, and it points to a variable of that type.

Syntax

*type *pointer_name;*

- type: The type of data the pointer will point to (e.g., int, float).
- (*) Indicates that the variable is a pointer.
- pointer_name: The name of the pointer.

Example 1

```
int x = 10;
int *p = &x;
cout << *p;
// Output: 10
```

Explanation: p stores the address of x, and dereferencing it with *p accesses the value at that address.

Example 2

```
int a = 5;
int *ptr = &a;
cout << ptr;
// Output: address of variable a
```

Explanation: ptr holds the address of a, and printing ptr outputs the address where a is stored.

---

2. Declaring and Initializing Pointers

To declare a pointer, specify the data type followed by an asterisk (*). You can initialize a pointer by assigning it the address of a variable.

Example 1

```
int x = 10;
int *p = &x;
```

Explanation: p is a pointer to an integer and is initialized to the address of x.

Example 2

```
double y = 3.14;
double *q = &y;
```

Explanation: q is a pointer to a double and holds the address of y.

---

3. Pointer Dereferencing

Dereferencing a pointer means accessing the value stored at the memory address the pointer points to. This is done using the * operator.

Example 1

```
int x = 10;
int *p = &x;
*p = 20;
cout << x;
// Output: 20
```

Explanation: Dereferencing p with *p modifies the value of x indirectly through the pointer.

Example 2

```
int num = 30;
int *ptr = &num;
cout << *ptr;
// Output: 30
```

Explanation: Dereferencing ptr gives the value stored at the address it points to, which is 30.

---

4. Pointer Arithmetic

Pointers support arithmetic operations, allowing you to navigate through elements in an array or memory block.

Example 1

```
int arr[] = {10, 20, 30};
int *p = arr;
```

```
p++;
cout << *p;
// Output: 20
```

Explanation: p++ moves the pointer to the next element, and dereferencing it gives 20.

Example 2

```
int arr[] = {5, 10, 15};
int *p = arr;
cout << *(p + 2);
// Output: 15
```

Explanation: p + 2 moves the pointer to the third element, and dereferencing it gives 15.

---

5. Null and Uninitialized Pointers

Uninitialized pointers contain garbage values, which may lead to undefined behavior. Null pointers are explicitly set to nullptr, indicating they do not point to any valid memory.

Example 1 (Null Pointer)

```
int *p = nullptr;
if (p == nullptr)
cout << "Null pointer";
// Output: Null pointer
```

Explanation: A null pointer (nullptr) explicitly points to no valid memory.

Example 2 (Uninitialized Pointer)

```
int *p;
cout << *p;
// Undefined behavior
```

Explanation: Dereferencing an uninitialized pointer leads to undefined behavior.

---

6. Double Pointers (Pointer to Pointer)

A double pointer is a pointer that stores the address of another pointer. This allows for multi-level indirection.

Example 1

```
int x = 10;
int *p = &x;
int **pp = &p;
cout << **pp;
// Output: 10
```

Explanation: **pp accesses x by first dereferencing pp to get p, and then dereferencing p to get x.

Example 2

```
int a = 5;
int *p = &a;
int **pp = &p;
cout << **pp;
// Output: 5
```

Explanation: **pp dereferences pp twice to access the value of a.

---

7. Pointers and Arrays

In C++, arrays and pointers are closely related. The name of an array is a pointer to its first element.

Example 1

```
int arr[] = {10, 20, 30};
int *p = arr;
cout << *(p + 1);
// Output: 20
```

Explanation: p + 1 moves the pointer to the second element, and dereferencing it gives 20.

Example 2

```
int arr[] = {100, 200, 300};
int *p = arr;
cout << *(p + 2);
// Output: 300
```

Explanation: p + 2 moves the pointer to the third element, and dereferencing it gives 300.

---

8. Function Pointers

A function pointer is a pointer that stores the address of a function. It allows dynamic function calls.

Example 1

```
int add(int a, int b) {
return a + b;
}
int (*funcPtr)(int, int) = add;
cout << funcPtr(5, 3);
// Output: 8
```

Explanation: funcPtr stores the address of the add function and is used to call it.

Example 2

```
void greet() {
cout << "Hello, World!";
}
void (*greetPtr)() = greet;
greetPtr();
// Output: Hello, World!
```

Explanation: greetPtr is a pointer to the function greet and is called to print "Hello, World!".

---

9. Pointers to Structures

Pointers can be used to point to structures, allowing access to the members using the -> operator.

Example 1

```
struct Point {
int x, y;
};
Point p1 = {10, 20};
Point *ptr = &p1;
cout << ptr->x;
// Output: 10
```

Explanation: ptr->x accesses the x member of p1 via the pointer.

Example 2

```
struct Student {
string name;
int age;
};
Student s1 = {"John", 21};
Student *ptr = &s1;
cout << ptr->name;
// Output: John
```

Explanation: ptr->name accesses the name member of the structure s1.

---

10. Pointers and Dynamic Memory Allocation

C++ provides new and delete operators for dynamic memory allocation and deallocation.

Example 1

```
int *p = new int(25);
cout << *p;  // Output: 25
```

```
delete p;
```

Explanation: new allocates memory for an integer, and delete frees it after use.

Example 2

```
int *p = new int[3]{1, 2, 3};
cout << p[1];  // Output: 2
delete[] p;
```

Explanation: new[] allocates memory for an array, and delete[] frees it after use.

---

11. Smart Pointers

Smart pointers, such as std::unique_ptr and std::shared_ptr, automatically manage memory and prevent memory leaks.

Example 1 (std::unique_ptr)

```
std::unique_ptr<int> p(new int(100));
cout << *p;
// Output: 100
```

Explanation: std::unique_ptr automatically deletes the memory when it goes out of scope.

Example 2 (std::shared_ptr)

```
std::shared_ptr<int> p1 = std::make_shared<int>(50);
cout << *p1;
// Output: 50
```

Explanation: std::shared_ptr shares ownership of the allocated memory and frees it when no references are left.

Dynamic Memory Allocation using new and delete Operators

C++ allows you to allocate memory dynamically during runtime using the new operator and release it using the delete operator.

Syntax:

new: Allocates memory from the heap.

- `pointer = new data_type;`

delete: Deallocates memory previously allocated by new.

- `delete pointer;`

Example:

```
#include <iostream>
using namespace std;
int main() {
int *ptr = new int; // Dynamically allocating memory
*ptr = 5; cout << *ptr << endl; // Output: 5
delete ptr; // Deallocating memory
return 0;
}
```

Explanation: new int allocates memory for an integer on the heap, and delete ptr deallocates it when no longer needed.

Output: 5

Inline Functions

Inline functions are functions defined with the keyword inline. They are expanded in place where they are called, rather than being invoked by the function call mechanism.

Syntax:

```
inline return_type function_name(parameters) { // body of the function }
```

Example:

```
#include <iostream>
using namespace std;
inline int square(int x) { return x * x; }
int main() {
cout << square(5) << endl; // Output: 25
return 0;
}
```

Explanation: The square function is marked as inline, and the compiler substitutes its body directly at the call site.

Output: 25

Function Overloading

Function overloading allows multiple functions with the same name but different parameters. The compiler decides which function to call based on the arguments.

Example:

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

void display(int i) {
cout << "Integer: " << i << endl;
}

void display(double d) {
cout << "Double: " << d << endl;
}

int main() {
display(5); // Output: Integer: 5
display(3.14); // Output: Double: 3.14
return 0;
}
```

Explanation: The compiler chooses the appropriate function (display) based on the argument type.

Output:

- Integer: 5
- Double: 3.14

Function with Default Arguments

In C++, functions can have default arguments, which are used when no argument is passed during the function call.

Example:

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

void display(int x = 5) {
cout << "Value: " << x << endl;
}

int main() {
display(); // Output: Value: 5
display(10); // Output: Value: 10
return 0;
}
```

Explanation: The default value 5 is used when no argument is provided.

Output:

- Value: 5
- Value: 10

Constructors and Destructors

Types of Constructors:

Default Constructor: A constructor with no parameters that initializes the object with default values.

```
class Box {

public:

int length;

Box() {

length = 10;

} // Default constructor

};
```

Parameterized Constructor: A constructor that takes parameters to initialize an object with custom values.

```
class Box {

public:

int length;

Box(int l) {

length = l;

} // Parameterized constructor

};
```

Copy Constructor: A constructor that initializes an object by copying data from another object of the same class.

```
class Box {

public:

int length;

Box(const Box &b) {

length = b.length;

} // Copy constructor

};
```

Destructor:

A destructor is a special member function that is invoked when an object is destroyed.

```
~Box() { // Destructor // Clean-up code }
```

Friend Functions and Friend Classes

Friend Function:

A friend function allows a non-member function to access private and protected members of a class.

Example:

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

class Box {
int length;
public:
Box(int l) : length(l) {}
friend void printLength(Box b); // Friend function declaration
};

void printLength(Box b) {
cout << "Length: " << b.length << endl; // Access private member
}
```

```
int main() {
Box box(10);
printLength(box); // Output: Length: 10
return 0;
}
```

Explanation: The friend function printLength can access the private length member of Box.

Output: Length: 10

Friend Classes:

A friend class allows another class to access private and protected members of the current class.

Example:

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

class Box {
int length;
public:
Box(int l) : length(l) {}
friend class Display; // Declare Display class as a friend
};

class Display {
public:
void print(Box b) {
cout << "Length: " << b.length << endl; // Access private member of Box
}
};

int main() {
Box box(10);
Display display;
display.print(box); // Output: Length: 10
return 0;
}
```

Explanation: The Display class is a friend of Box, allowing it to access the private length member.

Output: Length: 10

---

this Pointer

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

Example:

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

class Box {
public:
int length;
Box(int l) {
length = l;
}
void display() {
cout << "Length: " << this->length << endl; // Access through this pointer
}
};

int main() {
Box box(15);
box.display(); // Output: Length: 15
return 0;
}
```

Explanation: The this pointer is used to access the current object's length member.

Output: Length: 15

Operator Overloading

Operator overloading allows user-defined types to behave like built-in types by defining how operators work with those types.

Example: Overloading the + Operator

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

class Box {
public:
int length;
Box(int l) {
length = l;
```

```
}
Box operator + (const Box &b) {
return Box(length + b.length);
}
};

int main() {
Box box1(5), box2(10);
Box box3 = box1 + box2;
cout << box3.length << endl; // Output: 15
return 0;
}
```

Explanation: The + operator is overloaded to add the length of two Box objects.

Output: 15