

UNIT - V

Pointers, Dereferencing And Address Operators ,Pointer And Address Arithmetic , Array Manipulation Using Pointer, Modifying Parameters Inside The Function Using Pointers ,Array As Parameters

*******Pointers** are one of the core components of the C programming language. A pointer can be used to store the **memory address** of other variables, functions, or even other pointers. The use of pointers allows low-level memory access, dynamic memory allocation, and many other functionality in C.

What is a Pointer in C?

A pointer is defined as a derived data type that can store the address of other C variables or a memory location. We can access and manipulate the data stored in that memory location using pointers.

Syntax of C Pointers

The syntax of pointers is similar to the variable declaration in C, but we use the (*) **dereferencing operator** in the pointer declaration.

```
datatype * ptr;
```

where

- **ptr** is the name of the pointer.
- **datatype** is the type of data it is pointing to.

The above syntax is used to define a pointer to a variable. We can also define pointers to functions, structures, etc.

How to Use Pointers?

The use of pointers in C can be divided into three steps:

1. **Pointer Declaration**
2. **Pointer Initialization**
3. **Pointer Dereferencing**

1. Pointer Declaration

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the (*) **dereference operator** before its name.

Example

```
int *ptr;
```

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called wild pointers.

2. Pointer Initialization

Pointer initialization is the process where we assign some initial value to the pointer variable. We generally use the (&: ampersand) **addressof operator** to get the memory address of a variable and then store it in the pointer variable.

Example

```
int var = 10;  
int * ptr;  
ptr = &var;
```

We can also declare and initialize the pointer in a single step. This method is called **pointer definition** as the pointer is declared and initialized at the same time.

Example

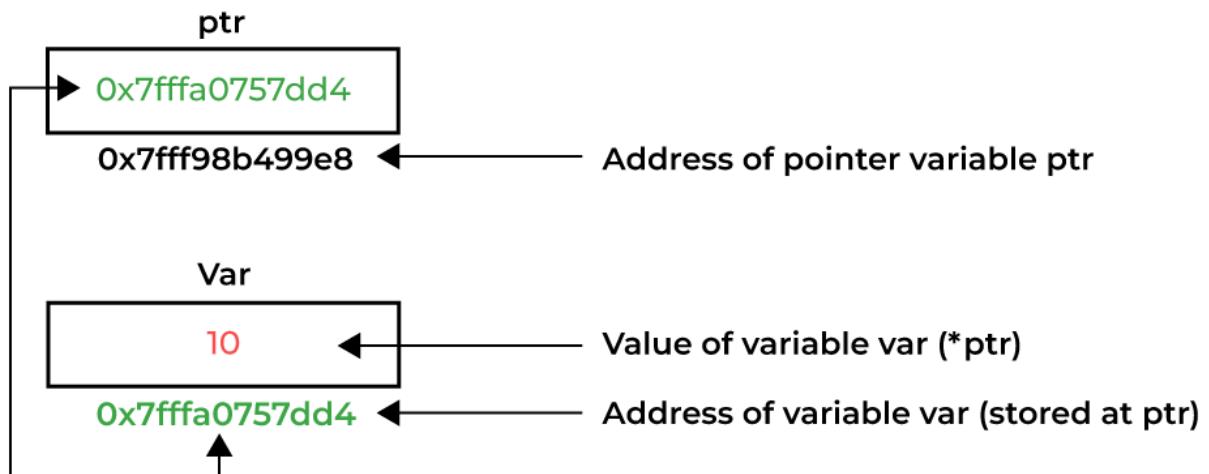
```
int *ptr = &var;
```

Note: It is recommended that the pointers should always be initialized to some value before starting using it. Otherwise, it may lead to number of errors.

3 Pointer Dereferencing

The dereference operator (*) is used to access the value stored at the memory location pointed to by a pointer.

Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer. We use the same (*) **dereferencing operator** that we used in the pointer declaration.



Dereferencing a Pointer in C

C Pointer Example

```
// C program to illustrate Pointers
#include <stdio.h>
void geeks()
{
    int var = 10;
    // declare pointer variable
    int* ptr
    // note that data type of ptr and var must be same
    ptr = &var;
    // assign the address of a variable to a pointer
    printf("Value at ptr = %p \n", ptr);
    printf("Value at var = %d \n", var);
    printf("Value at *ptr = %d \n", *ptr);
}
```

```
// Driver program
```

```
int main()
{
    geeks();
    return 0;
}
```

Output

```
Value at ptr = 0x7ffca84068dc
```

Value at var = 10

Value at *ptr = 10

Types of Pointers in C

Pointers in C can be classified into many different types based on the parameter on which we are defining their types. If we consider the type of variable stored in the memory location pointed by the pointer, then the pointers can be classified into the following types:

1. Integer Pointers
2. Array Pointer
3. Structure Pointer
4. Function Pointers
5. Double Pointers
6. NULL Pointer
7. Void Pointer
8. Wild Pointers
9. Constant Pointers
10. Pointer to Constant

*****Dereference Pointer in C

Accessing or manipulating the content that is stored in the memory address pointed by the pointer using dereferencing or indirection operator (*) is called dereferencing the pointer.

Syntax for Dereferencing a Pointer

We use the indirection operator (*) as the prefix to dereference a pointer:

`*(pointer_name)`

For modifying the data stored in the memory, we use

`*(pointer_name) = new_value;`

It is to be noted that the new_value must be of the same type as the previous.

Consider the above examples where **ptr** points to **num**, the content in the memory address can be accessed by the dereferencing operator *. Now, the *ptr will fetch the content stored in the address which is 10.

The num and ptr memory address and values will look like this.

Note: We have assumed that the architecture in the above example is byte addressable i.e. minimum unit that has a separate address is a byte.

Variable	Memory Address	Value
	202020	
	202021	10
	202022	
	202023	
ptr = &num	202024 - 202032	202020

Examples of Pointer Dereferencing

Example 1:Using a pointer to access and modify the value of an integer variable

```
// C Program to illustrate the dereferencing of pointer

#include <stdio.h>
int main()
{
    // Declare integer variable number
    int num = 10;
    // Declare pointer to store address of number
    int* ptr = &num;
    // Print the value of number
    printf("Value of num = %d \n", num);
    // Print Address of the number using & operator
    printf("Address of num = %d \n", &num);
    // Print Address stored in the pointer
    printf("Address stored in the ptr = %p \n\n", ptr);
    printf("Dereference content in ptr using *ptr\n\n");
    // Access the content using * operator
    printf("Value of *ptr = %d \n", *ptr);
    printf("Now, *ptr is same as number\n\n");
    printf("Modify the value using pointer to 6 \n\n");
    // Modify the content in the address to 6 using pointer
    *ptr = 6;
    // Print the modified value using pointer
    printf("Value of *ptr = %d \n", *ptr);
    // Print the modified value using variable
    printf("Value of number = %d \n", num);
    return 0;
}
```

Output

Value of num = 10

Address of num = 0x7ffe47d51b4c

Address stored in the ptr = 0x7ffe47d51b4c

Dereference content in ptr using *ptr

Value of *ptr = 10

Now, *ptr is same as number

Modify the value using pointer to 6

Value of *ptr = 6

Value of number = 6

***Address Operator & in C

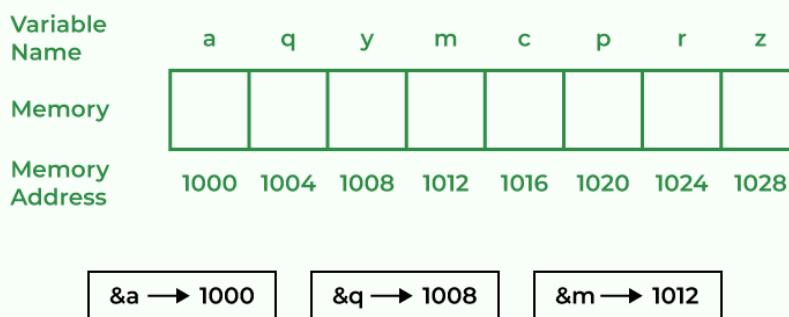
The **Address Operator** in C is a special unary operator that returns the address of a variable. It is denoted as the **Ampersand Symbol (&)**. This operator returns an integer value which is the **address of its operand** in the memory. We can use the address operator (&) with any kind of variables, array, strings, functions, and even pointers.

Syntax

The address operator is generally used as a prefix to its operand:

&operand

where **operand** can be a variable, array, function, pointer, etc.



Examples of Address Operators

Example 1:

Simple C example to demonstrate how to use the address operator in our program.

```
#include <stdio.h>

int main()
{
    // declaring a variable
    int x = 100;

    // printing the address of the variable
    printf("The address of x is %p", &x);

    return 0;
}
```

Output

The address of x is 0x7ffffe8f5591c

Address Operator Incompatible Entities in C

There are some entities in C for which we cannot use the address operator i.e. we cannot get the address of those entities in C. Some of them are:

1. Register Variables
2. Bit Fields
3. Literals
4. Expressions

Applications of Address Operator (&):

The address operator (&) is widely used in C programs to get the addresses of different entities. Some of the major and most common applications are:

1. Passing Pointers as Function Arguments
2. Pointer Arithmetic
3. Implementing Data Structures

*******Arithmetic Operations on Pointers in C**

Pointer Arithmetic is the set of valid arithmetic operations that can be performed on pointers. The pointer variables store the memory address of another variable. It doesn't store any value.

Hence, there are only a few operations that are allowed to perform on Pointers in C language. The C pointer arithmetic operations are slightly different from the ones that we generally use for mathematical calculations. These operations are:

1. Increment/Decrement of a Pointer
2. Addition of integer to a pointer
3. Subtraction of integer to a pointer
4. Subtracting two pointers of the same type
5. Comparison of pointers

1. Increment/Decrement of a Pointer

Increment: It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores **address 1000** is incremented, then it will increment by **4(size of an int)**, and the new address will point to **1004**. While if a float type pointer is incremented then it will increment by **4(size of a float)** and the new address will be **1004**.

Decrement: It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores **address 1000** is decremented, then it will decrement by **4(size of an int)**, and the new address will point to **996**. While if a float type pointer is decremented then it will decrement by **4(size of a float)** and the new address will be **996**.

Note: It is assumed here that the architecture is 64-bit and all the data types are sized accordingly. For example, integer is of 4 bytes.

To fully understand how to use pointers in complex data structures, the [C Programming Course Online with Data Structures](#) provides practical examples and in-depth explanations.

Example of Pointer Increment and Decrement

Below is the program to illustrate pointer increment/decrement:

```
#include <stdio.h>

// pointer increment and decrement

//pointers are incremented and decremented by the size of the data type they point to

int main()

{

    int a = 22;

    int *p = &a;

    printf("p = %u\n", p); // p = 6422288

    p++;

    printf("p++ = %u\n", p); //p++ = 6422292 +4 // 4 bytes

    p--;

    printf("p-- = %u\n", p); //p-- = 6422288 -4 // restored to original value


    float b = 22.22;

    float *q = &b;

    printf("q = %u\n", q); //q = 6422284

    q++;

    printf("q++ = %u\n", q); //q++ = 6422288 +4 // 4 bytes

    q--;

    printf("q-- = %u\n", q); //q-- = 6422284 -4 // restored to original value


    char c = 'a';
```

```

char *r = &c;

printf("r = %u\n", r); //r = 6422283

r++;

printf("r++ = %u\n", r); //r++ = 6422284 +1 // 1 byte

r--;

printf("r-- = %u\n", r); //r-- = 6422283 -1 // restored to original value

return 0;

}

```

Output

```

p = 1441900792
p++ = 1441900796
p-- = 1441900792
q = 1441900796
q++ = 1441900800
q-- = 1441900796
r = 1441900791
r++ = 1441900792
r-- = 1441900791

```

2. Addition of Integer to Pointer

When a pointer is added with an integer value, the value is first multiplied by the size of the data type and then added to the pointer.

For Example:

Consider the same example as above where the **ptr** is an **integer pointer** that stores **1000** as an address. If we add integer 5 to it using the expression, **ptr = ptr + 5**, then, the final address stored in the ptr will be **ptr = 1000 + sizeof(int) * 5 = 1020**.

Example of Addition of Integer to Pointer

// C program to illustrate pointer Addition

```
#include <stdio.h>

// Driver Code
int main()
{
    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores the address of N
    ptr1 = &N;
    ptr2 = &N;

    printf("Pointer ptr2 before Addition: ");
    printf("%p \n", ptr2);

    // Addition of 3 to ptr2
    ptr2 = ptr2 + 3;
    printf("Pointer ptr2 after Addition: ");
    printf("%p \n", ptr2);

    return 0;
}
```

Output

```
Pointer ptr2 before Addition: 0x7ffca373da9c
Pointer ptr2 after Addition: 0x7ffca373daa8
```

3. Subtraction of Integer to Pointer

When a pointer is subtracted with an integer value, the value is first multiplied by the size of the data type and then subtracted from the pointer similar to addition.

For Example:

Consider the same example as above where the **ptr** is an **integer pointer** that stores **1000** as an address. If we subtract integer 5 from it using the expression, **ptr = ptr - 5**, then, the final address stored in the ptr will be **ptr = 1000 - sizeof(int) * 5 = 980**.

Example of Subtraction of Integer from Pointer

Below is the program to illustrate pointer Subtraction:

// C program to illustrate pointer Subtraction

```
#include <stdio.h>
```

```
// Driver Code
```

```
int main()
```

```
{
```

```
    // Integer variable
```

```
    int N = 4;
```

```
    // Pointer to an integer
```

```
    int *ptr1, *ptr2;
```

```
    // Pointer stores the address of N
```

```
    ptr1 = &N;
```

```
    ptr2 = &N;
```

```
    printf("Pointer ptr2 before Subtraction: ");
```

```
    printf("%p \n", ptr2);
```

```
    // Subtraction of 3 to ptr2
```

```
    ptr2 = ptr2 - 3;
```

```
    printf("Pointer ptr2 after Subtraction: ");
    printf("%p \n", ptr2);

    return 0;
}
```

Output

```
Pointer ptr2 before Subtraction: 0x7ffd718ffebc
```

```
Pointer ptr2 after Subtraction: 0x7ffd718ffeb0
```

4. Subtraction of Two Pointers

The subtraction of two pointers is possible only when they have the same data type. The result is generated by calculating the difference between the addresses of the two pointers and calculating how many bits of data it is according to the pointer data type. The subtraction of two pointers gives the increments between the two pointers.

For Example:

Two integer pointers say **ptr1(address:1000)** and **ptr2(address:1004)** are subtracted. The difference between addresses is 4 bytes. Since the size of int is 4 bytes, therefore the **increment between ptr1 and ptr2** is given by **(4/4) = 1**.

Example of Subtraction of Two Pointer

Below is the implementation to illustrate the Subtraction of Two Pointers:

C

```
// C program to illustrate Subtraction

// of two pointers

#include <stdio.h>

// Driver Code

int main()

{
    int x = 6; // Integer variable declaration

    int N = 4;

    // Pointer declaration

    int *ptr1, *ptr2;
```

```

ptr1 = &N; // stores address of N
ptr2 = &x; // stores address of x

printf(" ptr1 = %u, ptr2 = %u\n", ptr1, ptr2);
// %p gives an hexa-decimal value,
// We convert it into an unsigned int value by using %u

// Subtraction of ptr2 and ptr1
x = ptr1 - ptr2;

// Print x to get the Increment
// between ptr1 and ptr2
printf("Subtraction of ptr1 "
"& ptr2 is %d\n",
x);

return 0;
}

```

Output

```
ptr1 = 2715594428, ptr2 = 2715594424
```

```
Subtraction of ptr1 & ptr2 is 1
```

5. Comparison of Pointers

We can compare the two pointers by using the comparison operators in C. We can implement this by using all operators in C $>$, \geq , $<$, \leq , $==$, $!=$. It returns true for the valid condition and returns false for the unsatisfied condition.

1. **Step 1:** Initialize the integer values and point these integer values to the pointer.
2. **Step 2:** Now, check the condition by using comparison or relational operators on pointer variables.
3. **Step 3:** Display the output.

Example of Pointer Comparision

```
// C Program to illustrate pointer comparision
```

```
#include <stdio.h>
```

```

int main()
{
    // declaring array
    int arr[5];

    // declaring pointer to array name
    int* ptr1 = &arr;

    // declaring pointer to first element
    int* ptr2 = &arr[0];

    if (ptr1 == ptr2) {
        printf("Pointer to Array Name and First Element "
               "are Equal.");
    }
    else {
        printf("Pointer to Array Name and First Element "
               "are not Equal.");
    }
}

return 0;
}

```

Output

Pointer to Array Name and First Element are Equal.

*******Array Manipulation Using Pointer**

In C, pointers can be used to manipulate arrays by storing the starting address of the array or creating an array of pointers. Here are some ways to use pointers to manipulate arrays

1 Accessing array elements

Pointers can be used to access array elements by treating the array name as a pointer to its first element. For example, `*(array + 2)` is equivalent to `array[2]`.

2 Changing array element values

Pointers can be used to change the value of array elements. For example, `*myNumbers = 13` changes the value of the first element in `myNumbers` to 13.

3 Iterating through arrays

Pointers can be used to iterate through arrays. For example, a loop can be used to increment the pointer along the array.

4 Storing base addresses

The base address of an array can be stored in another pointer variable. This allows the array to be manipulated using pointer arithmetic.

Here are some steps for accessing array elements using pointers:

Define an array and a pointer

Initialize the pointer to point to the array

Use the pointer to access array elements

Pointer arithmetic can be faster and more efficient than traditional array indexing in some situations

Incrementing the Value of Pointer in an Array

```
#include <stdio.h>
int main(void) {
    int scores[5] = {100, 235, 275, 50, 100};
    int *ptr = NULL;
    ptr = scores;
    printf("Value stored in pointer after increment is: %d", *++ptr);
}
```

Output

Value stored in pointer after increment is: 235

******* Modifying Parameters Inside The Function Using Pointers**

To modify parameters inside a function using pointers, you can pass the address of the variable to the function as a pointer:

1. Declare the parameter as a pointer type

In the function's parameter list, specify the pointer type.

For example, `void updateValue(int *p)`.

2. Pass the address of the variable

In the calling function, pass the address of the variable

. For example, `int main() { int var = 20; updateValue(&var); }`.

3. Modify the value

Within the function, modify the value of the variable pointed to by the pointer.

For example, `*p = 100;`

When you pass a pointer to a function, the function can affect the actual variables in the caller's scope. This is different from regular function parameters, which are pass-by-value. In this case, the function receives a copy of the argument value, and changes to the local parameter variable don't affect the original argument.

Pointers are variables that store memory addresses as their values. They are important for dynamic memory management, array manipulation, and implementing complex data structures

Example 1: Passing Pointers to Functions

```
#include <stdio.h>

void addOne(int* ptr) {
    (*ptr)++; // adding 1 to *ptr
}

int main()
{
    int* p, i = 10;
    p = &i;
    addOne(p);
    printf("%d", *p); // 11
    return 0;
}
```

Output 11

*******Array As Parameters**

Declaring a function parameter variable as an array really gives it a pointer type. C does this because an expression with array type, if used as an argument in a function call, is converted automatically to a pointer (to the zeroth element of the array). If you declare the corresponding parameter as an “array”, it will work correctly with the pointer value that really gets passed.

This relates to the fact that C does not check array bounds in access to elements of the array (see [Accessing Array Elements](#)).

For example, in this function,

```
void
clobber4 (int array[20])
{
    array[4] = 0;
}
```

the parameter array’s real type is `int *`; the specified length, 20, has no effect on the program. You can leave out the length and write this:

```
void
```

```
clobber4 (int array[])
{
    array[4] = 0;
}
```

or write the parameter declaration explicitly as a pointer:

```
void
clobber4 (int *array)
{
    array[4] = 0;
}
```

They are all equivalent.