# C Programming Short Notes

# Part 6

## 1. Pointer

- A pointer in C is a variable that stores the memory address of another variable.
- It "points to" the location of a value rather than holding the value itself.

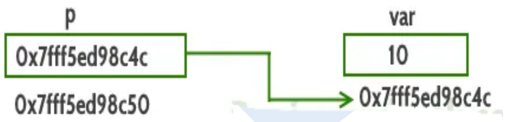
A. Pointers are declared by specifying the data type of the variable they point to, followed by an asterisk (*).

Declaration:

```
int *ptr;
```

B. Pointers can be initialized with the address of a variable using the address-of operator (&)

Initialization:



```
int num = 10;

int *ptr = &num; // Pointer ptr now holds the address of num
```

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

Dereferencing:

```
int num = 10;

int *ptr = &num;

printf("Value of num: %d\n", *ptr); // Output: Value of num: 10
```

D. Pointers can be manipulated using arithmetic operations like addition and subtraction. When arithmetic operations are performed on pointers, they move to the next or previous memory location based on the size of the data type they point to.

Pointer Arithmetic:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr; // Pointer to the first element of the array
ptr++; // Moves to the next element
```

**Array of Pointer:**

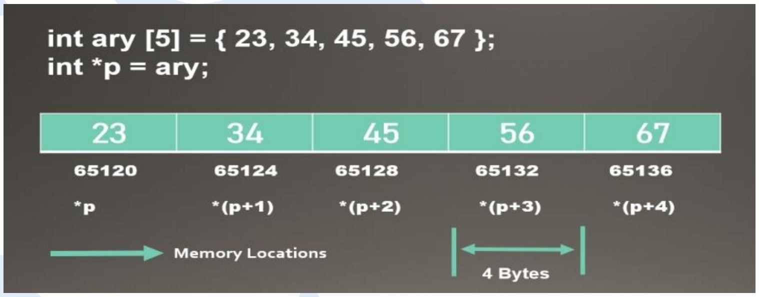
In C, arrays and pointers are closely related. An array name acts as a pointer to the first element of the array.

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr;
    printf("First element: %d\n", *ptr);
    for (int i = 0; i < 5; i++) {
        printf("Element %d: %d\n", i+1, *(ptr + i));
    }
    return 0;
}
```

Output

```
First element: 1
Element 1: 1
Element 2: 2
Element 3: 3
Element 4: 4
Element 5: 5
```



```
int ary [5] = { 23, 34, 45, 56, 67 };
int *p = ary;
```

| 23 | 34 | 45 | 56 | 67 |
|---|---|---|---|---|
| 65120 | 65124 | 65128 | 65132 | 65136 |
| *p | *(p+1) | *(p+2) | *(p+3) | *(p+4) |

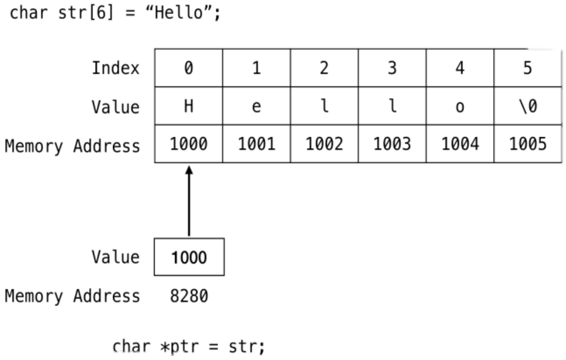
**Pointers with Strings:**

Strings in C are represented as arrays of characters. Pointers can be used to manipulate and access string elements.

```
#include <stdio.h>
int main() {
    char *str = "Hello, World!";
    printf("String: %s\n", str);
    while (*str != '\0') {
        printf("%c ", *str);
        str++;
    }
    return 0;
}
```

Output

```
String: Hello, World!
H e l l o ,   W o r l d !
```



```
char str[6] = “Hello”;
```

| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Value | H | e | l | l | o | \0 |
| Memory Address | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 |

| | |
|---|---|
| Value | 1000 |
| Memory Address | 8280 |

```
char *ptr = str;
```

**Pointers with Structures:**

Pointers can also be used with structures to manipulate and access structure members.

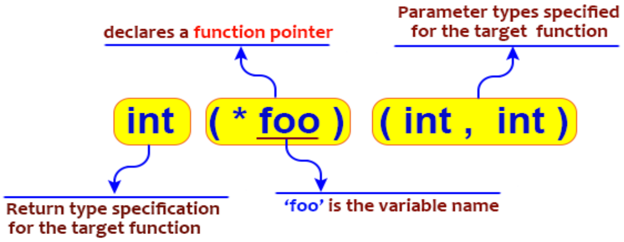
```
#include <stdio.h>
struct Point {
    int x;
    int y;
};
int main() {
    struct Point p1 = {5, 10};
    struct Point *ptr = &p1;
    printf("Coordinates: (%d, %d)\n", ptr->x, ptr->y);
    ptr->x = 20;
    ptr->y = 30;
    printf("New Coordinates: (%d, %d)\n", ptr->x, ptr->y);
    return 0;
}
```

```
Output
            Coordinates: (5, 10)
            New Coordinates: (20, 30)
```

**Pointers to Functions:**

Declare pointers to functions just like declare pointers to variables. This allows you to call different functions dynamically based on the pointer you have.

```
#include <stdio.h>
int add(int a, int b);
int subtract(int a, int b);
int main() {
    int (*ptr)(int, int);
    ptr = add;
    printf("Addition: %d\n", ptr(5, 3));
    ptr = subtract;
    printf("Subtraction: %d\n", ptr(5, 3));
    return 0;
}
int add(int a, int b) {
    return a + b;
}
int subtract(int a, int b) {
    return a - b;
}
```


declares a function pointer
Parameter types specified for the target function
int
( * foo )
( int , int )
Return type specification for the target function
'foo' is the variable name

```
                Output
                            Addition: 8 Subtraction: 2
```

**Functions with Pointers:**

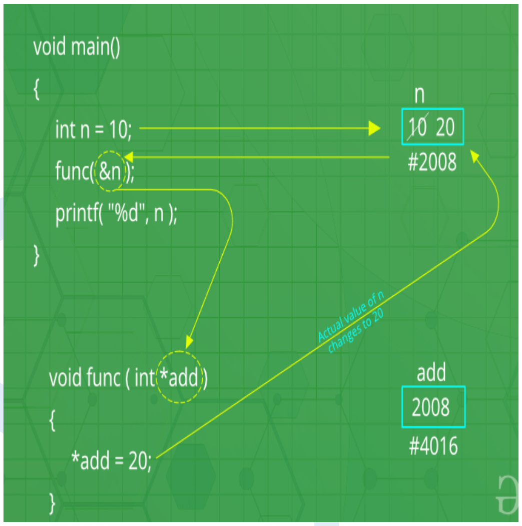
Functions in C can accept pointers as arguments, enabling them to directly modify variables outside of their scope or to operate on dynamically allocated memory.

```
#include <stdio.h>
void increment(int *ptr) {
    (*ptr)++;
}
int main() {
    int num = 5;

    printf("Before increment: %d\n", num);
    increment(&num);
    printf("After increment: %d\n", num);
    return 0;
}
```

Output

```
Before increment: 5
After increment: 6
```



```
void main()
{
  int n = 10;
  func( &n );
  printf( "%d", n );
}

void func ( int *add )
{
  *add = 20;
}
```

In this example, the increment function takes a pointer to an integer as an argument and increments the value it points to. By passing the address of the variable num to the increment function, we can modify num directly within the function.

**Types of pointers**

- ➢ Void pointers
- ➢ Wild pointers
- ➢ Null pointers
- ➢ Pointer to pointer
- ➢ Dangling pointer

**Void pointer**

- A pointers **can hold any address** like ( int, char, float, etc ).
- It’s also known as **generic pointer.**
- Its perform using **typecasting.**
- To store all datatype without declaration using single void pointer by using single void pointer by using typecast.

Syntax:

```
void  *variable_name
```

Example:

```
int main()
{
   void *p;
   int a=10;
   p=&a;
   printf("%d",*(int*)p);
}
```

**Wild pointer**

- **A pointer which is not initialized with any address/variables** is known as wild pointers.
- A wild pointer **holds the address of random memory location** and so it is also **bad pointer**.

Syntax:

```
datatype *variable_name
```

Example:

```
int main()
{
  int *p;
  printf("%d",p);

}
```

**NULL pointer**

- Its initialize with **NULL value at the time of declaration** is called NULL pointers.
- A NULL pointer doesn't refer to any valid address.
- The NULL pointer doesn’t to **any memory location**.
- The purpose of using **null pointer is handle errors**.

Syntax:

```
datatype*variable_naem=NULL;
```

Example:

```
int main()
{
   int *p;
   p=NULL;
   printf("%d",p);
}
```

**Dangling pointer**

- A pointer variable that **holds the address of inactive area location** this pointer is called dangling pointers.
- **A pointer which is pointing to a free memory location.**

Syntax:

```
*variable name;
```

**Pointer to pointer**

- A pointer which is stored the **address of another pointer**.
- It can implement the pointer to pointer operation up to 12 stages.
- Using pointer the **execution will be slower.**

Syntax:

```
int **variable_name;
```

Example:

```
int main()
{
   int a=10;
   int *p;
   int **q;
   p=&a;
   q=&p;
   printf("%d\t%d",p,**q);
}
```

We can use multiple pointers but c suggests **12 time of pointer** in the program because we initialize more pointer in the program, the  **execution time will be high** but more than 12 it's also possible to inside the program its execute.

Example:

```
int main()
{
   int a=10;
   int *b;
   int **c;
   int ***d;
   int ****e;
   int *****f;
   int ******g;
   int *******h;
   int ********i;
   int *********j;
   int **********k;
   int ***********l;
   int ************m;
   b=&a;
   c=&b;
   d=&c;
   e=&d;
   f=&e;
   g=&f;
   h=&g;
   i=&h;
   j=&i;
   k=&j;
   l=&k;
   m=&l;
   printf("%d\t%d",a,************m);
}
```