# Unit 9
# Pointers

## Introduction

- A pointer is a variable that stores the memory address of another variable as its value.
- A pointer variable points to a data type (like int) of the same type, and is created with the * operator.
- We can get the memory address of a variable with the reference operator &.

***Pointer Declaration***

- Pointer declaration is similar to other type of variable except asterisk (*) character before pointer variable name.

***Syntax:***

```
data_type *poiter_name;
```

***Example:***

```
int *ptr;
```

- Here, in this statement
    - ptr is the name of pointer variable (name of the memory blocks in which address of another variable is going to be stored).
    - The character asterisk (*) tells to the compiler that the identifier ptr should be declare as pointer.
    - The data type int tells to the compiler that pointer ptr will store memory address of integer type variable.
- Finally, ptr will be declared as integer pointer which will store address of integer type variable.

***Pointer Intialization***

- Pointer ptr is declared, but it not pointing to anything; now pointer should be initialized by the address of another integer variable.

***Example:***

```
int x;
int *ptr;
ptr=&x;
```

***Accessing address and value of x using pointer variable ptr***

- We can get the value of ptr which is the address of x (an integer variable)
    - ptr will print the stored value (memory address of x).
    - *ptr will print the value which is stored at the containing memory address in the ptr (value of variable x).

***Example:***

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

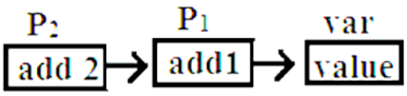
```c
int main()
{
        int x=20;        //int variable
        int *ptr;        //int pointer declaration

        ptr=&x;                //initializing pointer

        printf("Memory address of x: %p\n",ptr);
        printf("Value x: %d\n",*ptr);

        return 0;
}
```

***Output:***

```
Memory address of x: 0061FE98
Value x: 20
```

## Chain of pointers

- It is possible to make a pointer to point to another pointer, thus creating a chain of pointers.




- Here, the pointer variable p2 contains the add of the pointer variable P1, which points to the location that contains the desired value. This is known as multiple indirection's.
- A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name.

***Example:***

```c
#include<stdio.h>
int main()
{
        int x, *p1, **p2;
        x=100;
        p1=&x;
        p2=&p1;
        printf("x = %d", x);
        printf("\np1 = %d", p1);
        printf("\np2 = %d", p2);
        return 0;
}
```

***Output:***

```
x = 100
p1 = 6422168
p2 = 6422168
```

## Pointer Arithmetic

- Arithmetic operations can be performed on the pointers like addition, subtraction, etc.
- However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer.
- In pointer-from-pointer subtraction, the result will be an integer value.
- Following arithmetic operations are possible on the pointer in C language:
    - Increment
    - Decrement
    - Addition
    - Subtraction
    - Comparison

*Example:*

```
#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 = 6422160
p++ = 6422164
p-- = 6422160
q = 6422156
q++ = 6422160
q-- = 6422156
r = 6422155
r++ = 6422156
r-- = 6422155
```

## Pointers and Arrays

- Array elements can also be accessed using pointers.

*Example:*

```
#include <stdio.h>
int main()
{
        int i, x[6], sum = 0;
        printf("Enter 6 numbers: ");
        for(i = 0; i < 6; ++i)
        {
                // Equivalent to scanf("%d", &x[i]);
                scanf("%d", x+i);
                // Equivalent to sum += x[i]
                sum += *(x+i);
        }

        printf("Sum = %d", sum);
        return 0;
}
```

*Output:*

```
Enter 6 numbers: 10 20 30 40 50 60
Sum = 210
```

## Pointers and Character Strings

- String is a data type that stores the sequence of characters in an array.
- A string in C always ends with a null character (\0), which indicates the termination of the string.
- Pointer to string in C can be used to point to the starting address of the array, the first character in the array.

- These pointers can be dereferenced using the asterisk * operator to identify the character stored at the location.
- 2D arrays and pointer variables both can be used to store multiple strings.

***Example:***

```
#include<stdio.h>
int main()
{
        char str[11] = "HelloWorld";

        // pointer variable
        char *ptr = str;
        while (*ptr != '\0')
        {
                printf("%c", *ptr);
                // move to the next character.
                ptr++;
        }
        return 0;
}
```

***Output:***

```
HelloWorld
```

## Array of Pointers

- An array of pointers is similar to any other array in C Language.
- It is an array which contains numerous pointer variables and these pointer variables can store address values of some other variables having the same data type.

***Example:***

```
#include <stdio.h>
int main()
{
        char *fruits[5] = {"apple", "banana", "mango", "grapes", "orange"}, i;
        for(i = 0; i < 5; i++)
        {
                printf("%s\n", fruits[i]);
        }
        return 0;
}
```

***Output:***

```
apple
```

```
banana
mango
grapes
orange
```

## Pointers as Function Arguments

- See in chapter 7

## Function Returning pointers

- Pointers can be passed to the function as well as return pointer from a function.
- But it is not recommended to return the address of a local variable outside the function as it goes out of scope after function returns.

*Example:*

```c
// C program to illustrate the concept of
// returning pointer from a function
#include <stdio.h>

// Function returning pointer
int* fun()
{
	static int A = 10;
	return (&A);
}

// Driver Code
int main()
{
	// Declare a pointer
	int* p;

	// Function call
	p = fun();

	printf("%p\n", p);
	printf("%d\n", *p);
	return 0;
}
```

*Output:*

```
00403004
10
```

## Pointers and Structures

- There are two ways to access the member of the structure using Structure pointer:
    1. Using ( * ) asterisk or indirection operator and dot ( . ) operator.
    2. Using arrow ( -> ) operator or membership operator.

*Example*

```
#include <stdio.h>
struct person
{
        int age;
        float weight;
};

int main()
{
        struct person *personPtr, person1;
        personPtr = &person1;

        printf("Enter age: ");
        scanf("%d", &personPtr->age);

        printf("Enter weight: ");
        scanf("%f", &personPtr->weight);

        printf("Displaying:\n");
        printf("Age: %d\n", personPtr->age);
        printf("weight: %f", personPtr->weight);

        return 0;
}
```

*Output:*

```
Enter age: 22
Enter weight: 55
Displaying:
Age: 22
weight: 55.000000
```

## Dynamic Memory Allocation

- Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
- C provides some functions to achieve these tasks.
- There are 4 library functions provided by C defined under <stdlib.h> header file to facilitate dynamic memory allocation in C programming.

- They are:
    - malloc()
    - calloc()
    - free()
    - realloc()

## malloc() method

- The “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size.
- It returns a pointer of type void which can be cast into a pointer of any form.
- It doesn’t Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

*Syntax:*

ptr = (cast-type*) malloc(byte-size)

*Example:*

ptr = (int*) malloc(100 * sizeof(int));

## calloc() method

- “calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type.
- It is very much similar to malloc() but has two different points and these are:
    - It initializes each block with a default value ‘0’.
    - It has two parameters or arguments as compare to malloc().

*Syntax:*

ptr = (cast-type*)calloc(n, element-size);

here, n is the no. of elements and element-size is the size of each element.

*Example:*

ptr = (float*) calloc(25, sizeof(float));

## realloc() method

- “realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory.
- In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory.
- re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

*Syntax:*

ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.

## free() method

- "free" method in C is used to dynamically de-allocate the memory.
- The memory allocated using functions malloc() and calloc() is not de-allocated on their own.
- Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

*Syntax:*

```
free(ptr);
```

*Example:*

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int* ptr;
        int n, i;
        n = 5;
        printf("Enter number of elements: %d\n", n);

        // Dynamically allocate memory using calloc()
        ptr = (int*)calloc(n, sizeof(int));
        //ptr = (int*)malloc(n*sizeof(int));

        // Check if the memory has been successfully
        // allocated by malloc or not
        if (ptr == NULL)
        {
                printf("Memory not allocated.\n");
                exit(0);
        }
        else
        {
                // Memory has been successfully allocated
                printf("Memory successfully allocated using calloc.\n");

                // Get the elements of the array
                for (i = 0; i < n; ++i)
                {
                        ptr[i] = i + 1;
                }

                // Print the elements of the array
                printf("The elements of the array are: ");
                for (i = 0; i < n; ++i)
                {
```

```c
			printf("%d, ", ptr[i]);
		}
		// Get the new size for the array
		n = 10;
		printf("\n\nEnter the new size of the array: %d\n", n);

		// Dynamically re-allocate memory using realloc()
		ptr = (int *)realloc(ptr, n * sizeof(int));

		// Memory has been successfully allocated
		printf("Memory successfully re-allocated using realloc.\n");

		// Get the new elements of the array
		for (i = 5; i < n; ++i)
		{
			ptr[i] = i + 1;
		}

		// Print the elements of the array
		printf("The elements of the array are: ");
		for (i = 0; i < n; ++i)
		{
			printf("%d, ", ptr[i]);
		}

		free(ptr);
	}
	return 0;
}
```

***Output:***

```
Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,

Enter the new size of the array: 10
Memory successfully re-allocated using realloc.
The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

## Exercise

1. List any one advantage and disadvantage of the pointer. How do you pass pointers as function arguments? (5) [TU 2079]
2. Explain dynamic memory allocation with example. (5) [TU 2078]
3. Define pointer. Flow to you return pointers from functions? Explain with example. (5) [TU 2077]

4. What is dynamic memory allocation? Discuss the use of malloc() in dynamic memory allocation with example. (5) [TU 2075]
5. Define pointer. Discuss the relationship between pointer and one-dimensional array. (5) [TU 2074]