# Structured Programming using C

RCP2SFCES101

Unit-6

Pointers

## Contents

1. Introduction

2. Call by value, call by Reference

3. Pointer Arithmetic

4. Pointer to Array

5. References

# Introduction

## Pointer

- A pointer is a variable that **stores the memory address of another variable**.
- Pointers provide an indirect way to access and manipulate data stored in memory.
- The size of a pointer variable depends on the architecture of the system, not the data type it points to.
- On most systems:

  32-bit architecture: The size of a pointer is 4 bytes (32 bits).

  64-bit architecture: The size of a pointer is 8 bytes (64 bits).

# Pointer contd...

- Whenever we declare a variable, the system allocates a memory location to hold the value of that variable.
- The location will have its own address number

int a = 12;

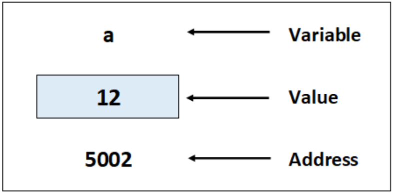
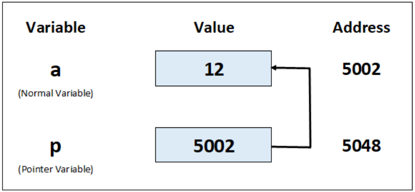


Figure: Representation of Variable

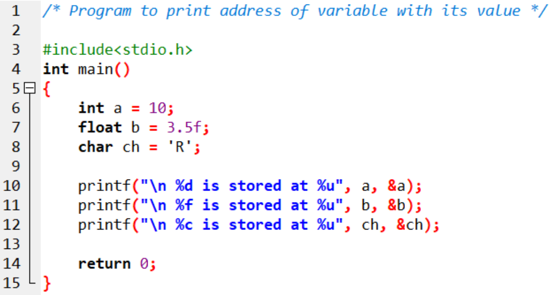


| Variable | Value | Address |
|---|---|---|
| a<br>(Normal Variable) | 12 | 5002 |
| p<br>(Pointer Variable) | 5002 | 5048 |

Figure: Pointer variable

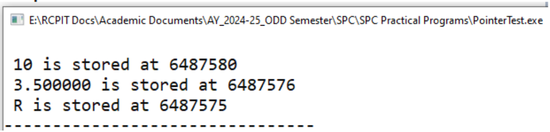
## Address Operator

- The address-of operator (&) is used to get the memory address of a variable.
- **Example:**



```c
/* Program to print address of variable with its value */

#include<stdio.h>
int main()
{
    int a = 10;
    float b = 3.5f;
    char ch = 'R';

    printf("\n %d is stored at %u", a, &a);
    printf("\n %f is stored at %u", b, &b);
    printf("\n %c is stored at %u", ch, &ch);

    return 0;
}
```

**Output:**



E:\RCPIT Docs\Academic Documents\AY_2024-25_ODD Semester\SPC\SPC Practical Programs\PointerTest.exe

```

10 is stored at 6487580
3.500000 is stored at 6487576
R is stored at 6487575
--------------------------------
```

## Declaration of Pointer variable

Following is syntax to declare pointer variable:

**Syntax:**

```
data_type *pt_name;
```

The above declaration tells the compiler three things about the variable pt_name.

1. The asterisk(*) tells that the variable pt_name is pointer variable
2. pt_name needs a memory location.
3. pt_name point to the variable of type data_type.

**Example:**

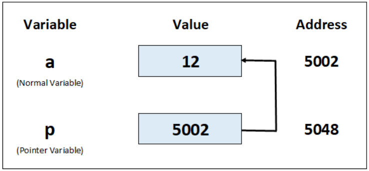
```
int *p;      //Declares p as pointer variable that point to variable of int data type
float *q;    //Declares q as pointer variable that point to variable of float data type
ch *r;       //Declares r as pointer variable that point to variable of char data type
```

# Initialization of Pointer variable

- The process of assigning the address of variable to pointer variable is known as pointer initialization.
- Once pointer variable is declared we can use assignment operator to initialize it.

**Example 1:**

```
int a = 12;
int *p;      //pointer declaration
 p = &a;     // Pointer initialization
```

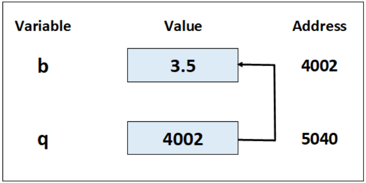


| Variable | Value | Address |
|---|---|---|
| a<br>(Normal Variable) | 12 | 5002 |
| p<br>(Pointer Variable) | 5002 | 5048 |

# Initialization of Pointer variable contd..

**Example 2:**

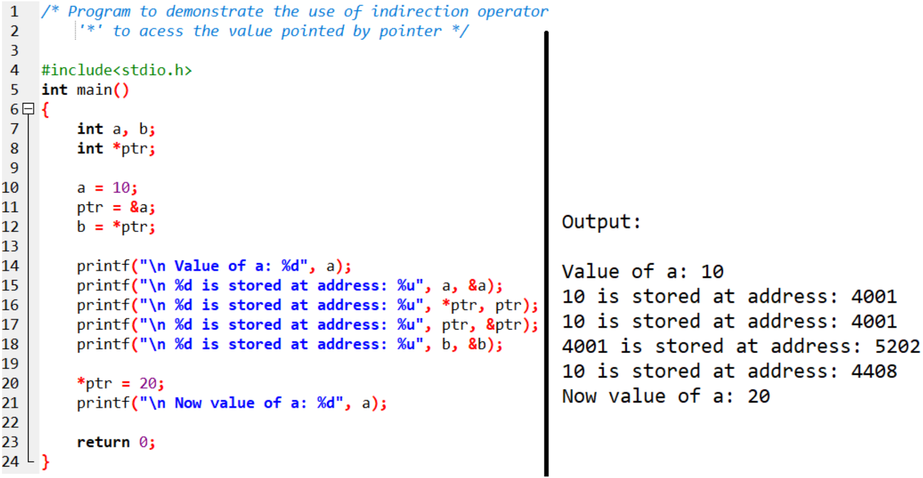
```
float b = 3.5f;
float *q;       //pointer declaration
 q = &b;        // Pointer initialization
```



## Accessing variable through its Pointer

- To access the value of variable using pointer **asterisk(*)** operator is used usually known as **indirection operator**.
- Another name of indirection operator is the ***dereferencing operator***
- The ***** can be remembered as **'value at the address'**.

# Accessing variable through its Pointer Example



```c
/* Program to demonstrate the use of indirection operator
    '*' to acess the value pointed by pointer */

#include<stdio.h>
int main()
{
    int a, b;
    int *ptr;

    a = 10;
    ptr = &a;
    b = *ptr;

    printf("\n Value of a: %d", a);
    printf("\n %d is stored at address: %u", a, &a);
    printf("\n %d is stored at address: %u", *ptr, ptr);
    printf("\n %d is stored at address: %u", ptr, &ptr);
    printf("\n %d is stored at address: %u", b, &b);

    *ptr = 20;
    printf("\n Now value of a: %d", a);

    return 0;
}
```

```
Output:

Value of a: 10
10 is stored at address: 4001
10 is stored at address: 4001
4001 is stored at address: 5202
10 is stored at address: 4408
Now value of a: 20
```

## Call by value, call by Reference

# Call by value, call by Reference

There are two ways to pass the parameter to the functions:

1. Call by value
2. Call by Reference

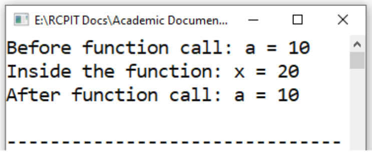
## Call by value

- In call by value, values of the actual parameter are copied to the variables in the parameter list of called function.
- The changes made to the parameters inside the called function do not affect the original values of actual parameter.

# Example: Call by value

```c
/*  Program to demonstrate passing parameter to function
    using-call by value */

#include <stdio.h>

void changeValue(int x)
{
    x = 20;      // Modifies the local copy
    printf("Inside the function: x = %d\n", x);
}

int main()
{
    int a = 10;

    printf("Before function call: a = %d\n", a);
    changeValue(a);       // Call by value
    printf("After function call: a = %d\n", a);

    return 0;
}
```

**Output:**

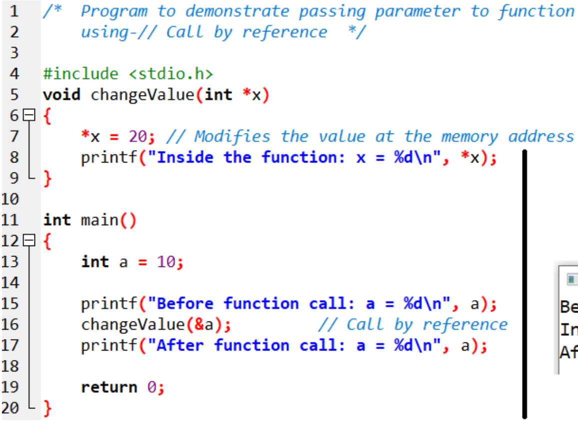


```
Before function call: a = 10
Inside the function: x = 20
After function call: a = 10

--------------------------------
```

## Call by Reference

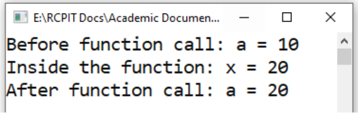
- In call by reference, the memory addresses of the actual parameter are passed to the called function.
- In this case called function directly works on the data in the calling function.
- Means, The changes made to the parameters inside the called function do affect the original values of actual parameter.

# Example: Call by Reference



```
/*  Program to demonstrate passing parameter to function
    using-// Call by reference  */

#include <stdio.h>
void changeValue(int *x)
{
    *x = 20; // Modifies the value at the memory address
    printf("Inside the function: x = %d\n", *x);
}

int main()
{
    int a = 10;

    printf("Before function call: a = %d\n", a);
    changeValue(&a);          // Call by reference
    printf("After function call: a = %d\n", a);

    return 0;
}
```

**Output**



```
Before function call: a = 10
Inside the function: x = 20
After function call: a = 20
```

# Pointer Arithmetic

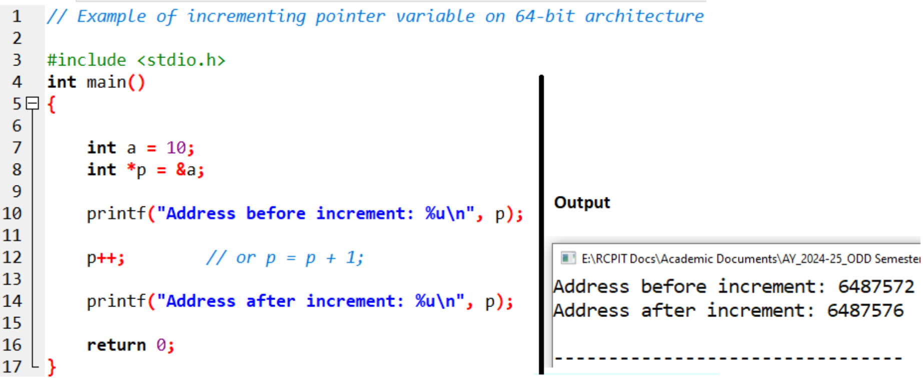
## Pointer Arithmetic

The following are the supported operations on the pointer:

- **Addition (+):** Increment a pointer.
- **Subtraction (-):** Decrement a pointer.
- **Difference (-):** Calculate the number of elements between two pointers.
- **Increment/Decrement (++/- -):** Move the pointer forward or backward by one element.

When we increment a pointer, its value is increased by the length of the data type to which it points. This length is called as **scale factor**.

# Pointer Arithmetic contd...



```c
// Example of incrementing pointer variable on 64-bit architecture

#include <stdio.h>
int main()
{

    int a = 10;
    int *p = &a;

    printf("Address before increment: %u\n", p);

    p++;        // or p = p + 1;

    printf("Address after increment: %u\n", p);

    return 0;
}
```

**Output**

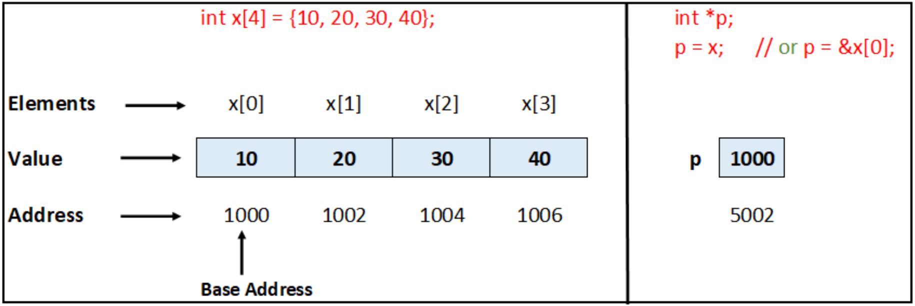
E:\RCPIT Docs\Academic Documents\AY_2024-25_ODD Semester

```
Address before increment: 6487572
Address after increment: 6487576

--------------------------------
```

# Pointer to Array

## Pointer to Array

- When an array is declared, the compiler allocates a base address and a sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- The base address is the location of the first element of the array.

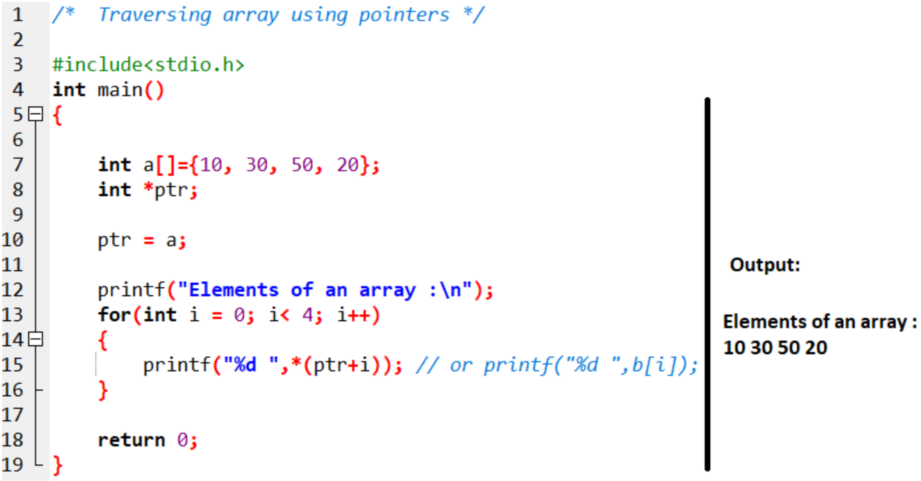


```
int x[4] = {10, 20, 30, 40};
```

| | | | | |
|---|---|---|---|---|
| **Elements** → | x[0] | x[1] | x[2] | x[3] |
| **Value** → | **10** | **20** | **30** | **40** |
| **Address** → | 1000 | 1002 | 1004 | 1006 |

```
int *p;
p = x;     // or p = &x[0];
```

# Traversing array using pointers



```c
/*  Traversing array using pointers */

#include<stdio.h>
int main()
{

    int a[]={10, 30, 50, 20};
    int *ptr;

    ptr = a;

    printf("Elements of an array :\n");
    for(int i = 0; i< 4; i++)
    {
        printf("%d ",*(ptr+i)); // or printf("%d ",b[i]);
    }

    return 0;
}
```

**Output:**

```
Elements of an array :
10 30 50 20
```

## NULLPointer

- A null pointer is a pointer that does not point to any valid memory location.
- It is often used to indicate that a pointer is not initialized.
- In most of the libraries the value of NULL pointer is **0**(**zero**)

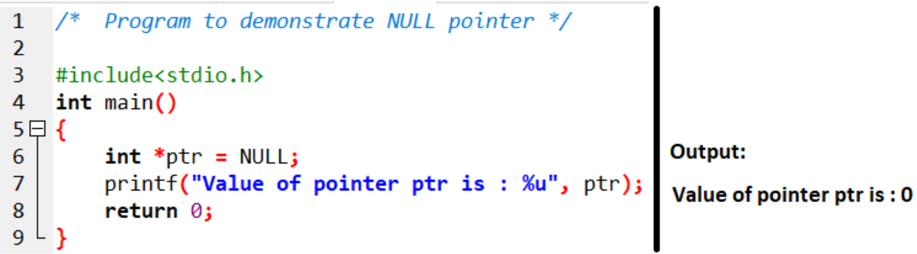
**Syntax:**

```
data_type *pt_name = NULL;
```

**Example:**

```
int *ptr = NULL;
```

# Example: NULL Pointer



```c
/*  Program to demonstrate NULL pointer */

#include<stdio.h>
int main()
{
    int *ptr = NULL;
    printf("Value of pointer ptr is : %u", ptr);
    return 0;
}
```

**Output:**

```
Value of pointer ptr is : 0
```

## Uses of Pointers

- Pointers allow for passing arguments by reference, enabling functions to modify the actual argument.
- Used to iterate through arrays and manipulate strings efficiently.
- Pointers are essential for allocating memory dynamically during program execution using functions like malloc, calloc and free from <stdlib.h>.
- Used for implementing data structures like linked lists, stacks, queues, and trees.
- Pointers allow for direct access and manipulation of memory addresses, enabling low-level programming.

# References

# References

- "MASTERING C" by K.R.Venugopal and Sudeep R.Prasad , Tata McGraw-Hill Publications
- “Programming in ANSI C”, by E. Balaguruswamy, Tata McGraw-Hill Education.
- “Let Us C”, by Yashwant Kanetkar, BPB Publication.
- “Programming in C”, by Pradeep Day and Manas Gosh, Oxford University Press.
- https://www.javatpoint.com/c-programming-language-tutorial
- https://www.tutorialspoint.com/cprogramming/index.htm
- https://www.geeksforgeeks.org/c-programming-language/
- https://www.youtube.com/playlist?list=PLdo5W4Nhv31a8UcMN9-35ghv8qyFWD9_S

Thank you