Pointers:-

In C, pointers are variables that store memory addresses. Declaring and initializing pointers involves defining a pointer variable and assigning it an address.

1. Declaring a Pointer:

To declare a pointer, you need to specify the type of data it will point to. The syntax is as follows:

```
type *pointer_name;
```

-type is the data type the pointer will point to (e.g., int, char, float).

-pointer_name is the name of the pointer.

Example:

```
int *ptr;   // Declares a pointer to an integer
char *ch;   // Declares a pointer to a character
```

2. Initializing a Pointer:

To initialize a pointer, you assign it the address of a variable of the same type.

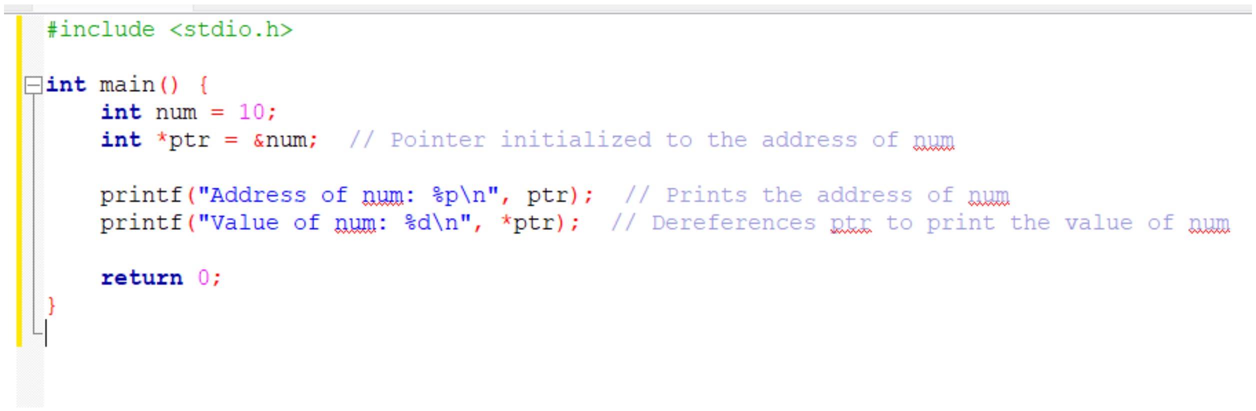
```
pointer_name = &variable;
```

Where &variable gives the memory address of the variable.

Example:

```
int num = 10;
int *ptr = &num;  // Initializes ptr to point to num's address
```

Example:



```c
#include <stdio.h>

int main() {
    int num = 10;
    int *ptr = &num;  // Pointer initialized to the address of num

    printf("Address of num: %p\n", ptr);  // Prints the address of num
    printf("Value of num: %d\n", *ptr);  // Dereferences ptr to print the value of num

    return 0;
}
```

3. Size of a Pointer Variable in C:

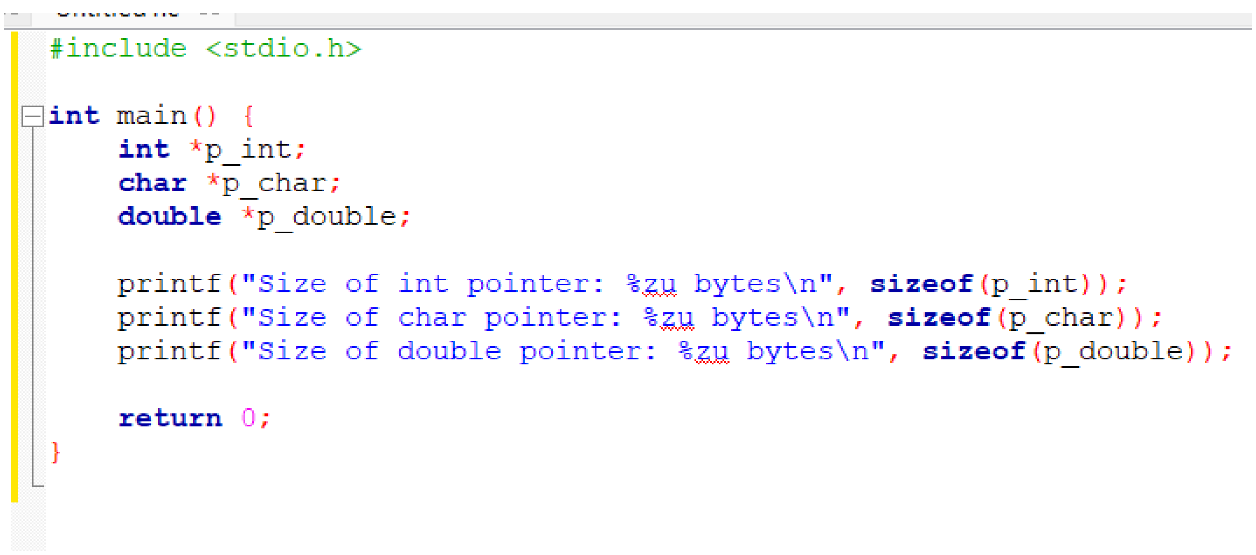
The **size of a pointer variable** depends on the **architecture of the system**, not on the type of data it points to.

-Key Concept:

- On a **32-bit system**, pointers are usually **4 bytes**.
- On a **64-bit system**, pointers are typically **8 bytes**.

This is because a pointer must be large enough to hold any memory address, and address size is determined by the system architecture.

Example:

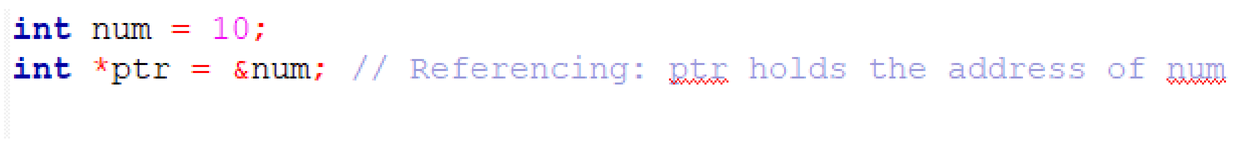


```c
#include <stdio.h>

int main() {
    int *p_int;
    char *p_char;
    double *p_double;

    printf("Size of int pointer: %zu bytes\n", sizeof(p_int));
    printf("Size of char pointer: %zu bytes\n", sizeof(p_char));
    printf("Size of double pointer: %zu bytes\n", sizeof(p_double));

    return 0;
}
```

4. **Referencing (Getting the Address)**

Referencing means obtaining the memory address of a variable using the & (address-of) operator.

Example:



```c
int num = 10;
int *ptr = &num; // Referencing: ptr holds the address of num
```

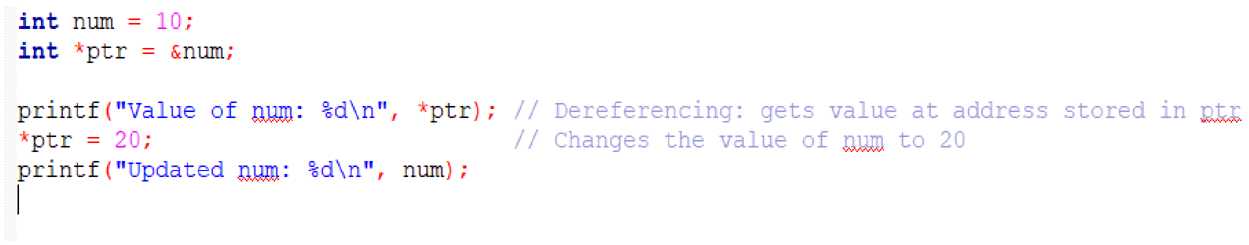
- &num gives the address of num.

- ptr now "points to" num.

5. **Dereferencing** (Accessing the Value)

Dereferencing means accessing or modifying the value stored at the memory address a pointer is holding. This is done using the * (dereference) operator.

Example:

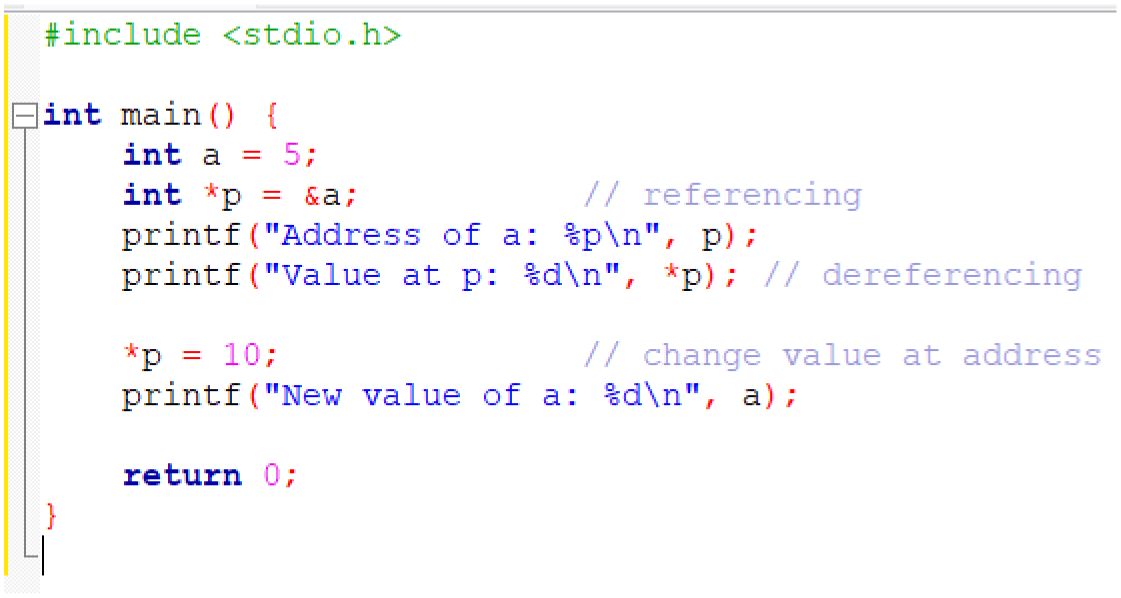


```c
int num = 10;
int *ptr = &num;

printf("Value of num: %d\n", *ptr); // Dereferencing: gets value at address stored in ptr
*ptr = 20;                          // Changes the value of num to 20
printf("Updated num: %d\n", num);
```

- *ptr accesses the value stored at the address ptr is pointing to.

- So *ptr = 20; is equivalent to num = 20;.

Example:



```c
#include <stdio.h>

int main() {
    int a = 5;
    int *p = &a;           // referencing
    printf("Address of a: %p\n", p);
    printf("Value at p: %d\n", *p); // dereferencing

    *p = 10;               // change value at address
    printf("New value of a: %d\n", a);

    return 0;
}
```

6. What is Pointer Arithmetic?

In C, you can perform arithmetic operations on pointers like:

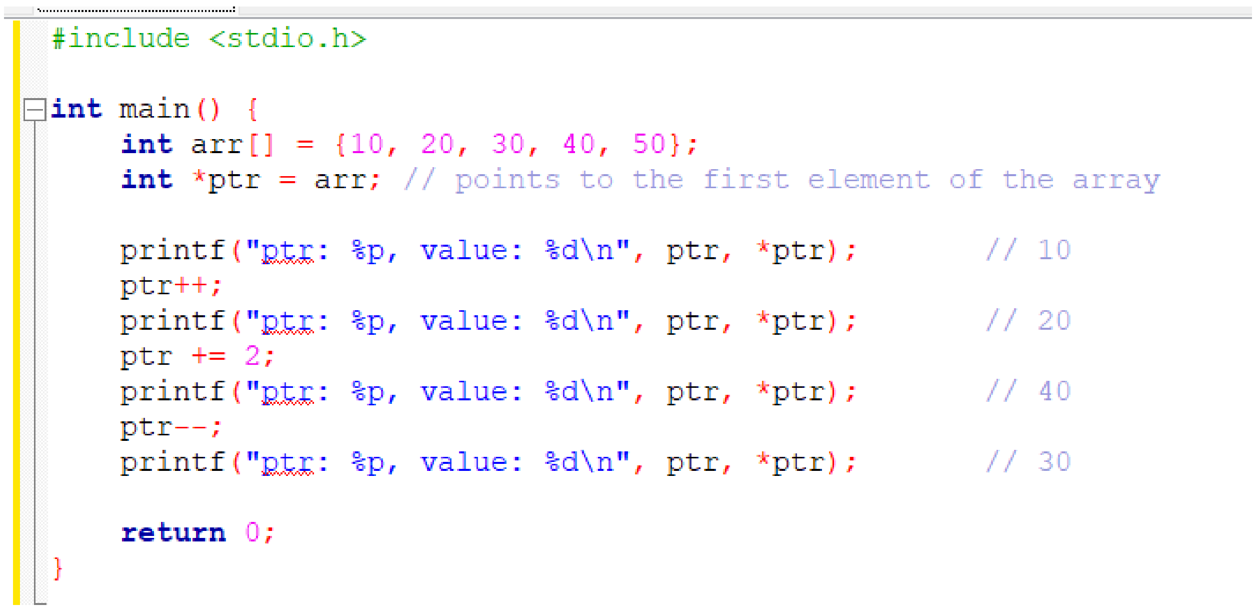
- ++ (increment)
- -- (decrement)
- + (add an integer)
- - (subtract an integer)

These operations move the pointer **by the size of the data type** it points to.

- If ptr is a pointer to a data type T, then:

- ptr + 1 moves the pointer forward by sizeof(T) bytes.
- ptr - 1 moves the pointer backward by sizeof(T) bytes.

Example with int Array:



```
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr; // points to the first element of the array

    printf("ptr: %p, value: %d\n", ptr, *ptr);        // 10
    ptr++;
    printf("ptr: %p, value: %d\n", ptr, *ptr);        // 20
    ptr += 2;
    printf("ptr: %p, value: %d\n", ptr, *ptr);        // 40
    ptr--;
    printf("ptr: %p, value: %d\n", ptr, *ptr);        // 30

    return 0;
}
```

Valid Pointer Operations:

Operation	Description
ptr + n	Move forward n elements
ptr - n	Move backward n elements
ptr1 - ptr2	Returns number of elements between two pointers
++ptr / --ptr	Move to next/previous element

7. Access & Modify Values Using a Pointer

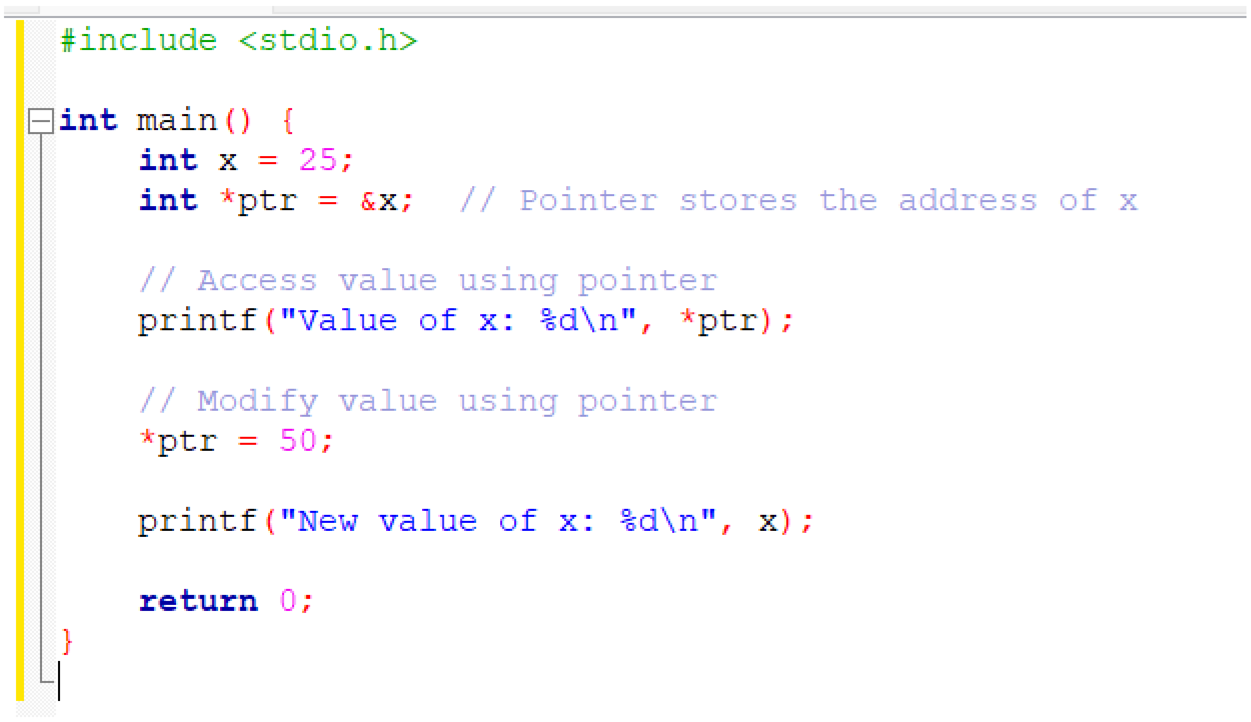
1. Accessing a Value

You use the **dereference operator *** to access the value stored at the memory location the pointer points to.

2. Modifying a Value

Use the same * operator on the left-hand side of an assignment to **change** the value at that memory location.

Example:



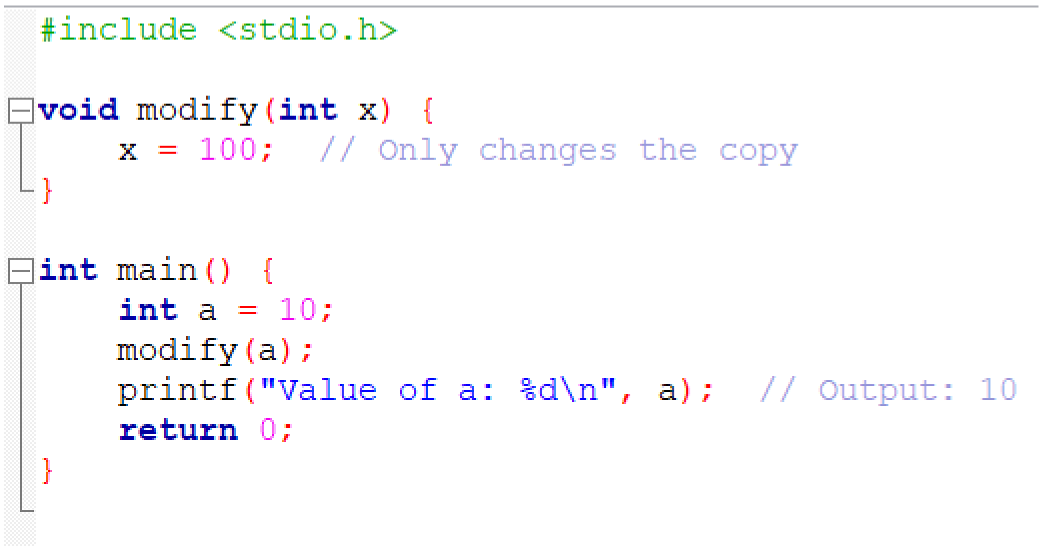
```c
#include <stdio.h>

int main() {
    int x = 25;
    int *ptr = &x;  // Pointer stores the address of x

    // Access value using pointer
    printf("Value of x: %d\n", *ptr);

    // Modify value using pointer
    *ptr = 50;

    printf("New value of x: %d\n", x);

    return 0;
}
```

8. Pass by Value vs Pass by Reference

1. Pass by Value (Default in C)

When you **pass a variable to a function by value**, a **copy** of the variable is made.
Changes made inside the function **do NOT affect** the original variable.

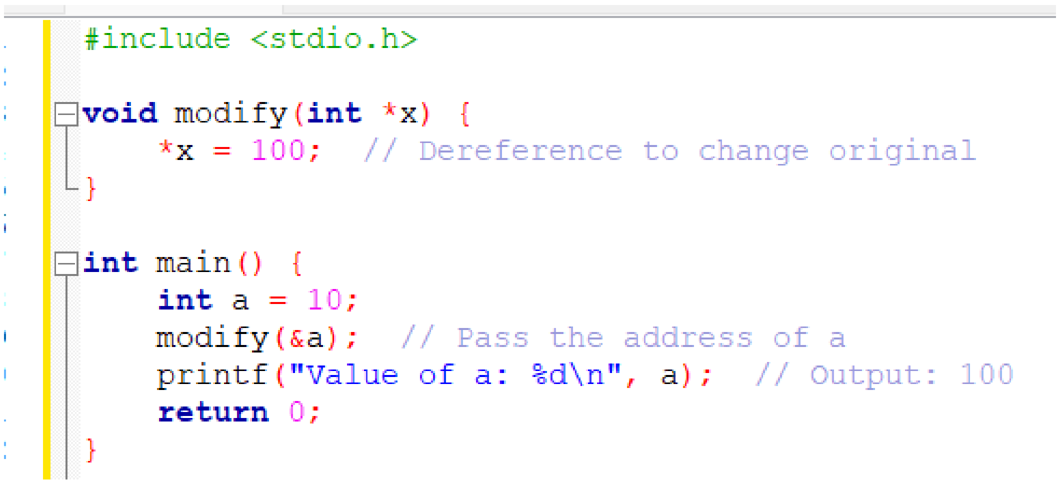
Example:



```c
#include <stdio.h>

void modify(int x) {
    x = 100;  // Only changes the copy
}

int main() {
    int a = 10;
    modify(a);
    printf("Value of a: %d\n", a);  // Output: 10
    return 0;
}
```

2. Pass by Reference (Using Pointers)

When you **pass a variable's address**, you allow the function to directly modify the original variable — this is called **pass by reference**.



```c
#include <stdio.h>

void modify(int *x) {
    *x = 100;  // Dereference to change original
}

int main() {
    int a = 10;
    modify(&a);  // Pass the address of a
    printf("Value of a: %d\n", a);  // Output: 100
    return 0;
}
```

9. Types of Pointers in C with Examples:

a. Null Pointer

```
int *ptr = NULL; // points to nothing
```

b. Void Pointer

```
void *ptr;

int x = 10;

ptr = &x; // must be type-cast before dereferencing
```

c. Wild Pointer

```
int *ptr; // uninitialized: wild pointer
```

d. Dangling Pointer

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

free(ptr); // now ptr is dangling
```

e. Pointer to Pointer

```
int x = 5;

int *ptr = &x;

int **pptr = &ptr;
```