CSE 06131223 ♦ CSE 06131224

Structured Programming

Lecture 20

Pointers in C (1)

Prepared by

Md. Mijanur Rahman, Prof. Dr.
Dept. of Computer Science and Engineering
Jatiya Kabi Kazi Nazrul Islam University, Bangladesh
www.mijanrahman.com

Contents

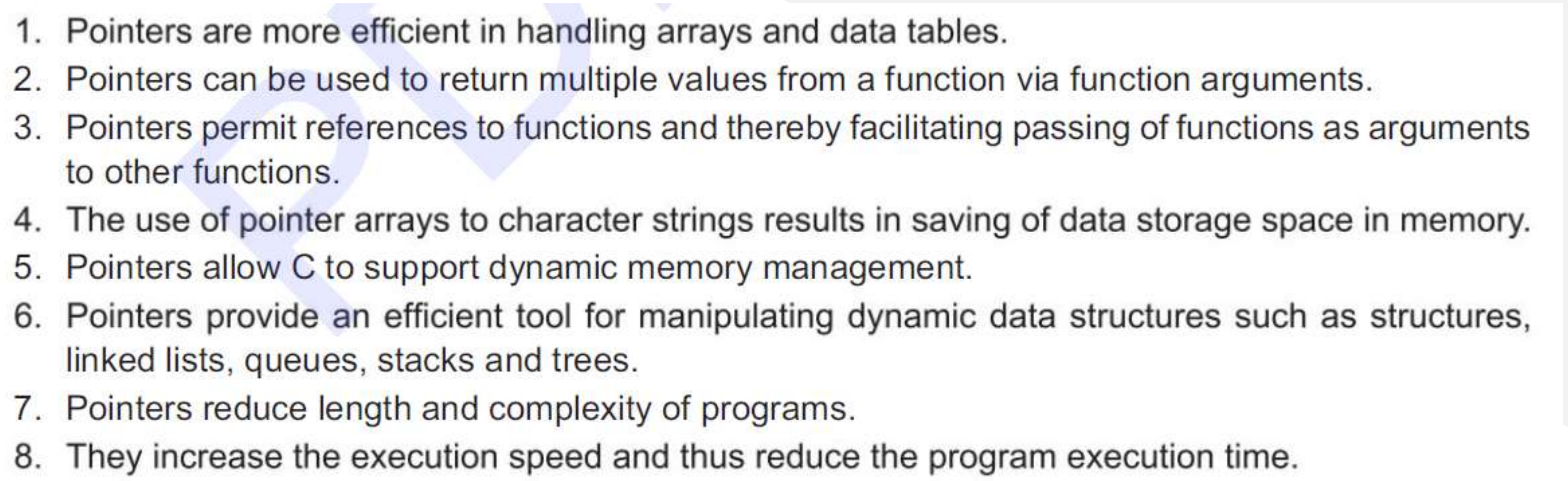
Pointers in C

Pointers in C

- A pointer is a derived data type in C. It is built from one of the fundamental data types available in C. Pointers contain memory addresses as their values.
- In C, a pointer is a variable that stores the memory address of another variable. Instead of storing the actual value of a variable, a pointer holds the location (address) in memory where the variable is stored.
- This allows for dynamic memory allocation, efficient memory usage, and manipulation of data indirectly.

Pointers in C

- Pointers offer a number of benefits to the programmers. They include:



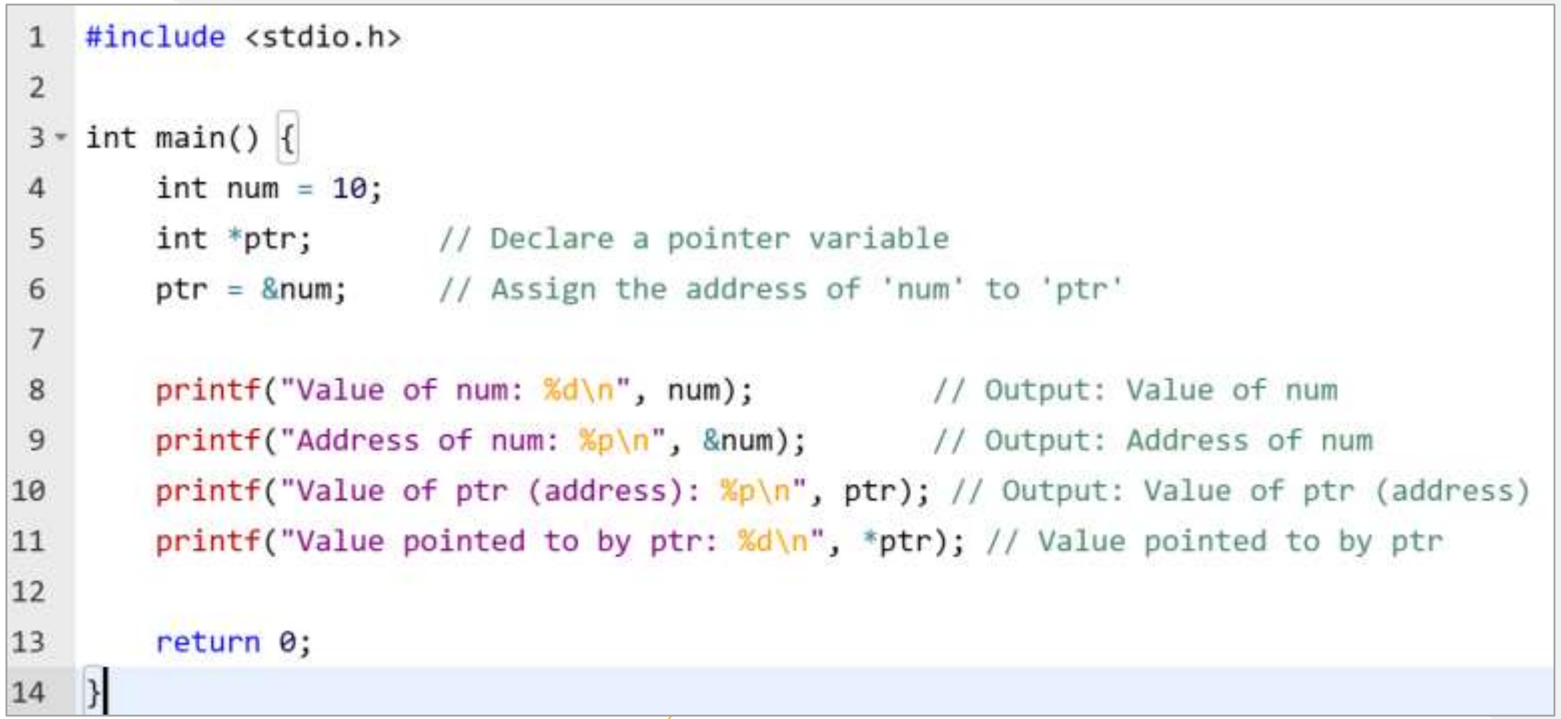
1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function arguments.
3. Pointers permit references to functions and thereby facilitating passing of functions as arguments to other functions.
4. The use of pointer arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduce the program execution time.

Pointers in C

- Pointers are extensively used in C for various purposes, such as:
  1. **Dynamic Memory Allocation**: Pointers are used to dynamically allocate memory during runtime using functions like **malloc()**, **calloc()**, and **realloc()**. This enables programs to efficiently manage memory as needed, especially for data structures like arrays, linked lists, and trees.
  2. **Passing Parameters by Reference**: Unlike passing parameters by value, which creates a copy of the data, passing parameters by reference using pointers allows functions to modify the original data. This is particularly useful when dealing with large data structures or when multiple values need to be returned from a function.
  3. **Manipulating Arrays**: Pointers can be used to iterate through arrays efficiently. They can also be used to dynamically allocate multi-dimensional arrays and to access elements in memory directly, bypassing array indexing.
  4. **String Manipulation**: C-style strings are represented as arrays of characters terminated by a null character (**'\0'**). Pointers are commonly used to manipulate strings efficiently, including tasks such as copying, concatenating, and comparing strings.
  5. **Implementing Data Structures**: Pointers are essential for implementing various data structures such as linked lists, stacks, queues, trees, and graphs. They allow for the creation of dynamic data structures that can grow or shrink as needed.
  6. **Function Pointers**: Pointers can be used to store addresses of functions, allowing for the creation of callback mechanisms and implementing advanced programming techniques like function pointers arrays, which are commonly used in event-driven programming.
  7. **Dynamic Data Structures**: Pointers enable the creation of complex data structures with dynamic memory requirements, such as linked lists, trees, and graphs. These data structures can adjust their size during runtime, leading to efficient memory usage and improved performance.
  8. **Pointer Arithmetic**: Pointers can be manipulated using arithmetic operations like addition and subtraction, allowing for efficient traversal of data structures and array manipulation.

Pointers in C

- Here's a simple example to illustrate pointers:

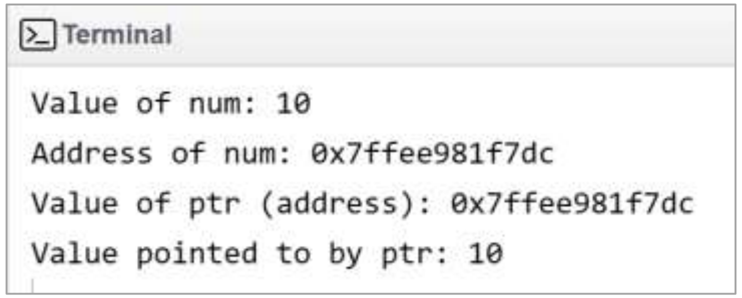


```c
#include <stdio.h>

int main() {
    int num = 10;
    int *ptr;       // Declare a pointer variable
    ptr = &num;     // Assign the address of 'num' to 'ptr'

    printf("Value of num: %d\n", num);          // Output: Value of num
    printf("Address of num: %p\n", &num);       // Output: Address of num
    printf("Value of ptr (address): %p\n", ptr); // Output: Value of ptr (address)
    printf("Value pointed to by ptr: %d\n", *ptr); // Value pointed to by ptr

    return 0;
}
```



Terminal

```
Value of num: 10
Address of num: 0x7ffee981f7dc
Value of ptr (address): 0x7ffee981f7dc
Value pointed to by ptr: 10
```

Memory Organization

- The computer's memory is a sequential collection of storage cells as shown in the Figure. Each cell, commonly known as a byte, has a number called address associated with it.
- Typically, the addresses are numbered consecutively, starting from zero. The last address depends on the memory size.
- A computer system having 64KB memory will have its last address as 65,535.

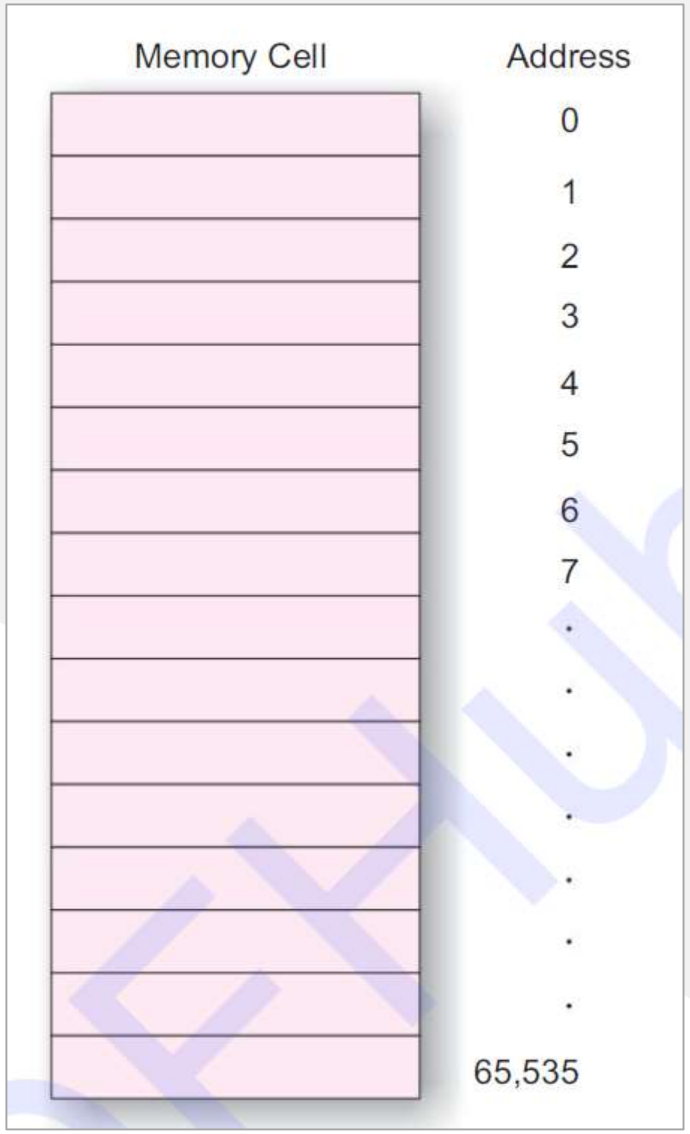


Fig: Memory Organization

Memory Organization

- **Representation of a Variable in Memory:**
- Whether we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable.
- Consider the following statement:

  Int quantity = 179;
- This statement instructs the system to find a location for the integer variable **quantity** and puts the value 179 in that location, as shown in the Figure.
- During execution of the program, the system always associates the name **quantity** with the address 5000.

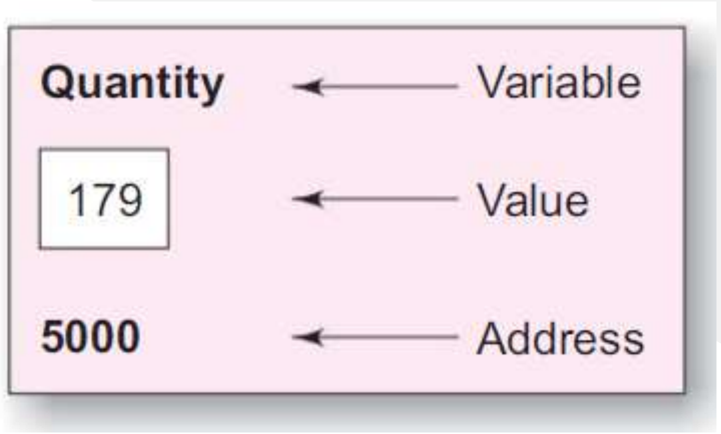


Fig: Representation of a variable.

Memory Organization

- Since a pointer is a variable, its value is also stored in the memory in another location.

- Suppose, we assign the address of **quantity** to a variable **p.** The link between the variables **p** and **quantity** can be visualized as shown in the Figure.
- The address of **p** 5048.

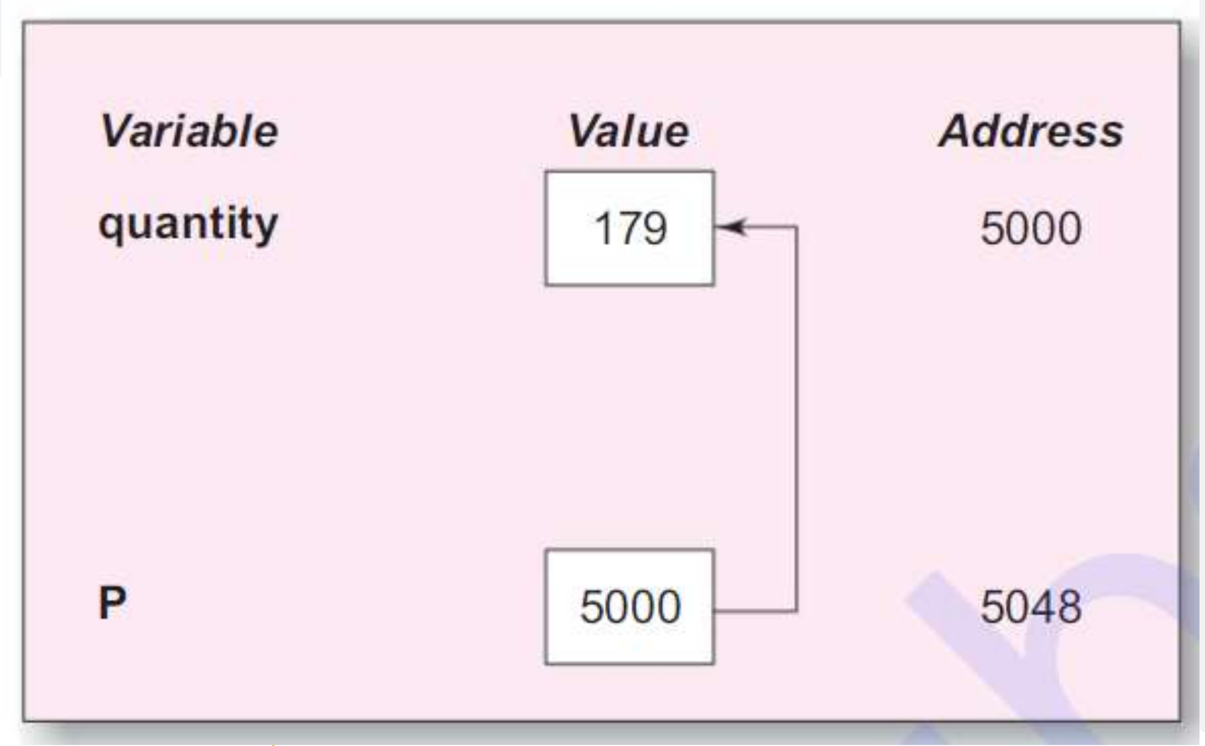
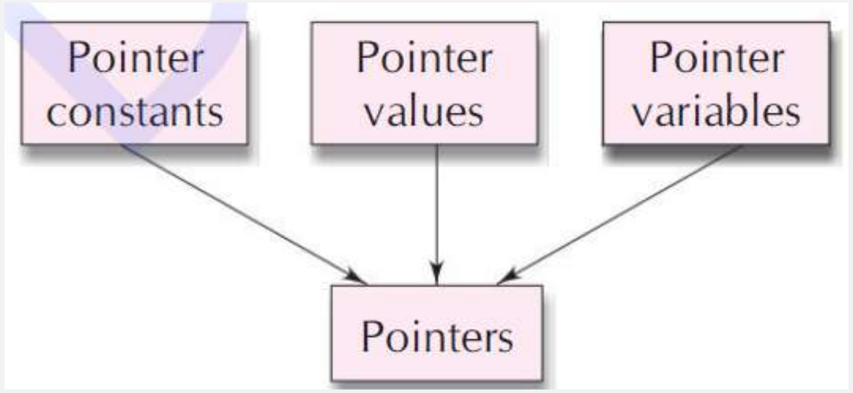


Fig: Representation of a pointer variable.

Memory Organization



- Pointers are built on the three underlying concepts, such as:
  - Pointer Constants
  - Pointer Values
  - Pointer Variables
- Memory addresses within a computer are referred to as **pointer constants.** We cannot change them; we can only use them to store data values. They are like **house numbers.**
- We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the **address operator (&).** The value thus obtained is known as **pointer value**; it may change from one run of the program to another.
- Once we have a **pointer value,** it can be stored into another variable. The variable that contains a pointer value is called **pointer variable.**

Accessing the Address of a Variable

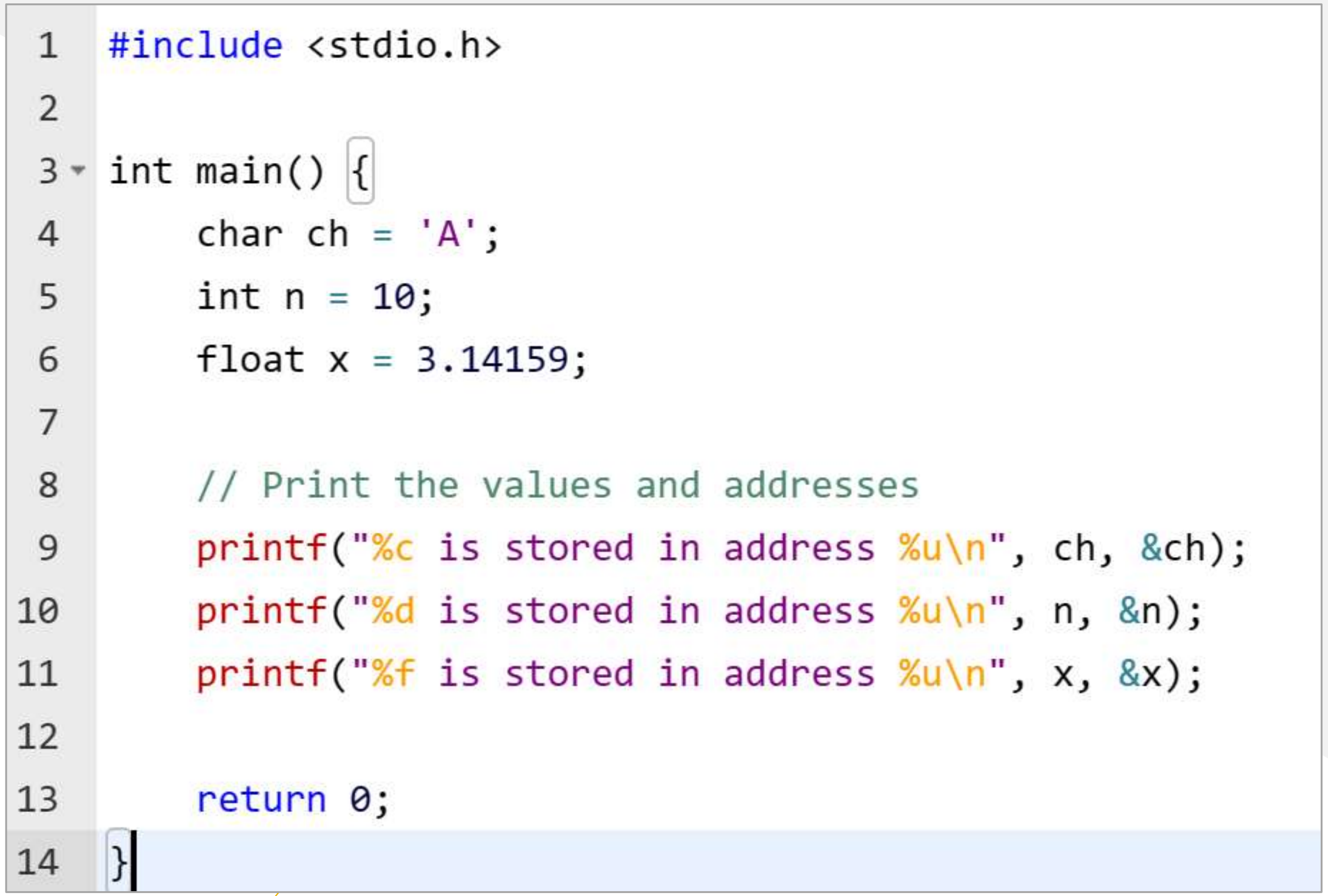
- The actual location of a variable in the memory is system dependent, and, therefore, the address of a variable is not known to us immediately.
- The address can be determined with the help of the operator **&** available in C. The operator & immediately preceding a variable returns the address of the variable associated with it. For example, the statement:

```
p = &quantity;
```

- Would assign the address 5000 (the location of **quantity**) to the variable p. The & operator can be remembered as ‘address of’.

Accessing the Address of a Variable

- Here's a simple C program that prints the address of a variable along with its value:

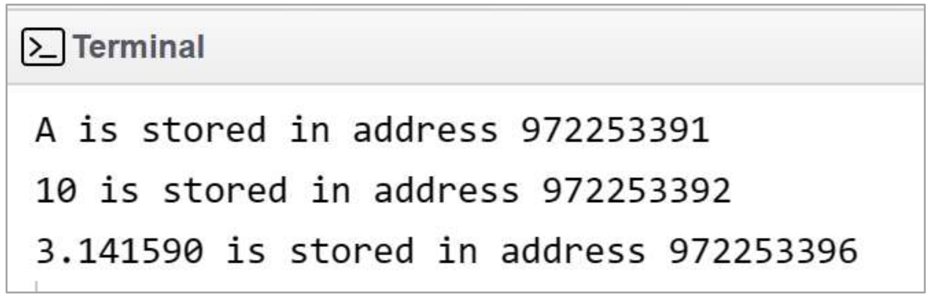


```c
#include <stdio.h>

int main() {
    char ch = 'A';
    int n = 10;
    float x = 3.14159;

    // Print the values and addresses
    printf("%c is stored in address %u\n", ch, &ch);
    printf("%d is stored in address %u\n", n, &n);
    printf("%f is stored in address %u\n", x, &x);

    return 0;
}
```



```
A is stored in address 972253391
10 is stored in address 972253392
3.141590 is stored in address 972253396
```

Declaring Pointer Variable

- The declaration of a pointer variable takes the following form:

  ```
  data-type *ptr-name;
  ```

- This tells the compiler three things about the variable **ptr-name**.
  1. The asterisk (*) tells that the variable **ptr-name** is a pointer variable.
  2. **ptr-name** needs a memory location.
  3. **ptr-name** points to a varable of type **data-type**.

- For example,

  ```
  int *p;
  float *x;
  ```

Declaring Pointer Variable

- **Pointer declaration style:**
- The symbol * can appear anywhere between the type-name and the pointer variable-name. We can use the following styles:

```
int*    p;
int     *p;
int     * p;
```

- The second style is popular in C programs, such as:

```
int *p, x, *q;
```

Initialization of Pointer Variables

- The process of assigning the address of a variable to a pointer variable is known as initialization. Once a pointer variable has been declared, we can use the assignment operator to initialize the variable. For example:

```
        int number;
        int *p;
        p = &number;
```

- We can also combine the initialization with the declaration. That is,

```
        int *p = &number;
```

  is allowed.

Initialization of Pointer Variables

- In C, pointer variables can be initialized in several ways:
- **Initializing with Address of Another Variable:** We can initialize a pointer variable with the address of another variable using the address-of operator (&). For example:

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

- **Initializing with NULL:** We can initialize a pointer variable to NULL if we don't want it to point to any valid memory address initially. This is often done to indicate that the pointer is not currently pointing to anything. For example:

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

Initialization of Pointer Variables

- **Initializing during Declaration:** Pointer variables can be initialized during declaration. For example:

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

- **Dynamic Memory Allocation:** Pointers can be initialized with dynamically allocated memory using functions like malloc(), calloc(), or realloc(). For example:

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

Initialization of Pointer Variables

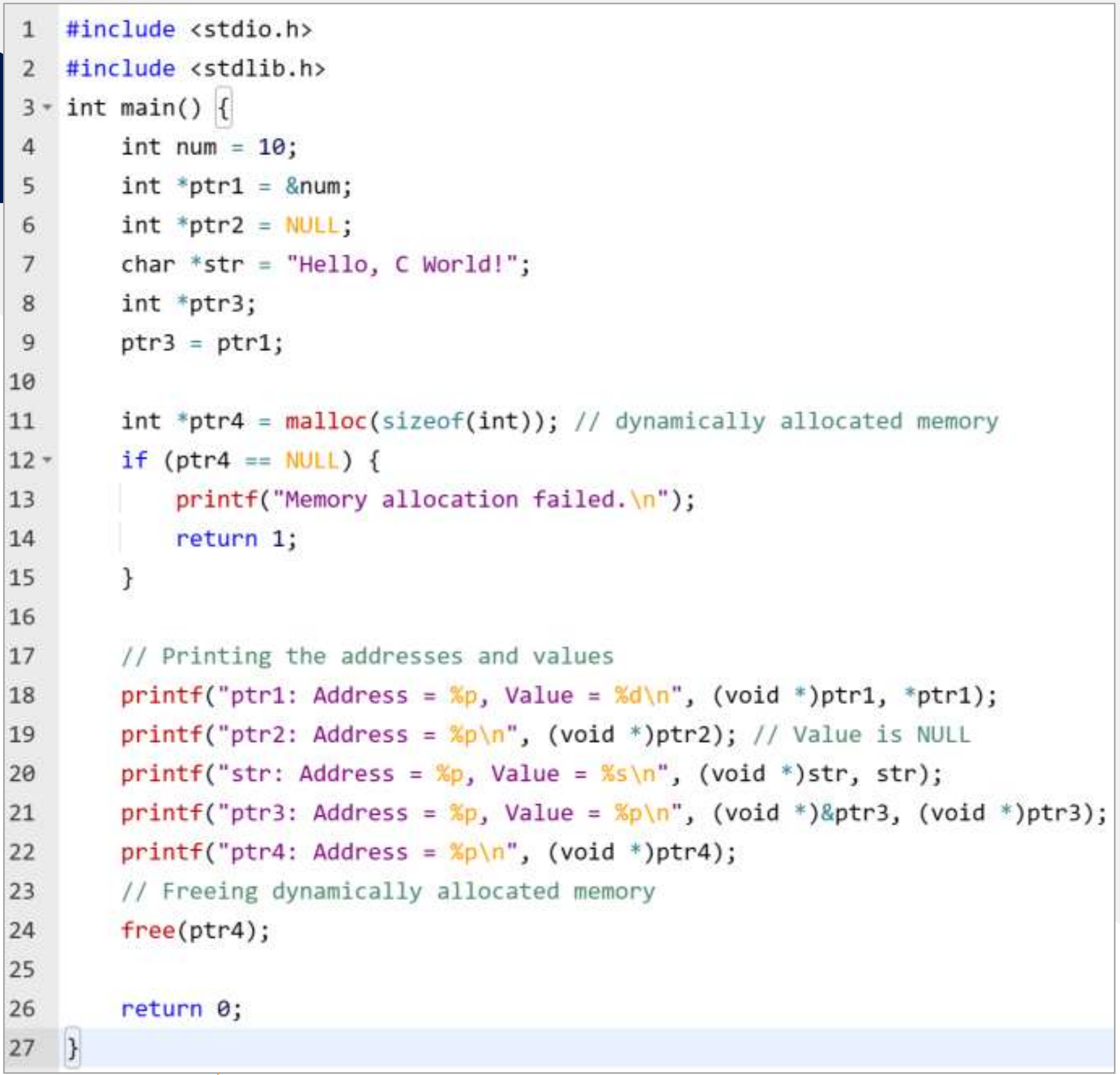
- **String Literal Initialization:** Pointers can be initialized with string literals, which are arrays of characters. For example:

```
char *str = "Hello, C World!";
```

- **Initializing with Another Pointer**: Pointers can be initialized with the value of another pointer of the same type. For example:

```
int *ptr1;
int *ptr2 = ptr1;
```

- Here's a C program demonstrating various ways to initialize pointer variables:

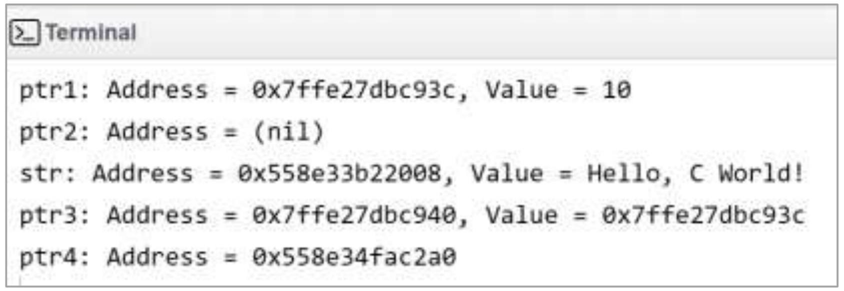


```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int num = 10;
    int *ptr1 = &num;
    int *ptr2 = NULL;
    char *str = "Hello, C World!";
    int *ptr3;
    ptr3 = ptr1;

    int *ptr4 = malloc(sizeof(int)); // dynamically allocated memory
    if (ptr4 == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Printing the addresses and values
    printf("ptr1: Address = %p, Value = %d\n", (void *)ptr1, *ptr1);
    printf("ptr2: Address = %p\n", (void *)ptr2); // Value is NULL
    printf("str: Address = %p, Value = %s\n", (void *)str, str);
    printf("ptr3: Address = %p, Value = %p\n", (void *)&ptr3, (void *)ptr3);
    printf("ptr4: Address = %p\n", (void *)ptr4);
    // Freeing dynamically allocated memory
    free(ptr4);

    return 0;
}
```



Terminal

```
ptr1: Address = 0x7ffe27dbc93c, Value = 10
ptr2: Address = (nil)
str: Address = 0x558e33b22008, Value = Hello, C World!
ptr3: Address = 0x7ffe27dbc940, Value = 0x7ffe27dbc93c
ptr4: Address = 0x558e34fac2a0
```

? THE END