Programming in C

Chapter 5: Pointers

These notes cover the concept, declaration, arithmetic, and application of pointers in the C programming language, along with their use with functions, arrays, strings, and dynamic memory. Prepared for engineering students, the notes conclude with practice MCQs and broad (descriptive) questions for exam preparation.

5.1 Understanding Pointers, Declaring and Accessing Pointers, Null Pointers, Generic Pointers, Pointer Arithmetic and Expressions

Understanding Pointers

A pointer is a special type of variable that stores the memory address of another variable, rather than storing a data value directly. Every variable created in a C program occupies some location in the computer's memory, and each such location has a unique address. A pointer variable "points to" another variable by holding this address.

Pointers are one of the most powerful and distinctive features of the C language. They allow a program to access and manipulate memory directly, which makes it possible to:

- Pass large data structures to functions efficiently, without copying them.
- Allow a function to modify the actual variables of the calling function (call by reference).
- Work efficiently with arrays and strings, since array names themselves behave like pointers.
- Allocate and manage memory dynamically at run time.
- Build complex data structures such as linked lists, stacks, queues, trees, and graphs.

Declaring and Accessing Pointers

A pointer variable is declared by writing the data type of the value it will point to, followed by an asterisk (*), and then the variable name. The general syntax is:

```
datatype *pointer_name;
```

For example:

```
int *p;      // pointer to an integer
float *fp;   // pointer to a float
char *cp;    // pointer to a character
```

Two operators are essential when working with pointers:

- The address-of operator (&): returns the memory address of a variable.
- The dereference or indirection operator (*): accesses (reads or writes) the value stored at the address a pointer holds.

Consider the following example:

```
#include <stdio.h>
int main() {
    int x = 10;
    int *p;
    p = &x;             // p now stores the address of x
    printf("%d", *p);   // *p dereferences p, prints 10
    return 0;
}
```

Here, p is said to "point to" x. Writing *p = 20; would change the value of x itself to 20, since *p and x refer to exactly the same memory location.

Null Pointers

A null pointer is a pointer that is not currently pointing to any valid memory location. It is conventionally initialized with the macro NULL (commonly defined via header files such as stdio.h) or with the literal value 0.

```
int *p = NULL;
```

- A null pointer indicates that the pointer is not (yet) associated with any object, avoiding the danger of holding an unpredictable, garbage address.
- Good programming practice requires checking a pointer against NULL before dereferencing it, for example: if (p != NULL) { ... }
- Dereferencing a null pointer (attempting *p when p is NULL) is undefined behaviour and typically causes a run-time crash known as a segmentation fault.
- Functions such as malloc() return NULL when memory allocation fails, so checking for NULL is essential after every dynamic allocation.

Generic Pointers

A generic pointer, declared using the keyword void, can hold the address of a variable of any data type. It is called "generic" (or a void pointer) because it is not tied to any particular data type.

```
void *gp;
int a = 5;
gp = &a;   // valid: void pointer can store address of any type
```

Because the compiler does not know the size of the object a void pointer refers to, a void pointer cannot be dereferenced directly, and pointer arithmetic cannot be performed on it directly. It must first be explicitly typecast to an appropriate pointer type before use, for example: printf("%d", *(int*)gp);. Void pointers are widely used to write generic, type-independent functions, such as the standard library functions malloc() and qsort().

Pointer Arithmetic and Expressions

Pointers support a restricted set of arithmetic operations, and these operations are scaled automatically according to the size of the data type the pointer refers to, rather than moving one byte at a time.

Valid pointer operations include:

- Adding an integer to a pointer (p + 1), which advances the pointer by one element (sizeof(type) bytes).
- Subtracting an integer from a pointer (p - 1).
- Incrementing or decrementing a pointer (p++, p--).
- Subtracting two pointers of the same type (p2 - p1), which gives the number of elements between them.
- Comparing two pointers of the same type using ==, !=, <, >, <=, >= (typically used while traversing arrays).

Operations that are NOT valid include adding two pointers together, and multiplying or dividing a pointer by a value. Example of pointer arithmetic used to traverse an array:

```
int arr[5] = {10, 20, 30, 40, 50};
int *p = arr;              // p points to arr[0]
for (int i = 0; i < 5; i++) {
    printf("%d ", *(p + i));   // equivalent to arr[i]
}
```

If p is an int* and currently holds address 1000 (with int occupying 4 bytes on the given system), then p + 1 will hold address 1004, not 1001 — the compiler automatically multiplies the offset by sizeof(int).

5.2 Passing Arguments to Functions Using Pointers, Pointers and Arrays, Passing an Array to a Function, Array Name and Pointer

Passing Arguments to Functions Using Pointers

By default, C passes arguments to functions by value — the function receives a copy of the argument, so changes made inside the function do not affect the original variable in the calling function. Pointers make it possible to simulate call by reference: instead of passing the value itself, the address of the variable is passed, allowing the function to modify the original variable directly through dereferencing.

A classic example is a function to swap two numbers:

```
#include <stdio.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main() {
    int x = 5, y = 10;
    swap(&x, &y);              // addresses of x and y are passed
    printf("%d %d", x, y);     // prints 10 5
    return 0;
}
```

Because the addresses of x and y are passed, the function operates on the original variables themselves, and the swap is reflected back in main().

Pointers and Arrays

Arrays and pointers are closely related in C. The name of an array, when used in most expressions, decays into a pointer to its first element. This means that if a is declared as int a[10];, then a is equivalent to &a[0], and array elements can be accessed either using array indexing or pointer notation:

```
a[i]  is equivalent to  *(a + i)
```

This equivalence works because array elements are stored in contiguous memory locations, so adding i to the base address (with automatic scaling by the element size) reaches the i-th element directly.

Passing an Array to a Function

When an array is passed as an argument to a function, C does not copy the entire array. Instead, only the address of the first element is passed, so the function actually receives a pointer. Because of this, information about the array's total size is lost inside the function, and the size must be passed explicitly as a separate parameter.

```
int sum(int *arr, int n) {     // could also be written as int arr[]
    int total = 0, i;
    for (i = 0; i < n; i++)
        total += arr[i];       // arr[i] works even though arr is a pointer
    return total;
}
int main() {
    int nums[5] = {1, 2, 3, 4, 5};
    printf("%d", sum(nums, 5));   // passing array name and its size
    return 0;
}
```

Because the function receives only an address, any changes made to array elements inside the function are reflected in the original array in the caller — arrays are always effectively passed "by reference" in C.

Array Name and Pointer

Although an array name behaves like a pointer to its first element in most expressions, an array name and a pointer variable are not exactly the same thing. Key differences include:

- An array name is a constant address — it cannot be reassigned to point elsewhere (a = &b; is illegal for an array a), whereas a pointer variable can be made to point to different locations at different times.
- sizeof(array_name) returns the total number of bytes occupied by the entire array, whereas sizeof(pointer_name) always returns the size of the pointer variable itself (typically 4 or 8 bytes), regardless of what it points to.
- Memory for an array is allocated at compile time (for a fixed-size array) as a single contiguous block, whereas a pointer is simply a variable that can be made to reference memory allocated elsewhere, including memory allocated dynamically at run time.
- An array name always refers to the same block of memory throughout its scope, while a pointer variable's value (the address it stores) can change during program execution.

5.3 Pointers and Strings, Array of Pointers, Function Pointers, Pointers to Pointers

Pointers and Strings

In C, a string is stored as an array of characters terminated by a special null character ('\0'). Strings can be represented in two closely related ways: as a character array, or as a character pointer pointing to a string literal.

```
char str1[] = "Hello";     // modifiable array containing 'H','e','l','l','o','\0'
char *str2 = "Hello";      // pointer to a (typically read-only) string literal
```

Because a string is simply an array of characters, pointer notation can be used to traverse it. For example, a simplified version of the standard strlen() function can be written using a pointer:

```
int my_strlen(char *s) {
    int count = 0;
    while (*s != '\0') {
        count++;
        s++;             // move pointer to next character
    }
    return count;
}
```

Array of Pointers

An array of pointers is an array whose elements are themselves pointers. This structure is especially useful for handling a collection of strings of varying lengths, since each element can point to a string of a different size, avoiding the wasted space of a fixed-width two-dimensional character array.

```
char *names[3] = {"Amit", "Rina", "Suman"};
for (int i = 0; i < 3; i++)
    printf("%s\n", names[i]);
```

Here, names is an array of three character pointers, and each pointer holds the address of the first character of a separate string literal.

Function Pointers

A function pointer is a pointer that stores the address of a function rather than the address of a data variable. Since compiled functions also reside at specific memory addresses, their addresses can be stored, passed around, and used to call the function indirectly.

```
int add(int a, int b) { return a + b; }
int main() {
    int (*fp)(int, int);   // fp is a pointer to a function taking two ints, returning
int
    fp = add;
    printf("%d", fp(3, 4));   // calls add(3, 4) indirectly through fp
    return 0;
}
```

Function pointers are commonly used to implement callback functions, event handlers, and dispatch tables (arrays of function pointers used to select behaviour at run time), and they form the basis of dynamic and flexible program design in C.

Pointers to Pointers

A pointer to a pointer (also called double indirection) is a variable that stores the address of another pointer variable, rather than the address of ordinary data. It is declared using two asterisks.

```
int x = 10;
int *p = &x;      // p points to x
int **pp = &p;     // pp points to p (pointer to a pointer)
printf("%d", **pp);  // dereferencing twice gives the value of x
```

Pointers to pointers are used when a function needs to modify the value of a pointer itself (for example, to allocate memory and have the change reflected back in the caller), and they are essential when working with dynamically allocated two-dimensional arrays and arrays of strings.

5.4 Memory Usage, Dynamic Memory Allocation, Drawbacks of Pointers

Memory Usage

The memory used by a running C program is typically organised into distinct segments:

- Text (code) segment — stores the compiled machine instructions of the program.
- Data segment — stores global and static variables, which exist for the entire lifetime of the program.
- Stack — stores local variables and function-call information (such as return addresses and parameters). Memory here is allocated and released automatically as functions are called and return, and it grows and shrinks in a strict last-in-first-out order.
- Heap — a pool of memory available for dynamic allocation at run time. Unlike the stack, memory in the heap must be requested and released manually by the programmer using functions such as malloc() and free().

Dynamic Memory Allocation

Arrays declared normally (for example, int a[10];) have a fixed size decided at compile time, which can be wasteful or insufficient if the required amount of memory is not known in advance. Dynamic memory allocation allows a program to request memory from the heap at run time, using functions declared in stdlib.h:

- malloc(size) — allocates a single block of the requested number of bytes and returns a void pointer to it; the allocated memory is uninitialised (contains garbage values).
- calloc(n, size) — allocates memory for n elements of the given size and initialises all bytes to zero.
- realloc(ptr, new_size) — resizes a previously allocated block of memory to a new size, preserving existing contents as far as possible.
- free(ptr) — releases previously allocated memory back to the system so that it can be reused; every successful dynamic allocation should eventually be matched with a corresponding free().

Example using malloc() and free():

```
#include <stdlib.h>
int *arr;
int n = 5;
arr = (int *) malloc(n * sizeof(int));   // request memory for 5 integers
if (arr == NULL) {
    printf("Memory allocation failed");
} else {
    for (int i = 0; i < n; i++)
        arr[i] = i * i;
    free(arr);          // release the memory once it is no longer needed
}
```

Drawbacks of Pointers

Despite their power and flexibility, pointers must be used carefully, since misuse is a common source of serious bugs. Notable drawbacks include:

- Uninitialized pointers hold garbage addresses; dereferencing them leads to unpredictable and undefined behaviour.
- Memory leaks occur when dynamically allocated memory is not released with free(), gradually exhausting available memory.
- Dangling pointers arise when a pointer continues to be used after the memory it referred to has already been freed, causing unpredictable results.
- Incorrect pointer arithmetic can cause the program to read or write outside the bounds of an array, leading to buffer overflows or segmentation faults.
- Pointer-heavy code is generally harder to read, understand, and debug compared to code using only ordinary variables.
- Improper pointer use is a common source of security vulnerabilities, such as buffer-overflow exploits.
- Pointer size and behaviour can vary across different compilers and hardware platforms, which can affect the portability of pointer-based code.

Multiple Choice Questions (MCQs)

Answer the following 15 multiple choice questions. The answer key is provided at the end of this section.

1. What does a pointer variable store?

(a) The value of a variable (b) The memory address of a variable (c) The name of a variable (d) The data type of a variable

2. Which operator is used to access (dereference) the value pointed to by a pointer?

(a) & (b) * (c) % (d) ->

3. Which of the following header files commonly defines the NULL macro?

(a) stdio.h (b) conio.h (c) time.h (d) ctype.h

4. A void pointer is also commonly known as a:

(a) Null pointer (b) Generic pointer (c) Wild pointer (d) Dangling pointer

5. Which of the following pointer operations is NOT valid in C?

(a) p + 1 (b) p - 1 (c) p1 + p2 (adding two pointers) (d) p1 - p2 (same type)

6. If int *p; holds address 1000 and int occupies 4 bytes, what address does p + 1 hold?

(a) 1001 (b) 1002 (c) 1004 (d) 1008

7. Which expression correctly relates array indexing to pointer notation?

(a) a[i] is equivalent to *(a + i) (b) a[i] is equivalent to *a + i (c) a[i] is equivalent to &a + i (d) None of these

8. When an array is passed to a function in C, what is actually passed?

(a) A complete copy of the array (b) The address of the first element (c) The total size of the array (d) Nothing is passed

9. For int a[10];, which expression gives the total number of bytes occupied by the array?

(a) sizeof(a) (b) sizeof(a[0]) (c) sizeof(*a) (d) sizeof(a) / sizeof(int *)

10. A string in C is terminated by which character?

(a) '\0' (b) '\n' (c) EOF (d) NULL

11. What does the declaration char *names[5]; represent?

(a) A pointer to five characters (b) An array of five character pointers (strings) (c) Five pointers to arrays (d) A two-dimensional character array

12. A function pointer is used to:

(a) Store the return value of a function (b) Store the address of a function (c) Declare a function prototype (d) Allocate memory for a function

13. The declaration int **pp; represents:

(a) A pointer to an integer array (b) A pointer to a pointer to an integer (c) An array of integer pointers (d) A double-precision integer variable

14. Which function allocates dynamic memory and initializes all of it to zero?

(a) malloc() (b) calloc() (c) realloc() (d) free()

15. A pointer that refers to a memory location that has already been freed is called a:

(a) Null pointer (b) Wild pointer (c) Dangling pointer (d) Generic pointer

Answer Key

1-b 2-b 3-a 4-b 5-c

6-c 7-a 8-b 9-a 10-a

11-b 12-b 13-b 14-b 15-c

Broad (Descriptive) Questions

Answer the following 10 questions in detail, with suitable examples and diagrams/code wherever applicable.

1. Define pointer. Explain the concept of pointers with a suitable example, and discuss why pointers are considered a powerful feature of the C language.

2. Explain the syntax for declaring and initializing a pointer variable. Differentiate between the address-of (&) operator and the dereference (*) operator with examples.
3. What is a null pointer? Explain its significance and demonstrate, with a code snippet, how to safely check for a null pointer before use.
4. Discuss the concept of generic (void) pointers. Why can a void pointer not be dereferenced directly, and how is it typecast for use?
5. Explain pointer arithmetic in C. List the valid and invalid arithmetic operations on pointers, with suitable examples of each.
6. Describe how arguments can be passed to a function using pointers (call by reference). Write a C program to swap two numbers using pointers.
7. Explain the relationship between arrays and pointers in C. How is an array name treated when it is passed to a function, and what information is lost in this process?
8. Differentiate between an array name and a pointer variable, highlighting at least four key differences with respect to memory allocation and behaviour.
9. Explain pointers and strings, array of pointers, and pointers to pointers with the help of example programs. In what situations is each of these particularly useful?
10. What is dynamic memory allocation? Explain the library functions malloc(), calloc(), realloc(), and free() with syntax and examples, and discuss the drawbacks/risks associated with the use of pointers.