C Programming For Absolute Beginners Part II, Pointers and Dynamic Memory

Sahas Talasila

Contents

Arrays

They are the simplest data structure that we will use. You can think of them as ***static blocks of contiguous memory*** (contiguous means right next to each other), which we will use to store items. Before we start, arrays in C and C++ are ***zero-based***, meaning we will start the count at zero, and at the end of an array. Below, we will look at what we can do with arrays in Listing 1, but first, here's a visual representation of an array in **Figure 1**:

Figure 1: Array, visualised

```c

#include <stdio.h>

int main() {
    // One-dimensional array declaration and initialisation (1)
    int numbers[5] = {10, 20, 30, 40, 50};

    // Accessing and modifying array elements (2)
    printf("First element: %d\n", numbers[0]);  // Prints 10
    numbers[2] = 35;  // Modify third element

    // Iterating through an array (3)
    printf("Array contents: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", numbers[i]);
    }
    printf("\n");

    // Multidimensional array (matrix) (3)
    int matrix[2][3] = {
        {1, 2, 3},    // First row
        {4, 5, 6}     // Second row
    };

    // Nested loops for matrix traversal (4)
    printf("Matrix contents:\n");
    for (int row = 0; row < 2; row++) {
        for (int col = 0; col < 3; col++) {
            printf("%d ", matrix[row][col]);
        }
        printf("\n");
    }

    return 0;
}
```

Listing 1: Array and Matrix Operations

We initialise the array (1) by using square brackets. In the square brackets, we specify how large the array will be, so `int numbers[5]` means **An array of 5 integers, with 5 indices**.

We can access the array by typing the name and the position/index we want to see (2). We can also iterate through an array using a **for** loop, which I mentioned earlier. This is shown in Listing 1. We can also create 2D matrices by adding another set of square brackets, to simulate a matrix.

1.1 1. Declaring and Initialising a One-Dimensional Array

`int numbers[5] = {7, 25, 9,13, 42};`

This line declares an array named **numbers**, which can hold **5 integers**. In C, the number inside the square brackets (in this case, 5) specifies how many elements the array will store. We then use braces `{ }` to give the array its initial values.

`int arr[5] = {7, 25, 9, 13, 42};`

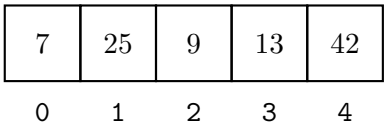


Figure 2: Array Indices and Access

Once again , C arrays are ***zero-indexed***, meaning:

- `numbers[0]` corresponds to the first element (which is 7 here),
- `numbers[1]` is the second element (first index = 25),
- ...
- `numbers[4]` is the fifth element (4th index = 50).

If we tried to access `numbers[5]` or beyond, we would be going *out of bounds*, which leads to *undefined behavior* or memory errors. Listing 2 shows access to elements.

```
1 printf("First element: %d\n", numbers[0]);
2 numbers[2] = 7;
```

Listing 2: Element modification and access

Here, we see two operations:

- **Accessing an element**: We use `numbers[0]` to print out the first element (7).
- **Modifying an element**: We change the third element `(numbers[2])` from `9` to `99`.

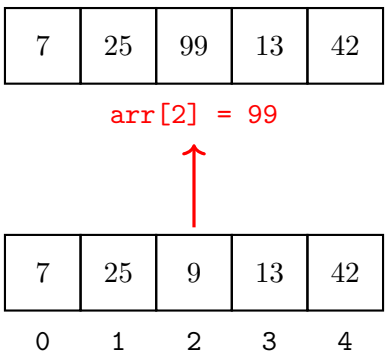


Figure 3: Element Modification

1.2 Iterating (Looping) Through the Array

```c
for (int i = 0; i < 5; i++) {
    printf("%d ", numbers[i]);
}
```

Listing 3: Array Iteration

This `for` loop starts at `i = 0` and continues up to `i < 5`, thereby printing all five elements in the array. Each iteration prints `numbers[i]`. Listing 3 shows how we can use for loops from Set 1 for this case. Loops are commonly used for:

- Printing or displaying array contents
- Performing calculations on each element (e.g., summing up values)
- Searching for specific values

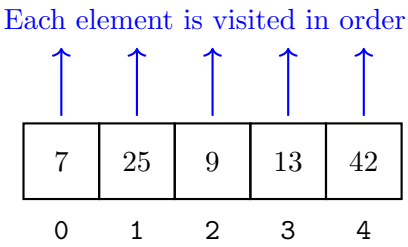


Figure 4: Array Iteration

1.3 Declaring and Initialising a Multidimensional Array

A matrix in mathematics is a structured arrangement of numbers in rows and columns. In programming, particularly in C, C++, and Python, matrices are represented using **two-dimensional (2D) arrays**.

A matrix of order $m \times n$ has:

- m rows
- n columns

For example, the following is a 3×3 matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

1.3.1 Matrix Representation in Programming

In programming, a matrix is stored as a **2D array**, where each row is an array and multiple rows form a larger array structure, we can alter our initial array image so it looks like this:

12	47	33
5	89	26
71	14	62

1.3.2 C Representation

In C, a 2D array is declared as:

```
int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

Here, the matrix is stored **row-wise** in memory. This is a special feature of the C language.

1.3.3 Accessing Elements

Each element $A[i][j]$ in the matrix can be accessed using its row and column indices.

`printf("%d", matrix[1][2]); // Output: 6`

In this example, `matrix[1][2]` refers to the element in the **second row** and **third column** (0-based indexing).

1.3.4 Memory Representation

Internally, a 2D array is stored in **contiguous memory locations** (slots of memory right next to each other):

$$\text{Row-wise storage: } [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

The compiler maps the index using:

$$\text{Address} = \text{Base Address} + (i \times \text{column size} + j)$$

where i is the row index and j is the column index. Listing 4 shows this clearly.

```
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Listing 4: Row and column indices

A *multidimensional array* in C is essentially an array of arrays. In this example:

- `matrix` has **2 rows** and **3 columns**.
- The first row is `{1, 2, 3}`.
- The second row is `{4, 5, 6}`.

Conceptually, you can imagine `matrix` as a small table, as I have previously mentioned.

1.4 5. Traversing a 2D Array (Nested Loops)

```
for (int row = 0; row < 2; row++) {
    for (int col = 0; col < 3; col++) {
        printf("%d ", matrix[row][col]);
    }
    printf("\n");
}
```

Listing 5: Matrix Traversal

Here, we use two **nested** `for` loops:

- The `row` loop goes from `0` to `1` (2 rows).
- The `col` loop goes from `0` to `2` (3 columns).

Inside these loops, `matrix[row][col]` accesses each element. This results in printing each row of the matrix on a new line. Listing 5 shows how they can accessed.

1.5 Summary of Key Takeaways

- **Array Declaration and Size**: `int numbers[5]` defines an array of size 5. The size must be known at compile time for static arrays.
- **Zero-Based Indexing**: Always remember that indices in C start at 0. Thus, valid indices for an array of size `n` range from `0` to `n-1`.
- **Initialisation**: You can initialise an array at declaration using braces, e.g., `{10, 20, 30, 40, 50}`.
- **Accessing and Modifying**: Use the array name plus an index (e.g., `numbers[i]`) to read or change that element's value.
- **Iterating Through an Array**: `for (int i = 0; i < size; i++)` is the typical approach for working through every element in sequence.
- **Multidimensional Arrays**: Similar to one-dimensional arrays, but you have multiple indices (e.g., `matrix[row][col]` for a 2D array).
- **Nested Loops for 2D Arrays**: One loop for rows, another loop for columns, to traverse the entire matrix.

Putting these concepts into practice will help solidify your understanding of how arrays and matrices work in C.

Next, we will look at memory in a lot of detail, specifically talking about the **stack, heap** and **memory interaction**. This section is **OPTIONAL**, I will not ask any questions relating to the heap and stack. There is also the stack data structure, which is different from the stack in memory.

A **stack** is a linear data structure that follows the **Last-In-First-Out (LIFO)** principle. Elements are added (pushed) and removed (popped) only from the **top** of the stack. It is very similar to how an array works.

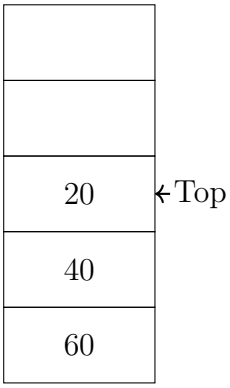


Figure 5: Accessing Top-most Element in Stack

Key operations of a stack:

- **Push** – Add an element to the top.
- **Pop** – Remove the top element.
- **Peek/Top** – View the top element without removing it.
- **IsEmpty** – Check if the stack has no elements.

Here is an example of a push operation below:

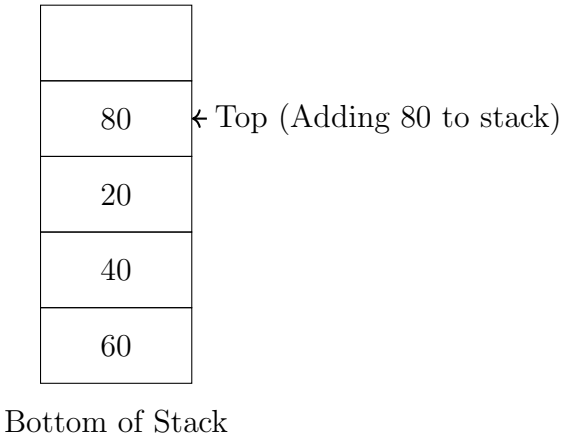


Figure 6: Push Operation

And a pop operation (next page):

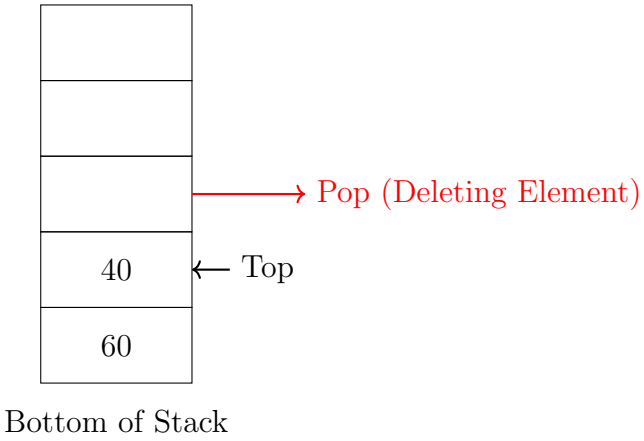


Figure 7: Pop Operation

Stack can be implemented in C using arrays. A simple implementation:

```c
#define MAX 100
int stack[MAX];
int top = -1;

void push(int value) {
    if (top < MAX - 1)
        stack[++top] = value;
}

int pop() {
    if (top >= 0)
        return stack[top--];
}
```

Listing 6: Stack operations

Using a struct provides better encapsulation and modularity:

```c
typedef struct {
    int data[100];
    int top;
} Stack;

void push(Stack *s, int value) {
    if (s->top < 99)
        s->data[++(s->top)] = value;
}
```

Listing 7: Stacks with structs

Comparison between a stack and a regular array:

- An **array** offers direct access to any element by index; it is static and doesn't enforce access order.
- A **stack** enforces order—only the most recent element is accessible for removal or inspection.
- While a stack can be implemented using an array, it provides abstraction by restricting operations to `push()`, `pop()`, etc.

Stacks are useful in scenarios like expression evaluation, recursion management, backtracking, and undo functionality. They abstract memory and flow control in a structured, LIFO-constrained way.

Memory

In the C programming language (and many other compiled languages), the program's memory is typically organised into several regions. Two of the most important are:

- The **stack**: for local variables, function parameters, and function-call management (roughly 8 MB).
- The **heap**: for dynamically allocated memory (e.g., via `malloc`, `free`). Heap size varies depending on the operating system.

I will provide a concise explanation of how the stack and heap work, their differences, and their relationship in a typical C program. Again, this section is **OPTIONAL**, but the extra knowledge will be very useful for memory-efficient program design.

Skip to the 'Pointers' section if you aren't interested

2.1 High-Level Memory Layout

A simplified view of memory layout on many systems might look like Figure 8.

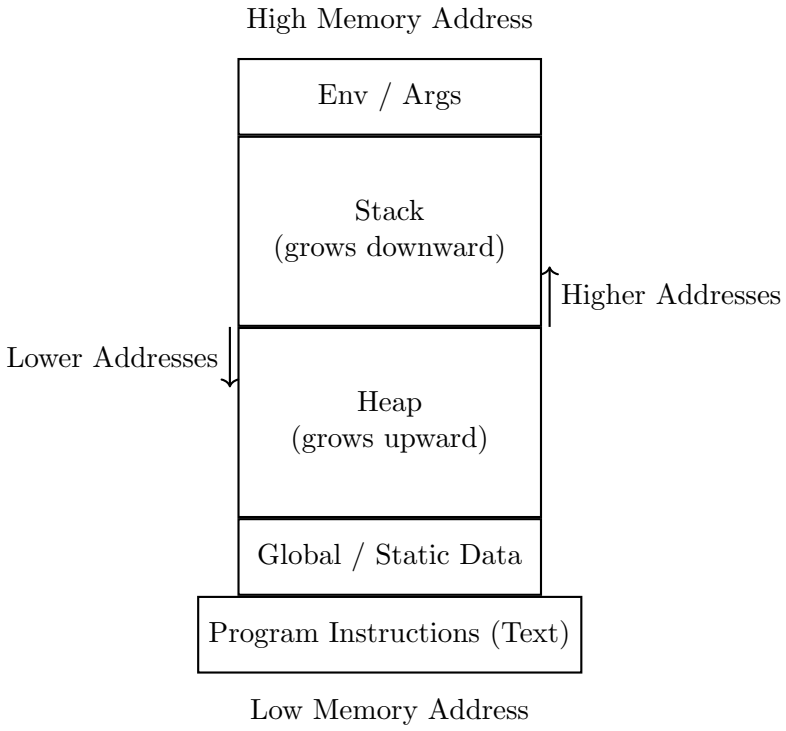


Figure 8: A Simplified Memory Layout in C

- **Program Instructions (Text Segment)**: Holds the compiled machine code.
- **Global/Static Data**: Stores global and static variables.
  - Sometimes split into *initialised data* and *BSS* (uninitialised data).
- **Heap**: Used for dynamically allocated memory. Grows upward in most implementations.
- **Stack**: Used to manage function calls, local (automatic) variables, parameters, and return addresses. Grows downward in most implementations.
- **Env/Args**: Stores environment variables, command-line arguments, etc.

2.2 The Stack

2.2.1 What Is the Stack?

The **stack** is a region of memory that follows a **Last-In-First-Out** (LIFO) discipline. Each function call in C creates a new *stack frame* that typically includes:

- **Local variables** (automatic variables).
- **Function parameters**.
- **Return address** (the address where the function should return when finished).

2.2.2 Growth and Lifetime

When one function calls another, the stack pointer moves (often downward) to allocate space for the new function's local variables and parameters. When that function returns, the stack pointer moves back, effectively discarding those variables.

Key points:

- Allocation is *automatic* and very fast (just pointer arithmetic).
- Lifetime of local variables ends as soon as the function returns.
- The stack has a limited size; large local arrays or deep recursion can cause *stack overflow*.

2.3 The Heap

2.3.1 What Is the Heap?

The **heap** is a region for *dynamic* memory allocation. In C, you request memory with:

- `malloc()` - allocates a block of bytes.
- `calloc()` - allocates and zeroes a block.
- `realloc()` - resizes an existing block.
- `free()` - releases previously allocated memory.

2.3.2 Growth and Lifetime

Heap memory typically grows upward, starting above the static data region. Unlike the stack, where allocation and deallocation happen automatically as functions are called and return, *you* (the programmer) control when to allocate and free memory on the heap.

Key points:

- Must call `free()` to release memory no longer needed.
- Failing to free leads to *memory leaks*.
- Requesting very large blocks or repeated allocations without proper frees can exhaust available memory.
- Allocation on the heap is generally slower than on the stack due to more complex bookkeeping.

2.4 Relationship Between Stack and Heap

- Typically, the stack is near the top of the process's memory space and grows downward, while the heap is lower in memory and grows upward.
- If they grow into each other (which can happen in extreme cases), memory allocation fails.
- **Stack** is ideal for short-lived, known-size local variables; **heap** is for dynamic data whose size or lifetime can't be known in advance (e.g., user input, variable-sized structures, etc.).
- Stack and Heap access has been shown below in Listing 6.

2.5 Sample Code Demonstration

```c
/* Demonstrates using stack and heap in a single function */

#include <stdio.h>
#include <stdlib.h>

void example() {
    // 1. Stack allocation:
    int localVar = 42;              // allocated on the stack

    // 2. Heap allocation:
    int *heapVar = malloc(sizeof(int));
    if (!heapVar) {
        fprintf(stderr, "Memory allocation failed\n");
        return;
    }
    *heapVar = 100;                 // store a value in the heap-allocated memory

    printf("localVar (on stack): %d\n", localVar);
    printf("*heapVar (on heap): %d\n", *heapVar);

    // 3. Freeing heap memory
    free(heapVar);
}

int main(void) {
    example();
    return 0;
}
```

Listing 8: Stack and heap access example

- **localVar** is automatically allocated and freed when `example()` returns.
- **heapVar** points to memory on the heap. We manually allocate with `malloc` and release it with `free`.

2.6 Conclusion

In summary, understanding the distinction between the **stack** and the **heap** is crucial in C:

- **Stack**: Automatic, limited size, fast, local scope.
- **Heap**: Manual, large and flexible, slower allocation, must be freed explicitly.

Using them properly ensures efficient and error-free programs.

Pointers

Pointers are variables that store the memory address of other variables. We use them to make our program simpler, especially if we have a large program with lots of variables and functions. The key reason we use them is to create **dynamic programs**. Here's a simple example below (Listing 7), along with a diagram to show the **referencing** aspect:

```c
#include <stdio.h>

int main() {
    int x = 10;
    int *ptr = &x; // Pointer to x

    printf("Value of x: %d\n", x);
    printf("Address of x: %p\n", &x);
    printf("Value stored in ptr: %p\n", ptr);
    printf("Value pointed by ptr: %d\n", *ptr);

    return 0;
}
```

Listing 9: Referencing with pointers and addresses

We use the star operator (*), to **reference AND dereference** the pointers. The & symbol is the address symbol, telling us what the address actually is to us, the people using the program and the computer.

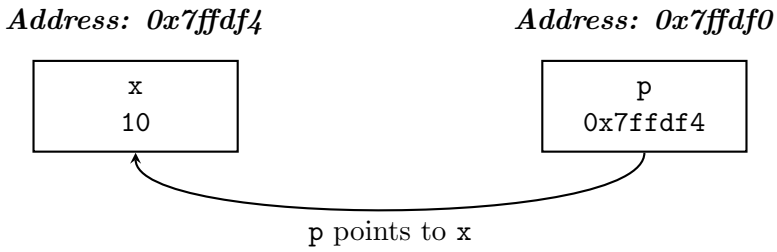


Figure 9: Pointer Referencing

3.1 Pointer Arithmetic

Pointer Arithmetic and Operations in C:
Pointers are integral in C for memory manipulation and accessing data efficiently. Pointer arithmetic allows you to perform operations on pointers to traverse through memory addresses or manipulate them.

Basic Pointer Arithmetic:
Increment (++): Moves the pointer to the next memory location of its type.
Decrement (–): Moves the pointer to the previous memory location of its type.
Addition/Subtraction (+, -): Adds or subtracts an integer to/from the pointer, moving it forward or backward by that many elements.
Difference (-): The difference between two pointers gives the number of elements between them.

Here are the examples on the next page, with the first part looking at the address of the pointer and how it can be altered using the **'increment'** and **'decrement'** operations. Listings 8, 9 and 10 show the possible operations.

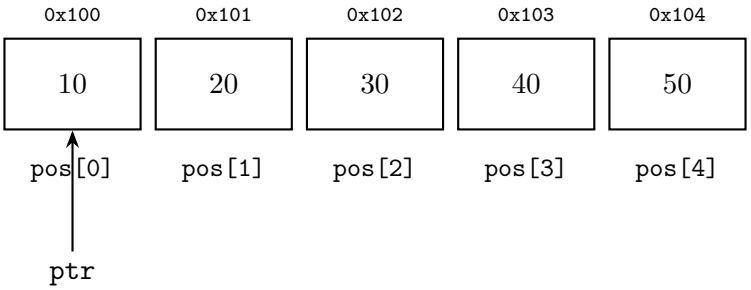


Figure 10: Pointer Visibility

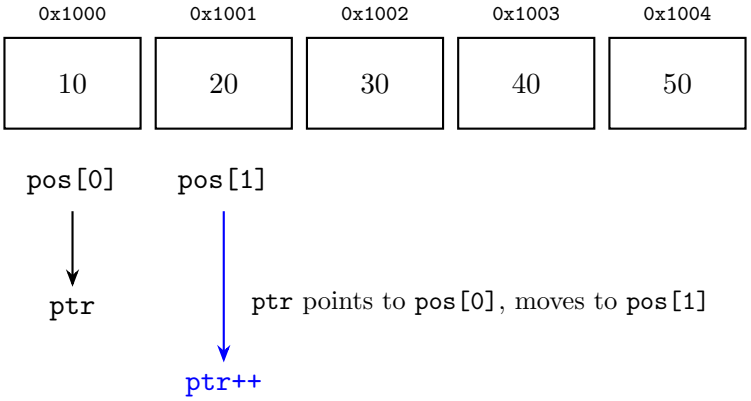


Figure 11: Pointer Increment

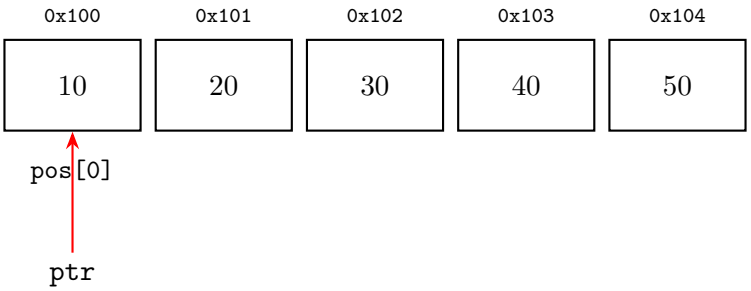


Figure 12: Pointer Decrement (moving back from the position in Figure 11)

```c
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr;

    printf("Initial pointer address: %p\n", ptr);
    printf("Value at pointer: %d\n", *ptr);

    ptr++; // Move to the next element
    printf("After increment, address: %p, value: %d\n", ptr, *ptr);

    ptr--; // Move back to the previous element
    printf("After decrement, address: %p, value: %d\n", ptr, *ptr);

    return 0;
}
```

Listing 10: Pointer Arithmetic

And adding and subtracting values with pointers:

```c
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr;

    printf("Value at pointer: %d\n", *ptr);

    ptr = ptr + 2; // Move forward by 2 elements
    printf("After adding 2, value: %d\n", *ptr);

    ptr = ptr - 1; // Move back by 1 element
    printf("After subtracting 1, value: %d\n", *ptr);

    return 0;
}
```

Listing 11: Incrementing and decrementing with pointers

And another example of difference pointer arithmetic:

```c
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr1 = &arr[1];
    int *ptr2 = &arr[4];

    printf("Address of ptr1: %p, ptr2: %p\n", ptr1, ptr2);
    printf("Difference between ptr2 and ptr1: %ld\n", ptr2 - ptr1);

    return 0;
}
```

Listing 12: Pointer differences

Below we can use pointers to iterate/traverse arrays. This is another example of **pass-by-reference**.

```c
// We can also use a pointer to traverse an array
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr;

    printf("Array elements: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", *(ptr + i)); // Access elements using pointer arithmetic
    }

    return 0;
}
```

Listing 13: Iterating through an array by reference

Pointers are very useful and powerful, but we have to be careful when we use them. There are lots of ways in which pointer usage can go wrong. I have some examples below.

```c
//Accessing Memory Outside the Array Bounds
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr;

    for (int i = 0; i < 7; i++) { // Array size is 5, loop goes beyond bounds
        printf("%d\n", *(ptr + i)); // Undefined behavior when i >= 5
    }

    return 0;
}
```

Listing 14: Example of out-of-bounds errors

```c
// Issue: Dereferencing an uninitialised pointer leads to garbage values or crashes.
#include <stdio.h>

int main() {
    int *ptr; // Pointer is not initialised
    printf("Value at uninitialised pointer: %d\n", *ptr); // Undefined behavior

    return 0;
}
```

Listing 15: Incorrect pointer referencing

In programming, especially in languages like C and C++, a null pointer is a pointer that does not point to any valid memory location or object. It is often represented by the value `NULL` (in C). It is useful for more complex data structures, so you don't have to worry too much about them for now.

```c
// Pointer Arithmetic on NULL or Dangling Pointers
#include <stdio.h>

int main() {
    int *ptr = NULL; // NULL pointer
    ptr++;           // Invalid arithmetic on NULL pointer
    printf("Value: %d\n", *ptr); // Undefined behavior

    return 0;  // Issue: Performing arithmetic on a NULL pointer is invalid.
}
```

Listing 16: NULL and dangling pointer operations

3.2 Dynamic Memory Management

There are 4 memory related commands that you need to worry about. Let's start with `malloc`. It is displayed like this: `malloc(size_t * size)`, which allocates a block of memory of `size` bytes. It returns a pointer of type `void*` (which can be cast to the appropriate type (remember from the start of the document)) to the beginning of the block. If the allocation fails, `malloc` returns `NULL`. Memory is uninitialised (i.e., it contains garbage data). This has been shown in Listing 15.

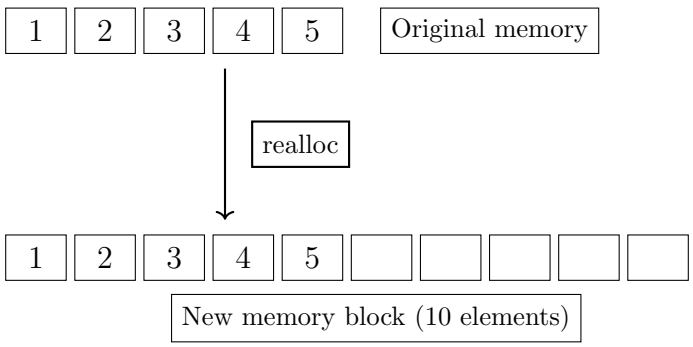
```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    // Allocate memory for an array of 10 integers
    int *arr = malloc(10 * sizeof(int));

    if (arr == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    // Initialize and use the array
    for (int i = 0; i < 10; i++) {
        arr[i] = i * 2;
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Free the allocated memory
    free(arr);

    return 0;
}
```

Listing 17: Dynamic memory management using the Heap

Next, we have `realloc`, which reallocates the memory we have initialised for another use case or purpose.

It resizes a previously allocated block (pointed to by `ptr`) to `new_size` bytes. The content of the old memory block is copied to the new location (if the new block is at a different address), up to the minimum of the old and new sizes.

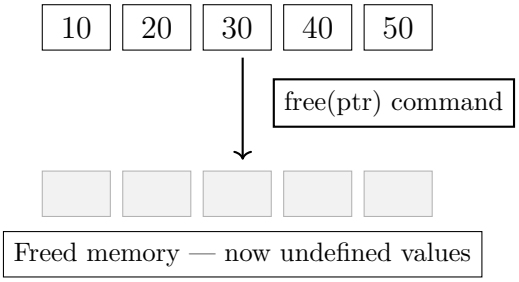
If `ptr` is `NULL`, `realloc` behaves like `malloc(new_size)`. If `new_size` is 0, `realloc` behaves like `free(ptr)`, and returns `NULL`. (We will look at `free` next).



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

int main(void) {
    // Allocate memory for 5 integers
    int *arr = malloc(5 * sizeof(int));
    if (arr == NULL) {
        fprintf(stderr, "Initial allocation failed\n");
        return 1;
    }

    // Initialise the 5 integers
    for (int i = 0; i < 5; i++) {
        arr[i] = i + 1;
    }

    // Resize the array to hold 10 integers
    int *temp = realloc(arr, 10 * sizeof(int));
    if (temp == NULL) {
        fprintf(stderr, "Re-allocation failed\n");
        // We still have arr allocated, so we should free it.
        free(arr);
        return 1;
    }
    arr = temp; // update our pointer to the new location

    // Initialize the new elements
    for (int i = 5; i < 10; i++) {
        arr[i] = (i + 1) * 10;
    }

    // Print all 10 elements
    for (int i = 0; i < 10; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

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

Listing 18: Full example of dynamic memory usage

Lastly, we'll look at `free()`, this essentially 'clears' the memory we allocated at the start. If you use `malloc`, then you **MUST** use `free()`, they essentially work in a pair, if `free()` isn't used, then there will be memory leaks and overflow errors, which causes the compiler and machine to crash. Correct usage has been shown in Listing 18.



Here are some best practices:

Check Return Values:
Always check if the pointer returned by `malloc, calloc, or realloc is NULL` before using it.

Avoid Memory Leaks:
Every successful `malloc, calloc, or realloc` call must eventually be paired with a corresponding free call to release the allocated memory.

Initialise Pointers:
Set pointers to `NULL` when you declare them. If you free a pointer, consider setting it to `NULL` immediately afterward.

Use `calloc` When Needed:
calloc automatically zero-initialises the memory, which is useful for certain applications (e.g., arrays that require initial values of zero).

Be Careful with `realloc`:
Use a temporary pointer when using realloc to avoid losing the original pointer if realloc fails.
Remember that if realloc fails and returns NULL, your original pointer is still valid.

Avoid Dangling Pointers:
If you free a pointer, do not dereference it or use it again unless you reassign it properly.

Segmentation Faults:
Occur if you write beyond the allocated region (e.g., indexing past the end of an array). This invokes undefined behavior and can crash or corrupt data.

Casting `malloc` in C:
In pure C, it is not required to explicitly cast the return of malloc (because void* can be implicitly converted).

Best Practices for Dynamic Memory

- Always check return values of memory functions.
- Avoid memory leaks — free all allocated memory.
- Initialize pointers and set them to `NULL` after freeing.
- Use `calloc` for zero-initialized memory.
- Handle `realloc` carefully using a temporary pointer.
- Avoid dangling pointers and segmentation faults.
- No cast is needed when using `malloc` in C.

On the next page, you can see the questions, ranging from multiple choice to short answer to open-ended. I would expect these to be done in a week.

Memory Management Questions

4.1 Multiple-Choice Questions (10)

1. Which of the following statements correctly declares a pointer to an `int`?
   (a) `int ptr;`
   (b) `int *ptr;`
   (c) `int &ptr;`
   (d) `pointer int;`
2. Given `int arr[5];`, how do you correctly get the address of the first element?
   (a) `&arr`
   (b) `arr + 1`
   (c) `arr[0]`
   (d) `&arr[0]`
3. Which of the following functions dynamically allocates memory for an array in C?
   (a) `malloc()`
   (b) `alloca()`
   (c) `scanf()`
   (d) `printf()`
4. What does the `free()` function do?
   (a) Initialises a dynamic array
   (b) Allocates memory on the stack
   (c) Deallocates memory previously allocated by `malloc()`, `calloc()`, or `realloc()`
   (d) Returns the size of allocated memory
5. Which pointer arithmetic operation moves the pointer by the size of one element?
   (a) `p + 1`
   (b) `p * 2`
   (c) `&p + 1`
   (d) `p / 1`
6. If `ptr` is a pointer to `int`, which expression correctly dereferences `ptr`?
   (a) `*ptr`
   (b) `&ptr`
   (c) `ptr*`
   (d) `(ptr)`

7. Which of the following statements about `NULL` pointers is **true**?
   (a) `NULL` is defined as `(void *)1`
   (b) A `NULL` pointer does not point to any valid address
   (c) You can safely dereference a `NULL` pointer
   (d) `NULL` is the same as `0.0`
8. Which library should be included to use `malloc()` and `free()`?
   (a) `#include <stdio.h>`
   (b) `#include <stdlib.h>`
   (c) `#include <memory.h>`
   (d) `#include <string.h>`
9. What happens if you call `free()` on a pointer that was not allocated by `malloc()`, `calloc()`, or `realloc()`?
   (a) It safely does nothing
   (b) It will always crash the program
   (c) The behavior is undefined
   (d) The memory doubles
10. What does the function prototype `int *f(void);` indicate?
    (a) `f` is a pointer to an `int`
    (b) `f` is a function that returns a pointer to `int`
    (c) `f` is a function pointer to `int`
    (d) `f` is an integer variable

4.2 Short-Answer Questions (10)

1. What is the difference between a pointer and an array in C in terms of how they are stored or used?
2. Write a single `malloc()` call (using the correct syntax) that allocates memory for 10 `float` values.
3. Explain briefly what a `NULL` pointer is and why we often check for it before using a pointer.
4. In one sentence, describe pointer arithmetic: how does adding 1 to a pointer differ from adding 1 to a normal integer?
5. What is the purpose of the `sizeof` operator in C, and how is it commonly used with pointers?
6. Provide an example of how to correctly `free()` memory allocated with `malloc()`.
7. Name two functions (other than `malloc()`) that can be used to allocate memory dynamically in C.
8. If you have `int *p = NULL;`, is it safe to do `*p = 10`? Why or why not?
9. Write a single statement that declares a pointer to `char` named `strPtr` and initialises it to `NULL`.
10. How does passing a pointer to a function allow that function to modify the original data in the caller?

4.3 10 Coding Tasks: Pointers and Dynamic Memory Management

1. **Pointer-based Array Reversal:** Write a function `void reverseArray(int *arr, int n)` that reverses the elements of an integer array `arr` *in place* using only pointer arithmetic (no indexing with `[]`). Demonstrate the function in `main()` by reading an array from the user, calling `reverseArray()`, and then printing the reversed array.
2. **Dynamic Array (Manual Resizing):** Implement a program that reads integers from the user until they enter a negative number. Use `malloc()` (and `realloc()` when necessary) to dynamically expand the array as more integers are read. After the user stops entering numbers, print them back to confirm the contents.
3. **Matrix Allocation (2D Array):** Write a program that prompts the user for the dimensions `m` (rows) and `n` (columns). Dynamically allocate a 2D array of integers (using a pointer to pointers or a single pointer with row offset calculation). Fill it with data from the user, then print the matrix in a neatly formatted table.
4. **Linked List Insertion:** Create a singly linked list data structure using a `struct` for the node (containing `data` and a pointer to the next node). Implement a function `void insertAtEnd(Node **head, int value)` that inserts a new node at the end of the list. In `main()`, build a list of integers from user input and print the final list.
5. **String Duplication Function:** Write a function `char *myStrDup(const char *src)` that dynamically allocates enough memory to hold a copy of the string `src`, copies it, and returns a pointer to the new string. Test it by printing both the original and the duplicated strings. Make sure to `free()` the duplicated string after use.
6. **Pointer-based String Split (Basic):** Prompt the user for a string and a character `delim`. Write a function that scans the string (using pointers), and splits it into two new dynamically allocated strings at the first occurrence of `delim`. Print both resulting strings separately, then `free()` them.
7. **Sorting with Pointer Arithmetic:** Implement a function `void pointerSort(int *arr, int n)` that sorts an integer array in ascending order but uses pointer notation for iterating and swapping elements instead of array indexing. In `main()`, read `n`, dynamically allocate an integer array, fill it with user data, call `pointerSort()`, then print the sorted array.
8. **File I/O with Dynamic Buffers:** Write a program that opens a text file (prompt the user for the file name), reads its contents dynamically into a buffer (using `malloc()` or `realloc()`), then displays the file content on the screen. Ensure you properly `free()` memory at the end and handle any file or memory allocation errors gracefully.
9. **Command-line Argument Parsing:** Create a program that takes command-line arguments for a list of numbers (e.g., `./prog 10 20 30`). Dynamically store these numbers in an integer array, compute the sum, and print the result. If no numbers are provided, print an appropriate error message.
10. **Dynamic "Safe" String Concatenation:** Write a function `char *concatStrings(const char *s1, const char *s2)` that returns a dynamically allocated string containing `s1` followed by `s2`. Demonstrate it by reading two strings from the user, calling `concatStrings()`, printing the result, and then freeing the allocated memory.

4.4 Spot The Error Questions

1. Spot the errors if there are any and fix it.

```c
#include <stdio.h>

int main() {
    int *ptr;
    printf("Value: %d\n", *ptr);
    return 0;
}
```

2. Spot the errors if there are any and fix it.

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

int main() {
    int *ptr = (int*)malloc(sizeof(int));
    *ptr = 42;
    return 0;
}
```

3. Spot the errors if there are any and fix it.

```c
#include <stdio.h>

int main() {
    int num = 10;
    int *ptr = &num;
    ptr = ptr + 3;
    printf("Address: %p\n", (void*)ptr);
    return 0;
}
```

4. Spot the errors if there are any and fix it.

```c
#include <stdio.h>

int main() {
    int arr[3] = {1, 2, 3};
    printf("Fourth element: %d\n", arr[3]);
    return 0;
}
```

5. Spot the errors if there are any and fix it.

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

int main() {
    int *ptr = (int*)malloc(sizeof(int));
    free(ptr);
    free(ptr);
    return 0;
}
```

6. Spot the errors if there are any and fix it.

```c
#include <stdio.h>

int main() {
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int *ptr = arr;
    printf("%d\n", *ptr);
    return 0;
}
```

7. Spot the errors if there are any and fix it.

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

int main() {
    int *ptr = (int*)malloc(sizeof(int));
    free(ptr);
    printf("%d\n", *ptr);
    return 0;
}
```

8. Spot the errors if there are any and fix it.

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

int main() {
    int **matrix;
    matrix = (int**)malloc(3 * sizeof(int*));
    for (int i = 0; i < 3; i++) {
        matrix[i] = (int*)malloc(3);
    }
    return 0;
}
```

For advanced data structures and algorithms, please wait. I am still creating that section.