A05 Cheatsheet - Ria Sharma

Linux Operating System Commands:

- **man:** Short for "manual," this command displays the user manual of any command that we type on the console. It provides a detailed view of the command's function, options, and usage.
    - Example: man ls shows the manual for the ls command.
- **ls:** Lists directory contents. It can show file and directory names, file types, permissions, and other metadata depending on the options used.
    - ls -l
        - lists files with detailed information.
    - ls -a
        - Hidden files also
    - Ls -al
        - Hidden + all with details
- **cat:** Concatenates and displays files. It can be used to view the contents of a file, create a new file, and concatenate files.
    - Example: cat file1
        - displays the content of file1.
- **echo:** Prints text to the terminal window. It's often used in scripting to display messages or output the value of variables.
    - Example: echo "Hello, World!"
        - prints Hello, World! to the console.

File path, filename: absolute vs relative:

- **Absolute path:** Specifies a file or directory location in relation to the root directory (/). It always starts with a slash.
    - Example: /home/user/document.txt.
- **Relative path:** Specifies a file or directory location in relation to the current directory. It does not start with a slash.
    - Example: document.txt (if you're already in the directory where document.txt is located).
- **ssh:** Secure Shell, a protocol used to securely access remote servers. It's used for secure data communication, remote command-line login, remote command execution, and other secure network services.
    - Example: ssh user@host
        - to connect to a remote host as "user".

Andrew File System (AFS) Commands to Manage Your Linux Networked File Space:

- **cd:** Changes the current directory. It's used to navigate through the directories in the file system.
    - Example: cd /home/user
        - changes the current directory to /home/user.
- **mkdir:** Creates a new directory.

    - Example: mkdir new_directory
        - creates a new directory named new_directory.
- **rmdir:** Removes a directory. It only works on empty directories.
    - Example: rmdir old_directory
        - removes old_directory if it is empty.
- **mv:** Moves or renames files and directories.
    - Example: mv old_name new_name
        - renames old_name to new_name, or moves it to a new location if the path is specified.
- **rm:** Removes files or directories.
    - Example: rm file.txt deletes file.txt.
    - Use rm -r to recursively delete a directory and its contents.

cp vs scp:

- **cp:** Copies files and directories within the same system.
    - Example: cp source.txt destination.txt
        - copies source.txt to destination.txt.
- **scp:** Secure Copy Protocol, used to securely transfer files between systems in a network.
    - Example: scp source.txt user@host:/path/destination.txt
        - copies source.txt to a remote host.

GCC and GDB Usage:

- **GCC:** GNU Compiler Collection, a compiler system for C, C++, and other programming languages.
    - The command gcc prog.c -Wall -m32 -std=gnu99 -o prog compiles a C program prog.c with all warnings enabled (-Wall), in 32-bit mode (-m32), adhering to the C99 standard (-std=gnu99), and outputs an executable named prog.
- **GDB:** GNU Debugger, used for debugging programs written in C, C++, and other languages.
    - To debug a program, compile it with the -g option (e.g., gcc -g prog.c -o prog) to include debugging information.
    - Launch GDB with the executable (gdb prog), and use commands like run, list, next, step, print, and quit to control the debugging session.

VIM: Create File, Insert Text, Escape to Command Mode, Save File, and Exit VIM

- **VIM:** A highly configurable text editor built to enable efficient text editing. It's an improved version of the vi editor distributed with most UNIX systems.
- **Create file:** Open or create a new file with vim filename.
- **Insert text:** Press i to enter insert mode, then type your text.
- **Escape to command mode:** Press Esc to return to command mode, where you can enter commands.
- **Save file:** In command mode, type :w to save changes.

- **Exit VIM:** Type :q to quit. If you've made changes, save them first with :w or use :wq to save and quit, or :q! to quit without saving.

C Programming

Libraries and Standard Functions:

- **stdio.h:** This library is fundamental for input and output operations in C programs.
    - **printf:** Allows you to output text and formatted data to the standard output (usually the screen). It can handle various data types by using format specifiers (e.g., %d for integers, %s for strings).
        - Example usage: printf("Age: %d\n", age);
    - **scanf:** Used for input, it reads formatted data from the standard input (usually the keyboard). Like printf, it uses format specifiers to determine the type of data being read and requires addresses of variables to store the input data.
        - Example: scanf("%d", &age);
- **stdlib.h:** This library provides functions for memory allocation, process control, conversions, and more.
    - **Dynamic Memory Management Functions:**
        - **malloc:** Allocates a block of memory of the specified size and returns a pointer to the beginning. The memory is not initialized.
            - Example: int *arr = malloc(n * sizeof(int));
                - int *arr: This declares a pointer arr of type int. This pointer will store the address of the first byte of the allocated memory block.
                - malloc(n * sizeof(int)): This is the call to malloc.
                - n: Represents the number of elements you want in your array.
                - sizeof(int): Calculates the size of an int in bytes. Assuming an int is 4 bytes (as per your example), sizeof(int) evaluates to 4.
                - n * sizeof(int): Calculates the total number of bytes required to store n integers.
            - Int - 4 bytes
            - Char - 1 byte
            - Double - 8 bytes
            - Float - 4 bytes
            - Pointer to int - 4 bytes
        - **calloc:** Similar to malloc but initializes the allocated memory to zero.
            - Example: int *arr = calloc(n, sizeof(int));
        - **realloc:** Adjusts the size of previously allocated memory block.
            - Example: arr = realloc(arr, newsize * sizeof(int));
        - **free:** Frees the memory that was previously allocated by malloc, calloc, or realloc.
            - Example: free(arr);

- **string.h:** Useful for handling strings, this library includes functions for copying, concatenation, comparison, and calculation of string length.
    - **strcpy:** Copies the source string into the destination string including the terminating null byte ('\0').
        - Example: strcpy(dest, source);
    - **strcat:** Concatenates the source string to the end of the destination string.
        - Example: strcat(dest, source);
    - **strlen:** Returns the length of the string, not including the terminating null byte.
        - Example: size_t len = strlen(str);
    - **strncpy:** Similar to strcpy but includes a limit on the number of characters copied.
        - Example: strncpy(dest, source, num);
- **unistd.h:** Typically used in Unix/Linux environments for system calls like read, write, close, etc. (Not the primary focus for this guide, but crucial for system-level programming in Unix/Linux environments).
    - **fopen:** Opens a file.
    - **fclose:** Closes an opened file.
    - **fprintf:** Writes a formatted string to a file.
    - **fscanf:** Reads formatted input from a file.
    - **Format Specifiers:**
        - %d (integer) - decimal only
        - %i (integer) - hex, decimal, or octal works
        - %c (character)
        - %s (string)
        - %p (pointer address)

Variables:

Declaring, Assigning to, and Initializing Variables

- **Declaration:** To declare a variable is to inform the compiler about its name and the type of data it will hold. Declaration does not necessarily allocate storage space.
    - Example: int age;
- **Assignment:** Assignment involves setting a value to a variable after it has been declared.
    - Example: age = 30;
- **Initialization:** Initialization is the process of assigning a value to a variable at the time of declaration.
    - Example: int age = 30;

Types: char, int, double, and pointers

- **char:** Used to store a single character.
    - Example: char initial = 'A';
- **int:** Used to store integers.
    - Example: int count = 100;
- **double:** Used for floating-point numbers with double precision.

    - Example: double pi = 3.14159;
- **Pointers:** A pointer in C is a variable that stores the memory address of another variable. Pointers are declared by placing an asterisk (*) before the variable name.
    - Example: int *ptr = &count; where ptr is a pointer to an integer variable.

Implicit vs Explicit Casting of Types

- **Implicit Casting:** Automatic conversion of one data type to another when necessary. This happens according to the hierarchy of data types to avoid loss of information.
    - For example, in an operation involving an int and a double, the int is automatically converted to double.
    - Int to double
    - Char to int
    - Float to double
- **Explicit Casting:** This is when you manually convert one data type to another by specifying the new data type in parentheses before the variable or value. This is necessary when you need to convert a higher data type into a lower one, or when implicit conversion does not occur.
    - Example: double num = 9.8; int integerPart = (int)num;
    - Double to int
    - Double to char
    - Int to char

Type Compatibility of Argument Values and Parameters Variables

This compatibility determines how data is passed between functions and what operations are permissible on these data types.

- **Argument Values:** These are the actual values or expressions provided to a function when it is called. They represent the input that the function will work with.
- **Parameter Variables:** Also known as formal parameters, these are the variables listed by a function in its definition. They act as placeholders for the values that will be passed to the function at runtime.

Rules of Type Compatibility

- **Basic Compatibility Rule:** When a function is called, each argument value is compared with its corresponding parameter variable to ensure they are of compatible types. If the types are compatible, the argument value is passed to the parameter variable, possibly with implicit type conversion.
- **Implicit Type Conversion:** If the argument's type does not exactly match the parameter's type but is compatible, an implicit conversion (also known as "automatic type coercion") may occur.
    - For example, if a function parameter is of type double, but it is passed an int argument, the int value is automatically converted to double before the function executes.
- **Incompatibility and Explicit Casting:** When argument types are not compatible with parameter types, and implicit conversion is not possible, the code may not compile, or unexpected behavior may occur.

    - To resolve such issues, explicit casting can be used to convert an argument's type to match the parameter's type. This is done by using a type cast operator, e.g., (double)intValue.
- **Pointer Type Compatibility:** Pointer types require special attention. A pointer argument must match the type of the pointer parameter, including the const qualifier. However, a pointer to void can receive a pointer of any type. Strict type checking applies to function pointers, ensuring the argument types and return type of the function pointed to match expectations.
- **Array and Function Argument Compatibility:** When passing an array to a function, the array decays to a pointer to its first element, and the function parameter should be a pointer to the correct element type. For functions, passing a function name as an argument automatically converts it to a pointer to the function.

Dynamic Memory Management (HEAP) in C Programming

- Dynamic memory management in C is a powerful feature that allows programs to allocate memory at runtime from the heap segment.
- Unlike static memory (allocated at compile time) and stack memory (allocated at function call time), heap memory is managed through a set of standard library functions provided in stdlib.h.
- Understanding these operations is crucial for managing memory efficiently and avoiding memory leaks or corruption.

four fundamental dynamic memory management functions

- **malloc - Function**
    - **Purpose:** Allocates a specified number of bytes of uninitialized memory.
    - **Prototype:** void* malloc(size_t size)**;**
    - **Usage:**
        - **size:** The number of bytes to allocate.
        - Returns a pointer to the allocated memory, or NULL if the allocation fails.
    - **Example:**
        - int *arr = (int*)malloc(10 * sizeof(int));
        - if (arr == NULL) {
        - // Allocation failed
        - }
- **calloc - Function**
    - **Purpose:** Allocates memory for an array of elements of a certain size, and initializes all bytes to zero.
    - **Prototype:** void* calloc(size_t num, size_t size);
    - **Usage:**
        - **num:** Number of elements.
        - **size:** Size of each element.
        - Returns a pointer to the allocated and initialized memory, or NULL if the allocation fails.
    - **Example:**
        - int *arr = (int*)calloc(10, sizeof(int));

        - if (arr == NULL) {
        - // Allocation failed
        - }
- **realloc - Function**
    - **Purpose:** Changes the size of the previously allocated memory block without losing the old data.
    - **Prototype:** void* realloc(void* ptr, size_t newSize);
    - **Usage:**
        - **ptr:** Pointer to the previously allocated memory block. If NULL, realloc behaves like malloc.
        - **newSize:** New size in bytes. If newSize is 0, realloc frees the memory and returns NULL.
        - Returns a pointer to the newly allocated memory, or NULL if the allocation fails.
    - **Example:**
        - arr = (int*)realloc(arr, 20 * sizeof(int));
        - if (arr == NULL) {
        - // Allocation failed or was freed
        - }
- **free - Function**
    - **Purpose:** Deallocates the memory previously allocated by malloc, calloc, or realloc.
    - **Prototype:** void free(void* ptr);
    - **Usage:**
        - **ptr:** Pointer to the memory to be freed. If ptr is NULL, no operation is performed.
    - **Example:**
        - free(arr);
        - arr = NULL; // Good practice to avoid dangling pointers
    - **Best Practices:**
        - **Check for NULL:** Always check if the memory allocation functions return NULL, indicating allocation failure.
        - **Avoid Memory Leaks:** Ensure every allocated block is freed once it is no longer needed.
        - **Prevent Dangling Pointers:** Set pointers to NULL after freeing the memory they point to.
        - **Size Calculation:** Use sizeof operator to calculate the size of the elements being allocated.
        - **Realloc Usage:** Be cautious with realloc as it can move the memory block, potentially invalidating pointers to the old location.

Defining Functions

- **Naming:** Function names should be descriptive and follow the same naming conventions as variables. Typically, function names use lowerCamelCase or snake_case.
- Names must start with a letter (a-z, A-Z) or an underscore (_), followed by letters, digits (0-9), or underscores.
- **Parameters:** Parameters (also known as arguments) are specified in the function declaration and definition. They represent the input that the function operates on. Each parameter must have a type and can optionally have a name in the declaration.
    - **Syntax:** returnType functionName(parameterType1 parameterName1, parameterType2 parameterName2, ...)
- **Return Types:** The return type specifies the type of value a function is expected to return. If a function does not return a value, its return type should be void.
    - The type can be any valid C data type, including custom types like structs or unions.

Returning Values

- To return a value from a function, use the return statement followed by the value to be returned. The type of this value must match the function's declared return type.
- If the function's return type is void, it can either use a return statement with no value or not use one at all.
- **Example:**
    - int add(int a, int b) {
    - return a + b;
    - }

Multi-dimensional Array Parameters

When passing multi-dimensional arrays to functions, the parameter must be compatible with the layout of the arrays in memory. This compatibility is crucial for the correct interpretation and manipulation of array elements within the function.

- **Type Compatibility:** The type of the multi-dimensional array parameter must match the type of the array being passed.
    - For a two-dimensional array, this means the function parameter must specify the size of the second (and any higher) dimensions, while the first dimension can remain unspecified because it represents the array's size, which can vary.
- **Memory Layout:** Multi-dimensional arrays are stored in row-major order in memory, meaning the rows are stored in contiguous memory blocks. Understanding this layout is essential when working with array parameters, as it affects indexing and pointer arithmetic.
- **Function Parameter Declaration:** When declaring a function that takes a multi-dimensional array, you only need to specify the sizes of the second and subsequent dimensions.
    - **Example for a 2D array of integers where each row has 5 columns:**
        - void printMatrix(int matrix[][5], int rows) {
        - for (int i = 0; i < rows; i++) {
        - for (int j = 0; j < 5; j++) {
        - printf("%d ", matrix[i][j]);

```
        }
        printf("\n");
    }
}
```

- In this example, the first dimension size is left unspecified because it can vary, but the second dimension size is specified as 5, matching the layout of the arrays sent to the function.

POINTERS, ARRAYS, and STRUCTS in C Programming

STACK Allocated Array Structure (SAA)

- **1D Arrays:** A one-dimensional array on the stack is declared and initialized with a fixed size known at compile time.
  - For example, int arr[10]; allocates an array of 10 integers on the stack.
- **2D Arrays and Beyond:** Multidimensional arrays extend this concept. Memory layout for multidimensional arrays is contiguous, with row-major ordering.
  - For example, int arr[2][3]; creates a 2D array with 2 rows and 3 columns on the stack.

Address of vs. Dereferencing

- **Address of (&):** Retrieves the memory address of a variable.
- For instance, &x gives the address of variable x.
- **Dereferencing (*):** Accesses the value at a given memory address.
  - If ptr is a pointer to x, *ptr accesses the value of x.
    - int value = 10;
    - int* ptr = &value; // ptr holds the address of value
    - int valueFromPtr = *ptr; // Dereference ptr to get its pointed-to value (10)

Accessing Elements

- **Dereferencing and Indexing:** You can access elements of arrays through dereferencing and indexing.
  - For example, *(arr + i) is equivalent to arr[i]. This uses pointer arithmetic to access elements, where i is the index.
- **Address Arithmetic:** Useful for navigating arrays and structs by moving the pointer a certain number of elements forward or backward.
- **Reading/Writing to Char Arrays:** Reading or writing to elements of char arrays is direct, using indexing, as long as you stay within bounds. Char arrays often end with a null terminator (\0) to signify the end of a string.

Syntax for Declaring Array Variables

- **1D Arrays:** type arrayName[arraySize];
- **Multi-dimensional Arrays:** type arrayName[size1][size2];

Accessing and Modifying Elements

On STACK vs. HEAP: Whether on the stack or heap, accessing and modifying elements use the same syntax (array[index] for 1D, and array[i][j] for 2D). The key difference is in how memory for these arrays is allocated and freed.

- **Stack-allocated arrays:** Have automatic storage duration and are automatically freed when they go out of scope.

- **heap-allocated arrays:** Memory is allocated dynamically (using malloc, calloc) and must be manually freed using free. This allows for dynamic sizing but requires careful memory management to avoid leaks.

Storing and Passing Along Array Sizes

C does not inherently store size information for arrays. Thus, when passing arrays to functions or working with them dynamically, explicitly passing the size is necessary for safe access and manipulation.

Scalars, Arrays, and Structs on the Stack

- **Scalars on the Stack:** Simple data types (e.g., char, int, double) declared within a function are allocated on the stack.
- **Arrays on the Stack:** Declared with a fixed size within a function or as globals (not in stack).
- **Structs on the Stack:** Similar to arrays, structs declared within a function are allocated on the stack.
    - Static example:
        - struct Person {
        - char name[50];
        - int age;
        - float salary;
        - };
        - struct Person person = {.name = "John Doe", .age = 30, .salary = 50000.0};
        - // struct Person person = {"John Doe", 30, 50000.0};
    - If not on stack and dynamic example:
        - struct Person {
        - char name[50];
        - int age;
        - float salary;
        - };
        - // Dynamic initialization
        - struct Person person;
        - scanf("%s %d %f", person.name, &person.age, &person.salary);

- **Pointers to Scalars, Arrays, Structs:** Can point to memory either on the stack or heap, determined by where the memory was allocated.

Dot (.) and Arrow (->) Operators

- **Dot Operator:** Used to access members of a struct when you have the struct itself.
    - For example, structVar.member.
- **Arrow Operator:** Used to access members of a struct through a pointer to the struct.
    - For example, ptrToStruct->member.

Passing Pointers, int, char, double to/from functions

- When you pass pointers to basic data types (int, char, double) to functions, you're giving the function the ability to directly modify the original variables' values. This technique is

useful when you need a function to update or manipulate data stored in variables outside its scope.

```
void updateValue(int *ptr) {
    *ptr = 10; // Directly modifies the variable that ptr points to
}
```

- **Stack**

```
void modifyInt(int *x) {
    *x = 100; // Modify value in stack memory
}
int main() {
    int a = 5; // `a` is stored on the stack
    modifyInt(&a); // Pass address of `a` to function
    // `a` is now 100
}
```

- **Heap**

```
void modifyDouble(double *x) {
    *x = 3.14159; // Modify value in heap memory
}
int main() {
    double *b = (double*)malloc(sizeof(double)); // Allocate on the heap
    *b = 2.71828;
    modifyDouble(b); // Pass pointer to `b` to function
    // *b is now 3.14159
    free(b); // Clean up
}
```

<u>Passing Single-Dimensional Arrays (SAA) to/from Functions</u>

- Arrays are passed to functions as pointers to their first element, effectively allowing the function to access and modify the array's contents directly. This passing method is efficient because it doesn't involve copying the entire array's data.

```
void fillArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = i * 2;
    }
}
```

- **Stack**

```
void fillArrayStack(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = i;
    }
}
int main() {
    int arrStack[5];
    fillArrayStack(arrStack, 5); // Modify stack-allocated array
```

```
    }
```

- **Heap**

```
    void fillArrayHeap(int *arr, int size) {
        for (int i = 0; i < size; ++i) {
            arr[i] = i;
        }
    }
    int main() {
        int *arrHeap = (int*)malloc(5 * sizeof(int));
        fillArrayHeap(arrHeap, 5); // Modify heap-allocated array
        free(arrHeap);
    }
```

Watch video on:

Passing Arrays of Pointers to/from Functions

- Passing an array of pointers to a function allows it to manipulate multiple pieces of data, such as strings or arrays of dynamically allocated memory. This method is especially useful for handling complex data structures or for operations on an array of strings.

```
    void resetPointers(int *arr[], int size) {
        for (int i = 0; i < size; i++) {
            arr[i] = NULL;
        }
    }
```

- **Stack**

```
    void modifyPointerValues(int *arr[], int size) {
        for (int i = 0; i < size; ++i) {
            *arr[i] = i * 10;
        }
    }
    int main() {
        int x = 5, y = 6, z = 7;
        int *arrStack[] = {&x, &y, &z}; // Array of pointers on the stack
        modifyPointerValues(arrStack, 3); // Modify values pointed to by pointers
    }
```

- **Heap**

```
    void allocateAndAssign(int **arr, int size) {
        for (int i = 0; i < size; ++i) {
            arr[i] = (int*)malloc(sizeof(int));
            *arr[i] = i * 10;
        }
    }
    int main() {
        int *arrHeap[3]; // Array of pointers on the stack
        allocateAndAssign(arrHeap, 3); // Allocate and assign heap memory
        // Cleanup
```

```
    for (int i = 0; i < 3; ++i) free(arrHeap[i]);
}
```

Passing Arrays of Arrays to/from Functions

- When passing multi-dimensional arrays to functions, the size of all dimensions except the first must be known to the function. This allows the function to correctly interpret the memory layout of the array, enabling direct access and modification of its elements.

```
void modify2DArray(int arr[][3], int rows) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 3; j++) {
            arr[i][j] = i + j;
        }
    }
}
```

- **Stack**

```
void modify2DArrayStack(int arr[][3], int rows) {
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < 3; ++j) {
            arr[i][j] = i + j;
        }
    }
}
int main() {
    int arr2DStack[2][3]; // 2D array on the stack
    modify2DArrayStack(arr2DStack, 2); // Modify stack-allocated 2D array
}
```

- **Heap**

```
void modify2DArrayHeap(int **arr, int rows, int cols) {
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            arr[i][j] = i * j;
        }
    }
}
int main() {
    int *arr2DHeap[2]; // Array of int pointers on the stack
    for (int i = 0; i < 2; ++i) {
        arr2DHeap[i] = (int*)malloc(3 * sizeof(int)); // Allocate 2D array on heap
    }
    modify2DArrayHeap(arr2DHeap, 2, 3); // Modify heap-allocated 2D array
    // Cleanup
    for (int i = 0; i < 2; ++i) free(arr2DHeap[i]);
}
```

Passing Structs to/from Functions

- Structs can be passed to functions either by value or by reference (using pointers). Passing by value copies the entire struct, which can be inefficient for large structs. Passing by reference allows the function to modify the original struct directly without copying.
    - **By Value:** Safe but may be inefficient for large structs.
        - void processStructByValue(MyStruct s) {
        - s.id = 1; // Only modifies the copy
        - }
    - **By Reference:** Efficient and allows direct modification.
        - void processStructByReference(MyStruct *s) {
        - s->id = 1; // Modifies the original struct
        - }
- **Stack**
    - typedef struct {
    - int id;
    - double value;
    - } MyStruct;
    - void modifyStructStack(MyStruct s) {
    - s.id = 1; // Changes here won't affect the original struct in main
    - }
    - int main() {
    - MyStruct sStack; // Struct on the stack
    - modifyStructStack(sStack); // Pass by value (copy)
    - }
- **Heap**
    - void modifyStructHeap(MyStruct *s) {
    - s->id = 1; // Changes will affect the original struct
    - }
    - int main() {
    - MyStruct *sHeap = (MyStruct*)malloc(sizeof(MyStruct)); // Struct on the heap
    - modifyStructHeap(sHeap); // Pass by reference (pointer)
    - free(sHeap); // Clean up
    - }

<u>Passing Pointers to Structs to/from Functions</u>

- Passing a pointer to a struct to a function is a common practice for manipulating or accessing the data within the struct. This method passes the address of the struct, enabling the function to modify the struct's fields directly, which is especially useful for large structs or when maintaining state across function calls.
    - Example: Updating struct fields.
        - void updateStruct(MyStruct *s) {
        - s->value = 100; // Directly updates the original struct
        - }
- **Stack**

```
void initStructOnStack(MyStruct *s) {
    s->id = 2;
    s->value = 4.2;
}
int main() {
    MyStruct s; // Struct on the stack
    initStructOnStack(&s); // Pass address of stack-allocated struct
}
```

- **Heap**

```
void initStructOnHeap(MyStruct *s) {
    s->id = 3;
    s->value = 6.4;
}
int main() {
    MyStruct *s = (MyStruct*)malloc(sizeof(MyStruct)); // Struct on the heap
    initStructOnHeap(s); // Pass pointer to heap-allocated struct
    free(s); // Clean up
}
```

Virtual Address Space (VAS):

- The Virtual Address Space (VAS) of a process in operating systems like Linux and Windows is a concept used to abstract physical memory, allowing a program to view memory as a contiguous and linear address space.
- This space is divided into several segments and sections, each with its own purpose and characteristics.
- Understanding these segments and how they relate to each other is crucial for programming, especially in languages like C that allow low-level memory manipulation.

Memory Segments: CODE, DATA, HEAP, STACK, KERNEL

- **CODE:** This segment, also known as the text segment, contains the executable code of a program. It is typically read-only to prevent the program from accidentally modifying its instructions.
    - Machine code - .txt
    - String literals - .rodata
- **DATA:** This segment stores global and static (and static local) variables used by the program. It's divided into initialized data and uninitialized data.
    - .data - variables initialized to non 0 values
    - .bss - variables not initialized or initialized to 0
- **HEAP:** Dynamically allocated memory (using malloc, calloc, realloc, new, etc.) during runtime is allocated here. The heap grows upward towards higher memory addresses as more memory is allocated.
- **STACK:** Used for local variables, function parameters, return addresses, and control flow. The stack grows downward towards lower memory addresses as new functions are called and local variables are declared.

- **KERNEL:** Reserved for the operating system's kernel and its operations. User programs cannot directly access this space in a protected mode operating system.

Memory Sectoins: .text, .rodata, .data, .bss

- **.text:** Contains the executable instructions of a program. Corresponds to the CODE segment.
- **.rodata:** Read-only data, such as constant strings and literals. It's part of the DATA segment but is marked read-only.
- **.data:** Stores initialized global and static variables.
- **.bss (Block Started by Symbol):** Contains uninitialized global and static variables. The .bss section occupies no actual space in the executable file; it's allocated by the OS when the program starts.

Where memory segments are located relative to each other and Kernel

- In a simplified view of the virtual address space: (little endian order - least significant is smallest address - which is code first)
    - **Kernel:** Occupies the highest addresses. Access is restricted to the operating system.
    - **Stack:** Just below the kernel space, growing downwards.
    - **Heap:** Located after the BSS and data segments, growing upwards.
    - **Data:** Follow the text segment.
    - **Code:** Placed at lower memory addresses.

When and how Stack and Heap grow and shrink

- The stack and heap are spaced apart in the address space to minimize the chance of collision, but excessive allocation on either can lead to a "stack overflow" or heap exhaustion, respectively.
- **Stack Growth:** The stack grows downwards (towards lower memory addresses) as functions call other functions and as local variables are declared. Each function call creates a new frame on the stack for its parameters and local variables.
    - Modern operating systems implement a stack guard to detect and prevent stack overflow by placing a guard page below the stack; if the stack grows into this page, the program is terminated.
    -
- **Heap Growth:** The heap grows upwards (towards higher memory addresses) as dynamic memory allocations occur. When more memory is allocated using functions like malloc, the heap expands towards the stack.
    - The heap, conversely, will fail to allocate more memory and return NULL pointers if it grows too large and cannot be accommodated.