Notes on C programming language and GCC

Chrys

August 23, 2023

Abstract

Taking personal notes while studying the following resources:

1. The C programming Language – Brian W. Kernighan and Dennis M. Ritchie.
2. C for technical interview - online course
3. An introduction to GCC – Brian Gough

1 Programming in C

Immutable and Mutable Data Types in C

In C, string literals are immutable. Once a string literal is defined, its contents cannot be changed. However, not many other data types in C can be considered strictly immutable like string literals. Most data types in C can be modified after their initial assignment.

- Array Elements The elements of an array can be modified after initialization. For example, if you have an array `int numbers[5];`, you can modify the individual elements like `numbers[0] = 42;`.
- Structures Structures in C can contain various data types, and their members can be modified independently after the structure is created.
- Constants You can use the `const` keyword to declare constant variables that should not be modified after initialization. For example, `const int x = 10;` declares `x` as a constant that cannot be changed.
- Pointers While the values pointed to by pointers can be changed, the pointers themselves can be made constant using the `const` keyword. This means that you cannot modify the memory location the pointer points to.
- Enumerations Enumerations define a set of constant values, and once they are defined, their values cannot be changed within the program.
- Function Parameters Function parameters are essentially local variables that hold values passed to the function. While you can modify their values within the function, it won't affect the original values passed to the function.

1.1 Allocation

1.2 Stack and Heap Memory allocation

In C programming, stack and heap are two different areas of memory used for memory allocation and management.

1.2.1 Stack Allocation:

The stack is a region of memory used for storing local variables and function call information. It operates in a Last-In-First-Out (LIFO) manner, meaning the last item pushed onto the stack is the first one to be popped off. Function parameters, return addresses, and local variables are typically stored on the stack.

When a function is called, its local variables and parameters are pushed onto the stack, and when the function returns, the stack pointer is adjusted to "pop" these variables, effectively deallocating their memory. The stack allocation and deallocation are usually fast since it involves simple pointer manipulation.

However, the stack size is usually limited, and excessive stack usage can lead to a stack overflow, causing the program to crash.

1.2.2 Heap Allocation:

The heap is another area of memory used for dynamic memory allocation. It is a larger and more flexible memory pool than the stack. Memory allocated on the heap doesn't have a specific order, and it's the programmer's responsibility to manage memory allocation and deallocation manually.

Heap memory is typically used for data structures like linked lists, trees, dynamic arrays, and objects with a variable lifetime that extends beyond the scope of a single function. In C, you can allocate memory on the heap using functions like "malloc()", "calloc()", and "realloc()", and you need to free the allocated memory using the "free()" function when it's no longer needed to prevent memory leaks.

Heap allocation can be more flexible than stack allocation, but it's also more error-prone if not managed carefully. Failing to free memory on the heap can lead to memory leaks, where memory is not reclaimed after it's no longer needed, causing the program's memory usage to grow over time.

1.2.3 Malloc, Calloc, and Realloc

Aspect	`malloc()`	`calloc()`
Memory Initialization	Uninitialized (garbage values)	Initialized (zero)
Arguments	Size of memory to allocate	Num. of elements, size of each
Return Value	Pointer to allocated memory	Pointer to allocated memory
Error Handling	Returns `NULL` on failure	Returns `NULL` on failure
Usage	No immediate initialization	Zero initialization

Table 1: Comparison between `malloc()` and `calloc()`

The "realloc()" function in C is used to dynamically resize memory that was previously allocated using functions like "malloc()" or "calloc()". It takes two arguments: a pointer to the previously allocated memory block and the new size that you want to allocate. There are three possible scenarios when using "realloc()":

1. If there is enough adjacent free space after the existing block, "realloc()" may simply extend the block's size without moving it, and the original pointer is returned.

2. If there isn't enough contiguous free space, "realloc()" may allocate a new block elsewhere, copy the contents of the old block to the new one, and free the old block. The pointer to the new block is then returned.

3. If "realloc()" fails to allocate memory, it returns "NULL", leaving the original block unchanged.

   It's important to note that the contents of the memory block may be copied to a new location during resizing, so any pointers to the old block should not be used after calling "realloc()". Additionally, "realloc()" does not guarantee to initialize the newly allocated memory; you need to explicitly initialize it if necessary.