Memory Management

Learn how the C language handles memory.

In this article, we will explore how the C programming language manages memory. This article is divided into the following sections:

- Introduction
- Memory maps
- Variables in memory
- Memory access
- Memory allocation
- Conclusion

Introduction

A key resource for the execution of a program is memory. When a program is running, memory is used to store relevant variables and other data. Memory in a computer is a finite and expensive resource. If a process occupies too much memory, it will lead to poor program performance; therefore it is important to be conservative with memory, and only use as little as possible.

Programs are created by writing code in programming languages. Each language has its own rules regarding how to work with memory. The C language, being a low-level language one step above assembly, offers a programmer access to memory with very little restrictions; other languages such as Java and Python do not allow the programmer to directly interact with memory. This has its advantages and disadvantages as we will see in the remainder of this article.

Memory Maps

In a computer, the physical memory is a piece of hardware with complex electrical circuits on it that are used to implement the function of "remembering something." This is known as RAM (Random Access Memory) or ROM (Read-only Memory). If you've ever taken apart (or put together) a computer, you have likely seen these pieces of hardware. They are the memory stick for RAM and your hard disk for ROM. The programmer, not being a computer engineer, is not concerned with how these pieces of hardware work, as long as they do! To the programmer, the memory is represented as a *memory map* like so:

RAM is represented in this way. It is very likely that you know how much RAM your computer (or phone, or tablet) has. This number is usually quoted as some amount of gigabytes (GB). The fundamental unit of memory is a *byte*. Each byte of memory has its own address that is used to store data. This address is usually enumerated using the hexadecimal number system. The addresses can be seen in the middle column of the table (it is in decimal because it is intuitive for beginners). C allows direct access to these addresses.

Variables in Memory

When writing code, programmers frequently work with variables to store and manipulate data. A variable occupies a contiguous block of bytes in memory with the size of the block depending on the data type of the variable.
An `int` variable (on a personal computer) typically occupies four bytes of memory. A floating-point variable occupies eight bytes of memory which is why it is known as a `double`. A character (`char`) occupies one byte of memory. The amount of memory occupied by a variable can be found by using the `sizeof()` function in C. The address of the variable in memory is the address of the first byte occupied.

Memory Access

A running program is allocated a section of physical memory by the underlying operating system (if there is one) to store relevant data. Because it is considered a "low-level" language, C allows you to directly interact with this memory through the use of what is called a *pointer*. A pointer is a special type of integer variable that stores the memory address of a variable that the pointer is "pointing" to. To obtain the address of a variable, C provides a special operator called the *reference operator*. To read or write data to an address stored in a pointer, C provides an operator called the *dereference operator*.

Allowing such unrestricted access requires great care. Errors involving memory access with pointers are sometimes silent as they don't cause compile-time or run-time errors. Erroneous memory access (for example, you accessed and changed data at the wrong address) causes the program to produce incorrect results and maybe even behave unpredictably. These types of errors are incredibly difficult to find and correct!

Memory Allocation

Program memory is organized into two main categories: the *stack* and the *heap*. A stack is a section of memory that is highly ordered and data stored on the stack are only available within a certain scope (you'll learn about

this in the lesson on functions). In contrast, the heap is not as ordered as a stack. When you create a variable in the usual way, you are *statically* allocating memory and it is stored on the stack. That memory will be released by the program when the variable is no longer needed.

The heap allows you to reserve as much available memory as you want and that memory will remain available until you explicitly release it. This is known as *dynamically* allocating memory. Forgetting to release dynamically allocated memory will cause a *memory-leak* leading to poor program performance.

C provides four special functions for you to dynamically allocate (and release) memory, provided it is available. They are:

- `malloc()`
- `calloc()`
- `realloc()`
- `free()`

These functions are stored in the `stdlib` library. To use them, you must import this library.

You will need a pointer to work with the memory provided by any of these functions.

Function	Use Case
`malloc()`	Use this function to reserve as many bytes as you want on the heap
`calloc()`	Use this function to reserve memory for some number of `int`s, `double`s, or any other da
`realloc()`	Use this function to expand or contract a block of reserved memory (reserved by eith
`free()`	Use this function to release previously allocated memory.

The most important of these functions is `free()`. It is necessary to release memory that is no longer needed to maintain efficient program performance.

Conclusion

C is one of the most powerful and versatile languages available. Because of its memory management style, it is used to create programs for many industries. However, when working with memory in C, it is imperative to be very careful. Memory is one of the most precious resources a program needs; therefore it should be used conservatively. Errors resulting from incorrect memory access or leaks are detrimental to the operation of a program and are difficult to find and correct. As the age-old saying goes: "with great power, comes great responsibility." Be careful when directly working with memory in C!