**Chapter 14: Interlude: Memory API**

We will talk about memory allocation interfaces in UNIX systems.

**14.1 Types of Memory**

In a running C program, there are two types of memory that are allocated. The first is called the **stack** memory. Allocations and deallocations of it are managed implicitly by the compiler for you, the programmer. For this reason, it is sometimes called **automatic** memory.

When we declare a piece of memory, like “*int x;*” The compiler will do everything: make sure to make space on the stack and deallocate the memory.

This needs for a long-lived memory, named **heap** memory where all allocations and deallocations are explicitly handled by us, the programmer.

Text

Description automatically generated with low confidence

In the above declaration of heap memory, the compiler also allocates for a pointer in the stack.

**14.2 The malloc() call**

The malloc() call is simple as we just need to pass it a size asking for some room in the heap. If it succeeds and gives you back a pointer to the newly-allocated space, else if it fails and returns NULL. TO use this, include the header file stdlib.h. Example:



We must also be careful with string. When declaring space for a string, use the following idiom: malloc(strlen(s) + 1), which gets the length of the string using the function strlen(), and adds 1 to it in order to make room for the end-of-string character.

Malloc returns a pointer to type void. This is just for the programmer to decide what to do with is using **cast**. In the above example, we use (double \*).

**14.3 The free() Call**

To free memory that is no longer in use in the heap, programmers simply call free(), which takes one argument, which is the pointer returned by malloc().

**14.4 Common Errors**

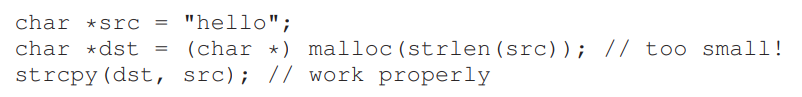
Newer languages now support for **automatic memory management**. In such languages, we never have to free space. Instead, a garbage collector runs and figures out what memory you no longer have references to and frees it for you.

**Forgetting to allocate memory:**

Many routines expect memory to be allocated before we call them. For example, when we use strcpy(dst, src), the dst pointer must be allocated first. On the other hand, if we use strdup(dst, src), we would not have to use it.

**Not allocating enough memory:**

When we allocate not enough memory, we call it buffer overflow. For example:

****

Depending on how malloc is implemented and many other details, this program will often run seemingly correctly.

**Forgetting to initialize Allocated Memory**

This error occurs when you call malloc() properly, but forget to fill in some values into your newly-allocated data type. This error is called **uninitialized read**.

**Forgetting To Free Memory**

This will cause **memory leak** as we forget to free memory. In long-running applications or systems, this is a huge problem as slowly leaking memory eventually leads one to run out of memory, at which point a restart is required. Thus, in general, when you are done with a chunk of memory, you should make sure to free it.

*If you still have a reference to some chunk of memory, no garbage collector will ever free it, and thus memory leaks remain a problem even in more modern languages.*

**Freeing Memory Before You Are Done With It**

This mistake is called dangling pointer. The subsequent use can crash the program, or overwrite valid memory

**Freeing Memory Repeatedly**

This mistake is called double free. The memory-allocation library might get confused and do all sorts of weird things; crashes are a common outcome.

**Calling free() Incorrectly**

When we pass in some other value, bad things can (and do) happen. Thus, such invalid frees are dangerous and of course should also be avoided.

**14.5 Underlying OS Support**

Malloc and free are not system calls. They are library calls.

One such system call is called brk, which is used to change the location of the program’s break: the location of the end of the heap. It takes one argument (the address of the new break), and thus either increases or decreases the size of the heap based on whether the new break is larger or smaller than the current break.

Mmap() creates an anonymous memory region within our program. The region that is not associated with nny particular file but rather with swap space.

**14.6 Other Calls**

calloc() allocates memory and also zeroes it before returning.

realloc() allows us to add something to allocated space by making a new larger region of memory, copying the old region to it and returning the pointer to the new region.