In UNIX/C programs, understanding how to allocate and manage memory is critical in building robust and reliable software. What interfaces are commonly used? What mistakes should be avoided?

Types of Memory

- In running a C program, there are two types of memory that are allocated. The first is called stack memory, and allocations and deallocations of it are managed implicitly by the compiler for you, the programmer; for this reason it is sometimes called automatic memory.
    - Declaring memory on the stack in C:
        - Void func(){
            int x; // declares an integer on the stack
          }
- The compiler does the rest, making sure to make space on the stack when you call into func(). When you return from the function, the compiler deallocates the memory for you;
- It is this need for long-lived memory that gets us to the second type of memory, called heap memory, where all allocations and deallocations are explicitly handled by the programmer
    - void func(){
        int *x = (int *) malloc(sizeof(int));
      }
- both stack and heap allocation occur on this line: first the compiler knows to make room for a pointer to an integer when it sees your declaration of said pointer (int *x);
    - Subsequently, when the program calls malloc(), it requests space for an integer on the heap; the routine returns the address of such an integer (upon success, or NULL on failure), which is then stored on the stack for use by the program.

The malloc() Call

- The malloc() call is simple: you pass it a size asking for some room on the heap, and it either succeeds and gives you back a pointer to the newly-allocated space, or fails and returns NULL.
    - Ex:
        - #include <stdlib.h>
          void *malloc (size_t size);
- The single parameter malloc() takes is of type size t which simply describes how many bytes you need.
- For example, to allocate space for a double-precision floating point value, you simply do this:
    - double *d = (double *) malloc(sizeof(double));
- This invocation of malloc() uses the sizeof() operator to request the right amount of space;
    - in C, this is generally thought of as a compile-time operator, meaning that the actual size is known at compile time and thus a number (in this case, 8, for a double) is substituted as the argument to malloc().

- You can also pass in the name of a variable (and not just a type) to sizeof() but in some cases you may not get the desired results.
- For ex:
    - int *x = malloc(10 * sizeof(int));
      printf("%d\n", sizeof(x));
    - In the first line, we've declared space for an array of 10 integers
    - when we use sizeof() in the next line, it returns a small value, such as 4 (on 32-bit machines) or 8 (on 64-bit machines).
    - The reason is that in this case, sizeof() thinks we are simply asking how big a pointer to an integer is, not how much memory we have dynamically allocated
- However, sometimes sizeof() does work as you might expect:
    - int x[10];
      printf("%d\n", sizeof(x));
    - In this case, there is enough static information for the compiler to know that 40 bytes have been allocated.
- 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
- malloc() also returns a pointer to type void.
    - Doing so is just the way in C to pass back an address and let the programmer decide what to do with it. The programmer further helps out by using what is called a cast; in our example above, the programmer casts the return type of malloc() to a pointer to a double.
    - Casting doesn't really accomplish anything, but by casting the result of malloc(), the programmer is just giving some reassurance; the cast is not needed for the correctness.

The Free() Call

- To free heap memory that is no longer in use, programmers simply call free():
    - int *x = malloc(10 * sizeof(int));
      … free(x);
    - The routine takes one argument, a pointer returned by malloc().

Common Errors

- There are a number of common errors that arise in the use of malloc() and free().
- Correct memory management has been such a problem, in fact, that many newer languages have support for automatic memory management.
- In such languages, while you call something akin to malloc() to allocate memory (usually new or something similar to allocate a new object), you never have to call something to free space; rather, 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 you call them. For example, the routine strcpy(dst, src) copies a string from a source pointer to a destination pointer. However, if you are not careful, you might do this:
        - char *src = "hello";
          char *dst; // oops! //Unallocated

strcpy(dst, src); // segfault and die
        - If we run this code, we might run into a segmentation fault; segmentation faults occur when a program attempts to access memory that it does not have permission to access or when it tries to access memory that does not exist.
        - This is what the proper code would look like:
        - char *src = "hello";
          char *dst = (char *) malloc(strlen(src) + 1);
          strcpy(dst, src); // work properly
- Not Allocating Enough Memory:
    - A related error is not allocating enough memory, sometimes called a buffer overflow.
    - char *src = "hello";
      char *dst = (char *) malloc(strlen(src)); // too small!
      strcpy(dst, src); // work properly
    - This program will often run seemingly correctly.
        - In some cases, when the string copy executes, it writes one byte too far past the end of the allocated space, but in some cases this is harmless, perhaps overwriting a variable that isn't used anymore.
        - In some cases, these overflows can be incredibly harmful, and in fact are the source of many security vulnerabilities in systems
- Forgetting to Initialize Allocated Memory
    - With this error, you call malloc() properly, but forget to fill in some values into your newly-allocated data type.
    - If you do forget, your program will eventually encounter an uninitialized read, where it reads from the heap some data of unknown value.
    - Who knows what might be in there? If you're lucky, some value such that the program still works (e.g., zero). If you're not lucky, something random and harmful.
- Forgetting To Free Memory
    - Another common error is known as a memory leak, and it occurs when you forget to free memory.
    - In long-running applications or systems (such as the OS itself), this is a huge problem, as slowly leaking memory eventually leads one to run out of memory, at which point a restart is required.
    - when you are done with a chunk of memory, you should make sure to free it.
    - using a garbage-collected language doesn't help here: 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.
    - it is good to get in the habit of freeing each and every byte you explicitly allocate.
    - When you write a short-lived program, you might allocate some space using malloc(). The program runs and is about to complete: is there a need to call free() a bunch of times just before exiting? While it seems wrong not to, no memory will

be “lost” in any real sense. The reason is simple: there are really two levels of memory management in the system.
    - The first level of memory management is performed by the OS, which hands out memory to processes when they run, and takes it back when processes exit (or otherwise die).
    - The second level of management is within each process, for example within the heap when you call malloc() and free(). Even if you fail to call free() (and thus leak memory in the heap), the operating system will reclaim all the memory of the process (including those pages for code, stack, and, as relevant here, heap) when the program is finished running.
        - No matter what the state of your heap in your address space, the OS takes back all of those pages when the process dies, thus ensuring that no memory is lost despite the fact that you didn’t free it.
    - For short-lived programs, leaking memory often does not cause any operational problems (though it may be considered poor form).
        - When you write a long-running server (such as a web server or database management system, which never exit), leaked memory is a much bigger issue, and will eventually lead to a crash when the application runs out of memory
- Freeing Memory Before You Are Done With It
    - Sometimes a program will free memory before it is finished using it; such a mistake is called a dangling pointer
    - The subsequent use can crash the program, or overwrite valid memory (e.g., you called free(), but then called malloc() again to allocate something else, which then recycles the errantly-freed memory).
- Freeing Memory Repeatedly
    - Programs also sometimes free memory more than once; this is known as the double free.
- Calling free() Incorrectly
    - free() expects you only to pass to it one of the pointers you received from malloc() earlier.
    - When you pass on some other value, bad things can (and do) happen. Thus, such invalid frees are dangerous and should also be avoided.

Underlying OS Support

- we haven’t been talking about system calls when discussing malloc() and free(); The reason for this is simple:
    - they are not system calls, but rather library calls. Thus the malloc library manages space within your virtual address space, but itself is built on top of some system calls which call into the OS to ask for more memory or release some back to the system.
- 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.

- you should never directly call either brk or sbrk. They are used by the memory-allocation library; if you try to use them, you will likely make something go (horribly) wrong. Stick to malloc() and free() instead
- you can also obtain memory from the operating system via the mmap() call. By passing in the correct arguments, mmap() can create an anonymous memory region within your program – a region which is not associated with any particular file but rather with swap space, which can then also be treated like a heap and managed as such.

Other Calls

- calloc() allocates memory and also zeroes it before returning; this prevents some errors where you assume that memory is zeroed and forget to initialize it yourself
- The routine realloc() can also be useful, when you've allocated space for something (say, an array), and then need to add something to it: realloc() makes a new larger region of memory, copies the old region into it, and returns the pointer to the new region.