Lecture 06 - Allocation and memory safety (last updated: 11/03/22)

Pointers can point to more than just other variables: they may point to anywhere in the address space, and allow programs to manipulate data beyond what is in scope. However, their power can also be misused, and is the source of many bugs and vulnerabilities in C programs.

## Pointer values and NULL pointers

Pointer values are just numbers that "happen" to refer to a memory address. The specific value doesn't usually matter, with the exception of the zero value, also called a "NULL pointer". For example:

```
#define NULL (void *) 0   // provided in the C standard library

int *pi = 0;
double *pd = NULL;
```

Dereferencing a NULL pointer is considered undefined behavior, and usually leads to a type of runtime error known as a "segmentation fault" (colloquially, a "segfault"). For example, running this program:

```
// segfault.c
int main(void) {
    int *i = NULL;
    return *i;            // dereferencing a NULL pointer
}
```

will lead to this error message:

```
$ ./segfault
Segmentation fault (core dumped)
```

Don't confuse a NULL pointer with a pointer to a variable that holds 0:

```
char c = 0;
char *p = &c;             // pointer to zero
char *n = 0;              // NULL pointer

if (p) {
    // reached
}

if (n) {
    // not reached
}

if (*p) {
    // not reached
}

if (*n) { // segfault!
    // not reached
}
```

NULL pointers are usually used to signify the lack of something to point to, i.e., "this pointer doesn't point to anything." For example, a function that

usually returns a pointer to some useful data may return a NULL pointer to indicate an error.

## Dangling pointers

Local and global variables are easy to use, but they also are also limited:

- global variables live for the entire duration of the running program, but the amount of memory we allocate for them is constant and cannot grow.
- local variables are newly allocated every time we call a function, but their lifetime is limited to the duration of that function call.

To emphasize this second point, consider a function that tries to "allocate" an integer and return a pointer to it:

```
int *alloc_int(void) {
    int i = 0;
    return &i;
}

int *p = alloc_int();     // dangling pointer!
```

After alloc_int() returns, p points to where alloc_int()'s stack frame used to live on the stack, but that stack frame is no longer in use! We call p a "dangling pointer," because it points to an address that should no longer be pointed to. If the program calls another function, p may end up pointing to somewhere unpredictable in that other function's stack frame.

## Heap allocation

To overcome the complementary limitations of local and global variables, we allocate memory from the "heap," using malloc() (short for "Memory ALLOCate"):

```
int *malloc_int(void) {
    int *i = malloc(sizeof(int));

    if (i == NULL) {
        perror("malloc failed");
        exit(1);
    }

    *i = 0;               // initialize heap-allocated memory to 0
    return i;
}

int *p = malloc_int();    // no longer a dangling pointer!
```

The argument to malloc() is the amount of memory we are requesting, in bytes. malloc() will return a pointer to the beginning of a contiguous range of memory of at least that size (or NULL if it encountered some error).

Unlike with global variables, we determine the amount of memory we want to heap-allocate at runtime by computing the argument to malloc(). And unlike stack memory, heap memory allocated using malloc() is persistent: it will remain in the heap for as long as we need.

However, after we are done using heap memory, we must free() it:

```
free(p);
```

If we forget to free heap-allocated memory, it is considered "leaked."

Some safety tips about using the heap:

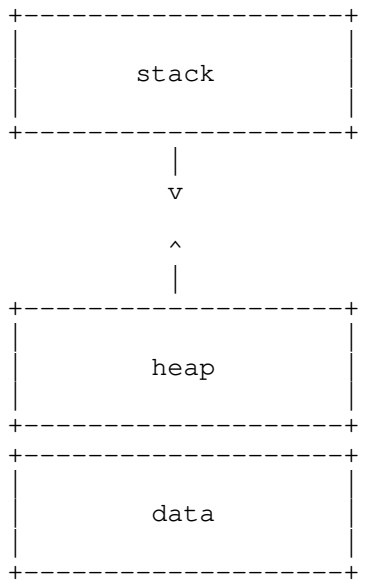
- Using memory beyond what is allocated (e.g., int *p = malloc(1);) is a memory error.
- Reading from uninitialized memory (i.e., has not been written to since being returned by malloc()) is a memory error.
- Using a pointer after it has been free()d is a memory error.
- free()ing a pointer twice is a memory error.
- Forgetting to free() a pointer you obtained from malloc() is NOT a memory error; it is a memory leak.

The difference is that a memory error can cause your program to crash, or worse, behave unpredictably. A memory leak will cause your program to occupy more memory than it should, but is not always a bad thing. That being said, you are required to write leak-free programs for this course, since it is considered good practice to always free() what you malloc().

Note that malloc() and free() are NOT built into C; they are just regular functions provided by the C standard library, like printf() and exit().

## Heap layout

The heap lives right above the data section in memory, and grows upward, toward the stack:




The same caveats about memory diagrams apply here: memory regions are not depicted to scale. In particular:

- the space between the heap and the stack is still extremely large
- the heap is typically much larger than the stack (or at least allowed to grow much larger than the stack)

Unlike the stack, memory allocated from the heap does not need to be freed in LIFO order. The internal organization of the heap is implementation-defined, very complex, and beyond the scope of this course. However, you should know that:

- where possible, malloc() might reuse previously freed space
- malloc() might not grow the heap, if it is able to reuse space
- small, non-contiguous chunks of free space in the heap are known as "external fragmentation," and prevent malloc() from reusing that free space for large size requests

## Finding memory errors and leaks with Valgrind

Valgrind is a memory multitool that can help you find memory errors and leaks. For example, if you normally run your program like this:

```
./my-program some args
```

You can run it with Valgrind like this:

```
valgrind --leak-check=full ./my-program some args
```

Valgrind will execute your program for you and report any memory errors and leaks it encounters. If you compiled your program with the -g flag, Valgrind will tell you which file and line number those errors or leaks come from.

Note that if your program expects your input (e.g., if it prompts your user to type something in), you should still interact with your program as you normally would while testing it.