Lecture 07 - Heap (last updated: 2023-10-01)

Heap allocation

Recall that an automatic variable defined in a function will go out of scope when the function returns, so if the function returns a pointer to it, the pointer will be a "dangling pointer" because it points to an address that should no longer be pointed to:

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

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

We can avoid returning a dangling pointer by allocating 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 that size (or NULL if it encountered some error).

This means we can also allocate an array of integers rather than a single int:

```
int *malloc_int_array(int n) {
    int *a = malloc(n * sizeof(int));

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

    return a;
}

int *squares = malloc_int_array(10);

for (int i = 0; i < 10; i++) {
    squares[i] = i * i;
}
```

Memory allocated on the heap strikes a happy medium between static and stack variables. Unlike static variables that are always fixed in size, the size of

each heap allocation is determined at runtime. Unlike stack variables, heap memory allocated using malloc() will remain in the heap 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:

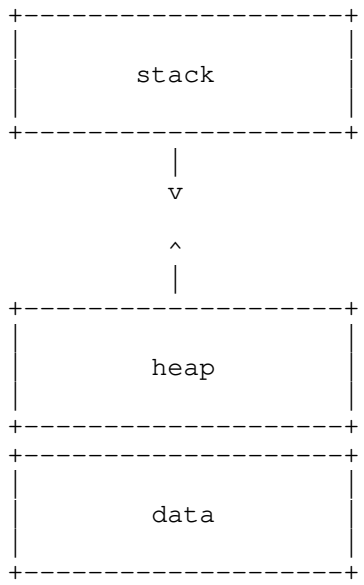
- Using memory beyond what is allocated (e.g., int *p = malloc(1);) is a memory error.
- Reading from uninitialized memory (i.e., has not been written to since being returned by malloc()) is a memory error.
- Using a pointer after it has been free()d is a memory error.
- free()ing a pointer twice is 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. This is particularly bad for a long-running process, especially if it keeps leaking, because it will reduce the amount of memory available to other processes in the system. Memory leak is less of a problem for processes that run for a short time and terminate because the operating system reclaims all the memory used by a process when the process terminates. 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.