/unc/comp211

Systems Fundamentals

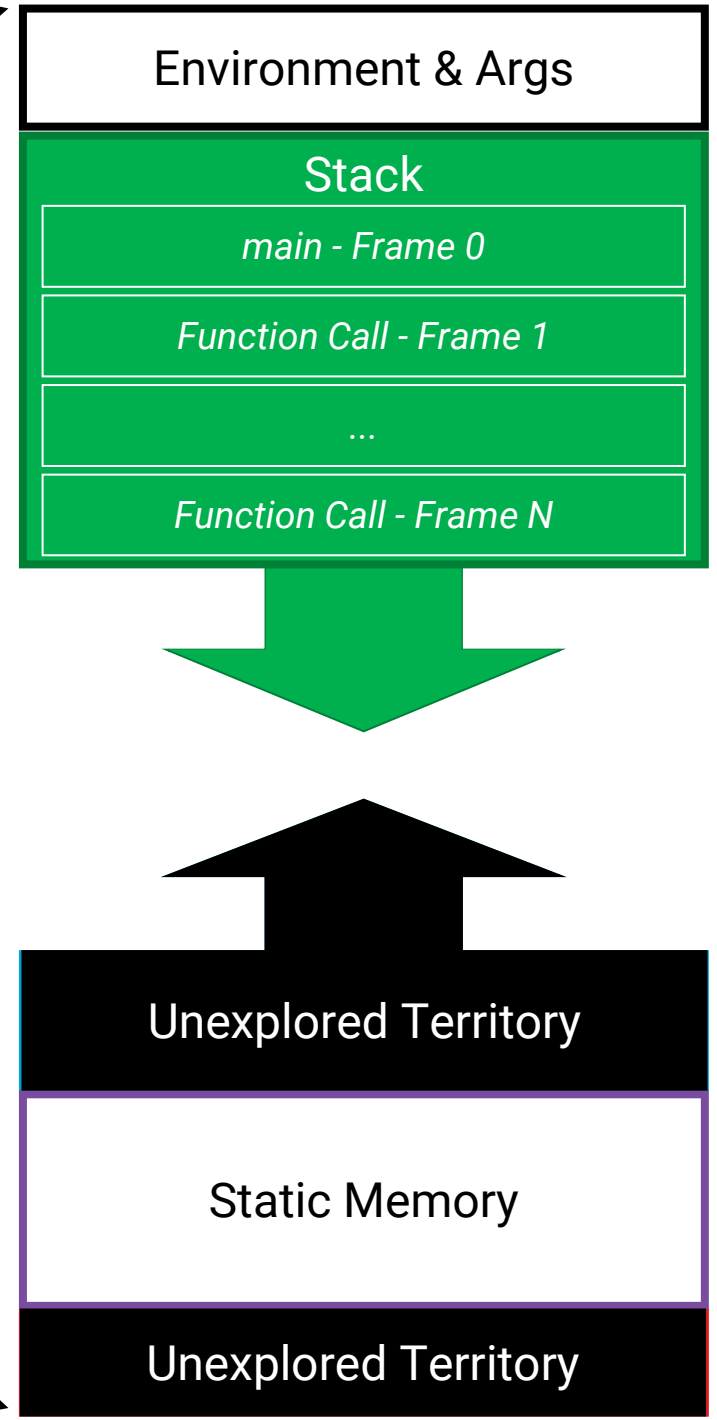
Dynamic Memory !

Stack Memory

- Stack memory is automatically allocated and freed by function calls and returns.
  - The compiler produces the code to control the stack pointer to push a stack frame, arrange automatic variables (and other state) within the frame, and pop it once the call returns.

- Stack memory has useful qualities:
  - Automatically setup and cleaned up for you!
  - Frames containing parameters and local variables provide a working space for functions to do their jobs with some degree of isolation from other functions' values.

- Stack memory also has downsides:
  - Lifetime spans duration of a function call (often very short!)
  - Can't pass arrays by copy (pointers are passed) nor return them because copying arrays is an expensive (in both time and space) task.

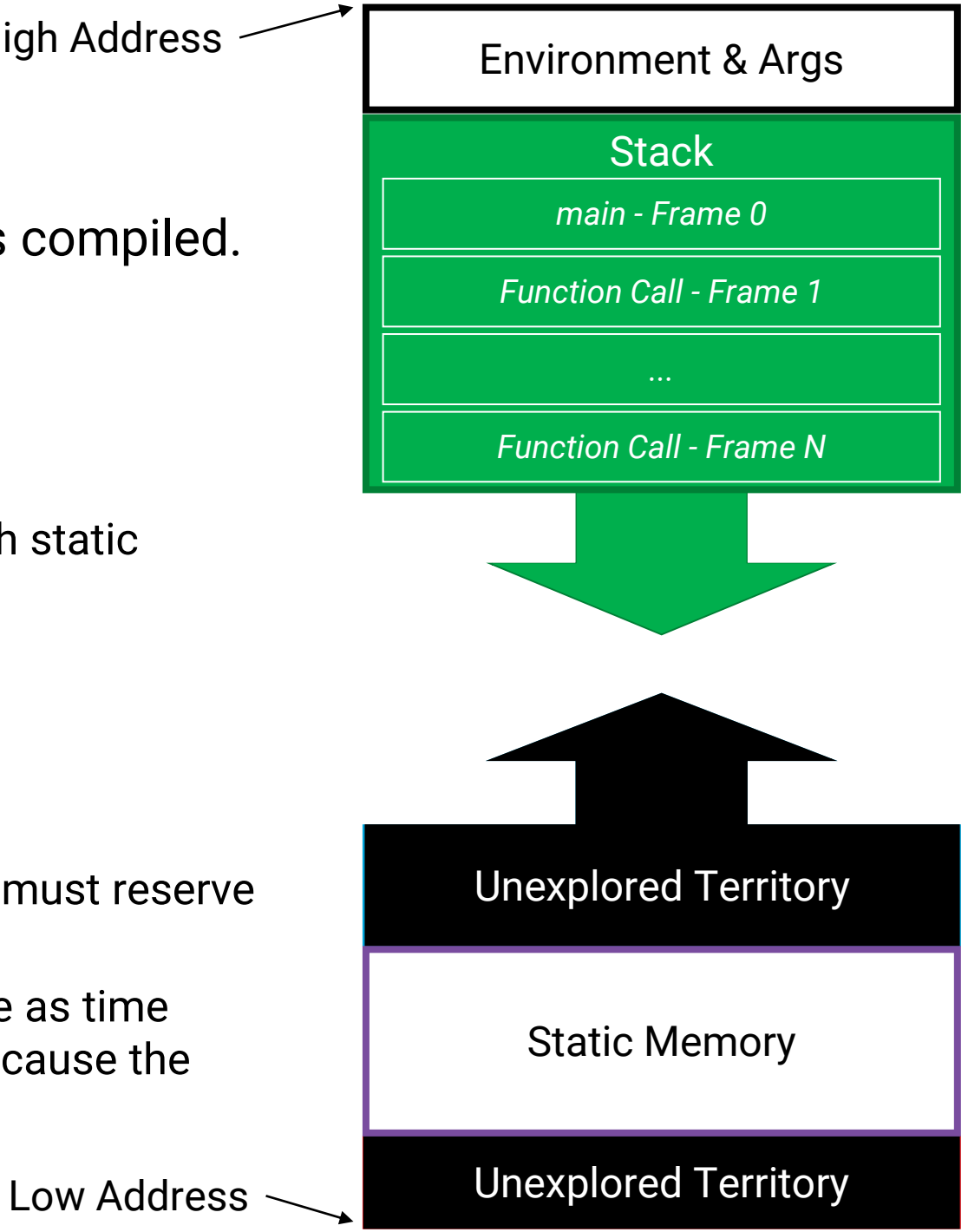


Static Memory

- Static memory is *fixed in size* when the program is compiled.

- Static memory has useful qualities:
  - Lifetime spans duration of running program
  - If your program is running you know there was enough static memory to run it
  - Efficient because the allocation happens once

- Static memory also has limitations:
  - For a program to handle a range of dataset sizes you must reserve enough to fit maximum (wasteful in most cases)
  - Machines have various amounts of memory and more as time passes on, you can't know how much until runtime because the same program can be installed on many machines!

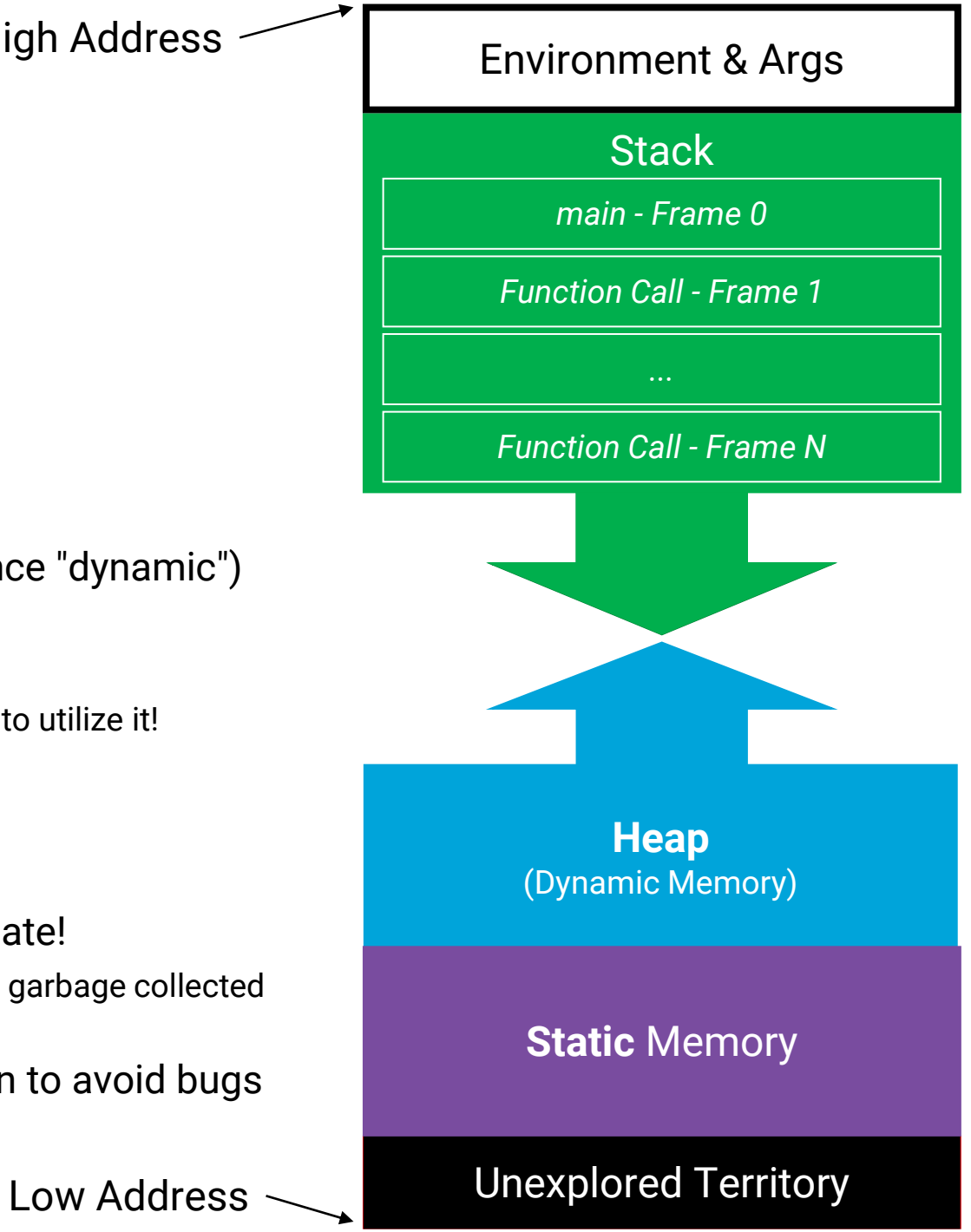


Dynamic Memory

- Dynamic memory is **allocated at runtime**.

- Dynamic memory has useful qualities:
  - ***You*** decide the lifetime of the memory you allocate!
    - Safe after you allocate and initialize it
    - Expired after you free it
  - Processes can request more space as they are running (hence "dynamic") and the operating system will oblige (until it can't!)
    - Allows for a process' memory use to grow to "fit" the dataset size.
    - Also allows for programs running on machines with more memory to utilize it!

- Dynamic memory also has limitations:
  - ***You*** are responsible for the lifetime of the memory you allocate!
    - You must clean up your memory allocations in ways you're used to garbage collected languages doing for you automatically.
  - Manual memory management requires careful consideration to avoid bugs
    - Classes of bugs you haven't needed to *ever* worry about before!



Allocating Dynamic Memory with `malloc`

- **`malloc`** is a standard dynamic **m**emory **alloc**ator
  - http://bit.ly/malloc211
- You ask **malloc** for a raw block of dynamic memory
  - The **size** you request is specified in **bytes**!
  - **To malloc larger data types you must compute bytes via: `sizeof(type) * # elems`**
    - This demo is simple because 1 char = 1 byte
- **malloc** returns a pointer to you
  - **In exceptional cases**, **the pointer will be NULL** if it is unable to provide the sized block you asked for. This means your system is out of memory.
    - **You must handle NULL pointers.** Typically you'll print an error and exit immediately. If you don't, your program will crash sometime later when the null pointer is dereferenced.
  - **Normally the pointer will be the address of the first byte in the block of dynamic memory you requested.**
    - You must assume this memory is *not* yet initialized.

```c
/**
 * Allocates a character array on the heap of size_t and zeros
 * the memory out. Program exits if out of memory.
 */
char *string_malloc(size_t size)
{
    char *str = malloc(size);
    if (str == NULL) {
        fprintf(stderr, "out of memory");
        exit(EXIT_FAILURE);
    }
    for (size_t i = 0; i < size; ++i) {
        str[i] = '\0';
    }
    return str;
}
```

What gets `malloc`'ed must be `free`'d !

- Dynamic memory management in C is manual:
  - When you allocate dynamic memory, you must free it back up when you're done.
  - Where's the cleanup in this example? There is none! We have a memory leak!

- How do you know you have a memory leak?
  1. Look at your code: Do you have a malloc without a matching free?
     - **If so, you have a memory leak!**
  2. Use valgrind: Valgrind is a tool that (among other things) can test for leaks.
     - **valgrind --leak-check=full ./a.out**

Docs: The *free*() function shall cause the space pointed to by *ptr* to be deallocated; that is, made available for further allocation. Any use of a pointer that refers to freed space results in undefined behavior.

- Fix the memory leak in this demo after printing str's contents in main.
  - Is this specific memory leak *really* a big deal? Yes and no.
  - Yes: Being lazy about freeing allocated dynamic memory is a slippery slope. If you're doing any manual memory management *at all* you should habitually free it.
  - No: This specific silly program is not long running and it ends immediately after printing anyway, so the operating system would reclaim the memory no matter what.

```
free(str);
str = NULL;
```

The Lifetime of Dynamically Allocated Memory

- Dynamic Memory Lifetime
  - **Safe** after **alloc**ation *and* **initialization**.
  - **Expires** after **free**.

- Lifetime is ***NOT*** scope!
  - The pointer variable you freed is still in scope *after* free and still points to the same location in memory... ***but*** *you must treat the pointer as expired!*
    - Why? Because future calls to malloc will try to *reuse* the space you just freed. To free dynamic memory means to be done with it.
  - Good Practice: assign NULL to a pointer variable immediately after freeing it.
    - Why? If you accidentally dereference it again after freeing, your program will segfault.

`malloc` vs. `calloc`

- When dynamically allocating array-like memory space, prefer **calloc**.
  - Documentation: http://bit.ly/calloc211
  - It inherently zeroes out all elements of the array.
  - Its function signature is more fool proof:
    - number of elements
    - size of an individual element

- When dynamically allocating memory for structs, prefer **malloc**
  - Structs allow you to cluster multiple values together in memory and access individual values by field names (next chapter is about structs)

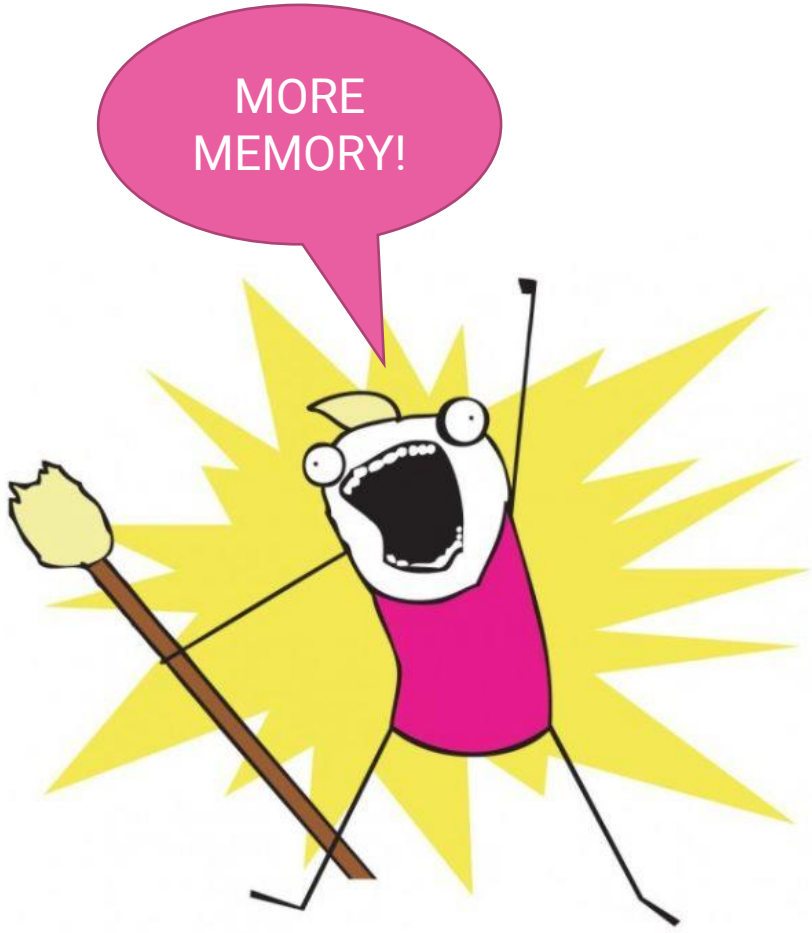
```c
/**
 * Allocates a character array on the heap of size_t and zeros
 * the memory out. Program exits if out of memory.
 */
char *string_calloc(size_t size)
{
    char *str = calloc(size, sizeof(char));
    if (str == NULL) {
        fprintf(stderr, "out of memory");
        exit(EXIT_FAILURE);
    }
    return str;
}
```

What do you do when you need more space?

- Unlike automatic and static memory, dynamic memory gives us the ability to GROW our memory usage on-demand.

- Suppose we have array-like data on the heap and want to store more values in the array. We could:
  1. Allocate memory for old + new space needed.
  2. Copy old values to new memory.
  3. Free pointer to old memory.
  4. Use pointer to new memory instead.

- The above steps are what the standard library's **realloc** function does, if necessary.



```c
/**
 * Resize char array to be of `size` bytes and return pointer
 * to (potentially) new location of char array. If moved,
 * all bytes copied automatically. Program exits if out of mem.
 */
char *string_realloc(char *str, size_t size)
{
    str = realloc(str, size);
    if (str == NULL) {
        fprintf(stderr, "out of memory");
        exit(EXIT_FAILURE);
    }
    return str;
}
```

```c
int main()
{
    char *str = string_calloc(ALPHABET_LEN + 1);
    fill_lowercase(str);
    printf("%s\n", str);

    str = string_realloc(str, ALPHABET_LEN * 2 + 1);
    fill_lowercase(&str[ALPHABET_LEN]);
    printf("%s\n", str);

    free(str);
    str = NULL;
}
```

Lifetimes

Memory Segment	Safe After	Expires When	Runtime Dangers (Things that will (hopefully!) crash your program.)
Call Stack • automatic variables • parameters	Initialization	Function Call Returns	1. Returning or sharing address of automatic variable outside its scope. 2. Stack overflow (infinite recursion).
Static Memory • static variables • global variables	Always* (Uninitialized variables are zeroed.)	Process Exits	1. Writing to read-only static memory. 2. Unintended conflicting writes from many places (grave concern in multithreaded programs).
Heap Memory • "dynamic" memory	malloc and initialization calloc	**free**'d **realloc**'ed (argument)	1. Out of memory 2. Use before initialization 3. Writes overflowing allocated space 4. Failing to free (memory leak) 5. Use after free 6. Double free