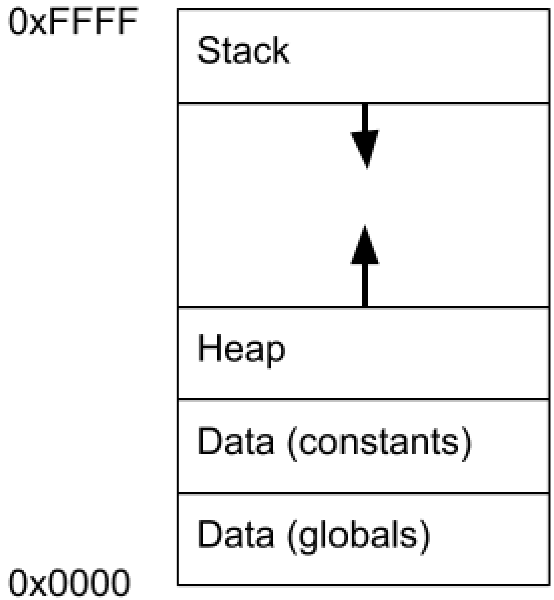
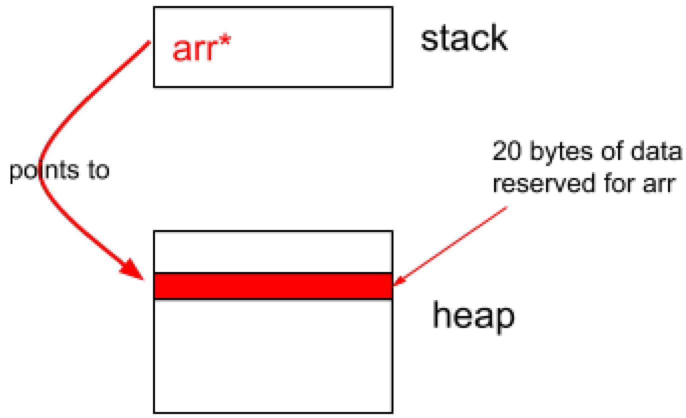
Memory Management



- Local variable allocated on the stack
    - Lifetime: when it's called → when the function ends
- Global variables put in the data segment
    - Automatically allocated by the OS
    - Lifetime: entire program
    - There is also a read-only data segment for constants
- The heap
    - Programmer-managed area of memory
        - Has nothing to do with the heap data structure
    - You can create and destroy pieces of memory on demand
        - Heap lifetimes can cross boundaries
- Stack and the heap both change size
    - One end of the stack is fixed, one end changes
    - Stack grows down (high addresses to lower addresses)
    - Heap grows up (lower addresses to higher addresses)
    - If they meet, the program is out of memory
    - Why is it arranged this way?
        - Arbitrary convention
    - Why not grow outward from the center?
        - You would be basically throwing away half of the memory
        - The heap would only be able to occupy half the memory
    - Why in the same area of memory at all?
        - If you have a 16/32 bit computer, you don't have any options

Heap functions
- `<stdlib.h>`
- `malloc`
    - To allocate memory

- `int* arr = malloc(sizeof(int) * 20;`
    - Gives space for 20 ints on the heap assigned into a pointer named arr

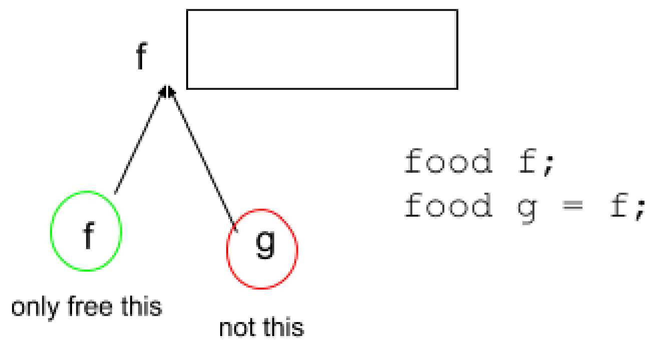


    -
- `malloc` takes the number of bytes to allocate, makes a block of bytes at least that big, then returns a pointer to the block
- `free`
    - When you're done with that block of memory, use `free`
    - `free(arr);`
    - If you `malloc` a second thing, it might use that freed data
    - But there's still a pointer in the stack
- Void pointers
    - `void* malloc(size_t size);`
    - An `int*` points to an `int`, a `FILE*` points to a `FILE` object
    - A `void*` is a universal pointer
        - It can point to anything
    - Any address can go into `void*`
    - A memory address with no meaning
    - Can be assigned any other type of pointer
        - `int x; float y;`
        - `void* vp = &x;          // ok`
        - `vp = &p;          // ok`
    - Can't access data from a `void*` without casting

Allocating a struct on the heap

- Similar to new in Java
- `food* f = malloc(sizeof(food));`
- Like anything on the stack, everything you `malloc` contains garbage
    - Papers on the stack of papers might have writing on them
    - Need to allocate an initialized struct to avoid undefined behavior
- To fix this, you can use `memset` from `<string.h>`
    - `memset(f, 0, sizeof(food));      // will fill the memory at f of size of food number of bytes with 0s`
- Could also use `calloc`
    - `food* f = calloc(sizeof(food), 1);`

        - `calloc` is a `malloc` followed by a `memset`
        - "Allocated 1*sizeof(food) bytes and fill them with 1s"
    - Zeroing out all the data fields so they're not filled with garbage
    - It's faster to `malloc` and initialize yourself
    - Caveats:
        - Do NOT do:
            - while(1) malloc(1048576);
                - Program will run out of memory without error; it loops forever because if you run out of memory, malloc returns NULL
        - `int* arr = malloc(sizeof(int)*20);`
        - `free(arr);`
        - `arr[0] = 100;`
            - As soon as you free pointers, all pointers to it become invalid and have undefined behavior
            - It's your responsibility as the user to not use invalid pointers
        - Can only `free` a piece of memory ONCE



        -

Heap rules:

1. When you `malloc` you should check if it returns NULL
2. You must `free` everything you allocate
3. You must only `free` it exactly once
4. You must not access memory that has been freed
    a. Who is responsible for deallocating? How do you know how many things are using a piece of memory? What if nobody points to it?

VLAs in C99

- Modern c code can allocate variable-sized pieces of memory on the stack instead of the heap
    - `int len = strlen(str);`
    - `char newArray[len + 1];     //not a constant size`
- Called a variable-length array (VLA)
- It's still on the stack
- Increases the size of the activation record
- Gives automatic lifetime
    - Automatically freed

- Not used for everything because stack space is limited
    - On Thoth, it’s only 10 MB
- Also doesn’t solve the problem of ownership
    - Sometimes you need data to stick around longer than the length of the function
    - Have to hand the pointer off
        - Can’t return pointers as local variables
- Also stack is linear, cannot implement a tree on the stack

Garbage collection:

- If nothing is pointing at something and the something isn’t pointing at anything, it gets garbage collected
- Roots: any part of memory that isn’t the head (stack, globals, etc.)
    - Program is using it currently, cannot get rid of it
- Reachability: an object is reachable if there is a path from the roots to the object
- When you remove the stack variable, the only way to reach the object, it becomes garbage
- In C, there is no garbage collector and the heap can get filled up
    - This is a memory leak: losing the last reference to a piece of memory
        - No link to an object from the roots
        - We can never deallocate it, it leaked out of our program
        - The only way to deallocate it is to exit your program
        - All of your program’s memory is deallocated when you exit
    - Never take garbage collection for granted