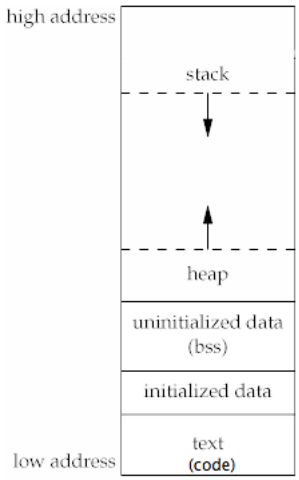
15-122: Principles of Imperative Computation

Recitation 11 — Josh Zimmerman

Memory layout

So far in 122, we've been ignoring the details of how and where C0 stores things in memory. Today we're going to talk about how exactly that works.

On the machines we'll be talking about in this class (those that have an x86 architecture), memory is laid out as follows:



A key idea here is that everything in memory has an address. C0 doesn't let you get all of these addresses, though. (Being able to get every address can be useful, but also makes it much easier to write code that crashes or has security vulnerabilities.) In C0 the only way to get a pointer (an address of something) is to use the `alloc` or `alloc_array` functions.

The key sections of memory we'll be talking about are the *stack*, the *heap*, *text* (where code is stored—yes, it's a confusing name) and *data*, where constants and global variables are stored. We won't talk about the data section for now, since C0 doesn't have any constants or global variables.

The *text* section contains the instructions that tell the computer what to do (the compiled version of your programs, for instance). For this class you don't need to know much more about it than that, but if you take systems classes like 213 and courses that follow it you'll learn more about it.

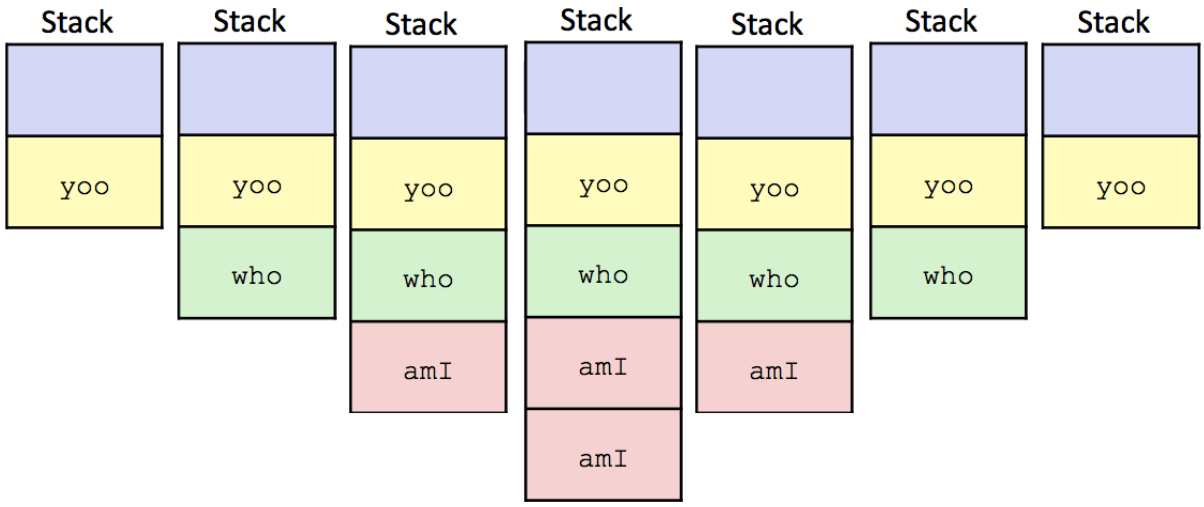
The *heap* is where memory allocated when you call `alloc` or `alloc_array` is placed. For instance, if I say `int[] a = alloc_array(int, 30);` then the array `a` will be located on the heap. Similarly, if I write `int* b = alloc(int);` then `b` will be a pointer to an `int` and that `int` will be on the heap. `b` itself is stored on the stack (more on that in a moment).

The *stack* contains local variables that aren't allocated with `alloc` or `alloc_array`, arguments to functions (depending on your architecture, which you don't need to worry about for this class), and various information necessary for your functions to run and return. Whenever you call a new function, the stack grows (towards lower addresses) to make room for the *callee*'s variables and arguments.

Let's look at an example of how the stack grows when we call functions. (Credit for these diagrams goes to the 15-213 lecture slides. If you want to view them (you don't need to know the material until you take 213), they're at `http://www.cs.cmu.edu/~213/lectures/07-machine-procedures.pdf`)

Consider this code (someone in 213 chose strange names for functions) after a call to `yoo`.

```
1 int amI(int a, int b)
2 {
3    if (b == 1) {
4       // do some stuff
5       return a;
6    }
7    return amI(a, 1) + b;
8 }
9 int who(int a, int b, int c)
10 {
11    // do something
12    return amI(a * tmp, c);
13 }
14 int yoo(int a, int b)
15 {
16    int someLocalVariable = 2 * a + b;
17    // do something
18    return who(a, b, someLocalVariable);
19 }
```



The diagrams show how the stack changes as we call functions: The first diagram is before we call `who`, the second is after we call `who`, the third is after we call `amI`, the fourth is after `amI` calls itself, and then the rest are after functions start returning: the fifth is after `amI` returns, the sixth is after `amI` returns, and the seventh is after `who` returns.

In the stack frame for `yoo` we have `someLocalVariable` stored as well as necessary information for the function call to work.

Recursion

As we saw above, when a function calls itself it grows the stack. This is because each call of the function needs its own space for local variables and information that allows it to return to the function that called it. For more detail, see the section in this handout called "Recursion".

To prove correctness of a recursive function, we follow a somewhat similar process to the one we use to show that an iterative function is correct.

First, we show partial correctness (i.e., if the code terminates it is correct) and then we show termination, just as we do when we have loops. The way we do this is somewhat different, though.

We show partial correctness by induction: First, we show correctness for some kind of base case and then we show that if our function is correct for some fixed input it is true for a larger input. (In other words: we assume that the recursive call returns correctly and show that that means this function call returns correctly)

Then, we show that for any valid input, the function must eventually terminate.

Checkpoint 0

In lecture, we discussed the relatively efficient "tortoise and hare" algorithm for checking whether a list is circular.

```
1  bool is_circular(list* l)
2  {
3    if (l == NULL) return false;
4    list* t = l;  // tortoise
5    list* h = l->next; // hare
6    while (t != h)
7    {
8      if (t == NULL) return false;
9      if (h == NULL || h->next == NULL) return false;
10     t = t->next;
11     h = h->next->next;
12   }
13   return true;
14 }
```

Is this a correct implementation? Is the hare capable of "skipping over" the tortise when approaching from behind? If so, what is the appropriate fix?

Checkpoint 1

How many times is a pointer accessed within the loop? How do we know each access is safe? What happens if `h->next->next` is `NULL` at the beginning of a loop?

Checkpoint 2

The check `t == NULL` on line 8 is unnecessary. First come up with a rough operational reason why this is the case, then state this reason in terms of a loop invariant involving `is_segment`.