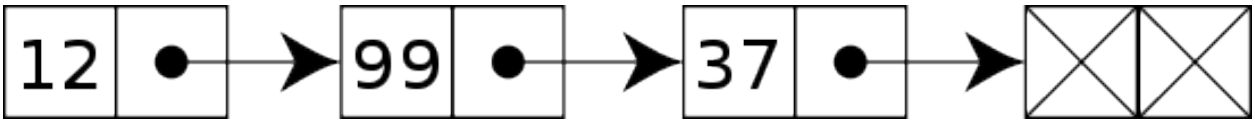
15-122: Principles of Imperative Computation

Recitation 10 — Josh Zimmerman

Linked lists

A linked list is a data structure that allows you to easily store a variable amount of data in a list.



Note that by convention we will normally use a *sentinel* (or *dummy*) node in 122 that flags the end of the list. It is uninitialized. When we see that sentinel node, we know we're at the end of the linked list.

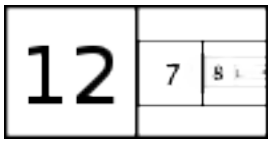
Adding a node to the linked list is as simple as initializing the data in the sentinel node, making a new sentinel node, and pointing the old sentinel node to the new sentinel node.

In C0 , we can define a node in a linked list using a `struct` datatype (in this case, the linked list we make stores `int`s):

```
struct list_node {
    int data;
    struct list_node* next;
};
```

A `struct` is simply a way of defining a datatype that is an aggregation of other datatypes. In this case, we're saying that a `struct list_node` is a datatype that has an `int` and a pointer to another `struct list_node`. Note that it's crucial that we have a pointer to another `struct list_node` rather than a `struct list_node`: if we had a `struct list_node` instead, we'd have a `struct list_node`-ception (to be more formal, we'd recurse infinitely in our definition of the `struct list_node`): in every `struct list_node`, we'd have another `struct list_node`.

For a more concrete explanation of why that alternate definition is bad, see this image:



The reason that this doesn't happen with a `struct list_node*` is that if we have a pointer to a `struct list_node` all that this does is say "this variable (`next`) will either be `NULL` or tell us *where to find* a `struct list_node`."

Pointers, or "Analogies are like your car. They can break down if you're not careful about taking it to the mechanic"

At this point, we need to go into more depth about what exactly a pointer is.

In C0 , for every datatype, there is a corresponding *pointer* that can point to that datatype. For example, for `int`s, there is an `int*` datatype. We know that anything of type `int*` will either be `NULL` or tell us where to find an `int`.

You can imagine a pointer as sort of like a physical address. If I have the address of my favorite restaurant saved on my computer, then that tells me where I can go to find it. If the restaurant goes out of business and gets replaced with a different restaurant, I'll still have the address saved on my computer and I can still go to that address, I just won't find what I expect to. Similarly, if a demolition crew also has a copy of the address of my favorite restaurant and it goes there and knocks the building down, then the building will be knocked down when I next visit it later that day.

A pointer is just the address of a location in your computer's memory and works similarly. If someone else has a copy of that address and they modify the part of memory that the pointer points to (or, in other words, that is at that address), then you'll see that modification the next time you visit that address.

Note that if you want to look at what is at the address that a pointer talks about, you need to do use the * operator. If `p` is a pointer, `*p` says "go to the address that `p` specifies and bring me what is there." This is referred to as *dereferencing* a pointer.

Since we C0 requires that we pass around pointers to `structs` rather than the `structs` themselves (and since it's good normally good practice to do that in C), there's some syntax that we use when we want to both dereference a pointer to a `struct` and access a field of that struct: `(*p).a` will dereference `p` and get the field in it that's called `a`, and so will `p->a`. These two functions do the same thing (but the one on the right is easier to read).

```
void increment_data(struct list_node* l)
//@requires l != NULL;
{
    (*l).data = (*l).data + 1;
}
```

```
void increment_data(struct list_node* l)
//@requires l != NULL;
{
    l->data = l->data + 1;
}
```

A `NULL` pointer is sort of like an address of a house that someone else lives in and that is guarded by violent armed guards. If you attempt to go to it, the guards will shoot you (unless you are already authorized to go to the house, which you won't be at least until you take OS and maybe not even then).

In C0 , following a `NULL` pointer will crash your program with a segmentation fault. For this reason, it's critical to *always* make sure that any time you follow (or *dereference*) a pointer that that access is safe, just like what you must do when accessing an array. You should add contracts and conditional statements to your code to ensure that any pointer access is safe. For instance, the code fragment on the left is BAD because it could cause a segmentation fault, but the code segment on the right will work assuming that its preconditions are met. (Note that neither of these code fragments modify the data that `p` points to since there are no assignments.)

BAD – DO NOT EVER DO

```
int follow_and_add_one(int* p)
{
    // If p is NULL this crashes
    return *p + 1;
}
```

Good

```
int follow_and_add_one(int* p)
//@requires p != NULL;
{
    return *p + 1;
}
```

There will be some cases that you *cannot* use contracts like the above to guarantee you won't dereference `NULL`. In those cases, you should instead use conditional statements to guarantee that you won't dereference `NULL`.

It's **really, really important** to note that the analogy of physical addresses and doesn't work as well when you examine it in depth. I'm using it because it can help you to understand the general concept behind a pointer. Later in the semester, we'll examine pointers in more depth and see how they work in C, but for now the main important takeaways are:

- A pointer tells you *where* to find something. If that something changes, you can still look where the pointer tells you to look, but you'll find something different than what you originally put there.
- If `p` is a pointer that is not `NULL`, `*p` *dereferences* `p`, meaning it lets us directly read and modify the data that is stored at the place where `p` points.
- If `sp` is a non-`NULL` pointer that points to a `struct`, then `sp->data` means the same thing as `(*sp).data`.
- Dereferencing `NULL` pointers is an error and will cause your program to crash in C0 . Whenever you dereference a pointer, you should have proof that it is not `NULL`.

alloc

Now we know what a pointer is, but how do we actually *get* one?

The expression `alloc(t)` will give us a pointer to an area of memory that can hold something of type `t`. For example, if I write `int* a = alloc(int);` then the computer will make space somewhere in memory for an `int` and will put the address of that area of memory in `a`. In other words, after we execute that expression, `a` will be a pointer that points to an `int`.

You can use this for any type: If I want space for a node in a linked list, I'd type `struct list_node* list = alloc(struct list_node);` and then `list` would be a pointer to an `struct list_node`.

Practice!

Credit for this section goes to CMU alumna Caroline Buckey. It was updated by current 122 TA Alex Cappiello to account for differences in exactly how we teach certain material.

Suppose you have queues implemented using linked lists as shown in lecture. Specifically, you have the following structs:

```
struct list_node {
    int data;
    struct list_node* next;
};
typedef struct list_node list;

struct queue_header {
        list* front;
        list* back;
};
typedef struct queue_header* queue;
```

Recall from lecture that in both stacks and queues, we always keep one dummy node. In queues, `back` points to this dummy node. Its fields are uninitialized, and it simply ensures that we never need to worry about `front` or `back` being null.

Visualizing `queue_new()`

Here's the code for `queue_new()`. For each of lines 5-8 (inclusive) draw a diagram that shows the state of the queue after that line executes. Use `X` for uninitialized fields.

```
1  queue queue_new()
2  //@ensures is_queue(\result);
3  //@ensures queue_empty(\result);
4  {
5     queue Q = alloc(struct queue_header);
6     list* p = alloc(struct list_node);
7     Q->front = p;
8     Q->back = p;
9     return Q;
10 }
```

5.

6.

7.

8.

Visualizing `deq`

This is the code for `deq`:

```
1 int deq(queue Q)
2 //@requires is_queue(Q);
3 //@ensures is_queue(Q);
4 {
5    int x = Q->front->data;
6    Q->front = Q->front->next;
7    return x;
8 }
```

Suppose `deq(Q)` is called on a queue `Q` that contains before the call, from front to back, `(4, 5, 6)`. Draw the state of the queue after lines 5 and 6 execute. Include an indication of what data the variable `x` holds.

5.

6.

Visualizing `push`

This is the code for `push`, and the definitions of the relevant `structs`.

```
1 struct list_node {
2    int data;
3    struct list_node* next;
4 };
5 typedef struct list_node list;
6
7 struct stack_header {
8    list* top;
9    list* bottom;
10 };
11 typedef struct stack_header* stack;
12
13 void push(stack S, int e)
14 //@requires is_stack(S);
15 //@ensures is_stack(S);
16 {
17    list* l = alloc(struct list_node);
18    l->data = e;
19    l->next = S->top;
20    S->top = l;
21 }
```

Suppose `push(S, 3)` is called on a stack `S` that contains before the call, from top to bottom, `(1, 2)`. Draw the state of the stack after each of lines 17 - 20 (inclusive). Include the list struct separately before it has been added to the stack.

17.

18.

19.

20.