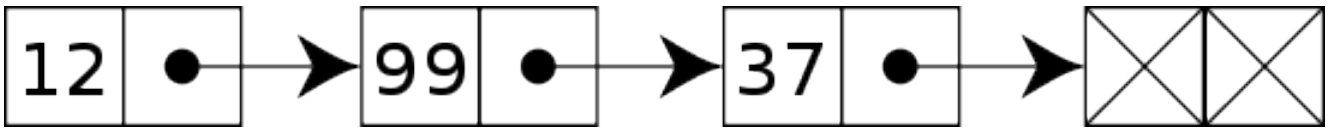
15-122: Principles of Imperative Computation Section G

Recitation 9 — Josh Zimmerman

Linked lists

A linked list, as many of you likely already know, 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.

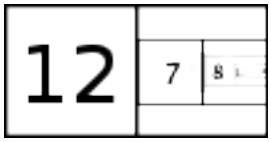
Adding a node to the linked list is as simple as initializing the data in the sentinel node, making a new sentinel node, and pointing the old sentinel node to the new sentinel node.

In C_0, 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`."

If I have a variable that is a `struct list_node`, then I can access it like this:

```
struct list_node increment_data(struct list_node l)
{
    l.data = l.data + 1; // Can also be written as l.data++
    return l;
}
```

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 C_0, 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 (assuming no one rebuilds it).

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 what is called *dereferencing* the pointer. 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 often pass around pointers to `struct`s rather than the `struct`s themselves, 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`. If we wanted to rewrite the above `increment_data` function to take a `struct list_node *`, these two implementations of it would be the same:

```
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 during 122 you will never be).

In C_0, 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. Similarly to how we ensure that array access is safe, 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 on the next page is BAD because it could cause a segmentation fault, but the code segment on the right will work. (Note that neither of these code fragments modify the data that `p` points to since there are no assignments.)

BAD – DO NOT EVER DO	Good

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

There will be some cases, like if you are traversing a linked list, that you *cannot* use contracts like the above to guarantee you won't dereference `NULL`. If you traverse a linked list and throw an annotation error as soon as you see a `NULL` pointer, then your code will throw an annotation error on any finite list because the sentinel node in a list points to `NULL`. In that case, you'd need to use a condition (like a loop guard or an if statement) to ensure that you don'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 C_0. Whenever you dereference a pointer, you should have proof that it is not `NULL`.

alloc

All this talk of pointers is well and good, but how do we actually *get* a pointer?

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 a pointer to the place that that `int` is located 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`.

Typedefs: a brief interlude

It's often annoying to type out the full name of a type, either because it's a long name or because it's difficult to conceptually think about the original name.

For instance, typing `struct list_node` every time we want to refer to a node in a linked list is annoying. So, if we execute the command `typedef struct list_node list;` that tells the compiler "every time I use the type name `list`, what I actually mean is `struct list_node`." Now, we can just say `list` instead of `struct list_node`.

On assignment 1, some of you may have noticed that `imageutil.c0` had the line `typedef int pixel;` in it. This is because we wanted you to think about `pixels` just in terms of the underlying bits that make them up, rather than as numbers that you can do arithmetic with. In this case the new type name is longer, but using a `typedef` allows you more easily think about the problems we asked you to solve rather than the underlying implementation.

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, with integer data. 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.

Problem 1

Recall the code for `queue_new()`:

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

Draw the state of the queue after each given line of code. Use `X` for uninitialized fields.

1.

2.

3.

4.

Problem 2

Recall the code for `deq`:

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

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 each given line of code. Include an indication of what data the variable `x` holds.

1.

2.