Pointers

A pointer of type $t^*$ is either NULL or a valid address where a value of type $t$ is stored.

Obtaining a pointer:
Suppose you want a pointer to a value of type $t$. In C0, we have the $\text{alloc}$ function for that.

$t^* \text{ alloc}(t)$ - returns the address of a chunk of memory that can hold a $t$.

Allocating and dereferencing a pointer:

```c
int* ptr = alloc(int); // Pointer to an int at some address, like 0xFF57DFF0
*ptr = 23; // The value stored at address 0xFF57DFF0 is the int 23
/*@assert (23 - *ptr) == 0 @*/
```

NULL:

```c
string* n = NULL; // Local string pointer declared as NULL pointer.
*n... – don’t you dare! The null pointer cannot be dereferenced. C0 will be very upset with you if you try.
```

Aliasing:

```c
bool* b1 = alloc(bool);
*b1 = true;
bool* b2 = b1; //@assert (*b1 == *b2) && (b1 == b2);
*b2 = false; //@assert (*b1 == false);
```

Structs

A struct is a data type that contains other data types. In C0, you obtain a struct $t$ by calling $\text{alloc(struct } t\text{)}$.

Notice that doing so will return to you a pointer to where your struct $t$ exists.

```c
struct sentence {
    string sent;
    int length;
};

struct sentence* my_sentence = alloc(struct sentence);
//@assert my_sentence != NULL;
my_sentence->sent = "I love C0";
my_sentence->length = 3;
//@assert string_equal(*my_sentence).sent , "I love C0";
//@assert (my_sentence->length) == 3;
```

Linked Lists

Notice that each node contains a pointer to the next node, rather than the next node itself. Why?

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

```
42 ➔ 17 ➔ 31 ➔ 3 ➔ ⬤
```
Let's look at the implementation of queues as we covered in lecture yesterday. A queue is a header struct with a pointer to the front of the queue and a pointer to the back of the queue, where the queue elements are nodes in a linked list. The pointer to the back of the queue is a dummy node with no data and whose next pointer is null. We know the queue is empty when the front pointer and the back pointer are the same (pointing to the dummy node).

```c
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;
```

The command `typedef old_t new_t` means that when you see a value of type `new_t`, it has underlying type `old_t`.

For example, recall homework 1: `typedef int pixel`. We treated pixels as their own type, even though their representation was that of an int. The purpose was to make you think about the ARGB bytes in the `pixel` rather than the numerical value of the `int` as a whole.

We interpret these typedefs as follows: a `list` is a `struct list_node`, and a `queue` is a pointer to a `struct queue_header`.

How would you determine the length of a queue? How would you look at the first element without dequeuing it?

```c
int queue_length(queue Q)
//@requires Q != NULL;
//@ensures 0 <= \result;
{
    int count = 0;
    list* elt = Q->front;
    while (elt != Q->back)
        //@loop_invariant elt != NULL;
        {
            count++;
            elt = elt->next;
        }
    return count;
}

int peek(queue Q)
//@requires Q != NULL;
//@requires queue_length(Q) > 0;
{
    //@assert Q->front != Q->back;
    return Q->front->data;
}
```