Wesleyan University, Fall 2022, COMP 211

Lab 8: Queues, Structs, and Pointers

---

1. Overview

The goal of this lab is to is to introduce you to pointers and structs, give you practice working with pointers in the context of arrays and structs, and use these concepts to implement an queue_new data type called a queue, using an array as the data structure.

2. Create and organize your program files

For this lab, you should create a new directory lab8 in your labs directory to hold your code. Then do the following.

(1) Download the following files from the lab website, `ptr_example.c, Makefile, queue.c, queue.h, qdriver.c`. During this lab you will add code to the following files: `queue.c`, `queue.h`, and `qdriver.c`.
(2) Use the `Makefile` to compile the current state of the program by typing `make qdriver`. Then type `./qdriver` to run the program.

3. Working with queues

A queue is an abstract data type which you will be familiar with since it frequently occurs in everyday life. For instance, when you choose a line to checkout in at the grocery store, you are essentially adding yourself to a queue. A queue has the following properties: new elements are always added to the end of the queue (it is bad behaviour to cut ahead of people in a queue in a grocery store), and elements are always removed (dequeued) from the front of the queue (the grocery store clerk always serves the first person in line). Essentially what we have is what is called a first-in-first-out (fifo) queue: i.e., the first person in the queue is the first person out of the queue

In lab today you will implement a queue abstract data type along with functions implementing operations on a queue. Some of this code has already been written for you, and some code you will need to write.

Creating a queue struct. There are actually many different ways to implement a queue, but today we are going to focus on an implementation which creates what is called a `struct` in C, and uses an array plus a few pieces of additional information.

As you will see in your reading, structures in C are a way for you to create a new type that comprises one or more fields, each of those field corresponding to a variable of an existing type. For example, we might create a point struct as in Figure 1. In the file `queue.h`, I have already defined for you a struct named `queue` representing the components of a queue using three fields:

- a `data` field of type `char[]`. This holds the contents of the queue (e.g., the first initial of each person in line) and will have a fixed maximum queue length of 10 for this lab.
- a `front` field of type `int`. This keeps track of the position in the array where the next element will be dequeued from (i.e., it represents the "first person in line" in the queue).
- a `back` field of type `int`. This represents the position in the array where the next element will be put in the queue (i.e., the "next position in line" or back of the queue).

```
struct point{
        double x;
        double y;
};

struct point start = {100, 200};
struct point end = {200, 3};
double slope = (end.y - start.y) / (end.x - start.x);
```

FIGURE 1. Struct example

Passing a struct to a function. If we are going to perform operations, like enqueuing an element, on our queue struct in a function, we need to understand what happens when we pass a struct to a function.

Now a structure is a rather complicated and arbitrary data type, even more complicated than an array. Recall that the value of an array is an address: this means that when an array variable is passed to a function the address is what is passed, and so any operations on the array in the function modify the memory assigned to the array. This is what we would like to happen when we pass our queue struct to a function, but unfortunately, structs are treated differently than arrays in C. In C, when a struct is passed to a function in C, a copy of that entire struct is made, so that any modifications of the struct in the function do not change the original memory associated with the struct variable that was passed. However, we can get around this limitation by explicitly passing the address of the queue struct to our functions, and forcing functions to treat our struct in the same way that they would treat an array.

How do we explicitly pass an address of a variable to a function? Recall we know how to get the address of a variable in C by putting the & operator in front of the variable name. But what type of variable stores an address in C? A pointer. A pointer is a variable that contains the address of another variable: i.e., the value of a variable of type pointer is an address. How do we create a pointer variable? By putting a * operator in front of the variable name when we declare the variable. This allows us to create a pointer variable for any type: this is important because every variable (of any type) has a memory address where the value of that variable is stored in memory. This is shown in the code in Figure 2, where `p` is a pointer variable and `v` is the variable at which `p` is pointing: i.e., `p` contains the address of variable `v`.

Let's now take a look at the `queue_new` function signature in `queue.c`.

```
void queue_new(struct queue *Qptr)
{
        ...
}
```

We see that the type of the parameter passed to the function is of type `struct queue *`: i.e., `Qptr` is of type pointer and its value will be an address for a `struct queue` variable. So we need to pass the address for a `struct queue` to the function. How would we get such an address? Suppose we had already declared our variable with `struct queue Q;`. Then to get the address of `Q` we use the & operator on `Q`, writing `&Q`. We then pass the address to the function using the function call `queue_new(&Q)`. If you look in `qdriver.c`, you'll see that the function calls are all using `&Q` as the value being passed to the functions, to be assigned to the new value `*Qptr`.

This gets us most of the way there. However, if `Qptr` contains an address, how do we work with `Qptr` inside the function `queue_new`? How do we access the various fields in the queue struct that

```
int v = 5;
int *p; // p is a pointer to a variable of type int, indicated by * before p
p = &v; // Assign address of v to be the value of pointer variable p

printf("\n v = %d, &v = %p, p = %p &p = %p, *p = %d \n", v, &v, p, &p, *p);
// v = 5, &v = 0x7fff507d39bc, p = 0x7fff507d39bc &p = 0x7fff507d39c0, *p = 5
// Note that p contains the address of the variable that p is pointing at, while &p
// is the address of p itself in memory. So at location &p in memory, the value of p,
// which is another address, is stored

*p = 10; // Access memory location that p is pointing at by putting * in front of p
// Since p contains the address of v, then *p accesses the memory where the
// value of v is stored. Here, we are assigning the value of 10 to this memory
// location. So we are actually changing the value of v to 10.

printf("v = %d, &v = %p, p = %p, &p = %p, *p = %d \n\n", v, &v, p, &p, *p);
// v = 10, &v = 0x7fff507d39bc, p = 0x7fff507d39bc, &p = 0x7fff507d39c0, *p = 10
```

FIGURE 2. Pointer examples. For more examples see the file `ptr_example.c`. You should play around with this code and make sure you understand what is happening, however, our main focus with pointers is understanding them enough to understand how to pass a struct to a function.

`Qptr` is pointing at? We do this by putting the * operator in front of `Qptr` when we want to access the values in memory at the address `Qptr` stores. For instance, we would write the `queue_new` function as follows.

```
void queue_new(struct queue *Qptr)
{
        (*Qptr).front = 0; // Sets front to 0 in struct Q passed to queue_new
        (*Qptr).back = 0;  // Sets back to 0 in struct Q passed to queue_new
}
```

An alternative, more compact way to write out `(*Qptr).front` and access the values in memory is to instead write out `Qptr->front`. The following code is exactly equivalent to the implementation of `queue_new` that we just saw and is what is implemented for you in the file `queue.c`

```
void queue_new(struct queue *Qptr)
{
        Qptr->front = 0;
        Qptr->back = 0;
}
```

4. Fill out the bodies of the remaining queue functions.

Now it is your turn to do some coding. The standard operations on a queue are the following.

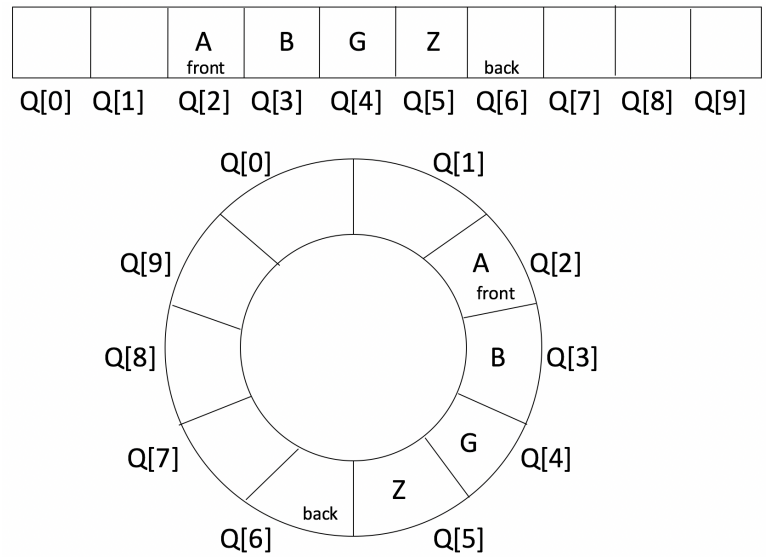

FIGURE 3. Representation of a circular array.

- `queue_new`: initialize the fields in the queue structure.
- `queue_empty`: check whether the queue is empty.
- `enqueue`: add an element to the queue.
- `dequeue`: remove an element from the queue.
- `queue_print`: print all of the elements in the queue.

These function signatures have already been written for you in `queue.c` and `queue.h`, and we have filled out the body of `queue_new` for you. Your job now is to fill out the bodies of the following functions in `queue.c`: `queue_empty`, `enqueue`, `dequeue`, and `queue_print`.

Before starting to code, you should think about what needs to happen to the fields of the queue struct when an element is enqueued and when an element is dequeued. As a hint for dequeueing, there is some shifting of elements that will need to occur.

Your code should print out "Error: out of space in queue" when the queue capacity is exceeded.

Compile and test your code using `qdriver.c` which has already been written for you.

5. Using a circular array to represent a queue

As you were implementing the queue functions, you may have realized that rather than shifting all of the elements in the data array in order to dequeue an element, that it is actually more efficient to just change the values of `front` and `back`. In this task, we will formalize this notion and represent a queue using what is called a circular array. Figure 3 shows how an array can be interpreted as a circular array. Now, when an element is dequeued from `head`, rather than needing to shift all array elements in [`head+1`,`back-1`] to [`head`,`back-2`], instead `head` is shifted to `head+1`.

Fill out the function bodies for the following function signatures in `queue.c` and `queue.h`, now using a circular array to represent the queue: `queue_new`, `queue_empty_circ`, `queue_full_circ`, `enqueue_circ`, `dequeue_circ`, and `queue_print_circ`. One approach you could use to indicate the queue is full is by setting `back = -1`.