CMPS 12M

Introduction to Data Structures

Assignment 6

Objective

The purpose of this assignment is to get started writing code with C and implement a working circular queue with unit tests.

Task

Your task is to implement a circular queue class in C. You are given a set of unit tests that your implementation must pass. Unit tests and template code for this assignment are available for download in a ZIP file.

ISO C99

The C programming language has evolved since its creation in the early 1970s. The latest version in common use is C99. This lab cannot give a full introduction to C programming so will only cover some C idioms that are used in this assignment that may be unfamiliar to you. If you have never used C you are encouraged to work through introductory assignments in any good book on C programming.

C Preprocessor

One distinctive feature of C is the preprocessor. The preprocessor is a step in C compilation that involves textual substitution in source files. Using the preprocessor you can define text macros that automatically expand into strings of source code. A simple example is provided in the template code.

```
#define MAXSIZE 100
```

This is a preprocessor macro definition. After this definition, any instance of MAXSIZE in the source code will be replaced by 100. This allows you to change the number in one place instead of scanning through your code to look for every instance of a hard-coded 100.

The C preprocessor is used in this lab to implement a tiny unit testing framework in minunit.h. You do not need to understand all these macros, just know that they are using preprocessor features to dynamically declare unit testing functions. Any source file that includes minunit.h is able to use the unit testing functions.

Return Values

There are no exceptions in C. Since your queue has operations that can fail, you must use another mechanism to signal error conditions. The standard solution in C is to use the return value of the function to signal success or failure. Usually the value 0 signals success and nonzero values indicate error conditions. Different data structures may have different types of errors. Error number 1 for queues is probably different than error number 1 for stacks.

Enumerated Types

To help keep track of different names and values, C has the ability to create enumerations. An enumeration (enum) is a data type that is implemented as an integer. It provides names for each of the possible values in the enumeration. Here is an example enum for suits in a deck of cards.

```
enum cardsuit {
   CLUBS = 100,
   DIAMONDS,
   HEARTS,
   SPADES
};
```

In the example the constant CLUBS is mapped to the value 100, then the remaining suit constants are automatically mapped to succeeding values.

In C, the type of the enum is "enum cardsuit". This can be somewhat awkward to read and type. For example, here is the type declaration for a function that takes a suit as input and returns a suit:

```
enum cardsuit next_suit(enum cardsuit s);
```

There is a trick to simplify the type. You can introduce a new name for an existing type using "typedef". The following typedef creates a new name "myint" for integers.

```
typedef int myint;
```

After that declaration you can use myint as a regular type in C. It is just another name for int.

We can create a typedef for the enumerated type. Here is code from the queue template:

```
typedef enum {
    /* Enumerated status codes for queue operations */
    q_success = 0,
    q_failure
} q_status;
```

The above code declares that "q_status" is a new name for the anonymous enumerated type included in the typedef. In other words we can use the enumerated type by just typing "q_status" instead of "enum q_status".

Pointers

We've seen references in Java when working with objects. In Java, every variable with object type is really an arrow (reference) to an object. Assigning variables switches where the arrow points.

In C there is no notion of references but there is an explicit notion of pointers. A pointer is a data type in C that contains the address of memory where a value is stored.

Pointers in C use the "*" notation in the type. Declare that v is an integer:

```
int v;
```

Declare that v is a pointer to an integer:

```
int *v;
```

The notation "*" also follows a pointer. Store the value 5 into the location pointed by v:

```
*v = 5;
```

Set x to the value in the location pointed by v plus 1:

```
x = *v + 1;
```

To get the address of an existing location you use the "&" operator before a variable. Read it as "address of". Use a pointer to modify the contents of another variable:

```
int x = 17;
int *y;
y = &x;
*y = 18;
```

Pointers are a powerful part of the C language but can easily be abused. Trying to follow a pointer that does not point to a valid memory location can easily crash your program. Even worse, it can corrupt other data in your program without any adverse effects other than causing weird "impossible" bugs.

Every pointer can be marked invalid by assigning it the NULL value. This is similar to Java's "null" value. Trying to follow the NULL pointer will crash your program.

Output Parameters

Because we decided that the queue functions return status codes to indicate success or failure we have a problem. Functions such as remove() and peek() on queues need to return an item from the queue. But functions can only return one value in C which is already being used for the status.

The solution is to pass a pointer as an argument to the function. The function can then put values into the memory addressed by the pointer. When the function returns, the caller will get a return value that indicates success or failure. If the function returns success the caller can look into the memory location it gave to the function to get the actual returned item from the queue.

Here is an example function that returns a success status and returns 42 in the output parameter:

```
int get_answer(int *answer) {
      *answer = 42;
      return 0;
}
```

Task

Your task is to implement a circular queue class in C. You are given a set of unit tests that your implementation must pass. Unit tests and template code are available in the ZIP file.

Your queue implementation must have the following functions:

q_init	Initialize any data structures needed for your queue, setup any variables
q_insert	Insert an element (integer) into the back of the queue
q_remove	Remove the element at the front of the queue and save it in the provided area
q_peek	Peek at the element in the front of the queue, save it in the provided area
q_destroy	Clean up any data structures needed for your queue

Note that C does not have classes and objects. Your implementation will be a collection of functions and global variables in the queue.c file. Your queue must be able to hold 100 items without problems. Imlement your queue using a circular buffer in a fixed size array of integers.

Your functions must pass all the unit tests. Your functions must correctly return success or failure status for each unit test. Removing elements from an empty queue should fail. Passing a NULL argument as the output parameter should also fail gracefully (it should not crash).

Once your implementation is working, add one additional unit test that inserts three different integers into the queue, then removes three items and checks that the removed values are all correct. Verify your implementation passes your test. Look at the existing unit tests to see how to format unit tests.

Now add an q_is_empty() function that checks if the queue is empty or not. Note that this function cannot fail, so it can return true (1) or false (0) directly rather than a status code. Implement this function and add two unit tests to test it. One unit tests should check a situation where it returns true, one should check a situation where it returns false.

When adding unit tests, make sure the test is actually run. You need to include the test in the all_tests() function as well as defining it.

What to Turn In

In addition to comments in each file, create a file called README which lists all the files being submitted (including itself) along with any special notes to the graders.

For this lab, submit the following files:

```
README
Makefile
queue.c
minunit.h
```

Your makefile should have a default target that builds an executable that does unit tests, along with phony targets "clean" that removes compiled object files, "spotless" that cleans up all built files, and "test" which runs the executable.