CS201: Data Structures and Algorithms

Assignment 0

Version 1

Introduction

For your first programming assignment, you are to build the beginnings of a library of elementary data structures. This initial library will consist of a dynamic array class, a circular dynamic array class, a stack class, and a queue class. The stack class is to be built upon the dynamic array class, while the queue class is to be built upon the circular dynamic array class. The language of implementation is Portable C99. Each of the four classes need to be implemented in its own module.

The classes need to be generic. That is, the type of the value stored in a node object will not be specified by the any of the classes. The user of your array, stack, and queue classes will determine the type of value stored in these data structures. Start by reading this tutorial: `http://beastie.cs.ua.edu/C-first.html`.

Specifications

The specifications for your classes can be found here:

- *dynamic arrays*
- *circular dynamic arrays*
- *stacks*
- *queues*

Your classes will be tested individually by integrating them with my testing code.

Generic data pointers

Generic values in C are stored in `void *` pointers. Any data pointer can be stored in a `void *` pointer with no cast:

```
void *p = newInteger(5);
```

The reverse requires a cast:

```
void *p = newInteger(5);
integer *q = (integer *) p;
```

It is a grave error to cast a `void *` pointer to a type that does not reflect the original value stored in the `void *` pointer.

Function pointers

You can pass a function to another function in C by using a function pointer. Suppose you wish to pass a function with this signature:

```
int plus(int,int);
```

to a function named *accumulate*, in addition to an array of integers. The signature of *accumulate* would be:

```
int accumulate(int *array,int size,int (*combine)(int,int));
```

That third argument is a function pointer. To derive the proper function pointer type, simply take the signature of the function to be passed and perform the following transformations:

```
int plus(int,int);      //original signature
int plus(int,int)       //remove the semicolon
int (plus)(int,int)     //wrap the function name in parens
int (*plus)(int,int)    //place an asterisk immediately before the function name
int (*combine)(int,int) //change the name of the function to the name of the formal parameter
```

Inside the *accumulate* function, the *combine* function pointer can be called like a regular function:

```
total = combine(total,array[i]);
```

Programming style

You must implement your data structures in C99. You must follow the C programming style guide for this project: `http://beastie.cs.ua.edu/cs201/cstyle.html`.

Makefiles

You should provide a *makefile* which responds to the commands:

- make
- make clean
- make test

All compilations should proceed cleanly with no warnings or errors at the highest level of error checking (the `-Wall` and `-Wextra` options). the `make clean` command should remove object files and any executables.

The `make` command should compile all your modules, if they have not already been compiled.

The `make test` command runs all your tests on your implementations. You should provide sufficient testing, especially testing the boundaries of your data structures. Examples of boundary testing are:

- displaying an empty stack
- having a stack go non-empty, then empty, and then non-empty again
- printing the size of an empty queue
- put a large number of values in a dynamic array

Warnings

Depending on where you develop your code, uninitialized variables may have a tendency to start with a value of zero. Thus, a program with uninitialized variables may work on your system but fail when I run it. I won't care, as you are mature enough not to have uninitialized variables. You may have other errors as well that do not reveal themselves until run on my system. Again, that's not my problem.

At any time, you may submit your project to the *compile* drop box, which will send you a report on how well the compilation went. Use this command to submit to this drop box:

```
submit cs201 lusth compile
```

Submissions to the drop box will be automatically compiled, on the order of every 10 minutes.

You can also automatically test your programs by submitting them to the *test0* dropbox:

```
submit cs201 lusth test0
```

Documentation

All code you hand in must be attributed to its authors. Comment sparingly but well. Do explain the purpose of your program. Do not explain obvious code. If code is not obvious, consider rewriting the code rather than explaining what is going on through comments.

Grading

The output of your programs and modules will be tested using the *diff* utility. Your ouput and the expected output must match exactly. A single extra space or a space in the wrong place will cause a test to fail, leading to a resubmission with an accompanying deduction.

Submission

To submit assignments, you need to install the *submit* system.

- *linux, windows 10 bash*
- *mac instructions*

You will hand in (electronically) your code for final testing with the command:

```
submit cs201 lusth assign0
```

Make sure you are in the same directory as your makefile when submitting. The submit program will bundle up all the files in your current directory and ship them to me. Thus it is very important that only the source code and any testing files be in your directory. This includes subdirectories as well since all the files in any subdirectories will also be shipped to me. You may submit as many times as you want before the deadline; new submissions replace old submissions. Old submissions are stored and can be used for backup.