CSC 230 Project 6

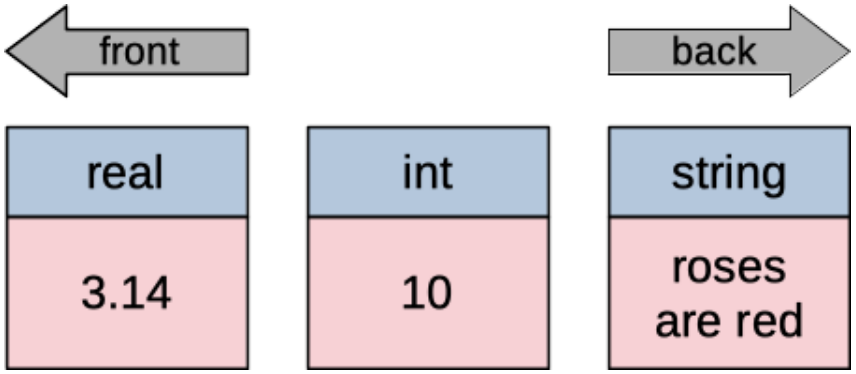
Generic Stack / Queue

This project should be a little easier than the last few. You're going to be implementing a stack / queue data structure that can hold integers, real numbers and strings in its elements. The program will provide a simple mechanism for adding support for additional value types (other than int, real and string).

Your program will be called `omni-q` (since it can store a queue containing different types of values). The following example shows how you can run it. The text in bold is what's typed by the user. The non-bold text is output generated by the program.

```
$ ./omni-q
push int 10
push real 3.14
enqueue string roses are red
pop
3.140
pop
10
pop
roses are red
```

The `omni-q` program maintains a sequence of values. In this example, the first command adds an integer value of 10 to the front of the sequence. The next command puts a real number value of 3.14 in front of the 10, and the third command puts a string value of "roses are red" at the back of the sequence. After these three commands, the data structure will look like the following figure.



Sequence created by the example above

The last three commands remove these three values, one after another, from the front of the sequence. Each value is printed as it is removed, so the program prints the real value, 3.14, then the int value, 10, then the string value, "roses are red".

You will be developing this project in the p6 directory of your csc230 git repo, and, as usual, you'll submit by pushing your changes up to the NCSU github repo before the due date.

We're providing you with a starter that includes files to help you organize and test your program. See the Getting Started section for instructions on how to get the starter and unpack it into your repo.

This project supports a number of our course objectives. See the Learning Outcomes section for a list.

Rules for Project 6

You get to complete this project individually. If you're unsure what's permitted, you can have a look at the academic integrity guidelines in the course syllabus.

In the design section, you'll see some instructions for how your implementation is expected to work. Be sure you to follow these guidelines as you're planning and implementing your solution.

Requirements

Program Execution

The `omni-q` program doesn't take any command line arguments. It reads commands from standard input and writes all of its output to standard output.

All commands start with a word, either `push`, `enqueue` or `pop`. The `push` and `enqueue` commands take parameters after the command.

The command name and its parameters can be separated by one or more whitespace characters (spaces, newlines (`\n`), horizontal tab (`\t`), vertical tab (`\v`), form-feed (`\r`) or carriage return (`\r`)). Commands aren't required to be given on a single line, and multiple commands (or parts of commands) can be given the same line. This should make it easier to parse commands, since most of the conversion specifiers used by scanf() automatically skip any type of whitespace. For example, standard input may contain commands entered like:

```
push int 25
enqueue real 33.3
```

or, they may be entered as:

```
push   int    25    enqueue    real    33.3
```

or, these same commands may be entered as:

```
push
int
25

enqueue
real
33.3
```

Push command

A push command starts with the word, `push`. This is followed by a value that should be pushed onto the front of the sequence. If the sequence was previously empty, it adds the given value as the only value in the sequence.

A value is given by a word that indicates its type, either `int` for integer values, `real` for floating point numbers, or `string` for strings. Each of these words is followed by text indicating the value that should be added to the front of the sequence.

The word `int` should be followed by input that can be parsed as an int by scanf(). For example: `push int 25` or `push int -192`

The word `real` should be followed by input that can be parsed as a double by scanf(). For example: `push real 75.2` or `push real 0.001` or `push real 19`

The word `string` should be followed by one or more whitespace characters, then a sequence of one or more of characters (starting with a non-whitespace character) up to the end of the line. For example, the following example would push the string, "hello world" onto the front of the sequence:

```
push string hello world
```

There could be multiple whitespace character or even a newline before the start of the string, so the following example would also push the string, "hello world" onto the front of the sequence:

```
push string
      hello world
```

There may be whitespace inside the string (although there couldn't be a newline character). These spaces inside the string should be preserved as they are entered (even if they occur at the end). So, the following example would push the string "`hello      world`", with extra spaces inside, onto the front of the sequence:

```
push string hello     world
```

String values must be non-empty, so the following command would be invalid if it occurred on the last line of input.

```
push string
```

The previous example would be valid if there were non-whitespace characters on a subsequent line. For example, the following command would simply push the string "add int 25" onto the front of the sequence.

```
push string
add int 25
```

Enqueue command

The `enqueue` command is just like the `push` command, except it adds values to the back of the sequence rather than the front. It uses the same syntax as `push` for specifying values, and if a value is enqueued when the sequence was empty, the new value becomes the only value on the sequence.

Starting from an empty sequence, the following two commands would put an integer value, 107 at the front of the sequence, with a real value of "98.6" behind it and a string value of "abc" at the back.

```
enqueue int 107
enqueue real 98.6
enqueue string abc
```

Pop command

The pop command removes the value at the front of the sequence and prints it to standard output on a line by itself. Real values are printed out rounded to three fractional digits of precision. For example, the following commands:

```
push int 12345
push real 67.8901
push string ignorance is strength
pop
pop
pop
```

would produce the following output:

```
ignorance is strength
67.890
12345
```

If the sequence is empty, a `pop` command doesn't do anything. It should just be ignored and should not produce any output (it's not an error). For example, starting from an empty sequence, the following commands would just print a value of 100:

```
pop
pop
pop
enqueue int 100
pop
```

Invalid Commands

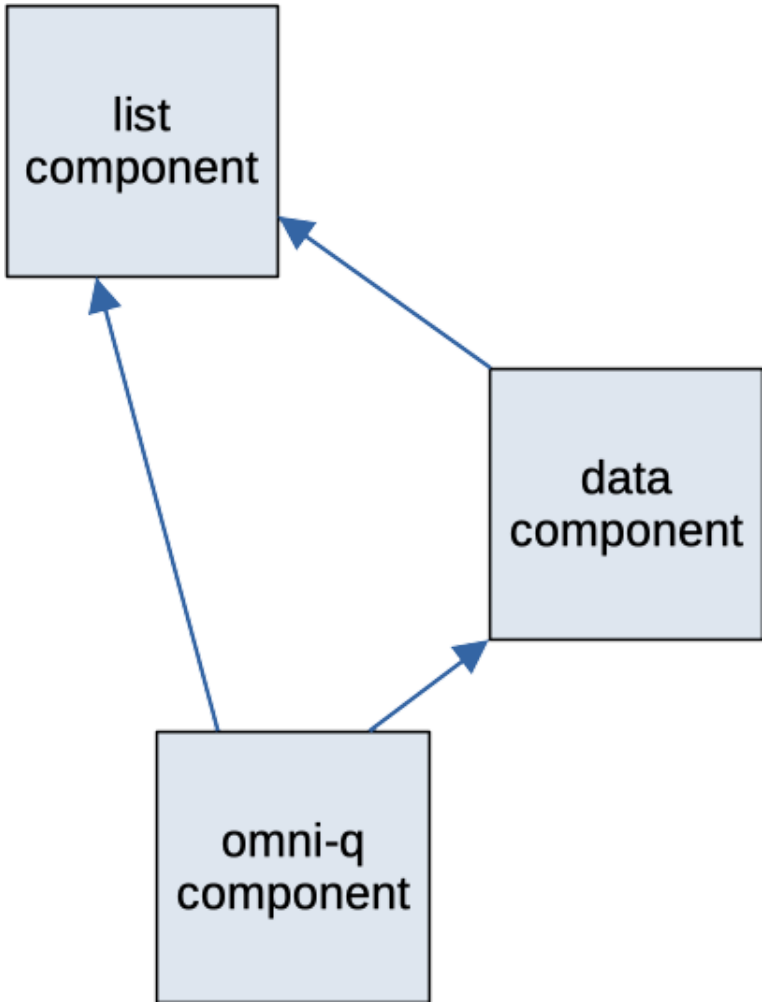
If the input contains an invalid command, the program should terminate immediately with an exit status of 1. It should not print out any error message; it should just terminate as soon as the invalid command is encountered. Output from previous, valid commands should still be printed, but no subsequent commands should be processed. For example, the following input would print the word "string" then exit with an exit status of 1.

```
push real 70
push string string
pop
bleep bloop bloam
```

A command would be invalid if it didn't match the format of one of the three commands described above. This would include commands that don't start with `push`, `enqueue` or `pop`, commands there the value isn't described as an `int`, a `real` or a `string` or commands with values that couldn't be parsed.

Design

You will implement your solution using three components. Partial header files for two of these components are included in the starter (you may have to add some code to them). The `list` component represents the sequence of values using a linked list. The `data` component provides support for the three types of values that can be stored in the list. The `omni-q` component is the top-level component. It reads and processes commands from standard input.



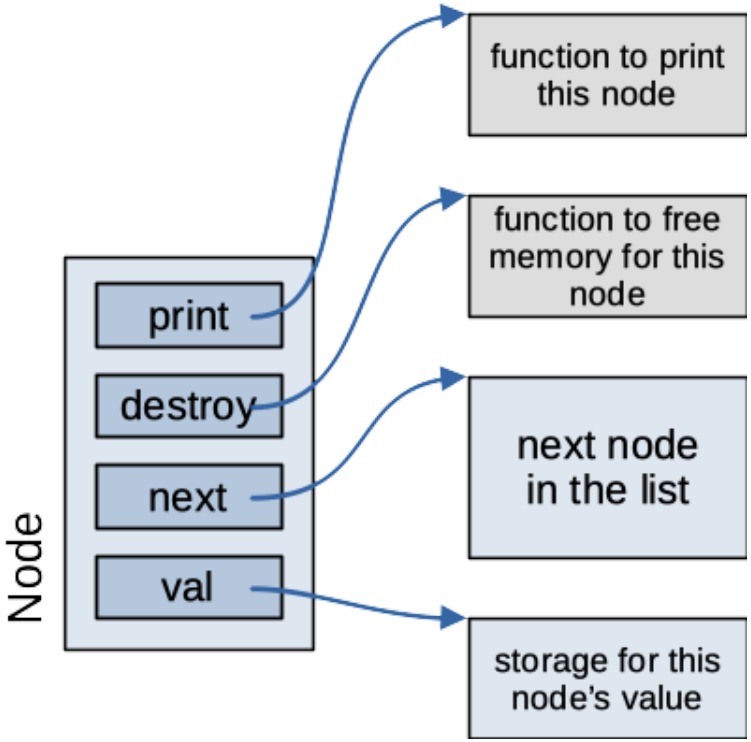
Dependency structure of the components

The dependencies among these components are shown in the figure above. The `omni-q` component can use functions and types defined in the other two. The `data` component can use features from `list`, while the `list` component doesn't depend on the other two.

Sequence Representation

The sequence of values will be represented as a linked list. As shown in the figure below, each node contains four pointers. The next pointer points to the next node in the list, like you'd normally expect for a linked list. Each node also has a void pointer pointing to memory for the value contained in that node. We're using a void pointer here so that a node can contain any type

of value. The value isn't stored in the node itself, it's stored in the memory pointed to by the `val` pointer. So, this region of memory can be as big as it needs to be in order to store the particular value stored at that node.



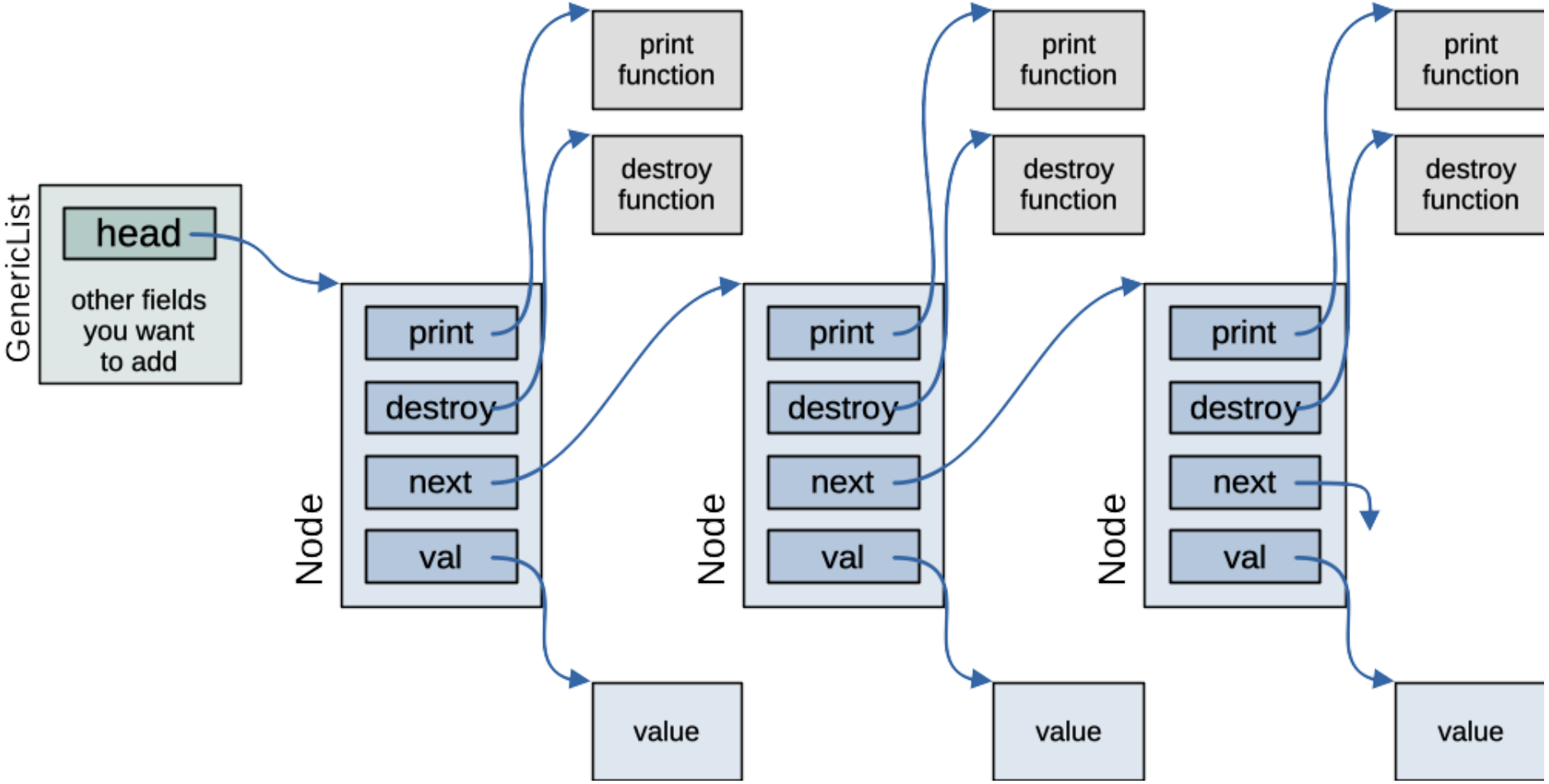
Representation of a linked list node

Each node also contains two function pointers, `print` and `destroy`. The print function is for printing the value contained in that node. Since different nodes can contain different types of values, the right way to print a value in a node depends on the type of value it's storing. Client code can call the function pointed to by a node's `print` field in order to call the right function to print the value stored in that node. As a parameter, the print function expects a pointer to the node it's supposed to print.

Like the print function, each node also has a pointer to a `destroy` function. The job of this function is to free all the memory associated the node, including memory for the node's value and for the node itself.

The list component also defines a struct named `GenericList`. This struct represents an entire list. It contains a pointer to the first node on the list. You can add other fields that you need while you're completing the assignment.

The figure below shows how a linked list is organized. An instance of `GenericList` contains a head pointer to the first node. Each node contains a pointer to the next node, with a NULL pointer indicating the end of the list. Each node contains a pointer to the value it stores, along with pointers to functions for printing that value or for freeing all the memory for that node.



Representation of a linked list using Node and GenericList

Required Functions

The `list` component should provide the following functions that may be used by the other two components. You can add more functions if you'd like, but be sure to declare a function as static if it doesn't need to be used by other components.

- `GenericList *makeList()`
  This function makes an empty list, initializing its field(s) and returning a pointer to it.

- `void push( GenericList *list, Node *n )`
  This function adds the given node to the front of the given list.

- `void enqueue( GenericList *list, Node *n )`
  This function adds the given node to the back of the given list.

- `Node *pop( GenericList *list )`
  This function removes the node at the front of the given list, returning it to the caller. If the list is empty, it should return `NULL`.

- `void freeList( GenericList *list )`
  This function frees all the memory used to store the given list, including the memory for each of the nodes on the list.

The `data` component should provide the following functions that may be used by the top-level component. You can add more functions if you'd like (but make them static wherever you can).

- `Node *makeIntNode( int val )`
  This function makes a node that contains an integer value. It allocates space to store the value and fills in pointers to appropriate functions for printing the node's value and freeing memory for the node.

- `Node *makeRealNode( double val )`
  This function makes a node that contains an double value. Like the previous function, it allocates space to store the value and fills in the `print` and `destroy` function pointers appropriately.

- `Node *makeStringNode( char const *str )`
  This function makes a node that contains an arbitrary string. Like the previous functions, it allocates space to store the string and fills in the `print` and `destroy` function pointers.

Build Automation

As in previous assignments, you get to create your own Makefile for this assignment. The default rule should build the `omni-q` target, using separate compile and link steps for building the executable. It should have a clean rule to delete the executable and any temporary files that could be easily re-created by rebuilding the program. The automated test script depends on your Makefile having a clean rule and a default target that builds the `omni-q` executable.

As in recent assignments, include the "-Wall" and "-std=c99" compile flags, along with "-g" to help with debugging.

Testing

Automated Test Script

The starter includes a test script, along with several test input files and expected outputs. When we grade your program, we'll test it with this script, along with a few other tests we're not giving you. To run the automated test script, you should be able to enter the following:

```
$ chmod +x test.sh # probably just need to do this once
$ ./test.sh
```

As with previous test scripts, this one reports how it's running your program for each test. This should help you to see how to try out your program on any specific tests you're having trouble with, to figure out what's going wrong.

Testing by Hand

If your program is failing on a test case, you can try out just that one by hand. For a successful test, like test 7, you can run it like:

```
$ ./omni-q < input-07.txt  >| output.txt
$ echo $?
0
$ diff output.txt expected-07.txt
```

Memory Error and Leaks

On successful test cases, your program is expected to free all of the dynamically allocated memory it allocates and close any files it opens. When your program exits unsuccessfully, this isn't required (since you may be exiting from within a nested sequence of function calls and you may not have access to all resources you've allocated).

Although it's not part of an automated test, we encourage you to try out your executable with valgrind. We certainly will when we're grading your work. Any leaked memory, use of uninitialized memory or access to memory outside the range of an allocated block will cost you some points. Valgrind can help you find these errors before we do.

The compile instructions above include the `-g` flag when building your program. This will help valgrind give more useful reports of where it sees errors. To get valgrind to check for memory errors, including leaks, you can run your program like this:

```
$ valgrind --tool=memcheck --leak-check=full ./omni-q < input-07.txt
-lots of valgrind output deleted-
```

With different options, you can get valgrind to check for file leaks instead:

```
$ valgrind --tool=memcheck --leak-check=full ./omni-q < input-07.txt
-lots of valgrind output deleted-
```

Test Files

The starter includes a test script, `test.sh`, and several input and expected output files to help you make sure your program is behaving correctly. Here's a summary what each of the test cases tries to do:

1. Pushes an int value onto the front of the sequence then pops it.
2. Pushes three int values onto the front of the sequence then pops all three.
3. Enqueues four int values onto the end of the sequence, then pops just three of them.
4. Pushes and enqueues six int values onto the sequence, then tries to pop seven values (so the last pop shouldn't print anything). Finally, it pushes one more int value and then pops it.
5. Enqueues two real numbers then pops them.
6. Enqueues four string values then pops them.
7. This is the example given at the start of the project description. It pushes and enqueues one of each possible type of value.
8. This test checks the command parsing, with extra whitespace in some commands and multiple commands given on some input lines.
9. This is a large test, enqueuing and pushing 10000 values of various types, popping 5000 of them, enqueuing and pushing 20000 more and then popping everything.
10. This is a test for error handling. It attempts to push an int value that doesn't parse as an int.
11. This is a test for error handling. It attempts to run a command other than push, pop or enqueue.
12. This is a test for error handling. It attempts to push a value that's not of type string, int or real.

Grading

We'll be grading your project by making sure it builds cleanly, runs correctly on all our test cases (including some we're not giving out), follows the required design and adheres to the style guide.

- Working makefile: **3 points**
- Program compiles cleanly on the common platform: **5 points**
- The interpreter behaves correctly on all test cases: **40 points**

- Follows the style guide: **12 points**
- Potential Deductions:
  - Up to **-75 percent** for not following the required design.
  - Up to **-30 percent** for exhibiting memory errors, memory leaks or file leaks.
  - Up to **-30 percent** for failing to submit required files or submitting files with the wrong name.
  - **-20 percent** for a late submission.

Getting Started

Clone your Repository

You should have already cloned your assigned NCSU github repo when you were working on project 2. If you haven't done this yet, go back to the assignment for project 2 and follow the instructions for for cloning your repo.

Unpack the starter into your cloned repo

You will need to copy and unpack the project 6 starter. We're providing this file as a compressed tar archive, starter6.tgz. You can get a copy of the starter by using the link in this document, or you can use the following curl command to download it from your shell prompt.

```
$ curl -O https://www.csc2.ncsu.edu/courses/csc230/proj/p6/starter6.tgz
```

Temporarily, put your copy of the starter in the p6 directory of your cloned repo. Then, you should be able to unpack it with the following command:

```
$ tar xzvpf starter6.tgz
```

Once you start working on the project, be sure you don't accidentally commit the starter to your repo. After you've successfully unpacked it, you may want to delete the starter from your p6 directory, or move it out of your repo.

```
$ rm starter6.tgz
```

Submission Instructions

If you've set up your repository properly, pushing your changes to your assigned CSC230 repository should be all that's required for submission. When you're done, we're expecting your repo to contain the following files in the p6 directory. You can use the web interface on github.ncsu.edu to confirm that the right versions of all your files made it.

- `omni-q.c` : the main source, written by you.

- `data.h` : header file for the data component, completed by you.
- `data.c` : implementation file for the data component, written by you.
- `list.h` : header file for the list component, completed by you.
- `list.c` : implementation file for the list component, written by you.
- `Makefile` : Makefile for the project, written by you.
- `input-*.txt` : Input files for testing, provided with the starter.
- `expected-*.txt` : expected output files, provided with the starter.
- `test.sh` : automated testing script, provided with the starter.
- `.gitignore` : .gitignore file for this project, provided with the starter.

Pushing your Changes

To submit your project, you'll need to commit your changes to your cloned repo, then push them to the NCSU github. Project 2 has more detailed instructions for doing this, but I've also summarized them here.

Whenever you create a new file that needs to go into your repo, you need to stage it for the next commit using the `add` command. You should only need to add each file once. Afterward, you can get git to automatically commit changes to that file:

```
$ git add some-new-file
```

Then, before you commit, it's a good idea to check to make sure your index has the right files staged:

```
$ git status
```

Once you've added any new files, you can use a command like the following to commit them, along with any changes to files that were already being tracked:

```
$ git commit -am "<meaningful message for future self>"
```

Remember, you haven't really submitted anything until you push your changes up to the NCSU github:

```
$ unset SSH_ASKPASS # if needed
$ git push
```

Checking Jenkins Feedback

Checking jenkins feedback is similar to the previous projects. Visit our Jenkins system at http://go.ncsu.edu/jenkins-csc230 and you'll see a new build job for project 6. This job polls your repo periodically for changes and rebuilds and tests your project automatically whenever it sees a change.

Learning Outcomes

The syllabus lists a number of learning outcomes for this course. This assignment is intended to support several of theses:

- Write small to medium C programs having several separately-compiled modules.
- Explain what happens to a program during preprocessing, lexical analysis, parsing, code generation, code optimization, linking, and execution, and identify errors that occur during each phase. In particular, they will be able to describe the differences in this process between C and Java.
- Correctly identify error messages and warnings from the preprocessor, compiler, and linker, and avoid them.
- Find and eliminate runtime errors using a combination of logic, language understanding, trace printout, and gdb or a similar command-line debugger.
- Interpret and explain data types, conversions between data types, and the possibility of overflow and underflow.
- Explain, inspect, and implement programs using structures such as enumerated types, unions, and constants and arithmetic, logical, relational, assignment, and bitwise operators.
- Trace and reason about variables and their scope in a single function, across multiple functions, and across multiple modules.
- Allocate and deallocate memory in C programs while avoiding memory leaks and dangling pointers. In particular, they will be able to implement dynamic arrays and singly-linked lists using allocated memory.
- Use the C preprocessor to control tracing of programs, compilation for different systems, and write simple macros.
- Write, debug, and modify programs using library utilities, including, but not limited to assert, the math library, the string library, random number generation, variable number of parameters, standard I/O, and file I/O.
- Use simple command-line tools to design, document, debug, and maintain their programs.
- Use an automatic packaging tool, such as make or ant, to distribute and maintain software that has multiple compilation units.
- Use a version control tools, such as subversion (svn) or Git, to track changes and do parallel development of software.
- Distinguish key elements of the syntax (what’s legal), semantics (what does it do), and pragmatics (how is it used) of a programming language.