

CMPT 300: Operating Systems I

Project 1: Review Unix Programming Tools and Basics of the C Language

This is a very important project! Investing time in it will greatly help you in later projects. Please do the following reading in order:

1. [Unix Programming Tools](#) for information on how to use the gcc compiler, make tool, and gdb debugger. Make sure that you actually use gdb on some small C programs. Make sure that you actually use gdb on some small C programs (Read this debugging [tutorial](#)). You may want to use the [Gnu Visual Debugger \(GVD\)](#), which is a graphical interface for gdb. It is already installed on *csil-cpu*, run it using `/bin/gvd`.
2. [Essential C](#) or any similar document/book. Pay careful attention to pointers and memory management in C.

Now, do the following programming problems **using C**. All code **MUST** run on a Linux machine. We will grade your code on a Linux machine. You should create a directory for your project, such as `~/cmpt300/prj1/` and put all files related to this project in it.

1. String Manipulation Functions [45 points]

In this problem, you will develop a few string manipulation functions similar to the standard C string functions. You are **not** allowed to use any standard string functions (e.g., `strcpy`, `strlen`, etc) in your code (however, your functions may call each other as necessary). Download the file [mystring.tar.gz](#) to `~/cmpt300/prj1/` After the download, issue the command:

```
$ tar xvfz mystring.tar.gz
```

which creates the directory `~/cmpt300/prj1/mystring`. In that directory, you will find the following files:

- `mystring.h` -- header file containing the prototypes for the functions that you will develop.
- `mystring.c` -- source file in which you will do your coding.
- `test_mystring.c` -- simple test file to help you testing your code.
- `Makefile` -- this file is used by the Unix command *make* to compile your source code and build the executable files.

Start by issuing the command `make` to ensure that you got everything ready to start coding. The code should compile without a problem and you should find an executable file named `test_mystring`. Try it by issuing the command `./test_mystring`. Of course, the tests there

will fail. Now, start coding the functions in `mystring.c` one at a time and modify `test_mystring.c` to add more tests.

In general, you can assume that all pointers passed into your code will be valid, and that there will be enough space allocated for any copy operations to succeed. (We *must* assume this because in C we have no way of checking if there is enough space.) You may `#include` any extra `.h` files as required (such as `stdlib.h`).

2. Linked List Operations [55 points]

The linked list is a simple, yet powerful, data structure that appears-- in one way or another--in almost all reasonable-size programs that you will encounter in your career. In this problem, you will implement a number of functions that create and manipulate a linked list. A node in the list is defined as:

```
struct nodeStruct {
    int item;
    struct nodeStruct *next;
};
```

You must implement at least the following functions:

```
/*
 * Allocate memory for a node of type struct nodeStruct and
 * initialize
 * it with the value item. Return a pointer to the new node.
 */
struct nodeStruct* List_createNode(int item);

/*
 * Insert node at the head of the list.
 */
void List_insertHead (struct nodeStruct **headRef, struct
nodeStruct *node);

/*
 * Insert node after the tail of the list.
 */
void List_insertTail (struct nodeStruct **headRef, struct
nodeStruct *node);

/*
 * Count number of nodes in the list.
 * Return 0 if the list is empty, i.e., head == NULL
 */
int List_countNodes (struct nodeStruct *head);

/*
```

```

    * Return the first node holding the value item, return NULL if
    none found
    */
    struct nodeStruct* List_findNode(struct nodeStruct *head, int
    item);

    /*
    * Delete node from the list and free memory allocated to it.
    * This function assumes that node has been properly set to a valid
    node
    * in the list. For example, the client code may have found it by
    calling
    * List_findNode(). If the list contains only node, the head of the
    list
    * should be set to NULL.
    */
    void List_deleteNode (struct nodeStruct **headRef, struct
    nodeStruct *node);

    /*
    * Sort the list in ascending order based on the item field.
    * Any sorting algorithm is fine.
    */
    void List_sort (struct nodeStruct **headRef);

```

Each function (other than sort) should not alter the item value of nodes in the linked list. For example, to insert a new node at the head of the list, a new node must be linked in at the front, rather than all the item values shifted. (This does not apply to the sort function, which may alter item values.)

Note that `struct nodeStruct **headRef` in the functions above enables you to modify the memory location referred to by the variable `headRef` (e.g., you can code: `*headRef = node`). This is important to handle boundary conditions such as inserting the first node in the list or deleting the last node in the list. In such cases, you would need to change the head of the list itself, it is why we pass pointer to the pointer. You can find more information on linked lists [in this document](#).

Here is how you should structure your source code:

Create a directory called **~/cmpt300/prj1/list** under which create the following files:

1. `list.h` -- contains the definition of `struct nodeStruct` and the function prototypes. No head or tail pointer; they are just in the application (such as `main.c`).
2. `list.c` -- contains the implementation of the above functions.
3. `test_list.c` -- to test your code, contains the `main()` function. [Here is a simple test_list.c](#) to start you off.
4. `Makefile` -- you can start with the Makefile of the previous problem and modify it.

Test your code very carefully. Your code will be run through the instructor's test which don't respond well to seg-faults!

In general, you can assume that all pointers passed into your code will be valid.

What to Submit and How

- Make sure that your files are organized as follows:
 - prj1/mystring -- everything related to the mystring part
 - prj1/list -- everything related the linked list part
- Change to each of your folders and issue the command `make clean`. This will remove all object files as well as all output and temporary files
- Change to your prj1 folder:
`$ cd ~/cmpt300/prj1`
- Then, issue:
`$ tar cvf prj1.tar *`
which creates a tar ball (i.e., a single file) that contains the contents of the folder.
- Compress your file using gzip:
`$ gzip prj1.tar`
- Submit via [CourSys](#)