ECE 220 Summer 2017 Final Exam

On-campus and Western Hemisphere

- Absolutely no interaction between students or any external help is allowed!

- You are not allowed to use any applications on your desktop outside the Exam VM.
- You can use any applications inside the Exam VM.

- You are permitted **one** double-sided page of **hand-written** notes.

- Read entire problem description before starting to work on the solution.

- **IMPORTANT: Once done, you must commit your work to Subversion:**

```
cd ~/final
svn commit -m "I am done"
```

- To verify your submission, point web browser inside the Exam VM to the following page: **https://subversion.ews.illinois.edu/svn/su17-ece220/NETID/final** where NETID is your actual NetID.

- If your attempt to commit/verify fails, make sure you are still connected to the UIUC network via the VPN client. If your connection to UIUC network gets lost during the exam, you must reconnect again.

Good luck!

Do not forget to commit your work!

Part 1: C to LC-3 with run-time stack & linked data structures (20%)

Convert the following function from C to LC-3 assembly language. This function recursively counts number of nodes in a binary tree whose value is larger than its parent node's value.

```
int CountNodes(node *nd, int parent_value)
{
    int count = 0;

    if (nd == NULL) return 0;

    if (nd->value > parent_value) count = 1;

    count += CountNodes(nd->left, nd->value);
    count += CountNodes(nd->right, nd->value);

    return count;
}
```

The tree node data structure is defined as follows:

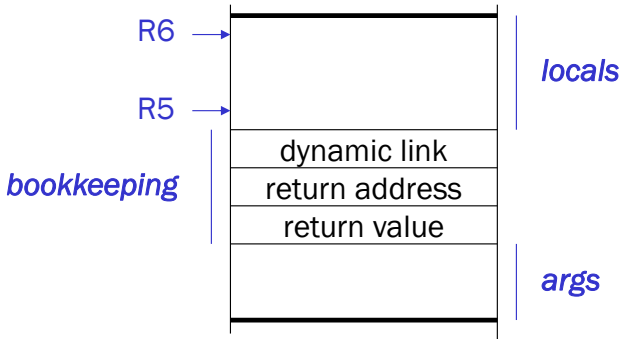
```
struct s_node
{
    int value;
    struct s_node *left;
    struct s_node *right;
};
typedef struct s_node node;
```

Remember that structures merely consist of simple data types where one element always comes after another in memory. Also, we assume each basic data type takes only one memory position. That is, if `nd` points to an address of x6000, the data `value` is contained within x6000, the `left` pointer address is contained within x6001, and the `right` pointer address is at x6002.

Write your answer in `part1/count.asm` file. The file is already pre-filled with most of the assembly code and you only need to add a few lines marked with *IMPLEMENT ME* to complete it. **Add your code segments between labels GxS and GxE only.** For example, code that implements "count = 0;" statement should be inserted between G1S and G1E labels. This is necessary to automate code grading.

Note that only the subroutine implementation is provided and there is no code to test it. Make sure your code assembles without errors!

As a reminder, function activation record format used in this class is as follows:



Do not forget to commit your work!

Part 2: C++ classes (25% (+2% extra))

In this problem, you will implement `circle` class:

```
class circle
{
  protected:
    float x, y; // center
    float radius;  // radius

  public:
    circle() { x = y = 0.0; radius = 1.0; }
    circle(float cx, float cy) { x = cx; y = cy; radius = 1.0; }
    // additional constructors /* IMPLEMENT ME */
    // set methods /* IMPLEMENT ME */
    // get methods /* IMPLEMENT ME */
    // compute methods /* IMPLEMENT ME */
    // operators /* IMPLEMENT ME */
    // I/O (already implemented)
    friend ostream& operator<<(ostream& os, const circle& b);
};
```

The class should have these additional constructors, which you need to declare and implement:

```
    circle(float cx, float cy, float r);                    (+1pts)
    circle(const circle& c); // copy constructor            (+4 pts)
```

It should have these methods for setting object values, which you need to declare and implement:

```
    void setX(float);                                       (+1 pt)
    void setY(float);                                       (+1 pt)
    void setRadius(float);                                  (+1 pt)
```

It should have these methods for getting object values, which you need to declare and implement:

```
    float getX();                                           (+1 pt)
    float getY();                                           (+1 pt)
    float getRadius();                                      (+1 pt)
```

It should have these methods for computing circle properties, which you need to declare and implement:

```
    float computeArea();       /* area = PI * r^2 */        (+3 pts)
    float computePerimeter();  /* perimeter = 2 * PI * radius */   (+3 pts)
```

It should have these operators, which you need to declare and implement:

```
// returns new circle with radius set to current circle's radius * scalar
    circle operator* (float scaler);                        (+4 pts)
// compares radii of two circles and returns true or false
    bool operator> (const circle& b);                       (+4 pts)
```

Write your implementation in `part2/circle.cpp` file. Included `main()` function already has the code necessary to test all methods; **you can modify it** as needed. To compile, type `g++ circle.cpp`.

To receive extra credit, implement error checking in just one of the methods where you think it makes sense. When an error is detected, your program must terminate immediately. *Hint: see commented out code at the bottom of* `main()`. If you decide to implement this extra credit code, you must leave uncommented the relevant test code in order to verify and grade your implementation.

Do not forget to commit your work!

Part 3: Linked data structures (25%)

Given a linked list with nodes defined as

```
typedef struct node_t
{
    int data;
    struct node_t *next;
} node;
```

you are asked to implement the following two **recursive** functions:

```
int get_node_count_rec(node *head);                    (10%)
```

This function takes as an argument a pointer to the *head node* of a linked list and counts the number of nodes in the linked list which have data value larger than previous node's value.

Example: given a linked list {1, 2, -4, -5, 6}, the answer should be 2.

```
int remove_negative_nodes_rec(node **head);            (15%)
```

This function takes as an argument a pointer to the *address of the head* of a linked list and removes all nodes with negative data values in the list. It also counts and returns the number of removed nodes.

Example: given a linked list {1, 2, -4, -5, 6}, the resulting list should be {1, 2, 6} and the return value should be 2.

Note that both functions should be implemented as recursive functions. If your implementations are not recursive, you will receive 50% of the grade for each non-recursive function.

You are given the following three files and the `Makefile` in `part3` folder:

- `llist.c` contains the functions for you to implement.
- `llist.h` contains the function prototypes for the linked link functions. **Do not modify this file.**
- `main.c` contains the `main` function for testing. **There should be no need to modify it.**

To compile, type `make`. To run, type `./llist`.

To test for memory issues, you may use `valgrind`, e.g.:

```
echo "5 -1 -2 -3 -4 -5" | valgrind ./llist
```

To debug, feel free to use `gdb`.

Do not forget to commit your work!

Part 4: File I/O, dynamic memory allocation, and strings in C (30%)

Implement a program that reads a text file line-by-line and saves it into another file in the reversed order. For example:

Input file (data.txt)	Output file (output.txt)
8 string 1 string 2 another sting 3 yet another string 4 some more text 5 and some more more text 6 still a bit more 7 very long string of text here 8	8 very long string of text here 8 still a bit more 7 and some more more text 6 some more text 5 yet another string 4 another sting 3 string 2 string 1

File format

Line #	Content
1	Number of strings in the file, N (8 in the above example)
2	String 1 ("`string 1`" in the above example)
	….
N+1	String N ("`very long string of text here 8`" in the above example)

You can assume that strings cannot be longer than `MAX_STRING_LEN` and the file contains no more that MAX_LINES lines of text, both defined in `file.h`.

Functions to implement in `file.c`

```
file_data *read_file(char *filename);                    20%
int write_file(file_data *data, char *filename);         10%
```

`read_file` function takes file name as an argument, reads data from the file, and returns pointer to a dynamically allocated structure that contains loaded data. It should return NULL if memory allocation or file operations fail.

`write_file` function takes pointer to the data structure created by `read_file` and a file name as arguments and writes data to the file in reversed order. The first record should be the number of strings in the file. The function should return 1 on success, or 0 otherwise.

You are provided with `free_memory` subroutine which is used to clear memory before exiting from the program. Study its implementation in order to understand how to properly allocate memory in `read_file` function.

You are given the following three files and the `Makefile` in `part4` folder:

- `file.c` contains functions for you to implement.
- `file.h` contains the function prototypes and data declaration. **Do not modify this file.**
- `main.c` contains the main function for testing. **There should be no need to modify it.**
- `data.txt` is a sample input file. **Do not modify it.**

To compile, type `make`. To run, `type ./file data.txt output.txt`. The results will be saved in output.txt file, which you can open to see if it is correct.

Do not forget to commit your work!