Project 9, Program Design

1. (60 points) Modify Project 8 so that the program is split into three source files and two header files.

    1) Put all functions related to operations on the list of requests into `request.c`
    2) Create a header file named `request.h` that contains `struct request` declaration and prototypes for the functions in `request.c`. The header file should enclose the contents of the header file in an `#ifndef-#endif` pair to protect the file.
    3) Put the `read_line` function in a separate file named `readline.c`.
    4) Create a header file named `readline.h` that contains a prototype for the `read_line` function. The header file should enclose the contents of the header file in an `#ifndef-#endif` pair to protect the file.
    5) `rec_classes.c` contains the main function.
    6) Include appropriate header files in the source files.

2. (40 points) Write a makefile to build the program on **student cluster**. The makefile should contain the following rules:

1) Build `readline.o` by compiling `readline.c`
2) Build `request.o` by compiling `request.c`
3) Build `rec_classes.o` by compiling `rec_classes.c`
4) Build `rec_classes` by linking `readline.o, request.o, and rec_classes.o`

Each rule should include the name of the target file, dependencies among files, and the command to be executed. The makefile should name the executable file for the program `rec_classes`.

Before you submit:

1. (part 1) Compile with the following command and test with `try_rec`

```
gcc –Wall request.c readline.c rec_classes.c
./try_rec
```

2. (part 2) Be sure your `makefile` contains the information necessary to build the roster program. Test your makefile:

```
make rec_classes
./rec_classes
```

3. Put `request.c, request.h, readline.c, readline.h, rec_classes.c, try_rec, waiting_list.txt (the original file)` and `makefile` in a zipped folder. Submit the zipped folder on Canvas.

Grading

Total points: 100

1. A program that does not compile will result in a zero.
2. Runtime error and compilation warning 5%
3. Commenting and style 15%
4. Functionality 80%:
   Part 1: Program divided into appropriate source files and header files.
   Part 1: Source files include appropriate header files
   Part 1: Header files protected using ifndef-endif
   Part 2: makefile implemented the way specified

Programming Style Guidelines

The major purpose of programming style guidelines is to make programs easy to read and understand. Good programming style helps make it possible for a person knowledgeable in the application area to quickly read a program and understand how it works.

1. Your program should begin with a comment that briefly summarizes what it does. This comment should also include your **name**.
2. In most cases, a function should have a brief comment above its definition describing what it does. Other than that, comments should be written only *needed* in order for a reader to understand what is happening.
3. Information to include in the comment for a function: name of the function, purpose of the function, meaning of each parameter, description of return value (if any), description of side effects (if any, such as modifying external variables)
4. Variable names and function names should be sufficiently descriptive that a knowledgeable reader can easily understand what the variable means and what the function does. If this is not possible, comments should be added to make the meaning clear.
5. Use consistent indentation to emphasize block structure.
6. Full line comments inside function bodies should conform to the indentation of the code where they appear.
7. Macro definitions (#define) should be used for defining symbolic names for numeric constants. For example: **`#define PI 3.141592`**
8. Use names of moderate length for variables. Most names should be between 2 and 12 letters long.
9. Use underscores to make compound names easier to read: **`tot_vol`** or **`total_volumn`** is clearer than totalvolumn.