MAT 4370: Programming Assignment 11

Due: Monday, April 1

The Big Picture

Your task is to produce a software utility modeled after the Unix `uniq` program. Your program doesn't need to do everything the Unix utility does — you will end up with a "poor man's version," which we will name `pmuniq`. Here's what I hope you will learn by completing this exercise:

- How to process command-line arguments.

  To process command-line arguments, your program will use the `getopt()` function found in the Standard C Library. Examples of how to use this function may be found on the course website.

- How to use the C `struct`.

  You will also be asked to use C's structure type to summarize the command-line arguments given to your program. See Chapter 16 of our textbook for more information about the `struct` data type.

- How C programs can be subdivided into separate compilation units.

  You are already familiar with this idea; completing this exercise will provide reinforcement.

- Appreciate how to validate command-line arguments and perform essential "sanity tests."

  Users make mistakes. When we implement utilities, we must adopt the point of view that our program might be given incorrect inputs. Planning for the validation of command-line arguments takes a conscious effort.

Requirements

Requirement 1: Syntax

The syntax for your program is as follows:

```
pmuniq [ -c | -d | -u ] [ inputfile ]
```

The square brackets in this syntax show optional components and the vertical bar can be read as "or." Here are some legal ways to invoke this program:

```
pmuniq
pmuniq -c
pmuniq -d
pmuniq -u
pmuniq myfile
pmuniq -c myfile
pmuniq -c -c -c myfile
```

The first few examples do not specify an input file. For these situations, `pmuniq` should assume the "standard input" file is to be processed. The meaning of the three optional switches is identical to the Unix `uniq` command.

The last example is a little bit silly, but it should not be considered an error. It's simply asking for the `-c` option in a redundant way.

There are a variety of ways to incorrectly specify an execution of this program, including these:

```
pmuniq -d -c myfile
pmuniq -x myfile
pmuniq -c myfile anotherfile
pmuniq missing
```

(In the last example, assume the file `missing` does not exist.) Your program will need to process the given command-line arguments and validate them to see if they are correct. Ensure that all error messages are written to the "standard error" file. When in doubt as to what an error message should look like, follow the pattern of errors given by `uniq`.

Requirement 2: Memory Efficiency

Lines in a text file can be arbitrarily long. Therefore, use the `read_line()` function from the previous assignment. However, your program may **not** store more than a small number of lines — do not use an array of strings and do not use `read_lines()`. Since `pmuniq` only needs to check for adjacent lines, it would be very wasteful of memory to store the entire contents of the file in memory.

Your program should have no memory leaks. You will need to keep this in mind as you design your program. Use Valgrind during your testing.

Requirement 3: Modularity

Your program should be subdivided into files, as follows:

`Makefile`	minimum targets: `pmuniq` and `clean`
`util.h` and `util.c`	as in assignment 10
`arg.h` and `arg.c`	for argument processing
`pmuniq.c`	main program and possible support functions

Processing Command-Line Arguments

For a correct invocation of `pmuniq`, there are two important facts — the name of the input file and the requested option, if any. This information is to be stored in a variable of type `struct args_t`, declared as follows:

```
struct args_t{
    char option;
    char *inputFile;
};
```

Observe the trailing semicolon above!

If one of the legal options was given on the command line, `option` will store that letter. If no options were given, then we can use any other letter, such as `x` to indicate this. Similarly, `inputFile` is used to store the name of the desired input file, as given on the command line. If no input file was given, use `NULL` to indicate this fact.

Include a function, `process_args()`, which will parse the command-line arguments, check for usage errors, and fill in the two members in an `args_t` structure. The prototype for this function is:

```
void process_args(int argc, char *argv[ ], struct args_t *argument);
```

If errors are detected, issue an appropriate message and terminate the program. The files `arg.h` and `arg.c` will hold the relevant definitions for argument-related processing.

Desired Results

As mentioned earlier, `pmuniq` should act like `uniq` in many respects. To get a better sense of what `uniq` does, first take a look at its manual page. Then consider the following simple `testfile`:

```
alpha
beta
beta
beta
gamma
delta
delta
```

Create a file with this content, then experiment with the following commands:

```
uniq testfile
uniq -c testfile
uniq -d testfile
uniq -u testfile
```

Incremental Development and Advice

- Respect the problem. Although it is relatively "easy," if you wait until the night before it is due, you might be unpleasantly surprised. As I'm sure you know, computer programming is often the source of such surprises — syntax errors, incorrect behavior, and all types of peculiar problems.
- Your first steps should not involve writing C code. Instead, step back and think about what steps are needed to get the desired output. Sketch a solution using pseudo-code, making use of higher-level thinking.
- Work in small sections. Identify something much smaller than the final result and work on just that much. You could, for example, ignore the command-line processing at first and just "hard-code" the file name and option to use. Alternately, you may choose to focus on just the command-line processing aspect, ignoring how to determine the unique lines. You may find it helpful to build small "throw-away" programs which allow you to experiment with `getopt()` and C's `struct` type.

What to Submit

Before submitting your work, ask yourself these questions:

- Does `make clean` put your directory in a pristine state? (In other words, only source code remains?)
- Does `make pmuniq` compile (without warnings) all the required components?
- Does Valgrind claim that your program is free of memory leaks?

If so, put all the required files in a directory named `hw11` and submit in the usual manner.