Date due: Wednesday, February 16, 11:59:59 p.m.

1 Introduction

This is a relatively short project (in terms of the amount of code to be written) intended to use some additional features of C and UNIX that Project #1 did not require– input, reading until the end of the input, simple use of arrays, input redirection and UNIX pipes, and the UNIX `diff` command, which was explained recently in discussion section. However, the project might involve some unexpected complexities, so you need to start working on it **right away**.

Carefully read the course project style guide on ELMS (under Administrative and resources). It describes what good programming style is considered to consist of for projects. Also **carefully read the constraints in Appendix C below, because you are only allowed to use certain C language features in writing the project, and will lose significant credit for using any other parts of the language.** As in Project #1 you can **lose credit** for submitting the project too many times (details are also in Appendix C).

Due to the size of the course it is not feasible for us to be able to provide project information or help via email/ELMS messages, so we will be unable to answer such questions. However you are welcome to ask any questions verbally during the TAs' office hours (or during, before, or after discussion section or lecture if time permits).

Note that if you put your code on a website (like GitHub, SourceForge, PasteBin, etc.) for others to access, you will be forwarded to the Office of Student Conduct.

2 Project description

As mentioned above, the project style guide handout describes what good programming style consists of for projects in this course. A significant part of your grade for this project will be based on your style. One aspect of style the handout describes is that none of your program lines should exceed the standard terminal or printer width of 80 characters. Although you could visually inspect your code to ensure this, having a program to check it for you would be much more convenient. You will write that program in this project. To avoid losing credit for style you should run it on **itself– and on your code in all future projects as well**.

In most projects in this course you will write functions that we will call from our main programs (our tests of your code), as in Project #1. This project is an exception– you will write a standalone program for it. Actually, you will write two different standalone programs, `checklength.c` and `tabexpand.c`, which could be used separately or together.

2.1 The line length check program `checklength.c`

As mentioned, this program will check for lines that are more than 80 characters. First, here are concepts and the terminology we use. A line is a sequence of zero or more characters terminated with the special *newline character*. The *size* of a line refers to the number of characters in the line. The newline character indicates where the line ends, but it is **not** part of the line's contents, so it does not count towards its size. The *length* of a line refers to the number of spaces that it occupies when printed. If a line has no tab characters its length equals its size. The length of a line that contains tab characters is defined in Section 2.2 below.

Your `checklength.c` program needs to do the following:

- Read **one input line at a time** and process it as described next. As each line is being read, store its characters into a **one–dimensional** array that is large enough to store just one input line, then process that one line. (Do **not** try to read and store all of the lines in the entire input at once, just read and process one line at a time.)
- Determine the size of the line, either during or after reading the line.
- For the line that was just read, print the following on one output line (with no spaces other than what's described):
  - In the first column or character position print either a single blank space character if the size of the current line that was just read was **80 or less**, or a single uppercase `X` if the line's size was **more than 80**.
  - In either case, then print one single blank space to the output.
  - Then print the size or number of characters in the current input line. No input line will ever have more than 999 characters before its terminating newline, so its size can be at most three digits. As a result, the line's size should be printed in a field of **exactly** 3 places, padded on the left with **blank spaces** if it is less than 3

digits. (Note that this formatting can **trivially** be performed in C with the appropriate `printf()` formatting options, explained in a recent discussion section.)

Note that its terminating newline indicates where an input line **ends**, but it is **not** part of the line's contents, so it does not count as one of the characters in the line when its size is printed.

A line's size can be determined either while reading it, or after it has been read and stored but before it is printed.

– Following the line's size a colon should be printed, followed by another single blank space, then the characters of the line, in the same order they appeared in the input, ending with a newline character.

The actual input line as printed will always begin in the eighth column or output position, because the space or `X`, the following space, the line size, colon, and following space all occupy the first seven positions.

Your program's input could be anything– it doesn't even have to be a C program. But if you run it reading its own code, it will tell you whether it has any lines longer than 80 characters, which you would lose credit for during grading.

Here is an example of the output that should be produced for a single input line exactly 85 characters long, with blank spaces shown as ␣. The public test outputs (discussed below) have more examples of the expected output format.

```
X␣␣85:␣This␣is␣a␣line␣with␣85␣characters.␣␣␣(Words␣were␣chosen␣carefully␣to␣have␣exactly␣85.)
```

2.2 Tab characters

As described above the program is straightforward, although you may encounter a few issues due to differences between C and Java that the project is intended to bring up. But tab characters make it slightly tricky, because a tab character occupies anywhere between 1 and 8 spaces when printed. Output devices act as if there is an invisible tab stop at every eighth character position (the eighth position, sixteenth, etc.) Printing a tab character causes the output to skip at least one space and stop just **past** the next tab position. A printable character immediately after the tab will appear at that position. The length of a line that has one or more tabs is the **number of spaces the line would occupy when printed**– each tab would contribute between 1 to 8 spaces to its length, but only 1 to its size. Some examples:

- The size of the line `fat\tcat` (where `\t` is the escape sequence indicating a tab character) is 7 but its length is 11. When printed, `fat` occupies the first three positions and the tab character causes `cat` to begin at the ninth position and occupy the ninth through eleventh positions.
- The length of the line `hi\tgood\tbye` is 19: `hi` occupies two positions, the tab advances to the next tab stop so it occupies six positions, `good` is four positions, the next tab occupies four positions, and `bye` occupies three positions.
- The length of `\t\toctopus` is 23. The first tab advances to the eighth position, the second tab advances to the sixteenth position, and the `o` would be at the seventeenth position, the `c` at the eighteenth, etc.
- A tab character always appears as if it occupies at least one space. Suppose a line is `hedgehog\tZ`. After `hedgehog` is printed the output device will be at the ninth position, but the `Z` cannot appear there, because if it did the tab would not be visible at all. Instead the tab causes printing to advance eight spaces so the `Z` would be at the seventeenth position, and the line's length is 17.

For a line with one or more tabs, the number of spaces that are advanced for each tab depends on the length of the part of the line before that tab (which may itself contain tabs).

Note that although a tab character may **appear** to occupy multiple spaces in an output line, your `checklength.c` program must print characters **exactly as they were read from the input**, so any tabs **must** be printed **as tabs**. **Do not print spaces instead of tabs.** (We could have said to do this but it would have involved some complexities that are not immediately apparent; in any event, the submit server is expecting all tabs to be printed as tabs to say that your output is correct. However, see the next section.) And for `checklength.c`, a tab just counts as one character towards of a line's length.

2.3 The tab character program `tabexpand.c`

We want to be able to use `checklength.c` to let us know how programs would appear when displayed, even if some of their lines contain tabs. We can achieve this, without changing `checklength.c`, by writing another program, `tabexpand.c`, which takes each line of its input (with zero or more tabs) and produces output in which each tab is replaced by the appropriate number of spaces. For input that contains tabs, the output of this program will be exactly

how the input would appear on a screen or printer, but the size of each line, i.e., the number of characters that it contains, will now be exactly the same as its length, or appearance when printed, once tabs are changed to spaces. To revisit the examples above:

- If a line consists of `fat\tcat`, `tabexpand.c` should print `fat` followed by five blank spaces, then `cat`.
- For the line `hi\tgood\tbye`, `tabexpand.c` should print `hi`, six blank spaces, `good`, four blank spaces, then `bye`.
- And for the line `\t\toctopus`, `tabexpand.c` should print sixteen blank spaces and then `octopus`.

Note that all tab characters in a line must be replaced by the appropriate number of spaces, based on the description above, whether a tab appears at the beginning of a line, in the middle, at the end, or whether a line has consecutive tabs.

Other than tab characters, all other characters in the input of `tabexpand.c`, including newlines, should just be printed to its output unmodified. `tabexpand.c` only replaces tab characters with spaces.

2.4 Assumptions and guarantees about the input

- There can be **zero or more** lines in the input of either program. If the input to either program is empty it will just not produce any output at all,
- You are guaranteed, for both programs, that if their input is nonempty, then each input line– even the last line– will always end with a newline character. Both programs should print a newline at the end of every line of output, so the number of lines of their output will be the same as the number of lines in their input.

  When you create (nonempty) input data files of your own with Emacs for use in testing your program, be sure to press return at the end of the last line you type, so it will also end in a newline. Our input data may contain any printable characters appearing on a standard US keyboard, but for simplicity we will avoid nonprintable or whitespace characters other than spaces, tabs, and newlines.
- As mentioned above, any line in either program's input may have more than 80 characters, but you may assume there will never be more than 999 characters (any of which may be tabs) prior to the terminating newline character of any input line. This applies to both programs (neither one will ever have an input line longer than 999 characters before the newline).

  (Since any– or even all– of the 999 characters in an input line can be tabs, this means that the maximum length of a line could be more than 999. However, `checklength.c` does not print the length of a line, it always just prints the size of a line, meaning its number of characters without treating tab characters specially. If the user wants to know the length of input lines they can run `tabexpand.c` on it before running `checklength.c`.)
- Although you are guaranteed that no input line will have more than 999 characters before its terminating newline, there is **no limit** to the number of lines that can appear in either program's input. So do **not** try to read all of the lines in the input at one time! Both programs should just read and process **one line at a time**. (Actually `tabexpand.c` could even read and process only one character at a time if you prefer, but the important point is that it should not try to read the entire input at once and then process it all.)
- Note that `checklength.c` may see input containing tabs. (The user is not forced to run `tabexpand.c` on input before running `checklength.c` on it.) If `checklength.c` sees any tab characters they just count as **one single character** in the size of the line, although they may **appear** to occupy more space when printed.

3 Development suggestions

It is strongly recommended that you develop your program in parts or phases, testing at each stage that what you have written does what it is supposed to so far. Although you can use different steps, here is a suggestion:

- First write your `checklength.c` program to read just one line of input and simply print it. Even if the input has more than one line, at this point it should just read and print the first line and quit. Run it using UNIX input redirection (which is discussed in the UNIX tutorial and was illustrated in class, and which is shown below as well) with an input file that you create (running your program on input data is discussed further below), and make sure that it does this correctly. This will give you confidence that you are correctly reading basic input in C.
- Then modify `checklength.c` to repeat reading a line at a time, printing each one, until the end of the input is seen. Stop and test your program (as above, run it with input redirection) to ensure it works right before going

on. (Note that two of the lecture examples in `~/216public` on Grace illustrate reading until the end of the input; it would be wise to study the lecture examples carefully.)

- Then add code to `checklength.c` to count the length of each line and, depending on it, print either an `X` or space before the line is printed. Make sure this works right.
- Then `checklength.c` should print the line length itself, in the format specified (with the colon and space), between the `X` or space and the line itself.

  (Note that if you have tested things yourself carefully, and your code so far works right, at this point your `checklength.c` program should be passing the majority of the public tests.)
- Lastly, write `tabexpand.c`, which should be straightforward if you follow an analogous stepwise approach.

At this point your programs should be complete, so test them thoroughly on various inputs, **and** make sure you have used good style throughout in writing them– carefully read the project style guide on ELMS– before submitting.

A Development procedure

A.1 Obtaining the project files

You can obtain the project files on the Grace machines using commands similar to those in Project #1:

```
cd ~/216
tar -zxvf ~/216public/project02/project02.tgz
```

This will create a directory `project02` that contains the project files extracted from the project tarfile. You **must** have your coursework in your course disk space for this class (e.g., the `cd` command above accomplishes this), otherwise your submission **will not be accepted**. After this, `cd` to the `project02` directory and edit the file named `checklength.c`, where you should write the first program. A trivial version of this file exists in the tarfile, which just has a comment and a skeleton `main()` function. Remove the comment and write your first program there. **Do not rename the file.** Its name must be spelled and capitalized **exactly** as shown, otherwise it will not compile on the submit server. Even a small difference, like `Checklength.c` or `checklen.c`, would result in receiving **zero points** when submitting. Then write the second program in the file `tabexpand.c`. The tarfile also has an initial file with this name.

A.2 Compiling your code

Some variations in the commands to compile your programs and check their results will be needed, because in this project each program will be complete and have a `main()` function. Just use the command `gcc checklength.c -o checklength.x` to compile your first program, and `gcc tabexpand.c -o tabexpand.x` to compile the second one. These commands will name the executable versions of the programs `checklength.x` and `tabexpand.x` (assuming there were no syntax errors). (You can also run the compiler from Emacs, as your TA explained.) If you set up your account correctly in discussion section, an alias exists for `gcc` that adds the required compilation options mentioned in Project #1 to the command.

A.3 Running your programs and checking their results

The public test inputs are just text files, and your program will be run with input redirected from them. The files are named `public01.input`, `public02.input`, etc.

The submit server can check that the output of your programs is correct, but you should not submit until after you have thoroughly tested them yourself. Expected outputs for all the public tests are also included in the project tarfile, for example, `public01.input` has an associated output file named `public01.output`. It would be tedious and error–prone to have to compare your output to the expected results manually, however the UNIX `diff` utility described in a recent discussion section can do this automatically.

The typical use of the second program would be to use it to convert tabs to spaces in an input file, then send the output of that program– using a UNIX pipe (explained in the UNIX tutorial)– for the first program to read as its input. Either program could be used without the other one though.

A text file named `README` in the project tarfile gives the exact commands you should use to run your programs on the public tests. Look at this file (using `less`) to see the commands to use. Some tests will only run the `checklength.x` program, and will be run, and their results compared, using a command like this:

```
checklength.x < public01.input | diff - public01.output
```

This sends the output of the `checklength.x` executable, when run reading `public01.input`, into the `diff` command. The dash in the `diff` command means that instead of comparing two files it will compare its input– the output of the `checklength.x` program– against the file `public01.output` whose name is next in the command, and if there are any differences between them they will be displayed. `diff` will not print anything if no differences exist between your output and the correct output, meaning that your `checklength.c` program passed the test.

Other tests will test just your second program, so the commands to run them will be similar to the one above except with `tabexpand.x` in place of `checklength.x`. And other tests will run both programs, using a command like this:

```
tabexpand.x < public08.input | checklength.x | diff - public08.output
```

This will send the output of the `tabexpand.x` executable, when it is run reading the file `public02.input`, into the `checklength.x` program, which reads it and in turn sends its output into the `diff` command, which will then compare it against the file `public02.output`. As above, if `diff` doesn't print anything then the output of the `checklength.x` program is identical to the expected output for this test (in fact, both programs must have worked right), and you passed the test.

A.3.1 Viewing tab characters in input files and in the output

Because tab characters are whitespace they can't be visually distinguished from spaces, and spaces at the ends of lines can't be seen. A UNIX utility `od` ("octal dump") can help view a file containing either of these by printing the file's contents, character by character, showing the escape sequence representation of any special control characters. For example, look at the sixth public test input file `public06.input` using `less`, then examine the results of the command `od -t c public06.input`. `od` can be useful when examining your own input files, and your programs' output, in case of bugs. If you're not sure why a program is giving incorrect results you can run it with its output redirected to a file (explained in the UNIX tutorial and discussion) and look at the redirected output file's contents using `od`.

Another easy way to see where tab characters are in a file is to display the file using `cat -T` instead of `less`, for example, `cat -T public7.input`. This shows tabs as `^I`.

A.4 Submitting your programs

As before, the command `submit` from the project directory will submit your project, but **before** you submit, you **must** first make sure you have passed all the public tests, by compiling and running your code on them. **Unless you have versions of both programs that will at least compile, your code will fail to compile at all on the submit server.**

It's possible for a program to work fine on one machine (like the Grace machines), but not work at all on another machine (like the submit server), and this can occur more frequently in C, due to the nature of the language. Consequently, after you submit you **must** log into the submit server and see whether your program worked there.

To receive credit do **not** submit your project by uploading a zipfile or individual files to the submit server.

B Grading criteria

Your grade for this project will be based on:

public tests	35 points
secret tests	40 points
programming style	25 points

Style will be a significant part of your score. To know what good style for projects consists of, and to avoid losing credit, carefully read the project style guide on ELMS. Note that if you submit more than once **your last submission may not be the one that is graded** (see details in the project policies handout), so **use good style (according to the style guide) from the beginning** when you start working on the project, and **throughout all of your coding**.

Make sure none of your program lines are longer than 80 characters! If you are ready to submit you should have a working program that will tell you if you do have any lines that are too long. To avoid losing credit, use your first program to check both of them. First run:

```
tabexpand.x < checklength.c | checklength.x | less
```

Look for lines longer than 80 characters in your `checklength.c` program and fix them (make them shorter, break them up into two lines, etc.). Then run this command and do the same for your `tabexpand.c` program:

```
tabexpand.x < tabexpand.c | checklength.x | less
```

C Notes, constraints, and allowable language features

- You must implement this project using **only** the features of C that have been covered **so far in class this semester as of when this project is assigned**. We want you to practice using these features before going on to new ones. The project can be written more easily using features we haven't covered yet, but one purpose of the project is to get experience with some of the differences between Java and C that the project will illustrate if written as specified. In particular:
  - Your programs can **not** include any library header files other than `stdio.h`,
  - Your programs can **not** call any C library functions other than `printf()`, `scanf()`, and `feof()`,
  - Your programs can **not use any format specifiers other than the ones covered so far in class** (`%d`, `%c`, `%f`, `%o`, `%u`, and `%x`),
  - Even if you already know something about using strings in C, **do not attempt to use them** (because they have also not been covered), and
  - Even though Chapter 1 in the Reek text uses various C features that have not been covered in class yet, you can **not** use them if they were not also covered in class, or in the lecture slides, or lecture examples.

    (Exception: as in Project #1, you can use any C operators covered in Sections 5.1.4 through 5.1.8 of the Reek text, and any C features in Chapter 4 of the textbook, because these are also similar to Java, with the only exception being that you **cannot** use the `goto` statement mentioned in Chapter 4.)

  You will **lose significant credit** when your project is graded if you use C features that weren't covered yet. The project can be written in a relatively small number of lines of code using just features covered so far.

  Reread this entire item carefully again. Please keep in mind what it says.
- All of your code `must` be in the files `checklength.c` and `tabexpand.c`. Do not create any new source or header files for the project.
- In just a few keystrokes Emacs can reindent an entire program, so you would be certain of not losing any credit for any indentation problems. The UNIX tutorial explains how to do this.
- Because of the way the submit server works, the `main()` functions of both programs **must** end with `return 0` (which in C and UNIX means the program ran and terminated successfully).
- For this project you will **lose one point** from your final project score for every submission that you make (up through the end of the one–day late period) in excess of four submissions. You will also **lose one point** for every submission that does not compile, in excess of two noncompiling submissions. Therefore be sure to compile, run, and test your project's results **before** submitting. Hopefully everyone will check their code themselves carefully, and avoid these penalties.
- If you get an error that says "Segmentation fault" when running a program this is an indication of a fatal error. For the moment, the best way to debug this or any other type of error is to add debug `printf()` statements to your program, to print the values of the variables that are being used. Note that for technical reasons that will come up later in the course, the last thing printed by every debug `printf()` should be a newline character, to ensure the results appear correctly.
- Besides the public tests, you should write your own tests for your program! Just use Emacs to edit and save any files that you can then run your program with input redirected from. But **don't modify the public tests**, otherwise you won't be testing your code on the cases that the submit server is running. Just create your own test inputs.

- Note that the project policies handout says that all your projects must work on **at least half of the public tests** (by the end of the semester) in order for you to be eligible to pass the course. See the project policies handout for full details.
- If you are using Windows, note that if you resize the Mobaterm window the appearance of printed tabs may change after that (they could appear to take up a different number of spaces). So you may want to avoid resizing the window while writing this project.

D Academic integrity

Please **carefully read** the academic honesty section of the syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, publicly providing others access to your project code online, or unauthorized use of computer accounts, **will be submitted** to the Office of Student Conduct, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand the academic integrity requirements in the syllabus. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. More information is in the course syllabus– please review it now.

The academic integrity requirements also apply to any test data for projects, which must be **your own original work**. Exchanging test data or working together to write test cases is also prohibited.