

Arrays and Functions

This programming assignment uses many of the ideas presented in sections 5, 6, 7 and 8 of the course notes. Also be sure to carefully read through chapters 5-8 and 12 of the Dale textbook. You are advised to read those sections carefully. Read and follow the following program specification carefully.

Your score on this project will be a weighted average of the score you receive for runtime testing and the score you receive for the design document turned in for this program and for following the programming standards below. You must write at least two functions in your program, and the design for these should also be shown in the design document. See "What You Must Do" below for further information on the functions and the design document.

The Program Specification:

Word Processing

Modern word processors allow users to decide whether they want their text left justified only, right justified only, or both left justified and right justified. For example, the short paragraph below on the left is right justified only, while the example shown on the right is left and right justified:

This is a small paragraph
in which I used right
justification. It looks a
little different than we are
used to.

This is a small paragraph
in which I used right and
left justification. It looks
like what we are used to
seeing

Notice that in the left and right justified example, there is more space between words on some of the lines. This is how the line is stretched to fit the required number of columns. In this program, you will use a stretching method to format paragraphs so that they are left and right justified into a specified number of columns. The paragraphs used as input data to this program are mostly paragraphs from well-known texts, obtained with permission from Project Gutenberg (<http://promo.net/pg/>).

In this specification, we will use the following terminology:

Blank: the character literal ' '

Word: any sequence of characters, none of which is a blank

Paragraph: a sequence of two or more words, with a single blank space between each word

Space: a single, designated character that represents space added between two words

Formatted paragraph: a paragraph, where words have had spaces appended, and newline characters have been inserted as necessary so that output is left and right justified within a specified number of columns

For each paragraph that you must process, the input file will contain two components: the number of columns to make the paragraph fit into, and the actual characters comprising the paragraph. Because we are using the autograder, which does not assess horizontal layout, instead of inserting additional blanks between words, we will insert the pound character '#' to represent an added space. Thus if the paragraph you are formatting is

This is the exact one that I want.

and the number of columns is 16, the formatted paragraph would be

```
This### is## the
exact one that I
want.
```

From the above example, notice the following details:

1. Additional spaces are added one by one, between words on a line, starting at the left and proceeding to the right. If additional spaces are needed after one pass over the line, the same process is done again. This ensures that the additional space is distributed fairly evenly across the whole line, instead of all being placed between two words in the line.
2. The last line is not stretched.

Input file description and sample

Your program **must** read its input from a file named `inputPars.dat` — use of another input file name will result in massive deductions. The input file contains an unknown number of sets of data. Each data set consists of two lines. The first line contains the number of columns that the output paragraph must use. The second line is the text of the paragraph. A newline character ends the paragraph text. There is a single empty line between each data set. Below is a very small sample input file. Most input files contain very long paragraphs, and the whole paragraph would not fit on this page, since there are no newline characters within the paragraph.

```
35
A short example paragraph is necessary so that its entire text can be seen in the spec.

18
My dogs and I have fun playing agility. What do you do for fun? Maybe I don't want to know!
```

Note that you must **not make any assumptions** about the number of sets of data in the input file. Your program must be written so that it will detect when it is out of input and terminate correctly.

Reiterating the above discussion, and adding some details, you should assume the following holds for every input file and for each data set in the file:

1. The maximum length of a paragraph is 1000 characters.
2. The maximum number of columns into which the paragraph will be justified is 75.
3. The paragraph can be justified within the given number of columns. That is, every line that must be stretched will contain at least two words so that spaces can be added between them to stretch the line.
4. The number of columns will be on a single line.
5. The paragraph to process will be on the following line.
6. The end of the paragraph will be indicated by the occurrence of a newline.
7. An empty line will follow each paragraph.

Output description and sample

Your program **must** write its output to a file named `outputPars.dat` — use of another output file name will result in massive deductions. The first line of the output file should contain your name. The next line is blank. Then, for each paragraph in the input file, output two lines that together display the number of the column. After the column number lines, the formatted paragraph is output. A blank line follows each formatted paragraph. The example below is the output corresponding to the input file shown above.

```
Pamela J Vermeer

0000000001111111112222222222333333
12345678901234567890123456789012345
A## short## example## paragraph# is
necessary# so# that its entire text
can be seen in the spec.

000000000111111111
123456789012345678
My dogs and I have
fun##### playing
agility.# What# do
you# do# for# fun?
Maybe I don't want
to know!
```

What You Must Do

First, you must write a design document containing a modular/functional decomposition in outline form, from which you will write this program. Your design document must be produced using Microsoft Word or a text file format and submitted through the Curator or before the due date specified in the assignments page for your class. Don't forget to put your name on the design document, or you will not receive credit for it. Follow the layout of the design for the payroll program shown in Appendix 4 of the course notes. Note also that Appendix 5 of the notes shows an implementation of the payroll program that uses functions and arrays. Copy the text of your outline design as comments at the top of your source code file following the Honor Code Pledge.

Second, implement your design so that you have a program that meets the specifications given above. To receive full credit for this assignment, you must write and use at least two **functions** (not including `main`) in your program. At least one of these functions must have an array as one (or more) of its parameters. However, you are encouraged to use more functions if you think they are helpful. As a point of reference, the Curator solution to this assignment uses over six functions. For more information about functions, see section 7 of the class notes.

Testing

At minimum, you should be certain that your program produces the output given above when you use the given input file. However, verifying that your program produces correct results on a single test case does not constitute a satisfactory testing regimen. Be sure to test your program on all the provided data files. **Also, an easy way to make additional test files is to change the number of columns for the paragraphs in the provided sample input files.**

Incremental Development

You'll find that it's easier and faster to produce a working program by practicing incremental development. In other words, don't try to solve the entire problem at once. First, develop your design. Remember that a careful analysis of the program and development of an algorithm on paper is an extremely important part of the problem-solving method. Also, the algorithm developed in this process is the modular decomposition to be turned in, as discussed above.

When the time comes to implement your design, do it piece by piece. Here's a suggested implementation strategy for this project:

- First, write the code necessary to read the entire input file. This should be similar to the input code used in your previous programs. To test your work, include code to write what you're reading (and nothing else) to the output file. There's not much point in worrying about processing the data further until you know you're reading it correctly.
- Second, add code to print out your name to the output file.
- Third, add code to output the column counter lines. Verify that your column counter lines are correct for a variety of column numbers.
- Fourth, add code to copy characters from the paragraph into a single line. The number of characters in the line extracted from the paragraph should be less than or equal to the number of columns specified for the column. Also, the line should contain only complete words from the paragraph, and not start or end with a blank. Verify that this is correct by outputting each line as you extract it. Since you already have the column counter lines printed, you can check that your lines all fit within the required number of columns, and that you are not losing any words or characters from the paragraph.
- Fifth, add code to stretch/justify a line, following the rules of space distribution described above. Verify this is working by stretching all but the last line of your paragraph, and outputting them to the output file.

Now you have a substantially complete program. At this point, you should clean up your code, eliminating any unnecessary instructions and fine-tuning the documentation you already wrote. Check your implementation and output again to be sure that you've followed all the specifications given for this project, especially those in the Programming Standards section above. Be sure you have entered the Honor Code statement at the beginning of your program, and your modular decomposition outline. At this point, you're ready to submit your solution to the Grader.

Programming Standards

You'll be expected to observe good programming/documentation standards. All the discussions in class about formatting, structure, and commenting your code will be enforced. A copy of *Elements of Programming Style* is included with the course notes — if you don't have a copy it is strongly suggested you read the on-line edition (available from the course web page). Some specifics:

- You must include header comments specifying the compiler and operating system used and the date completed.
- Your header comment must describe what your program does; don't just plagiarize language from this spec.
- You must include a comment before each function stating its name, explaining what it does, explaining the purpose of each of the parameters and whether the parameter is used for input, output, or both, and what is returned from the function.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- You must use named constants for values that are constants in the program.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.
- You must use indentation and blank lines to make control structures like loops and if-else statements more readable.

Your submission that receives the highest score may be graded for adherence to these requirements, whether it is your last submission or not. If two or more of your submissions are tied for highest, the earliest of those will be graded. Therefore: implement and comment your C++ source code with these requirements in mind from the beginning rather than planning to clean up and add comments later.

Pledge:

As with the previous projects, each of your submissions to the Curator must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the header comment for your program:

```
// On my honor:
//
// - I have not discussed the C++ language code in my program with
//   anyone other than my instructor or the teaching assistants
//   assigned to this course.
//
// - I have not used C++ language code obtained from another student,
//   or any other unauthorized source, either modified or unmodified.
//
// - If any C++ language code or documentation used in my program
//   was obtained from another source, such as a text book or course
//   notes, that has been clearly noted with a proper citation in
//   the comments of my program.
//
// - I have not designed this program in such a way as to defeat or
//   interfere with the normal operation of the automated grader.
```

Failure to include this pledge in a submission is a violation of the Honor Code.