CSCI 301 - Requirements for Programming Assignments

Design Document

A design document describes the design of the program: what it does and how it does it. This document should be typed or word-processed. It should provide a description of the problem and enough detail about the program you have written to solve it that a reader, seeing nothing more than the design document, could substantially reproduce your program.

A design document should include at least following parts:

- An introduction
- Data structures used (such as classes, arrays and so on.)
  - If C++ classes are used, UML diagrams are required (see about UML in Section 13.17 of the Gaddis Textbook [in Materials/Content/Reference Books in our D2L site])
- Functions and their pseudocodes
- A *structure chart* of main program

Listing of code

Your source code should be formatted for good readability and should include the usual comments, including an introductory comment that summarizes the program's purpose and design and includes the author's name, the course and section numbers, and the date the project is submitted. Each function's comments must include *pre-* and *post-conditions*: what must be true of the values going in, and what the calling code can expect to be true of the values coming back, assuming that the precondition is satisfied.

For all projects, include **`javadoc`**-style comments.
(see the textbook regarding **`javadoc`**-style comments in both Appendix C and Appendix E.2.8)

User document

A user document describes *how* to *use* the program. This document should include an example of how to run the program. You must clearly instruct how to compile the program, and then how to run the program. Please be sure to describe any requirements that input data must satisfy. The grader will run your program by following this document for directions. Note that a user should *not* be told anything about how the program is implemented.

Test Document: plan, test data, and running results

The test data should be verified and be appropriate. Generally, you should have at least three sets of test data: valid values, boundary values and invalid values, to show that the program does what it is supposed to do. You will design your own test data. On the printout, annotate each data file and test to describe what feature(s) of the program the run is testing. Include the testing program source file and testing script file test data files if the testing program used them.

Summary document

A summary of what you learned from the assignment, how your solution to the problem might be extended or improved, and answers to any questions posed on the project page. This, too, should be typed or word-processed.
If you used **AI**, state how, and describe and assess your experiences.
If you worked with one or more students in this class to *independently* generate your own projects, describe and assess your collaborations.

EXAMPLES

1. **Design Document**

Introduction

Formatting text is arranging it on the screen or page in a particular way. This program is a simple text formatter: it reads an input file of text and writes the text to an output file, arranged into lines no longer than a fixed maximum length. The program considers any

block of contiguous characters to be a word and white space---blanks, tabs, ends of lines---separates words. The program breaks output lines between words.

The program reads the names of the input and output files and the output line length from the terminal.

Data Structures

The program uses only one data structure, a string (array of characters) named *s*. This array holds each word read from the input file, in turn. A program constant sets the maximum allowed value of the output line length, and the string *s* is declared to be of this maximum length; an output line may be filled by one word. Another program constant sets the minimum allowed value of the output line length. It is assumed that no word in the input file exceeds the line length the user specifies.

A variable keeps track of the current length of the output line. This variable is reset when the program begins each new output line.

Functions

The program uses three functions.

- Two read the names of the files and open them for input and output, respectively. They continue to prompt for file names until each file is opened successfully.
- Another function reads the output line length, which must fall between the bounds set by the two program constants. This function, too, continues to prompt for input until the user enters a value within those bounds. The program also calls `strlen()` from the `string.h` library and `eof()` and `fail()` from `fstream.h`

The Main Program

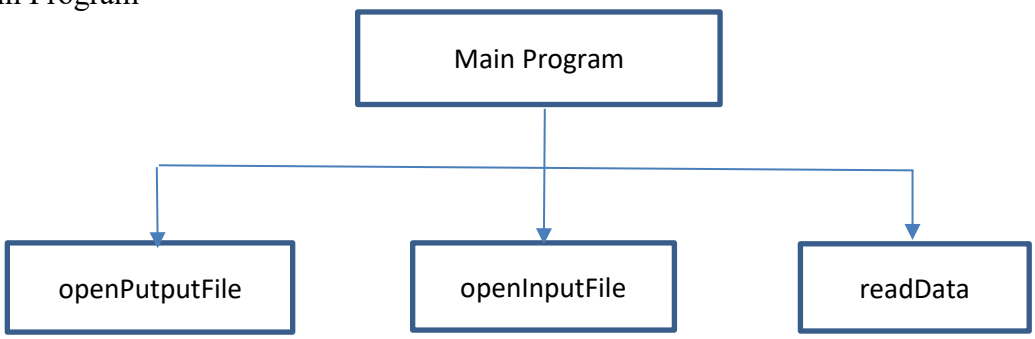
The main program calls the functions that open the input and output files and read the output line length. It then reads words (contiguous blocks of non-blank characters) from the input file into the string *s* one at a time. From *s*, it writes them to the output file.

A variable in the main program always holds the length of the current output line so far. If the word in *s* fits on the line without exceeding the output line length, it is printed, and the line length is increased by the word's length. If the next word will not fit on the current line, the program issues an end-of-line character, writes the word on the next line, then resets the line length variable to the length of that word. After each word, the program writes a blank if there is room on the line; the line length variable is incremented by one in this case.

This process continues until the input file has been exhausted, when the program closes both files and terminates.

Structure Chart of the Main Program

Main Program



2. **Code list**

This program reads an input file of text and writes an output file of the same text, formatted into lines no longer than a maximum length. The names of the input and output files and the maximum line length are read from the terminal. Functions open the data files, and continue prompting for file names until names are entered than can be successfully opened. Another function reads the maximum line length, which must fall within bounds set by two program constants.

This program reads and writes words from the input file one at a time. It keeps track of the length of the current line so far; if the next word would cause that line to exceed the maximum length, the program terminates that line, writes the word on the next line, and resets the line length. This program writes a blank after each word, except perhaps the last word on a line. A word is a string of contiguous non-blank characters, and we assume that no input word is longer than the input line.

When you have your final version of your source code, have your source code editor print the ***syntax-highlighted*** source code to a PDF file, then include that PDF file, `format.pdf`, in addition to the source file itself, `format.cpp`.
(*if you edit the source file, re-print the* PDF *file*)

3. User Document

Formatting text is arranging it on the screen or page. The program format is a text formatter. It reads a file of text and prints a new file containing the same text, but formatted into lines no longer than a maximum length. The user enters the names of the input and output files and the maximum line length, which must fall between 30 and 80. The program inserts line breaks in the white space between words, which are defined to be blocks of contiguous non-blank characters.

The program's name is **`format.cpp`**.

To compile and link it, simply enter:

```
g++ -o format format.cpp
```

To run this program, enter **format**, then respond to the program's prompts to specify the input and output files and the maximum line length. The program will continue to prompt for input should it be unable to open a file or should the entered maximum line length not fall between 30 and 80. The program assumes that no word is longer than the entered line length.

For example, if an input file called `inp.dat` contains this text:

The puzzle was invented by the French mathematician Édouard Lucas in 1883. There is a legend about a Vietnamese temple which contains a large room with three time-worn posts in it surrounded by 64 golden disks. The priests of Hanoi, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the rules of the puzzle, since that time. The puzzle is therefore also known as the Tower of Brahma puzzle.
According to the legend, when the last move of the puzzle is completed, the world will end. It is not clear whether Lucas invented this legend or was inspired by it.

If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them 264−1 seconds or roughly 585 billion years;[1] it would take 18,446,744,073,709,551,615 turns to finish.

The following exchange will run the program on this file:

```
prompt> format
  Enter input file name: inp.dat Enter
  output file name: outp.dat
  Enter an integer value between 30 and 80: 80
```

When the program terminates, the output file out.dat will contain the following:

The puzzle was invented by the French mathematician Édouard Lucas in 1883. There is a legend about a Vietnamese temple which contains a large room with three time-worn posts in it surrounded by 64 golden disks. The priests of Hanoi, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the rules of the puzzle, since that time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the legend, when the last move of the puzzle is completed, the world will end. It is not clear whether Lucas invented this legend or was inspired by it. If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them 264.1 seconds or roughly 585 billion years;[1] it would take 18,446,744,073,709,551,615 turns to finish.

Note that no line in the output file contains more than 80 characters, corresponding to the value entered in response to the program's prompt.

4. **Test Data Plan**

Your test data should contain the following sections:

a) **valid input** values and reasons to choose these values
expected results

b) **boundary input** values (if applicable) and reasons to choose these values
expected results

c) **invalid input** values and reasons to choose these values
expected results

For example, here is the test plan for the example used in this document. You don't have to use the same format:

valid input values		
	input: a number between 30 to 80	60
	input: input file name	`inp.dat`
	input: output file name	`oup.dat`
	expected output: formatted output	NO MESSAGE
boundary values		
	input: a number between 30 to 80	80
	input: input file name	`inp.dat`
	input: output file name	`oup.dat`
	expected output: formatted output	NO MESSAGE
	input: a number between 30 to 80	30
	input: input file name	`inp.dat`
	input: output file name	`oup.dat`
	expected output: formatted output	NO MESSAGE
invalid input		
	input: a number between 30 to 80	
		85
	input: input file name	`inp.dat`
	input: output file name	`oup.dat`
	expected output: formatted output	ERROR
	input: a number between 30 to 80	25
	input: input file name	`inp.dat`
	input: output file name	`oup.dat`
	expected output: formatted output	ERROR
	input: a number between 30 to 80	70
	input: input file name	`inputFile.dat`
	input: output file name	`oup.dat`
	expected output: formatted output	ERROR

5. **Summary**

In this project I implemented a program that formats input text into lines no longer than a maximum length. The program reads from and writes to both the terminal and files. Because it manipulates text, it uses a stringtype (an array of characters) and string functions from the `<cstring>` library.

Because the program uses functions to read the file names and open the input and output files, it illustrates passing file streams as parameters. Because these functions each continue to prompt for a file name when the named file cannot be opened, they also illustrate the use of the ifstream and `ofstream` member functions both called `fail()`. These functions return TRUE if a file operation, like opening a file, fails.

The program could be extended in several ways that would make it more useful as a text formatter. These include writing two blanks, rather than one, when a string ends in a period; treating blank lines in the input file as paragraph indicators and then formatting the output text to indicate the start of a paragraph; and indenting the beginnings of paragraphs a specified number of space.

This document was created by Prof. Meichsner then edited by Prof. Anda.