

Assignment 3: Printing a formatted table to a file

Due: Tuesday 10/20/09 (before class)

Your client is a small store owner who likes to see the periodic sales activity summarized in a table, which your program will produce in the output file `receipt.txt`. The user will be prompted for cost, price, item name, and item description. We will do some basic error checking: cost must be greater than zero and less than eight thousand, price must be greater than or equal to cost, but not more than twice cost. If these basic error checks fail, you will print out an informative message to both the screen **and the output file**, and then stop execution (being sure to close any open files, of course). We will compute the profit as the price minus the cost.

We will print a table line in the following format (all columns separated by 2 spaces, but no leading or trailing spaces):

- **name:** 1st 8 characters of line
- **desc:** next 40 characters
- **cost:** space for two decimal places and the maximum number of digits allowed by our input selection
- **price:** space for two decimal places and the maximum number of digits allowed by our input selection
- **profit:** space for two decimal places and the maximum number of digits allowed by our input selection

You must prompt the user for everything except profit (which you will compute), and you must prompt for them in this order: cost, price, name, desc. A negative value for **cost** is your signal to stop accepting input. **Note: input order is extremely important!**

At the end of the output, you will print two summary lines indicating the cost of the entire listed inventory, and the expected profit when it is all sold. Both total profit and cost outputs must use the same `FORMAT` statement, which will have space for up to 10 digits before the decimal and two after.

Here are some notes to help you:

- If you need to declare a variable to hold string which may be as long as 30 characters, the declaration (assuming the variable name is `mystring`) is:

```
CHARACTER (LEN=30) :: mystring
```

- When reading using the default format, Fortran assumes a space indicates the end of the string. Strings with embedded spaces must be surrounded by single or double quotes.
 - Rules for using quotes within a string are the same as we covered for printing them.

The shop owner has provided you with the data and results of several actual runs. You can get these files by issuing the following command **within a subdirectory of your own**

home area:

```
cp /home/whaley/classes/fall09/cs1073/filetab/* .
```

This will bring in the input files `bad[1,2].in` and `good[1,2].in`. These files contain the input to your program, where the `good` files are valid input, and the `bad` files are input that fail the required error checks in some way. The file `oneline.in` and `oneline.out` are the input/output with just one record.

You will also get the files `bad[1,2].out` and `good[1,2].out`. These show the precise format the shop owner expects the output `receipt.txt` to have when run with the associated `.in` file. **YOU MUST MATCH THE FORMATTING EXACTLY!**

To make this easy, we need to use a few Unix commands. You can display the contents of any file to the screen using the `cat` command, which simply outputs the contents of any supplied filenames to the screen. Therefore `cat good1.in` will show you the contents of the file `good1.in`, while `cat good1.in good2.in` is the same as `cat good*.in` which is the same as `cat good?.in`, which all concatenate the contents of both good input files to the screen.

So, now you have a way to see the contents of the required input files. However, it would be very tedious to have to retype this information every time you wished to run your program. Therefore the next thing we will discuss is Unix's pipe feature. Unix allows you to join any two command on the command line with the `|` operator, which *pipes* the standard output of the first program into the standard input of the second program. Therefore, you can save yourself the problem of typing the contents of the file `good1.in` every time you want to test your program by issuing the command (assuming you have named your executable `xfiletab`):

```
cat good1.in | ./xfiletab
```

Now, your program must take this input and use it to create the required output file `receipt.txt`. You could attempt to then compare by eye the contents of `receipt.txt` and `good1.out` by eye, but this is time consuming and error prone. Instead, you can use yet another Unix command, `diff`. This command compares two text files, and shows every line that is different. If you get no output from `diff`, then the files are exactly the same. For this assignment, you will continue modifying your program until your output matches the provided examples exactly (i.e. `diff` shows no output). To compare the above-created `receipt.out` with the associated provided file, we would issue:

```
diff receipt.txt good1.out
```

Here is an approach to solving this problem:

1. Create a program that opens a file called 'receipt.txt', and prints your name to that file, and closes the file. Run the program, make sure the file is created, and that it contains your name (using `cat receipt.txt`).
2. Edit this program so that instead of printing your name, it prints the header lines to the file. Diff your file with `oneline.out`, and be sure that your header lines match mine exactly (your data and summary lines will obviously not exist, and so will show up as different).
3. Change the program to read in all the input data items, and to print them with the appropriate format line. At this point, you can do (assuming your executable is called `xfiletab`)

```
cat oneline.in | ./filetab
```

and then compare your output to the provided `oneline.out`. Keep modifying your your format and write statements until your header and table lines match mine exactly (footer info will still be different).
4. Add a loop to repeat the read-in and table row printing. You can now compare your entire table to mine in the `good*` files.
5. Add the error checking for the input. You can now compare your answers to mine using the `bad*` files.
6. Add the logic to keep track of total cost and profit. Print answers in any format you like, but make sure that the totals are correct.
7. Make your footer output match mine exactly (you should now be able to reproduce all outputs exactly).
8. Adapt the two footer statements so that they both write using the same `FORMAT` statement.