Lab 2: Simple C Programs

Due date: Wednesday, January 12, beginning of the lab period.

Lab Assignment

Assignment Preparation

Lab type. This is an **individual lab**. Each student will submit his/her set of deliverables.

Collaboration. Students are allowed to consult their peers[1] in completing the lab. Any other collaboration activities will violate the *non-collaboration* agreement. No direct sharing of code is allowed.

Purpose. Wwe are starting to work with the main concepts of C. This lab will allow you to write a few simple C programs. Another important goal of this lab is to introduce the concepts of **debugging** and **testing**. Finally, this is the first lab, where your programming must follow the required style.

Programming Style. All submitted C programs must adhere to the programming style described in detail at

http://users.csc.calpoly.edu/∼cstaley/General/CStyle.htm

When graded, the programs will be checked for style. Any stylistic violations are subject to a 10% penalty. Significant stylistic violations, epsecially those that make grading harder, may yield stricter penalties.

Please note, that not all instructions from the link above are applicable to your Lab 2 assignments. If you do not understand the meaning of an instruction, please consult your instructor.

The following instructions are **in addition** to the rules outlined at the link above.

- **Short lines.** No line in your program shall be longer than 80 characters.

[1] A peer for the purpose of CPE 101 is defined as *"student taking the same section of CPE 101"*.

- **Header Comments.** The header comment **must be supplied** for each file submitted. The header comment must contain the following information:
  - Course number, section.
  - Instructor name.
  - Your name.
  - Name/Purpose of the program.
  - Date.

  Extra information in the header comment may be provided as well (version, extra dates - e.g., one for assignment commencement, one — for submission, etc...).

Testing and Submissions. Any submission that does not compile using the

```
gcc -ansi -Wall -Werror -lm
```

compiler settings will receive an automatic score of 0.

For each program you have to write, you will be provided with instructor's executable and with a battery of tests. The programs you submit must pass all tests made available to you. You can check whether or not a program produces correct output by running the instructor's executable on the test case, then running yours, and comparing the outputs.

Program Outputs must co-incide. This means that your output should match the output of my executables **character-for-character** including white-space, line breaks and the letter case. Any deviation in the output is subject to penalties.

Please, make sure you test all your programs prior to submission! Feel free to test your programs on test cases you have invented on your own[2].

The Task

Note: Please consult the instructor if any of the tasks are unclear.

For this lab, you will write and submit three simple programs. The programs will involve standard input/output functions, use of numeric (integer, floating point) variables, and some arithmetic operations.

Program 1: House Price Computation (price.c)

The first program is a simple calculator that given the price of a house and its square footage computes price per square foot. (*Note:* there is usually a wide diversity of houses on the market and a wide range of house sizes and prices. Price per square foot is one of the few measures that allows apples-to-apples comparisons between two houses. As such, it is commonly used by both real estate agents and perspective buyers to compare a variety of houses to each other).

[2] In this lab, we provide you with a large collection of test cases. In some future labs, test case development will be a major part of the lab assignment, so it never hurts to start early

Non-functional requirements

Non-functional requirements are general requirements that describe the nature of the problem and the expectations from the program. All requirements in the specifications you receive are numbered for your (and my) convenience.

SN1. The file name of your program shall be `score.c`.

Functional requirements

Functional requirements describe the behavior of the program.

SR1. At the begining of the execution, your program shall output the following text:

```
What's the price of the house?
```

SR2. The program then shall read the house price. The score shall be stored as an `integer` variable.

SR3. The program shall then output the following text:

```
What's the square footage?
```

SR4. The program shall read the square footage of the house. This value shall also be stored as an `integer` variable.

SR5. Next, the program shall compute the price per square foot. The formula is:

$$\mathsf{PricePerSquareFoot} = \frac{\mathsf{Price}}{\mathsf{Footage}}.$$

Here, Price is the first input to your program, (see **SR2**), and Footage is the second input to your program (see **SR4.**).

The computed PricePerSquareFoot value must be **floating point**.

SR6. The program shall output an empty line, and then on the next line shall output the following text:

```
Price per square foot is:  <PricePerSquareFoot>
```

Here, you shall substitute the value computed in **SR5** for `<PricePerSquareFoot>` (note, NO angle brackets!!!). The output shall end with a new line.

Sample Run. Here is a sample run of the program.

```
> gcc -ansi -Wall -Werror -lm -o price price.c
> price
What's the price of the house? 500000
What's the square footage? 2450

Price per square foot is: 204.081635
>
```

Additional instructions

- The house price and the footage will be **non-negative** values.
- The instructor's executable is available from the class web page. It's name is `price-alex.out`.
- See Appendix A. on more information about testing and testing scripts.

Program 2: Temperature Converter (converter.c)

In the U.S., all temperatures are measured in degrees Fahrenheit, a measurement scale that, in the eyes of the instructor, lacks any meaningful intuition. Most European countries, including instructor's home country of Russia measure temperature in degrees Celcius, which provide a much more intuitive way of measuring temperature. In particular, 0 degrees Celcius is the freezing point of water, while 100 degrees Celcius is the boiling point of water[3]. The (normal) human body temperature is 36.6 degrees Celcius.

You will write a program that converts temperature in degrees Celcius into degrees Fahrenheiht.

Functional Requirements

You shall create from scratch a C program `converter.c`. The program shall do the following:

CR1. Upon startup, the program shall display the following text:

```
Temperature Conversion:  Celcius to Farenheight
```

CR2. After the text above, the program shall skip one line, and then output the following text:

```
Enter temperature in degrees Celcius:
```

CR3. The program then shall read the temperature from input as a floating point number.

CR4. The program shall convert the entered temperature into degrees Fahrenheit.

This shall be done using the formula:

$$\mathsf{degreesF} = \frac{9}{5} \cdot \mathsf{degreesC} + 32$$

where degreesF is the temperature in degrees Fahrenheit and degreesC is the temperature in degrees Celcius.

CR5. The program shall then output the following text:

```
The temperature in degrees Fahrenheit:  <degreesF>
```

where `<degreesF>` is replaced by the number computed in **CR5**.

[3]According to Wikipedia, this is no longer the case. However, for practiacal purposes, water freezes at 0 degrees Celcius and boils at 100 degrees Celcius under normal atmospheric pressure.

Sample Run. Here is a sample output for the example discussed above.

```
> gcc -ansi -Wall -Werror -lm -o converter converter.c
> converter
Temperature Conversion: Celcius to Fahrenheit

Enter temperature in degrees Celcius:36.6
The temperature in degrees Fahrenheit: 97.879997
>
```

Additional instructions

- There are no restrictions on the value of temperature entered in your program (although, realistically, the temperature cannot be less than -273.15 degrees Celcius).
- Name your program `converter.c`.
- The instructor's executable is available from the class web page. It's name is `converter-alex.out`.
- Use `#define` preprocessor instructions for the constants in your program.
- See Appendix A. on more information about testing and testing scripts.

Program 3: Naïve Electoral College (naive-elections.c)

The last program of the lab is rather large, but repetetive — feel free to use copy-and-paste as it suits you.

In 2008, the United States held a Presidential election. There were two major party candidates running, Barak Obama (Democratic Party) and John McCain (Republican Party). The electoral system of the United States, which you all have studied in high school is unique in its use of the electoral college system. Each US state is awarded a number of votes in the electoral college equal to the total number of its Congressional delegation (two Senators plus the number of Representatives; District of Columbia gets 3 electoral votes). The popular vote winner in the state receives the state's electoral college votes. Nebraska and Maine award their electoral votes slightly differently: by giving two electoral college votes to the winner of the popular vote in the state, and giving the remaining votes individually to winners in each of the Congressional districts (Maine has two Congressional districts, Nebraska has three). There is a total of 538 electoral votes at stake, and 270 votes win the election. The 269-269 tie is broken in the House of Representatives.

In the 2008 Presidential election Barak Obama has won 365 electoral votes to John McCain's 173 to become the President of the U.S.

You will write a program that will compute the electoral college votes received by the two major candidates for President in a Presidential election. The input to the program will be a sequence of 51 numbers indicating who won the popular vote in each state (and the District of Columbia). The output of the program will be the total number of electoral college votes each candidate received.

Non-Functional Requirements

This is going to be a rather naïve and tedious-to-implement program, but correct implementation will help you understand the need for C constructs we will be studying in the next few weeks.

ECN1. The table of Electoral College votes is found in **Appendix B.**.

ECN2. You shall use 51 C constants (using `#define` directive) to represent the electoral college votes of each state/DC. Each constant shall be named after the two-letter state abbreviation: AL, AK, AZ, CA, CT, DC, GA, FL, etc.... The value of each constant comes from the table mentioned in **ECN1.**.

ECN3. You shall also use one more C constant, `ELECTORALCOLLEGE`, whose value is set to 538 — the total number of votes in the Electoral College.

ECN4. Your program shall be named `naive-elections.c`.

Functional Requirements

ECF0. The program shall work in a batch mode. It will read from the input stream 51 numbers representing the election results in each of 50 US states and DC. Along the way, it will keep track of the number of electoral college votes for one of the major party candidates. After the election results from all 50 states and DC are read in, the program will obtain the electoral college vote total for each candidate and will output it.

The batch mode means that the input to the program will come from a file via input redirection. What it entails is that unlike the first two programs, for this program, there is no need in multiple `printf` statements prompting the user to enter the next value.

ECF1. The input to the program is a sequence of 51 zeroes and ones. Each number in a sequence represents the results of the popular vote in one of the states or DC. The vote results come in order of appearance of states in the **Appendix B** table (alphabetical order by full state name). That is, first number in the sequence specifies the results of popular vote in Alabama, second — in Alaska, third – in Arizona, and so forth.

The input will mean the following:

- **0**: popular vote in the state is **won by John McCain**.
- **1**: popular vote in the state is **won by Barack Obama**.

The only exception to this rule is the input number representing the voting results in Nebraska. This is the input number 28 to this program (See Appendix B). This number **is equal to the total number of electoral votes Barack Obama received from the State of Nebraska.** This number is guaranteed to be in the range between 0 and 5[4]

[4] In the actual election, Obama won Nebraska's Congressional district 2 centered around Omaha, NE and received one Electoral College vote from this state.

Example. Suppose the first seven numbers in the input sequence are `0 0 0 0 1 1 1`. This would mean (check the Electoral College table in **Appendix B**) that John McCain has won the popular vote in Alabama, Alaska, Arizona and Arkansas, while Barack Obama has one the popular vote in California, Colorado and Connecticut.

ECF2. Your program shall use two integer variables to keep track of the electoral college vote for each candidate. While you have the option of naming them differently, in the rest of the requirements we will refer to them as `votesMcCain` and `votesObama`.

Your program also shall contain a single integer variable, we refer to as `stateDecision` to represent the input information.

ECF3. Both `votesMcCain` and `votesObama` are initialized to 0 at the beginning of the program.

ECF4. The program repeats the following sequence of actions 51 times (for each input number read):

ECF4-1. The program reads the next input number into the `stateDecision` variable.

ECF4-1. The program updates the popular vote tally for Barack Obama. This is done by computing the number of electoral college votes Barack Obama received from the state whose decision has just been read from the input stream and adding this number to the current popular vote tally for Barack Obama.

Notice that for all states but Nebraska, stateDecision =1 if Barack Obama won the state, and 0 if he lost it. Thus, the number of electoral college votes Barack Obama receives from the state can be computed as `stateDecision * <ST>` where `<ST>` is the constant representing the electoral college vote of the state in question (i.e., `AL` for the first input number, `AK` for the second, etc...). In case of Nebraska, the number of votes Barack Obama won will be read into the `stateDecision` variable directly (this is the easy case!).

Note: ECF4 actually means that you have to **repeat** roughly the same piece of code 51 times (this is where copy-and-paste help). *Even if you DO know how to use loops in C, **don't use them here** - you will be penalized for it.*

ECF5. After the actions described in **ECF4** have been performed 51 times, your `votesObama` variable will contain the number of votes in the entire electoral college received by Barack Obama. Next, your program shall compute the number of electoral collect votes received by John McCain. To do this, subtract the number of the electoral college votes Barack Obama received from the total number of the electoral college votes (represented as the `ELECTORALCOLLEGE` constant in your program).

ECF6. Finally, your program shall output the electoral college votes for both candidates. The following two lines shall be printed:

```
Electoral College vote total for John McCain (R) is <votesMcCain>
Electoral College vote total for Barack Obama (D) is <votesObama>
```

Here, your program outpus the values of variables `votesMcCain` and `votesObama` where the placeholders `<votesMcCain>` and `<votesObama>` are used above. After this, the program ends its work.

Sample Output. Consider an input file `election-test04` which looks as follows:

```
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 5 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Here is the result of the `naive-elections.c` program being compiled and run using this file as input:

```
> gcc -ansi -Wall -Werror -lm -o elections naive-elections.c
>  elections < election-test04
Electoral College vote total for John McCain (R) is 9
Electoral College vote total for Barack Obama (D) is 529
```

Submission.

Files to submit. You shall submit three files: `price.c`, `converter.c` and `naive-elections.c`.

Your file names shall be as specified above (and remember that Linux is case-sensitive). We use automated grading scripts. Any submission that had to be compiled and run manually will receive a deduction.

Submission procedure. Use `handin` to submit your work. The procedure is as follows:

- `ssh` to `vogon` (`vogon.csc.calpoly.edu`).
- When prompted for `vogon password`, enter your CSL password – the one you use to log onto lab machines in 14-302.
- Upon login, change to your Lab 2 work directory[5].
- Execute the `handin` command. Please note the following:

  Section 01 students:

  ```
  > handin dekhtyar lab02-01 price.c converter.c naive-elections.c
  ```

  Section 09 students:

  ```
  > handin dekhtyar lab02-09 price.c converter.c naive-elections.c
  ```

  Note: Submission from vogon will always work. Submitting from your CSL machine without ssh-ing to vogon first may or may not work, depending on some internal settings. Most of the time it will, but there is no stict guarantee. Try, and if does not work, ssh vogon.

[5]Lab 1 experience should teach you to create a new working directory for each assignment you do for this class inside your `cpe101` directory.

Other submission comments. Please, DO NOT submit binary files. Please, DO NOT submit binary files. (this has been an issue in the past.)

`handin` is set to stop accepting submissions 24 hours after the due time.

Grading

The lab grade is formed as follows:

`price.c`	30%
`converter.c`	30%
`naive-elections.c`	40%

Any submitted program that does not compile earns 0 points.

Any submitted program that compiles but fails at least one **public** (i.e., made available to you) test earns no more than 30% of its full score (and can possibly earn less).

Any submitted program that compiles and succeeds on **all** publically available tests earns at least 50% of its full score.

All programs will be checked for style conformance. Any style violation will be noted. The program will receieve a 10% penalty.

Appendix A. Testing

This appendix provides a brief description of testing procedures employed in this lab, as well as a general overview of testing.

General Notes. From now on testing is a mandatory activity for each of your assignments. Testing, i.e., *the process of running your program on a specific set of inputs* is done to ensure that your program is designed and implemented correctly.

Each set of inputs used in testing is called a test case.

Interactive testing. The simplest way to test your `attendance.c` and `converter.c` programs is interactive mode. Simply start the program, and at each prompt pick a desired input, enter it into the program and, at the end, observe the output. Once the output produced it needs to be validated (see below).

Batch testing. Testing a program in a batch mode involves two steps. On step 1, you create a *file that contains the test case.* Open any text editor and enter all inputs for the program (separate them with spaces). Save your file (give it an appropriate name). On step 2 you run your program using input redirection to read inputs from the file you have created, rather than from standard input (i.e., keyboard).

The `<` is the input redirection symbol. The syntax of an input-redirection command is

```
 >  Command < file
```

Here `command` is the command/executable program to be run and `file` is the file from which the inputs for the command/program are read.

For example, the following commands run the Lab 2 programs in batch mode:

```
> price < price-test01
  ...
> converter < converter-test04
  ...
> elections < election-test03
```

Test Files for Lab 2. The web page for Lab 2 contains a set of test cases for each of the programs. Sample test cases (`price-test10`, `converter-test10`, `election-test01`) can be downloaded directly and/or viewed within the browser. A complete set of *public* test cases for each program can be downloaded as a gzipped tar file. The file names are `score-tests.tar.gz`, `converter-tests.tar.gz` and `election-tests.tar.gz`.

Each archive should be downloaded to the directory where the respective C programs resides. A gzipped tar file is an archive that was constructed in two steps. First, a collection of files had been *merged together* into a single file. This is done using the tar command, and the resulting file is usually called a tar file and a `.tar` extension is used to designate it.

Second, the tar file was turned into a archive using Unix's (and now -Linux's) compression program `gzip`. Gzipped files gain a `.gz` extension.

To unpack the archive, follow the following two steps (the example below uses `converter-tests.tar.gz` file:

```
> gunzip converter-tests.tar.gz
> tar -xvf converter-tests.tar
```

After the first command (`gunzip`), the `converter-tests.tar.gz` will disappear from the current directory, and will be replaced by its extracted version, `converter-tests.tar`. The second command, if executed correctly will produce the following:

```
converter-test01
converter-test02
converter-test03
converter-test04
converter-test05
converter-test06
converter-test07
converter-test08
converter-test09
converter-test10
converter-test11
converter-test12
converter-test13
converter-test14
converter-test15
converter-test16
converter-test17
converter-test18
```

```
converter-test19
converter-test20
```

Each file listed in the output of the `tar` command will now appear in your current directory.

Executables. To facilitate testing, and provide for you a "golden standard" by which your programs' output shall be measured, we provide three executable files, `price-alex`, `converter-alex` and `elections-alex` on the Lab 2 web page. Download these files in your Lab 2 directory and use them with the test cases to find the correct output for each test case.

Please note, that the instructor's executables will work on the CSL Linux machines, but won't work under either MS Windows or MacOS operating systems (neither will they work on non-Linux CSL servers).

Script testing. A script is a simple program, typically written in an interpreted language (awk, perl, sed, C shell script language, etc...) designed to perform some simple task or collection of tasks. A test script is a script that runs the program being tested on different test cases and, if needed, records program output.

For Lab 2, you are provided with six grading scripts. Three of them, `score-test.csh`, `converter-test.csh` and `elections-test.csh` are test scripts designed to run **your** programs. The other three, `score-alex-test.csh`, `converter-alex-test.csh` and `elections-alex-test.csh`, use my executables to provide a baseline for validating the results of your programs.

csh in the name of the files stands for C Shell — the environment running inside your Linux terminal window[6]. C Shell comes with a simple script language, which allows one to write simple programs that interact with the shell and the operating system environment.

.csh files are executable scripts. You run them by typing their name at the Linux prompt, and hitting `<Enter>`. E.g., to run `elections-alex.test.csh`, simply type:

```
> elections-alex-test.csh
```

or

```
 > ./elections-alex-test.csh
```

If neither of these commands works, try the following command:

```
 > chmod u+x elections-alex-test.csh
```

and run `elections-alex-test.csh` as shown above after that[7].

[6] In reality, a slightly different shell is used on CSL machines by default, bash. Both provide the same set of services concentrated around accepting commands from the command-line and deciding how they get executed.

[7] Linux has a special mark to determine whether a specific file is "executable", i.e., can be run. For example, binary files created by the `gcc` compiler are marked as "executable" and hence you can run them. C Shell scripts also must be marked as "executable". However, when you copy a file from my web page, the "executable" flag (mark) may get lost — this depends on the settings of your Linux system. The `chmod u+x <File>` command restores the "executable" flag for a given file.

Output of the program being tested will contain results of running the program on all public test cases.

Interactive vs. batch mode. If you run your `score` or `converter` program in an interactive mode (i.e. by typing inputs from the keyboard) you will see a slightly different output than if you run the same program in the batch mode with the same data. This is because when input is redirected from the file, values read by the program are not automatically displayed (and no `<Enter>` key gets hit). Please note that **this is expected behavior**. To see what it is, run each of the first two programs in an interactive mode, and then in batch node with the same inputs. (note also, that **my** programs exhibit exactly the same behavior.)

Your own test cases. For your programs you are encouraged to create new test cases, beyond the ones provided by the instructor and verify that your program provides correct outputs for them.

Appendix B. Electoral College

No.	State	Abbr.	Population	Electoral College Votes
1.	Alabama	AL	4,500,752	**9**
2.	Alaska	AK	648,818	**3**
3.	Arizona	AZ	5,580,811	**10**
4.	Arkansas	AR	2,725,714	**6**
5.	California	CA	35,484,453	**55**
6.	Colorado	CO	4,550,688	**9**
7.	Connecticut	CT	3,483,372	**7**
8.	Delaware	DE	817,491	**3**
9.	District of Columbia	DC	563,384	**3**
10.	Florida	FL	17,019,068	**27**
11.	Georgia	GA	8,684,715	**15**
12.	Hawaii	HI	1,257,608	**4**
13.	Idaho	ID	1,366,332	**4**
14.	Illinois	IL	12,653,544	**21**
15.	Indiana	IN	6,195,643	**11**
16.	Iowa	IA	2,944,062	**7**
17.	Kansas	KS	2,723,507	**6**
18.	Kentucky	KY	4,117,827	**8**
19.	Louisiana	LA	4,496,334	**9**
20.	Maine	ME	1,305,728	**4**
21.	Maryland	MD	5,508,909	**10**
22.	Massachusetts	MA	6,433,422	**12**
23.	Michigan	MI	10,079,985	**17**
24.	Minnesota	MN	5,059,375	**10**
25.	Mississippi	MS	2,881,281	**6**
26.	Missouri	MO	5,704,484	**11**
27.	Montana	MT	917,621	**3**
28.	Nebraska	NE	1,739,291	**5**
29.	Nevada	NV	2,241,154	**5**
30.	New Hampshire	NH	1,287,687	**4**
31.	New Jersey	NJ	8,638,396	**15**
32.	New Mexico	NM	1,874,614	**5**
33.	New York	NY	19,190,115	**31**
34.	North Carolina	NC	8,407,248	**15**
35.	North Dakota	ND	633,837	**3**
36.	Ohio	OH	11,435,798	**20**
37.	Oklahoma	OK	3,511,532	**7**
38.	Oregon	OR	3,559,596	**7**
39.	Pennsylvania	PA	12,365,455	**21**
40.	Rhode Island	RI	1,076,164	**4**
41.	South Carolina	SC	4,147,152	**8**
42.	South Dakota	SD	764,309	**3**
43.	Tennessee	TN	5,841,748	**11**
44.	Texas	TX	22,118,509	**34**
45.	Utah	UT	2,351,467	**5**
46.	Vermont	VT	619,107	**3**
47.	Virginia	VA	7,386,330	**13**
48.	Washington	WA	6,131,445	**11**
49.	West Virginia	WV	1,810,354	**5**
50.	Wisconsin	WI	5,472,299	**10**
51.	Wyoming	WY	501,242	**3**