Fall 2008	**CPE 101:** Fundamentals of Computer Science I	Alexander Dekhtyar

Lab 1: Simple C Programs

Due date: Thursday, October 2, 11:59pm.

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. This week we 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. Any deviation in the output is subject to penalties (e.g., use of different words in the output).

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: Course Attendance Estimation (attendance.c)

Let's be honest, college students tend to skip classes when it suits their needs. Some professors, however, keep track of the number of lectures skipped by each student, and when that number exceeds a certain percentage, they tend to assume that the student will fail the course. So, a crafty student needs to keep track of his/her missing lectures.

Your first program will, given the number of courses skipped by the student, tell what percentage of the course had been missed.

[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.

AN1. Your program will be computing the percentage of classes missed for a Tuesday–Thursday class schedule for the Fall 2008 quarter at Cal Poly.

AN2. The Fall 2008 quarter has 11 weeks. Each week, except for week 9, there are two classes taught. In week 9, the Thursday class falls on the Thanksgiving day.

AN3. You will use C constants declared via C's `#define` directive to represent the number of weeks, the number of classes per week and the total number of classes that fall on holidays. The names of the constants shall be: `FALLQUARTERWEEKS`, `CLASSESPERWEEK` and `HOLIDAYS` respectively (see **AN2.** for the values of the constants).

AN4. The file name of your program shall be `attendance.c`.

Functional requirements

Functional requirements describe the behavior of the program.

AR1. Your program shall start by computing the total number of classes that will take place during the Fall 2008 quarter. This number can be computed from the constants defined in your program (see requirement **AN3.**). The formula is left up to you (feel free to consult your peers or instructor if you have questions). This number shall be stored in a `floating point` variable.

AR2. Next, the program shall output to screen the following text:

```
How many classes skipped?
```

AR3. The program then shall read the number of classes. The number of classes shall be stored as an `integer` variable.

AR4. Next, the program shall compute the percentage of all classes skipped. The formula is:

$$\mathsf{SkippedPercentage} = \frac{\mathsf{ClassesSkipped}}{\mathsf{AllClasses}} \cdot 100.$$

Here, AllClasses is the total number of lectures that you have computed (see **AR1.**) and ClassesSkipped is the number of classes skipped that was read from input (see **AR3.**).

AR5. The program shall output an empty line, and then on the next line shall output the following text:

```
Total number of lectures in your course is <AllClasses>
```

Here, you shall substitute the total number of classes your program has computed (see **AR1.**) for `<AllClasses>` (note, NO angle brackets!!!).

AR6. Finally, your program shall output the following text:

```
You skipped <SkippedPercentage>% of your classes.  Beware!
```

The text shall end with a new line.

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

```
> gcc -ansi -Wall -Werror -lm -o attendance attendance.c
> attendance
How many classes skipped? 4

The number of lectures in your course is 21.000000
You skipped 19.047619 % of your classes. Beware!
>
```

Additional instructions

- The value of the number of classes skipped will be *non-negative* and will not be greater than the total number of lectures in the Fall 2008 quarter. Your program is responsible for correctly behaving when the values of input parameter is as described above.
- The instructor's executable is available from the class web page. It's name is `attendance-alex.out`.
- See Appendix A. on more information about testing and testing scripts.

Program 2: Currency Converter (converter.c)

This summer, for the first time 13 years I visited Russia. As you might know, Russian currency, the ruble (RR) is not the world's most stable currency. Over the years, it has gone through a number of de-nominations, and the RR-to-USD exchange rate has had significant fluctuations.

A typical way to exchange US Dollars for Russian Rubles in Russia is a currency exchange kiosk. Such kiosks can be found virtually on every street of every city in Russia. Each kiosk comes with its own exchange rate and its own rules of exchange.

In this program, you will emulate the work of a Russian currency exchange kiosk. Such kiosks exchange currency in both directions: they both buy and sell USD, but you will be only emulating the purchase of USD currency by the kiosk. Each kiosk has two parameters guiding its operation:

- Exchange Rate. How much Russian currency $1 buys. The exchange rate is supplied in terms of rubles and *copecks*[3].

  For example, exchange rate of 24.45 means that $1 can be exchanged for 24 rubles and 45 copecks.

- Comission. The amount of money the kiosk charges for its services. Typically, this is either a percentage of the total sale, or a predetermined transaction fee. In the exchange kiosk you are simulating, comission will be a percentage of the overall transaction.

  For example a comission of 1% means that the kiosk exchanges 99% of the USDs using its exchange rate, and takes the remaining 1%.

Example scenario. The exchange rate of the kiosk is 25.00. The comission is 1%. Customer wants to exchange $100. The kiosk exchanges $100 for $(1 - 0.01) * 100 * 25.00 = 2175$ rubles. The kiosk takes 1% commission, that is $1, and exchanged the remaining $99 at the rate of 25.00 rubles per dollar.

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:

```
Currency Converter Kiosk:  USD to RR
```

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

```
Enter exchange rate:
```

CR3. The program then shall read the exchange rate from input. The exchange rate shall be a floating point number.

CR4. On the next line, the program shall output

```
Enter commission %:
```

CR5. The program the shall read the `comission` value. The comission value is a floating point number.

CR6. On the next line, the program shall output

```
How much money do you want to change?
```

CR7. The program shall read the amount of USD currency that needs to be exchanged. The amount is an integer number.

CR8. The program shall compute the amount of RR to be returned. This shall be done using the formula:

$$\text{Rubles} = \frac{100 - \text{comission}}{100} \cdot \text{Dollars} \cdot \text{exchange},$$

where Rubles is the amount of rubles to be returned, dollars is the amount of dollars being exchanged, commission is the kiosk's commission rate and exchange is the kiosk's exchange rate.

[3]Russian analogs of cents

CR9. The program shall also compute the amount of comission. The formula is:

$$\mathsf{commissionRR} = \frac{\mathsf{comission}}{100} \cdot \mathsf{Dollars} \cdot \mathsf{exchange} = \mathsf{Dollars} \cdot \mathsf{exchange} - \mathsf{Rubles}.$$

1. The program shall then output the following text:

   ```
   Here are your <Rubles> rubles. The commission was <commissionRR>
   rubles.
   ```

   Note 1: The program shall print an empty line before the `Here are your <Rubles> rubles.` line.

   Note 2: Your program shall replace `<Rubles>` and `<commissionRR>` with the actual values computed.

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

```
> gcc -ansi -Wall -Werror -lm -o convert convert.c
> convert
Currency Converter Kiosk: USD to RR

Enter exchange rate: 25.00
Enter commission %: 1.00
How much money do you want to change? 100

Here are your 2475 rubles.
The commission was 25 rubles.
```

Additional instructions

- The tests will use the following restrictions on the values of the inputs in the program:
  - exchange rate: *positive* (not 0!).
  - commission: *non-negative* (0 is OK!), less than 100.
  - dollars to be exchanged: *non-negative*.

  Your program is responsible for correctly behaving when the values of input parameters are as described above.
- 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 instruction for the constant 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 cut-and-paste as it suits you.

The Presidential elections are coming up in about a month, and, of course, everyone wants to know who wins. 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, but we will ignore it for now[4]. 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.

You will write a program that will compute the electoral college votes received by the two major candidates for President this year: John McCain (Republican Party) and Barack Obama (Democratic Party). The input to the program will be a sequence of 51 numbers indicating who won th popular vote in each state (and DC). 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.

[4]Historically, neither state has ever wound up splitting the its vote.

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**.

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 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. . .).

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>
```

After this, the program ends its work.

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

```
0 1 1 1
```

– Students in **Section 011** will submit to the instructor's directory `lab02-11`.

Thus, students from **Section 009** will execute the following submission command:

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

Students from **Section 011** will execute the command:

```
> handin dekhtyar lab02-11 attendance.c converter.c naive-elections.c
```

Notifications. We have turned on some assignment management features in the `handin`. In particular, each submission will trigger a confirmation email sent to your calpoly.edu account (CalPoly email). The email will specify the list of files you have submitted. If there is any discrepancy between your submission and the filelist in the email, please contact me.

Other submission comments. `handin` will now accept only the filename specified in the beginning of this section. You will receive an email, if you submitted wrong files. **Please, DO NOT submit binary files**.

`handin` now archives **all** your submissions.

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

Grading

The lab grade is formed as follows:

`attendance.c`	30%
`converter.c`	35%
`naive-elections.c`	35%

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:

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

Test Files for Lab 2. The course web page contains a set of test cases for each of the programs. Sample test cases (`attendance-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 `attendance-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 reside. 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, `attendance-alex`, `converter-alex` and `elections-alex` on the course 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.

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, `attendance-test.csh`, `converter-test.csh` and `elections-test.csh` are test scripts designed to run **your** programs. The other three, `attendance-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
```

[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.

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 attendance 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 tests cases. For the `attendance.c` program, the set of test cases provided to you is **exhaustive** — every possible (according to the spec) value of the input parameter is a test case in the battery of tests that you download.

For other two programs you are encouraged to create new test cases 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**