Week 01 Laboratory Sample Solutions

Objectives

Preparation

Before the lab you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this lab called `lab01`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab01
$ cd lab01
$ 1521 fetch lab01
```

Or, if you're not working on CSE, you can download the provided code as a zip file or a tar file.

EXERCISE — INDIVIDUAL:

Remove All Vowels from STDIN

Write a C program `no_vowels.c` which reads characters from its input and writes the same characters to its output, except it does not write vowels.

Your program should stop only at the end of input.

`no_vowels.c` must only use:
scanf to read one character at a time from stdin.
printf to print one character at a time to stdout.

Once you have `no_vowels.c` working correctly it should behave as follows:

```
$ ./no_vowels
Hello, World!
Hll, Wrld!
Ctrl-D
$ echo "Peter Piper picked a peck of pickled peppers." > input
$ ./no_vowels < input
Ptr Ppr pckd  pck f pckld ppprs.
$ ./no_vowels
Andrew is the LiC of COMP1521
ndrw s th LC f CMP1521
Are you saying 'Boo' or 'Boo-Urns'?
r y syng 'B' r 'B-rns'?
In this house, we obey the laws of thermodynamics!
n ths hs, w by th lws f thrmdynmcs!
Ctrl-D
$
```

HINT:

```
$ man 3 scanf
$ man 3 printf
```

NOTE:

You can assume a vowel, is one of the ASCII values 'a', 'e','i', 'o', 'u' and their upper case equivalents 'A', 'E','I', 'O', 'U'

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest no_vowels
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab01_no_vowels no_vowels.c
```

You must run `give` before **Monday 28 September 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `no_vowels.c`

```c
#include <stdio.h>
#include <string.h>

int main(void) {

    char c;

    // scanf will read from STDIN
    // the `format string` "%c" tells scanf to read a single byte
    // the single byte is stored at the location of c
    // ie. c is set to the value of a character from STDIN
    // scanf will return the number of format specifiers matched
    // we have one format specifier "%c" so we expect to have 1 returned

    while (scanf("%c", &c) == 1) {

        // strchr will return NULL iff
        // c isn't in the string "aAeEiIoOuU"
        // ie. check that c isn't a vowel

        if (strchr("aAeEiIoOuU", c) == NULL) {

            // printf will write to STDOUT
            // the `format string` "%c" tells printf to write a single byte

            printf("%c", c);
        }
    }

    return 0;
}
```

EXERCISE — INDIVIDUAL:

Transform All Uppercase letters to Lowercase

Write a C program `no_uppercase.c` which reads characters from its input and writes the same characters to its output, any upper case letters are replaced by their as lower case equivalent.

Your program should stop only at the end of input.

Add code to `no_uppercase.c` so that, given text on stdin, any uppercase letters are printed as lowercase to stdout.

`no_uppercase.c` must only use:
getchar to read one character at a time from stdin.
putchar to print one character at a time to stdout.

Once you have `no_uppercase.c` working correctly it should behave as follows:

```
$ ./no_uppercase
ABC
abc
ABCabc123
abcabc123
123!@#
123!@#
Hello, World!
hello, world!
Andrew is the LiC of COMP1521
andrew is the lic of comp1521
Ctrl-D
```

```
$ echo "Peter Piper picked a peck of pickled peppers." > input
$ ./no_uppercase < input
peter piper picked a peck of pickled peppers.
$
```

HINT:

```
$ man 3 getchar
$ man 3 putchar
$ man 3 tolower
$ man 7 ascii
```

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest no_uppercase
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab01_no_uppercase no_uppercase.c
```

You must run `give` before **Monday 28 September 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `no_uppercase.c`

```c
#include <stdio.h>
#include <ctype.h>

int main(void) {

    // the type of c must be int
    // using char here is a common C programming error
    int c;

    // getchar will read a single byte from STDIN
    // the single byte is returned by getchar
    // getchar will return the special value EOF if it can not read a byte

    while ((c = getchar()) != EOF) {

        // tolower will write a single byte to STDOUT

        // If c is an uppercase letter, tolower() returns its lowercase equivalent
        // Otherwise, it returns c.

        putchar(tolower(c));

    }

    return 0;
}
```

EXERCISE — INDIVIDUAL:

Remove Uneven Lines of Input

Add code to `no_odd_lines.c` so that, given text on stdin, only print lines with an even number of characters to stdout.

Add code to `no_odd_lines.c` so that, given text on stdin, only print lines with an even number of characters to stdout.

`no_odd_lines.c` must only use:
fgets to read one line at a time from stdin.
fputs to print one line at a time to stdout.

Once you have `no_odd_lines.c` working correctly it should behave as follows:

```
$ ./no_odd_lines
Hello, World
Hello, World!
Hello, World!
Andrew is the LiC of COMP1521
Andrew is the LiC of COMP1521
Andrew is the LiC of COMP1521, and Dylan is the Admin
Andrew is the LiC of COMP1521, and Dylan is the Admin
Ctrl-D
$ echo "Peter Piper picked a peck of pickled peppers." > input
$ echo "A peck of pickled peppers Peter Piper picked." >> input
$ echo "If Peter Piper picked a peck of pickled peppers," >> input
$ echo "Where's the peck of pickled peppers Peter Piper picked?" >> input
$ ./no_odd_lines < input
Peter Piper picked a peck of pickled peppers.
A peck of pickled peppers Peter Piper picked.
Where's the peck of pickled peppers Peter Piper picked?
$
```

HINT:

```
$ man 3 fgets
$ man 3 fputs
$ man 3 strlen
```

NOTE:

The newline character(s) should be included in your count of characters.

You can assume lines contain at most 1024 characters including the newline character(s).

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest no_odd_lines
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab01_no_odd_lines no_odd_lines.c
```

You must run `give` before **Monday 28 September 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `no_odd_lines.c`

```c
#include <stdio.h>
#include <string.h>

#define MAX_LINE_LENGTH 1024

int main(void) {

    // line needs 1 extra byte of space to hold the `NULL byte`

    char line[MAX_LINE_LENGTH + 1];

    // fgets reads at most `sizeof line` - 1 bytes from `stdin`
    // the bytes are stored in `line` followed by a `NULL byte`
    // fgets will also stop reading when it reads a newline (\n)
    // The address of `line` is returned, or NULL on EOF

    while (fgets(line, sizeof line, stdin) != NULL) {

        // strlen return the number of bytes in `line`
        // until it reaches a `NULL byte`

        if (!(strlen(line) % 2)) {

            // fputs will write the contests of `line` to `stdout`
            // untill it reaches a `NULL byte`

            fputs(line, stdout);
        }
    }
    return 0;
}
```

EXERCISE — INDIVIDUAL:

Pretty Print Command Line Arguments

Add code to `my_args.c` so that, given *0 or more* command line arguments, the command line arguments are "pretty printed".

Once you have `my_args.c` working correctly it should behave as follows:

```
$ ./my_args
Program name: ./my_args
There are no other arguments
$ ../lab01/my_args
Program name: ../lab01/my_args
There are no other arguments
$ ./my_args hello world
Program name: ./my_args
There are 2 arguments:
    Argument 1 is "hello"
    Argument 2 is "world"
$ ./my_args "hello world" 1 2 3 4 5
Program name: ./my_args
There are 6 arguments:
    Argument 1 is "hello world"
    Argument 2 is "1"
    Argument 3 is "2"
    Argument 4 is "3"
    Argument 5 is "4"
    Argument 6 is "5"
$
```

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest my_args
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab01_my_args my_args.c
```

You must run `give` before **Monday 28 September 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `my_args.c`

```
#include <stdio.h>

int main(int argc, char **argv) {
    printf("Program name: %s\n", argv[0]);
    if (argc == 1) {
        printf("There are no other arguments\n");
    } else {
        printf("There are %d arguments:\n", argc-1);
        for (int i = 1; i < argc ;i++) {
            printf("\tArgument %d is \"%s\"\n", i, argv[i]);
        }
    }
    return 0;
}
```

EXERCISE — INDIVIDUAL:

Statistical Analysis of Command Line Arguments

Add code to `arg_stats.c` so that, given *1 or more* command line arguments, it prints the minimum and maximum values, the sum and product of all the values, and the mean of all the values.

Once you have `arg_stats.c` working correctly it should behave as follows:

```
$ ./arg_stats
Usage: ./arg_stats NUMBER [NUMBER ...]
$ ./arg_stats 1
MIN:  1
MAX:  1
SUM:  1
PROD: 1
MEAN: 1
$ ./arg_stats 1 2 3 4 5 6 7 8 9
MIN:  1
MAX:  9
SUM:  45
PROD: 362880
MEAN: 5
$ ./arg_stats 9 8 7 6 1 2 3 5 4
MIN:  1
MAX:  9
SUM:  45
PROD: 362880
MEAN: 5
$ ./arg_stats 1 9 1 9 1 9
MIN:  1
MAX:  9
SUM:  30
PROD: 729
MEAN: 5
$
```

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest arg_stats
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab01_arg_stats arg_stats.c
```

You must run `give` before **Monday 28 September 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `arg_stats.c`

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {

    if (argc == 1) {
        fprintf(stderr, "Usage: %s NUMBER [NUMBER ...]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int min = atoi(argv[1]);
    int max = atoi(argv[1]);
    int sum = 0;
    int prod = 1;

    for (int i = 1; i < argc ;i++) {
        min = min < atoi(argv[i])? min : atoi(argv[i]);
        max = max > atoi(argv[i])? max : atoi(argv[i]);
        sum += atoi(argv[i]);
        prod *= atoi(argv[i]);

    }

    printf("MIN:  %d\n", min);
    printf("MAX:  %d\n", max);
    printf("SUM:  %d\n", sum);
    printf("PROD: %d\n", prod);
    printf("MEAN: %d\n", sum / (argc - 1));

    return 0;
}
```

EXERCISE — INDIVIDUAL:

(Dis)Proving the Collatz Conjecture

Add code to `collatz.c` that take a single positive integer asa command-line argument and prints the collatz chain for that number.

This is how the collatz chain is calculated:

If the current number is `1`, the series terminates.
If the current number is **ODD**, then multiply by 3 and add 1 to get the next number
If the current number is **EVEN**, then divide by 2 to get the next number

NOTE:

You must implement `collatz.c` using a recursive function.

$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

Once you have `collatz.c` working correctly it should behave as follows:

```
$ ./collatz
Usage: ./collatz NUMBER
$ ./collatz 1
1
$ ./collatz 12
12
6
3
10
5
16
8
4
2
1
$ ./collatz 10
10
5
16
8
4
2
1
$
```

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest collatz
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab01_collatz collatz.c
```

You must run `give` before **Monday 28 September 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `collatz.c`

```c
#include <stdio.h>
#include <stdlib.h>

void collatz(int);

int main(int argc, char *argv[]) {

    if (argc != 2) {
        printf("Usage: %s NUMBER", argv[0]);
        return EXIT_FAILURE;
    }

    collatz(atoi(argv[1]));

    return EXIT_SUCCESS;
}

void collatz(int n) {
    printf("%d\n", n);
    if (n == 1) {
        return;
    } else if (n % 2 == 1) {
        collatz((3 * n) + 1);
    }  else {
        collatz(n/2);
    }
}
```

EXERCISE — INDIVIDUAL:

Calculating the Fibonacci Sequence The (Not So) Fast Way

Add code to `fibonacci.c` so that, given a line of input containing a fibonacci number the corresponding fibonacci number is printed.

This is how you obtain your fibonacci number:

Starting with 0 and 1:
Add the preceding two fibonacci number together to get the current fibonacci number:

$$F_0 = 0, \quad F_1 = 1,$$

$$F_n = F_{n-1} + F_{n-2},$$

> **NOTE:**
>
> You must implement `fibonacci.c` using a recursive function.
>
> `fibonacci.c` should never calculate the sam fibonacci number twice (hint: memoization).

Once you have `fibonacci.c` working correctly it should behave as follows:

```
$ ./fibonacci
0
0
1
1
2
1
5
5
9
34
Ctrl-D
$
```

> **NOTE:**
>
> You can assume the largest integer entered will be 46.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest fibonacci
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab01_fibonacci fibonacci.c
```

You must run `give` before **Monday 28 September 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `fibonacci.c`

```c
#include <stdio.h>
#include <stdlib.h>

#define SERIES_MAX 46

int fib(int n, int already_computed[]);

int main(void) {
    int already_computed[SERIES_MAX + 1] = { 0, 1 };
    int n;

    while (scanf("%d", &n) == 1) {
        printf("%d\n", fib(n, already_computed));
    }

    return EXIT_SUCCESS;
}

int fib(int n, int already_computed[]) {
    if (n <= 0 || n > SERIES_MAX) {
        return 0;
    }
    if (already_computed[n] != 0) {
        return already_computed[n];
    }
    int value = fib(n - 1, already_computed) + fib(n - 2, already_computed);
    already_computed[n] = value;
    return value;
}
```

Submission

When you are finished each exercises make sure you submit your work by running `give`.

You can run `give` multiple times. Only your last submission will be marked.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

Remember you have until **Mon Sep 28 21:00:00 2020** to submit your work.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

You check the files you have submitted here.

Automarking will be run by the lecturer several days after the submission deadline, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run by the lecturer you can view your results here. The resulting mark will also be available via give's web interface.

Lab Marks

When all components of a lab are automarked you should be able to view the the marks via give's web interface or by running this command on a CSE machine:

```
$ 1521 classrun -sturec
```