Information Processing 2 — Week 10 exercises

Complete the exercises in-class on your own laptop. You do not need to submit the answers online. Instead, ask an instructor to check your work during the class as soon as you finish answering each question. If you do not finish all the questions in-class, try to finish them outside class and have them checked *at the start of the next exercises class*.

Note: Exercises checked **more than one week** after the class will have a **50% late penalty** applied.

1 Character arrays vs. arrays of strings

Consider the function `char *monthName(int n)` shown in the slides. I decided to abbreviate all month names to three characters (including the 'illegal month').

```
char *monthName(int n)
{
  static char *name[] = {
    "???", // illegal month
    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec",
  };
  return 1 <= n && n <= 12 ? name[n] : name[0];
}

int main()
{
  printf("%s\n", monthName(7));
  return 0;
}
```

It worked perfectly. I then decided to convert the array of strings into a two-dimensional array of characters.

```
char *monthName(int n)
{
  static char name[][3] = {
    "???", // illegal month
    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec",
  };
  return 1 <= n && n <= 12 ? name[n] : name[0];
}
```

It did not work. Why did it not work?

What is the smallest possible change that would make it work?

▷▷ Ask an instructor to check and record your work.

Writing the ‘echo’ program

Let’s write the `echo` program, which prints its command-line arguments on the output separated by spaces.

2 The ‘echo’ program using array indexing

Write ‘`echo`’ using a loop with an integer variable, stepping the variable from `1` to `argc-1`, and using it as an index into `argv` to obtain each argument. Print one space *between* each argument. Print one newline after the last argument.

▷▷ Ask an instructor to check and record your work.

3 The ‘echo’ program using pointers

Modify your `echo` program to avoid using an integer variable. Hints:

- Since `argv` is a variable, you can increment it to make it point to the next command-line argument.
- Use a `while` loop, accessing each argument as `*argv`, and terminate the loop when `*argv` is zero (the null pointer at the end of the argument array).
- You don’t have to worry about printing one space too many at the end of the arguments.

▷▷ Ask an instructor to check and record your work.

Parsing numeric command-line options

You will often need to read command-line options containing integers. Let’s practice how to do that.

4 Converting a command-line argument to an integer value

Write a program that prints its first command-line argument. Check `argc` to make sure an argument was given on the command line (`argc > 1`). If no command-line argument was given, don’t print anything.

When that works, use the function `int atoi(char *s)` (declared in <`stdlib.h`>) in your program to convert the first command-line argument from a string to an `int` before printing its integer value. (Without any special treatment, your program should print `0` if you give a non-number as the command line argument.)

▷▷ Ask an instructor to check and record your work.

5 Default argument value

Modify your program so that

- if no command line arguments are given then it prints '10', but
- if one command-line argument is given then the argument is converted to an integer and printed, as in the previous exercise, and
- if more than one command-line argument is given then your program prints '`too many arguments`' and exits.

Store the value to be printed (default 10) in an `int` variable and use only one `printf()` statement.

▷▷ Ask an instructor to check and record your work.

6 Command line options starting with '−'

Modify your program so that it only recognises the number if it is preceded by a hyphen '−'. In other words, running your program as

```
$ ./program -42
```

will print '42'. If the argument does not begin with a hyphen, your program should print '`illegal argument`' and exit. Keep the same behaviour as above for zero arguments and for two or more arguments. Hints:

- To check if an argument starts with a hyphen you could use '`if ('-' == *argv[i])`'.
- If the argument starts with a hyphen, the text immediately following it (containing the number) begins at address '`argv[i]+1`'.

▷▷ Ask an instructor to check and record your work.

Writing the '`tail`' program

Let's develop a version of the 'tail' command that prints the last few lines of its input. We'll start by printing the last 10 lines. Then we'll generalise that by reading the number of lines to print as a command-line option. We will use pointers to dynamically allocate and manage the storage needed to remember the input lines.

The function `char *strdup(char *s)` duplicates string `s` by allocating enough storage for a copy (including the nul terminate), copying `s` into the new storage, and returning a pointer to the new copy. The new storage is allocated using `malloc()`. When your program no longer needs that storage (for example, if it is about to overwrite an old pointer with a new one) then you should call `free()` on the old pointer first.

Be very careful when writing code that `free()`s unwanted pointers. Storage management looks easy but getting it right is often difficult and bad storage management code is one of the most common causes of bugs in C programs.

Note: From Teams you can download `tail.c`, as a template to start from, and `words.txt` for testing.

7 Using dynamic storage [2 points]

The following ‘`tail`’ program prints the last 10 lines of its input (download ‘`tail.c`’ from Teams):

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

int getchars(char *s, int limit)
{
    int i = 0, c;
    while (i < limit - 1 && EOF != (c = getchar()) && c != '\n')
        s[i++] = c;
    if (EOF == c && i == 0) return -1;
    s[i] = '\0';
    return i;
}

#define LINEMAX 1024

int main(int argc, char **argv)
{
    char   line[LINEMAX];
    char  *lines[10];

    int index = 0;
    while (getchars(line, sizeof(line)) >= 0) {
        if (index >= 10) free(lines[index % 10]);
        lines[index++ % 10] = strdup(line); // this must be free()d
    }

    for (int i = index - 10;  i < index;  ++i) {
        if (i >= 0) {
            printf("%s\n", lines[i % 10]);
            free(lines[i % 10]);
        }
    }

    return 0;
}
```

This program uses a fixed-size array `char *lines[10]` to store the most recent ten lines of the input. That works well for printing the last 10 lines of the input, but not so well for printing more (or fewer) than the last 10 lines. In preparation for printing more (or fewer) than 10 lines, modify the program so that it obtains the storage for the last 10 lines using `malloc()`.

Hints:

- You will have to change the type of `lines` to be ‘pointer to pointer to `char`’.
- You will have to calculate the amount of storage to request from `malloc()` by multiplying the number of entries needed in the array (10) by the size of each element in the array `lines`.
- There are a few ways to calculate the size of each element of `lines`. The obvious way is to use ‘`sizeof(char *)`’. Perhaps the safest and best is to use ‘`sizeof(*lines)`’ which is my personal favourite because it will always be correct, regardless of the type of element stored in `lines`.
- Do not forget the ‘`*`’ before ‘`lines`’ in the previous hint!

▷▷ Ask an instructor to check and record your work.

8 Using command-line options in ‘`tail`’

Modify your `tail` program so that it checks its command line arguments. If it finds an argument starting with ‘`-`’ then it should convert the rest of that argument into an integer and use that value (instead of 10) as the number of lines to print from the end of the input. If more than one ‘`-`*num*’ argument is present, use the last value. If any other kind of command line argument is present, print ‘`illegal argument`’ and exit. Hints:

- When reading the number of lines to print, detect negative numbers and take some appropriate action.
- The program uses the number of lines to print (a constant, ‘`10`’) in many places. You will have to modify these appropriately, to use the (variable) number of lines to be printed instead.
- Don’t forget to allocate the correct amount of storage for `lines` too!

▷▷ Ask an instructor to check and record your work.

9 Using options to select different behaviours

Modify your `tail` program so that it accepts two forms of command-line option:

- `-`*n* should work as above, printing the last n lines of the input (default 10), and
- `+`*n* should start printing from the n^{th} line of the input, and continue printing until the end of the input.

Hints:

- You can search for and process the ‘`+`*n*’ option in a similar way to the ‘`-`*n*’ option.
- If multiple valid command-line options are given then you can either ignore all but the last option or make one of the options ‘override’ (have higher priority than) the other one.
- One way of giving priority to a particular option would be to process all command-line arguments and then check whether the `+`*n* was given. If it was then print the input staring at line n, otherwise print the last few lines as in the previous exercises.
- The two behaviours (printing from a specific line, or printing the last few lines) are different enough that you should have completely different code for each behaviour.

▷▷ Ask an instructor to check and record your work.

Challenges

I sometimes wish that `tail` would let me specify both `+`*m* and `-`*n* at the same time. It would then print the n lines of the file starting at line number m, equivalent to: ‘`tail +`*m* `| head -`*n*’.

10 Combining command-line options to give a third behaviour [1 point]

Modify your `tail` program so that '`tail +`m `-`n' prints the n lines starting at line number m in the input (equivalent to '`tail +`m `| head -`n'). Of course, the behaviour should otherwise be unchanged from previous versions of the program.

▷▷ Ask an instructor to check and record your work.

11 Wizard's challenge: implement two programs in one [1 point]

Running your program as '`./tail +1 -`n' is equivalent to running '`head -`n'.

You can make an 'alias' for your `tail` program using a *symbolic link*:

```
$ ln -s ./tail ./head
```

or, on Windows:

```
$ mklink /h head.exe tail.exe
```

Create such a link and then make sure you can run `./head` to produce the same results as `./tail`.

A program can find out its own name by looking in `argv[0]` which contains the exact thing the user typed as the command on their command line.

Modify your `tail` program to check `argv[0]` to see if your program was run as '`tail`' or '`head`'. If it was run as '`head`' then make the '`+`' option *illegal*, but behave as if the user had entered '`+1`' as a command-line option. Your program should now behave like either the system '`tail`' or '`head`' program, depending on which name was used to run it.

Hint: there are lots of ways the user could run your program. For example, you will have to ignore any directory components in `argv[0]` and check only the final file name component of the path.

▷▷ Ask an instructor to check and record your work.