

Assignment 1

Due Jan 31 by 4pm **Points** 5


Assignment A1: Intro C

For this assignment, you will be writing two command-line utilities: a simple game of life and a program that counts member references in other programs. Both programs will require that you use command-line arguments. The first also requires that you use arrays and the second requires that you process standard input (using `scanf`).

Please remember that we are using testing scripts to evaluate your work. As a result, it's very important that (a) all of your files are named correctly and located in the specified locations in your repository and (b) the output of your programs match the expected output precisely.

Part 0: Getting Started (0%)



Follow the instructions carefully, so that we receive your work correctly.

Your first step should be to log into [MarkUs](https://markus.cdf.toronto.edu/csc209-2019-01/)  (<https://markus.cdf.toronto.edu/csc209-2019-01/>) and navigate to the **a1: Intro C** assignment. Like for the labs, this triggers the starter code for this assignment to be committed to your repository. You will be working alone on this assignment, and the URL for your Git repository can be found on the right hand side of the page.

Pull your git repository. There should be a new folder named `a1`. All of your code for this assignment should be located in this folder. Starter code, including a Makefile that will compile your code and provide a sanity check, has already been placed in this folder. To use the Makefile, type `"make test_life"`, `"make test_trim"`, or `"make test_trcount"`. That will compile the corresponding program and run our sanity checks. To remove `.o` files and the executable, type `"make clean"`.

Once you have pushed files to your repository, you can use MarkUs to verify that what you intended to submit was actually submitted. The Submissions tab for the assignment will show you what is in the repository, and will let you know if you named the files correctly.

Part 1: Game of Life (life.c and life_helpers.c) (2.5%)

You will write a C program called `life.c` that implements a 1D variant of the [Game of Life](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)  (https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life) called [Rule 90](#) .

(https://en.wikipedia.org/wiki/Rule_90). You'll also write two helper functions in `life_helpers.c`.

Details

Your program reads two command line arguments, the first represents the initial state and the second represents the number of states to print. Your program will then print the number of states specified, one per line, starting with the initial state. For example, given the initial state `.....X.....X.....X.....` and 10 states, the program would print:

```
.....X.....X.....X.....
....X.X.....X.X.....X.X.....
...X...X...X...X...X...X...
...X.X.X.X.X.X.X.X.X.X.X...
..X.....X.....X.....
.X.X.....X.X.....
...X.....X.....
...X.X.....X.X...
..X...X.....X...X...
.X.X.X.X.....X.X.X.X.
```

Requirements

You may assume that, if given, the first argument is an initial state made up of only the characters "." and "X" and, if given, the second argument is an integer greater than or equal to 1. For this program, the user might still forget command-line arguments altogether. When this happens, your program shouldn't crash. Instead you should print to standard error the message "USAGE: life initial n" (followed by a single newline `\n` character), and return from `main` with return code `1`. The starter code actually implements this behaviour for you already. Following the standard C conventions, `main` should return 0 when the program runs successfully.

Along with the `main` function, you are required to write (and use) two helper functions. (These helper functions are to be implemented in the file `life_helpers.c`. The `main` function is in `life.c`.) The first helper function is named `print_state`. It takes a character array and the size of that array as its arguments, and returns `void`. This function's job is to print the characters of the given array following by a single newline character.

The second helper function is named `update_state` and it takes a character array and the size of that array, and returns `void`. This function should update the state of the array according to the following rules:

- the first and last elements in the array never change

- an element whose immediate neighbours are either both "." or both "X" will become "."
- an element whose immediate neighbours are a combination of one "." and one "X" will become "X"

Command line arguments and return codes

For this program, you may assume that if exactly two command-line arguments are provided, they will have the correct format. If the user calls the program with too few or too many arguments, the program should not print anything to stdout (only the provided message to stderr), but should return from main with return code

1. Following the standard C conventions, main should return 0 when the program runs successfully.

Part 2: Memory Reference Traces (trim.c and trcount.c) (2.5%)

In this part of the assignment, we examine how programs use memory by tracing the memory usage of a program and then counting the number of accesses to different parts of memory. Before we explain your tasks, we start by explaining how to generate memory traces.

Generating a memory trace

We use a program called `valgrind` to print information about every memory access made by a running program. This information includes the type of access, the address, and the number of bytes accessed.

`valgrind` runs on Linux (and is installed on the teach.cs machines). It does *not* run on Mac or Windows in a way that is useful for this assignment. The output of `valgrind` is saved to a file, and your program will read this file. Since getting exactly the output we want is a little complicated, we have given you a sample input file in your repository, and have included below some instructions and programs to help you generate some real traces. If you download these to your account on teach.cs, and run `make traces`, the four programs will be compiled and use `runit` to generate a trace for each program.

[trace-programs.zip](#) contains the following files:

- `Makefile`
- `runit` - a shell program that produces a memory reference trace from a given program. (Read the comments in the file.)
- `heaploop.c` - a program that iterates over data allocated in the heap
- `matmul.c` - a program that multiplies two randomly generated matrices. The size of the matrices is given as a command line argument.
- `simple.c` - a program that assigns values to three variables
- `stackloop.c` - a program that iterates over data allocated on the stack

You can download the zip file using a web browser on teach.cs. If you instead download it on to your own computer, you can use `scp` to transfer it to teach.cs (`scp trace-programs.zip username@teach.cs.toronto.edu:`). Once you have the zip archive on teach.cs, you can run `unzip trace-programs.zip` to expand it.

Note: Do not commit these programs or the traces they generate. Only the traces provided with the starter code should be committed to the repo.

In the output of `runit`, each line represents one memory access. The first non-blank character will always be one of 'I' (Instruction), 'S' (Store), 'M' (Modify), or 'L' (Load). Then there will be one or two spaces followed by an address in hexadecimal format, followed by a comma, and a number that represents the size in bytes of this memory access. In the trace file snippet below, there are two Instruction references, one Load reference and one Store reference.

```
I 0400baa6,7
S 04226ce0,8
I 0400baad,1
L fff000948,8
```

The `trace-programs` directory also contains several small C programs that exhibit different memory reference behaviours. Let's look at `simple.c`. The main thing to notice in this program are the lines that refer to `MARKER_START` and `MARKER_END`. They are declared as `volatile` which prevents the compiler from re-ordering access to these variables. The addresses of these variables are written to a file called `marker-simple` in hexadecimal.

We set a value for `MARKER_START` just before the "interesting" part of the program, set a value for `MARKER_END` at the end of the "interesting" part of the program. The addresses for these variables will appear in the tracefile at the point where the variables are assigned a value. This will allow us to "trim" the trace file to keep only the interesting memory references.

Task 1: Trimming the trace file (trim.c)

The first task is to write a program, `trim.c` that reads a file containing the output of `runit`, and outputs only the memory references between the marker addresses.

Details

- The program `trim` takes two arguments. The first is the name of (or path to) a trace file produced by `runit`. The second argument is the name of a "marker" file: a file containing the start and end address

markers that define which part of the trace file we want to keep. The marker addresses are not kept.

- The marker file will contain two hexadecimal numbers on a single line. The first is the starting marker address, and the second is the ending marker address.
- When using `fscanf`, the format conversion specification for a hexadecimal address value is `%lx`. To print a hexadecimal literal use `%#lx` so the initial "0x" is printed.
- Assume that all input files are formatted correctly and you do not need to do any error checking on the file format.
- If the `start_marker` address does not appear in the trace file, then the output of `trim` will be empty. If the `end_marker` does not appear in the trace file then the remainder of the trace file will be printed.
- You must use the output format strings specified in the `printf` statements in the starter code.

Example marker file from running `simple` on `teach.cs.toronto.edu`:

```
0xffff00090a 0xffff00090b
```

Example output from `simple`:

```
I,0x400769
S,0x601070
I,0x400773
S,0x60106c
I,0x40077d
L,0x601070
I,0x400783
L,0x60106c
I,0x400789
I,0x40078b
S,0xffff00090c
I,0x40078e
```

Now that we are only looking at the the memory references for a small part of the program, you can see where we are setting values for the variables. The first two "S" references are storing values for `i` and `j`. The third "S" reference is storing the value for `k`. You can see that it is a memory reference on the stack because of the high value of the address. (The `f`'s give it away.)

Task 2: Counting memory references (`trcount.c`)

The next step is to complete the program `trcount.c` that prints out counts of the different types of instructions.

If `trcount` has one argument then it is the name of a trace file from which to read. If it has no arguments, then it will read from `stdin`.

For example the output of `trcount` given using the example trace output above is:

```
Reference Counts by Type:
  Instructions: 7
  Modifications: 0
  Loads: 2
  Stores: 3
Data Reference Counts by Location:
  Globals: 4
  Heap: 0
  Stack: 1
```

Note that you must use **exactly** this format in your output. The formatting strings for the `printf` statements are given to you in the starter code to make this easy for you.

Values for Instructions, Modifications, Loads, and Stores are determined by counting the number of times each type of reference appears in the trace.

The starter code contains several constants that define the start and end of different regions of memory. For example, all references to heap memory occur between address `0x4000000` and `0x8000000`. Due to complexities in how programs are laid out in memory, these values were determined for the set of example programs provided on the teach.cs machines. If you try this out on other machines, these constants might be different.

These constants are used to determine which of the three regions of memory each reference comes from. Your program will only count **data** references (Load, Store, Modify) and will not count instruction references.

Command line arguments and return codes

For the programs in part 2, you may assume that if the correct number of command-line arguments are provided, they will have the correct format. If the user calls the program with too few or too many arguments, the program should not print anything to stdout (only the provided message to stderr), but should return from main with return code 1. Following the standard C conventions, main should return 0 when the program runs successfully.

Reminder

Your programs must compile on teach.cs using gcc with the `-Wall` option and should not produce any error messages. Programs that do not compile, will get 0. You can still get part marks by submitting something that doesn't completely work but does some of the job -- but it **must at least compile** to get any marks at all. Also check that your output messages are **exactly** as specified in this handout.

Submission

We will be looking for the following files in the a1 directory of your repository:

- life.c
- life_helpers.c
- trim.c
- trcount.c

Do not commit .o files or executables to your repository.