Project 2: Latin Squares

Due Nov 20, 2021 by 11:59pm **Points** 150 **Submitting** a file upload **File Types** c
Available Oct 29, 2021 at 12am - Dec 24, 2021 at 11:59pm

This assignment was locked Dec 24, 2021 at 11:59pm.

Submission

✓ **Submitted!**
Nov 20, 2021 at 11:19pm
Submission Details
Download
latin_square_functions-4.c

Grade: 150 (150 pts possible)
Graded Anonymously: no
View Rubric Evaluation

Comments:
Last graded at: 11/20/2021 11:07:27
Bujji Selagamsetty, Nov 20, 2021 at 11:07am
Last graded at: 11/20/2021 19:35:33
Bujji Selagamsetty, Nov 20, 2021 at 7:35pm
Last graded at: 11/20/2021 19:49:46
Bujji Selagamsetty, Nov 20, 2021 at 7:49pm
Last graded at: 11/20/2021 23:22:23
Bujji Selagamsetty, Nov 20, 2021 at 11:22pm

Project 2: Latin Squares

A Latin Square is an n x n puzzle filled with n different symbols. Each symbol occurs exactly once in each row and exactly once in each column. The following three puzzles are all examples of valid Latin Squares.

```
abcd        123        %#
bcda        312        #%
cdab        231
dabc
```

Corrections and Additions

1. None yet.

Learning Goals

This project will help you gain familiarity working with files, dynamic memory allocation, pointers, and multidimensional arrays. Additionally, you'll gain experience using a Linux tool, Valgrind, to evaluate the memory consumption of your program.

Specifications

In this project, we will read an nxn puzzle from a file and verify it is a valid Latin Square. You will complete four functions for this project. Templates for the functions can be found in the file latin_square_functions.c . **Please see the template file for the exact function description and specifications.**

- Read_Latin_Square_File(...) reads the puzzle board from a file, reserves memory for the puzzle with malloc, and fills in the data structure.
- Verify_Alphabet(...) counts the symbols used and verifies that exactly n symbols are each used exactly n times.
- Verify_Rows_And_Columns(...) Identifies rows or columns that use symbols more than once.
- Free_Memory(...) calls free to return the memory reserved by malloc when we're done with it.

The primary data structure used is declared in latin_square_main.c on line 41.

```
    char **latin_square = NULL;
```

This variable can be thought of as a pointer to an array of strings. Or a pointer to an array of character arrays. You will need to make calls to dynamically allocate memory twice; first, to reserve space for an array of pointers to the arrays of characters. Then for each row, you will need to call malloc again to reserve space for the character arrays representing each row of symbols.

All puzzles tested will be square. They will have the same number of rows as columns.

All work must be done in the latin_square_functions.c file, and your code must run with the original latin_square_main.c and latin_square_functions.h.

Valid symbols or characters include all printable symbols. Those with ascii codes 33 (!) to 126 (~) inclusive.

This means that the largest possible valid puzzle will be 94x94. There is no upper limit to the size of an invalid puzzle. We will not test your work with puzzles larger than 1000x1000.

No [ ] Allowed Your Functions

A key objective of this assignment is for you to practice using pointers. To achieve this, you are **not allowed** to use array indexing with square brackets to access arrays in the latin_square_functions.c file. Instead, you are required to use address arithmetic and dereferencing to access arrays. You may not use brackets, e.g., writing “latin_square[1][2]” to access character (1,2) of the latin square. Submitting a solution using indexing to access the latin square or any other array will result in a **50% reduction of your score**.

No External Libraries

All libraries needed for the completion of this have been included for you already. It is not permitted to use any external C libraries beyond those present in the starter files.

No Memory Leaks

Your code must terminate with no memory leaks. See the Valgrind section below for more details on how to check your program for memory leaks.

Files

- latin_square_main.c // your code should work with the original version of this file
- latin_square_functions.h // header file
- latin_square_functions.c // do your work here
- Makefile // simple makefile to make compiling and using Valgrind easier
- Test Files
  - Latin_4x4.txt // a valid puzzle
  - Latin_5x5.txt // a valid puzzle
  - Latin_4x4_Invalid.txt // an invalid puzzle - fails both alphabet and rows/columns tests
  - Latin_4x4_RC_Invalid.txt // an invalid puzzle - fails the rows/columns test
  - Latin_5x5_Alphabet_Invalid.txt // an invalid puzzle - fails the alphabet test
  - You will need to write several more test files to verify your work. Feel free to share your test puzzles on Piazza

Compiling

For this (and maybe future assignments) we will be providing you with basic Makefile to ensure that you are compiling and using your code in the same way that our grading setup does. If you are curious about this, feel free to inspect and make any additions to the Makefile . Just remember that if you modify the commands originally provided, you test your code with the unmodified versions of these commands before submitted, as our grading scripts will use the original version of this file. Thus, to compile your code, all you need to run is:

```
make build
```

Note that to get all the points in the assignment, your code must compile **without any warnings or errors** from running the above command.

Running

Running the command produces an executable file called latin_square. For this assignment, we will be providing input files via command line arguments, and so, to run your code with the example Latin_4x4.txt file, you would enter the following command to the terminal:

```
./latin_square Latin_4x4.txt
```

Strategy

Write your code in small pieces and test each line written by printing out a message. This technique is called scaffolding. Remove the debugging messages after you verify the success of your code.

Start working in main first and transfer working code into the functions when ready.

You might find it useful to use array notation with [] to begin, but then after your code is working, you must upgrade to use pointers instead to access the arrays.

Resources

See sections 7.5 and 7.7 of K&R for help working with Files

DynamicMemoryAllocation.pdf

Chapter 5 of K&R for help with pointers.

Check out the Matrix Transpose Example beginning at about 55 minutes.

Turn in

Upload your latin_square_functions.c file to Canvas.

Style

Please follow the style guide linked on the Canvas Homepage.

Valgrind

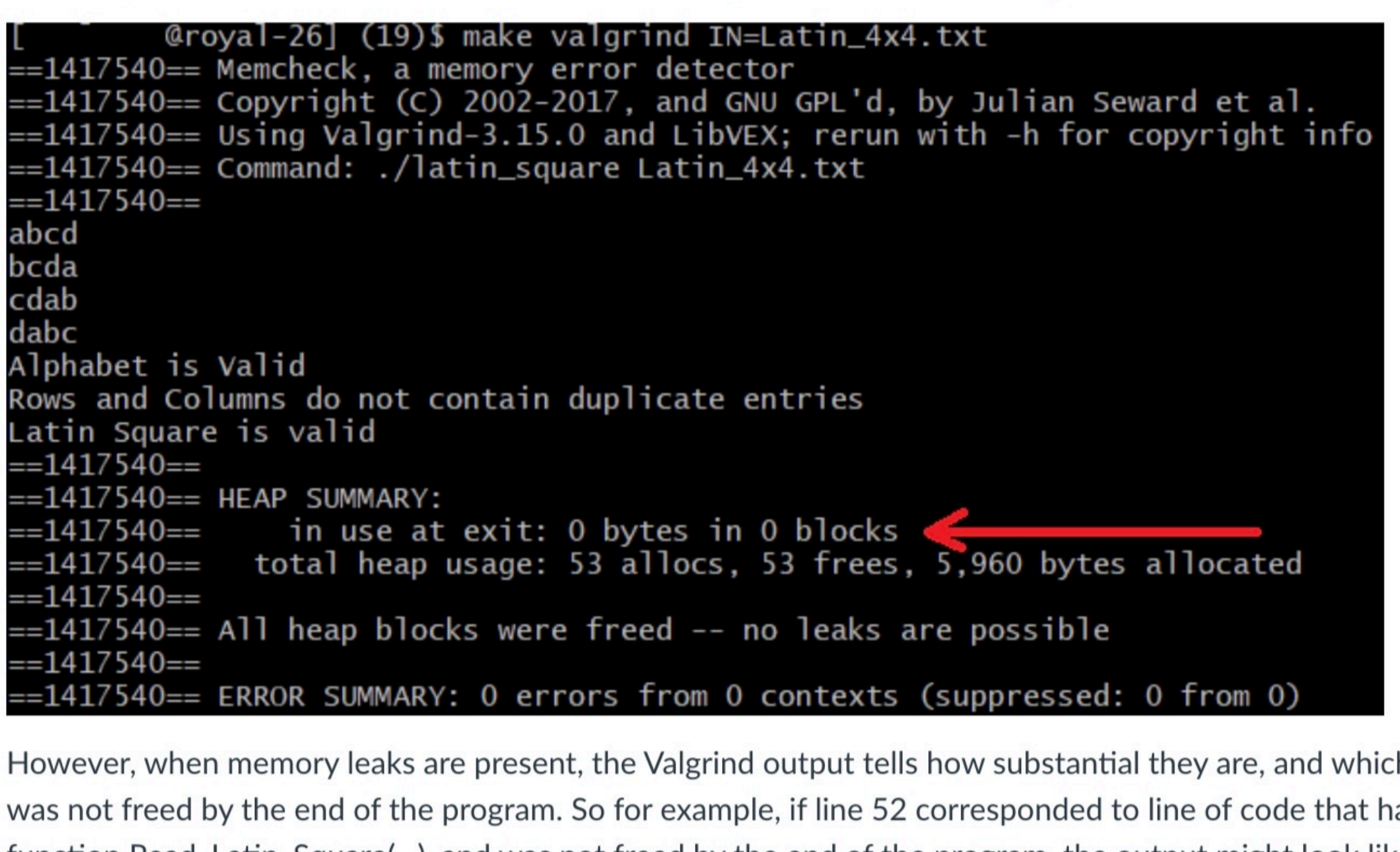
Valgrind is a useful tool that is included with many Linux distributions including the CSL machines. It is a analysis tool that evaluates the memory footprint of your program after it runs. In this assignment, you will be using the malloc family of functions to dynamically allocate various chunks of memory for use in your program. A common programming error is forgetting to free up all the memory that your program uses. Forgetting this (or doing this incompletely) can cause serious performance issues on the system running your code, as an OS needs to know that you are done using the memory it had provided you with (via calls to malloc/calloc/realloc/reallocarray). The free() function fulfills this role of notifying the OS that the memory can be reclaimed.

Using Valgrind

```
make valgrind IN=Latin_4x4.txt
```

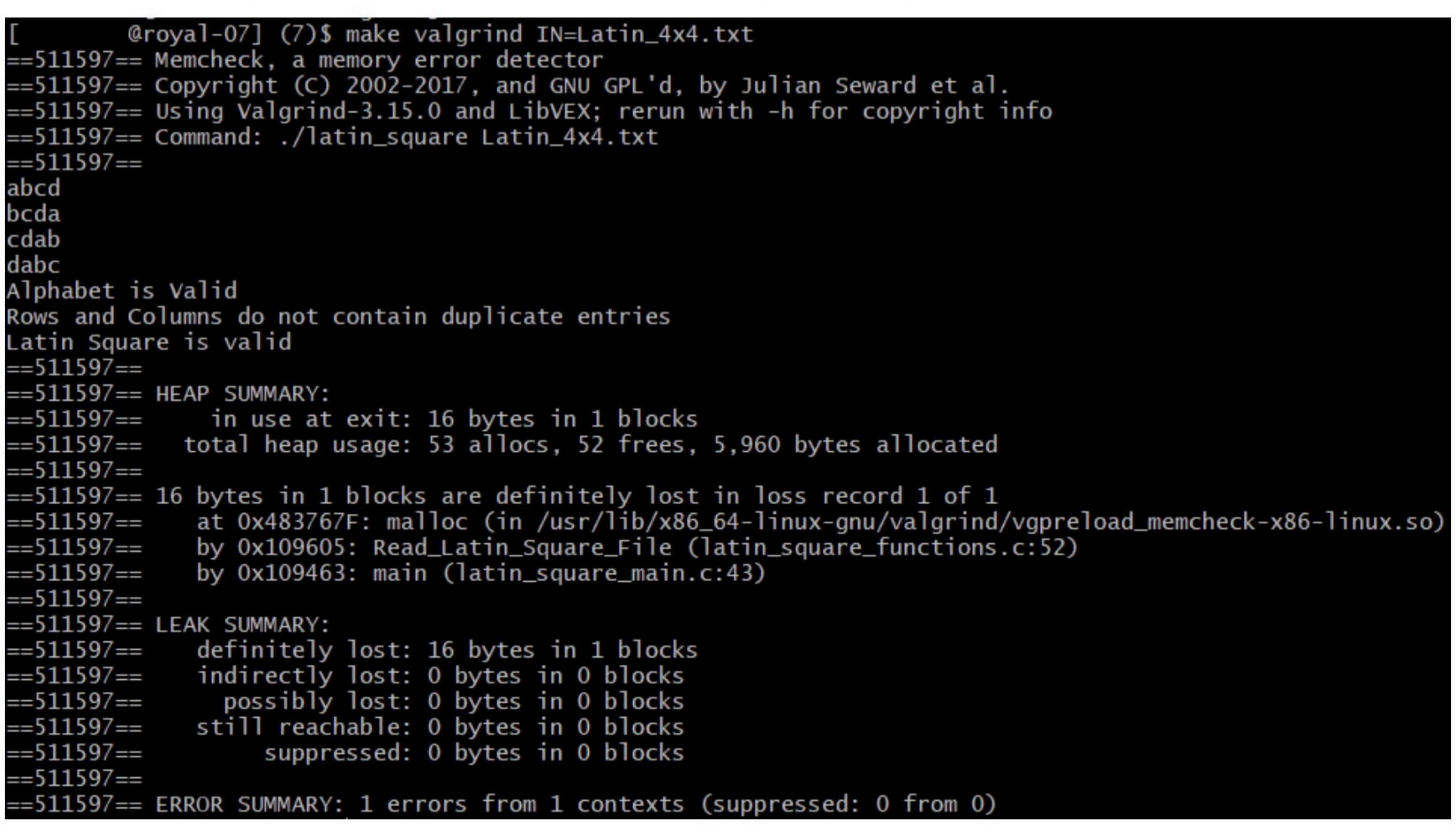
Interpreting Valgrind Output

In the ideal case, every byte that was dynamically allocated was freed, and thus the output from Valgrind is very uninteresting. Note the arrow indicating that no heap allocated memory is still being used at end of our program:



```
[        @royal-26] (19)$ make valgrind IN=Latin_4x4.txt
==1417540== Memcheck, a memory error detector
==1417540== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1417540== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==1417540== Command: ./latin_square Latin_4x4.txt
==1417540==
abcd
bcda
cdab
dabc
Alphabet is Valid
Rows and Columns do not contain duplicate entries
Latin Square is valid
==1417540==
==1417540== HEAP SUMMARY:
==1417540==     in use at exit: 0 bytes in 0 blocks
==1417540==   total heap usage: 53 allocs, 53 frees, 5,960 bytes allocated
==1417540==
==1417540== All heap blocks were freed -- no leaks are possible
==1417540==
==1417540== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

However, when memory leaks are present, the Valgrind output tells how substantial they are, and which line in the code requested memory that was not freed by the end of the program. So for example, if line 52 corresponded to line of code that had a call to malloc, was located in the function Read_Latin_Square(...), and was not freed by the end of the program, the output might look like this:



```
[        @royal-07] (7)$ make valgrind IN=Latin_4x4.txt
==511597== Memcheck, a memory error detector
==511597== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==511597== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==511597== Command: ./latin_square Latin_4x4.txt
==511597==
abcd
bcda
cdab
dabc
Alphabet is Valid
Rows and Columns do not contain duplicate entries
Latin Square is valid
==511597==
==511597== HEAP SUMMARY:
==511597==     in use at exit: 16 bytes in 1 blocks
==511597==   total heap usage: 53 allocs, 52 frees, 5,960 bytes allocated
==511597==
==511597== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==511597==    at 0x483767F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-x86-linux.so)
==511597==    by 0x109605: Read_Latin_Square_File (latin_square_functions.c:52)
==511597==    by 0x109463: main (latin_square_main.c:43)
==511597==
==511597== LEAK SUMMARY:
==511597==    definitely lost: 16 bytes in 1 blocks
==511597==    indirectly lost: 0 bytes in 0 blocks
==511597==      possibly lost: 0 bytes in 0 blocks
==511597==    still reachable: 0 bytes in 0 blocks
==511597==         suppressed: 0 bytes in 0 blocks
==511597==
==511597== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Project 2 Latin Squares

Criteria	Ratings		Pts
RawCompiles	**2 pts** **Full Marks**	**0 pts** **No Marks** The file you submitted had compile time warnings, errors, or both. Please resolve these issues and resubmit.	2 pts
LibCheck	**2 pts** **Full Marks**	**0 pts** **No Marks** Your solution seems to depend on an external library that was not permitted. Please remove this dependency and resubmit.	2 pts
Read_Latin_Square_File	**40 pts** **Full Marks**	**0 pts** **No Marks** Your solution didn't seem to produce the correct output when testing the Read_Latin_Square_File function. The issue could be: left file open, incorrect memory allocation, incorrect setting of n, the values written to latin_square did not match those from the file passed in, etc.	40 pts
SquareBracketsCheck	**1 pts** **Full Marks**	**0 pts** **No Marks** Your solution used stack allocated arrays or addresses into a heap allocated array using the bracket notation. As per the specification, neither of these are permitted and doing either of these results in a 50% reduction in your score. Please resolve these errors and resubmit.	1 pts
Verify_Alphabet	**20 pts** **Full Marks**	**0 pts** **No Marks** Your Verify_Alphabet function returned the wrong value. Please revise your solution and resubmit.	20 pts
Verify_RC_Ret	**20 pts** **Full Marks**	**0 pts** **No Marks** Your Verify_Rows_and_Columns function returned the wrong value. Please revise your solution and resubmit.	20 pts
Verify_RC_Out	**20 pts** **Full Marks**	**0 pts** **No Marks** Your Verify_Rows_and_Columns function produced the wrong output. Please revise your solution and resubmit.	20 pts
MemCheck	**45 pts** **Full Marks**	**0 pts** **No Marks** When running your program, Valgrind detected memory leaks, warnings, and/or errors. Please resolve these and resubmit. This may be a result of an incorrect Free_Memory() function or memory leaks from other functions. Please resolve these errors and resubmit.	45 pts
Stats	**0 pts** **Full Marks** Comments contain how long it took to run your code on the largest valid test, as well as how much of the heap your program used.	**0 pts** **No Marks**	0 pts
			Total Points: 150