This course has already ended.

« 2 Multidimensional arrays Course materials 4 Round feedback »

ELEC-A7100 / 7. Multidimensional Arrays / 3 Command line arguments

Seventh round tasks as a `ZIP file.`

This section is mainly for **multidimensional arrays**, but before dealing with multidimensional arrays, a brief introduction is given to **Enumerated data types**, which are used to improve readability of the program.

Multidimensional arrays is not a new feature of the C language, but they are arrays, which consist of arrays. Arrays can be built either statically or dynamically. Especially in the latter case, you must be careful with memory allocations, use and releasing memory.

Command line arguments¶

Command line parameters are a traditional form of passing instructions and information to the program when it is started from a command line terminal interface (or from a script). With graphical user interfaces and embedded mobile gadgets, command line parameters may not have much significance, but they are a very common interface for controlling the application behavior in Unix systems. We will discuss them briefly here.

Until this point, we have seen several examples of a main function that does not have parameters. The main function also has an alternative form that allows passing the command line arguments that were given when program was started. The command line parameters are represented as an array of strings. The following small piece of code illustrates how they work:

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[]) {
4      printf("This program was called with %d parameters\n", argc);
5      printf("They were (including the program name itself):\n");
6      for (int i = 0; i < argc; i++) {
7          printf("%s\n", argv[i]);
8      }
9  }
```

Here the main function comes with two parameters: **argc** tells the number of command line parameters, and **argv** is an array of strings, where each element contains one command line parameter. The program name itself is the first command line parameter, so there at least one array member is always included. For example, when user types the following on the command line:

```
./program one two three
```

argc will be 4, and the **argv** array will have the following content: `{ "./program", "one", "two", "three" }`.

A common format on command line is to use command line options that start with dash (-) and are followed by a single character. Such options can be standalone, or come with an additional parameter. For example, consider **tail** command below:

```
tail -f -q -n 10 file.txt
```

Command is given with three options, 'f', 'q' and 'n'. 'f' and 'q' are without additional parameters, but 'n' comes with an integer parameter 10. Finally, there is a command line parameter that does not follow the option format. Typically the options can be given in any order.

Task 7.4: Split string¶

In this exercise you will split string based on another string. For example `"String to be split"`, which is split by `" "`, will result in an array containing the split parts: `["String","to","be","split",NULL]`. The resulting arrays will always end with a NULL pointer.

The exercise has two parts:

a) Implement a function `void print_split_string(char** split_string)`, which prints a split string, each part on its own line. For example when calling the function for a split string `["String","to","be","split",NULL]`, the function prints:

```
String
to
be
split
```

b) Implement functions

- `char** split_string(const char* str, const char* split)`

  The function splits a string in the aforementioned way and returns the parts in a dynamically reserved array. Parameters are as follows:

  - `const char* str`, the string to be split
  - `const char* split`, the string used to split `str`,

    for example `"One. .more. .test. .for. .string. .splitting."` and `". ."`, when the result is an array consisting of the following pieces: `["One","more","test","for","string","splitting.",NULL]`

  Also, the pieces need to have their memory dynamically reserved and they need to be copied to the array. The function `strstr` may be helpful in this exercise.

- `free_split_string(char** split_string)`

  The function frees the memory for the split string array.

Points 20 / 20 My submissions 9 Deadline Friday, 23 July 2021, 19:59 To be submitted alone

⚠ This course has been archived (Saturday, 31 December 2022, 20:00).

Split String

Select your files for grading

stringsplit.c

Choose File No file chosen

Submit

Task 7.5: Game of life¶

Objective: Practice allocating and manipulating dynamic two-dimensional arrays.

Conway's Game of Life is a traditional zero-player game that runs a cellular automaton based on an initial system state and a simple set of rules. Game of Life operates on a two-dimensional grid, where each cell can have two states: "dead" or "alive". The state of the grid progresses in discrete steps (or "ticks"), where each cell may change its state based on the states of its neighbors.

The rules determining state changes on an individual cell are:

- Any **live** cell with **fewer than two live** neighbors **dies**
- Any **live** cell with **two or three live** neighbors **lives** on to the next generation.
- Any **live** cell with **more than three live** neighbors **dies**
- Any **dead** cell with **exactly three live** neighbors becomes a **live** cell.

Also diagonal cells are considered neighbors, i.e., each cell (that is not on the edge of the area) has 8 neighbors. Remember to consider cells that are on the edge of the area: you should not access positions that are out of the bounds of the area.

The state changes on each cell occur simultaneously, (i.e.), as you process the area one cell at a time, the intermediate changes made earlier during the current iteration must not affect the result.

For example, a area with the following setting ('*' indicates live cell):

```
..........
.**.***...
.**.......*
.*.*.....*
........*.
```

will in the next generation be:

```
.....*....
.***.*....
*...**....
.*.......**
..........
```

The wikipedia page on Conway's Game of Life gives further background, and examples of some interestingly behaving patterns.

In this exercise you will implement the necessary components for Game of Life such that you should be able to follow the progress of the game area between generations. The main function in **main.c** has the core of the game, that generates the area and goes through the generations one by one using the functions you will implement.

The exercise has the following subtasks:

a) Create and release game area

Implement the function **creategameArea** that allocates the needed space for the **GameArea** structure that holds the game area, and for a two-dimensional array of dimensions given in parameters 'x_size' and 'y_size'. Each cell in the area must be initialized to 'DEAD' status, as well as the 'x_size' and 'y_size' areas in the structure.

Note: the tests assume that the pointers to the rows (y-axis) of the area are the first dimension of the array (and allocated first), and the rows are the second dimension. I.e., the cells are indexed as [y][x]. See the picture in dynamically allocated multi-dimensional array.

You will also need to implement the function **releaseGameArea** that frees the memory allocated by createGameArea(). The tests will use this function for all memory releases, so failing to implement this will result in Valgrind errors about memory leaks.

b) Initialize area

Implement the function **initGameArea** that turns a given number of cells into 'ALIVE' state. You can decide the algorithm by which you set up the area, but the outcome must be that exactly 'n' cells in the area are alive. One possibility is to use the *rand* function to choose the live cells randomly.

c) Print area

Implement the function **printGameArea** that outputs the current state of the area to the screen. Each dead cell is marked with a period ('.') and each live cell is marked with an asterisk ('*'). There are no spaces between the cells, and each row ends in a newline ('\n'), including the last line. The output should look similar, as in the above examples in this task description.

d) Progress game area

Implement the function **gameTick** that advances the game area by one generation according to the rules given above. Note that all cells need to be evaluated "simultaneously". One way to do this is to maintain two game areas: one that holds the state before the transition, and another where you will build the next generation of the game area.

(If you allocate new memory in this function, don't forget to release it.)

Hint: When counting neighbors, don't go through the border cases separately with conditions. Instead, use loops and before accessing the cell value check that it is not out of bounds.

Points 25 / 25 My submissions 3 Deadline Friday, 23 July 2021, 19:59 To be submitted alone

⚠ This course has been archived (Saturday, 31 December 2022, 20:00).

Game of Life

Select your files for grading

gameoflife.c

Choose File No file chosen

Submit