Applications on arrays in C (TIC TAC TOE)

In this lab, we continue the introduction of the ‘C’ language we have started two discussions ago. So far we have seen how ‘C’ code is translated into assembly code and compiled into an executable. We have also seen how to implement a simple linked list in ‘C’. In this discussion, we continue examining how to allocate arrays and access them to implement some programs such as the `TIC TAC TOE` game in this case.

It is a good idea to partner up for this lab so that you can discuss and reflect upon what you are doing.

Discussion goals

1. To use preprocessor directives like (`#include` and `#define`).
2. To understand arrays and how to access them.
3. To write modular code: functions prototype and labels.
4. To write recursive functions.
5. To implement the remaining game functions.

Setup

Follow the post on piazza to create your github classroom lab repository.

Log into your UNIX environment and clone the repository and change directories into the repository working copy.

The lab repository includes both the skeleton code `tictactoe.c`, a complete solution `sol.c`, and a `Makefile`. The goal is to complete the skeleton in order to develop your version of the program. We encourage you to avoid looking at the solution until you have attempted to develop your own.

Note that we have also included a file called `csetup.gdb` to setup gdb in a way that is nice for debugging programs that were built from C source code.

Getting started

Before getting to understand the game logic and to write the code, let’s first touch on a few important concepts that will make our life easier implementing the logic.

Preprocessor directives.

We have seen before in the assembly language that we could translate an assembly code file into an object file using the assembler; We have also seen that we could translate a `C` language code file into an object file using the compiler. However, before the compiler starts its magic, there is an initial step called preprocessing. What is the goal of this step? Sometimes, it is easier for programmers to not write code, but to write code that writes code. The preprocessor translate that “code that writes code” for us. It takes some file as an input and searches for some find and replace patterns to execute. Once all the patterns are resolved, new files are generated. Those new files are the files then fed to the compiler to translate into an object file.

Take the following `C` language snippet for example and see how the pre-processor would do its magic to create a new file understandable by the compiler.

```
#define PLAYERS_NUMBER 2

int Players_ids[PLAYERS_NUMBER];
int Players_scoress[PLAYERS_NUMBER];
```

The above code snippet is translated into the following code by the preprocessor:

```
int Players_ids[2];
int Players_scoress[2];
```

How was this important? It provides a sustainable way to change things because instead of going everywhere inside the code to change a small number, we just need to modify the value in one place where the `#define` is written. Moreover, it makes your code more readable. It is better to read `int Players_ids[PLAYERS_NUMBER];` than reading `int Players_ids[2];` and having no idea what this `2` means. When code gets more complex, those small details really really matter.

One other directive we have seen before in addition to the `#define` directive is the `#include` directive. This later directive mainly tells the preprocessor to go to some file, grab its content, and add to this current file.

For more about preprocessor directives, check the following optional tutorial

Arrays Declaration

We have seen before examples where using arrays came handy; for example we used arrays to represent vectors which we manipulated in the calculator assignment. Also, we have used arrays to represent strings. Note that a string is nothing but a null-terminated array of characters. Similar to the concept where we used arrays to describe vectors, we can use what we call 2-D arrays to represent matrices. Good examples for real world applications for matrices is storing the state for a chess board or in our discussion today to store the state of the `tic tac toe board`.

2D-arrays are called this way because we access its data using 2 access indices. We can create a new 2D-array using the following syntax in `C` language.

```
#define ROWS_NUMBER 3
#define COLUMNS_NUMBER 3
// This create a new 2D array called "Array_name"
// that has "ROWS_NUMBER" rows.
// Each row has "COLUMNS_NUMBER" elements of type "data_type".
data_type Array_name[ROWS_NUMBER][COLUMNS_NUMBER];

// We then specify which "i"th row to choose
// and which "j"th element to choose from this "i"th row.
int i = 0; int j = 2;
Array_name[i][j] = 5;
```

Something worth noting in the previous example is that the way we created the above example is called static allocation. Static in this sense means that at the time of compilation we knew how large our array should be because we asked to allocate a size that was defined using a preprocessor directive.

Also, We could have initialized the previous array using the following syntax:

```
int Array_name[ROWS_NUMBER][COLUMNS_NUMBER] = {
    {01, 02, 03},                // the row where i = 0
    {11, 12, 13},                // the row where i = 1
    {21, 22, 23}                 // the row where i = 2
};

// Array_name[1][2] is equal to 12
```

Writing modular code: Functions and labels

It is good practice to divide the code implementation into different parts where each part has its own homogeneous role. That again has two main advantages; it makes code development more sustainable and it makes code understanding easier.

Moreover, in the process of developing or implementing some solution, you may want to construct the main logic first and then fill in the details as you get more ideas. In other words, we write what we expect to have our code do abstractly using high level logic and then start implementing those high level functions later on.

For example, the following could be a very simple abstract way to write our game and then as we go deeper to start implementation, we find the need to have more and more functions.

```
// Board functions
void Board_display(void);

// Player functions includes game logic
bool Player_move(int player);

int main(void)
{
  Board_display();
  while (true) {
    if (Player_move(PLAYER_ONE) == true) break;
    Board_display();
    if (Player_move(PLAYER_TWO) == true) break;
    Board_display();
  }
  Board_display();
  return EXIT_SUCCESS;
}
```

Notice in the previous code snippet we have declared a new function called `void Board_display(void);`. You might wonder if such a line is allowed or how does the program know what this function do. `C` enables us to actually separate both the function declaration and implementation from each other. The previous line without implementation is called a function prototype. Function prototypes indicate the function name, parameters, and return type. The reason we need them is because the compiler starts doing its magic line by line. So for example if we use the function `Board_display` inside the `main` function but the implementation of this function is written after the implementation of the `main` function, we will get an error. One way to fix this error is by simply writing the function prototype before the `main` function. This way the compiler knows what to expect.

Similar to assembly language, `C` languages gives us the option to use labels. Labels are created very similar to the assembly language as follows.

```
label_1:
  x = 5;
```

You can jump to this label inside your code by using the following statement:

```
goto label_1;
```

Recursive functions

We can split functions in terms of how they are constructed into two approaches; `iterative function` and `recursive function`.

`Iterative functions` simply takes a problem instance input and then work on solving it within just one call like the following instance.

```
int iterative_sum(int array[], int size){
    int sum = 0;
    for (int i = 0; i < size; ++i){
        sum+= array[i];
    }
    return sum;
}
```

On the other hand, `recursive functions` take the input and reduce it to a smaller problem that it can be given to the same function again. It keeps doing that till it reaches what we call `base case`. The `base case` is the simpliest problem instance that we solve directly without further need for reduction.

```
int recursive_sum(int array[], int size, int i = 0){
    // Base case: no more elements in array
    if( i >= size){
        return 0;
    }

    // Not simple problem reduce it
    // to sum current element in addition to the sum for
    // the rest of the array.
    return array[i] + recursive_sum(array, size, i + 1);
}
```

Game Logic

Now that we understand our cool tools, let's see how we can use them in order to build our game.

The game logic is as follows:

1. The `TIC TAC TOE` has 9 cells that start as empty at the beginning of the game.
2. Players take turns in order to fill one of the 9 cells each time.
3. The first player to occupy 3 cells horizontally, vertically, or diagonally wins.
4. If all cells are occupied but no winning player, the game ends in draw.

The program is implemented using the following functions:

1. `main`: The main function that drives the whole game. It keeps the game running while no one won and there are still empty cells.
2. `Board_display`: Shows the current board state to the players using ASCII characters in the terminal.
3. `Player_move`: Tries to get valid input from the player. While the input is invalid, it keeps asking for a valid input. Once it gets valid inputs, it updates the game state and check if the program should terminate.
4. `Player_marker`: Returns the character 'x' or 'o' depending on whose turn it is.
5. `Board_num_blanks`: Returns the number of remaining empty cells.
6. `Player_is_winner`: Checks if the current state indicates that some player is a winner.
7. `Board_scan_line_for_marker`: Checks whether a horizontal, a vertical, or a diagonal indicates that this player is a winner.

Your goal is to implement the functions called `Player_is_winner` and `Board_scan_line_for_marker`.