Series 2: *C* Programming - Pointers 1

Reminder

Have you read the tips for these sets of exercises?

Exercise 1: Exploring memory (level 2)

The goal of this exercise is to develop a rudimentary memory "examinator": a function which, given an address and a size, displays the contents of each byte.

To do this, we propose to break your program down into several tool functions.

1.1 Display in binary

The appropriate type to represent a byte without worrying about the sign bit is "`unsigned char`" and not "`char`" whose sign interpretation is not defined ("implementation defined").

Start by setting the type "`byte`" as "`unsigned char`".

Next, since there is no format for `printf()` to display a variable in binary, you will write your own.
Write the function "`binary_print()`", that takes a `byte` and displays it in binary. (It's up to you to come up with your own algorithm for doing this; there are several possible solutions, more or less technically advanced.)

NOTE: to display a single character: "`printf("%c", a);`" can be written more efficiently and compactly as: "`putchar(a);`".

1.2 Display of the i-th byte

Write a `display()` function that takes a `size_t` and a byte and displays:

- the `size_t` (for example, we could use the `"%02zu"` `printf()`-format for this);
- the value of the byte in binary;
- the value of the byte as a positive integer; for example, we could use the `"%3u"` format for this;
- if the byte value is between 32 and 126 (inclusive), display it as a character.

For example, we could have the following result:

```
00 : 01010000  80 'P'
```

1.3 Displaying memory contents

Write a function `dump_mem()`, that takes a pointer to a byte and a size `N` (of type `size_t`) and displays:

- the starting address (to display an address, we use the `"%p"` format of `printf()`);
- the contents of all `N` bytes from the given address.

For example, we could have the following result:

```
Starting from 0x7fffb7ed88ac:
00: 01010000  80 ('P')
01: 00000000   0
02: 00000000   0
03: 00000000   0
```

To display addresses following a given pointer, simply use array syntax (we'll return to this later in the lectures): if `ptr` is your pointer, access its contents with `ptr[0]`, then content immediately after with `ptr[1]`, etc.; exactly as if it were an array.

1.4 Using the tool and understanding the contents of the memory

In the `main()`, define an `int` of value 80 (= 64 + 16), another of -80, a `double` of 0.5 and another of 0.1; then use your `dump_mem()` function to look at how these values are represented in memory.

Analyze/verify the results obtained (see former 1st year courses or perhaps here: http://www.binaryconvert.com/convert_double.ht

---

Exercise 2: dynamic arrays (pointers, level 2)

NOTES:

1. This exercise and the following require you have watched the "C02 B Dynamic Allocation" videos.
2. You must do this exercise on your own **without** copying the lecture. Otherwise there's no point!

[end of notes]

Fully implement the "dynamic array" data structure.
We must be able to: create, destroy a dynamic array, add an element at the end, change the value of an element at a given position, and read the value at a given position.

Pay particular attention to access integrity (the accessed location must be defined), and use intelligent conventions for return values in the case of non-compliant access.

Exercise 3: Matrix multiplications revisited (pointers + typedef, level 2 + 3)

Part 1: first improvement: exercise on pointers

Repeat last week exercise 7: copy your program `mulmat.c` (or its solution, if you have not done the exercise) into a new program. We will modify this program to use pointers

The main issue of last week's code is that the `read_matrix()` and `multiply()` functions create their own `Matrix`, which may be **copied** as output (exchange of information between the function's `return` and its call).
There are therefore two `Matrix` each time:

1. that of the instruction which makes the call; and
2. that of the return value of the called function.

This is costly and unnecessary.

A solution to avoid this duplication of `Matrix` is to use **pointers**.

Change the `read_matrix()` and `multiply()` functions so that they return a *pointer* to a `Matrix` (which they allocate themselves).

In the `main()`, declare three pointers to `Matrices`: `M`, `M1` and `M2`, then modify the program accordingly and calculate `M = M1 * M2`.

Remember to free the memory as soon as you no longer need it!

Part 2 (level 3, OPTIONAL): second improvement

Now suppose that we want to perform matrix multiplication several times in a row. The previous solution is not very satisfactory either because in this case it would be necessary **each time** to free the memory in the block calling the `read_matrix()` and `multiply()` functions, and **each time** these functions would allocate a new place in memory. Waste of time!

The good solution to avoid both local copies (part 1) and too frequent memory allocations/frees is that the `read_matrix()` and `multiply()` functions modify the value of an additional argument (passed by reference), which contains the result of the function and would be allocated/initialized by the calling block.

For more practical use of these functions (chaining), they also return the address of this modified argument. This allows the functions themselves to be used as arguments to other functions (chaining).

This solution would produce the following prototypes:

```
Matrix*  read_matrix(Matrix* read);
```

and

```
Matrix*  multiply(Matriy const * M1,
                  Matrix const * M2,
                  Matrix * result);
```

Copy the program again then edit it according to the above prototypes. Test it in the `main` with the following calls: (**Note:** *warning*, here `M1`, `M2`, `M3` and `M4` are again `Matrix` and no longer pointers to `Matrix`, hence the sign "&").

```
read_matrix(&M1);
multiply(&M1, read_matrix(&M2), &M3);
```

or

```
multiply(&M1, add(&M2, &M3, &temp), &M4);
```

Part 3 (level 2): third improvement

Transform the code so that matrices are allocated dynamically.

Provide the necessary additional functions: initialization, reallocation, and release(/free).

Exercise 4: IP network (pointers + typedef, level 2)

In the Internet network, the TCP/IP routing is done "step by step" (as you will see later in the course), each node in the network only knows its direct neighbors. This is what we are going to model here.

Define a data type that can represent a node having (at least) an address and an array of neighbors. The neighbors array will represented other nodes, *using* the same data type but *without copying*.

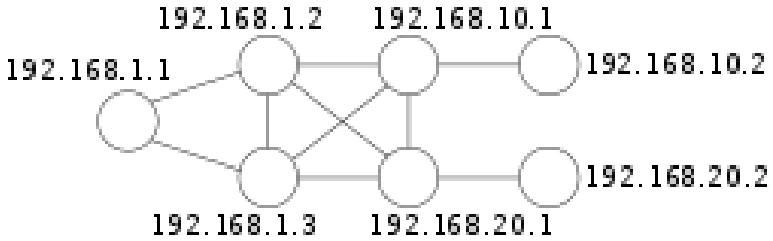
An address is simply an array of 4 positive integers of 8 bits each (i.e. between 0 and 255). To do this, you can use either an array of 4 `unsigned char`, or a `uint32_t` from C99 (library `<stdint.h>`).

Design the previous data type to be as dynamic as possible (no "hard" constants in the code). Also try to make your code as robust as possible.

Then write (at least) the following functions:

- `create()`: allows you to create a node by giving the four parts of its address (four `unsigned char`s) as an argument;
  at first, a node has no neighbors;
- `link_neighbors()`: takes two nodes as a parameter and creates a "neighborhood" link between these two nodes; this relationship is symmetrical (if A is a neighbor of B, then B is a neighbor of A), it will therefore be necessary to represent it as such (add a neighbor to each of the two nodes);
- `common_neighbors()`: takes two nodes as parameters and returns the number of common neighbors;
- `node_print()`: takes a node as a parameter and displays its address; it will also have to display the addresses of all its direct neighbors.

Finish your program by writing, in the `main()`, the code corresponding to the following situation:



Show node 192.168.10.1,
then show the number of common neighbors between 192.168.1.1 and 192.168.20.1,
then show the number of common neighbors between 192.168.1.2 and 192.168.1.3.

Example of output:

```
192.168.10.1 has 4 neighbors: 192.168.1.2, 192.168.1.3, 192.168.10.2, 192.168.20.1.
192.168.1.1 and 192.168.20.1 have 2 common neighbors.
192.168.1.2 and 192.168.1.3 have 3 common neighbors.
```

Exercise 5: snake game (pointers + typedef, level 3)

In this exercise, you are asked to implement the snake game where the player controls a moving snake and ensures that it does not touch obstacles, the edges of the screen, or their own body. The snake grows every time it encounters food in its path.

Start from the provided file `snake.c` which contains the code to display the game on the screen and interact with the user. You will have to fill in the missing parts: define the data structures and implement the logic of the game.

You can compile the game code either as usual, or if you are on a machine that has the ncurses library (installed for example with `sudo apt-get install libncurses5-dev`), then use the command:

```
gcc -ansi -Wall -pedantic -DUSE_CURSES snake.c -o snake -lncurses
```

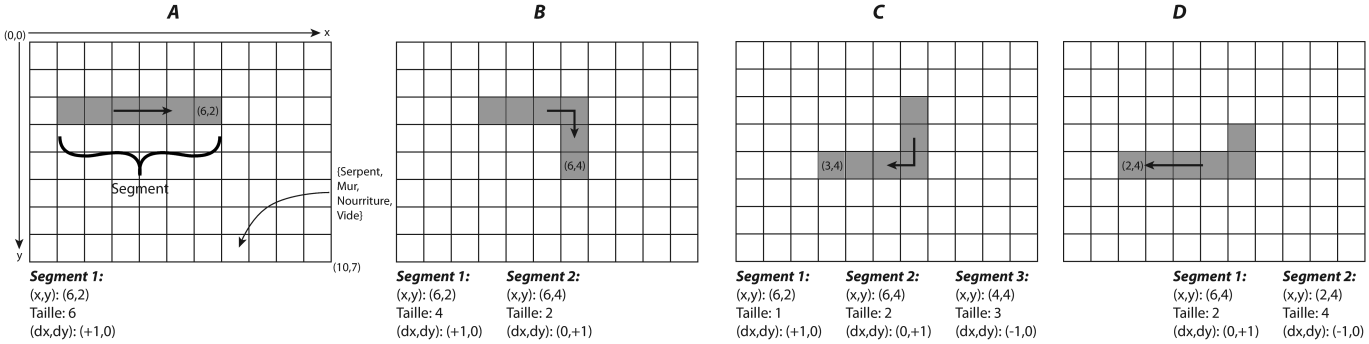
When you have coded everything, you can move the snake with the direction arrows. The game ends if the snake hits a wall or bites its tail.

The snake moves horizontally or vertically in a rectangle represented by a two-dimensional array whose origin corresponds to the upper left corner of the screen. The example in the figure below shows a board of size 11 by 8. Each square on the board can contain part of the snake, a wall, food, or be an empty cell.

A snake is represented by a list of segments. When moving only in a straight line, the snake has only one segment. Each time the snake changes direction, a new segment is created. The longer the snake, the more it can wriggle and the more segments there can be.

Representing the snake by a list of segments optimizes the display. When the snake moves, simply redraw the tail and head; the rest of the body does not move visually.

Let us illustrate with an example by describing steps A, B, C and D in the following figure:



A. The snake moves in a straight line to the right (direction (1,0)). The head of the segment is at coordinates (6,2) and has a length of 6 squares.

B. The snake changed direction (the player pressed the down arrow) and a new segment was created. As the snake moves forward, the head segment increases in size and the tail segment decreases.

C. After the snake has moved two squares forward, the player presses the left arrow. As in the previous step, a new segment is created, bringing their number to three. After the snake has moved forward three spaces, segments 1, 2, and 3 have lengths of 1, 2, and 3, respectively.

D. The snake moves forward by one unit, increasing the size of the head by 1 and reducing the size of the tail by 1. The tail segment now has zero size and is removed from the segment list. The oldest remaining segment becomes the new tail segment.

5.1 - Data types

In the file provided `snake.c`, define the data types to represent (see above) the: direction of movement of a segment, a segment, a snake, the content of 'a square on the board, as well as the game itself. Please carefully follow the names of the structure types, members and functions indicated as some of them are used by the rest of the program already written.

5.1.1. - Direction of movement Define a structure of type `direction_t` having two integers `dx` and `dy`
We will take the convention that these integers are worth -1, 0 or 1, and that only one of them at a time can be non-zero (a snake cannot move diagonally).

5.1.2 - Segment Define a structure of type `segment_t` grouping the following elements:

- the `x` and `y` coordinates of the segment;
- the length of the segment;
- the direction of the segment;
- a pointer to the previous segment.

5.1.3 - Snake A snake is simply a linked list of segments. The first element in the list is the tail, the last is the head. Define a structure of type `snake_t` containing a pointer to the tail and a pointer to the head.

5.1.4 - Game boxes The snake moves on a board that may contain emptiness, walls, food, or a piece of the snake itself. Define an enumerated type `map_cell_t` with the elements `EMPTY`, `WALL`, `FOOD` and `SNAKE`, in that order.

5.1.5 - Definition of the game Define a structure of type `game_t` which groups the following elements:

- a snake;
- the `width` of the table;
- the `height` of the table;
- a *one-dimensional* `map` array of `map_cell_t` elements of any size (i.e. not known at compile time).

5.2 - Functions

In this part, you will implement a series of functions for the game engine. You must write these functions in the second part of the `snake.c` file provided.

For each of these functions, be sure to check the validity of the parameters passed and if necessary return an appropriate error. All functions that return an integer must return a non-zero value on error, and zero on success.

5.2.1 - Snake Debugging Write the function (of prototype:)

```
void snake_info(const snake_t* snake);
```

which iterates through the list of segments and displays the contents of each segment on a separate line. You can set the display format and content freely.

This function is called every time the snake moves to make debugging easier (you can also use it elsewhere if it's useful for you).

5.2.2 - Destruction of snakes Write the function:

```
void snake_erase_tail(snake_t* snake);
```

which deletes the snake's tail segment (i.e. removes it from the list and frees its memory).

Then, write the function:

```
void snake_destroy(snake_t* snake);
```

which erases all the segments of the snake; reuse the `snake_erase_tail` function.

5.2.3 - Snake movement Write the function:

```
int snake_add_segment(snake_t* snake, direction_t direction);
```

which creates a segment, attaches it to the head of the snake and initializes it with the specified direction. A possible algorithm is:

1. allocate memory for the segment;
2. initialize the segment direction;
3. add the segment to the list:
   - if it is the first segment, initialize its size to 1 and initialize the snake's tail;
   - otherwise, initialize its size to 0 and define its coordinates (x,y) as the sum of the direction vector and the coordinates of the previous segment (= the head);
4. Update the head.

Then, write the function:

```
int snake_move(snake_t* snake, direction_t direction);
```

which moves the snake one square in the indicated direction. A possible algorithm is:

1. if the indicated direction is the same as that stored in the snake's head segment, add the direction to the head's (x,y) coordinates; otherwise create a new segment.
2. if there is more than one segment (i.e. the head is different from the tail), increment the length of the head and decrement the size of the tail.
3. if the length of the tail is zero, destroy the corresponding segment.

5.2.4 - Game board update Write the function:

```
map_cell_t* cell(const game_t* game, unsigned int x, unsigned int y);
```

which returns (the address of) the coordinate box (`x`, `y`) in the `map` array of the `game` game.

The one-dimensional representation convention in this array is that the coordinate box (`x`, `y`) is stored at the index "`x` plus `y` times the width".

Then, write the function:

```
int game_update(game_t* game, direction_t direction);
```

which updates the state of the game board. A possible algorithm is:

1. Calculate the current coordinates (x_q,y_q) of the end of the tail: x_q = x - (n-1) d_x, y_q = y - (n- 1) d_y, where x and y are the coordinates of the tail segment, n is its size, and (d_x, d_y) is the direction.
2. Move the snake in the direction passed in parameters.
3. If the snake's head hits a wall (`WALL`) or if the snake runs into it (`SNAKE`), return an error. Otherwise, if the head encounters food (`FOOD`), increment the size of the tail. Otherwise, empty (`EMPTY`) the space occupied by the end of the queue.
4. Update (`SNAKE`) the square occupied by the head.

5.2.5 - Initializing the game Finally, write the function:

```
int game_init_snake(game_t* game, unsigned int orig_x, unsigned int orig_y);
```

which creates a snake with a single segment of length 1 and whose starting coordinates are `orig_x` and `orig_y`, and updates (`SNAKE`) the corresponding box.

You don't need to worry about the rest of the `game_t` structure because it is initialized in the `snake.c` file provided to you.

Exercise 6: Some illustrative examples for L03 (fork, exec, wait)

Finally, we provide you with some examples to illustrate `fork`, `exec` and `wait` (lecture L03); to go through them by yourself, modify them, develop them further. . . :

- fork, exec and wait: forkdemo.c
- output redirection: forkdemo2.c