Pointers in C

Luís Nogueira, Paulo Baltarejo Sousa

{lmn,pbs}@isep.ipp.pt

2024/2025

Notes:

- Each exercise should be solved in a modular fashion. It should be organized in two or more modules and compiled using the rules described in a Makefile
- If you got some logical errors, you should debug the program (using GDB) before asking for help
- Function signatures cannot be altered or else the program will fail the unit tests
- Unless clearly stated otherwise, the needed data structures for each exercise must be declared locally in the main function and their addresses passed as parameters to the developed auxiliary functions
- The code should be commented and indented

1. Implement a program that declares the following variables:

   ```
   int x = 5;
   int* ptr_x = &x;
   float y = *ptr_x + 3;
   ```

   - The program should print:
     - The value of x and y
     - The addresses (in Hexadecimal) of x, y, and ptr_x
     - The value of ptr_x
     - The value pointed by ptr_x
   - Analyze the information that appears on the screen and (**write answers as code comments**).

- Add the following code:

  ```
  int vec[] = {10, 20, 30, 40};
  int* ptr_vec = vec;
  int z = *ptr_vec;
  int h = *(ptr_vec + 3);
  ```

- The program should print:
  - The value of `z` and `h`
  - The addresses (in Hexadecimal) of `vec` and `ptr_vec`
  - The values of `ptr_vec` and `vec`
  - The value pointed by `ptr_vec`
- Analyze the information that appears on the screen.
- Explain the relationship between the address of `vec` and the value of `ptr_vec`.
- Add the following code:

  ```
  int i;
  for(i = 0; i < 4; i++){
     printf("1: %p,%d\t", &vec[i],vec[i]);
  }
  printf("\n");
  for(ptr_vec = vec; ptr_vec < vec + 4; ptr_vec++){
     printf("2: %p,%d\t", ptr_vec,*ptr_vec);
  }
  printf("\n");
  for(ptr_vec = vec + 3; ptr_vec >= vec; ptr_vec--){
     printf("3: %p,%d\t", ptr_vec,*ptr_vec);
  }
  ```

- Relate the code to the information that appears on the screen.
- What does `ptr_vec++` or `ptr_vec--`?
- Add the following code:

  ```
  printf("\n");
  ptr_vec = vec;
  printf("4: %p,%d\n", ptr_vec,*ptr_vec);
  a = *ptr_vec++;
  printf("5: %p,%d,%d\n", ptr_vec,*ptr_vec, a);
  ptr_vec = vec;
  a = (*ptr_vec)++;
  printf("6: %p,%d,%d\n", ptr_vec,*ptr_vec, a);
  ptr_vec = vec;
  a = *++ptr_vec;
  printf("7: %p,%d,%d\n", ptr_vec,*ptr_vec, a);
  ptr_vec = vec;
  a = ++*ptr_vec;
  ```

```
printf("8: %p,%d,%d\n", ptr_vec,*ptr_vec, a);

printf("\n");
for(ptr_vec = vec; ptr_vec < vec + 4; ptr_vec++){
   printf("9: %p,%d\t", ptr_vec,*ptr_vec);
}
```

- Relate the code to the information that appears on the screen.
- Add the following code:

```
printf("\n");
unsigned int d = 0xAABBCCDD;
printf("10: %p,%x\t", &d,d);
printf("\n");
unsigned char* ptr_d = (unsigned char*)&d;
unsigned char* p;
for(p = ptr_d; p < ptr_d + sizeof(unsigned int); p++){
   printf("11: %p,%x\t", p,(unsigned char)*p);
}
```

- Relate the code to the information that appears on the screen.

2. Write a program that declares and initializes (to any value you like) a double, an int, and a char. Next, declare and initialize a pointer to each of the three variables. Your program should then print the address of, and value stored in, and the memory size (in bytes) of each of the six variables.

   Use the "0x%x" formatting specifier to print addresses in hexadecimal. You should see addresses that look something like this: "0xbfe55918". The initial characters "0x" tell you that hexadecimal notation is being used; the remainder of the digits give the address itself.

3. The function `swap_nums` seems to work, but not `swap_pointers`. Fix it.

```
#include <stdio.h>

void swap_nums(int *x, int *y){
  int tmp;
  tmp = *x;
  *x = *y;
  *y = tmp;
}

void swap_pointers(char *x, char *y){
  char *tmp;
  tmp = x;
  x = y;
  y = tmp;
}

int main(){
  int a=3,b=4;
  char *s1,*s2;

  swap_nums(&a,&b);
  printf("a is %d\n", a);
  printf("b is %d\n", b);
  s1 = "I should print second";
  s2 = "I should print first";
  swap_pointers(s1,s2);
  printf("s1 is %s\n", s1);
  printf("s2 is %s\n", s2);
  return 0;
}
```

4. Implement the function `void capitalize(char *str)` that receives the address of a string and replaces all its lowercase characters by their uppercase counterpart.
   - Your function should not use any other function available in the C standard library.
5. Implement the function `void copy_vec(int *vec1, int n, int *vec2)` that copies `n` integers from `vec1` into `vec2`.
   - Try to solve the exercise with pointer arithmetic.
6. Implement the function `int sum_even( int *vec, int n)` that receives an address of an array, `vec`, and the number of elements in that array, `n`, and returns the sum of all of its even elements.
   - Try to solve the exercise with pointer arithmetic.
7. Implement the function `void capitalize2(char *str)` that receives the address of a string, `str`, and replaces all its lowercase characters by their uppercase counterpart.
   - Your function should not use any other function available in the C standard library.
   - Try to solve the exercise with pointer arithmetic.
8. Implement the function `int sum_integer_bytes(unsigned int *p)` that receives the address of an integer, (`p`), and returns the sum of its bytes.

   Example:

   ```
   unsigned int d = 0xAABBCCDD;
   int r = sum_integer_bytes(&d); // r = 0xDD+0xCC+0xBB+0xAA = 782
   ```

- Try to solve the exercise with pointer arithmetic.

9. Implement the function `void get_array_statistics(int* vec, int n, int* min, int* max, float* avg)` that receives the address of an array of integer vales, `vec`, and the number of elements in that array,`n`. The function should compute the minimum, maximum, and average of the elements in that array. The computed values should be written in the addresses given by the remaining function parameters, `min`, `max`, and `avg`, respectively.
   - Try to solve the exercise with pointer arithmetic.

10. Using pointer arithmetic, implement the function `char* where_is(char *str, char c)` that receives the address of a string, (`str`), and a character (`c`). The function should iterate through `str` looking for the character `c`. When it finds the character `c`, it should returns the address of the position of `c` into `str`. If it does not find the character `c`, it should return `NULL`.

11. Using pointer arithmetic, implement the function `int sum_odd(int *p)` that receives the address of an array of integer values, (`p`), as its single parameter. The first element of the array indicates how many elements are stored on it. The function should return the sum of the odd elements of that array, excluding its first element.

12. Using pointer arithmetic, implement the function `void array_sort(int *vec, int n)` that, given the address of an array of integers, `vec`, and the array length, `n`, orders the array in ascending order.

13. Using pointer arithmetic, implement a function `int sort_without_reps(short *src, int n, short *dest)` that receives the address of an array `src` with `n` integer elements and the address of an empty second array of the same size. The function should fill the second array with the elements of the first in ascending order, eliminating all repeated values. The function must return the number of items placed in the second array. The content of the second array should be printed in the main function.

14. Using pointer arithmetic, implement the function `void count_words(char *str)` that receives the address of a string, `str`, and returns the number of words in that string. Consider that each word is ends by at least one space or the end of the string.

    Example:

    ```
    string str [] = "     the     numBERr     must be  saved";
    int x = count_words(str);
    printf("%d\n", x); // 5
    ```

15. Using pointer arithmetic, implement the function `void trim_string(char *str)` that receives the address of a string, `str`, and removes all spaces at the beginning,

at the end, and also multiple spaces between words (leaving only one).

Example:

```
string str [] = "     the     numBERr     must be  saved      ";
trim_string (str);
printf("%s\n", str); // "the numBERr must be saved"
```

16. Using pointer arithmetic, implement the function `void format_word(char *str)` that receives the address of the first character of a word whitin a string, `word`). The function should uppercase that first letter of the word and lowercase the remaining characters. Consider that a word ends with a space or with the end of the string.

17. Implement a function `void frequencies(unsigned float *grades, int n, int *freq)` that receives the address of and array `grades` with the students' exam grades at ARQCP (a float value between 0.0 and 20.0), the number of elements in that array, `n`, and the address of a second array, `freq` to be filled with the absolute frequency of the integer part of the grades stored in the array grades. Use pointer arithmetic to solve this exercise.

    Example:

    - the array `grades` with content 8.23, 12.25, 16.45, 12.45, 10.05, 6.45, 14.45, 0.0, 12.67, 16.23, 18.75 should produce a `freq` array with 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 0

18. Implement a program to determine how many sets of three consecutive elements exist in an array of integer values that satisfy the condition `vi < vi+1 < vi+2`. Create three auxiliary functions:

    - a function `void populate(unsigned char *vec, int num, int limit)` to populate an array `vec` of 50 integers, `num`, with random values between 0 and 20, `limit`;
    - a function `int check(int x, int y, int z)` that verifies if a set of three integers satisfies (1) or not (0) the given condition;
    - a function `void inc_nsets()` to increment the number of determined sets. The number of sets should be stored in a global variable defined in the same module of this function.

    Each of these functions should be placed in a separate module. Use pointer arithmetic to solve this exercise.

19. Implement a function `void swap(short* vec1, short* vec2, int size)` that receives the address of two arrays of the same size and swaps the contents of both arrays (i.e contents of `vec1` will be copied to `vec2` and vice versa). The new content of each array must be printed in the main function.

20. Implement a function `void compress(int* vec_ints, int n, long* vec_longs)` to "compress" the values of an array of ints into an array of longs. The function must, in a sequential manner, compress two consecutive ints into a single long that must be stored in the array `vec_longs`. Assume that the number of elements in the array `vec_ints` is even. The content of the `vec_longs` array must be printed in the main function.

21. Consider the following code. Two arrays are defined in the `main` function: one is an array of integers (called `data`) and the other is an array of integer pointers (called `data_ptrs`).

```
#include <stdio.h>
#include <stdlib.h>
#define DATA_LENGTH 5

void init_data(int data, int n);
void print_data(int data, int n);
void init_data_ptrs(int data, int n, int ptrs);
void print_data_ptrs(int* ptrs, int n);
int * find_max_data_ptrs(int** ptrs, int n);
void sort_data_ptrs(int** ptrs, int n);

int main(void){
  int data[DATA_LENGTH];
  int *data_ptrs[DATA_LENGTH];
  int *res;

  init_data(data, DATA_LENGTH);
  print_data(data, DATA_LENGTH);
  init_data_ptrs(data, DATA_LENGTH, data_ptrs);
  print_data_ptrs(data_ptrs, DATA_LENGTH);
  res = find_max_data_ptrs(data_ptrs, DATA_LENGTH);
  printf("0x%x\n", *res);
  sort_data_ptrs(data_ptrs, DATA_LENGTH);
  print_data(data, DATA_LENGTH);
  return 0;
}
```

Using this as a reference, implement the following functions, ensuring to utilize pointer arithmetic for loops and other operations where applicable:

a) `void init_data(int *data, int n)`: This function takes the address of an integer array (`data`) and the number of elements (`n`). It randomly initializes all elements of the array.

b) `void print_data(int *data, int n)`: This function receives the address of an integer array (`data`) and the number of elements (`n`), then prints its contents in hexadecimal format.

c) `void init_data_ptrs(int *data, int n, int** ptrs)`: This function takes the address of an integer array (`data`), the number of elements (`n`), and the address of an integer pointer array (`data_ptrs`). It initializes each element of `data_ptrs` with the address of the corresponding element in the `data` array.

d) `void print_data_ptrs(int** ptrs, int n)`: This function takes the address of an integer pointer array (`data_ptrs`) and the number of elements (`n`). It prints each element of `data_ptrs`, its value (which is an address), and the value pointed to by that address.

e) `int * find_max_data_ptrs(int** ptrs, int n)`: This function takes the address of an integer pointer array (`data_ptrs`) and the number of elements (`n`). It returns the address of the highest value pointed to by the array.

f) `void sort_data_ptrs(int** ptrs, int n)`: This function takes the address of an integer pointer array (`data_ptrs`) and the number of elements (`n`). It sorts the array in descending order.

22. Consider the following code. Two integer arrays (`data1` and `data2`) are defined in the `main` function, along with two integer pointers (`bigger` and `smaller`).

```
#include <stdio.h>
void swap_pointers(int **b, int **s, int n);

int main(void){
  int data1[5] = {1, 2, 3, 4, 5};
  int data2[5] = {10, 20, 30, 40, 50};
  int *bigger = data1;
  int *smaller = data2;

  swap_pointers(&bigger, &smaller, 5);
  return 0;
}
```

Implement the function `void swap_pointers(int **b, int **s, int n)`. This function receives the addresses of two integer pointers (`b` and `s`), each pointing to an array, and the number of elements in each array (`n`). The function computes the sum of the elements in each array and assigns to `b` the address of the array with the larger sum, and to `s` the address of the array with the smaller sum.

23. You are tasked with implementing a series of functions to manipulate a static two-dimensional array (a matrix) using pointer arithmetic. You will implement functions that perform operations such as transposing, rotating, and finding the sum of elements in the matrix.

The matrix is a 2D array of integers of size $N \times M$, where both N and M are fixed constants. You are required to manipulate the matrix directly using pointer arithmetic, rather than using array indexing.

Implement the following functions using pointer arithmetic to access and manipulate the matrix elements:

a) `void transpose_matrix(int *matrix, int N, int M)`:
   - Transpose the matrix in-place if $N = M$, or into another matrix if $N \neq M$.
   - The function receives a pointer to the first element of the matrix and its dimensions.

b) `void rotate_matrix_clockwise(int *matrix, int N)`:
   - Rotate the matrix 90° clockwise, assuming it's a square matrix ($N = M$).
   - You are required to rotate the matrix in-place.

c) `int sum_matrix(int *matrix, int N, int M)`:
   - Return the sum of all the elements in the matrix.

d) `void print_matrix(int *matrix, int N, int M)`:
   - Print the matrix in a formatted way. Each element should be accessed using pointer arithmetic, and the output should resemble the matrix's layout.

The `main` function should define a 3×3 matrix, initialize it with values, and test all of the above functions. Example usage:

```
#include <stdio.h>
#define N 3
#define M 3

int main(void) {
    int matrix[N][M] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    printf("Original Matrix:\n");
    print_matrix((int*)matrix, N, M);

    transpose_matrix((int*)matrix, N, M);
    printf("\nTransposed Matrix:\n");
    print_matrix((int*)matrix, N, M);

    rotate_matrix_clockwise((int*)matrix, N);
    printf("\nRotated Matrix (90 degrees clockwise):\n");
    print_matrix((int*)matrix, N, M);

    int sum = sum_matrix((int*)matrix, N, M);
    printf("\nSum of all elements: %d\n", sum);

    return 0;
}
```

Implementation details:

a) `transpose_matrix`:
   - Transpose the matrix by swapping elements. You will need to access each element as `*(matrix + i * M + j)`.

b) `rotate_matrix_clockwise`:
   - Perform the rotation by swapping elements between the first and last rows, etc., and handle elements in a circular manner using pointer arithmetic.

c) `sum_matrix`:
   - Traverse the matrix using pointer arithmetic to calculate the sum.

d) `print_matrix`:
   - Loop over the matrix elements using pointer arithmetic to print each element in its correct position.