Einführung in die Programmierung für Physiker

SoSe 2024 – Marc Wagner

Michael Eichberg: eichberg@itp.uni-frankfurt.de

Exercise sheet 7

To be handed in on 07.06.2024.

Exercise 1 [*Arrays decaying to pointers II*] (3 + 3 + 3 + 1 = 10 pts.)

Now that you understood that arrays cannot naively be passed by value to a function and that only a copy of the pointer to its address is passed to the function, you should be ready to generalize this idea to multi-dimensional arrays. We will just focus on two-dimensional arrays, again with fixed sizes and type `int`.

(i) Definine a two-dimensional array `int M[2][3]` in the `main` function of your program and initialize it with integers between 1 and 6. Such an array can be seen as a matrix with 2 rows and 3 columns, but you could think of `M` as an array with two components, which in turn are 3-component one-dimensional arrays. Define then a pointer `M_ptr` pointing to `M`. This has to be a pointer to a one dimensional array with three integer components. Find two equivalent ways to do so. Print the matrix to the screen both using the matrix itself and using the pointer. Call the `sizeof` operator on `M`, `M[0]`, `M[0][0]`, `M_ptr` and `*M_ptr`. Can you explain the output?

(ii) Consider the following function.

```
void changeMatrix(int matrix[][3], int rows, int columns){
    printf("In changeMatrix:\n");
    printf("       sizeof(matrix) = %lu\n", sizeof matrix);
    printf("    sizeof(matrix[0]) = %lu\n", sizeof matrix[0]);
    printf(" sizeof(matrix[0][0]) = %lu\n", sizeof matrix[0][0]);
    if(rows>1)
        matrix[1][0]=111;
    matrix=NULL;
}
```

Call `changeMatrix` from `main`, using `M` and print `M` before and after the function call. Can you explain the discrepancy between the various `sizeof` results in `main` and `changeMatrix`? What did the last line in the function body do? Did `M` change in `main`? Can you call `changeMatrix` passing `M_ptr` as first argument?

(iii) Consider changing the signature of the function to any of the following.

(a) `void changeMatrix(int (*matrix)[3], int rows, int columns)`

(b) `void changeMatrix(int  *matrix[3], int rows, int columns)`

(c) `void changeMatrix(int  matrix[][], int rows, int columns)`

(d) `void changeMatrix(int matrix[1][3], int rows, int columns)`

(e) `void changeMatrix(int  matrix[][2], int rows, int columns)`

Which is a valid alternative for the function signature in task (ii)? Those that do not work, why are they not working?

(iv) Change now the signature to `void changeMatrix(int** matrix, int rows, int columns)`. Why is it wrong to call this function either with `M` or `&M_ptr` as first argument? Your code might compile, but the compiler will print warning messages, which should not be ignored.

Exercise 2 [*Dynamic memory allocation*] (2 + 2 + 4 + 2 = 10 pts.)

In the lecture you learned how to reserve memory for your program at run time. In one of the next lectures, you will discuss in detail a possible way to allocate memory to store a matrix later on. The corresponding code is conceptually equivalent to the following.

```
void allocateMatrix(int*** matrix, int rows, int columns){
    *matrix = (int**)malloc(rows * sizeof(int*));
    if( *matrix == NULL){
        printf("There has been an error allocating memory!\n");
        exit(1);
    }
    for(int i = 0; i < rows; ++i){
        (*matrix)[i] = (int*)malloc(columns * sizeof(int));
        if((*matrix)[i] == NULL){
            printf("There has been an error allocating memory!\n");
            exit(1);
        }
    }
}
```

(i) Starting from the above code, define a matrix of type `int**` in the `main` function, allocate memory using the above function and initialize it to some values. How is the function `allocateMatrix` reserving memory? Is the allocated chunk of memory contiguous?

(ii) Devise a way to understand how far away in memory are two variables using the difference of memory adresses. The `sizeof` operator might be useful in this respect. Use your method to check your answer to task (i).

(iii) Implement a function `allocMatrixContiguous` which allocates memory in a contiguous way, i.e. it reserves a chunk of `rows*columns*sizeof(int)` bytes of memory. What should you do to still be able to access the matrix elements using the access operator, i.e. via the `matrix[][]` notation?

(iv) Check that your allocation is indeed contiguous and that `size(int)` is indeed the distance between contiguous elements in the matrix. In this regard, e.g., `matrix[0][columns-1]` and `matrix[1][0]` should be contiguous. Why might this approach be advantageous?