Einführung in die Programmierung für Physiker

SoSe 2024 – Marc Wagner

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

Exercise sheet 6

To be handed in on 31.05.2024.

Exercise 1 [*Const-correctness*] (4 pts.)

The use of the `const` keyword in C is, strictly speaking, not mandatory, but it helps us to defend ourselves... from ourselves! Actually, there are typical use cases, one of which we will analyse in a moment. Before, let us clarify the meaning of the keyword and how to use it, especially together with pointers. What `const` means should be clear from the keyword itself. *Something* is marked as constant and it cannot be changed (and this allows the compiler to give you an error if you try to violate this rule you set yourself). However, it is very important to understand *what* exactly is marked as constant. Consider the following code.

```c
int main(void){
    double var = 5, value = 10;
    double* ptr_var = &var;
    const double c_var = 6;
    const double* ptr_c_var = &c_var;
    double* const c_ptr_var = &var;
    const double* const c_ptr_c_var = &c_var;
    // Modyfing attempts:
    /*(0)*/  ptr_c_var = ptr_var;   /*(4)*/  *c_ptr_var = value;
    /*(1)*/  c_var = value;         /*(5)*/  c_ptr_var = ptr_var;
    /*(2)*/  var = c_var;           /*(6)*/  *c_ptr_c_var = value;
    /*(3)*/  *ptr_c_var = value;    /*(7)*/  c_ptr_c_var = ptr_var;
}
```

Which of the assignments are allowed? Can you predict which error the compiler will give you? How do you understand the types of the variables `var`, `ptr_var`, `c_var`, `ptr_c_var`, `c_ptr_var`, `c_ptr_c_var`?

Exercise 2 [*Linear algebra*] (4 pts.)

In the lecture, you saw examples for the use of arrays in linear algebra operations. The library is still missing important features, our goal is to implement some of them. Copy the code presented in the lecture. Considering $N = 4$ dimensions, add new functions to perform the operations $C = A + B$, $C = c \cdot A$, $w = u + v$ and $w = c \cdot v$ for $c \in \mathbb{R}$, $u, v, w \in \mathbb{R}^N$ and $A, B, C \in \mathbb{R}^{N\times N}$. Use the `const` keyword wherever you can.
Test your code by computing the expression

$$\left(\begin{pmatrix}1&0&0&0\\0&0&0&0\\0&-1&0&0\\0&0&0&1\end{pmatrix} + 2\cdot\begin{pmatrix}0&0&0&0\\0&0&-0.5&0\\0&1&0&0\\0&0&0&0\end{pmatrix}\right)\left(\begin{pmatrix}0\\2\\0\\3\end{pmatrix} + \frac{1}{2}\cdot\begin{pmatrix}2\\0\\-2\\0\end{pmatrix}\right) = \begin{pmatrix}1\\1\\2\\3\end{pmatrix}.$$

Exercise 3 [*Arrays decaying to pointers I*] (3 + 3 + 1 + 1 = 8 pts.)

In this exercise we explore in detail what exactly happens when an array is passed to a function.

(i) Define an array `int v[5]` in the `main` function of your program and initialize it. Define then a pointer `v_ptr` and find two equivalent ways to let it point to `v`. Print the array, one time using the pointer. Call the `sizeof` operator on `v` to check the size of the array. What does the result of `sizeof` mean?

Remark: at this point you might think that the name of an array is implicitly like a pointer to the first element of the array. This statement is slightly inaccurate and there are subtle differences which we will now explore. A more precise statement would be that *array names can decay to pointers.*

(ii) Consider the following function.

```
void changeArray(int* array, int size){
    printf("In changeArray, sizeof(array) = %lu\n", sizeof array);
    if(size>2)
        array[2]=111;
    array=NULL;
}
```

Call `changeArray` from `main`, using `v_ptr` and print `v` to the output before and after the function call. Can you explain the discrepancy between the call of `sizeof` in `main` and in `changeArray`? Is `v_ptr` still a valid pointer after the function call? Did `v[2]` change in the `main`? Can you call `changeArray` passing `v` as first argument?

(iii) Change the signature of the function to `void changeArray(int array[], int size)` and repeat task (ii) passing `v` to it. Did anything change? Can you call this function passing `v_ptr` as first argument?

Exercise 4 [*The Sieve of Sundaram*] (4 pts.)

The sieve of Sundaram is an algorithm to find all the prime numbers up to a specified integer. It was discovered by the Indian mathematician S. P. Sundaram in 1934.
For fixed $N \in \mathbb{N}$ and given a list of the integers between 1 and N, remove all the numbers of the form

$$i + j + 2\,i\,j \qquad \text{where} \qquad i \in \mathbb{N} \quad , \quad j \in \mathbb{N} \quad , \quad 1 \le i \le j \quad \text{and} \quad i + j + 2\,i\,j \le N \quad .$$

The remaining numbers, each doubled and incremented by one, give all the prime numbers smaller than $M \equiv 2N + 2$ except 2.
Implement the Sieve of Sundaram with $N = 500$ and print all prime numbers smaller than M, along with the number of prime numbers up to M.
Hint: Even if it sounds natural, do *not* use an array of `int` variables. Instead, think of the array indices as your list of numbers for the sieve and keep track of wether the index corresponds to a prime number by setting the value to `0` or `1`.
Time to Test! For $N = 500$, your code should tell that there are 168 prime numbers up to $M = 1002$.