Einführung in die Programmierung für Physiker

WS 2017/2018 – Marc Wagner

Francesca Cuteri: cuteri@th.physik.uni-frankfurt.de
Alessandro Sciarra: sciarra@th.physik.uni-frankfurt.de

Exercise sheet 7

To be corrected in tutorials in the week from 04/12/2017 to 08/12/2017

Exercise 1 [*Scope rules*[1]]

The *scope of an identifier* is the portion of the program in which the identifier can be referenced. *Identifiers* are the names you supply for variables, types, functions, and labels in your program. In this exercise we will try to understand the scope rules for two different identifier scopes i.e. *file scope* and *block scope*. The basic principle behind scope rules is the *principle of least privilege* according to which every part of a program (function, file, block) must be able to access only the information (e.g. variables) that is necessary to accomplish its specific task, but nothing more. Scope rules establish that:

(A) An identifier declared outside any function has *file scope*, hence it is accessible, from the point at which it is declared, in all functions until the end of the file. Such rule applies e.g. to *global variables* and *function definitions*.

(B) Identifiers defined inside a block `{...}` (e.g. the body of functions or control structures) have *block scope* and such scope ends at the terminating right brace `}` of the block. Such rule applies to variables that are defined inside the block and hence called *local variables*[2].

In order to understand the above scoping issues, you can write a simple program where:

(i) A global variable `x` of type `int` is defined and initialized to 1.

(ii) The `int main(void)` function of your program is written to contain:

  (a) The definition of a local variable `x` of type `int`, initialized to 5 and whose value is printed to the screen.

  (b) A nested scope `{...}` that, in turn, contains the definition `int x = 7;` and a `printf` statement to have the value printed in standard output. Copy your `printf` statement also before declaring the variable `x`. Which `x` is printed?

  (c) Another `printf` statement to reprint the local `x` in the main scope of your `main` function.

  (d) Multiple calls in some order of your choice to the functions:

    (1) `void useLocal(void)` in which you will have the definition `int x = 25;` and two `printf` statements, one before and one after having incremented by 1 the local `x`.

    (2) `void useStaticLocal(void)` in which you will have the `static int x = 50;` definition and two `printf` statements, one before and one after having incremented by 2 the local `x`.

    (3) `void useGlobal(void)` in which you will just have two `printf` statements, one before and one after having multiplied by 10 the global `x`.

  (e) In each function, add a `printf` statement at the very beginning to print the value of `x`. Which `x` is printed?

Nothing left to do, than looking at the output of your program to check that *scope rules* are respected, so you will need to keep them in mind in your programmer's life!

[1] The topic addressed in this exercise will be discussed in the very near future also in the lectures. However, we decided to tackle it here in advance, because it will already turn out very helpful in avoiding mistakes in your programs.

[2] Local variables declared as `static` also have a block scope even though they remain in memory while the program is running and preserve their value even after they are out of scope.

Exercise 2 [*Pointers and variables*]

Pointers and variables are logically different, but sometimes some confusion arises when dealing with the operators `&` and `*`. Let us try to play a bit with them in order to explore some typical use cases.

(i) Start writing a very basic program. Declare an integer variable `N`, set it to 17 and declare a pointer `Np` to an integer variable making it pointing to the previously defined integer variable.

(ii) Print to the screen the content of `N`, its address, the content of `Np`, its address, and the integer it points to. Which conversion specifier of `printf` you have to use? Is it clear to you where you have to use `N`, `&N`, `Np`, `&Np`, `*Np` and what they mean?

(iii) Set the content of `N` to 101 and print both `N` and `*Np`.

(iv) Set again the content of `N` to 17, this time using `Np`, and print both `N` and `*Np`.

Now it should be clear to you what a pointer is and how to use it. However, you could still wonder why pointers should be used and how to decide in favour of a pointer instead of a simple variable. There are many reasons, some more involved than others, but probably the simplest one is connected to delegate responsibility to functions. Before looking more in detail what this means, let us play another bit with our previous code.

(v) Add the following couple of functions to your code.

```
void SetToTen(int n){ n = 10; }
void SetToTenPtr(int *n){ *n = 10; }
```

(vi) In your main, call each function once with `N` and once with `Np`, printing `N` and `*Np` both before and after each call. Restore `N = 17` before calling the second function. Are you surprised of the result? What is happening?

As you saw yourself, whenever we want a function to change the value of a variable belonging to another scope, we need to use pointers. People often refer to this as *delegating* a task to function. Functions and pointers can bring some danger with them, though. Consider the following function.

```
#include <stdio.h>
int* GetTen(void){ int n = 10; return &n; }

int main(void){
    int* Np = GetTen();
    printf("Ten from function is %d\n", *Np);
}
```

There is a subtle bug in it. Can you find it? Try to compile and run your code (your compiler could warn you about something going on in the code).

To conclude, write another short program which uses *only* one function to calculate the perimeter and the area of a square, given its side. What do you learn from this small task? Can you `return` multiple data types in `C`?

Exercise 3 [*Const-correctness*]

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.

```
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)*/  var = value;          /*(5)*/  ptr_c_var = ptr_var;
    /*(1)*/  *ptr_var = value;     /*(6)*/  *c_ptr_var = value;
    /*(2)*/  c_var = value;        /*(7)*/  c_ptr_var = ptr_var;
    /*(3)*/  var = c_var;          /*(8)*/  *c_ptr_c_var = value;
    /*(4)*/  *ptr_c_var = value;   /*(9)*/  c_ptr_c_var = ptr_var;
}
```

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

Consider now the following signatures of a function which takes a number and returns its square root.

```
double sqrt_I(double  x);          double sqrt_IV(const double* x);
double sqrt_II(const double x);    double sqrt_V(double* const x);
double sqrt_III(double* x);        double sqrt_VI(const double* const x);
```

(i) Be sure to understand the signatures. Then implement them in a code and call each of them from your `main`. Do they all work?

(ii) Are there differences between the six versions? What happens exactly underneath when each of them is called? What is copied from the `main` to the scope of the function?

(iii) A `double` variable is usually 8 bytes large. Imagine for a moment of having it 8 *Mega*bytes large. Which version of the square root function would you choose? Why?