Lecture 7: Functions

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/07-functions.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

In this lecture we discuss functions.

Functions

Functions have a name, zero or more parameters, each with a type and a name, and a return type. In this example we have a function named `add_two_integers`:

```c
int add_two_integers(int a, int b) {
    int result = a + b;
    return result;
}

int main() {
    int x = 1, y = 2;
    int sum = add_two_integers(x, y);
    printf("result: %d", sum);

    if(add_two_integers(x, y))
        printf(" (non zero)\n");
    else
        printf(" (zero)\n");
    return 0;
}
```

It takes two integers as parameters named `a` and `b`. It returns an integer which is the sum of the two parameters. Like variables, functions must be declared before being called. We declare the function by first writing the return value type, then the function name, followed by the parameters list between parentheses, each with its type. We can call a function as presented in the example, passing the parameter values between parentheses. The call evaluates to the function return value, so we can affect a variable with it or use as a condition. If a function does not need to return anything, `void` should be set for return type.

Call by Copy

An important thing to note is that in C, function parameters are passed by copy and not by reference as it is the case in other languages such as Python. What this means is that each function call gets its own local copy of the parameters' values and updating will not modify the calling context. In the example we have `x` set to `10`, then passed as a parameter of a function that sets this parameter to `12`:

```c
void my_function(int parameter) {
    parameter = 12; // does not update x in main
}
int main() {
    int x = 10;
    my_function(x);
    printf("x is %d\n", x); // prints 10
    return 0;
}
```

If we print the value of x after the function call, it is still 12 because the function, when called, got his own copy of the value of x. We will see in a later lecture how to have a function update a variable from the calling context: this is achieved through a mechanism named pointers.

Forward Declarations

The function **signature** or **prototype** suffices for the declaration. As shown on the example it contains the function return type, name, parameters, and simply ends with a semicolon:

```c
/* Forward declaration, also called function _prototype_ */
int add(int a, int b);
int main(int argc, char **argv) {
    int a = 1;
    int b = 2;
    /* Here we need the function to be at least declared -- not necessarily defined */
    printf("%d + %d = %d\n", a, b, add(a, b));
    return 0;
}
/* The actual function definition */
int add(int a, int b) {
    return a + b;
}
```

The body can be declared further in the file. It can be located below statements in which the function is called. This is called a **forward declaration**. It gives the programmer more freedom for organising his/her code.

Global vs. Local Variables

Until now we only saw **local** variables visible only from within the context they are declared in. It can be a function, a loop body, etc. On the contrary **global** variables are declared outside functions. They can be read or written from everywhere in the sources. For example here `global_var` is set to `100` and printed in `main`, and it is also incremented in the `add_to_global_var` function:

```c
int global_var;
void add_to_global_var(int value) {
    global_var = global_var + value;
}
int main() {
    global_var = 100;
    add_to_global_var(50);
    printf("global_var is %d\n", global_var);
    return 0;
}
```

There are many issues with global variables. An important one is that because they can be read or written from anywhere, they make it harder to understand and reason about the program. They should be used only when needed.

Variables Scope and Lifetime

Contrary to globals, local variables are visible only within the enclosing block of code, delimited with braces. Consider this example:

```c
int x = 12;
if(x) {
    int y = 14;
    printf("inner block, x: %d\n", x);
    printf("inner block, y: %d\n", y);
}
printf("outer block, x: %d\n", x);  // working: x is in scope
printf("outer block, y: %d\n", y);  // error: y only visible in the if body

for(int i=0; i<10; i++) {
    printf("In loop body, i is %d\n", i); // working, i is in scope
}
printf("Out of loop body, i is %d\n", i); // error, i only visible in loop body

int j;
for(j=0; j<10; j++) {
    printf("In loop body, j is %d\n", j); // working
}
printf("Out of loop body, j is %d\n", j); // working, j in scope
```

While `x` is visible from anywhere in main, `y` cannot be printed there because it's visible only within the `if` body. The `i` iterator here is visible only in the for loop body. For `j`, because it is declared in the scope of `main`, it is still visible after the loop finishes. Generally, it is a good practice to try to declare local variable at the beginning of the block.