Operating Systems 2016/17
Tutorial-Assignment 0

Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

You will find introductory slides to the C programming language in the lecture's ILIAS course along the regular course materials.

Question 0.1: C Basics

a. What basic data types are available in the C programming language?

Solution:

- ***char*** *one byte, usually for characters (ASCII)*
  Example: `char c = 5; char c = 'a';`
- ***int*** *usually 4 bytes, holds 32 bit integers*
  Example: `int i = 5; int i = 0xf; int i = 'a';`
- ***long long*** *usually 8 bytes, holds 64 bits integers*
  Example: `long long ll = 0xffffffffff;`
- ***float*** *4 bytes, floating point number*
  Example: `float f = 5; float f = 5.5;`
- ***double*** *8 bytes, double precision floating point number*
  Example: `double d = 5.98798;`

*Types are signed by default. Use the keyword **unsigned** if you need an unsigned type:* `unsigned int a;`

The data-type sizes/precisions may vary on different platforms. inttypes.h *defines types with fixed sizes.*

- ***int8_t*** *signed integer with 1 byte width*
- ***uint64_t*** *unsigned integer with 8 bytes width*

*Note that C does not define a native **boolean** type.*

b. How can you find out which size a type has?

Solution:
*Value ranges and number of bits per data type vary with architecture. The **sizeof** operator returns the length of a data type or variable in bytes. It is resolved at compile-time, not at runtime!*

```
int s = sizeof(double); // s = 8
int s2 = sizeof(s); // s2 = sizeof(int) = 4
```

c. What is the difference between local and global variables?

Solution:

```
int m; // global variable
       // (outside a function)

void myroutine(int j) {
    int i=5; // local variable
    m = i+j;  // m is accessible
}
```

- ***global variables***: *Global variables live while the program runs. They are placed in the data segment, a pre-defined place in the memory of the executable.*
- ***local variables***: *Local variables are accessible and valid only in the scope of the block / function in which their are declared. Local variables are places on the stack or in CPU registers.*

Question 0.2: Hello World

For programming assignments, we provide templates that you have to extend with functionality according to the respective question. You can download the templates for programming assignments from ILIAS.

a. Acquire the template *p1* for this programming question from ILIAS and unpack it. What are header files typically used for? Look how this is done in the programming assignment template.

Solution:
Functions are typically defined once and called from several other locations, potentially in other source files. A function should always be declared, before it is used (called or referenced otherwise) – if you omit the declaration, the compiler cannot perform checks such as matching the function's signature against a call to that function. These checks help to avoid a vast amount of silly bugs by exposing them automatically at compile-time.

Adding the necessary function declarations to each source file manually would cause much effort and the result would be terrible to maintain. A much better way is to use header files for this purpose: They serve to separate function declarations (in `.h` files) from function definitions (in `.c` files). That way, you need to write down (and change) the declarations only once (in one header file) and refer to that file (with `#include` statements) wherever you need the declarations.

In the template for this question, the header file `greet.h` is used to declare the function `greet`. The source file `greet.c` contains the definition of this function. The main source file `main.c` #includes the header file `greet.h`, so that the function is declared and the compiler can perform checks on the function call.

You might notice the `#ifndef`, `#define`, and `#endif` preprocessor statements framing the header file. This structure ensures that the declarations get included at most once, even when the same header file is included several times (e.g., recursively by other header files). In more complex software projects, it avoids endless loops of header files mutually `#include`ing each other.

b. Implement the function `greet` in *greet.c*. It takes an integer argument but does not provide a return value. The function shall print *Hello World!* as often as specified by the integer argument, each in a separate line. In addition, the printed lines shall be prefixed with a consecutive line counter (starting at 1). Compile the provided template with `make`.

Solution:
Your implementation should look like this:

```
void greet(int times) {
    int i;

    for (i = 1; i <= times; ++i) {
        printf("%d␣Hello␣World!\n", i);
    }
}
```

In addition, you need to add `#include <stdio.h>` to the top of `greet.c`. This includes the declaration of `printf` into your source file, so that the compiler can check the function call for mistakes.

Question 0.3: Pointers and Structs

a. Explain the concept of pointers. What do the `&` and `*` operators do when working with pointers?

Solution:
A pointer is a data type that is pointing at a value. The pointer variable itself holds the address of the value in memory. We can give a pointer a type which refers to the type of data stored at the address. `void` denotes the absence of type. This is useful if the type of data that a pointer should point to is not known (e.g., a generic memory allocation function will return a void pointer). To define a pointer variable we precede its name with an asterisk (*).
`int *p; // a pointer to an integer value;`
The `&`-operator returns the address of a variable:

```
int a = 5;
int *p = &a; // p holds the memory address of the variable a
```

To access the memory location pointed at by a pointer, we can dereference the pointer with the `*`-operator:

```
int a = 5;
int *p = &a; // p holds the memory address of the variable a
int b = *p;  // dereference p, b = 5;
```

b. What are the types of the following variables. Why can this code formatting be misleading? `int* a, b;`

Solution:
a is a pointer to an integer and b is an integer. The position of the star can be misleading, because its scope seems to include b, but it does not.

c. Consider an array of `int`s in memory. How can you use the following pointers to access the fourth element? Both point at the beginning of the array.

```
int *ip;
void *vp;
```

Solution:
Since `ip` is of the correct type, we can use array notation or pointer arithmetic and dereferencing to access the fourth integer.

```
int a = ip[3];
int a = *(ip + 3);
```

`vp` is an untyped pointer. Before we can perform pointer arithmetic we have to cast it to the correct type:

```
int a = *((int*)vp + 3);
```

d. Acquire the template *p2* for this programming question from ILIAS and unpack it. Implement the function `countchr` in *countchr.c*. It returns the number of occurrences of a character in an ASCII string, both supplied by the caller. The string is represented as a pointer to a contiguous sequence of `char`s in memory. The string is terminated with a null (0) character.

Solution:
Your implementation should look like this:

```
int countchr(char *string, char c) {
    int count = 0;

    while (*string != '\0') {
        if (*string == c) {
            count++;
        }

        string++;
    }

    return count;
}
```

To get the character pointed to by the given `string`-pointer, we have to dereference it. A while loop iterates over all characters in the string by incrementing the pointer until it finds the null character.

Note that incrementing `string` does not affect the pointer variable that the parent function used as argument in the call to `countchr`.

```
char *str = "Hallo World!";
countchr(str, 'o'); // str not affected!
```

e. What is a *struct* in C? What can it be used for?

Solution:
A struct *forms a compound type out of several individual variables. It allows to store several pieces of data (possibly of different data types) in one variable. This comes in handy to handle complex data structures, especially when using* arrays *of* structs. *Operating systems typically use* structs *for control data structures. For example, each running process is typically represented in a* struct *called* process control block *(see the upcoming lectures on process management for details).*

The individual components of a struct *are called* members*. You can access the* members *of a* struct *using the . operator. Use the -> operator to access members of a* struct *via a pointer to the* struct*. See the following code listing for examples on how to use these operators.*

```
// type declaration
struct foo
{
    int a;
    float b;
    char c;
};
// ...
struct foo foo1, foo2;
struct foo *foo1_ptr = &foo1; // a pointer to a struct

// accessing members
foo1.a = 5;
foo1.b = 3.141 + foo1.a;

// accessing struct members through pointers
foo1_ptr->a += 5;
foo1_ptr->c = 'C';

if(foo1.a != 10) {
    printf("something went very wrong !\n");
}
```