Übung Echtzeitsysteme WS 2014 / 2015
Introduction to C

Philipp Heise

Exercise 0 Getting to know the GNU C compiler and CMake

1. *(Hello World!)* Create a file named *hello.c* containing the following tiny C program. In order to compile the program use the command *gcc hello.c -o hello* and execute the resulting executable using the command *./hello*.

```c
#include <stdio.h>

int main( int argc, char** argv )
{
    printf("Hello World!\n");
    return 0;
}
```

2. *(Read the documentation)* You should familiarize yourself with the possible flags for the GNU C compiler. Use your browser or the gcc manual to find out the necessary information. What do the flags *Wall, c, O{1,2,3,s},g* and *D* do?
3. *(CMake)* Throughout the exercises we will use *CMake* as our build tool. In general your project folder should contain one *CMakeLists.txt* file like the one shown below and two additional directories. One for the sourcecode named *src* and another one called *build* for all the generated files. To use *cmake* your current working directory should be the

```
cmake_minimum_required (VERSION 2.6)
project(Tutorial)

include_directories("${PROJECT_SOURCE_DIR}/src")
file(GLOB SOURCES "${PROJECT_SOURCE_DIR}/src/*.c")

add_executable(hello ${SOURCES})
```

   build directory. You can then call *CMake* using the following command *cmake ..* *CMake* should then generate a *Makefile* for you. In the *build* directory the *make* command can then be used to compile and link our executable. What does the build folder contain? Find out what the commands in the *CMakeLists.txt* file do using the official *CMake* documentation. Can you easily add another executable? How do you tell *CMake* to use special compiler flags? What is the effect of the command *make VERBOSE=1*?

Exercise 1 C Basics

In the following we suppose that you know basic data-types like char, int and float. Further you should know about basic control flow e.g. if/else, while and for. You should also know what functions are and what they are for.

1. *(Functions)* Functions are extremely useful for structuring and reusing common code. The abstract general structure of function declarations can be simplified to the following. The declaration contains a return type, a function name and argument types with their

```
type function( type0 arg0, type1 arg1, ..., typeN argN )
{
   ...
}
```

respective names. Below a function for calculating the factorial is given:

```
int factorial( int n )
{
    if( n <= 1 )
        return 1;
    else
        return factorial( n - 1 ) * n;
}
```

- Write a function that has no parameters and returns no value, but prints a message to the standard-output.
- Write a function for calculating the n-th Fibonacci number. The return value should be the result and the sole parameter should be n.

2. *(Pointers)* The name of a variable refers to a particular location in memory. We can use the name to retrieve or store a value from this memory location. If you refer to the variable by name then
   1. the memory address is looked up
   2. the value at the address is read or set.

   C allows us to perform these steps independently (given a variable x):
   - &x evaluates to the address of x in memory
   - *(&x) dereferences the address of x and retrieves the value of x
   - *(&x) is the same thing as x

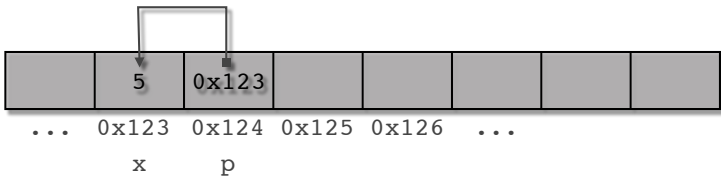
   The following code demonstrates this with an additional illustration of the memory (we assume x is at location $0x123$ in memory)

```
#include <stdio.h>

int main()
{
    int x;
    int* p = &x;

    x = 10;
    printf( "%d\n", x );
    *p = 5;
    printf( "%d\n", x );
    return 0;
}
```

   - The concept of pointers allows us to call functions that modify their parameters. Change your Fibonacci function so that the first parameter is a pointer to the memory location for the return value. The function should return nothing.



- Can you perform arithmetic operations like plus, minus etc. on the pointers? Can you assign arbitrary values to the pointer? What are the implications of this?
- What is necessary to change the address of a pointer that he is currently pointing to from within a function? What happens when you directly change the pointer inside the function? The current address of every pointer can be printed using *printf* and *%p*.

3. *(Arrays)* The general structure of array declarations in C looks like this. Array access

```
type name[ dimension ];
type name[ dimension1 ][ dimension2 ];
```

and various initialization techniques are shown below.

```
int array[ 4 ];

array[ 0 ] = 3;
array[ 1 ] = 5;
array[ 2 ] = 7;
array[ 3 ] = 42;

...

int array[ 4 ] = { 3, 5, 7, 42 };

...

int array[ ] = { 3, 5, 7, 42 };
```

In C just the name of an array is equivalent to a pointer of the same type, which points to the first element of the array. Indexing is equivalent to dereferencing a pointer with an offset. The following code snippet illustrates this equivalency.

```
int array[] = { 1, 2, 3, 4 };

int* ptr;
ptr = array;
// is identical to
ptr = &array[ 0 ];

array[ 2 ] = 5;
// is identical to
*( array + 2 ) = 5;
```

- Declare an array of a certain size and use 3 different techniques to fill the array with its index number. Use the normal bracket, the pointer plus offset and an incrementing pointer method to set the contents.

- Write a function to initialize an array with a fixed value. The parameters of the function should be a pointer to the array and the size of the array. Why is the size necessary? Can you use your function to set a single variable to the fixed value?

4. *(Memory management)* Local variables reside normally on the stack. The lifetime of variables on the stack is only as long as the function does not return. Heap memory is not affected by the current function and is persistent. To allocate and free memory of the heap several functions exist: *malloc,calloc* and *free*. The *stdlib.h* header must be included to use these functions.
   - Read the manuals for these functions and find out what they do.
   - What happens to the memory if *free* is not called.
   - Assume you have two pointers to the same memory location allocated using *malloc* - what happens to the other pointer if one of them is used to free the memory?

5. *(Structures)* Often it is useful to encapsulate several variables into one new data-type. In C this can be done with structures. The following simple example shows how to use them.

```
struct Vec2f {
    float x;
    float y;
};

int main()
{
    struct Vec2f v;

    v.x = 1.0f;
    v.y = 1.0f;

    struct Vec2f* vptr = &v;
    ( *vptr ).x = 2;
    // is identical to
    vptr->x = 2;

    return 0;
}
```

In order to avoid to write struct every time, we can use a typedef to allow the direct usage of Vec2f.

```
typedef struct Vec2f Vec2f;
```

- Write several functions to perform the following operations. The first parameter should always be a pointer to the *Vec2f* type (with the exception of *vec2f_new*).

  vec2f_new Allocate memory for the structure using *malloc* and *sizeof*.

  vec2f_delete Free the allocated memory.

  vec2f_zero Set the components to zero.

  vec2f_set Set the components to the by parameters provided values.

  vec2f_print Print the components.

  vec2f_add Add the components of two Vec2f and store the result in the third.

  vec2f_scale Scale the components using the value provided by parameter and store it into the destination.

Exercise 2 Common mistakes

The following code snippets contain some common mistakes. Spot the bug!

```
// Bug 1:
int* some_function( int a, int b )
{
    int retval;
    retval = a ^ b;
    return &retval;
}


// Bug 2:
void copydata( uint8_t* dst, const uint8_t* src, size_t n )
{
    for( size_t i = 0; i <= n; i++ ) {
        dst[ i ] = src[ i ];
    }
}

// Bug 3:

int* data = malloc( sizeof( int ) * 10 );
int* dptr = data;

dosomething( dptr, data );
free( data );
dosomethingdifferent( dptr );
```