ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Prof. Lothar Thiele

Embedded Systems - HS 2020

Embedded Systems Companion

Introduction

As its name indicates, the Embedded Systems Companion has been compiled to be a collection of useful definitions, reminders, and tips that will help you to successfully and joyfully go through the Embedded Systems lecture. At least, that's the plan :-)

Needless to say, we strongly recommend that you carefully read through the whole document at least once.

Contents

1 General Definitions

1.1 Software

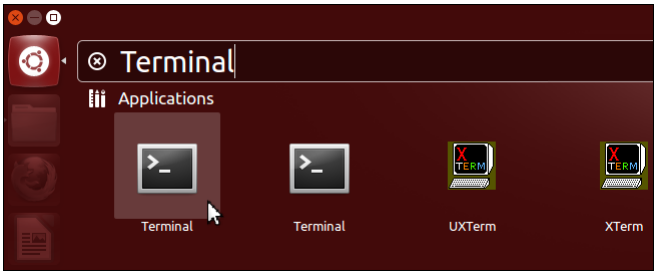
Integrated Development Environment

An *Integrated Development Environment* (IDE) is a software application that provides comprehensive facilities to computer programmers for software development. In the Embedded Systems labs, we use the Code Composer Studio IDE.

Terminal

A *terminal* is an interface for sending commands to and receivings outputs from a Linux system. It is a *textual user interface* (in constrast to a graphical user interface (GUI)).

A terminal *emulator* is a program that provides a terminal in a graphical environment. The most common one on Linux is called... 'Terminal'.



In the Embedded Systems labs, we use a terminal emulator included directly in Code Composer Studio.

Project

A *project* is the set of all the source files of a program. It also contains configuration information (which compiler to use, etc), including the intended target hardware. A large part of this information is stored in hidden files, e.g. `.ccsproject` (the filenames of hidden files start with "`.`").

Low-level programming language

A *low-level programming language* is a programming language that provides little or no abstraction. Such language typically uses instructions which are specific to the target hardware or CPU architecture. Assembly is the lowest-level of programming language you are likely to encounter.

High-level programming language

A *high-level programming language* (*e.g.,* Java or Python), is meant to be easier to use. This class of languages is not compiled at runtime (only interpreted) and is hence independent of the target hardware. However, it will generally require more memory, have longer computation time and be less capable of addressing a system's pecularities.

Operating System

An *operating system* (OS) is a software that manages the hardware and software resources of a computing platform and provides common services for computer programs.

Bare-metal programming

Programming bare-metal means to program an embedded system without using an underlying operating system like *e.g.,* Linux or FreeRTOS.

Instruction

An *instruction* (also sometimes called *statement*) is a basic operation (or combination of operations) that can be executed by a processor. The *instruction set* describes the set of operations that a processor can execute.

A computer programm is essentially a list of instructions.

Function

A *function* is a list of instructions that can be executed in multiple places in your code. A function takes (possibly) multiple parameters as input and returns (possibly) multiple output values.

Executable program

An *executable program* is a piece of software containing a list of instructions (machine code) that can be read and executed directly by a processor.

It is also referred to as an *executable* or *binary*.

Compiler

A *compiler* is a program that translates source code, written in some high-level programming language, to a lower-level language (*e.g.,* assembly language, or machine code) to create an executable program.

Compile, Build

In general, the *compilation* refers to the conversion of the source code into an executable program. In practice, it is a (very) complicated process that involves multiple steps and parameters (*e.g.,* the compilation flags).

Instead of *compile*, you may also find the term *build* (as it's called in Code Composer Studio). Strickly speaking, compiling is only one of the operations performed when building (which also include *e.g.,* linking).

Flash, Run, Debug

Flashing means loading an executable program onto the target platform's memory.
Running means starting the execution of the program in memory.
Debugging generally refers to stepping into the execution of the program, inspecting which instructions are executed, the current values of variables, etc.

In practice, these terms tend to be used interoperatively. For example, in Code Composer Studio, when hitting the *Debug* button, the IDE first flashes the program and then starts debugging.

Library

A *library* is a set of source or precompiled files that contains functions and definitions. Once you include a library in your project, all corresponding functions and definitions are usable in your own code.

Overflow

Standard computers possess plenty of memory and processing power. Contrarily,. embedded systems are rather limited. It is therefore important to carefully choose the integer types one uses: storing a counter value expected to range between 0 and 10 in a `uint64_t` wastes a lot of memory, as a `uint8_t` is sufficient!

However, one must be careful with *overflows*. A variable is said to overflow when it is assigned a value which is outside the range that its type can normally contain. For example, a `uint8_t` can contain any number between 0 and 255. What happens if you assign *e.g.,* 300 to such variable?

```
uint8_t a = 300;        // assign 300, even though the max value is 255
printf("%u", a);        // Prints: ''44''
```

300 requires 9 bits to be written in binary. As the variable "a" is only 8-bit long, only the lower 8 bits of 300 will be assigned to "a", that is 00101100, which is 44 in decimal; or put differently, 300%256.

Be also careful with subtractions on unsigned integers. Negative values are not defined! If you assign an negative value, the variable "wraps-around", like in the following example.

```
uint8_t a = 0;
a--;
printf("%u", a);        // Prints: ''255''
```

Warning 1: Stack Overflow

The term *overflow* is also used in the context of stack buffers. A *stack buffer overflow* occurs if the program writes to a memory address on the program's call stack outside the intended data structure.

1.2 Hardware

MCU

A *microcontroller* (or MCU – microcontroller unit) is a small computer on a single integrated circuit that contains one or more CPUs (central processing units) along with memory and programmable input/output peripherals. Program memory in the form of FRAM or Flash (several kB to few MB) is also often included on chip, as well as a small amount of RAM (few kB).

Microcontrollers are designed for embedded applications, in contrast to the microprocessors used in personal computers.

Registers

Registers are small memory blocks inside the processor with fast access times. Registers can have different sizes (*e.g.*, 8, 20, 32 bits) depending on the processor architecture.

Control registers are used to control the CPU at the lowest level possible, which is usually taken care of by the operating system (OS).

X-Bit CPU

A X-Bit CPU (*e.g.*, 32-Bit) refers to a processor architecture having X-Bit wide registers, typically using X-Bit memory addressing.

GPIO Pins

A *General-purpose input output* (GPIO) is a generic pin on an integrated circuit.

A *pin* is one physical line that conveys a binary information. From an logical point-of-view, a pin is either high (1) or low (0). From an electrical point-of-view, this may correspond for example to a voltage of +3V (1) and 0V (0) on the pin.

A *port* is a logical grouping of GPIO pins. The CPU embedded in the MPS432 Launchpad we use in the lab has 10 ports with 8 pins each. For example, `'P2.1'` refers to the pin 1 on port 2.

Target

The *target* of a program is the computing platform meant to run the program.

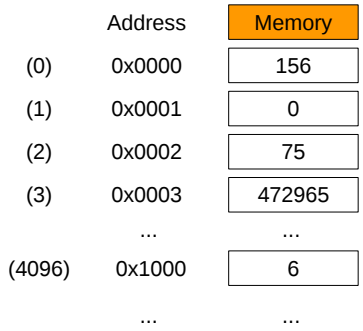
Debugger

A *debugger* is a hardware and software component used to flash, run, and debug an executable programm on a target.

Memory

At a high-level, *memory* can be seen as a linear array containing *values*. The index of a value in this array is called an *address*.

Addresses are usually written in hexadecimal format for legibility. The illustration below shows the correspondance in decimal in parenthesis.



2 C Programming Crash Course

2.1 Basics

main() Every C program must contain a `main()` function. This is the first function executed by the processor. When (or if) the main function returns, the program stops its execution.

End of line All instructions **must** be terminated by a semincolon ';'. If you forget one, the compilation will fail. Read the error log to find out which line causes the problem!

Comments There are two ways to write comments in C, as illustrated below. They are used to document the code.

```
1 ...       // A one-line comment
2
3 ...       /*
4            * A longer comment
5            * spreading across
6            * multiple lines
7            */
```

Variables In C, a variable is defined by an *address*, a *value*, and a *type*. A variable's *name* is only an alias for the variable's address in memory.

In memory, a variable's *value* is stored as a number in *binary* format. The variable's *type* allows the compiler to interpret this binary number. Example of types include integers (`int`), characters (`char`), etc.

Variables must be *declared* before they can be used. The declaration reserves the space in memory to store the variable and sets its type. A variable *definition* is a declaration plus the assignment of the variable's initial value.

```
1 <type> <name>;                        // Declaration without initialization
2 <type> <name> = <initial_value>;      // Declaration with    initialization
    = Definition
```

Recap 1: Variable naming

A variable's name can only contain lower- and upper-case letters, numbers, and underscores. It cannot start with a number.

Most significant bits The *most significant bit* (MSB) is the bit of a binary number having the greatest value (which depends on the number of bits: 2^{N-1} for a N-bit number), *i.e.,* the left-most one.

Least significant bits Conversely, the *least significant bit* (LSB) is the bit of a binary number having the smallest value, which is always 1, *i.e.,* the right-most one.

```
1 //     0b _ _ _ _ _ _ _ _
2 //          ^           ^
3 //       MSB             LSB
```

Endianness

The *endianness* refers to the sequential order in which Bytes are arranged into larger numerical values when stored in memory or when transmitted (over a bus, the radio, etc.).

In *big-endian format*, the most significant Byte (*i.e.,* the Byte containing the MSB) is stored/sent first.
In *little-endian format*, the least significant Byte (*i.e.,* the Byte containing the LSB) is stored/sent first.

2.2 Advanced Types and Keywords

Generic integer types

In C, the **number of bits used to encode integer types is not fixed**. This varies between implementations (*e.g.,* depending on the platform or the CPU). The problem is that when you declare an `int` variable in C, it is not clear how many bits are then reserved in memory. Thus, if you run the same program on different platforms, the size of an `int` can change, which may lead *e.g.,* to *overflows*.

To avoid this problem, embedded programmers use *generic integer types* (also called *fixed-width integers*). The idea is simple: the type `uintN_t` is used to store an unsigned integer number encoded with `N` bits. `N` can generally be 8, 16, 32, or 64. The following table summarizes these types and their value range.

Table 1: Ranges of 8, 16, and 32-bits fixed-width integer types

Unsigned types			Signed types		
Type	Min value	Max value	Type	Min value	Max value
uint8_t	0	255	int8_t	-128	127
uint16_t	0	65535	int16_t	-32768	32767
uint32_t	0	4294967295	int32_t	-2147483648	2147483647

Recap 2: Standard Integer Library

These generic types are defined in the `stdint.h` library. Don't forget to include it in your project, or the compiler will complain!

Qualifiers

A variable can be associated a *qualifier*. It will give additional information to the compiler about how the variable is intended to be used. Here are some qualifiers you should know.

const Short for *constant*. A `const` variable is not allowed to change its value at runtime. Thus, the variable value must be set with the declaration. It is defined as follows

```
1 const <type> <name> = <initial_value>;    // Definition of a 'const'
      variable
```

volatile The compiler usually performs complex optimizations to increase the performance of the binary. For example, it "removes" instructions that it sees as useless, like repeatedly reading the value of a variable which is not updated by the program.

The compiler assumes only the program can update the variables. However, external hardware events can also change the value of some variables. For example, when a button is pressed. In such cases, the `volatile` quantifier is used to tell the compiler that the value of the variable may change at any time – without any action being taken by the code.

To make it simple, the volatile qualifier turns off optimizations made by the compiler and therefore guarantees that all read and write operations on a volatile variable will be effectively executed by the target.

Qualifiers

static The effect of the `static` quantifier depends on where it is used:

Outside a function A variable declared `static` outside a function (in other word, a global variable) can be used only in the current file. For the code in the other files, this variable simply does not exist. This is used mostly to avoid name conflict (it reduces the scope of the variable).

Within a function A variable declared static within a function will hold its value until the end of the program execution (otherwise it would be reinitialized every time the function is executed). For example, it is commonly used for counting the number of times a given function executes, like in the example below. The declaration of `count` as `static` has the effect of *(i)* not reinitializing it to 0 every time the function executes and *(ii)* remembering its value between two executions.

```
int MyFunction(){
    static count = 0;
    count++;
    /*
     * MyFunction's code, with eventually instructions depending on
    the value of 'count'
     */
}
```

In summary, a `static` variable can be accessed only in the file it is declared. Furthermore, it is created once at the beginning of the program and kept during the whole runtime (even if defined within a function).

extern The *extern* qualifier can be used together with a variable declaration of a global variable and informs the compiler that the variable is defined elsewhere (typically in another source file). Therefore, the compiler does not need to allocate memory for this variable declaration. An external variable can be accessed by all functions in all modules of a program.

Structure

A *structure* is a customized type, constructed as a collection of variables of different types under a single name. These variables are so called *members* of the structure variable. A structure is declared with the keyword `struct`, as in the example below.

```
struct student{         // the structure name
    char   name[80];    // first member   an array of characters
    float  marks;       // second member  a decimal value
    int    age;         // third member   an integer
};                      // end of declaration
```

This declares a new type, which can then be used to declare (`struct`) variables as usual.

```
struct student studentA;    // Creates a variable 'studentA' of type '
    struct student'
studentA.age = 25;          // Assigns a value to the 'age' member of the
    'studentA' struct
```

A member `m` of a struct variable `s` can be accessed by using the "." operator: `s.m`. If `p` is a pointer to a structure variable, the "->" operator can be used to dereference the pointer and to access a member `m` at same time: `p->m`.

typedef

The `typedef` keyword is used to explicitly associate a type with an identifier, or name. Essentially, it creates an alias for the type. This is commonly used for renaming the structure types, but also for creating meaningful type names.

```
typedef struct student student_t;   // Create an alias for the type 'struct student'
student_t studentB;                 // Create a variable of type 'Student', which is in fact a 'struct student'
typedef int timer_t;                // Create an alias for 'int' to be used for timers
```

Casting

We mentioned that each variable has a type, which is used by the compiler to interpret the 'meaning' of the variable's value. *Casting*, also called *typecasting*, consists in virtually changing the type of a variable for one instruction operation. To typecast something, simply put the type you want the variable to act as inside parentheses in front of the variable, as in the example below.

```
int x = 40;
printf("x = %i\n", x);            // Print 'x = 40'
printf("x = %c\n", (char) x);     // Cast 'x' into a 'char' and print it
                                  // -> Print 'x = ('
```

Many numbers are *implicitely converted* into other types, e.g. when assigning values of a type to variable of another type or when passing variables to functions. In this case, no explicit typecast is necessary. However, the precision might be reduced in such an implicit conversion without your consent!

2.3 Strings

C-string

In C, a string is an *array of characters*. A character, or `char`, is an 8-bit integer.

```
myString            // a variable called myString
"myString"          // a string literal, i.e. representation of a string in the code, not a variable!
"1sdf _sdA^hb"      // another string literal
```

A string ends with a null character, literally the integer value `0`. Just remember that there will be an extra character at the end of a string, like a period at the end of a sentence. It is not counted as a letter, but it still takes up one character space. Thus, in a 50-char array, you can only 'use' 49 letters plus the null character at the end to terminate the string.

```
const char myString[] = "Embedded Systems is great!"; // Define a string (the size required for 'myString' is computed automatically at initialization)
printf("%s",myString);      // Print the whole string
printf("%c",myString[0]);   // Print the first character: 'E'
```

A string may also contain non-printable characters (identified by the backslash symbol '\'). You should know `\n` (New line – Moves the active position to the initial position of the next line) and `\r` (Carriage return – Moves the active position to the initial position of the current line).

```
printf("Embedded \nSystems\n is great!");
/* Print
 *       'Embedded
 *        Systems
 *        is great!'
 */
```

Warning 2: Array's index

Do not forget that in C, the indexing of arrays starts with 0, not 1.

printf()

The standard `printf` is provided by the `#include <stdlib>` library. As shown in previous examples, `printf` can display formatted strings. The format of the string may include specifiers (identified by the percent symbol '%') which are essentially placeholders for variables, passed as additional arguments to the `printf` function. The variable's value is then included in the string in the format given by the specifier.

```
const char myOpinion[] = "awesome";
printf("Embedded Systems is %s!", myOpinion);
/* Print
 *          'Embedded Systems is awesome!'
 */
```

Here is a short recap table of the specifiers you should know.

%c	Character
%s	String of characters
%d or %i	Signed decimal integer
%u	Unsigned decimal integer
%x	Unsigned hexadecimal integer

2.4 C operators

Arithmetic operators

Addition (+) subtraction (−) multiplication (*) division (/) and modulo (%)

Recap 3: Integer division and modulo

In "normal" calculation, $9/4 = 2.25$. However, in C, the result is 2. This is because the result of an operation on integers in also an integer. The compiler neglects the term after the decimal point. In an arithmetic operation with variables of mixed types, the compiler determines the result's type and (some of) the numbers are implicitly converted to other types before the evaluation. For example, the expression $9/2.0$ evaluates to 4.5 in C (automatic cast to float).
The modulo operator (%) computes the remainder. If `a = 9` is divided by `b = 4`, the remainder is `1` (*i.e.*, `a%b = 1`). The modulo operator can only be used with integers.

Increment

A short-hand notation; *e.g.*, "`a = a+1`", "`a += 1`" and "`a++`" both increment the variable a by 1. Similarly for decrement.

Assignment operators

Another short-hand notation (see Table 2).

Tests

Equal (==)	Greater (>)	Greater or equal (>=)
Not equal (!=)	Smaller (<)	Smaller or equal (<=)

Assignments vs Tests

Beware of one common mistake in C: the confusion between an assignment and a test.

- `a = b` is an *assignement*. It sets the value of `a` equal to the value of `b`.
- `a == b` is a *test*. It returns `1` if `a` and `b` have the same value and `0` otherwise.

Table 2: C Assignment Operators

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Logical operators Logical AND (&&), OR (||) and NOT (!)

Bitwise operators Bitwise AND (&), OR (|), XOR ($\wedge$) and complement ($\sim$)

Warning 3: Logical vs Bitwise operators

It is really important to understand the difference between logical and bitwise operators!

- A logical operator is *global* and returns a binary result (0 or 1).
- A bitwise operator operates *on each bit* individually and returns an integer (see Snippet 1).

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)


    Bit Operation of 12 and 25                  Bitwise OR Operation of 12 and 25
      00001100                                    00001100
    & 00011001                                  | 00011001
      --------                                    --------
      00001000  = 8 (In decimal)                  00011101  = 29 (In decimal)


    Bitwise XOR Operation of 12 and 25          Bitwise complement Operation of 12
      00001100
    ^ 00011001                                  ~ 00001100
      --------                                    --------
      00010101  = 21 (In decimal)                 11110011  = 243 (In decimal)
```

Snippet 1: Example of bitwise operators

Left- and right-shift

The left- and right-shift operators (< < and > > respectively) shift all bits by a certain number of bits. It is rather easy to understand them with examples:

```
212    =     11010100     // Unsigned int
212>>0 =     11010100     // No Shift
212>>2 =     00110101     // Right-shift by two bits
212>>7 =     00000001     // Right-shift by 7 bits

-10    =     11110110     // Signed int (two's-complement)
-10>>1 =     11111011     // Right-shift by one bit
```

```
212    =     11010100     // (In binary)
212<<1 =    110101000     // (In binary) [Left-shift by one bit]
212<<0 =     11010100     // (Shift by 0)
212<<4 = 110101000000     // (In binary) = 3392 (In decimal)
```

If the bits are interpreted as integer numbers, a right-shift is equivalent with integer-dividing the number by 2, and the left-shift is equivalent with multiplying the number by 2.

Note that, a left-shift can lead to an "increase" in the number of bits. However, keep in mind that this may only happen if there is "room left" in memory.

Note that for right-shifting signed integers, the bits inserted on the left need to be equal to the left-most bit before the shift in order to ensure that the variable's sign remains unchanged. This behaviour for negative numbers is implementation-dependent!

sizeof

The `sizeof` operator returns the size of data (constant, variables, array, structure etc) in Bytes, as illustrated below.

```
struct student{              // the structure name
    char      name[80];      // first variable   an array of characters
    int8_t    size;          // second variable  a decimal value
    int8_t    age;           // third variable   an integer
};
printf("%d bytes",sizeof(struct student));     // Prints ‘‘82 bytes’’
```

2.5 Preprocessor Programming

Preprocessor directives

Preprocessor directives are specific instructions executed before compilation (i.e. they are not executed by the CPU which runs the program). In C, preprocessor directives are easy to recognise: they start with the '#' symbol. These directives modify the source files on the text level: essentially they replace, insert, or delete blocks of text in the source files before the start of the compilation process.

There are three types of directives you should know:

`#include` This directive literally inserts the content of a source file in another one.

`#define` This directive replaces a word by an expression. For example, `MY_FAVORITE_LECTURE` replaces the string `"Embedded Systems"`.

```
1 #define MY_FAVORITE_LECTURE    "Embedded Systems"
```

This is performed by the *preprocessor* before the beginning of the compilation. Thus, the compiler does not 'see' `MY_FAVORITE_LECTURE`. When the compiler looks at the code, it is 'hard-coded' that your favorite lecture is '`Embedded Systems`'. Essentially, it has the same effect as running a 'Find/Replace all' command in your code.

`#if, #ifdef` These directives (and the associated `#else`, `#elif`, etc.) are used for conditional compilation.

Including files

A project often contains multiple files that must be *combined* together. This is done using the `#include` preprocessor directive. The common practice is to use it differently depending on where the files are located.

- To include a file from the compiler/system/IDE installation directories (more generally, anything in the include path), one uses `< ... >`
- To include a file from your project directory, one uses `" ... "`

```
1 #include <stdlib.h>      // 'stdlib.h'      is stored in the compiler
      installation directory
2 #include "myFunctions.h" // 'myFunctions.h' is stored in the project
      directory
```

Macro

The `#define` directive can also be use to associate any source code to a word (not only a static value like `"Embedded Systems"`). Such word is then called a *macro*. Macros can take parameters, similarly to functions. The macros are often written in full caps to deferentiate them from variables or functions in the source code.

```
#define MACRO_lab1_configureUART( config )                              \
{                                                                       \
    /* Selecting P1.2 and P1.3 in UART mode                  */        \
    GPIO_setAsPeripheralModuleFunctionOutputPin                         \
            (                                                           \
              GPIO_PORT_P1,                                             \
              GPIO_PIN2 | GPIO_PIN3,                                    \
              GPIO_PRIMARY_MODULE_FUNCTION                              \
            );                                                          \
    /* Configuring UART Module                               */        \
    UART_initModule(EUSCI_A0_BASE, config);                             \
    /* Enable UART module UART_enableModule(EUSCI_A0_BASE); */          \
}
```

In this example, the defined word is '`MACRO_lab1_configureUART( config )`' where `config` is a parameter. The preprocessor does textual replacement in the source code. If somewhere in the code it reads '`MACRO_lab1_configureUART( 3 )`', this will be replaced by the content of the macro with `config` replaced by 3.

The backslash symbol '\' is required to write a single macro accross multiple lines.

Conditional compilation

Preprocessor directives can also be used for *conditional compilation*. Essentially, you can tell the compiler that some instructions should not be executed, exactly as if they were commented out.

```
#if condition
    /* Source code to compile if condition is true */
#elif condition2
    /* Otherwise, if condition2 is true compile this code */
#else
    /* Otherwise, compile this code */
#endif
```

Using the `#define` directive, one can define a constant (*i.e.,* a word) without any associated value. This can then be used to do conditional compilations by using the `#ifdef` directive, as shown in the example below.

```
#define WINDOWS

#ifdef WINDOWS
    /* Source code for Windows */
#endif

#ifdef LINUX
    /* Source code for Linux */
#endif

#ifdef MAC
    /* Source code for Mac */
#endif
```

Similarly, the directive `#ifndef` ('if not defined') is extremely useful to prevent *infinite inclusions*. This would happen if a file `A` includes a file `B` which itself includes `A`. This is prevented by using an *include guard*, as shown in the example below.

```
#ifndef DEF_FILENAME     // If the constant 'DEF_FILENAME' is not yet
    defined, the file has never been included

#define DEF_FILENAME     // The constant is defined to prevent future
    inclusions

    /* Content of the file to include */

#endif
```

Predefined preprocessor constants

The preprocessor usually provides predefined constants that can be useful, *e.g.*, for debugging:

`__LINE__` Give the current (source) code line.

`__FILE__` Give the file name.

`__DATE__` Give the date of compilation.

`__TIME__` Give the time of compilation.

2.6 Functions

C functions

In C, functions can return at most one variable. But this variable may be of any type, including a `struct`. In case one wants to return more than one value, it is common to define a structure as a function output. Another approach consists in using pointers (see later). If a function does not return anything, the output type is set to `void`, which means 'empty'.

```
int MyFunction(){    // A function taking no input and returning an 'int'
    int output = 0;
    /*
     * MyFunction's code
     */
    return output;
}
```

Call by value

It is important to remember that a function **does not modify** the value of its input variables! Instead, a local copy of the variables is created. This is referred to as 'calling by value'. See the example below.

```
void triple(int var){
    var = 3*var;
    printf("%d", var);
}

int main(){
    int x = 10;

    printf("x = %d", x);     // Print: 'x = 10'
    triple(x);               // Print: '30'
    printf("x = %d", x);     // Print: 'x = 10'

    return 0;
}
```

Prototype

A function can also be declared using a *prototype*. A prototype is an instruction describing the inputs and output types of a function. Once the compiler has seen the prototype, it will know what to do if the function is called... assuming you still define the function somewhere in the project!

```
void MyFunction(int var1, char var2);    // Prototype of 'MyFunction', defining the types
                                         // of the inputs parameters and the return type
int main(){

    /*
     * some instructions
     */

    MyFunction(x,y);                     // The function can be called here
                                         // even though it has not been defined yet
    /*
     * some more instructions
     */

    return 0;
}

void MyFunction(int var1, char var2){    // 'MyFunction' declaration
    /*
     * MyFunction's code
     */
}
```

Header files

A *header file* is a type of source file that contains function prototypes, definitions and macros. It is distinguished by its file extension: `file.h`. This separation of code into separate files is considered good practice to keep the code structure clean. However, it is not enforced by the compiler, i.e. it's possible to have all the code in `.c` files only.

2.7 Addresses and Pointers

Address

The *address* of a variable identifies the memory block where the variable's value is stored. A variable address can be accessed using the ampersand symbol '&' as in the following example. This is important in particular when working with *pointers*.

```
int studentA_grade = 6;
printf('%d',  studentA_grade);    // Prints '6'
printf('%d', &studentA_grade);    // Prints '47965', i.e. the address of 'studentA_grade'
```

Pointers

A *pointer* is a specific type of variable intended to store an address (compared to regular variables, which contain values).

The initialization of a pointer is not conpulsory, but **strongly recommended**. The keyword `NULL` is often used to initialize a pointer. `NULL` is defined as the integer value 0, which is regarded as "invalid address".

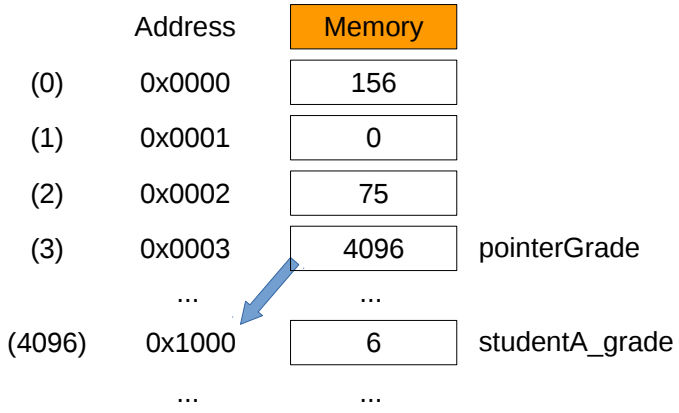
```
int *pointerName;           // a pointer on 'int', not initialized.
int *pointerName = NULL;    // a pointer on 'int', initialized to ''no address''
```

A pointer contains the address of a memory location (it 'points' to this memory location). It may also be used to access the value of the memory location it points to. This is called *dereferencing* a pointer. The value is obtained using the asterisk * symbol as shown in the example below.

```
int studentA_grade = 6;          // a variable of type 'int'
```

```
int *pointerGrade  = NULL;       // a pointer on a variable of type 'int'
pointerGrade = &studentA_grade;
printf('%d',pointerGrade);       // Returns the value of pointerGrade,
                                 // which is the address of studentA_grade
printf('%d',*pointerGrade);      // Returns the value of studentA_grade, that is: 6
                                 // We 'dereference' the pointer pointerGrade.
```

If we look in memory, here is what we would find



Call by reference

Pointers are interesting to use as function inputs. Remember that functions use the values of their inputs. The value of a pointer however is in fact the address of another variable. Thus, the function can use the address to **directly modify** the value of the variable pointed to! This technique is referred to as 'call by reference'.

```
void triple(int *x){             // The function takes a pointer as input
    (*x) = 3 * (*x);             // Dereference the pointer to work
                                 // on the pointed variable directly
}

int main(){
    int x = 10;                  // Define 'x'

    printf("x = %d", x);         // Print: 'x = 10'
    triple(&x);                     // Call 'triple' with the address of 'x
    '
    printf("x = %d", x);         // Print: 'x = 30'

    return 0;
}
```

3 Good Programming Practices

The ordering is arbitrary. It is used for references purposes only.

(a) Be generous when writting comments! There is a saying: "a good piece of code is one you can understand by only reading the comments"... Write comments! You will thank yourself later.

(b) The initialization of variables is optional. However, if it is not initialized, the variable does not have any defaut value, but simply inherits the current value stored in memory at this address, **which can be anything!** It is good practice to always initialize the variables you declare.

(c) It is good practice to write functions in a specific source file (say `myFunctions.c`) and to have a corresponding header file with the prototypes, usually with the same name (so, `myFunctions.h`). Then, it is sufficient to include the header file at the beginning of any source file were you want to use your functions. It helps to stay organized!

```
1 #include "myFunctions.h"
```

(d) Prefer the use of `#define` instead of hardcoded values in your code. This improves the maintainability and the readability of the code.

(e) Always use include guards in your header files. The problem of infinite inclusion may sound stupid at first, but it is in fact almost unavoidable in more complex project! Therefore, better be safe and ensure that files can be included only once.

(f) Some type conversions can be implicitly performed by the compiler. For example, if you pass an `int` as input to a function expecting a `char`. However, it is considered good programming practice to **always use the cast operator** whenever type conversions are necessary. Relying on the compiler to do it for you is always dangerous.