An Abridged Introduction to C

Foreword

This is intended as an introduction to the programming language `C` for students who are already fluent in at least another programming language like JavaScript, Python, Perl, PHP, FORTRAN, Java, etc.

It is also a hands-on, example-based exposition.

Among other things, this means that many details are left for the student to pick them up along the way either by analogy with other programming languages they know, or by promptly asking the teacher.

C

What is C

C is a **high-level language**. This means programming in `C` is based on code that resembles a familiar, natural language, in this case English. Other examples of high-level languages, in addition to those mentioned in the foreword, could be C++ or Haskell.

A counter-example, that is, an example of a language that is not a high-level one, would be the mnemonic-based, programming language called *assembler*.

This, however, is actually a relative concept: assembler is at a higher-level than writing directly in binary code, i.e., 0s and 1s!

What follows is an example of program in assembler written for the Mac OSX and to be *assembled* with the NASM assembler:

```
; helloworld.asm - a "hello, world" program using NASM written in Assembler for the MacOSX
; To create an executable called 'helloworld-as' from a code file called 'helloworld.asm':
; 1) nasm -f macho helloworld.asm (for Mach OSX) | nasm -f elf helloworld.asm (for Linux)
; 2) ld -o helloworld-as -e mystart helloworld.o

section .text

global mystart                  ; make the main function externally visible

mystart:

; 1 print "hello, world"

    ; 1a prepare the arguments for the system call to write
```

```
    push dword mylen            ; message length
    push dword mymsg            ; message to write
    push dword 1                ; file descriptor value (stdout)

    ; 1b make the system call to write
    mov eax, 0x4                ; system call number for write
    sub esp, 4                  ; OS X (and BSD) system calls needs "extra space" on stack.
    int 0x80                    ; make the actual system call

    ; 1c clean up the stack
    add esp, 16                 ; 3 args * 4 bytes/arg + 4 bytes extra space = 16 bytes

; 2 exit the program

    ; 2a prepare the argument for the sys call to exit
    push dword 0                ; exit status returned to the operating system

    ; 2b make the call to sys call to exit
    mov eax, 0x1                ; system call number for exit
    sub esp, 4                  ; OS X (and BSD) system calls needs "extra space" on stack
    int 0x80                    ; make the system call

section .data

  mymsg db "hello, world", 0xa  ; string with a carriage-return
  mylen equ $-mymsg             ; string length in bytes
```

In that sense, `C` is usually referred to as **low-level language** when compared to more modern ones like `Haskell` or to *scripting* languages like `Python`.

In general, the strength of a lower-level language lies in a more direct access to the machine's components and more freedom to do almost anything. That's certainly the case of assembler and, in lesser way, `C`. That comes at a cost, however.

Indeed, a high-level languages also means that the code you write has a higher level of **portability**, that is, it can usually be converted into executable code (*compiled*) and *run in many other OSs and CPUs without modifying the code*! The least portable program is one written in assembler: It depends on the specific CPU family, the actual assembler program that transforms it into 0s and 1s (see here) and the OS of the target machine (see here) !

Another distinction one makes is about **expressivity**. Higher-level languages like Haskell or JavaScript are more expressive, meaning it's easier to *express*, i.e., implement into code our ideas. For instance, in JavaScript a function has the same status as any other variable and can thus be passed around like such. In C, while there are ways to that in specific cases, it's neither as easy nor always

possible. In Haskell, for instance, it is easy to redefine the basic operators as `+, *, ==, etc` to work with our own data type *exactly as we use them for say numbers*! For instance, we could define our special type of list of 3 numbers, called it `mylist` and declare that two such lists are equal if the middle elements are equal. If `mylist1=[1,2,3]` and `mylist2=[7,2,103]` were two such lists, we would simply write `mylist1 == mylist2` in order to compare them. In this case, the comparison would return `true`!

Finally, current, popular high-level languages like Python, JavaScript, Perl or PHP *hide a lot of delicate, low-level details from the programmer* (e.g., specifying the type of variables or allocating the memory needed to store an array's content) which are done *automatically, and under the hood* by the interpreter or the compiler. This hiding of details is sometimes referred to as **abstraction**. The main goal of these languages is allowing for *fast prototyping* of ideas.

Origin of C

The operating system UNIX was developed by AT&T Bell Laboratories in 1971, in the USA. The first version of UNIX ran on the DEC PDP-7, an earlier version of the DEC PDP-11, and was written in Assembler.

In order to be possible to *port* UNIX to other type of machines they required to rewrite into a *higher-level language*. Such language had to be still capable of running UNIX very fast and provide access to the computer's hardware like the memory or the registers of the CPU.

Brian Kernighan and Dennis Ritchie set out to design such a language and implement the tools needed to use it. In 1978 they published their famous book "*The `C` Programming Language*" that became the *de facto* standard with the spread of UNIX as OS.

Standards

The same as natural languages like English, Spanish or German, programming languages *evolve* with time.

What happens is that new features get usually added to the language, some restrictions get removed and others enforced increasing thereby its expressivity and portability.

The first standardization of C, known as ANSI-C, was established in 1989 and lead to the unification of the different *dialects* of `C` that existed before.

The current standard of C, as of 2018, is called `C11` (technically it is the standard ISO/IEC 9899:2011).

Classification of `C`

C is considered an **imperative language**. This is a language that consists of variables, which are labels for locations in memory, and a series of instructions that manipulate the data[1].

The nature of imperative languages is directly related to the von Neumann's architecture of a computer. There the code that says how to manipulate the data and the data itself, both reside in the same memory of the computer.

The central point of an imperative language is the **Algorithm**. An algorithm is a precise recipe for solving a problem. All operational steps and their order must be established to the last detail in order to achieve the desired result.

The mindset of the programmer when writing some code is sometimes referred to as the *conceptual pattern* she uses in writing that code.

The *conceptual pattern* for the programming language `C` is the **paradigm of procedural programming**. This consist in the idea that code that gets repeated often should be solved by the concept of a procedure, that is, a subprogram containing some instructional steps that gets assigned a name. In `C` a procedure is simply a function.

Using instead an *object-oriented (OO) paradigm* (through a OO-oriented language) we think on the solution of our problem as creating objects that can be used only in very specific ways and with specific properties; in a functional language, however, we only thing instead in terms of functions mapping some input data into some other data which in turn become the input to another function and so on.

Opposite to the imperative languages are the **declarative**[2] **languages**. We don't tell the computer *how* to solve a problem, but only *declare the key relationships* that the solution must satisfy. Some examples are Haskell, which is a purely functional language, `Prolog`, a logic programming language, `SQL`, a standard for accessing and manipulating databases, or `Modellica`.

Examples of languages and the (main) paradigm they offer:

- imperative languages are most of the best known as are the
  - **machine-oriented**, chiefly `Assembler`
  - **procedural** languages like `C`, `FORTRAN`, `Perl`, `PHP`
  - **object-oriented** ones like `C++`, `C#`, `Scala`, `Java`, `Python`, `JavaScript`,...
- declarative languages are still more niche ones like the
  - **functional** languages like `Haskell` or `Lisp`
  - **logical-programming** languages like `Prolog`
  - **Relational-database** languages, like `SQL` or the modern `AQL`

[1] See for more wikipedia's page here.
[2] See wikipedia's page here.

Most modern languages, however, tend to be *multi-paradigm*, meaning, they allow us to design our code in procedural way, or an OO one or a functional language, all possibilities within the same language. Examples are `C++`, `JavaScript`, `Python`, `Scala`, where these all allow for programming in a procedural, OO or functional way.

Abstractions

The advantage brought by high-level languages like `C` lies in three improvements in abstraction level:

- Abstraction in expressions
- Abstraction wrt control structure (e.g, if_then_else).
- and Data abstraction

Abstraction can be thought of as a higher-view level that hides the lesser important details making visible those that are important to us.

Abstraction of expressions

In general, an expression is a concatenation of operands, operators and parentheses, as, e.g., in $9*(13+5)$. This is an abstraction of what actually is required: In Assembler we would need to first load each number into a register (e.g. eax,rpi, etc.), then add them, then implement a multiplication!

Homework: How can we multiply any number by 9 in binary? Define an algorithm, i.e., a defined set of steps, that on paper would do it.

Abstraction of control structures

A control structure refers to an instruction that affects the order of execution of other instructions.

Example: In the early versions of `FORTRAN` an `if-then` control-structure would be written as

```
      IF ( X - Y ) 10, 300, 500
10    ...
      GOTO 600
300   ...
      GOTO 600
500   ...
600   ...
```

If x would be smaller than y, the flow of the program would continue at *the line of code labelled by 10*, execute the statements found there until the statement

`GOTO 600`, which instructs the computer to continue with the code at the label 600; if equal, it would *jump* to the line labelled by 300 until the second `GOTO 600`; and, finally, if x would be larger, it would jump to the line labelled 500 and continue from there.

Assembler is even more primitive. There we also have the unconditional jump `GOTO`, with mnemonic `jmp`, but the `if-then` doesn't exist. Instead we have *conditional jumps* that depend on whether the result of the last operation was smaller than zero or not, or if it was simply equal to 0[3].

Puzzle: How could we write the previous `FORTRAN` example only with such conditional and unconditional jumps?

Homework: Write the previous `FORTRAN` example in `C`.

Clearly, the code between the GOTOs is a block of statements that in some sense work as a single statement: only in specific cases will each block be executed. To denote a block structure in `C` we use the braces `{,}`. From the location of the braces, the compiler then sets on its own the corresponding labels for all the jumps.

Data Abstraction

Data abstraction seeks to separate the *representation of the data* (e.g., do we model our data as an array or as a set of independent integer variables?) from the description of the *operations we can do on/with the data*. The advantage is a higher portability and better security.

Through data abstraction, the details of the representation of the data are hidden from the programmer. She only needs to know the *signature* of the operations on them. In other words, she knows only what operations are possible, what parameters they require and what are their return types.

Sometimes you may see the words *syntax* referring to the signature of the operations, and *semantic*[4] referring to what the operations do and how.

Example: What does it mean $7 * 2$? Well, we know the *meaning* of that operation. We also know the *syntax*: first write one number, then comes the asterisk and then the other number. The implementation in binary could consist in just shifting all the bits of 7 one position to the left.

The key concept in data abstraction is the concept of **Abstract Data Type (ADT)**. Basically, it refers to arbitrary (say, user defined) representations of data with arbitrary operations on it. Those operations require a precise signature (syntax) and a meaning (semantic) that needs to be implemented.

[3] For more examples of mnemonics available in assembler, see here

[4] This is a word that comes from Greek and means "meaning" or "related to the meaning".

Example: A stack is data structure where one can only add an element or remove an element to/from it. In particular it's a FILO (first-in, last-out) type of container.

However, we insist that once that is implemented, the ADT is defined only by its operations!

Solving problems computationally, ultimately boils down to define the *appropriate* ADT for each problem. Appropriate here means not only to be able to address the relevant feature we want to describe but also implement the operations in an **efficient** way!

Programming in C

Quick summary

(One of) The most basic programm

```c
#include<stdio.h>

int main(){
    float a = 1.5, b=3.0, x;
    x = a*b ;
    printf("a=%f, b=%f : a*b=%f\n",a,b,x); // a=1.500000, b=3.000000 : a*b=4.500000
    return 0;
}
```

In order to print out information on screen we need to use of one of the basic **C libraries**[5], the `stdio.h`. Without it we can't use the `printf` *instruction*.

In order to "load" the library we use the *statement* `#include <stdio.h>`. Other common and useful libraries are, e.g., the `stdlib.h` or the `math.h`.

Any statement that starts with the *number (pound) sign* `#` is not an actual instruction in `C`, but an instruction for the compiler. In this case, we are telling the compiler "*before you start processing the code we wrote,* ***copy&paste (include)*** *the code that's in the* ***`stdio.h`*** *file (library) right before our code, as if we had written it there ourselves*".

Every `C` program needs to have a `main` section (technically, a *function*). When we run a program, the computer starts executing the main section first. Actually, that's the only stuff that the computer will execute at all!

All `C` statements **must** end with a semi-colon `;`.

[5] More precisely, we use a header file of the **C standard library**. The latter is a collection of header files each providing a set of algorithms for dealing with different problems, e.g., printing, or comparing arrays or copying strings, etc.

Anything after `//` and till the end of the line is considered a **comment** intended for the programmer and is thus ignored by the compiler. We can write multiple-line comments by enclosing them between a starting `/*` and an ending `*/`.

Types

We must tell the compiler how we intend to use each and every variable we use. That's done by specifying the **type** of each variable.

```
int a = 3 ;      // a can only store integer values
float x = 11.1; // x is meant to store fractionary numbers
char c = 'A' ;  // c can only store a (single) character value
```

A `float` refers to *floating point number*[6]. Floating point numbers are represented in three parts: a sign, a mantissa (or significand), and an exponent. Given such a representation with sign s, mantissa m, and exponent e, the corresponding numerical value is $sm\hat{2}^e$.

We can use the constants `FLT_MIN` and `FLT_MAX` to find the smallest and largest positive (finite) float numbers that we can represent with such a type.

User defined type: `struct`

```
#include<stdio.h>
#include<string.h>

struct dwelling {
    char street[20];
    int number;
    char postalcode[7];
    char city[20];
};

struct student {
    int id;
    char name[20];
    char surname[20];
    struct dwelling address;
};

int main(){
    struct student randa = {
            123,
```

[6] In Mac OS X, open a terminal and type `main float` for more information. Use info in Linux to find more details.

```
            "randa",
            "adnar",
            {
            "Prince Arthur",
            35,
            "M5R2M8",
            "Toronto"
            }
    };
    printf("%s's postal code is %s\n", randa.name, randa.address.postalcode );
    struct student nic, sam;
    strcpy(nic.name, "nic"); // normal assignment of an array variable (name = "nic") doesn
    strcpy(sam.name, "sam");
    struct student class_csg12[3] = { randa, nic, sam};

    int i;
    for(i=0; i<3; i++) printf("%s\n", class_csg12[i].name );
    return 0;
}
```

The Caesar Cipher

The one-character cipher

Save the following code in a file called `caesar-ch.c`. Then generate an executable by compiling it using the Gnu `gcc` compiler in the following way: `gcc caesar-ch.c -o caesar-ch`.

This will create an executable called `caesar-ch` in the current directory.

```
#include<stdio.h>

int main(){
    char c = getchar();
    printf("You typed in a %c with ascii code %d\n",c,c);
    c += 3 ;
    printf("We'll encode it into code %d which corresponds to ascii char %c\n",c,c);
    return 0;
}
```

We can execute it by issuing the following in the shell prompt: `./caesar-ch` (press enter after typing all these characters). The program will then wait for you to type in a character, say `w` and see what the output says.

Remarks:

- Indentation is not mandatory but highly recommended. Keep the code easy to read!
- The braces `{,}` delimit what's called a *scope*: any variable defined inside a scope is only visible there.
- All lines must end with a semi-colon, except after an opening or a closing brace.
- Both variables and functions **require** to explicitly declare their ***type***. In this example, the return type of main is `int`, meaning an integer, and that of the variable `c` is a `char`, meaning one single character.
- There must always be one and only one single function called `main`. This is always were the program will start when it is executed.
- The functions `getchar` and `printf` come with `C` but wee need to *include* their code in our program. This we do by way of **including the *library* that contains their definition, namely the standard `C` input/output library `stdio.h`**.
- Inclusion of standard libraries must be done at the very beginning. The syntax is `#include<_name_of_lib_>`.
- The function `getchar` returns the next character read from the standard input device (the keyboard by default). By using the less-than character `<` in the command line when executing the program, say

  ```
  ./caesar-ch < input_text
  ```

  we can make the program read instead from the file `input_text`.
- The function `printf` can be given many arguments. The first one, however, must be a *literal* string, as in these examples. This string contains text that we want to print out as well as **formatting codes** (e.g., `%s` -for a string-, `%d` -for an integer- , `\n` -for inserting a new line-), that help us printing the additional arguments.

The Caesar cipher ver 0

```c
#include<stdio.h>

int main(){
    char c;
    while ( (c=getchar()) != EOF ){
        if( c == '\n' ) printf("%c",c);
        else printf("%c",c+3);
    }
    return 0;
}
```

Remarks:

- The `while` *keyword* denotes another form of a **loop** control-structure. It's syntax is

  ```
  while ( _expression_ ) { _code_ }
  ```

  It keeps repeating the code within its block while the expression evaluates to true. It stops when it's false and the the program flow continues right after the `while` block.

- The conditional expression consists in a **`value != EOF`**. Here **`EOF` denotes the character that the `getchar` function gets when it tries to read beyond the last character of its input**[7].

- The value in that expression is determined as follows: First a call to `getchar` is made, then its return value is assigned to the variable `c` (`c=getchar()`); finally, that value is the one used in the comparison.

The Caesar cipher ver 1

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
// We need stdlib.h in order to use atoi function
// We need string.h in order to compare strings

/*


Make it encrypt or decrypt depending on a command-line option.

Usage: enc_caesar option < file

Input: option
    option := (-e | -d) int
    int := the caesar de/en-cryption code.

       Notice, we require exactly 2 (two) arguments, the en/de-cryption switch
       and its code.

File is read from standard input by default.

Output: file encrypted or decrypted depending on given option

For using an option program needs to read the command-line text,
```

[7] Internally, `EOF` is represented by the integer -1.

```
more specifically what's called the command-line "arguments" of the program.

This is done using the argc and argv variables. Their type can be read from
the signature of the main function arguments.

argc is an integer counting the number of arguments given, including the
     very own program call.
argv is a character array containing the actual text of the command line up to,
     and w/o including the '<'.

*/

int main(int argc, char** argv){
    argc--; //we decrement by 1 the argument counter as the first argument argv[0]
            //corresponds to the program name
    argv++; //analogously, we increment the 'pointer' to the next command-line
            //arguments. After these 2 lines, we point to the switch.
    if ( argc != 2 ) {
        printf("ERROR: expecting 2 arguments, en/de-cryption switch and "
            "its code, but found %d\n",argc);
        return 1; //an exit value != 0 from main is standard for indicating an
                  //error happened while running the program.
    }
    char *option = argv[0] ; // string containing either "-e" or "-d"
    int edc = atoi(argv[1]) ; // second argument is the en/de-cryption code;
                              //we need to make sure to convert it to an integer
    //uncomment following line for testing
    //printf("Proceeding with option %s and code %d\n",option,edc);
    char c;
    while ( (c=getchar()) != EOF ){
        if ( strncmp(option, "-e",2 ) == 0 ) c += edc ;     // encrypt
        else if ( strncmp(option, "-d",2) == 0 ) c -= edc ; // decrypt
        else {
            printf("ERROR: unexpected option %s ...\n",option) ;
            return 1;
        }
        printf("%c",c);
    }
    return 0;
}
```

Remarks:

- Anything we write *after* two forward-slashes `//`, *and up to the end of that line* **is ignored by the compiler**. These are **1-line comments**.
- Anything *between* `/*` and `*/` is also considered comments. This type of

comment **can extend over multiple lines**. **Always add enough, but concise comments to your in order to help others (and yourself years later) to understand the meaning of your code**!

- In order to read the content of the line we issue when executing our program, we need to declare two parameters for `main`: `argc and argv`. The first is of type `int` (for integer).
- The second parameter of `main` is of type `char **`. There are several points of view to help us understand this type:
  - **argv is an array (aka list) of elements, each of type `char *`, i.e., each of type, an array of elements, each of type `char`**. But, an array of chars is exactly what we understand as a **string**. Whence,...
  - **argv is an array of strings**. That is, we can picture it as `["string1", "string-thing-2", "this-is-tedious",...]`.
  - **argv is a pointer to an array of pointers, each of them in turn pointing to an element of type `char`**. We can picture this in the following way:
    - `char c` means variable c is of type `char`, a character. Example: c='a'.
    - `char * ptr_c = &c`, here `ptr_c` is a **pointer to a `char`**. This means, `ptr_c` is a variable that contains the address in memory of a thing of type `char`. In this particular case, `ptr_c` is the address in memory of the `char` variable `c`. The syntax `&c` means **the address of the variable `c`**.
    - `char * str = "This is a pointer to chars too"` here `str` is again a pointer to a `char`. In particular it points to the address in memory of the first character of the string `"This is a pointer to chars too"`. The *pointer logic* allows for a unifying view of many "string-like" things like strings and arrays: We can do `printf("%c %c %c",str[0],str[8],str[10])` which would print `T a p`.
    - `str[0]` means the address contained in str, whence it points to the first, capital `T`;
    - `str[8]` means the address contained in str + 8 more chars, whence it points to the first, `a`;
    - `str[10]` means the address contained in str + 10 more chars, whence it points to the first, `p`;
- The function call `strncmp( option, "-e", 2)` compares the first 2 characters of the string `option` with the literal string `-e`. It returns an integer greater than, equal to, or less than 0, according as the string s1 is greater than, equal to, or less than the string s2.
- This code contains several checkings for reducing the risk that the user uses the program incorrectly. Introducing this kind of code is part of

working out the **human-computer interface** of your program. The goal of this area of computer science, called **Human-Computer Interaction (HCI)**, is to understand what are the best ways for humans to interact with computers and thus, to help **determine a simple, but logical and versatile interface to our programs**.

It's not easy to determine what such an interface is for any particular application, and, in general, programmers look at this task with certain contempt. They tend to ignore it and/or not devote enough effort into it, claiming it is the user's responsibility of learning how to use their program. This attitude, however, is nothing but hubris in an attempt to disguise the shortcomings as a programmer. A more humble and useful approach is to accept that we can probably always improve the interface to our programs.

The Caesar cipher ver 2

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
// We need stdlib.h in order to use atoi function
// We need string.h in order to compare strings

/*


Make it encrypt or decrypt depending on a command-line option.

Usage: enc_caesar option < file

Input: option
    option := (-e | -d) int
    int := the caesar de/en-cryption code.

       Notice, we require exactly 2 (two) arguments, the en/de-cryption switch
       and its code.

File is read from standard input by default.

Output: file encrypted or decrypted depending on given option

For using an option program needs to read the command-line text,
more specifically what's called the command-line "arguments" of the program.

This is done using the argc and argv variables. Their type can be read from
the signature of the main function arguments.
```

```
argc is an integer counting the number of arguments given, including the very
     own program call.
argv is a character array containing the actual text of the command line up to,
     and w/o including the '<'.

*/

int main(int argc, char** argv){
    argc--; //we decrement by 1 the argument counter as the first argument
            //argv[0] corresponds to the program name
    argv++; //analogously, we increment the 'pointer' to the next command-line
            //arguments. After these 2 lines, we point to the switch.
    if ( argc != 2 ) {
        printf("ERROR: expecting 2 arguments, en/de-cryption switch and its "
               "code, but found %d\n",argc);
        return 1; //an exit value != 0 from main is standard for indicating an
                  //error happened while running the program.
    }
    int option ; //1 := encrypt ; -1 := decrypt
    if ( strncmp(argv[0],"-e",2)==0 ) option =  1 ;
    else if ( strncmp(argv[0],"-d",2)==0 ) option = -1 ;
    else {
        printf("ERROR: unexpected option %s ...\n",argv[0]) ;
        return 1;
    }
    int edc = atoi(argv[1]) ; // second argument is the en/de-cryption code; we
                              //need to make sure to convert it to an integer
    //uncomment following line for testing
    //printf("Proceeding with option %s and code %d\n",option,edc);
    char c;
    while ( (c=getchar()) != EOF ){ //keep reading a character while it's not End-Of-File
        c = c + option * edc ;
        printf("%c",c);
    }
    return 0;
}
```

Remarks:

- This version contains only slight changes in the logic of the program. We first determine what option (encrypt or decrypt) did the user choose and store that information as an integer `option` (1 or -1). The choice of values is justified by the way we use this variable inside the `while` loop: decrypting is just using the same shift value but "to the opposite sense as during encryption". Whence it boils down to changing the sign of the shift value `edc`.

The Caesar cipher ver 3

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
// We need stdlib.h in order to use atoi function
// We need string.h in order to compare strings

/*


Make it encrypt or decrypt depending on a command-line option.

Usage: enc_caesar option < file

Input:  int
    int := the caesar de/en-cryption code.

       Notice, we require exactly 1 (one) argument, the en/de-cryption code.

File is read from standard input by default.

Output: file encrypted or decrypted depending on given option

For using an option program needs to read the command-line text,
more specifically what's called the command-line "arguments" of the program.

This is done using the argc and argv variables. Their type can be read from
the signature of the main function arguments.

argc is an integer counting the number of arguments given, including the very
     own program call.
argv is a character array containing the actual text of the command line up to,
     and w/o including the '<'.

*/

int main(int argc, char** argv){
    argc--; //we decrement by 1 the argument counter as the first argument
            //argv[0] corresponds to the program name
    argv++; //analogously, we increment the 'pointer' to the next command-line
            //arguments. After these 2 lines, we point to the enc/dec code.
    if ( argc != 1 ) {
        printf("ERROR: expecting 1 argument, the en/de-cryption code, "
               "but found %d\n",argc);
```

```
        return 1; //an exit value != 0 from main is standard for indicating an
                  //error happened while running the program.
    }
    int edc = atoi(argv[0]) ; // first argument is the en/de-cryption code; we
                              //need to make sure to convert it to an integer
    char c;
    while ( (c=getchar()) != EOF ){ //keep reading a character while it's not End-Of-File
        printf("%c",c+edc);
    }
    return 0;
}
```

Remarks:

- If consider the code of version 2, we can see that we do not actually need to track whether the user wants to encrypt or decrypt: In order to decrypt a cipher text, if the user knows the shift code used for encryption, she just needs to use the same process but the negative of that shift value.
- Whence, we changed the logic of our program completely: We only ask the user as input a shift value. Our program is agnostic of whether it corresponds to an encryption or a decryption phase -this has only a meaning for the user: **the math of the problem is the same in both cases**.
- If the logic is simpler, so can be as well our code. **A simpler code, is a more reliable and easier to troubleshoot code**.

The gist of the Caesar cipher

```
#include<stdio.h>
#include<stdlib.h>

int main ( int argc , char** argv){
        if ( argc < 2 ) return 1; // needs at least 1 argument, the shift code
        char c;
        int edc = atoi(argv[1]) ;
        while ( ( c=getchar() ) != EOF  ) printf("%c",c+edc) ;
        return 0;
}
```

Save this code into a file called `caeser.c`, compile it (`gcc caesar.c -o caesar`) and run it as

```
$ ./caesar edc < plain-text-file > cipher-text-file
$ xxd -p cipher-text-file
```

See the assignment 2 for an explanation of what the output of `xxd -p cipher-text-file` means.

Remarks:

- If a program is short enough, it's likely that it's easier to understand its meaning/what it does. Whence it's likely that we don't need as many comments.
- The structure of our code helps reading it. Put some effort into making it easy and logical to read your text: **your are indeed but writing a mathematical proof for solving a problem (getting a particular output given a particular input)**!
- Here the structure is: First, deal with a minimum of HCI: truly speaking there is no HCI addressed here. The only thing we do in the first line is **not letting the program crash with a cryptic error message, but smoothly, silently end with a non-zero return value**. Run `./caesar` without any arguments and check out the return code by issuing the shell command `echo $?` right after the program finishes. You should see a 1. On the contrary, if the program completes as expected, `echo $?` will print a 0.
- Second, we deal with the two **local variables** that our program need (local means that they are only visible inside the block defining `main`). Notice that his is done as close as possible to the point in the code where we first start using them.
- Finally, as the block of the `while` loop contains only one single statement, we can get rid of the braces and write the whole loop in one single line.

Questions:

1. Which of the previous codes for the Caesar cipher looks simpler to you?
2. Which of the previous codes for the Caesar cipher looks aesthetically more appealing to you?
3. Which of the previous codes for the Caesar cipher would you say is easier to debug/troubleshoot?

Assignment 2

Questions:

1. In Linux/OSX there is a command-line tool called `xxd`. It allows the conversion of file content into hexadecimal and/or binary. Consider the file `xxdexample` with the content:

   ```
   This is a text file to showcase
   what xxd program can do.

   Hope it helps.
   ```

   Issuing in the command line `xxd xxdexample` we get

   ```
   00000000: 5468 6973 2069 7320 6120 7465 7874 2066  This is a text f
   00000010: 696c 6520 746f 2073 686f 7763 6173 6520  ile to showcase
   ```

```
00000020: 0a77 6861 7420 7878 6420 7072 6f67 7261  .what xxd progra
00000030: 6d20 6361 6e20 646f 2e0a 0a48 6f70 6520  m can do...Hope
00000040: 6974 2068 656c 7073 2e0a                 it helps..
```

On the left there is a count of the number of bytes (in each line 16, which in hex is 10). In the middle there is the content of the file in hexadecimal and grouped into pairs of bytes, whence we see only 8 groups per line. Finally, the last column shows the content of the file in ascii.

A slight variation of the previous command yields just the hexadecimal version of the file's content, which we redirect into a file called `xxdexample.hex` using the `>` output-redirection command:

```
$ xxd -p xxdexample > xxdexample.hex
$ cat xxdexample.hex
$ 54686973206973206120746578742066696c6520746f2073686f77636173
$ 65200a7768617420787842070726f6772616d2063616e20646f2e0a0a48
$ 6f70652069742068656c70732e0a
```

We can revert the hexadecimal encoding of a file into its original version (be it ascii or binary) using the following options of `xxd`:

```
$ xxd -r -p xxdexample.hex
$ This is a text file to showcase
$ what xxd program can do.
$
$ Hope it helps.
```

Given the plain text of the above cipher examples, determine the encoding code used in order to get the following cipher text (hexdump version):

```
5b6f707a27707a277b6f6c2774767a7b277b7677277a6c6a796c7b27746c
7a7a686e6c117b6f687b275027517c73707c7a274a686c7a6879276a767c
736b277e79707b6c36796c6a6c707d6c35111151354a35276b707f707b11
```

2. Given the Caesar cipher ver 3, and the following encrypted text, find out the corresponding plain text. The ciphertext, that is, the encrypted text can be downloaded from http://msantos.sdf.org/G10/Term2/ciphertxt. **Hint**: In the shell, `man ascii` provides you with a table of all ascii characters. Among them, you may recognize the ones that are actually printable.

Bibliography

1. The C++ web site
2. Joachim Goll and Mandred Dausmann, *C als erste Programmiersprache*. 8th edition, Springer Verlag, 2014.
3. ...
4. ...