Chapter 3 – Introduction to the C language

The C language was created in the 1970s by Dennis Ritchie and Brian Kernighan, and was instrumental in the development of Unix. The language was made very popular in part thanks to a “tutorial” book by the two designers, simply titled *The C Programming Language*. This book remains relevant to this day, and we will be relying heavily on references to it in this course.

Exercise Search the web for an electronic copy of the K&R in a format that you like (web, epub, pdf...) Warning: we want the second edition from 1988, not the original from 1973. For instance, you can use this version from the Internet Archive: `https://archive.org/details/the-ansi-c-programming-language-by-brian-w.-kernighan-dennis-m.-ritchie.org`

1 Basic Syntax

A program written in the C language is a collection of *functions* which call each other. There must be a function named `main()`, which the operating system (Linux, in our case) is responsible for executing.

Exercise Create a file named `hello.c` and type in the program below. Compile it to executable form with command `gcc -g -Wall -Wextra -Werror -o hello hello.c`, then run it with `./hello`. Ask us questions about anything mysterious. From here on, always use these gcc options to compile your code.

```
#include <stdio.h>

void print_hello() {
    printf("Hello, world\n");
}

int main() {
    print_hello();
    return 0;
}
```

The `printf()` function (provided by library `stdio.h`) is useful to display text on screen but it can also be invoked with several arguments, in which case it will perform data *formatting*. The first argument, the so-called *format string* is searched for percent signs, which will then be interpreted as *format codes*. Each format code describes how one of the subsequent arguments should be displayed. For instance, `printf("%d vs %x vs %c\n",42,42,42)` prints the same value three times: as a number written in decimal, in hexadecimal, and as as an ASCII character. Try it out !

Exercise Type the program below in a file named `formatting.c`, then compile and execute it. Read the docs (K&R §7.2 and/or link below) and ask us questions until you understand what happens. `https://www.gnu.org/software/libc/manual/html_node/Table-of-Output-Conversions.html`

```
#include <stdio.h>

int main() {
    printf("0. %c \n", 'a');
    printf("1. %c \n", 65);
    printf("2. %d \n", 100);
    printf("3. %x \n", 100);
    printf("4. %o \n", 100);
    printf("5. %#x \n", 100);
    printf("6. %#o \n", 100);
    printf("7. %6.2f \n", 3.1416);
    printf("8. %6.2f \n", 31.416);
    printf("9. %E \n", 3.1416);
    printf("10. %*d \n", 5, 10);
    return 0;
}
```

Exercise Write a program that prints the number -1 in hexadecimal notation. What does it show on screen ? Give a one sentence explanation of why this is the case.

2 Variables

A *variable* is a symbolic (i.e. plain-text) name which identifies a memory location. In C, there are *global variables* which are visible from everywhere, and *local variables* which belong to one particular function. Each variable has a fixed *type*:

- `int` for whole numbers, such as $0, 1, 2$ or -42,
- `float` for numbers with a decimal point, like 3.14 or -5.3
- `char` for one-byte integer values (typically, ASCII-encoded characters).

For more info on types and variable declaration, read K&R §2.2 and §2.4.

Variables can be *assigned* a new value using the “equals” sign, e.g. `x = 36;` or `y = x+10;` The job of the compiler is to translate such a statement into machine instructions doing two things: 1) evaluating the expression on the right-hand side, and 2) storing the result in memory at the correct address. Warning: expressions are always evaluated with the same type as their operands. For instance, `5/10` is an integer division and evaluates to zero, but `5.0/10.0` evaluates to `0.5`. Try variations around `printf("%d %g\n",5/10,5./10.);` with different combinations of floats and ints.

Exercise Write a program which initializes two integer variables with positive values, computes their ratio as a `float` and then shows it as a percentage[1] For instance with `int a=3;` and `int b=9;` your program should print something like `3/9=33.33%`

Each function can also send a *return value* to its caller using the “return” keyword, with e.g. `return 42;` or `return x+y;` Functions that return nothing, like `print_hello()` on on the preceding page, must declare their *return type* as being `void` and must use the `return;` keyword with no operand.

---

[1] To find out how to get `printf()` to show a literal percent sign, refer to K&R §7.2

3 Control Structures

Conditional statements use the “if” and “switch” keywords (cf K&R §3.1 to §3.4) as illustrated below:

```
if( condition )
{
  ... // then do something
}
// in both cases,
// execute here afterwards
```

```
if( condition )
{
  ... // then do something
}
else
{
  ... // do something else
}
// in both cases,
// execute here afterwards
```

```
switch( integer )
{
    case 42:
    ...
    break;

    case -5:
    ...
    break;

    default:
    ...
    break;
}
// 'break' jumps here
```

Remarks:

- In C everything is an integer, there are no proper booleans: zero means “false” and anything non-zero means “true”. Even comparison operators like “==” or “<” produce an integer. For this reason, `if(x != 0)` and `if(x)` are equivalent (cf K&R §2.6).
- Within a `switch-case` construct, be sure to always add a `break` statement at the end of each branch, otherwise control will *fall through* to the next branch which is perfectly allowed but rarely what you wanted (cf K&R §3.4).

There are several looping constructs in C, which behave as repeated “if” statements (cf K&R §3.5).

```
while( expr )
{
   body;
}
```

$\sim$

```
label:
if( expr )
{
  body;
  goto label;
}
```

```
for( code1; expr; code2 )
{
   body;
}
```

$\sim$

```
code1;
while( expr )
{
  body;
  code2;
}
```

Remarks (cf K&R §3.7):

- The `continue` keyword skips the remainder of the loop body and jumps directly to the next iteration.
- The `break` keyword exits the current loop (or switch-case) entirely.

4 Functions

For the compiler to generate correct machine instructions when calling a function, it must know the type of every argument (and return value), collectively known as the *function’s signature*. For historical reasons, this information must come *before* (in line number order, in the source file) the place where the function is invoked. If the *definition* of a function is above its first call site, like is the case with `print_hello()` on on page 1, then everything is fine. Otherwise, the programmer must first provide a *declaration*, as illustrated below:

```
#include <stdio.h>

void print_hello() ;

int main() {
    print_hello();
    return 0;
}

void print_hello() {
    printf("Hello, world\n");
}
```

Exercise Open file `/usr/include/stdio.h` and find where `printf()` is declared. (You don't need to understand all the syntax details)

Exercise Implement a recursive function with signature `int fib(int n)` which computes the nth Fibonacci number: fib(0) = 0, fib(1) = 1, otherwise fib(n) = fib($n-1$) + fib($n-2$).
In your `main()` function, invoke `fib()` in a `for` loop to print the first 20 Fibonacci numbers.

Exercise In another source file, implement an iterative (i.e. non-recursive) version of `fib()`, with the same signature. You can reuse your `main()` function to check that both implementations produce the same results.

Exercise Write a program which loops over every number k from 1 to 50 and:

- when k is a multiple of 3, print "`IST`",
- when k is a multiple of 5, print "`OPS`",
- when k is both a multiple of 3 and 5, print both "`IST`" and "`OPS`",
- otherwise just print k.

The expected output is illustrated on the right.

```
1
2
IST
4
OPS
IST
7
8
IST
OPS
11
IST
13
14
IST OPS
16
17
...
```

To find out if `x` is a multiple of `y`, you can use the *modulo* operation (aka remainder of the integer division) with e.g. `if(x%y==0)`

Exercise Implement a function with signature `void tree(int R)`, which draws a pine tree with `R` horizontal rows of "leaves" and a 3x2 centered "trunk", as illustrated below.

R=2

```
 *
***
|||
|||
```

R=6

```
     *
    ***
   *****
  *******
 *********
***********
    |||
    |||
```

R=13

```
            *
           ***
          *****
         *******
        *********
       ***********
      *************
     ***************
    *****************
   *******************
  *********************
 ***********************
*************************
           |||
           |||
```