operating systems lab - week 2: exercise

nicolo colombo

This lab is about C types, variables and functions. You will see in practice how numbers and characters are represented in C, how you can define and call functions, and you will write programs that parse terminal input/output.

Set up

We suggest you edit, save, and compile the programs you write for this lab session in `CS2850Labs/week2` a dedicated sub-folder of the directory you created for the first lab session of the term, on the teaching server, `linux.cim.rhul.ac.uk`. On the course Moodle page you can find more details about connecting to `linux.cim.rhul.ac.uk`and editing and compiling your code from the command-line and debugging your programs using Valgrind.

1 Variables

Similarly to other programming languages, in C you can use variables of different *types* and different *storage classes*. This list of primitive data types contains all variables you can declare and use in C. In this section, you will write a program that prints on screen the size in bytes of the most common C types, i.e.

`char, int, unsigned int, float`

1.1 Integers

Start by declaring and initialising a variable of type `int` and print its value as in the following program

```
#include <stdio.h>                 1
int main() {                       2
  int a = 10;                      3
  printf("a=%d\n", a);             4
}                                  5
```

Copy the code above into a file called, `printInt.c` and compile and run it to check that it prints

```
a=10                               1
```

on screen. You can modify and recompile `printInt.c` as suggested in the following questions:

1. What happens if the variable is declared outside `main`?
2. What happens if you add a non-integer part in the initialisation of `a`, e.g. if you replace Line 3 with `a = 10.1234`?
3. What happens if you initialised `a` with a *very large* value, e.g. if you replace Line 3 with `a = 2147483647` and `a = 2147483648`?

The following code produces the same output as the program above.

```
1 #include <stdio.h>
2 int a = 10;
3 void printValue() {
4   printf("a=%d\n", a);
5 }
6 int main() {
7   printValue();
8 }
```

Copy the new program into a new file, `printInt2.c`, compile it, and run it to check that its output on screen is indeed

```
a=10
```

Modify `printInt2.c` as suggested in the following questions:

1. What happens if the variable is declared *inside* `main`?
2. What happens if you change the value of `a` inside main, e.g. if you add

   ```
   a = 11
   ```

   just before Line 7?
3. What happens if you change the value of `a` inside the definition of `printValue`, e.g. if you add

   ```
   a = 11
   ```

   just before Line 4?

1.2 `unsigned int`, `char`, and `float`

Write a modified version of `printInt.c` called `printTypes.c`, that prints

```
au=2147483648
ac=*
af=0.123456
```

on screen and where `au` is declared as an `unsigned int`, `ac` as a `char`, and `af` as `float`. To obtain the correct output you should also use the correct format identifiers, `%u` for `unsigned int`, `%c` for `char`, and `%f` for `float` in the call of `printf`. Have a look at this list of formatting symbols for more details. What happens if you use the `int` format, `%d`, instead of %`u` when you call `printf` in `printUnsigned.c`? Try also to print the value of the variables as an unsigned octal number, with `%o`, an unsigned hexadecimal number, with `%x`, and a floating-point number in exponential notation, with `%e`. Which conversions are allowed and which lead to a compilation error if the program is compiled using the `-Werror -Wall` flags? Force the conversion by including a type *cast* in the *second argument* of `printf` as in the following example

```
#include <stdio.h>
int a = 1234;
void printValue() {
  printf("a=%e\n", (float) a);
}
int main() {
  printValue();
}
```

1.3 Sizes

The size of a given type can be obtained by calling the operator `sizeof(type)`, e.g.

```
unsigned long int sizeOfChar = sizeof(char);
```

idem with `int`, `unsigned int`, or `float`, or by letting the argument of `sizeof` be a pre-declared variable, e.g.

```
char a;
unsigned long int sizeOfChar = sizeof(a);
```

See Section A7.4.8 of The C Programming Langaugeor Section 3.11 of the GNU Online Manual for more details about the `sizeof` operator. Write a program, `sizeOfTypes.c`, that prints on the terminal the size in bytes of a `char`, an `int`, an `unsigned int`, and a `float`. Your program should print the size of each type on a different line, with each line being of the form

```
the size of a long int is 8 bytes
```

Note that the output of `sizeof` is an `unsigned long int`. What happens if you use `sizeof` to get the memory size associated with an array? Modify your program so that it prints two extra lines reporting the size in bytes of a 10-dimensional array of `char` and `int` declared as

```
int vInt[10];
char vChar[10];
```

1.4 Signed or unsigned `char` (optional)

The conversion of characters to integers depends on whether the compiler treats the variables of type `char` as signed or unsigned quantities. Try to understand if on your system they are signed or unsigned by looking at the error messages produced by `gcc -Wall -Werror -Wpedantic` when you compile a program such as

```
#include <stdio.h>
int main() {
  char a = 150;
  unsigned char b = 150;
  printf("a=%d and b=%d\n", a, b);
}
```

The conversion of a variable of type `int` into type `char` may cause some information to be lost. Copy, compile, and run the following program:

```
#include <stdio.h>
int main() {
  int a = 128;
  char c;
  c = a;
  a = c;
  printf("a=%d\n", a);
}
```

What do you observe? Can you explain why all problems disappear if initialise `a` with the value 127?

2 Terminal input/output: `getchar` and `putchar`

In this section, you will write a program that transforms all lower case letters of an input string into upper case letters. The standard library contains functions for reading or writing one character at a time:

1. `getchar()`, which reads the next input character and returns it, and
2. `putchar(c)`, which prints the character `c` on the terminal.

Read, and try to guess what the following program does

```
#include <stdio.h>
int main() {
  char c;
  while ((c = getchar())!= 'q') {     4
    putchar(c);                       5
  }                                   6
}                                     7
```

Copy the code into a new file called `inputOutput.c`, compile, and run it to understand how it works by typing random character on the screen when the program starts.

2.1 Change the *exit* keyword

When you run the program in `inputOutput.c`, the terminal shows a new empty line where you can type your text. The program execution is paused until you send a newline character, `\n`. Once all characters in the input have been processed the program stops again, waiting for more input. For exiting, you need to send an exit keyword that makes the `while`-loop condition false. Try to modify the program above so that:

- the program exits when you type on the space bar
- the program exits when you send a newline character (`return`)
- the program exits when you type `ctrl-d`

The `ctrl-d` combination is a terminal shortcut for sending an *end of file* signal. In C, the end-of-file signal is represented by an `int`, called `EOF`, and quite often equal to -1, a value that is not taken by any *valid* `char`. Add a few lines to your code to check that `EOF` $= -1$ on your machine. In principle, you should be careful with comparing variables of type `char` to `EOF`, as the latter is defined as an `int`. We suggest you keep this in mind and have a look at Section 1.5.1 of The C Programming Langaugefor a discussion about `EOF` and `getchar()`. The easiest solution is to declare the variable used to store the output of `getchar()` as an `int`, i.e. to replace Line 3 with

```
int c;
```

2.2 Lower and upper cases

In the ASCII characters encoding, upper-case letters are ordered alphabetically from `A` to `Z` and followed by all lower-case letters, which are also ordered alphabetically from `a` to `z`, i.e.

$$\cdots,\quad \texttt{A},\quad \texttt{B},\quad \cdots,\texttt{Z},\quad \texttt{a},\quad \texttt{b},\quad \cdots,\quad \texttt{z},\quad \cdots$$

This fact can be exploited for converting upper-case letters into lower-case letters and *vice versa*. The size of the alphabet can also be computed by subtracting the value associated with `A` to the value associated `a`, e.g. through

```
int sizeOfAlphabet;
sizeOfAlphabet = 'a' - 'A';
```

Write a function, `int upper(int c){ ... }`, that checks if the input character, `c`, is a lower case letter and, in that case, transforms it into the corresponding upper case letter. `upper` can be a modified version of

```
int lower(int c) {
  if (c >= 'A' && c <= 'Z')
    return c + 'a' - 'A';
  else
    return c;
}
```

To see the effect of `lower`, replace `putchar(c)` with `putchar(lower(c))` in `inputOuput.c`. Finally, set the exit keyword of `inputOutput.c` to `EOF` and recompile it. Copy the following text

```
one two three
four five
six
```

into a file called **`someText.txt`** ans observe what happens when you run

```
./a.out < someText.txt
```