System-oriented Programming
Spring 2018

S02

Professor : Philippe Cudré-Mauroux
Assistant : Michael Luggen

Submitted by Sylvain Julmy

Exercice 4

- `./wcount`

  This command would just invoke the program `./wcount`, which would read the standard input (keyboard) in order to obtain something to process. We just have to type something, including <ENTER> char and then we send the EOF char by <CTRL+D>. The ouput is printed on the standard output.

- `./wcount < wcount.c`

  This command invoke the program `./wcount`, but this time the standard input of the program is the file named `wcount`. The ouput is printed on the standard output.

- `./wcount < wcount > test`

  This command invoke the program `./wcount`, but this time the standard input of the program is the file named `wcount.c`. The ouput is redirected to a file named `test`.

- `cat wcount.c | ./wcount`

  The `cat` command will print the content of the file `wcount.c` on the standard output, but using the `|` operator, the standard output is redirected into the standard input of the `./wcount` program.

- `grep { wcount.c`

  The `grep` program would filter and print the content of the file `wcount.c`. This command would print all the line of `wcount.c` that contains a `{` symbol.

- `grep { wcount.c | ./wcount`

  It is the same as before except that the standard output of the `grep` command is redirected to the standard input of the `./wcount` command. So the whole command would count the number of word, letter, ... of all the line of `wcount.c` that contains a `{` symbol.

- `grep -l { * | ./wcount`

  The `-l` option of `grep` is suppressing the normal output of the command and instead print the name of all the file from which the output would normally be printed. The `*` symbol represent all the file of the current directory. So the `grep` command would output all the filename of the current directory that contains the `{` symbol.

Exercice 5

- `printf("%c %i\n", c, c);`

  We don't need any type casting here, `c` is a variable of type `char` which is automaticaly seen as an integer when using the `%i` formatter. 65 is the ascii value of "A".

  Output : `A 65`

- `printf("%c %i\n", i, i);`

  We don't need any type casting here, `i` is a variable of type `int` which is automaticaly downgraded into a `char` by simply "cutting" the extra-part of the `integer` and put it into a `char`.

  Output : `A 65`

- `printf("%f %i\n", pi, (int)pi);`

  This time we need an explicit type casting from a `float` into an `integer` because the representation, in memory, of floating point number and integer number is different and the compiler has to to some additionnal work in order to correctly transform the float value (by rounding the value from 3.14 to 3) into an integer. Without the explicit type casting, the number would have been read directly (the significand, the base and the exponant would'nt have been read separetely).

  Output : `3.140000 3`

Exercice 6

First, the decimal value of the ascii character `@` is 64. The output of the program is the following

```
@ 64 100 40
@ 64 100 40
@ 64 100 40
```

Three times the line printed is the same because :

- the decimal value of `@` is 64,
- the decimal value of `\100` is 64,
- the decimal value of `\x40` is 64.

Exercice 7

The output of the program is the following :

```
1 0
0 1 2
```

In C, each enum fields is represented by an integer value from 0 to $n-1$ where n is the number of fields in the enum. Then, `TRUE` as the value 1 and `FALSE` as the value 0, that's why the first line is `1 0`.
For the second line, C_1 is initialize to `RED`, which is the first field of the `color_tag` enum, so the value of `RED` is 0. C_2 is initialize to $C_1 + 1$ which is $0 + 1 = 1$. Finally C_3 is initialize to `BLUE` which is the third field of the `color_tag` enum, so the value of `BLUE` is 3. Then outputing C_1, C_2 and C_3 would be `0 1 2`.

Exercice 8

We denote by n any integer not equal to 0.

`p || !q`

p	q	p	||	!	q
n	n	n	1	0	n
n	0	n	1	1	0
0	n	0	0	0	n
0	0	0	1	1	0

`p && (p == q)`

p	q	p	&&	(	p	==	q	)
n	n	n	1	n	1	n
n	0	n	0	n	0	0
0	n	0	0	0	0	n
0	0	0	0	0	1	0

`p && (p = q) || (p = !q)`

First we add parentheses to clearly show the order of evaluation :
`( p && (p = q) ) || (p = !q)`

p	q	(	p	&&	(	p	=	q	))	||	(	p	=	!	q	)
n	n	n	1	n	n	n	1	n	0	0	n
n	0	n	0	n	0	0	1	0	1	1	0
0	n	0	0	0	n	n	0	n	0	0	n
0	0	0	0	0	0	0	1	0	1	1	0

Finally, we verify the answer using the C program available in listing 1.

Exercice 9

a)

The output of the code would be 2. The ternary operator could be seen as an expression of the following form :

```
expr_t ? expr_1 : expr_2
```

- `expr_t` is an expression evaluated as a boolean expression to determine which of the `expr_1` or `expr_2` to use as an expression in the program.

```c
#include <stdio.h>

// @expr should contains variable with identifier p and q
#define truth_table_2(expr)\
    for(int i = 1; i >= 0; i--)\
    for(int j = 1; j >= 0; j--)\
    {\
        p = i;\
        q = j;\
        printf("%i\n",expr);\
    }

void main(void)
{
    int p;
    int q;

    truth_table_2(p || !q);
    printf("+---------------------------------+\n");
    truth_table_2(p && (p == q));
    printf("+---------------------------------+\n");
    truth_table_2(p && (p = q) || (p = !q));
}
```

Listing 1: C macro that print the result columns of the truth table for 2 variables

- `expr_1` is the expression "returned" if `expr_t` is evaluated to *true* (different from 0).
- `expr_2` is the expression "returned" if `expr_t` is evaluated to $false$ (equals to 0).

b)

The ternary operator is usefull for its semantics, instead of the if/else construction which is here to determine which statement to execute or not, the ternary construction is an expression. Which means that we can use it inside of another expression (as seen in Figure 5 from the exercice sheet).
Ternary operator are very usefull to express simple conditional expression like simulating the `max(a,b)` function. By the way, if we need to do more complicated stuff like statement or nested if/else construction, using the standard if/else is clearly better.

Exercice 10

a)

We use unsigned integer of $n = 8$ bits for the proof. The proof would be still valid for any value of n where $n > 1$. The case where the left most bit is not 0 is discussed in part **b)** below.

Left shift : any integer is encoded in binary and we use a sequence of 1 and 0 to encode it : $x = (x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0)$. A left shifting of x, denote $left$, would transform x into x' :

$$x' = (x_6, x_5, x_4, x_3, x_2, x_1, x_0, 0)$$

In order to compute the value of x and x', we use the following formula :

$$value(x) = x_0 \cdot 2^0 + x_1 \cdot +2^1 + \cdots + x_7 \cdot 2^7$$
$$value(x') = 0 + x_0 \cdot 2^1 + x_1 \cdot +2^2 + \cdots + x_6 \cdot 2^7$$

Then we have two cases :

- $x_i = 0$: if a 0 is left shifted, then the value of x_i in x' don't change.
- $x_i = 1$: if a 1 is left shifted, then the value of x_i in x' is two times greater that in x.

Now each 1 in x are doubling the value in x' so that's why a left shift multiply the value of x by 2.

Example : $x = 7$, we have $x = (0, 0, 0, 0, 0, 1, 1, 1)$. Then we compute x' :

$$x' = (0, 0, 0, 0, 1, 1, 1, 0) = 14_{10} = 2 \cdot 7$$

Right shift It is the same as the left shift, except that there is no special case if the left most or right most bits are 1 or 0.
The value of x_i in x' would be $x_i \cdot 2^{i-1}$, which explain the 2 time less value.

b)

Now we assume that x is of the following form

$$x = (1, x_6, x_5, x_4, x_3, x_2, x_1, x_0)$$

In **a)** we saw that x_7 is not consider in the computation of x', so the value of $_7$ is ignored when left shifting the variable on the left.
If we consider a world when left shifting a variable of n bits would create a variable of $n + 1$ bits. If the left most bit is 0, there is no problem, but if the left most bit is 1, then x' would have the value

$$value(x') = 0 + x_0 \cdot 2^1 + x_1 \cdot 2^2 + \cdots + x_{n-2} \cdot 2^{n-1} + 2^n$$

The value of x' is increase by 2^n more than expected, but in arithmetic modulus 2^n, the value of x' would be

$$value(x') = 0 + x_0 \cdot 2^1 + x_1 \cdot 2^2 + \cdots + x_{n-2} \cdot 2^{n-1} + 0$$

because $2^n \equiv 0 \mod 2^n$.
So because of the modulus 2^n, the left shifting of a variable where x_{n-1} is 1 is not true.

c)

We use the program from listing 2.
The implementation is quiet simple, we just create a mask with the specified bit set to 1 and operate a bitwise and between x and the mask.

```c
#include <assert.h>
#include <stdio.h>

unsigned long setbit(unsigned long x, int n);

#define t_setbit(x,n,r) assert(setbit((unsigned long)(x),n) == (unsigned long)(r));

void main(void)
{
    t_setbit(8,0,9)
    t_setbit(100,0,101)
    t_setbit(97,0,97)
    t_setbit(7,3,15)
}

unsigned long setbit(unsigned long x, int n)
{
    return x | (((unsigned long) 1) << n);
}
```

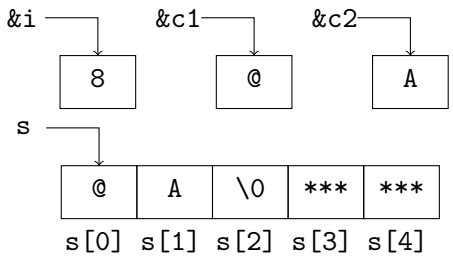
Listing 2: Implementation of `long setbit(unsigned long x, int n);` that returns x with the nth bit set to 1.

Exercice 11

Firstable, we suppose that the function f has the signature `int f()` because if f would have a number of argument greater that 0, then the statement $f()$ would produce an error and calling a function with too many argument is not an error.
The execution of the code can produce different results with different compilers because the order of evalution is not defined in C. One compiler could execute $f()$ before $g()$ and another could do the inverse.
The problem come from side effect that f and g could produce if, for example, global variable are accessed in f and g and they are modified, then the result of executing one function before another is strictly depending on the compiler.

Exercice 12

a)



b)

Variable		Address	
Name	Value	Name	Possible value
i	8	&i	0x00
c1	@	&c1	0x20
c2	A	&c2	0x40
	@A	s	0x60

c)

Address	**Big endian**				**Little endian**				**Address**
&i	0	0	0	8	0	0	0	8	&i
&c1	@	***	***	***	***	***	***	@	&c1
&c2	A	***	***	***	***	***	***	A	&c2
s	@	A	\0	***	***	\0	A	@	s
s+4	***	***	***	***	***	***	***	***	s+4

d)

Address	**Big endian**				**Little endian**				**Address**
0x0	0	0	0	8	0	0	0	8	0x0
0x4	0x40	***	***	***	***	***	***	0x40	0x4
0x8	0x41	***	***	***	***	***	***	0x41	0x8
0xc	0x40	0x41	***	***	***	***	0x41	0x40	0xc
0x10	***	***	***	***	***	***	***	***	0x10

Exercice 13

a)

We use the man page of *scanf* to figure out what is happening in the code and specially the following two paragraphs.

```
RETURN VALUE
       On  success,  these functions return the number of input items success-
       fully matched and assigned; this can be fewer  than  provided  for,  or
       even zero, in the event of an early matching failure.

       The  value EOF is returned if the end of input is reached before either
       the first successful conversion or a matching failure occurs.   EOF  is
       also returned if a read error occurs, in which case the error indicator
       for the stream (see ferror(3)) is set, and errno is set to indicate the
       error.

The  format  string consists of a sequence of directives which describe
       how to process the sequence of input characters.  If  processing  of  a
       directive  fails,  no  further  input  is read, and scanf() returns.  A
       "failure" can be either of the following: input failure,  meaning  that
       input  characters  were  unavailable, or matching failure, meaning that
       the input was inappropriate (see below).
```

The first line is correctly parsed and the values after the execution of the first line are the following :

- $str =$ "AA"
- $i = 33$
- $d = 2$

The second is not correctly parsed, the computer can't parse `ZZ` into an integer and it will stop to pull characters from the standard input. So 55 is parsed as a string and only 1 argument is parsed by the $scanf$ function. So we will have the following variables and value after the second line :

- $str =$ "55"
- $i = 33$
- $d = 1$

So "ZZ" remains in the standard input and that's why, when we are executing the third line, the parsing is computed correctly and "ZZ" and 77 are correctly parsed and put into str and i. So the variables and their value are :

- $str =$ "ZZ"
- $i = 77$
- $d = 2$

We note that 99 remains in the queue of the standard input.

b)

We use the following code in order to parse a character, an integer, a string and a float.

```c
#include <stdio.h>
#include <assert.h>

void main(void)
{
    char c;
    int i;
    char string[10];
    float f;

    // parse the char, integer, string and the float in one line
    int res = scanf("%c %d %s %f", &c, &i, string, &f);
    assert(res == 4);

    printf("char value : %c\n"
           "integer value : %i\n"
           "string value : %s\n"
           "float value : %f\n",
           c, i, string, f);
}
```

We can try our code on our machine :

```
c 123 Hello 3.4
char value : c
integer value : 123
string value : Hello
float value : 3.400000
```

Which seems to correctly parse a correct input.