SOP - Alain Schaller - 16-896-375

2. Word Count - Some Shell Commands

```
$ ./wcount
```

Starts the `wcount` program, which will indefinitely, read from the `stdin` and increment different integers to count the number of lines, words and characters.

The only way to stop the program we have to send an `EOF` which can be send using the `CTRL+D` key combination.

```
$ ./wcount < wcount.c
```

Starts the `wcount` program, however, this time, it passes the content of the `wcount.c` file as the `stdin`. Therefore, the program will output its different counters.

This time, no need to send and `EOF` as the file has a `EOF` at its end.

```
$ ./wcount < wcount > test
```

Same as the previous execution, however, we redirect the `stdout` of what the program prints in a file named `test` (re-write the entier content of the file with the output).

```
$ cat wcount.c | ./wcount
```

`cat` will read the argument's content, `wcount.c`, then the `stdout` is redirected as the `stdin` of the `wcount` prgram.

```
$ grep { wcount.c
```

`grep` reads the file argument and test in the same way as `RegExp` each line. Here, it will filter the line with an opening curly bracket and only print those.

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

Same as before but we pipe the `stdout` of the execution of `grep` into the program `wcount`. Therefore, it will count only the number of line with an opening curly brackets, the number of words and charcter of those lines.

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

The `-l` option of the `grep` will only print the name of the passed files and exit its program with a success or fail code if it matches the passed pattern.

The `*` symbol, will simply match all the files in the working directory. Therefore, in all the files in the same directory, look which contains an opening curly bracket and pipe those file names to the `wcount` program, which actually will count the number of files with the count of lines.

3. Type Conversion, Casting and ASCII

```
char c = 'A';
int i = 65;
float pi = 3.14;
```

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

The `%c` conversion specifier will first print the character `A` and the `%i` as a signed integer.

Therefore, it will print: `A 65`, since the `A` character is the 65 character in the ASCII table.

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

It will print: `A 65`, `A` is the 65-th character in the ASCII table.

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

The `%f` conversion specifier will first print a floating-point number. Here, no precision is defined, therefore, it will use the default precision of **6**.

Therefore, it will print: `3.140000 3`.

4. Constant, Variable, Escape Character ‘ ’, and Octal resp. Hexadecimal Digits

```
// macro definition in octal
#define AT '\100'
// variable in hexadecimal
char at = '\x40';

printf("%c %i %o %x\n", '@', '@', '@', '@');
```

The `%c`, `%i` have been seen in the previous exercise, for the `%o` and `%x` conversion specifier will respectively print *unsigned integers into their* ***octal*** *representation* and *unsigned integers into their* ***hexadecimal*** *representation.*

Therefore, it will print: `@ 64 100 40`.

```
printf("%c %i %o %x\n", AT, AT, AT, AT);
```

It will print the same, as the `AT` macro is in octal format and its decimal representation is 64, which is the same as the 64-th character of the ASCII table, the `@` character as seen in the previous line.

```
printf("%c %i %o %x\n", at, at, at, at);
```

Finally, it will again, print the same line, as `x40` in decimal is 64.

5. enum Type

```
// declaration of variable b, without tag
enum {FALSE, TRUE} b;
b = TRUE;
printf("%i %i\n", b, FALSE);
```

When integer values aren't specified in the enum declaration, it will start at 0 and increment by 1 for each new value declared.

Therefore, `b` which is `TRUE` will have a integer value of `1` as declared after `FALSE` which is `0`.

It will print: `1 0`.

```
// enum declaration, with tag 'color_tag'
enum color_tag {RED, GREEN, BLUE};

// declaration of variables ci
enum color_tag c1;
enum color_tag c2;
enum color_tag c3;

c1 = RED;
c2 = c1+1;
c3 = BLUE;
printf("%i %i %i\n", c1, c2, c3);
```

`RED` is equal to `0`, `GREEN` is equal to `1`, `BLUE` is equal to `2`, therefore, it will print the following signed integers: `0 1 2` as `0+1` for `c2`.

6. Logical Expressions

```
p || !q;
```

p	q	Result
1	*	1
0	0	1
0	1	0

```
p && (p == q);
// Beware: p == q
```

Can be simplified by `p && q`.

p	q	Result
0	*	0
1	0	0
1	1	1

```
p && (p = q) || (p = !q);
// Beware: p = q
```

p	q	Result
0	*	0
1	0	1
1	0	1

If the `p` is `true`, set the value of `q` to `p`, if `q` is `false`, then, it will set `!q` to `p`.

7. scanf

```
char str[3];
int i;
int d;
d = scanf("%s %i", str, &i);
d = scanf("%s %i", str, &i);
d = scanf("%s %i", str, &i);
```

`scanf` return documentation:

> *number of characters transmitted to the output stream or negative value if an output error or an encoding error (for string and character conversion specifiers) occurred*

When running the programm and writing the following lines in the `stdin`:

```
hello 42
WoRlD -42
SOP 0
```

It will read the first 3 characters and copy them in the `char*` pointer and the second's *word* and read it as an *signed integer*.

Ligne	str	i	d
1	"hel"	42	2
2	"WoR"	-42	2
3	"SOP"	0	2

`d` being equal to `2` as it could pick-up 2 "symbols" each time, the string of 3 character and an integer.

8. Order of Evaluation

```
  f() + g();
2 printf("%i\n", ++n, f(n));
```

Different compiler and their arguments/options will produce different program as each are specific to their system and different ways to optimize the execution.

Therefore, maybe, possible differences would be:

- "Skipping" the execution the first line `f() + g()` as the result isn't used
- Or display different warnings for the usage of the `printf` method, as the number of arguments, don't match the number of symbols in the format argument.
- The usage of the `f` method differs also, with a variable count of arguments, therefore, the compiler, might do some assumption on the behavior of the code and produce differences.