Master Calcul Scientifique - Mise à niveau en Informatique
Practical work exam

The aim of this exercise is to implement a "pretty printer" filter that formats a file containing a C program. We will only consider a simple version (see below). Then, we will create a script to run this command and add some options to it.

Remember that I will have to read your code, so do not hesitate to use understable names for variables and constants, to use structures and synonyms (via `enum`, `typedef`...), and most importantly, to comment your code.

1 Implementation of the program

1.1 Function requirements

The pretty printer will be encoded in a file `pp.c`. the name of the executable will be `pp`. It will do the following:

Indentation. Each brace will be put at a line by themselves; an opening brace will be indented by 2 spaces, except when opening a function definition, where it is not indented; a closing brace will be indented to the same level than the corresponding opening brace. In any case, the contained code is indented by 2 spaces from the braces.
Each line should begin at his current indentation level (be caution when reading space or tabulations - character `'\t'`). This is modified by braces, but also by control structures that contain a single instruction. In particular, this instruction should be on a separated line. Also, be caution with the `switch` command (instructions after a `case` do not need to be between braces but have to be indented).

Braces used to initialize tables or structures. Braces used to initialize a table or a structure (struct, enum...) at a declaration of a variable should be the exception to the indentation rule: they should not be on a single line but all the value should be kept on the same line than the variable and its type.

A single instruction by line. After the end of an instruction, we skip a line if it was not the case before (each semicolon at the end of an instruction or a declaration have to be the last character of the line).

Comments. Any comment will be put at a beginning of a new line. There must be only one comment per line. If the end of a line (character `'\n'`) appears in a comment, we close the current comment and start a new one on the second line. If one encounters a C++ comment `//`, it has to be transformed into a proper C comment. Finally, when a comment is in the middle of a line, the line should not be divided into two parts: the comment will take place at the next line, and the current line will stay the same (except if another rule makes it change), with the comment removed.

Preprocessor instructions. When a `#` is the first non space element of a line (we do not take into account spaces or tabulations here), it should be put at the beginning of the line. No new line should be created when dealing with inside a preprocessor instruction (with the exception of the comments removed).

Spaces. Indentation excepted, every sequence of several spaces and tabulations should be replaced by a single space. Also, if there are two or more empty lines in a row (a line containing only spaces and tabulations is considered as empty),they should be replaced by a single empty line.

What we do not want to change

- Nothing should been changed inside a string or a comment.
- Any "special" character in a string or a comment should not be taken into account by the program.
- Also, a special character with an anti-slash `\` before is considered as a normal character (for instance, `printf("This is one \"\n");` will print `This is one "` on the screen). Therefore, this should not affect the variables (as the indentation one) of the program.
- Simple quotes `'` transform character into their value in the ASCII table. Therefore, special characters inside such quotes should not influence the variables of the program.

Errors. In case where there is a problem in the source file (for instance, an opened brace is never closed), the program should exit with the `exit(EXIT_FAILURE)`. In such case, one should print on the standard output a description of the problem (for instance, "3 braces have been opened without being closed" or "There is a closing bracket without any opening one before").

How the executable should work. We want to create a filter. This means we are only working on the standard input and output (`stdin` and `stdout`). For instance, one way to use the executable would be:

```
$ ./pp <pp.c >pp2.c
```

1.2 Examples

Example 1

```
      #include    <stdio.h>
typedef struct   two_int_s {int a; int b;} two_int;
              two_int c={3,2}; int main()


{printf("The couple    of integers is %d  %d\n",c.a,c.b);
return 0;









}
```

should become

```
#include <stdio.h>
typedef struct two_int_s
{
  int a;
  int b;
}
```

```
  two_int;
two_int c={3,2};
int main()

{
  printf("The couple    of integers is %d  %d\n",c.a,c.b);
  return 0;

}
```

One can note that the `two_int;` after the brace is indented as it is related to typedef.

Example 2

```
#include    <stdio.h> /* a big
comment */
# define N 5 // a bad one
int main(){printf("This is not a /*comment*/\n");
printf("\"3+N=\"   %d\n",3+N)      ;return 0;}
```

will be transformed as:

```
#include <stdio.h>
/* a big*/
/*comment */
# define N 5
/* a bad one*/
int main()
{
  printf("This is not a /*comment*/\n");
  printf("\"3+N=\"   %d\n",3+N) ;
  return 0;
}
```

Last point. A thing you may test with your program:

```
$ make pp
$ pp < pp.c > pp2.c
$ gcc -o pp2 pp2.c
$ pp2 < pp2.c > pp3.c
$ diff pp2.c pp3.c
```

Here, the diff function should output nothing (which means that the files are the same).
I will also provide on the webpage some examples of files to try your command on.

2 A script to use this function

This section gives the instructions to make a script that will use our command. This script will be called `pp.sh`. We will also create a script `clean.sh` that will enable to remove all the created files.

2.1 Instructions

For all the parameters given to the script, if the parameter has `.c` or `.h` as a suffix, one will apply the `pp` filter on it to create, if the script has been successful, a new file with the same name, but with a `_pp` before the extension (we create a file `example_pp.c` from a file `example.c` for instance). If there was an error running the command, some error message should be printed.
The script will accept two options:

- `-f` instead of creating a new file, the transformation are made directly on the input file.
- `-r` for every directory given as a parameter, we apply recursively the function to all the files and directories inside it.
- `-rf` will use both options.

Options will only be taken into account if they are given before any others parameters, but it must be possible to use the two options together (not only with `-rf`, but also via `-r -f`).
Also, if no other parameter than the options are given, the script should ask for some input via the `read` command.

Finally, we will create a script `clean.sh` that, given a list of files and directory, will remove all the files ending by `_pp.c` and `_pp.h` in the given files, and also, if the option `-r` is given, inside the given directories, and all their subdirectories (recursively).

2.2 Examples

```
$ ./pp.sh pp.c
```

will create a file `pp_pp.c` that manages the requirements of section 1.1 if the call to the command `pp` has been successful.

```
$ ./pp.sh -r -f ./
```

will format all the `.c` and `.h` files contained in the current directory and all its subdirectories.

3 Makefile requirement

A makefile is required in this exercise. It will manage the compilation, but also will enable to run the script (and therefore construct the executable in the same call if it does not exist).
You are free to write your makefile as you wish, as long as the following targets are available:

- `pp` will create the executable (called pp),
- `run` will run the script without any parameter (therefore the script will use the `read` command),
- `runR` will run the script with the option `-r` but no other parameter.
- `mrproper` will remove all the files created by the compilation process, and will also run the script `clean.sh` recursively on the current directory.

Finally, a macro that gives the compilation options is required (you can of course use more than one if you feel the need).

4 Some advices

Going step by step. The function requirements can be a little tough to manage at once. Therefore, you should manage the requirements step by step, and make sure that you current requirements are well dealt before moving to the next one. Also, when dealing with a specific requirement, you can make some assumptions first, and get rid of them later on. For instance:

- One can first make no exceptions when dealing with braces.
- To manage the string, one can first assume that there is no `"` inside a string.
- One can first assume that there is no comment in the middle of a line.
- One can then assume that if there is such comment, it happens only once by line. Also, we can assume that such comment will not be longer than a constant number of characters (of your choice).

To conclude, if you are not able to deal with all the requirements, it is no big deal. In particular, the rule for comments in the middle of a line should be considered as a bonus more than something mandatory.

Automaton. A good way to program the pretty printer is to use an automaton, by using the `switch` command. Here is one example of automaton, that removes spaces and tabulations at the beginning of each line.

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

int main()
{
  int c;
  enum where_am_i {BEGGINING_LINE, MIDDLE_LINE};
  enum where_am_i w = BEGGINING_LINE;
  while ((c=getchar()) != EOF) {
    switch (w) {
    case BEGGINING_LINE:
      switch (c) {
      case ' ':
      case '\t':
        break;
      default:
        putchar(c);
        w = MIDDLE_LINE;
        break;
      }
      break;
    case MIDDLE_LINE:
      switch (c) {
      case '\n':
        putchar('\n');
        w=BEGGINING_LINE;
        break;
      default :
        putchar(c);
        break;
```

```
      }
    }
  }
  exit(EXIT_SUCCESS);
}
```

About the script. To manage the recursive option, here is an example of script that prints the files given as an input, and also all the files contained in the given directories and all their subdirectories.

```
#!/bin/bash

myls()
{
    local i
    for i in "$@"
    do
if [ -d "$i" ]
then
    echo "Entering in directory $i"
    cd "$i"
    myls *
    echo "Leaving directory $i"
    cd ..
else
    echo "$i"
fi
    done
}

if [ $# -eq 0 ]
then
    read A
    myls $A
else
    myls "$@"
fi
```

Also, remember that if you move to another directory during the process, a call like `pp` or `./pp` will not work !

5 Sending your files

I want a unique archive named `name-firstname.tar.gz`, that will contain a directory `name` containing all the files. In this directory, I expect to find the following files:

- `pp.c` containing the C code. If you want or feel the need to divide your code in several files, then the main function should be in `pp.c`.
- `makefile` the makefile.
- `pp.sh` and `clean.sh` the two scripts.