IGT test:

The exercise's description is written in English, so I'm delivering the answers in English.

The programs will be compiled using libc-2.28 and gcc-8.3.0.

All three exercises contain a Makefile to compile the code.

I didn't have much diffilculty when solving exercises 1 and 3.

Exercise 2, a bit longer, was a bit more difficult. My first aproach was printing the lines as they are read, it had a catch. If the line is too big it must be broken down, so it is possible to reach the columns limit when reading a line. So, the header and the extra new line may be inserted when printing a page or when splitting a line, making the program somewhat more complex.

So I tried a second aproach, this time a small memory module was written. A whole page is read from an input file and saved into memory. When the page reach "nRows" lines it is dump to the screen. Formatting gets more simple this way. That is the delivered solution.

Exercise 1:

Total time to solve the exercise: 4 h.

It is found under the "adder" directory.

There are two programs in this solution:

"alan.c" - Is the program than add asserts to Alan's program and makes it run.

"adder.c" - Exams the formula to compute the sum of the first n natural numbers.

The sum of the first n natural numbers is:

sn = 1 + 2 + ... + n = (1 + n) * n / 2

so, for n = 5:

s5 = 1 + 2 + 3 + 4 + 5 = 15

or:

s5 = (1 + 5) * 5 / 2 = 6 * 5 / 2 = 15

Issues:

We find two issues with the formula.

The first one arises because n / 2 is not an integer if n is odd. It can be solved by dividing by 2 at the end, like:

sn = ((1 + n) * n) / 2.

Note the use of parentheses.

The second one is a possible overflow. We should check for an overflow when computing ((1 + n) * n), so:

((1 + n) * n) <= INT_MAX

$$n^2 + n - INT\_MAX <= 0$$

and we get:

n = (-1 + sqrt (1 + 4 * INT_MAX)) / 2

that we can compute using floats.

The formula can be used to take care of overflows.

Issues found:

Most of the issues when using C arrays are memory related. If we try to write passed an array limit, we will most likely get a "segmentation fault" from the operating system and our program will be killed. Buffer overruns are another common pitfall when using arrays. We can just look for thoses issues. Arithmetic errors must be checked, too.

In first place, the length of the array must match the one assumed by "dummy_adder". So, we make sure the iteration on this function don't go further away than the array length.

The lower limit of that iteration must be checked, we shall get a wrong result if we do not. For this example the assertion gets trivial, but in more complex functions, with computed iteration limits, it may not.

For the iteration to work "s5 [0]" must be properly initialized.

Finally the subindex for s5 [4] must be checked to return the correct amount.

If we get the result with and array of 5 elements with the sum in every position we can skip the following asserts:

```
assert (s5 [0] == 1);
assert (LAST == LENGTH - 1);
```

Exercise 2:

Total time to solve the exercise: 12 h.

It is found under the "pager" directory.

There are two programs in this solution:

"checkPager.sh" - Bash script to check the pager.
"mPager.c" - The pager.

The "mPager" is, in turn, divided into three modules:

"mPager.c" - As the main program.
"memory.c" - For memory management,
"format.c" - To print to the screen.

The exercise description points to an "extra feature", we assume it is compulsary.

The description doesn't mention any kind of user input. It is only enabled when the input file is NOT stdin.

We need to accept an arbitrary number of file names from the command line. ANSI C program arguments,"argc" and "argv" can do it in a natural fashion for us.

We must read every file line by line. We can use C standard libraries for that task. We could use "gets" wich is portable but not so safe. We are using GNU C library "getline" instead. Since the exercise compels us to use gcc, we might reasonable assume that GNU C library will also be available, so "getline" will be allowed for us.

The exercise asks to concatenate one text file after the other and show it to the screen in 80 line pages. We assume a length of 80 lines of text, as they are read from the file, without counting the one at the begining showing a page number, or the one at the end to allow user input.

Here is a brief description for the "mPager":

First we step over every file on the function "pager", sending a stream pointer to "prnPage".

In this latter function we step over every line saving it to a queue. The memory queue module "module.c" does this. If the line length is greater than "nCols" it is broken into smaller blocks with "nCols" length, at the "splitLines" function. When the page queue gets full (with at least "nRows" lines), it is formatted and printed to the screen, the module "format.c" takes care of it. Once all contents is printed the queue is reset.

After printing a page, a footer invites the user to continue by hitting "Intro" key, but only if the input file is NOT "sdtin".

Improvements:

1.- We are using pointer arithmetics to break the lines in blocks. So we would use a memory debugger (like valgrind) to make sure we don't get buffer overruns or other memory issues.
2.- We could write some more tests.
3.- The name of the file could be printed.
4.- The number of lines and total bytes for every file could be printed.
5.- The Linux terminal could be set to non-canonical mode to get keyboard user input without having to resort to "gethar ()", or we could use ncurses.
6.- We could allow the user navigate through the listed files with arrow keys and the like.

To implement those improvements we could:

1.- Just run the program under valgrind with the makefile.
2.- We must add new text files and run the program against them. Those

files may have specific number of lines and line lengths to check the program behaviour. We can use "gcov" to ensure the desired code coverage.
3.- It can easily be printed at "prnHeader" function.
4.- Like above.
5.- Since reading a page and printing it is done on different modules we could modify the formating module indepently. All ncurses related code could be added to the format module, making the program easier to modify.
6.- Like above.

Exercise 3:

Total time to solve the exercise: 4 h.

It is found under the "dows" directory.

There is two *.c files in this solution:

"dows.c" - Is a verbatim copy of the exercise listing.
"check.c" - Is a program that checks the corrected version of "get_dow" and the modified version that tries to improve it.

Please realize that the warnings and errors thrown by the compiler are just intentional. They are left there to make the issues apparent. The program "check.c" sould compile and run without issues.

The routine "get_dow" searches the string array "* dows []" through an, index, "dow". That index is coded as a binary number. It maps the index to the string array the following way:

1 = 0000001 -> "Sun-"
2 = 0000010 -> "Mon-"
4 = 0000100 -> "Tue-"
8 = 0001000 -> "Wed-"
16 = 0010000 -> "Thu-"
32 = 0100000 -> "Fri-"
64 = 1000000 -> "sat-"

If we add some of those binary numbers and call "get_dow", we get a string with all the days of the week corresponding to the terms of the sum concatenated on it, thus:

0000010 + 0000100 + 0001000 = 0001110 = 2 + 4 + 8 = 14

returns the string: "Mon-Tue-Wed"

Issues found:

Sintactic errors can easily be checked by compiling the code. Recall that the file is intetionally compiled with those errors and warnings to make them obvious.

"dow_to_string" incorrectly calls "get_dow". It must be called with TWO parameters instead of only one, as we read on the code.
The compiler will issue an error.
To solve it we must pass the required parameters.

"dow_to_string" returns a pointer to a local variable.
That is incorrect because local variables will be destroyed as soon as the

function leaves its scope.
The compiler will issue a warning.
To solve it we can pas "dowt" as a parameter or dynamically allocate it. If we choose to pass "dowt" as parameter it is mostly the same than calling "get_dow" directly, so we can eliminate it.

Finally, the sentence "dowt [4] = '\0'" is slightly imprecise. Since in C an array subindex starts counting from 0, it should be "dowt [3] = '\0'"

We made the code a bit clearer:

By using "strcat" instead of the pointer arithmetic, we can eliminate that pointer arithmetic and, along with it, the "pos" variable.

By declaring a constant as:
char * notFound = " - ";
we can simplify a bit the case when "found = 0", by using "strcpy".

By initializing "found" as it is declared.

And Finally, by eliminating "dow_to_string".

The changes can be easily seen at the file "check.c". A Makefile is present so it is simple to check the changes.

Lluis Sala
Feb - 26 - 2023