Obligatory Handin

Topics covered in this hand in:

- Functions
- Code standard and code structure, Makefile
- Test (unit test, black box)
- String
- Struct
- Pointer
- Dynamic memory allocation

Introduction

You should write a small program that can search for texts in text input. The input text shall be read from stdin. When finding a match the entire line (matching the search criteria) shall be printed to stdout.

The program is useful if you want to find out if a file contains a specific word. Instead of opening the file in an editor, click some button, enter a search string and find/check if the search string can be found you use your sgrep program. Think about looking for a search string in 2 files - it already becomes boring and slow with the editor way. Imagine doing it with 1000, or 10 000, files. Let the computer do the work.

Specification

The program shall be called sgrep - there already is an excellent program called `grep` (try it out for inspiration) and our name, sgrep, is short for `small grep`.

The user must supply at least one argument (the text to search for). If the program malfucntions 2 shall be returned.

We need only care about text input.

Note: The lines in the text file can be looooong. If you set a "big enough" length for one line we will test your code with a line 100 characters longer than that. Go for a solution which can handle any length (you're allowed to use getline which is available in GNU/Linux and cygwin and BSD's libc on MacOS).

The syntax of the program is:

```
sgrep  [OPTIONS] searchstring

Where [OPTIONS] include:

-i

  search for the text string case insensitive.

RETURN values:

 0 - on match    (same as the grep program)
 1 - on no match (same as the grep program)
 2 - failure     (same as the grep program)
```

The following is not obligatory:

Extend the syntax to the following:

```
sgrep [OPTIONS] regexp [FILE]

Where [OPTIONS] include:

-i

  search for the text string case insensitive.

-c

  count the hits/matches and print it out.

-n

  print the line with the line number prepended.

If [FILE] is supplied the program shall open and read from that file
instead of stdin.
```

Example use:

We give no interactive examples.

`./sgrep` - shall exit with error code 2

`echo "Carl-Einar" | ./sgrep carl` - shall print nothing and return 1

`echo "Carl-Einar" | ./sgrep Carl` - shall print “Carl-Einar” and return 0

`cat inputfile | ./sgrep Carl` - printout and return value depends on the data in the file

Note: try the above with the grep command instead.

The following are not obligatory:

`echo "Carl-Einar" | ./sgrep -i carl` - shall print "Carl-Einar" and return 0

`echo "Carl-Einar" | ./sgrep -c Carl` - shall print 1 and return 0

`echo "Carl-Einar" | ./sgrep -i -c carl` - shall print 1 and return 0

`grep -i -c carl apa.tat` - search for carl (ignoring case) in the file apa.txt. If carl is found at least once, return 0. If no occurance of apa was found, return 1. If something went wront, return 2.

Note: try the above with the grep command instead.

Tips an tricks

About getting help

- Don't hesitate to ask during supervision (called 'övningar' at CTH).
- Use the help (see below) we supply.

About the code

- Divide the code into smaller parts (e g parse, search) and let each such part (module) do that one thing and make sure it does it well.
- Don't include C files. Only include header files.
- Don't use global variables. Pass information as arguments instead.
- Test your code (we have written tests for you!).
- Make your code robust - we will test your code with nasty values.

If you want, we offer help

We have supplied:

- Makefile

- boiler plate files (c and h)
- directory structure

to get you started and to "force" you to divide your code. You are free to use this help or to ignore it. If you want to use it, look in the API in the src folder (either api.pdf or html/index.html) for a description of how the functions shall work (treat it as a specification) and fill in the code in the existing c and header files.

Print statements (for debuging)

If you want to you can use printf statements while you develop the code - and remove them afterwards. Or you can use the SGREP_DEBUG macro to enable/disable the printouts. Exmaple below:

```
#ifdef SGREP_DEBUG
  printf ("bl bla ...\n");
#endif
```

If you type/invoke `make debug` you get sgrep with debug printouts.

If you type/invoke `make` you get compile sgrep without debug printouts.

Makefile

We have supplied a Makefile which we hope make things a bit easier. Here's some examples of how to use the Makefile.

`make clean` - cleans up the code

`make bb` - performs black box tests

`make check` - performs unit tests (assuming your using our structure and naming)

`make debug` - creates a program with debug printouts enabled

`make` - creates a program with debug printouts disabled

`make valgrind` - memory checks the program

Division

You are free to solve this the way you want, but we recommend sticking to our proposal. You get quicker help from us, you get a structure (which you can critisze afterwards!), you get a lot of unit tests for free.

Divide the program into smaller units:

- sgrep_data (sgrep_data.h) contains struct and constans for use in the program
- parser (parser/parser.h and parser/parser.c) that parses the command line options
- searcher (searcher/searcher.h and searcher/searcher.c) searches for text in text
- main (main.c) connects the code in parser and search.

sgrep_data

Instead of passing variables of all kinds between the functions we lump things together in to a data structure (which we write ourself).

The data structure shall be called `sgrep_data_` and contain information about:

`char *reg_exp` - text to search for

`FILE *in` - stream to read from

`int matches` - number of matches found

The following are not obligatory:

`int case_sense` - specifies if case sensitive or not

`int mode` - specifies if we shall print the number of hits or print the line with a hit

`char **text;` - 'array' of strings matching the pattern. Must be freed

Make sure to typedef the struct `sgrep_data_` to `sgrep_data`. Doing this we can use `sgrep_data` and skip `struct`.

Parser

The responsibility of the parse is to gather information from the user and store it in the passed sgrep_data struct. The user passes information to the program using command line via argc and argv - not by interactive communication such as scanning stdin.

Check the manual for more information about the parser functions. There are template files provided for you.

Note: we only gather information from the user and store that in the passed struct. No search for a string in a file is done here

Searcher

The responsibility of the searcher is to search for a string in the lines of file in a file (or stdin). The search string is passed to the functions using the variable `reg_exp` in the sgrep_data struct.

Check the manual for more information about the searcher functions. There are template files provided for you.

Note: we only search for a string in the lines in a file. All information from the user is stored for us in the sgrep_data struct

main

The basic flow of the program is:

- parse command line arguments (why not using the parse function).
- loop through the lines in the indata:
  - for each line search for a match (why not using the search function)
  - if match, print the line

Non obligatory parts:

The basic flow of the program (non obligatory) is:

- parse command line arguments (why not using the parse function)
- loop through the lines in the indata
  - for each line search for a match (why not using the search function)
  - if match, save the line
  - when loop is done return all lines

Test

Black box

Your program shall pass the black-box tests written in the supplied Makefile. To test them yourself:

```
make bb
```

Unit test

We strongly recommend you use the unit tests in the supplied Makefile. This only works if you follow the division of code (the structure) we supply. To test them yourself:

```
make check
```

Non obligatory parts

```
make bb-extra
```

```
make check-extra
```

APPENDIX

About return values

To see the return value of a program after execution in bash (GNU/Linux, cygwin, MacOS), type `echo $?`, e g:

```
ls
echo $?
```

Failure

List files in a non-exitsting directory and check return value:

```
ls -al /shdfkjhskdjfh
```

```
echo $?
```

Should return a 2. The text printed out is simply text (a error notice in this case).

Success

- List files in an exitsting directory and check return value:

```
ls -al /tmp
```

```
echo $?
```

Should return a 0. The text printed out is a dir and file listing.

Usage

Many of you have used this, or seen Henrik and Rikard use this, when compiling.

```
gcc apa.c -o apa && ./apa
```

Bash only executes `apa` if compliation succeeded. Bash checks this by looking at the return value - it cares nada about the printout.

Think if you were to write a backup script for backing up data to a remote host. The first thing you would do, or at least one of the first things, would be to check if the remote host was up and running, e g with the command `ping`. If it wasn't up it wouldn't make sense continuing. By checing the return value from the ping command we could check it and if we find problems we could send an email, lit a ligt, call the police or waht ever and exit the script.