191.002 VU Betriebssysteme

EXERCISE 1A

Last update 2025-10-23 (Version 038bbf0d)

Assignment A – mygrep

Implement a reduced variation of the Unix-command `grep`. Write a C-program `mygrep`, which reads in several files and prints all lines containing a keyword.

```
SYNOPSIS
    mygrep [-i] [-o outfile] keyword [file...]
```

The program `mygrep` shall read files line by line and for each line check whether it contains the search term `keyword`. The line is printed if it contains `keyword`, otherwise it is not printed.

Your program must accept lines of any length. The program must be able to process data with the following characters `[0-9]` `[A-Z]` `[a-z]` `.,:-!=?%` and whitespace. Terminate the program with exit status `EXIT_SUCCESS`.

If one or multiple input files are specified (given as positional arguments after `keyword`), then `mygrep` shall read each of them in the order they are given. If no input file is specified, the program reads from *stdin*.

If the option `-o` is given, the output is written to the specified file (`outfile`). Otherwise, the output is written to *stdout*.

If the option `-i` is given, the program shall not differentiate between lower and upper case letters, i.e. the search for the keyword in a line is case insensitive.

As only `-o` and `-i` are valid options, `mygrep` shall print usage and exit with `EXIT_FAILURE` whenever the command line contains unrecognized or badly formatted options. For example, if a user calls `mygrep -x` this should be considered a user error.

Hint: Take a look at the function `strstr(3)` for searching a keyword within a string. The functions `tolower(3)` or `toupper(3)` might be useful when implementing case insensitive search.

Testing

Input is shown in blue color. Output to *stdout* (and *stderr*) is shown in black. (Note that in the following output sections `EXIT_SUCCESS` equals `0`, and `EXIT_FAILURE` equals `1`. Refer to `stdlib.h` for further details.) `^C` indicates `CTRL+C`, `^D` indicates `CTRL+D`. The placeholder `<usage message>` must be replaced by a proper usage message (printed to *stdout*), `<error message>` must be replaced by a meaningful error message (which is printed to *stderr*).

Test your program with various inputs, such as:

```
$ ./mygrep rat
Trap a rat!
Trap a rat!
Never odd or even
Rating System
Operating Systems
Operating Systems
^D
$ ./mygrep -i rat
Rating System
Rating System
Operating Systems
Operating Systems
^D
$ cat example.in
Operating Systems
Rating System
$ ./mygrep -o example.out rat example.in
$ cat example.out
Operating Systems
```

Testcases

A-Testcase 01: usage-1

```
$ ./mygrep -x
<usage message>
$ echo $?
1
```

A-Testcase 02: usage-2

```
$ ./mygrep -i -i
<usage message>
$ echo $?
1
```

A-Testcase 03: usage-3

```
$ ./mygrep -o
<usage message>
$ echo $?
1
```

A-Testcase 04: usage-4

```
$ ./mygrep
<usage message>
$ echo $?
1
```

A-Testcase 05: easy-1

```
$ echo -e "Systems, welcome\nto\noperating systems.\n" | ./mygrep "sys"
operating systems.
$ echo $?
0
```

A-Testcase 06: easy-2

```
$ echo -e "Small\nis smaller\nthan huge." | ./mygrep -i "small"
Small
is smaller
$ echo $?
0
```

A-Testcase 07: easy-3

```
$ echo -e "Oh, z.,\nZZZzZ.\nz\nZ\nZZZz\nzZ...." | ./mygrep -i "Z."
Oh, z.,
ZZZzZ.
zZ....
$ echo $?
0
```

A-Testcase 08: easy-4

```
$ echo -e "new\nline" | ./mygrep ""
new
line
$ echo $?
0
```

A-Testcase 09: file-1

```
$ echo -e "2 is the oddest prime.\nwhat?\nAre YOU sure\n12345, yes." > infile.txt
$ ./mygrep -o outfile.txt 2 infile.txt
$ echo $?
0
$ cat outfile.txt
2 is the oddest prime.
12345, yes.
```

A-Testcase 10: file-2

```
$ echo '.,:-!=?% is %n%% TEST???' > infile1
$ echo '.,:-!=?% is %n%% yes!!!' >> infile1
$ echo -e "testing\ntester\nTESTEST\nteeest\nabc\ndef" > infile2
$ ./mygrep -i Test infile1 infile2
.,:-!=?% is %n%% TEST???
testing
tester
TESTEST
$ echo $?
0
```

A-Testcase 11: file-3

```
$ echo "some old content" > outfile
$ echo -e "abcdef\nabbcdef\nabscdef\nabecdef\naabcdef\n" | ./mygrep -i -o outfile "ABC"
$ echo $?
0
$ cat outfile
abcdef
aabcdef
```

A-Testcase 12: long-line

```
$ ( echo -n "yes"; printf -- "-%.0s" {1..8000}; echo "..." ) > longline
$ ./mygrep -i "yes" longline > longgrep
$ echo $?
0
$ cat longgrep | tr -d "-"
yes...
```

A-Testcase 13: file-error-1

```
$ rm -rf nonExistingTestfile
$ ./mygrep test nonExistingTestfile
<error message>
$ echo $?
1
```

A-Testcase 14: file-error-2

```
$ touch existingTestfile
$ chmod 0000 existingTestfile
$ echo "test" > existingTestfile
bash: existingTestfile: Permission denied
$ echo "test" | ./mygrep -o existingTestfile test
<error message>
$ echo $?
1
```

Recommendations and Guidelines

In order to make all solutions more uniform and to avoid problems with different C dialects, please try to follow the following guidelines.

General Guidelines

Try to follow the following guidelines closely.

1. All source files of your program(s) should compile with
   ```
   $ gcc -std=c99 -pedantic -Wall -D_DEFAULT_SOURCE -D_BSD_SOURCE -D_SVID_SOURCE
                                  -D_POSIX_C_SOURCE=200809L -g -c filename.c
   ```
   without *warnings and info messages* and your program(s) should link without warnings too.
2. There should be a Makefile implementing the targets: `all` to build the program(s) (i.e. generate executables) from the sources (this must be the first target in the Makefile); `clean` to delete all files that can be built from your sources with the Makefile.
3. Make all targets of your Makefile idempotent, i.e. execution of `make clean; make clean` should yield the same result as `make clean`, and should not fail with an error.
4. The program shall operate according to the specification/assignment without major issues (e.g., segmentation fault, memory corruption).
5. Arguments should be parsed according to UNIX conventions (we strongly encourage the use of `getopt(3)`). The program should conform to the given synopsis/usage in the assignment. If the synopsis is violated (e.g., unspecified options or too many arguments), the program should terminate with the usage message containing the program name and the correct calling syntax. Argument handling should also be implemented for programs without arguments.
6. Correct (=normal) termination should include a cleanup of resources.
7. Upon success the program shall terminate with exit code 0, in case of errors with an exit code greater than 0. We recommend to use the macros `EXIT_SUCCESS` and `EXIT_FAILURE` (defined in `stdlib.h`) to enable portability of the program.
8. If a function indicates an error with its return value, it *should* be checked in general. If the subsequent code depends on the successful execution of a function (e.g., resource allocation), it is very *important* to check the return value.
9. Functions that do not take any parameters should be declared with `void` in the signature, e.g., `int get_random_int(void);`.
10. Procedures (i.e., functions that do not return a value) should be declared as `void`.
11. Error messages shall be written to `stderr` and should contain the program name `argv[0]`.
12. Do not use the following functions to avoid crashes due to invalid inputs:
    - `gets(3)`; use `fgets(3)` instead.
    - `scanf(3)` and `fscanf(3)`; use `fgets(3)` and `sscanf(3)` instead.
    - `atoi(3)` and `atol(3)`; use `strtol(3)` instead.
13. Documentation is a nice bonus. Format the documentation in Doxygen style (see Wiki and Doxygen's intro).

14. Write meaningful comments. For example, meaningful comments describe the algorithm, or why a particular solution has been chosen, if there seems to be an easier solution at a first glance. Avoid comments that just repeat the code itself
(e.g., `i = i + 1; /* i is incremented by one */`).

15. The documentation of a module should include: name of the module, name and student id of the author (`@author` tag), purpose of the module (`@brief`, `@details` tags) and creation date of the module (`@date` tag).

    The Makefile should also include a header, with author and program name at least.

16. Each function shall be documented either before the declaration or the implementation. It should include purpose (`@brief`, `@details` tags), description of parameters and return value (`@param`, `@return` tags) and description of global variables the function uses (`@details` tag).

    You should also document `static` functions (see `EXTRACT_STATIC` in the file `Doxyfile`). Document visible/exported functions in the header file and local (`static`) functions in the C file. Document variables, constants and types (especially `structs`) too.

17. Documentation, names of variables and constants is best done in English.

18. Internal functions shall be marked with the `static` qualifier and are not allowed to be exported (e.g., in a header file). Only functions that are used by other modules shall be declared in the header file.

19. All exercises shall be solved with functions of the C standard library. If a required function is not available in the standard library, you can use other (external) functions too. Avoid reinventing the wheel (e.g., re-implementation of `strcmp`).

20. Name of constants shall be written in upper case, names of variables in lower case (possibly with the first letter capitalized).

21. Use meaningful variable and constant names (e.g., also semaphores and shared memories).

22. Avoid using global variables as far as possible.

23. All boundaries shall be defined as constants (macros). Avoid arbitrary boundaries. If boundaries are necessary, treat its crossing.

24. Avoid side effects with `&&` and `||`, e.g., write `if(b != 0) c = a/b;` instead of `if(b != 0 && c = a/b)`.

25. Each `switch` block should contain a `default` case. If the case is not reachable, write `assert(0)` to this case (defensive programming).

26. Logical values shall be treated with logical operators, numerical values with arithmetic operators (e.g., test 2 strings for equality by `strcmp(...) == 0` instead of `!strcmp(...)`).

27. Indent your source code consistently (there are tools for that purpose, e.g., `indent`).

28. Avoid tricky arithmetic statements. Programs are written once, but read many times. Your program is not always better if it is shorter!

29. For all I/O operations (read/write from/to `stdin`, `stdout`, files, sockets, pipes, etc.) use *either* standard I/O functions (`fdopen(3)`, `fopen(3)`, `fgets(3)`, etc.) *or* POSIX functions (`open(2)`, `read(2)`, `write(2)`, etc.). Remember, standard I/O functions are buffered. Mixing standard I/O functions and POSIX functions to access a common file descriptor can lead to undefined behaviour and is therefore a bad idea.

30. If asked in the assignment, you should implement signal handling (`SIGINT`, `SIGTERM`). You must use only *async-signal-safe* functions in your signal handlers.

31. Close files, free dynamically allocated memory, and remove resources after usage.

32. Don't waste resources due to inconvenient programming. Header files shall not include implementation parts (exception: macros).

33. To comply with the given testcases, the program output must exactly match the given specification. Therefore you should only print debug information if the compile flag `-DDEBUG` is set, in order to avoid problems with the automatic checker.