191.002 VU Betriebssysteme
EXERCISE 1A

Last update 2023-10-16

Assignment A – myexpand

Implement a variation of the Unix-command `expand`. Write a C-program `myexpand`, which reads in several files and replaces tabs with spaces.

```
SYNOPSIS:
    myexpand [-t tabstop] [-o outfile] [file...]
```

The program shall read each file given as positional argument (or *stdin* if there are no positional arguments) line by line and search for tab characters (`\t`).

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`.

Each of these tabs is replaced with a number of space characters, such that the next character is placed at the next multiple of the tabstop distance within the line. This position can be calculated as follows:

```
p = tabstop * ((x / tabstop) + 1)
```

where `x` is the position of the tab character (after expanding any previous tabs on that line) and `/` is the standard integer division (retaining only the quotient).

The tabstop distance is 8 by default, but this can be overridden with an strictly positive integer (smaller or equal to 64) using the option `-t`.

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

Testing

Test your program with various inputs, such as a file `t1.txt` with following content (where → represents the ASCII tab character, entered by hitting the tabulator key):

```
1234567890
123→567890
→→90
```

Executing your program should give an output similar to the following:

```
$ ./myexpand t1.txt
1234567890
123     567890
                90

$ ./myexpand -t 4 -o out.txt t1.txt
$ cat out.txt
1234567890
123 567890
        90
```

Mandatory testcases

Input shown in blue color. Output to *stdout* (and *stderr*) 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*).

A-Testcase 01: usage-1

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

A-Testcase 02: usage-2

```
$>./myexpand -t
<usage message>
$>echo $?
1
```

A-Testcase 03: usage-3

```
$>./myexpand -t 7 -t 7
<usage message>
$>echo $?
1
```

A-Testcase 04: usage-4

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

A-Testcase 05: usage-5

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

A-Testcase 06: usage-6

Note: This testcase was removed. (It was `./myexpand foo bar` before.)

A-Testcase 07: usage-7

```
$>./myexpand -t 0
<usage message>
$>echo $?
1
```

A-Testcase 08: usage-8

```
$>./myexpand -t 1000
<usage message>
$>echo $?
1
```

A-Testcase 09: usage-9

```
$>./myexpand -t foo
<usage message>
$>echo $?
1
```

A-Testcase 10: usage-10

```
$>./myexpand -t 7.1
<usage message>
$>echo $?
1
```

A-Testcase 11: easy-1

```
$>echo -e "1234567890\n123\t567890abcdefg\n\t\t90" | ./myexpand
1234567890
123     567890abcdefg
                90
$>echo $?
0
```

A-Testcase 12: easy-2

```
$>echo -e "1234567890\n123\t567890\n\t\t90" | ./myexpand -t 4 -o outfile
$>echo $?
0
$>cat outfile
1234567890
123 567890
        90
```

A-Testcase 13: file-1

```
$>echo "some old content" > outfile.txt
$>echo -e "a\tb\tc\td\te\tf\t\t." > infile.txt
$>./myexpand -t 2 -o outfile.txt infile.txt
$>echo $?
0
$>cat outfile.txt
a b c d e f   .
```

A-Testcase 14: file-2

```
$>echo -e 'please\n\tre-arrange\nthis\t%n%%\n.,:-!=?\t%\t where???' > infile
$>./myexpand -t 3 -o outfile infile
$>echo $?
0
$>cat outfile
please
   re-arrange
this  %n%%
.,:-!=?  %   where???
```

A-Testcase 15: long-line

```
$>( echo -ne "X\tx"; printf -- "-%.0s" {1..8000}; echo "Z" ) > longline
$>./myexpand -t 4 longline > longexp
$>echo $?
0
$>cat longexp | tr -d "-"
X   xZ
```

A-Testcase 16: file-error-1

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

A-Testcase 17: file-error-2

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

Coding Rules and Guidelines

Your score depends upon the compliance of your submission to the presented guidelines and rules. Violations result in deductions of points. Hence, before submitting your solution, go through the following list and check if your program complies.

Rules

Compliance with these rules is essential to get any points for your submission. A violation of any of the following rules results in 0 points for your submission.

1. All source files of your program(s) must compile via
   ```
   $ gcc -std=c99 -pedantic -Wall -D_DEFAULT_SOURCE -D_BSD_SOURCE -D_SVID_SOURCE
                                   -D_POSIX_C_SOURCE=200809L -g -c filename.c
   ```
   without *errors* and your program(s) must link without *errors*. The compilation flags must be used in the Makefile. The feature test macros must not be bypassed (i.e., by undefining these macros or adding some in the C source code).
2. The functionality of the program(s) must conform exactly to the assignment. The program(s) shall operate according to the specification/assignment given the test cases in the respective assignment. Additional white spaces or any other deviation from the specified input and output format may lead to a failure of the respective test case.

General Guidelines

Violation of following guidelines leads to a deduction of points.

1. All source files of your program(s) must 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) must link without warnings.
2. There must 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. All targets of your Makefile must be idempotent. I.e. execution of `make clean; make clean` must yield the same result as `make clean`, and must 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 have to be parsed according to UNIX conventions (we strongly encourage the use of `getopt(3)`). The program has to conform to the given synopsis/usage in the assignment. If the synopsis is violated (e.g., unspecified options or too many arguments), the program has to 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, including a cleanup of resources.
7. Upon success the program has to 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), then the return value *must* be checked.

9. Functions that do not take any parameters have to 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) have to be declared as `void`.

11. Error messages shall be written to `stderr` and should contain the program name `argv[0]`.

12. It is forbidden to use the functions: `gets`, `scanf`, `fscanf`, `atoi` and `atol` to avoid crashes due to invalid inputs.

FORBIDDEN	USE INSTEAD
gets	fgets
scanf	fgets, sscanf
fscanf	fgets, sscanf
atoi	strtol
atol	strtol

13. Documenation is mandatory. 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 must 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).

    Also the Makefile has to 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 `struct`s) too.

17. Documentation, names of variables and constants shall be 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 (maybe with fist letter capital).

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 must 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 more times. Your program is not 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 forbidden.
30. If asked in the assignment, you must implement signal handling (`SIGINT`, `SIGTERM`). You must only use *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 are only allowed to to print any debug information if the compile flag `-DDEBUG` is set.