



FIT2100 Assignment Part A: Building a File Utility with C Programming Semester 2 2021

Dr Tawfiq Islam
Lecturer, Faculty of IT
Email: tawfiq.islam@monash.edu
© 2021, Monash University

August 7, 2021

Revision Status

version 2: July 2021 by Tawfiq Islam

Contents

1	Introduction	3
2	Caveats	3
3	If you require extra help	4
3.1	References	4
4	Task 1: File content viewing functionality with hard-coded options	4
5	Support Command-line Arguments	5
5.1	Task 2: Functionality to support the use of a different source file	5
5.2	Task 3: Copy file functionality	5
5.3	Task 4: Custom View/Copy file functionality	6
5.4	Task 5: Tail View/Copy file functionality	6
5.5	Command-line argument summary	7
5.6	Example commands:	7
5.7	Important: commenting is required	8
5.8	Hints and tips	9
5.9	Marking criteria	9
6	Submission	10
6.1	Deliverables	10
6.2	Academic Integrity: Plagiarism and Collusion	11

1 Introduction

In this assignment, you will build a multipurpose file utility¹, which combines the **simplified** features of multiple Linux utilities such as `head`, `tail`, `cat`, and `cp`.

This document constitutes the requirement specification for this assignment. You will be assessed on your ability to both comprehend and comply with the requirements as specified herein.

Submission due date: 27th August 2021 (Friday) 5:00pm AEST.

Late submissions: A late submission penalty of 10% of the total available marks per day will apply.

This assignment is worth 15% of the total marks for this unit.

2 Caveats

To complete these tasks, you are allowed to use any of the standard C library functions found on your virtual machine environment², except the following:

You are *NOT* allowed to use any functions available in `<stdio.h>`. This means you cannot use `printf()` to produce output. (For example: to print output in the terminal, you will need to write to *standard output* directly, using appropriate file system calls.) In addition, you are *NOT* allowed to use the `system()` library function, and any library functions which spawns new process(s) from your program. This makes the assignment more challenging, but also means you are interacting with services of the operating system directly.

Your main C source file should be named with your student ID as: `123456789_fileutil.c`, where 123456789 is your student ID.³

¹An utility is a software that adds functionality to a computer, or helps users achieve certain tasks.

²For example, you might choose to use `getopt` or other advanced libraries according to your preference, but use of any particular library is neither expected nor required, except as required to make system calls.

³If your completed program contains multiple source files, you may name other source and header files as you wish.

3 If you require extra help

This assignment is an independent learning and assessment exercise.

You may utilise the Ed Discussion Forum to ask questions and obtain clarification, however you may not share details or code in your implementation with other students, nor may you show your code to the teaching team prior to submission. This is an assessment task: tutors and lecturers are not permitted to help you debug your assignment code *directly* (you are expected to debug and test your own code), but can help with more general queries, such as queries related to C programming syntax, concepts and debugging tips.

You may make use of online references with appropriate citation in accordance with academic integrity policies, however your work must be your own.

3.1 References

The following resources are available on the FIT2100 Unit Information page.

- Curry, David, *UNIX Systems Programming for SVR4*, Chapter 3: 'Low-Level I/O Routines'.
- Curry, David, *Using C on the UNIX System*, Chapter 3: 'Low-Level I/O'.
- He, Jialong, *LINUX System Call Quick Reference*.

4 Task 1: File content viewing functionality with hard-coded options

Write an utility called `fileutil` ('File Utility') which does the following:

1. Opens a file named `sample.txt` in the current working directory.
2. Outputs the first 10 lines of the file contents. If the file contains less than 10 lines, then show entire file contents (to *standard output*—usually the terminal). Note that, when the program completes normally, no other output should be produced.

Your program should always exit 'cleanly.' This means closing any open files and freeing any other resources you may have allocated before termination.

If there is a problem accessing the file (e.g. file does not exist), your program should display an appropriate and sensible error message (you should output this to the program's standard error stream rather than standard output) and exit cleanly with a return value of 1. On successful completion, your program should return an error code of 0.

Write up an instruction manual (user documentation) in a plain text file, explaining how to compile your program and how to use it. You may use your user documentation itself or any other text file to test your program.

5 Support Command-line Arguments

Now, extend the functionalities of the same program (from Task 1) to allow the user to run it with command-line arguments as follows. Extend your user documentation to include **all the added usage functionalities**.

5.1 Task 2: Functionality to support the use of a different source file

Allow the user to specify a different sourcefile as a source (instead of `sample.txt`) by putting the filename as a command-line argument. You can assume that the user will provide the *absolute path*⁴ for the file. Note that, if the sourcefile argument is specified, it **must** be the first argument. If the sourcefile argument is not specified, the `sample.txt` should be used as the source file by default.

5.2 Task 3: Copy file functionality

Instead of showing the contents of the file, allow the user to specify a `-d` argument at the command line in order to specify a destination directory where a copy of the source file will be created, but with only the first 10 lines. If the source file contains less than 10 lines, then the entire contents of the source file should be copied. When the `-d` argument is used, the argument immediately following it should be given as the absolute path of the destination directory. If it is not, the program arguments are invalid (see Section 5.6).

⁴An absolute path is defined as a **full** path beginning with /

If the destination directory already has a file with the same name as the source file or if the destination directory does not exist, your program should output an appropriate error message to standard error then exit cleanly with an error code of 2.

Upon successful completion of the program with this argument, there will be **2 versions** of the same file (i.e. the original file in the source directory with all the file contents, and the newly created file in the destination directory that contains only the first 10 lines of the original file). Also, you should output the message "Copy successful" to standard output.

5.3 Task 4: Custom View/Copy file functionality

Instead of copying only the first 10 lines of the file, allow the user to specify a `-n` argument at the command line in order to specify how many lines from the source file should be displayed/copied. When the `-n` argument is used, the argument immediately following it should be given as the number of lines on which an operation should be performed. If it is not, the program arguments are invalid (see Section 5.6).

Note that, when the `-n` argument is provided along with the `-d` argument, the file copy functionality should be performed. If the `-n` argument is provided without the `-d` argument, the file content viewing functionality should be performed. If the file contains less than `n` lines, then display/copy the entire contents of the file.

In case of the copy functionality, if the destination directory already has a file with the same name as the source file or if the destination directory does not exist, your program should output an appropriate error message to standard error, then exit cleanly with an error code of 2. If the copy functionality was performed, you should also output the message "Copy successful" to standard output.

5.4 Task 5: Tail View/Copy file functionality

Allow the user to specify a `-L` argument at the command line to indicate that the last 10 lines of the file should be displayed/copied. As before, the appropriate success or error messages should be outputted, and the entire file contents should be displayed/copied if the file contains less than 10 lines.

Note that, when the `-L` argument is provided along with the `-d` argument, the file copy functionality should be performed. If the `-L` argument is provided without the `-d` argument, the file content viewing functionality should be performed. If the `-n` argument is also present along with the `-L` argument, the desired operation should be performed on the last `n` lines.

5.5 Command-line argument summary

A summary of all the command-line arguments and their functionalities are shown in the table below:

Table 1: Summary of functionalities for different command-line arguments.

Command-line Argument	Functionality
<i>sourcefile</i>	Allows the user to input a path for the sourcefile to display/copy. <i>sourcefile</i> must appear immediately after the command (e.g., <code>./fileutil sourcefile</code>).
-d <i>destdir</i>	Allows the user to input a path for the destination directory for copying the source file. (<i>destdir</i> must appear immediately after -d)
-n <i>numlines</i>	Allows the user to input a number to specify how many lines of text should be displayed/copied. (<i>numlines</i> must appear immediately after -n)
-L	Makes the program to switch to Tail mode where the View/Copy should be performed on the last 10 (or <i>n</i>) lines.

Some command-line arguments have **dependencies** on other arguments (as mentioned above).

The position of the *sourcefile* argument is fixed. If it appears, it must be the first argument. However, the ordering of the rest of the arguments is NOT fixed and can vary. For example, check the last 2 example commands in Section 5.6.

Do not collect these options from standard input, or prompt the user to enter them. They should be specified as *program arguments*.

5.6 Example commands:

- `$./fileutil`
Displays the first 10 lines of the `sample.txt` file from the current directory
- `$./fileutil -n 20`
Displays the first 20 lines of the `sample.txt` file from the current directory
- `$./fileutil /home/student/dir1/a.txt`
Displays the first 10 lines of the `a.txt` file
- `$./fileutil /home/student/dir1/a.txt -n 20`
Displays the first 20 lines of the `a.txt` file

- `$./fileutil /home/student/dir1/a.txt -d /home/student/dir2/`
Copy a.txt to dir2 where the newly created a.txt only contains the first 10 lines
- `$./fileutil /home/student/dir1/a.txt -d /home/student/dir2/ -n 15`
Copy a.txt to dir2 where the newly created a.txt only contains the first 15 lines
- `$./fileutil /home/student/dir1/a.txt -d /home/student/dir2 -n 15 -L`
Copy a.txt to dir2 where the newly created a.txt only contains the last 15 lines
- `$./fileutil -d /home/student/dir2`
Copy the first 10 lines of sample.txt from the current directory to dir2
- `$./fileutil /home/student/dir1/a.txt -n`
Invalid argument, no line numbers specified after the -n argument!
- `$./fileutil /home/student/dir1/a.txt -d -L /home/student/dir2`
Invalid argument: immediately after -d, a directory path was expected
- `$./fileutil /home/student/dir1/a.txt -d /home/student/dir2 -L`
Copy a.txt to dir2 where the newly created a.txt only contains the last 10 lines
- `$./fileutil /home/student/dir1/a.txt -L -n 15 -d /home/student/dir2/`
Copy a.txt to dir2 where the newly created a.txt only contains the last 15 lines
- `$./fileutil /home/student/dir1/a.txt -d /home/student/dir2/ -n 15 -L`
Copy a.txt to dir2 where the newly created a.txt only contains the last 15 lines

5.7 Important: commenting is required

Commenting your code is essential as part of the assessment criteria (refer to Section 5.9). All program code should include three types of comments:

- (a) File header comments at the beginning of your program file, which specify your name, your Student ID, the start date and the last modified date of the program, as well as with a high-level description of the program.
- (b) Function header comments at the beginning of each function should describe the function, arguments and interpretation of return value.
- (c) In-line comments within the program are also part of the required documentation.

5.8 Hints and tips

Do...

- + read up on low-level I/O system calls
- + write normal program output to *standard output*
- + close open files before exiting and free any manually-allocated memory
- + check return values and handle error conditions
- + access and modify valid memory
- + read up on `argc` and `argv`
- + write user documentation in a plain text file
- + in user documentation: explain all functionality to those who might use your program
- + use descriptive, self-explanatory variable names
- + break your program logic into multiple functions
- + use consistent code indentation for clarity
- + comment your code with file header, function header and inline comments
- + test your program in the specified VM environment
- + **use the Ed Discussion Forum**
- + **read the assignment specification carefully and thoroughly**

Don't... (!)

- use fancy C `<stdio.h>` library functions that don't demonstrate your understanding of making system calls to the OS directly.
- write error messages to standard output (use standard error instead)
- terminate without cleaning up first
- act in undefined ways when a parameter is invalid or a file can't be opened
- attempt to set values in undefined pointers
- prompt the user to enter options after your program has started
- submit user documentation in a Word document or PDF
- expect your users to understand the C code inside your program (users are not always programmers!)
- use vague single-character variable names
- stuff everything into a single `main` function
- use inconsistent indentation or fail to indent nested code blocks
- write uncommented code or add comments at the last minute
- forget to run your finished program through `valgrind` to test for memory access bugs
- **implement each task as a separate program**
- **treat this table as a substitute for reading the spec carefully.**

5.9 Marking criteria

Each task is worth an equal share of the total. The same marking criteria will be applied on all tasks:

- 50% for working functionality according to specification.
- 20% for code architecture (algorithms, use of functions for clarity, appropriate use of libraries, correct use of pointers, etc. in your implementations of all the tasks.)
- 10% for general coding style (clarity in variable names, function names, blocks of code clearly indented, etc.)
- 20% for documentation (user documentation describes functionality for the relevant task, code is well-commented.)

6 Submission

There will be NO hard copy submission required for this assignment. You are required to submit all your deliverables (see Section 6.1) as **individual files**. Do ensure that your submission complies with the requirements set out in this specifications document.

Your submission will be done via the assignment submission link on the FIT2100 Moodle site, and should be submitted by the deadline specified in Section 1, i.e. **27th August 2021 (Friday) 5:00pm AEST**.

Note: You must ensure you complete the entire Moodle submission process (do not simply leave your assignment in draft status) to signify your acceptance of academic integrity requirements.

Additionally, your program must be able to run in the Linux Virtual Machine environment which has been provided for this unit. Any implementation that does not run at all in this environment will receive no marks.

6.1 Deliverables

Your submission **should include** the following files:

- All C source files (.c format) required to compile and run your program.
- A user documentation file (.txt format) of not more than 80 lines which provides clear and complete instructions on how to compile your program, and how to run all of the requested features.

Your submission **may optionally include** the following files, if you require them for your implementation:

- C header files (.h format)
- A makefile

Do not submit compiled executable programs. Marks will be deducted for any of these requirements that are not strictly complied with.

6.2 Academic Integrity: Plagiarism and Collusion

Plagiarism Plagiarism means to take and use another person's ideas and or manner of expressing them and to pass them off as your own by failing to give appropriate acknowledgement. This includes materials sourced from the Internet, staff, other students, and from published and unpublished works.

Collusion Collusion means unauthorised collaboration on assessable work (written, oral, or practical) with other people. This occurs when you present group work as your own or as the work of another person. Collusion may be with another Monash student or with people or students external to the University. This applies to work assessed by Monash or another university.

It is your responsibility to make yourself familiar with the University's policies and procedures in the event of suspected breaches of academic integrity. (Note: Students will be asked to attend an interview should such a situation is detected.)

The University's policies are available at: <http://www.monash.edu/students/academic/policies/academic-integrity>