INF1002 Programming Fundamentals – C Lab5

OBJECTIVES:

To understand the structure of files, and to open, read from, write to, and close files in C.

PRE-READING

For some of the exercises in this lab, it may be useful to know a little bit more about how C programs interact with the operating system.

You may know that, when starting a program from the command line, you can specify *command line arguments* that tell the program more about what you want it to do. When compiling a C program, for example, you might type '*cl hello.c*' or '*gcc hello.c*', which tell the *cl* and *gcc* programs, respectively, to read from a file called *hello.c*.

In C, the command line arguments are passed to the program as arguments of the main() function. The complete function prototype for main() is:

```
int main(int argc, char **argv)
```

The *argc* argument contains the number of arguments on the command line, including the name of the program itself. The *argv* argument is an array of strings containing the arguments in order. So, *argv[0]* is the name of the program, *argv[1]* is the first argument, and so on. Note that these are always strings, so you need to use *atoi()* and friends if you want to convert them to numbers.

Most compilers allow you to omit the arguments for *main()*, which is why we've been able to use just int *main()* up until now. You only need them if your program uses the arguments.

The return value of *main()* is an integer. This integer is returned to the program that invoked your program and is usually used to indicate whether your program succeeded. By convention, a return value of zero indicates that a program completed without any errors, and a non-zero value indicates that something went wrong. Some programs return specific non-zero values to indicate different kinds of errors. Shell scripts (in Unix) and batch files (in Windows) can use the return values from program to make decisions about what they should do.
Here is a link with examples: https://stackoverflow.com/questions/16869467/command-line-arguments-reading-a-file/16869591

Lab Assignment:

To help you better practice, you need to perform a set of tasks in one auto-grading system, Gradescope in LMS/xSiTe. Gradescope will provide you immediate feedback of your program, such that you will know the issue of your program.

In this lab, you need to finish one task in Gradescope system. Gradescope system will provide you immediate feedback about the correctness of your program. You can view the feedback of your system by clicking the failed case. Gradescope adopts a test cases-based approach to check your program. Your score is depending on the number of test cases that you program can pass. Note that to train

you to have a good programming practice, you must write your program strictly according to the requirement of the tasks, including your input and output format. If there is any difference (even one space), your program will fail on the test cases. SO, TRAIN YOURSELF TO BE AN EXACT THINKER!

To help you practice, you are allowed to do multiple attempts for each task. Enjoy your learning!

Task: MINI TAR

Task Description:

The *tar* program (for "**t**ape **ar**chive") creates an uncompressed binary file archive by joining a collection of files end to end. At the start of each file in the archive is a fixed-length header that stores information about the file, represented as a C structure as follows:

```
/*
 * Standard Archive Format - Standard TAR - USTAR
 * from https://www.fileformat.info/format/tar/corion.htm
 */

#define  RECORDSIZE  512
#define  NAMSIZ      100
#define  TUNMLEN      32
#define  TGNMLEN      32

struct header {
    char    name[NAMSIZ];
    char    mode[8];
    char    uid[8];
    char    gid[8];
    char    size[12];
    char    mtime[12];
    char    chksum[8];
    char    linkflag;
    char    linkname[NAMSIZ];
    char    magic[8];
    char    uname[TUNMLEN];
    char    gname[TGNMLEN];
    char    devmajor[8];
    char    devminor[8];
};
```

For the purposes of this exercise, we only need to worry about the *name* and *size* fields of the structure *struct header*.

The *name* field contains the name of the file, while the *size* field contains the length of the file in bytes. Note that the *size* field is a string, not an integer, so it needs to be converted using *atoi()* to perform arithmetic on it. We can ignore (no need to comment out and no need to initialise) all the other fields in our program.

A tar archive thus has the form:

struct header	*file1 data*	struct header	*file2 data*	struct header	*file3 data*	...

The header sections always have the same length, *sizeof(struct header)*, but the length of each file is given by the *size* field in the header.

Write a program called *miniTar* that accepts the two given file names ("*File1.txt*" and "*File2.txt*") on the command line (*miniTar File1.txt File2.txt*) and generate an archive file ("*Result.tar*") containing all these files, with the name and size fields filled in as described above. For this task, the file open/write/read mode for *Result.tar* is "binary".

Note:

1. Make the program flexible to accept any number of input files: *File1.txt*, *File2.txt*, …
2. Use `sys.argv[]` for user inputs.
3. 'tar' files are typically binary. Make sure that the file open and write mode for `Result.tar` is "binary".
4. Best Practices for Writing File Size in Tar Headers
   - **'tar'** format specification traditionally uses octal representation for numeric fields, including file sizes. This is a convention that dates to the early days of Unix and ensures compatibility with other tar implementations.
   - Use Octal Representation: Using octal representation for the file size to maintain compatibility with the tar format. The snprintf function with the format "%011lo" ensures the size is zero-padded to 11 characters, which is the standard for tar headers.
     - Ensure Correct Padding: The size field should be properly padded with leading zeros to fit the 11-character width. This is crucial for the tar format to be correctly interpreted by other tools.
     - %: Indicates the start of a format specifier.
     - 0: Pads the output with leading zeros.
     - 11: Specifies the width of the output, ensuring it is 11 characters long.
     - l: Indicates that the argument is of type long.
     - o: Formats the number as an octal (base-8) number.
     - Example:
       - For File1.txt, the file size in octal is 00000000037
       - For File2.txt, the file size in octal is 00000000076
   - Handle Large Files: If you need to handle very large files (greater than 8 GB), consider using the GNU tar format or another extended format that supports larger sizes, as the traditional tar format has limitations. We will ignore this scenario for this task.
5. Add error handlers and comments where necessary.

Hints:

See http://www.cplusplus.com/reference/cstdio/fread/ for an example of how to find the length of a file and read it into memory. Note that reading the whole file into memory at once may require quite a lot of memory; you might like to try finding a more efficient method of copying data from the input file to the output file.

Example:

```
miniTar File1.txt File2.txt

Archive 'Result.tar' created successfully.
```

Instructions to submit and auto grade your code to Gradescope:

1. Download the skeleton code file "*miniTar.c*" from the "C\Lab\Lab5" xSiTe folder.
2. Download `File1.txt` and `File2.txt` from the "C\Lab\Lab5" xSiTe folder.
3. Do not change file name or function names.
4. Add your code to the function "*main()*" and other required functions.
5. Use `sys.argv[]` for user inputs.
6. Make sure that the file open/write/read mode for `Result.tar` is "binary".

7. Drag and drop your locally tested code to Gradescope. It is not required to submit *File1.txt* and *File2.txt*.
8. You can submit and auto graded any number of times. Last submission counts.
9. There is only one test case and maximum score for this task is 5.
10. Autograder will be timed out after 10 minutes.