Project 2: Unix Utilities

Igor Zimarev, Daniil Komov

Link to source code GitHub repository: https://github.com/zimaBloo/Project-2-Unix-Utilities.git

Project description

In this project we implemented the simplified versions of basic and commonly known UNIX utilities in the C language. All together we implemented 4 utilities: **may-cat**, **my-grep**, **my-zip**, and **my-unzip**. The objective of the project was to learn the basic UNIX utility functionality and implementation in C. We improved our skills of working with I/O, error handling, and learned several useful C functions.

my-cat: Reads and prints contents of a specified file, corresponding to a standard UNIX utility *cat*. It is also capable of handling several files sequentially. The utility would be called in the following way, first by writing the name of the utility as the first argument, followed by the file name: `$ ./my-cat example.txt` or `$ ./my-cat test.txt test2.txt`

my-grep: Searches a term in a case-sensitive fashion from the specified file and prints lines that have a matching term. Corresponds to standard **grep** utility in UNIX. When no file is provided it is capable of handling standard input. Also, it is capable of performing the search if multiple files are provided, and the given lines that are searched can be arbitrarily long. The utility is called by writing the name as the first argument, followed by the search term as the second argument, and then zero or more files as the latter arguments. An example my-grep call would look like this: `$ ./my-grep example words.txt` or `$ ./my-grep example file.txt book.txt`

my-zip: Compresses a file with run-length encoding (RLE) method. This encoding method scans the occurrence of every character in order and checks whether it has any same consecutively occurring characters. Thus, every character is written into a 5-byte set of firstly the 4-byte binary integer representation of the number of consecutive occurrences of the character in question, followed by the 1-byte ASCII character. Using a simple example, the string mmmkkkkk would be turned into 3m5k (but with integers being written in binary form). As such, the zipped data is stored in the binary format, and since the utility is designed to provide compressed data through redirection of standard output, the terminal might provide weird results when the binary data in the standard output is displayed through it. In most cases, the terminal will not show the 4-byte binary integer representation, but only the 1-byte ASCII character corresponding to the binary number. So, the utility is efficient in creating zipped files through redirection, which can be later decompressed with my-unzip. The utility is also capable of handling multiple input files and compressing them into a single file, effectively erasing the boundaries of the input files. The utility can be used in the following way: `$ ./my-zip file.txt > zip.z` or `$ ./my-zip book.txt words.txt > compressed.z`

my-unzip: Reverses the functionality of my-zip. So, decompresses the file and prints the results into the standard output. It reconstructs the original file that was compressed from a binary format and displays it through the standard output. Since the data sent through standard output is not in binary format this time, the terminal has no difficulties interpreting and printing it. The standard output can also optionally be redirected to a text file. The utility can be used like this: `$ ./my-unzip zipped.z` or `$ ./my-unzip zip.z zip2.z > words.txt`

Testing

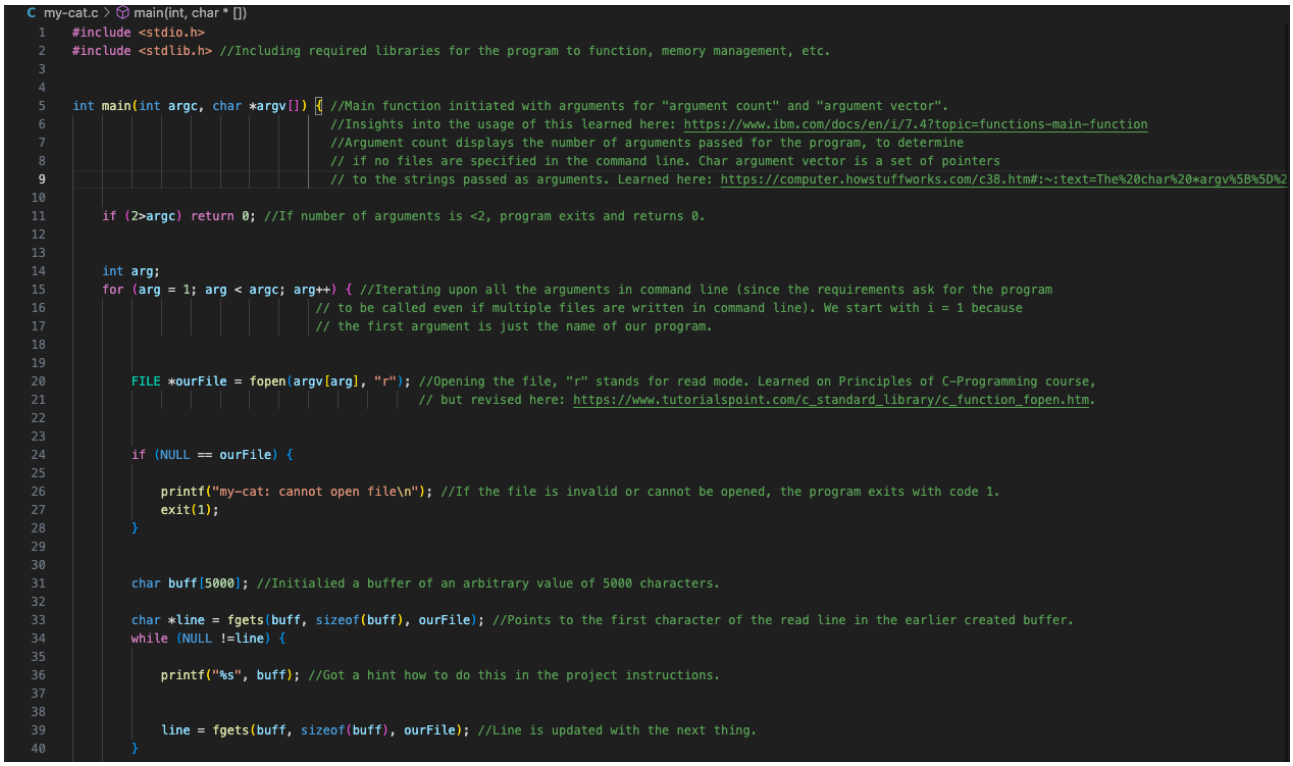
All the utilities passed all the provided tests in the project instructions. **my-cat** can be invoked with one or more files in the command line and will print out each of them. If multiple files are written in the command line, they are printed in order. For **my-grep**, the utility always passed a search term and 0 or more files, in turn going through each. The search is case-sensitive and once again, lines can be arbitrarily long. We also tested that if a search term is specified without a file, my-grep reads the standard input. If it is given an empty string "" as a search term, the program matches all lines. For **my-zip** and **my-unzip**, the format of compressed files matches the project instructions exactly, so a 4-byte integer and then an ASCII character. Also, if multiple files are given to my-zip, a single compressed file is formed. And when the file is uncompressed, the contents are given in a single string of text without any indication of boundaries that were present in the original files.

Error handling

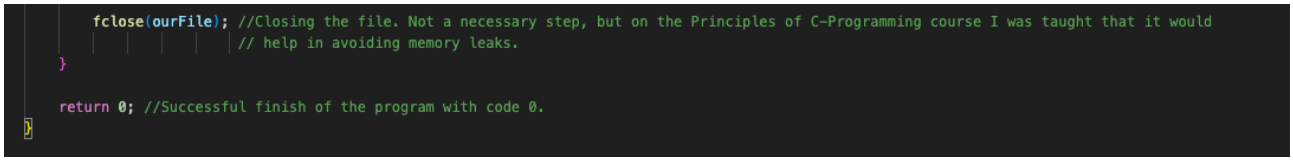
All error handling instructions were also followed. For **my-cat**, in non-error cases exists with status code 0. If no files are specified in the command line, my-cat exits with code 0. If the program tries to open a file unsuccessfully, it prints "my-cat: cannot open file\n". For **my-grep**, if it was not given any arguments, it prints "my-grep: searchterm [file ...]\n" with exit code 1. But for successful cases, it exits with code 0. For **my-zip** and **my-unzip**, during a correct call, one or more files are passed to the utility. If not, the program exits with code 1 and prints "my-zip: file1 [file2 ...]" or "my-unzip: file1 [file2 ...]" .

Source code screenshots

my-cat:



```c
#include <stdio.h>
#include <stdlib.h> //Including required libraries for the program to function, memory management, etc.


int main(int argc, char *argv[]) { //Main function initiated with arguments for "argument count" and "argument vector".
                                   //Insights into the usage of this learned here: https://www.ibm.com/docs/en/i/7.4?topic=functions-main-function
                                   //Argument count displays the number of arguments passed for the program, to determine
                                   // if no files are specified in the command line. Char argument vector is a set of pointers
                                   // to the strings passed as arguments. Learned here: https://computer.howstuffworks.com/c38.htm#:~:text=The%20char%20*argv%5B%5D%2

    if (2>argc) return 0; //If number of arguments is <2, program exits and returns 0.


    int arg;
    for (arg = 1; arg < argc; arg++) { //Iterating upon all the arguments in command line (since the requirements ask for the program
                                       // to be called even if multiple files are written in command line). We start with i = 1 because
                                       // the first argument is just the name of our program.


        FILE *ourFile = fopen(argv[arg], "r"); //Opening the file, "r" stands for read mode. Learned on Principles of C-Programming course,
                                               // but revised here: https://www.tutorialspoint.com/c_standard_library/c_function_fopen.htm.


        if (NULL == ourFile) {

            printf("my-cat: cannot open file\n"); //If the file is invalid or cannot be opened, the program exits with code 1.
            exit(1);
        }


        char buff[5000]; //Initialied a buffer of an arbitrary value of 5000 characters.

        char *line = fgets(buff, sizeof(buff), ourFile); //Points to the first character of the read line in the earlier created buffer.
        while (NULL !=line) {

            printf("%s", buff); //Got a hint how to do this in the project instructions.


            line = fgets(buff, sizeof(buff), ourFile); //Line is updated with the next thing.
        }
```



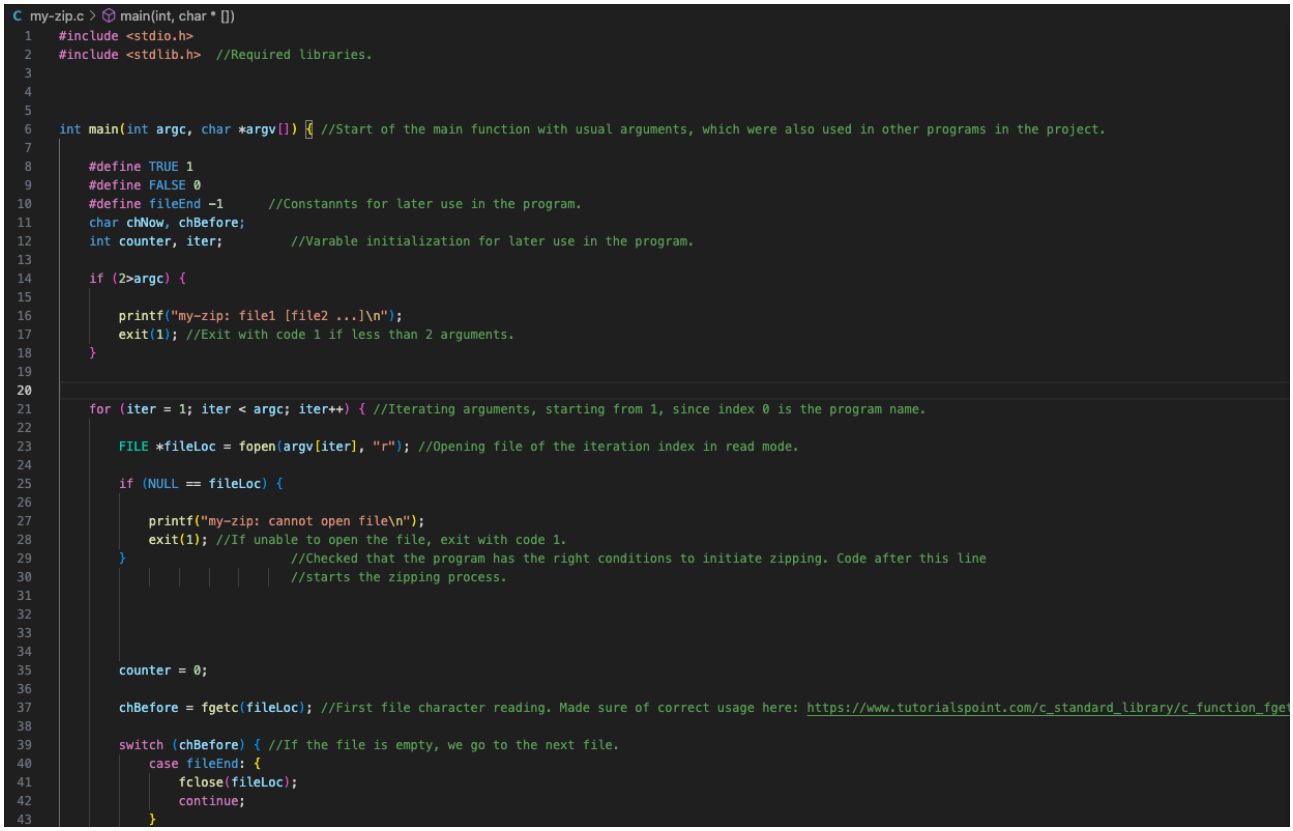
```c
        fclose(ourFile); //Closing the file. Not a necessary step, but on the Principles of C-Programming course I was taught that it would
                         // help in avoiding memory leaks.
    }

    return 0; //Successful finish of the program with code 0.
}
```

my-grep:

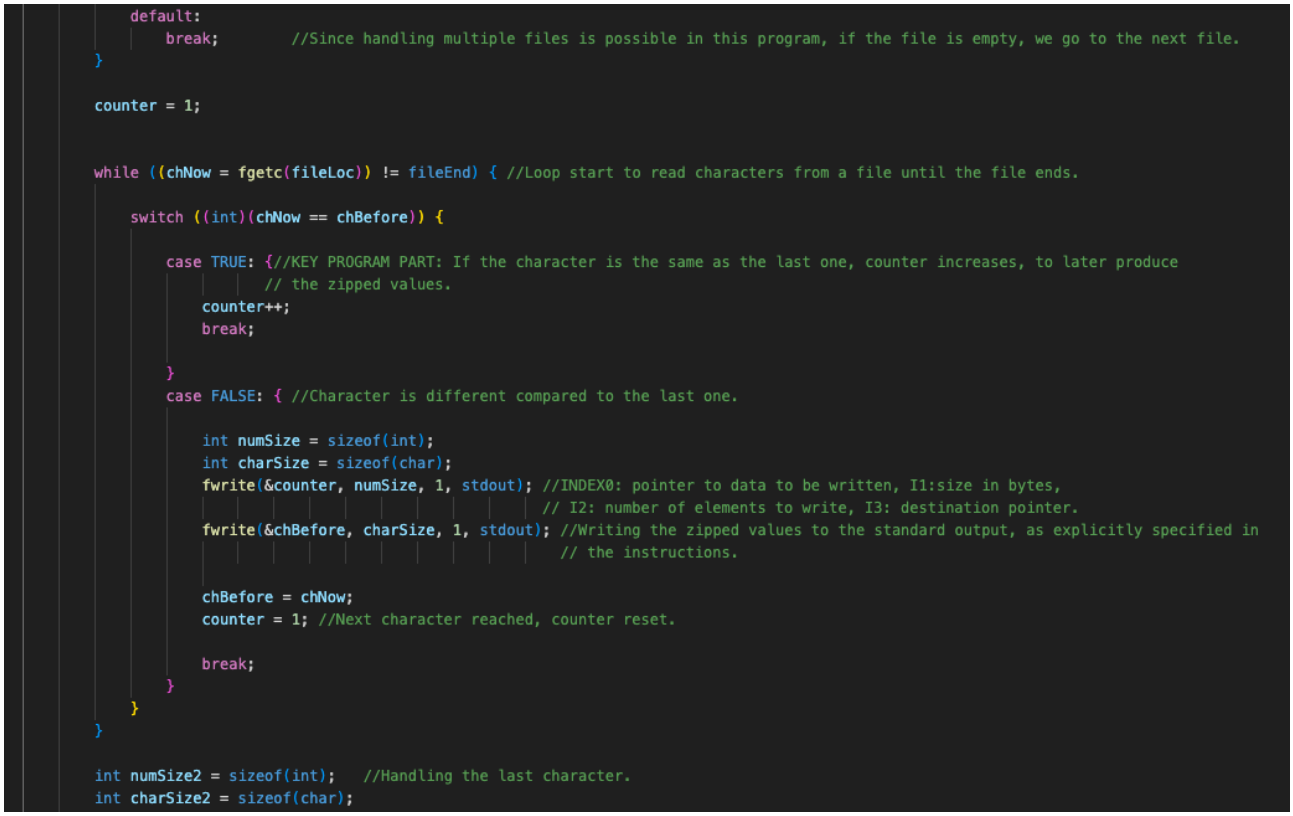
```c
#include <stdio.h>
#include <stdlib.h> //Including required libraries.

#include <string.h> //Found this library to search for substrings in a string (strstr). Found here: https://www.geeksforgeeks.org/c-library-string-h/.


void grepAlgorithm(char *searchVar2, FILE *userInp2) { //The actual algorithm for my-grep. Takes the search variable, and user input in case if no file specified,
                                                       // or the specified file name.

    char *singLine = NULL;                             //Pointer of the arbitraly long line.
    size_t singLineLen = 0;                            //After trying to find how to implement the func                arbitrary long
                                                       // lines, it was understood that its possible to                to achieve the
                                                       // result. This website helped: https://pvs-studio.com/en/blog/terms/0044/#:~:text=size_t~,Feb%2006%202023,pos


    while (-1 != getline(&singLine, &singLineLen, userInp2)) { //Reads stdin or file line by line in a loop.
                                                               //THe -1 loop exit condition was learned here: https://www.ibm.com/docs/en/zos/3.1.0?topic=functions-g

        if (NULL != strstr(singLine, searchVar2)) { //Returns pointer to first occurence of searched string, or NULL if absent. Got the ide of this
                                                    // kind of implementation here: https://www.ibm.com/docs/en/i/7.4?topic=functions-strstr-locate-substring.

            printf("%s", singLine); //Printing out the buffer with the same way that was recommended in the instructions for my-cat.c.
        }
    }
}



int main(int argc, char *argv[]) { //Similarly to my-cat.c, main function has arguments for argument number and vector.
                                   //(https://www.ibm.com/docs/en/i/7.4?topic=functions-main-function)



    if (2>argc) {
        printf("my-grep: searchterm [file ...]\n");

        exit(1);                //If there are less than 2 arguments, exits with code 1 and shows user how the program should be used.
    }



    char *searchVariable = argv[1];

```

```c
    if (2 == argc) {              //Check whether there are precisely 2 arguments.
        printf("Type your text. Press enter to search for the keyword in the input. (Ctrl+C to finish):\n");

        FILE *userInput = stdin; //If 2 argumentss, read from stdin.
        grepAlgorithm(searchVariable, userInput); //Call grep search algorithm with search term and user input as arguments.


    } else { //If there are more than 2 arguments, read from given files.

        int iter;
        int ArgcNum = argc; //Initializing iteration variable and number of arguments.

        for (iter = 2; iter < ArgcNum; iter++) { //Iterating through arguments, starting from 2 because argument 0 is program name, argument 1
                                                 // is search term, and starting from argument 2 are file names.

            FILE *currFile = fopen(argv[iter], "r"); //Opening file in read mode, as in my-cat.


            if (NULL == currFile) {

                printf("my-grep: cannot open file\n");

                exit(1);                            //If a file cannot be opened, informing the user and exiting with code 1.
            }

            grepAlgorithm(searchVariable, currFile); //Calling grep algorithm with search term and file name arguments. Since it is in a for- loop,
                                                     // it will iterate through all given file names and print resutls.



            fclose(currFile);                       //Closing the file.
        }

    }

    return 0;                                       //In all successful cases, program exits with code 0, as said in the instructions.
}
```

my-zip:



```c
#include <stdio.h>
#include <stdlib.h>  //Required libraries.



int main(int argc, char *argv[]) { //Start of the main function with usual arguments, which were also used in other programs in the project.

    #define TRUE 1
    #define FALSE 0
    #define fileEnd -1      //Constannts for later use in the program.
    char chNow, chBefore;
    int counter, iter;         //Varable initialization for later use in the program.

    if (2>argc) {

        printf("my-zip: file1 [file2 ...]\n");
        exit(1); //Exit with code 1 if less than 2 arguments.
    }


    for (iter = 1; iter < argc; iter++) { //Iterating arguments, starting from 1, since index 0 is the program name.

        FILE *fileLoc = fopen(argv[iter], "r"); //Opening file of the iteration index in read mode.

        if (NULL == fileLoc) {

            printf("my-zip: cannot open file\n");
            exit(1); //If unable to open the file, exit with code 1.
        }                       //Checked that the program has the right conditions to initiate zipping. Code after this line
                                //starts the zipping process.




        counter = 0;

        chBefore = fgetc(fileLoc); //First file character reading. Made sure of correct usage here: https://www.tutorialspoint.com/c_standard_library/c_function_fge

        switch (chBefore) { //If the file is empty, we go to the next file.
            case fileEnd: {
                fclose(fileLoc);
                continue;
            }
```



```c
            default:
                break;        //Since handling multiple files is possible in this program, if the file is empty, we go to the next file.
        }

        counter = 1;


        while ((chNow = fgetc(fileLoc)) != fileEnd) { //Loop start to read characters from a file until the file ends.

            switch ((int)(chNow == chBefore)) {

                case TRUE: {//KEY PROGRAM PART: If the character is the same as the last one, counter increases, to later produce
                           // the zipped values.
                    counter++;
                    break;

                }
                case FALSE: { //Character is different compared to the last one.

                    int numSize = sizeof(int);
                    int charSize = sizeof(char);
                    fwrite(&counter, numSize, 1, stdout); //INDEX0: pointer to data to be written, I1:size in bytes,
                                                          // I2: number of elements to write, I3: destination pointer.
                    fwrite(&chBefore, charSize, 1, stdout); //Writing the zipped values to the standard output, as explicitly specified in
                                                            // the instructions.

                    chBefore = chNow;
                    counter = 1; //Next character reached, counter reset.

                    break;
                }
            }
        }

        int numSize2 = sizeof(int);   //Handling the last character.
        int charSize2 = sizeof(char);
```

```
        fwrite(&counter, numSize2, 1, stdout);
        fwrite(&chBefore, charSize2, 1, stdout); //With fwrite() we write the integer value as a binary value and the character as ASCII in the
                                                 // standard output.

        fclose(fileLoc); // Closing the file.
    }

    return 0; // Exit successfully, program ends with code 0.
}
```

my-unzip:

```
C my-unzip.c > main(int, char * [])
#include <stdio.h>
#include <stdlib.h> //Added required libraries.



int main(int argc, char *argv[]) { //Start of the main function with usual arguments, which were also used in other programs in the project.


    if (2>argc) {

        printf("my-unzip: file1 [file2 ...]\n");
        exit(1); //Exit with code 1 if less than 2 arguments.
    }

    int iter;
    for (iter = 1; argc> iter; iter++) { //Argument iteration loop (index 0 is just the program name).

        FILE *currFilePointer = fopen(argv[iter], "rb"); //Opening the file, although this time in read binary mode.

        if (NULL == currFilePointer) {

            printf("my-unzip: cannot open file\n");
            exit(1); //If unable to open the file, exit with code 1.
        }



        int i;
        char character;
//fread usage, as recommended from project instructions learned here https://www.geeksforgeeks.org/fread-function-in-c/.

        while (fread(&i, sizeof(int), 1, currFilePointer) == 1) { //The first loop that reads the integer values stored as binary in the file.
                                                                  //=1 means that exactly one integer was successfully read.

            if (fread(&character, sizeof(char), 1, currFilePointer) != 1) {
                // Read the character value, at the same time handling error if character cannot be read

                printf("Unable to read the character.\n");

                break;
            }
```

```
            int j;
            for (j = 0; j < i; j++) { //Loop for printing a character i times, since the consecutive number of the character is specified
                                      // by the number.

                printf("%c", character);//Printing the character i times. Integer is not printed because it just shows the number of consecutive
                                        // characters.
            }
        }



        fclose(currFilePointer);//Closing the file after it was read.
    }

    return 0; //Successful program run, end with code 0.
}
```