Assignment: Command-Line Arguments, Binary File I/O, and Mixing C and C++

Learning Outcomes

This assignment will provide you with the knowledge and practice required to develop and implement software involving:

1. Demonstrate ability to process command-line parameters.
2. Demonstrate ability to process binary files.
3. Demonstrate ability to build program using source files implemented in both C and C++

Specs

Your task is to develop an application that allows the user to

- Split a file into smaller pieces with the size specified by the user.
- Join these smaller pieces into a single file.
- Use command-line switches to use the same program to implement the splitting and joining processes.

Splitting

Imagine a scenario in which a user must transfer a file `IN` of size 542 MB ($549{,}453{,}824$ bytes) between two computers using a 128 MB ($134{,}217{,}728$ bytes) USB drive. The file can be reliably transferred by splitting the file into chunks smaller than or equivalent to 128 MB and then using the USB drive to transfer each chunk from the source to destination computer. After transferring all the files, the smaller chunks can be joined in the destination computer to create the original file.

Suppose in the source computer, the input file `IN` is located in directory `./data`. The user should be able to call your program `sj.out` to split `IN` into chunks stored in directory `./split-data` by issuing the following command in the shell:

```
$ ./sj.out -s 134217728 -o ./split-data/piece_ -i ./data/IN
```

The command-line switches used by the above command have the following meaning:

- `-s` `chunk-size` means split file into chunks with each chunk having size `chunk-size`. This means that the 542 MB file `IN` is split into five chunks: the first four chunks will have size $134{,}217{,}728$ bytes (128 MB) while the fifth chunk will have size $31{,}457{,}280$ bytes (30 MB).
- `-o` `path-to-chunk-file-name` specifies the full pathname of the chunks created by splitting the input file. In the above example, `./split-data` specifies the directory [assume the directory exists] while `piece_` specifies that each of the 5 chunks will be named `piece_0001`, `piece_0002`, `piece_0003`, `piece_0004`, and `piece_0005`. Note that the output chunks are labeled from `piece_0001` and not from `piece_0000`.
- `-i` `input-file-name` specifies the full pathname of the input file to be split. In the above example, `./data` specifies the directory name [assume it exists] while `IN` specifies the file to be split into smaller chunks.

Joining

Suppose the user has transferred chunks named `piece_0001`, `piece_0002`, `piece_0003`, `piece_0004`, and `piece_0005` in directory `./split-data` to the destination computer. The user should now be able to call your program `sj.out` to combine these chunks into a new file `OUT` in directory `./jointed-data` [assuming the directory exists] by issuing the following command:

```
$ ./sj.out -j -o ./joined-data/OUT -i ./split-data/piece_0001 ./split-data/piece_0002 ./split-data/piece_0003 ./split-data/piece_0004 ./split-data/piece_0005
```

The command-line switches used in the joining process have the following meaning:

- `-j` means join chunks into a single file
- `-o` `path-to-output-file-name` specifies the full pathname of output file. In the above example, the full pathname `./joined-data/OUT` specifies `./joined-data` as the directory name [assume the directory exists] in which the merged file `OUT` is created by joining the smaller chunks.
- `-i` `input-file-name(s)` specifies the full pathname(s) of input chunks that must be merged. In the above example, `./split-data` specifies the directory name [assume it exists] containing the chunks `piece_001`, `piece_002`, and so on that are to be merged.

The size and contents of merged file `OUT` must be exactly equivalent to original file `IN` - you can check there are no differences between using command `diff`:

```
$ diff ./joined-data/OUT ./data/IN
```

Command-line switches

The entire list of command-line switches from splitting and joining [described above] are collected here:

- `-j` means join chunks into a single file
- `-s` `chunk-size` means split file into desired chunk size `chunk-size`
- `-o`
  - Name of joined file when used in conjunction with `-j`
  - Prefix used to name chunks when used in conjunction with `-s`. The suffix of each chunk file is a 4 digit number padded with 0's starting with `0001`.
- `-i` means input filename(s) for splitting or joining
  - `-i` `input-file-name` means name of file that must be split into smaller chunks when used in conjunction with `-s`
  - `-i` `input-file-name(s)` means filename(s) of input chunks that must be joined when used in conjunction with `-j`
  - The operating system will expand wild cards for you. That is, if you issue this command:

    ```
    $ ./sj.out -j -o ./joined-data/collected -i ./split-data/chunk*
    ```

and directory split-data contains files `chunk1`, `chunk2`, `chunk3`, and `chunk4`, then argument array `argv[]` will contain the following null-terminated strings:

```
./sj.out
-j
-o
./joined-data/collected
-i
./split-data/chunk1
./split-data/chunk2
./split-data/chunk3
./split-data/chunk4

```

Implementation details

1. You must define function `split_join`:

   ```
   SplitResult split_join(int argc, char *argv[]);
   ```

   in C source file `splitter.c` [using C standard library] and C++ source file `splitter.cpp` [using C++ standard library but not C standard library]. Rather than implementing the entire solution in `split_join`, it is recommended [but not required since the driver is only calling `split_join`] that you decompose the problem into functions that individually handle the following tasks: parsing command-line parameters, splitting a file into smaller chunks, and joining smaller chunks into a single file.
2. I've implemented both C [`driver.c`] and C++ [`driver.cpp`] versions of function `main`. Both provide a straightforward implementation of function `main`: the command-line parameters are passed to a function `split_join` that you must implement in source files `splitter.c` and `splitter.cpp`. The driver queries the enumeration value returned by `split_join` to determine the result of the split or join actions which is then printed to standard output.

   > ***Your implementation of function `split-join` must not print any characters to either standard output or standard error stream.***

3. Implement your code in `splitter.c` using the C programming language with `FILE*` based binary I/O only using C standard library functions `fread` and `fwrite` to read from and write to files.
4. Implement your code in `splitter.cpp` using the C++ programming language and C++ standard library but not C standard library. Your code will not compile if you use functions `fread` and `fwrite` from the C standard library.
5. You're also given ***incomplete*** interface file `splitter.h` that [currently] doesn't allow C and C++ code to be mixed:

   ```
   #ifndef SPLITTER_H
   #define SPLITTER_H

   typedef enum {
     E_BAD_SOURCE = 1,
     E_BAD_DESTINATION,
   ```

```c
  E_NO_MEMORY,
  E_SMALL_SIZE,
  E_NO_ACTION,
  E_SPLIT_SUCCESS,
  E_JOIN_SUCCESS
} SplitResult;

SplitResult split_join(int argc, char *argv[]);

#endif
```

As part of your submission, you must provide a complete implementation of this header that will facilitate mixing of C and C++ functions.

6. Using combinations of C and C++ source files, your submission must allow creation of these 4 executables:
   - C executable [linked using `gcc`] with both `driver.c` and `splitter.c` compiled using `gcc`.
   - C++ executable [linked using `g++`] with both `driver.cpp` and `splitter.cpp` compiled using `g++`.
   - C++ executable [linked using `g++`] with `driver.cpp` compiled using `g++` and `splitter.c` compiled using `gcc`.
   - C++ executable [linked using `g++`] with `driver.cpp` compiled using `g++` and `splitter.c` compiled using `g++`.
7. Scoped enumeration type `SplitResult` keeps track of the results of splitting a file or joining chunks into a single file. The values returned by `split_join` are:
   1. `E_NO_ACTION`: If user input is not sufficient [number of command-line parameters is not sufficient or `-j` switch is
      missing or `-s` switch is missing].
   2. `E_BAD_SOURCE`: If input file for either split or join doesn't exist.
   3. `E_BAD_DESTINATION`: If output file for either split or join cannot be created.
   4. `E_SMALL_SIZE`: If byte size of split files is negative or zero.
   5. `E_NO_MEMORY`: If `malloc` returns `NULL` [only in `splitter.c`].
   6. `E_SPLIT_SUCCESS`: If a file has been successfully split into smaller chunks.
   7. `E_JOIN_SUCCESS`: If chunks have been successfully merged into a single file.
8. To prevent clutter, assume data files are stored in directory `./data`, split files are stored in directory `./split-data`, while joined files are stored in directory `./joined-data`:

```
$ ./sj.out -s 134 -o ./split-data/piece_ -i ./data/IN
$ ./sj.out -j -o ./joined-data/collected -i ./split-data/chunk*
```

9. When reading from files, specify a read buffer size that is set to the minimum of desired chunk size [given through flag `-s`] and identifier `FOUR_K` [which is a macro in `splitter.c` and a read-only `int` variable in `splitter.cpp`] which is specified as 4096. This means that if the user specifies a chunk size of 10000 bytes, the program will use a read buffer size of 4096 bytes.

> ***Do not change value of `FOUR_K` since the online grader has been set up to allocate up to but not more memory from the free store.***

To create chunks of size 10000 bytes, the program must read 4096 bytes [the first time] from the input file into its read buffer and then write these 4096 bytes to the output chunk, read 4096 bytes [the second time] from the input file into its read buffer and then write these 4096 bytes to the output chunk, and read the remaining 1808 bytes [a third and final time] from the input file into its read buffer and then writes 1808 bytes to the output chunk. That is, to create a chunk of size 10000 bytes, the program must read from the input file and write to the chunk three times. In contrast, if the user specifies a chunk size of 1024 bytes, the program must use a read buffer size of 1024 bytes. To create chunks of size 10000 bytes, the input file must be read 10 times. Not following this recipe [by increasing the read buffer size] to a higher value will mean that your submission is not following the specified requirements and will receive a zero grade!!!

> ***Test your implementations with various values for `FOUR_K` such as 1024, 2048, and so on.***

> ***All buffers used in `split_join` must be dynamically allocated dynamically using function `malloc` [in `splitter.c`] and `new[]` [in `splitter.cpp`].***

10. The suffix of each chunk file is a 4 digit number padded with 0's. Suppose the following command creates 3 chunks:

```
$ ./sj.out -s 134 -o ./split-data/piece_ -i ./data/IN
```

Your splitter must name these chunks as follows: `piece_0001`, `piece_0002`, and `piece_0003`.

> ***Your submission will not receive a grade if your splitter/joiner doesn't name chunks [as described above] using a 4 digit suffix.***

11. The operating system will expand wild cards for you. That is, if you issue the command:

```
$ ./prog.exe -j -o ./joined-data/collected -i ./split-data/chunk*
```

and directory `split-data` contains files `chunk0001`, `chunk0002`, `chunk0003`, and `chunk0004`, then argument array `argv[]` will contain the following null-terminated strings:

```
./prog.exe
-j
-o
./joined-data/collected
./split-data/chunk0001
./split-data/chunk0002
./split-data/chunk0003
./split-data/chunk0004
```

12. Ensure your code compiles and links without generating any warnings or errors.
13. The online server will execute your submission through Valgrind. If Valgrind detects memory-related problems with your submission, the diagnostic output generated by Valgrind will be added to your program's output causing that output to be different than the correct output. This will obviously result in your submission being graded as incorrect.

Makefile

You'll need to generate 4 executables [described earlier] using combinations of `driver.c`, `driver.cpp`, `splitter.c`, and `splitter.cpp`. Rather than manually typing the commands to generate these six targets, use `make` to determine automatically which source files need to be recompiled, and issue the commands to recompile them and then create the appropriate executable. Use the example makefile [from the review section of the course web page] to describe which object files are used to create an executable target, and how the object files are created from corresponding source files.

First Steps

1. Begin by copying files in directory `source` into your parent directory.
2. Begin with either `splitter.c` or `splitter.cpp`. In that source, parse command-line switches passed to function `split_join` to determine whether to split or join. Implement the code for splitting a file into multiple chunks in function `split_join` and the code for joining multiple files in function `split_join`.
3. Use `make` to compile and link necessary source files to create the appropriate executable, say `sj1.out`
4. Create directories `split-data` and `joined-data`.
5. Use the generated executable `sj1.out` to initiate the splitting process:

   ```
   $ ./sj1.out -s 14 -o ./split-data/chunk_ -i ./data/test1
   ```

   If successful, the driver will print the following text to standard output stream:

   ```
   Split successfully completed

   ```

   Directory `split-data` will now contain the following chunk of files:

   ```
   ./split-data/chunk_0001
   ./split-data/chunk_0002
   ./split-data/chunk_0003
   ./split-data/chunk_0004
   ./split-data/chunk_0005
   ...
   ```

6. Initiate the joining process:

   ```
   $ ./sj1.out -j -o ./joined-data/OUT -i ./split-data/chunk_*
   ```

   If successful, the driver will print the following text to standard output stream:

   ```
   Join successfully completed

   ```

   Directory `joined-data` will now contain merged file `OUT`:

7. Use command `diff` to compare original file `./data/IN` and merged file `./joined-data/OUT`:

   ```
   $ diff ./joined-data/OUT ./data/IN
   ```

8. Test splitting and joining process on other binary files.
9. Complete the interface in `splitter.h` so that C function `split_join` can be used with both C and C++ compilers.
10. Once you've thoroughly tested the current splitter/joiner, implement the second source file.

Useful Tools

`od` and `diff` are available in Linux and are useful for testing your program.

- `od -x filename`

  `od` dumps the contents of a file in octal format by default.The output will look like this:

  ```
  0000000    6923  6e66  6564  2066  4946  454c  414e  454d
  0000020    485f  3590  0000  1293  4a2c  4090  0900  5100
  0000040
  ```

  The left column specifies octal offsets in the file while the remaining columns specify the two-byte contents of these addresses in hexadecimal format.

- `diff file1 file2`

  `diff` compares files `file1` and `file2` line by line. Let both files have the same first line with the second line in `file1` and `file2` containing a single character `a` and `b`, respectively. The output of the `diff` command will look like this:

  ```
  2c2
  < a
  ---
  > b
  ```

  showing that there is a difference in line 2 between the two files with the first file containing `a` while the second file is containing `b`.

Documentation is required

This module will use Doxygen to tag source and header files for generating html-based documentation. Every source and header file *must* begin with *file-level* documentation block. Every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value. The course web page provides more coverage on this topic.

Submission and Grading Rubrics

1. In the course web page, click on the appropriate submission page to submit necessary files.
2. F grade if your submission doesn't compile with the full suite of `g++` options or if your submission compiles but doesn't link to create an executable.

3. F grade if Valgrind detects even a single memory leak or error.
4. A deduction of one letter grade for each missing documentation block in your submission(s). Every source or header file you submit must have **one** file-level documentation block and function-level documentation blocks for ***every function you're declaring [in the header file]***. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to E.
5. Your implementation's output doesn't match correct output of the grader [you can see the inputs and outputs of the auto grader's tests]. The auto grader will provide a proportional grade based on how many incorrect results were generated by your submission. $A+$ grade if output of function matches correct output of auto grader.