Assignment 2 [Command-Line Arguments and Binary File I/O]

Topics

1. Command-line arguments
2. Binary file input and output using the C++ programming language

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.

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

128 128 128 128 128 30

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:

outputs to

```
$ ./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 prefix used to name smaller chunks. This means the 5 chunks are stored in directory `split-data` (assume the directory exists) and 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` means name of file to be split into smaller chunks. This means that file `IN` located in directory `data` (assume it exists) will be split into smaller chunks.

Joining

Suppose the user has transferred the 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 prefix used to name output file.
- `-i` `input-file-name(s)` means filename(s) of input chunks that must be joined

The size of merged file `OUT` must be exactly equivalent to sum of sizes of chunks `piece_0001`, `piece_0002`, `piece_0003`, `piece_0004`, and `piece_0005`. You can check there are no differences between original file `IN` and merged file `OUT` using `diff` command:

```
$ 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. Assume C++ and binary I/O using `<fstream>`. Submissions relying on `FILE*` based I/O using `<cstdio>` will be rejected by the online grader.
2. You're given `driver.cpp` that provides a straightforward implementation of function `main`: it passes the command-line parameters to a function `split_join` that you must implement in source file `splitter.cpp`. The driver queries the enumeration value returned by `split_join` to determine the result of the split or join actions which are then printed to standard output.
3. You're also given interface file `splitter.h`

```
#ifndef SPLITTER_HPP
#define SPLITTER_HPP

namespace CSD2125 {

inline constexpr int MAX_SPLIT_SIZE{4096}; // don't change

enum class SplitResult {
  E_BAD_SOURCE,
  E_BAD_DESTINATION,
  E_NO_MEMORY,
  E_SMALL_SIZE,
  E_NO_ACTION,
  E_SPLIT_SUCCESS,
  E_JOIN_SUCCESS
};

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

}

#endif
```

   that contains:

   1. Definition of constant `MAX_SPLIT_SIZE` that specifies the maximum size of a chunk when splitting a file. This value will also determine the maximum number of contiguous bytes that you can allocate from the free store using `new[]`.

      > *Do not change this value since the online grade has been set up to allocate up to but not more memory from the free store.*

When splitting and/or reading from files, specify a read buffer size that is set to the minimum of the desired chunk size (given through flag `-s`) and constant `MAX_SPLIT_SIZE`. That is, if the user specifies a chunk size of 8197 bytes and if constant `MAX_SPLIT_SIZE` is initialized with value `4096`, the program must use a read buffer size of 4096 bytes. And, if the user specifies a chunk size of 1024 bytes, the program must use a read buffer size of 1024 bytes.

> *All buffers used in `split_join` must be allocated dynamically using `MAX_SPLIT_SIZE`. Note that the online grade may use a different value than the value used to initialize `MAX_SPLIT_SIZE` in the interface file shared with you.*

2. A scoped enumeration class `SplitResult` to keep track of the results of splitting a file or joining chunks into a single file. Read the driver to understand the purpose of these enumeration constants in relation to the values returned by function `split_join`. Some of the values returned by `split_join` are:
   1. `E_NO_ACTION`: If user input is not sufficient (the 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 the byte size of split files is negative or zero.
   5. `E_NO_MEMORY`: If `new[]` throws an exception.
   6. `E_SPLIT_SUCCESS`: If a file has been successfully split into smaller chunks.
   7. `E_JOIN_SUCCESS`: If chunks have been successfully combined into a single file.
3. Declaration of function `split_join`:

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

   that you must implement in source file `splitter.cpp`. 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.
4. ***Assume binary file I/O since your implementations will only be tested with binary files.***

Tests

1. I've provided a sample that can be used to verify the following tests.
2. Test split using files `120-byte-file` and `200-byte-file`. File `120-byte-file` consists of character `1` repeated 119 times followed by a line break character which in Linux is the newline character `\n` having hex value `0x0a`. File `200-byte-file` contains character `1` repeated 199 times followed by a Linux line break character. The command

   ```
   $ ./sample.out -s 100 -o ./split-data/test120_ -i ./data/120-byte-file
   ```

   should produce 2 chunks of data `test120_0001` and `test120_0002` having sizes 100 and 20 bytes, respectively.

   Similarly, command

```
$ ./sample.out -s 100 -o ./split-data/test200_ -i ./data/200-byte-file
```

should produce 2 chunks `test200_0001` and `test200_0002` each having size 100 bytes.

3. Next, test join:

   ```
   $ ./sample.out -j -o ./joined-data/new-120-byte-file -i ./split-data/test120_*
   ```

   Use `diff` to ensure that file `new-120-byte-file` obtained after joining the split files is exactly similar to original file `120-byte-file`.

   Repeat joining process and subsequent test for split files `./split-data/test200_*`.

4. Test splitting and joining process on a pdf file. Ensure that you can open and read the pdf file obtained by combining the split files.
5. Finally, test your implementation with a zip 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.

Valgrind is required

Since your code is interacting directly with physical memory, things can go wrong at the slightest provocation. The range of problems that can arise when writing code dealing with low-level details has been covered in HLP1 and HLP2 lectures. You should use Valgrind to detect any potential issues that may lurk under the surface. 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 `splitter.cpp`.
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. A maximum of D grade if Valgrind detects even a single memory leak or error. A teaching assistance will check you submission for such errors.
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 defining***. 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.