CPSC 457 Spring 2021 - Assignment 2

Due date is posted on D2L.
Individual assignment. Group work is NOT allowed.
Weight: 19% of the final grade.

In this assignment you will write C++ code that uses many different system calls, as well as some standard C++ containers.

Overall description

You will start with an incomplete C++ program that contains an unfinished `getDirStats()` function, as well as other helpful functions and classes. Your job is to finish the program by implementing the `getDirStats()` function. This function must recursively examine the contents of a directory and then report some statistics about the files and subdirectories it discovers. The function will take a directory name and a number N as input, and it will compute and return the following results:

- Full path of the largest file and its size in the directory;
- The total number of files and sub-directories;
- The sum of all file sizes;
- A list of N most common file types and their respective counts;
- A list of N most common words inside the files and their respective counts;
- A list of N largest groups of duplicate files.

Please note that you do not have to write recursive code – but you do need to write code that discovers all subdirectories, sub-subdirectories, sub-sub-subdirectories, etc.

First step

Start by cloning the code from the GitLab repository, then compile the code using included `Makefile`.

```
$ git clone https://gitlab.com/cpsc457/public/dirstats.git
$ cd dirstats
$ make
```

This will create an executable `dirstats`. Then run the executable on the current directory `"."` with `N=5`. Please note that the executable will produce incorrect results because the program is incomplete.

```
$ ./dirstats 5 .
```

The `main()` function, defined in `main.cpp`, parses the command line arguments, then calls the unfinished `getDirStats()` function, and finally prints writes the results to standard output. You should not modify this function at all. You may only modify the `getDirStats.cpp` file, and you must not modify any other files.

The `getDirStats()` function

The `getDirStats()` is declared in `getDirStats.h` and defined in `getDirStats.cpp`:

```
Results getDirStats(const std::string & dir_name, int n);
```

The parameters are:

- `dir_name` is the top directory that your function must examine recursively; and
- `n` is the number of most frequent file-types, words and number of groups that the function will report.

You need to write code that recursively traverses the directory, and calls the `stat()` function on all encountered files. You then extract some information from the returned stat structure, e.g. size, and update the relevant statistics. Your TAs should cover how to recursively examine a directory during tutorials. You can also use the code from https://gitlab.com/cpsc457/public/find-empty-directories as a starting point.

The function returns an instance of `struct Results`, which is a defined in `getDirStats.h`. It has a number of fields, which you need to populate before returning it. These are described below:

```
bool valid;
```

If your function is able to finish processing all directories and files and calculates all remaining fields, set `valid=true`. If some of the files or directories could not be opened, set `valid=false`, and the rest of the fields do not have to contain any values.

```
std::string largest_file_path;
long largest_file_size;
```

You will use `largest_file_path` and `largest_file_size` to return a path and the size of the largest file discovered during your recursive traversal. The path must start with the value in `dir_name`. If no files are discovered, set `largest_file_path` to an empty string, and `largest_file_size` to `-1`.

```
long n_files, n_dirs;
```

`n_files` will contain the total number of files encountered while scanning the directory. `n_dirs` will contain the total number of sub-directories encountered while scanning the directory, not including the top directory. It is possible for both to be `0`.

```
long all_files_size;
```

Will contain the sum of all sizes of all files encountered during the scan. Hint: use `stat()` to determine file sizes for each file, then sum them up.

```
std::vector<std::pair<std::string, int>> most_common_types;
```

Will contain a list of N most common file types encountered, together with the number of occurrences of the file type. The list will be sorted by the number of occurrences, in descending order.

You will need to use `popen()` to call UNIX `file(1)` utility with the `-b` option to obtain the file type of the file. Your code will need to parse the output of `file(1)` and only consider the main type (the text before the first comma). Once you have the file types for all files, you will need to find the N most common types. There are many ways to do this, but I suggest using `std::unordered_map<std::string,int>` and `std::sort()` to make your life easy.

I created a sample program to illustrate how to use `std::unordered_map` to create a histogram of strings, and how to extract the top N entries from it using 2 different approaches. You can find it here: https://gitlab.com/cpsc457/public/word-histogram. Feel free to re-use this code in your solutions.

```
std::vector<std::pair<std::string, int>> most_common_words;
```

Will contain a list of N most common words inside all files, converted to lower-case, together with the number of occurrences of each word. The list will be sorted by the number of occurrences, in descending order.

A word is a **sequence of 3 or more alphabetic characters** (lower- and upper-case letters). For example, the following string `"My name is Pavol, my password: is abc1ab2ZYZ"` contains the words `"name"`, `"pavol"`, `"password"`, `"abc"` and `"zyz"`.

You need to open and read the contents of every file you find and extract the words from the file contents. You need create and maintain a histogram data structure as you extract the words. Look at the word-histogram example above for motivation, but please note that the definition of word for this assignment is different (min. 3 letter length, only alphabetic characters, converted to lower-case).

```
std::vector<std::vector<std::string>> duplicate_files
```

Will contain a list of N largest groups of duplicate files, sorted in descending order by the size of the groups. See below for more details.

Computing duplicate_files

You could try to compare the contents of every file to every other file, but this would be slow, $O(n^2)$!!! A better way, $O(n \log n)$, is to compute a hash of every file, sort the hashes, and then discover duplicates by detecting the same consecutive hashes. To get a hash of a file you can use the provided function:

```
std::string sha256_from_file( const std::string & fname);
```

This function computes a SHA256 digest for a given file and returns it as a string. It mimics some of the functionality of the `sha256sum` UNIX utility, described in little more detail in the appendix.

Another efficient way to find duplicates would be to use a dictionary that maps digests to an array of filenames, e.g. using:

```
std::unordered_map<std::string,std::vector<std::string>>
```

Remember, the `duplicate_files` field will contain the list of N largest groups of identical files. Each group will be represented as a list of files (`std::vector<std::string>`) belonging to that group. No matter which technique you use to identify the groups of identical files, eventually you will need to sort these groups by size (e.g. using `std::sort`). Then you populate the `duplicate_files` with the largest groups such that:

- `duplicate_files.size()` will return the number of groups of duplicate files, at most N;
- `duplicate_files[0]` will contain the largest group of identical files;
- `duplicate_files[i]` will contain the i-th largest group of identical files;
- `duplicate_files[i].size()` will return the number of duplicate files in the `i`-th group;
- `duplicate_files[i][j]` will contain the filepath of the `j`-th file in the `i`-th group of duplicates. Please make sure that all paths must start with the value in `dir_name`.

For example, to return 2 groups of files `["d/a"," d/b"," d/c"]` and `["d/x"," d/y"]`, you could write code like this:

```
std::vector<std::string> group1;
group1.push_back("d/a"); group1.push_back("d/b"); group1.push_back("d/c");
res.duplicate_files.push_back(group1);
std::vector<std::string> group2;
group2.push_back("d/x"); group2.push_back("d/y");
res.duplicate_files.push_back(group2);
```

Important

You need to use `popen()` in your code, but only to call the `file` utility to determine a single file's type.. You must call `popen()` once for every file. You **may not** use `popen()` for any other purpose, e.g. you **cannot** use it to find all files recursively, such as: `popen("find . -type f | xargs sha256sum")`.

All code you write must go in the `getDirStats.cpp` file, and that should be the only file you will submit for grading. Your TAs will test your code by supplying their own `main()` function, which may be different from the `main()` that you will be using. It is therefore vital that you maintain the same function signature as declared in `getDirStats.h`. Before you submit `getDirStats.cpp` to D2L, make sure it works with the provided `main()` function!!!

Additional requirements:

- You may assume none of the files and directory names will contain spaces.
- The total number of directories and files will be less than 10000.
- Each full file path will contain less than 4096 characters.
- Words will have less than 1024 characters.
- If multiple file types or words have the same number of occurrences, the order in which you list them is arbitrary.
- If multiple groups of duplicate files have the same size, the order in which you list them is arbitrary.
- The order in which the files are listed per group is arbitrary.
- If multiple files have the same maximum size, arbitrarily pick one of them to report as the largest file.

Grading

Your code will be graded on correctness and efficiency. Your code should be at least as efficient as the included Python solution (described in the appendix). You should design some of your own test cases to make sure your code is correct, and to measure how fast it runs.

Allowed libraries

You are free to use the following APIs from the lib/libc++ libraries for this assignment:

- `popen()` to get the output of the `file` utility
- `stat(), opendir(), closedir(), readdir(), getcwd(), chdir()`
- `open(), close(), read(), fopen(), fread(), fclose(), fget(), fgetc()`
- `std::map, std::unordered_map, std::vector, std::string, std::sort`
- `C++ streams`

If you want to use other APIs, please ask your instructor or TA.

Not allowed

You may not use `system(3)` utility at all.

You may not call any external programs, other than `file(1)`. For example, you may not use `find`, `awk`, `grep`, `sort`, `uniq`, etc...

Submission

Submit `getDirStats.cpp` file to D2L.

Appendix 1 – computing file digests using `sha256sum(1)` utility

Cryptographic hash functions are special hash functions that have many different uses. You will be using them, namely SHA256, to efficiently detect duplicate file contents. If you have never experimented with file digests, there is a UNIX utility `sha256sum` that allows you to compute digests for files from the command line. Here is an example of how to use it from command line to get a digest of a file `1.txt`. Let's start by creating a sample file `1.txt` and computing its digest:

```
$ echo "hello" > 1.txt
$ sha256sum 1.txt
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03  1.txt
```

The long hexadecimal string is the digest of the file contents. The idea behind the digest is that it will be different for every file – as long as the file contents are different. Here is an example of a digest for a slightly modified file:

```
$ echo "world" >> 1.txt
$ sha256sum 1.txt
4a1e67f2fe1d1cc7b31d0ca2ec441da4778203a036a77da10344c85e24ff0f92  1.txt
```

As you can see, the digest is now completely different. We can use digests to compare two really big files – all we have to do is compute the digests for both files, and if the digests are different, then the file contents are definitely different. If the digests are the same, there is a very high probability that the files are the same. What is the likelihood of getting the same digests for different file contents? It is very small. So small that git uses a weaker, 128-bit version of SHA, to detect differences in files.

How does this help us detect duplicate files? Instead of comparing the contents of every single file to every other file, which would be slow, we can instead compute the digests for all files first, then sort the digests + filename pairs by the digests, and then detect duplicates by scanning the result once. You can even do this on the command line. For example, to find all duplicates in directory `/usr/include/c++` you could execute the following from the command line:

```
$ find /usr/include/c++ -type f | xargs sha256sum  sort | uniq -w64  --all-repeated=separate
...
acaa059f12b827ac1b071d9b7ce40c7043e4e65c5b4ca2169752cd81d8993356  /usr/include/c++/9/backward/hash_set
acaa059f12b827ac1b071d9b7ce40c7043e4e65c5b4ca2169752cd81d8993356  /usr/include/c++/9/ext/hash_set

c0b15be47a20948066531a4570f13fe02dbe0f7580cd8b5cad5fb172fabe9f33  /usr/include/c++/9/x86_64-redhat-linux/32/bits/gthr-default.h
c0b15be47a20948066531a4570f13fe02dbe0f7580cd8b5cad5fb172fabe9f33  /usr/include/c++/9/x86_64-redhat-linux/32/bits/gthr-posix.h
c0b15be47a20948066531a4570f13fe02dbe0f7580cd8b5cad5fb172fabe9f33  /usr/include/c++/9/x86_64-redhat-linux/bits/gthr-default.h
c0b15be47a20948066531a4570f13fe02dbe0f7580cd8b5cad5fb172fabe9f33  /usr/include/c++/9/x86_64-redhat-linux/bits/gthr-posix.h
...
```

The output shows all duplicate files in groups. For example, you can see that the file `gthr-default.h` is repeated verbatim 4 times.

Appendix 2 – obtaining/guessing file types using file(1) utility

The `file(1)` utility is used on UNIX systems to guess the file type. It prints out the main type of the file, and sometimes followed by extra details, separated by commas. For example, to see all types of the files in the git repo you could type:

```
$ file *
digester.cpp:    C source, ASCII text
digester.h:      C++ source, ASCII text
getDirStats.cpp: C source, ASCII text
getDirStats.h:   C source, ASCII text
main.cpp:        C source, ASCII text
Makefile:        makefile script, ASCII text
README.md:       ASCII text
test1:           directory
```

The output above identified `digester.h` file as a “C++ source”, with the extra detail “ASCII text”. For this assignment we only care about the main type, i.e. “C++ source”. To make parsing easier, you can invoke the file utility with the `-b` option, which will omit the filenames from output.

Here is an example of a command line you can use to get the most common types from a large directory `/usr/share/doc/octave`:

```
$ find /usr/share/doc/octave/ -type f \
   | xargs file -b \
   | awk -F, '{print $1}' \
   | sort \
   | uniq -c \
   | sort -nr

   2651 HTML document
     36 C source
     29 ASCII text
     27 PNG image data
      5 PDF document
      3 UTF-8 Unicode text
      1 LaTeX document
      1 C++ source
```

The first `find` command in the pipe recursively scans the directory and prints all files. The second `xargs` command runs ‘`file -b`’ on every file. The third `awk` command splits the output by commas and prints only the first field. The fourth `sort` command sorts the output alphabetically. The fifth `uniq` command counts consecutive line occurrences and prints the counts next to the strings. The last command `sort` sorts the output numerically, in descending order.

Appendix 3 – python solution

The repository includes a Python program `dirstats.py` that implements the assignment. This should help you design your own test cases and see what the expected output should look like. Here is an example of running it on the `test1` directory:

```
$ ./dirstats.py 5 test1
--------------------------------------------------------------
Largest file:      "test1/dir1/down.png"
Largest file size: 202
Number of files:   7
Number of dirs:    9
Total file size:   253
Most common file types:
 - "ASCII text" x 5
 - "PNG image data" x 1
 - "empty" x 1
Most common words:
 - "contents" x 4
 - "test" x 2
 - "exdu" x 1
 - "idatx" x 1
 - "iend" x 1
Duplicate files - group 1
 - "test1/dir1/file1.txt"
 - "test1/dir1/file2.txt"
 - "test1/a/b/c/d/f"
--------------------------------------------------------------
```

General information about all assignments:

1. All assignments are due on the date listed on D2L. Late submissions will not be marked.
2. Extensions may be granted only by the course instructor.
3. After you submit your work to D2L, verify your submission by re-downloading it.
4. You can submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It is better to submit incomplete work for a chance of getting partial marks, than not to submit anything. Please bear in mind that you cannot re-submit a single file if you have already submitted other files. Your new submission would delete the previous files you submitted. So please keep a copy of all files you intend to submit and resubmit all of them every time.
5. Assignments will be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA, then you can contact your instructor.
6. All programs you submit must run on `linuxlab.cpsc.ucalgary.ca`. If your TA is unable to run your code on the Linux machines, you will receive 0 marks for the relevant question.
7. Unless specified otherwise, you must submit code that can finish on any valid input under 10s on linuxlab.cpsc.ucalgary.ca, when compiled with `-O2` optimization. Any code that runs longer than this may receive a deduction, and code that runs for too long (about 30s) will receive 0 marks.
8. **Assignments must reflect individual work**. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k-5.html.
9. Here are some examples of what you are not allowed to do for individual assignments: you are not allowed to copy code, pseudo-code or written answers (in part, or in whole) from anyone else; you are not allowed to collaborate with anyone; you are not allowed to share your solutions with anyone else; you are not allowed to sell or purchase a solution. This list is not exclusive.
10. We will use automated similarity detection software to check for plagiarism. Your submission will be compared to other students (from current and previous semesters), as well as to any known online sources. Any cases of detected plagiarism or any other academic misconduct will be investigated and reported.