3: Sniffing a Directory

CSCI 4547 / 6647 Systems Programming
Fall 2020

1 Goals

1. To work with Unix directories and iNodes.
2. To read and process directory entries.
3. To open and close files, and search their contents.

This program builds on and incorporates Program 2, tt main() will change, but your Params class will not. Therefore, you need to debug P2 before starting this one.

Please note: There are efficient ways to do all parts of this job. But efficiency is not the goal here. Rather, the goal is to keep the instructions simple enough to implement and debug in a short time, even if that means using less efficient methods sometimes.

2 The Story

Pretend you work for a school system where the students need to use the school's computers and need to be able to store files on them. The school is concerned about keeping its equipment free of problematic content. They have other utility programs that can sniff out photos and illicit URL's. Now they want one that can check on the content of files stored on the machine. They have asked you to write a program that can search through a student's file directory and find files that contain words that are currently of concern. We will call this process "sniffing the file".

For your alpha version, it is enough to be able to correctly search the .txt and .rtf formats; these file types are both line-oriented and have newline characters in the text. You do not need to be able to analyze binary files (.pdf, .docx).

3 Class FileID

Define a class FileID to hold information about one file.

Private data members should be:

- Simple file name (from a directory entry).
- iNode number (from a directory entry).
- Pathname relative to the starting directory (a C++ string). Construct the string by appending the simple file name to the path of the directory you are processing.
- A `vector<string>` in which to store the sniff-words (search strings) found in the file.

Function members should include:

- A constructor, with parameters for name, type, and path.
- `void print( ostream& out )` Print the iNode number and path to the output stream using using `out<<`. Use a tab to make neat columns.

- A function to insert a sniff-word into the vector when it is found. Check first; do not insert it again if it is already there.

Write other functions if you need them. I think you don't.

4 Instructions

4.1 main()

Write a C++ program to analyze the entries in a file directory. You will run this program from a Unix command shell, using command-line arguments. Your main function should accept argc and argv from the command shell, and process them according to the following instructions.

Call banner() (tools library). Print a welcome message on the screen. Declare an instance of the Params class and pass the command-line-arguments to the constructor. Declare an instance of the Sniff class and pass the Params object to the constructor.

Execute a `chdir()` into the starting directory named in Params. Call `Sniff::oneDir()` to process that directory. (This logic will be modified and extended for program 4). Call bye() from the tools library.

4.2 The Sniff Class

Your `Sniff` class will be the controller for this application. All functionality will go into this class, not into the main function.

You need these private data members:

- An instance of Params.
- The simple name of the first directory to be searched, initialized from the Params.
- The pathname of the directory to be searched, initially the same as the directory from Params.
- words, a `vector<string>` for unpacking the list of sniff-words.
- An empty `vector<FileID>` for the suspicious files found in the current directory.
- A struct dirent, to hold the current director entry.

Add more data members if needed. The necessary functions are listed in the next section.

The Sniff Constructor. Initialize the filename, pathname, and the vector of sniff-words at this time. The Params object holds a quoted string of one to five words. You need to unpack that string into `words`. Wrap an istringstream around the arg string. Then use your stringstream with `>>` to read the individual words from the string and store them in the vector. Refer to the demo stringstr.cpp for an example of how to use a stringstream of this kind and how to check for eof.

The oneDir() Function. Eventually, Sniff will do a recursive treewalk starting from the given directory. First, however, you need to debug the process for a single directory.

- Open the current working directory (opendir()). Your program should already have done a `chdir` into this directory.
- To aid in debugging, print the relative path name of the cwd.

- Read and discard the first two directory entries (dot and dot dot).
- Enter a loop that will read the rest of the entries using readdir(). Exit when the return value is nullptr.
- Read the next directory entry.
  - Test the type of the directory entry. Discard it unless it is a regular file or a directory. (Specifically, discard soft links, pipes, sockets, and devices).
  - If it is a directory or a file, and the verbatim switch is on, output the name.
  - If the entry is a directory, do nothing more for now. (Program 4 will process directories also.)
  - If the entry is a regular file, process it by calling oneFile() with the appropriate parameters.
  - Store the FileID object that is returned in the vector of files, if and only if at least one sniff-word was found in that file. If no sniff-words are found, discard the FileID.
  - End of loop.
- To aid in debugging, print a comment that you are done processing this directory.

The oneFile() Function

- Create and initialize a local FileID object.
- Open a stream to read the file.
- Search the file contents for each of the words in your sniff list.
- If a word is found, store it in the vector that is inside the FileID object. Before inserting it, check if it is already there; do not insert it a second time.
- There are several ways to search the file for a single string or all five strings. The easiest way is to read the file one word at a time using `>>` and compare the input to the set of sniff words. There are two complications with the comparisons:
  - Strip non-alphabetic characters off the end of the input word before comparing it.
  - Check the -i switch in Params and be sure to use the right comparison function.
- When the whole file has been checked for all the words, close the file.
- Return the FileID object to oneDir().

5 Other instructions: Due Oct 9

- Don't hesitate to ask for help or clarification.
- Test for error codes returned by the library functions. Do not let an error crash your program. Otherwise, don't do anything special about handling errors; that will come in the next phase of this program.
- Create a test folder. In it, put a few files of varied lengths. Now add some links, both hard and soft, to files in the same directory. Do `ls -l` on your test directory before the tests and put the results in your output file.
- Test all of the command-line options and capture the results from all tests in your same output file.