p003 - pStrider
CIS 310 Operating Systems
Fall 2023

Prior Knowledge

Before doing this assignment, a student *should* be able to:

- Access a machine with `make` (GNUMake 4.4.1) and `g++` (GNU Compiler Collection 13.1.0).

Learning Outcomes

Upon completing this assignment, students should be able to

- Write a program that checks the number of command-line parameters, printing a standardized `usage` message if the number is out of range.
- Write a recursive function that tracks the depth of the recursion.
- Use the C++-17 `filesystem` classes to
  - determine whether a `path` object refers to an existing file system object.
  - determine the *type* of file system object to which a `path` refers.
  - iterate over the contents of a `directory`.

Introduction

Students will implement a simple version of the `tree` utility in *nix. Given a list of paths in the file system, the program will visit each in turn and display the complete file system tree at and below that location. For example:

```
$ pStrider ~/310/pStrider/src
/home/blad/310/pStrider/src
  /home/blad/310/pStrider/src/main
    /home/blad/310/pStrider/src/main/Strider.cpp
    /home/blad/310/pStrider/src/main/executable.mk
  /home/blad/310/pStrider/src/allModule.mk
```

Where the indentation shows the nesting of files and directories at and below the given `src` directory. Note that the tilde (~) expansion is built in to most shell programs.

Method

Getting Started

Start with a fresh clone of `templateCPP` (`https://cs-devel.potsdam.edu/F23-310/CPP-Template.git`). The fresh clone gets the `Makefile` that removes an outdated link library reference.[1]

You will define a main program file in a package folder and a `Strider` class in a separate package folder. Define both, include the `Strider` header file into your main file, and make sure everything compiles.

[1] `c++-17` introduced `std::filesystem` and, the last time Operating Systems used C++, Gnu compilers needed a separate standard library to support it. That library has since been folded into the actual standard library and need no longer be linked separately.

`main` The main program should check the number of arguments provided on the command-line. Unlike Java, C++ arguments *include* a first argument that is the name of the executable (we will see more about this when we program with processes). This means that the argument count is never zero and that arguments to *your program* begin at index 1.

The `main` function should check that there is *at least* one parameter to the Strider program. If there are none, then `main` should call a `usage` function that prints out a short "how to use Strider" message.

When `usage` returns, `main` should terminate with a non-zero return code. **In your README** include a sentence or two on what the difference between a zero and non-zero return code means.

Assuming the argument count check is passed, your program instantiates a new `Strider`.

`Strider` The `Strider` class constructor takes a `std::vector<std::string>` with the command-line arguments (`main` copies the array of arrays of characters into a `vector`).

`Strider`'s one public method is `stride()`. It walks across the entries in the argument `vector` and checks if the location *exists*. If it does, `Strider` recursively prints out the file and directory contents of the file system, starting at the path specified by the argument. The topmost entry should be printed at the first character of a new line. Each subsequent, nested entry should be printed at a uniform indent from the entry in which it is nested.

If the entry does *not* exist, go ahead and print an error message to *standard error*.

After `stride()` returns, `main` terminates, returning zero.

Design Considerations

Levels of Recursion

Your recursive filesystem walker function should take two parameters: where it is in the file system (look at the types defined in `std::filesystem` for figuring out the type of this parameter) and some way to know how much indent to put at the beginning of lines that this call prints.

You should start with an indent step of 2 spaces (make sure you use *spaces*) per level of recursion. Note that it is a grading consideration that you (or another programmer) can *easily* change the indent step.

Filesystem Objects

The file system has many different objects: files, directories, symbolic links, *etc.* For the purposes of this program, there are only two sets of objects: *directories* and ***not*** *directories*. Directories must be visited and their contents processed by your `Strider`; non-directories just need their names printed at an appropriate indent.

You may choose to print *absolute* paths to the elements you print (as in the example in the introduction) or you may print *relative* paths (more like the output generated by the `tree` utility program). Do one or the other consistently. Your error message on non-existent items can use the string provided on the command-line.

Testing

You will want to work out a *testing schedule* or *plan* and **document** what you did to test your code. You may want to look at input/output redirection to make it easy to redo tests from files. And, maybe, even the `diff` command to compare actual to expected output.

Documentation

Note Repeacted or not these requirements apply to all programming assignments in CIS 310.

`.gitignore`

You should have an appropriate `.gitignore` file in the root of your project. It should be set up to ignore all junk files generated by *your* favorite editor and any artifacts that are **built** from the source code you are turning in (I will compile the code on my box.)

README

You `README` file must be in some plaintext format: text, Org, or Markdown, for example. The file should have an appropriate extension and be found in the root of your project.

The file must include at least the following:

Identification Block Who wrote the program. Just in case the folder it is cloned into has an issue with naming. Format is in the Coding Standard.

Problem Restatement You need to explain the problem that your code is meant to solve. This is a short restatement of the assignment. It is a paraphrase, your understanding of what you are doing. This could, probably, be written before you begin to write code to check your understanding.

Testing Plan "Of course it is right: it compiles!" hopefully makes you smile at this point in your programming career. So, how *do* you show that your code solves the given problem? You test it. You need to turn in (and document in `README`):

Test Input Data files or descriptions of input to type to run different tests.

Test Output Corresponding results, the expected output that would be generated if your code were correct.

Test Execution Instructions on how to run the same tests you ran with the given *inputs* and how to compare results to the expected *outputs*. How the customer can verify that at least your tests work.

Compilation/Execution Instructions Clearly describe the commandline that would be used, in your working tree, to *compile* and then *execute* your solution. This is separate from the testing runs listed above because this represents the *happy path* or the way a new customer will run your code. Make sure to explain the *meaning* of commandline prameters.

Deliverables

Submission medium: `git` to Gitea at `cs-devel.potsdam.edu`.

Gitea repo name: `F23-310-<ccid>-p003`.

Include everything except build artifacts.