Project 2

Due October 23, 2019 at 11:59 PM

This project specification is subject to change at any time for clarification. For this project you will be working with a partner. You will be implementing a tree data structure that can emulate a file system. In order to guide your development and to provide exposure to Test Driven Development, your implementation will be required to pass a set of GoogleTest tests that have been provided. After implementing your tree data structure you will be creating a simple command line prompt program that will allow navigating the tree as if it were navigating a real file system. The `CFileSystemTree` has an interface with three sub classes (`CEntry`, `CEntryIterator`, and `CConstEntryIterator`) described below. The `CFileSystemTree` has the following member functions:

```
// Construcotrs/destrucotr
CFileSystemTree();
~CFileSystemTree();

// Returns the CEntry for the root of the file system
CEntryIterator Root();
CConstEntryIterator Root() const;

// Converts the tree into a string that has the form similar to
// tree command. For example:
// /
// |--Makefile
// |--bin
// |--include
// |  |--DirectoryListing.h
// |  `--FileSystemTree.h
// |--obj
// |--src
// |  |--DirectoryListing.cpp
// |  |--FileSystemTree.cpp
// |  |--main.cpp
// |  `--testtree.cpp
// `--testbin
std::string ToString() const;
operator std::string() const;



// Find the CEntry and returns an iterator to it
CEntryIterator Find(const std::string &path);
CConstEntryIterator Find(const std::string &path) const;
```

```
// Returns an CEntryIterator that is equivalent to that returned
// when the path is not found with Find
CEntryIterator NotFound();
CConstEntryIterator NotFound() const;
```

The CEntry is how the individual directory and file entries will be accessed in the CFileSystemTree.

```
// Constructors/destructor
CEntry();
~CEntry();

// Returns if the CEntry is valid or not
bool Valid() const;

// Returns the name of the entry similar to BaseName() of CPath
std::string Name() const;

// Returns the full path to the CEntry
std::string FullPath() const;

// Converts the CEntry into a string that has the form similar
// to tree command. See the ToString for CFileSystemTree
std::string ToString() const;
operator std::string() const;

// Attempts to rename the CEntry
bool Rename(const std::string &name);

// Returns the number of children, files will always have zero
size_t ChildCount() const;

// Sets the iter to become a child of the entry with name.
// Can be used for moving of CEntries in the CFileSystemTree.
bool SetChild(const std::string &name, const CEntryIterator
&iter);
```

```
// Adds a child to the CEntry.
bool AddChild(const std::string &name);

// Adds a child to the CEntry including the intermediate
// entries.
bool AddPath(const std::string &path);

// Removes the child from the CEntry
bool RemoveChild(const std::string &name);

// Sets the data for a file. SetData will fail if the CEntry
// already has a child.
bool SetData(const std::vector< char > &data);

// Gets the data for a file, if there is any.
bool GetData(std::vector< char > &data) const;

// Gets the Parent of the CEntry
CEntryIterator Parent();
CConstEntryIterator Parent() const;

// Finds and returns an iterator of the CEntry specified by name
CEntryIterator Find(const std::string &name);
CConstEntryIterator Find(const std::string &name) const;

// Gets an iterator to the first child of the CEntry
CEntryIterator begin();
CConstEntryIterator begin() const;
CConstEntryIterator cbegin() const;

// Gets an iterator to the CEntry past the last child
CEntryIterator end();
CConstEntryIterator end() const;
CConstEntryIterator cend() const;
```

The `CEntryIterator` and `CConstEntryIterator` are iterators of the `CEntries`. They have the same interface with the exception of the `CConstEntryIterator` only allowing for `const` access. Only the `CEntryIterator` is shown for brevity.

```
// Constructors/destructor
CEntryIterator();
CEntryIterator(const CEntryIterator &iter);
~CEntryIterator();

// Assignment operator overload
CEntryIterator& operator=(const CEntryIterator  &iter);
```

```
// Equality comparison
bool operator==(const CEntryIterator &iter) const;
// Inequality comparison
bool operator!=(const CEntryIterator &iter) const;

// Pre increment of the iterator, returned value is the result
// of the incremented iterator
CEntryIterator& operator++();

// Post increment of the iterator, returned value is equal to
// iterator prior to increment
CEntryIterator operator++(int);

// Pre decrement of the iterator, returned value is the result
// of the decremented iterator
CEntryIterator& operator--();

// Post decrement of the iterator, returned value is equal to
// iterator prior to decrement
CEntryIterator operator--(int);

// Member access operator, allows access as if iterator was
// pointer.
CEntry *operator->() const;
```

Once all tests have passed a simple interactive program you will be able to start on the main command line program for project 2. The command line program will need to implement the following commands: `exit`, `pwd`, `cd`, `ls`, `cat`, `mkdir`, `rm`, `cp`, `mv`, and `tree`. The commands are elaborated below. If a command line argument is passed to `proj2`, the directory specified as the argument needs to be loaded into the tree (all directories will be added as CEntries, and all files will be read in with data set). If the command line argument is passed, the directory will be "mounted" as the root of the `CFileSystemTree`. A working example can be found on the CSIF in `/home/cjnitta/ecs34/`. If the example crashes please provide any error messages that are output. The commands are as follows:

`exit` – Exits the program

`pwd` – Prints the current working directory

`cd [dir]` – Changes directory to `dir` if specified, otherwise if no argument is specified changes directory to root

`ls [dir]` – Lists the entry names in the directory specified by `dir`, otherwise lists the entries from the current working directory if no argument is specified

`cat file` – Prints out the contents of the file specified by `file` `mkdir dir` – Makes a directory named `dir`

`rm file_dir` – Removes the file or directory specified by `file_dir`

`cp src dest` – Copies the `src` file to the `dest` directory (if exits) or to the `dest` name (if not existing)

`mv src dest` – Moves the `src` file or directory to `dest` directory (if exists) or to be `dest` (if not existing)

`tree [dir]` – Prints out the tree rooted at `dir` if specified, otherwise prints the tree rooted at the current working directory

When loading the "mounted" directory, you will find the `GetListing` function helpful. In the `DirectoryListing` namespace that has been provided. The `GetListing` function takes in a path and will fill out the vector of entries that have a name and a bool. The `bool` is true if there entry is a directory.

```
bool GetListing(const std::string &path, std::vector<
std::tuple< std::string, bool > > &entries);
```

A working `FileSystemTree.o` object file that can be linked against for developing the second half is available in `/home/cjnitta/ecs34`. To develop using this object file, you need to copy it into the obj directory of your project. So starting from the directory where you can type make to make your project, use the command:

```
cp /home/cjnitta/ecs34/FileSystemTree.o obj/FileSystemTree.o
```

This will copy the object file into your obj directory and will allow you to make the project even though your source code for `FileSystemTree` may not be complete. **IMPORTANT**: this will only work on the CSIF with `make`, this will not work if you are compiling on your home machine.

You can unzip the given tgz file with utilities on your local machine, or if you upload the file to the CSIF, you can unzip it with the command:

```
tar -xzvf proj2given.tgz
```

You **must** submit the source file(s), the provided unmodified Makefile, and README.txt file, in a tgz archive. Do a `make clean` prior to zipping up your files so the size will be smaller. You can tar gzip a directory with the command:

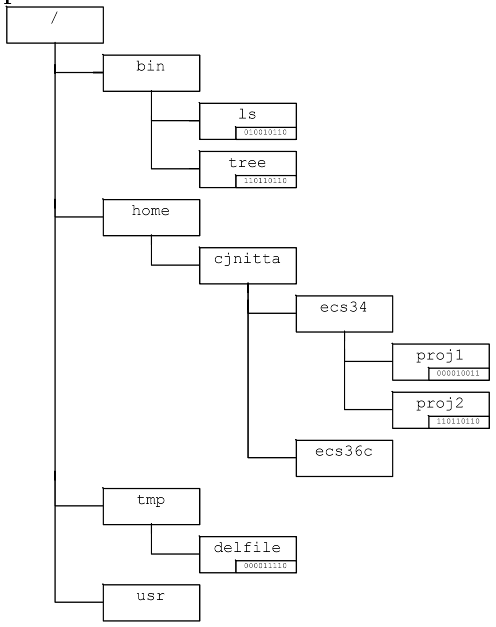
```
tar -zcvf archive-name.tgz directory-name
```

Provide your interactive grading timeslot csv file in the tgz. The directions for filling it out have been posted.

You should avoid using existing source code as a primer that is currently available on the Internet. You **must** specify in your readme file any sources of code that you have viewed to help you complete this project. All class projects will be submitted to MOSS to determine if students have excessively collaborated. Excessive collaboration, or failure to list external code sources will result in the matter being referred to Student Judicial Affairs.

Helpful Hints

- Read through the guides that are provided on Canvas
- See http://www.cplusplus.com/reference/, it is a good reference for C++ built in functions and classes
- You may use (and are encouraged to use) the StirngUtils and CPath from project 1.
- Make sure you are in the project directory with the `Makefile` when you type `make`.
- If the build fails, there will likely be errors, scroll back up to the first error and start from there.
- You may find the following line helpful for debugging your code:
  ```
  std::cout<<__FILE__<<" @ line: "<<__LINE__<<std::endl;
  ```
  It will output the line string "FILE `@ line: X`" where FILE is the source filename and X is the line number the code is on. You can copy and paste it in multiple places and it will output that particular line number when it is on it.
- Start your design by only thinking about the overall structure and what information needs to be in each node. Consider the file structure of directories and files below and how you would implement representing just it alone. What information is needed in each node to represent what is shown? If you have a pointer to one node, how would you find the parent or first child?



- Now consider that the `CFileSystemTree` is an interface to accessing the nodes inside the tree. The `CFileSystemTree` will likely just need a reference to the root of the tree, so will really only have a pointer to the root node. The `CEntry` is an interface to an individual node, and a `CEntryIterator/CConstEntryIterator` is used to iterate through the children of the nodes.

- Consider that if `Parent()` is called on a `CEntry`, then it needs to return a `CEntryIterator` associated with the specific parent internal node. The actual `CEntry` returned from a sequence like `begin()->Parent()->` does not need to be identical to the caller, but both need to reference the same internal node.
- After you have the general structure thought out and working, now consider the special cases. What if `Parent()` is called on `Root()`? How will you distinguish between a found entry iterator and a `NotFound()` iterator? What if a `CEntryIterator` that is equal to `NotFound()` is accessed with `operator->()`? You may need an additional data member in to keep track of this in one of the implementation classes.
- A suggested order to implementing the functions after you have your node type are:
  ```
  CFileSystemTree::CFileSystemTree()
  CFileSystemTree::Root()
  CFileSystemTree::CEntryIterator::CEntryIterator()
  CFileSystemTree::CEntryIterator::operator->()
  CFileSystemTree::CEntry::Name()
  CFileSystemTree::CEntry::FullPath()
  CFileSystemTree::ToString()
  CFileSystemTree::operator std::string()
  CFileSystemTree::CEntry::ToString()
  CFileSystemTree::CEntry::operator std::string()
  CFileSystemTree::CEntry::Parent()
  CFileSystemTree::CEntry::Valid()
  CFileSystemTree::CEntry::AddChild()
  ```
  Then into other functions