ECS 40 **Program #6: Final FUNIX (50 points, my time 2.5 hours)** **Winter 2015**

Due: Wednesday, February 25th at 11:59pm using handin to p6 directory of cs40a.
New concepts: inheritance, templates Name of executable: funix.out
File names: Makefile, file.cpp, file.h, funix.out, funix.cpp, funix.h, directory.cpp, directory.h, main.cpp, permissions.cpp, permissions.h, Time.cpp, Time.cpp, list.cpp, list.h, and authors.txt (in short, all source files)

For this assignment, you will modify your program #5 code to: 1) Make list a template class; 2) Create a File class that is the base class for the Directory class; 3) Make subDirectories a List<File*>; 4) Add a touch command that creates File(s). You are welcome to use my own code as a starting point. It will be available in ~ssdavis/40/p5/SeansSrc Thursday morning. Part of your design grade will be based on the proper use of const. I will add the const design detection to the design checker AFTER the deadline so you will have to think about this and not rely on it.

Further specifications:

1. Make the List and ListNode classes templates. (15 minutes)
   - 1.1. Use global search and replace in the list.* files to speed up the conversion.
   - 1.2. There should be no mention of Directory or File anywhere in your List files. Directory will all be replaced with "T".
   - 1.3. Add `template <typename T>` above every function in list.cpp, and above each class declaration in list.h
   - 1.4. To make the ListNode look more generic, replace the identifier "directory" with "data";
   - 1.5. Almost everywhere you mentioned List or ListNode you will now have to append a <T> to specify the template class instantiated. I would globally search and replace all of them, and then go back and eliminate the few <T> that are unnecessary.
   - 1.6. Remember to '#include "list.cpp"' at the bottom of list.h.
   - 1.7. Remove the two list.cpp lines, and any mention of list.o from the Makefile.
   - 1.8. Add list.cpp to any dependency list in the Makefile that currently has list.h in it. Otherwise, when you alter list.cpp, make won't recompile your code!
   - 1.9. The List in directory.h should now be a List <Directory>, or List <Directory>*. Surprisingly, you will not need to make any changes in directory.cpp.
2. Create a File class that is the base class for the Directory class.
   - 2.1. One of the goals for this part of the assignment is figure out ways to "hide" the variables from the public and derived classes by minimizing the number of mutator, e.g. setName(), and accessor, e.g. getName(), functions for each class. The only public accessor or mutator methods you may have in the File class are getPermissions(), touch(), read(), write(), and the two iostream friends.
   - 2.2. Another goal is for you to learn the differences between private and protected modes, and the subtle limitations of protected mode.
   - 2.3. The File class will contain protected variables permissions and name, but time will be a private variable.
     - 2.3.1. Make the Directory class a publicly derived class of the File class.
     - 2.3.2. The Directory class should not explicitly declare the base class variables anymore. It will still have parent, subdirectories, and subDirectoryCount though.
     - 2.3.3. You will find that you will often cut Directory code, and paste it into File code.
   - 2.4. The File class will need a default constructor, copy constructor, and a "standard" constructor.
     - 2.4.1. The "standard" constructor will take a name, and Permission. The Permission argument will have a default value, but will often have to be constructed "on the fly" from an owner and appropriate short.
   - 2.5. The File class virtual destructor will eliminate the need for a Directory destructor.
   - 2.6. Since name is protected you will need to add and move methods dealing with it.
     - 2.6.1. Move the overloaded Boolean operators from Directory to the public section of File.
     - 2.6.2. Add a public const ls() method to File that will print the proper formatted ls information based on its bool parameter, isLongFormat. This will help eliminate the need for a getName() method.
   - 2.7. Since time is private, you will need to add a protected updateTime() method to File.
3. Once you have created the File class, change Directory::subDirectories to a List of File. This will engender many, many errors. I will address some of these below. I suggest you that use multi-line comments to comment out chunks of functions that are riddled with errors for now. The overloaded iostream operators, in particular, are notorious for creating mountains of compiler errors when they are not just right. Then uncomment one chunk, and work on it until there are no more compiler errors.

3.1. cd() , cp(), and the Directory copy constructor must deal with the fact that the elements of the SubDirectories could be either regular Files or Directories. All three of these must know whether a particular element is a Directory or regular File to work properly. For example, you cannot cd into a regular File. A simple way to determine this at runtime is to use a dynamic_cast. This is covered on page 825 in the text. For our purposes you will often write lines like:

```
Directory *directory = dynamic_cast<Directory*> (subDirectories[i]);
```

You can then test if directory is NULL to determine if you are working with a regular File or a directory. As a bonus, since you have a Directory pointer now, the compiler will let you access the Directory variables of the element, e.g. parent.

3.1.1. For the Directory copy constructor, you can no longer use the List copy constructor, and must use the List += operator to construct the list with appropriate dynamically allocated type.

3.1.2. cd() should now report an error if you try to cd into a regular file.

3.1.3. cp() should now also simply copy a regular file.

3.2. Permission's status as protected causes lots of problems for the Directory class methods.

3.2.1. Since reading permissions is a public function in ls –l, there is no harm in adding a public const getPermissions() method to the File class. This will facilitate all of the isPermitted calls in Directory.cpp.

3.2.2. Though "protected" status allows methods of a derived class to access and change the protected variable of derived object, it does not allow a derived class method to access and change the variable if it is an object of the base class, or another derived class. For the Directory class this means it cannot naturally deal with the permissions of the subdirectories list elements because they can be either Directories or Files. There would seem to be many solutions but most have pitfalls.

3.2.3. You could cast all subDirectories[i] as Directory* at compile time, e.g. (Directory*) subDirectories[i], but this is dangerous because you could attempt to do something with a regular File that is simply not possible, e.g. access a subDirectoryCount, and the compiler has no way to detect that possibility. **So we will not allow any compile time casting in this program.**

3.2.4. You may think that you could create virtual protected methods in both classes, and let it choose at run time which to call. However, this is not allowed by the compiler because your program could pass a base class method a base class pointer that is pointing at an object from a class other than yours that was derived from the same base class, and then that object could be accessed or even changed by that base class method!

3.2.5. Dynamic_casting won't save us completely either. The actual Directory* will be able to change their permissions once they are actual Directory*, but File* cannot be dynamic_cast into Directory*.

3.2.6. I will leave it as puzzle worth 5 extra credit points for you to figure out how to change the permissions of a File without adding any other public methods to File. Hint: mine does update its time.

3.3. Reading and writing to directories.txt

3.3.1. The format of directories.txt must once again change so that the variable that only the Directory class contains, subDirectoryCount, is listed last.

3.3.2. Since you cannot have virtual friends, the overloaded iostream operators cannot be virtual.

3.3.3. For insertion, you will only have a File insertion operator now.

3.3.3.1. It calls virtual write() methods of the two classes that do all of its work.

3.3.3.2. The Directory write() should call the File write(). It will create a new line with just the subDirectoryCount on it.

3.3.3.3. To aid with extracting the file, the Directory::write will first insert a 'D', and the File::write will first insert an 'F'. Thus the beginning of each line will either be "DF" or "F"

3.3.4. For extraction, you will only have a File extraction operator now.

3.3.4.1. It calls virtual read() methods of the two classes that do all of its work.

3.3.4.2. The Directory::read relies on the first letter(s) in each line to determine what it should do with each "subdirectory". If an 'F', it creates a new File, calls the write(), and stores the pointer in the subDirectories array. If a 'DF', it creates a new Directory, calls the File::read method, and stores the pointer in the subDirectories array. Note that Funix::operator>> will have to read the 'DF' of root.

4. Write a touch command.

4.1. Syntax: **touch** [filename]+ This is used to create files, not directories. The created file is given 0666 permissions.

4.2. touch should require the same permissions as mkdir does, and provide similar error messages.

4.3. When an existing File (whether a regular File or a Directory) is touched its modificationTime is set to the current time if the user has write permissions. This will rely on a public File::touch() method.