CS 201: Problem Solving & Programming II
Lab #5

Exercise 1: Warm-up

For this exercise, we're going to write Sum functions that add two numbers together. We are going to overload this function so we have a version with reference variables and pointer variables.

Reference Sum:

This function will have an **int** return-type. It will have two parameters, which are **integer-references.** Inside the function, **return** the sum of the two parameters.

Pointer Sum:

This function will have an **int** return-type. It will have two parameters, which are **integer-pointers.** Inside the function, **return** the sum of the value of each pointer. You will have to use the de-reference operator, otherwise you'll be adding memory addresses.

Note:

If we wrote a third Sum function with **normal-variables**, when we try to call Sum(...) with two normal-variables, the compiler would give us an "ambiguous" error because it won't be able to tell whether to use the reference version or the normal version.

After you have both of the functions declared, we need to add three steps in **main()**:

1. Ask the user to enter two numbers. Store these in normal-variables of type **int**.
2. Create a normal-variable of type **int** named **sum**. Call the version of the Sum function for references and store the return in the sum variable. Output sum with cout.
3. Re-use the normal-variable **sum**. Call the version of the Sum function for pointers and store the return value in the sum variable. Output the sum with cout.

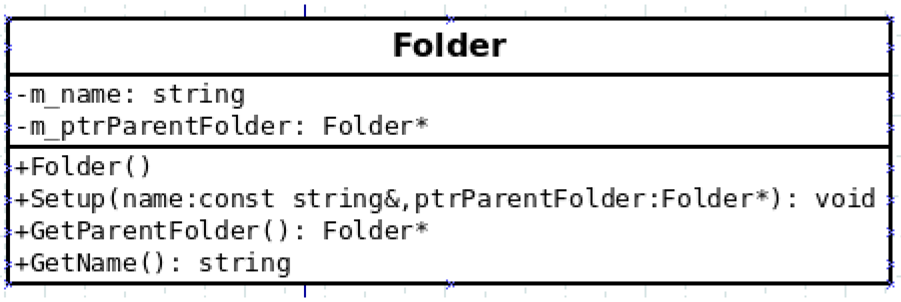
Remember that you will need both de-reference operators and address-of operators when dealing with pointers. Can you remember where our pointers need to be de-references and where we need to get address-of normal-variables?

Exercise 2: Filesystem Structure

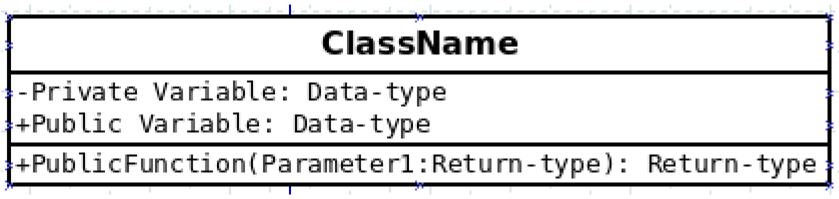
In this exercise, we are going to create a class called "Folder". A folder has a name, and has a pointer to its parent Folder. We are then going to declare a bunch of folders and set their relationships, and allow the user to pick a folder to look at. Whichever folder is picked, we will output that folder's name, and the name of all of its parents.

The Folder class

Our folder class will be relatively simple, though there may be some unfamiliar parts, such as returning a pointer-variable as a return-type (usually we've just been returning normal-variables). Here is the diagram for our Folder class:


Folder
-m_name: string
-m_ptrParentFolder: Folder*
+Folder()
+Setup(name:const string&,ptrParentFolder:Folder*): void
+GetParentFolder(): Folder*
+GetName(): string

If you're nor familiar with UML, here is a key:


ClassName
-Private Variable: Data-type
+Public Variable: Data-type
+PublicFunction(Parameter1:Return-type): Return-type

So, the top section of the table is **member-variables**, and the lower section is **member-functions**. + means public, - means private.
Parameters of a function is listed as **name:return-type**.

Create two files: Folder.h and Folder.cpp. Write your class declaration shell to match the diagram above, and we'll step through the definition of the functions.

Note that the Folder class is storing a Folder* pointer: It will have a pointer to its parent's memory-address. This will have to be passed in in the **main()** function.

Function: Folder Constructor
This is our constructor, with no parameters. Within the constructor, set **m_name** to a default value (such as "uninitialized"), and set **m_ptrParentFolder** to **NULL**. Some folders may have no parents, and when this is the case the parent folder pointer will stay NULL.

Function: Setup
Our Setup function will not return anything, and will have two parameters:

- constant reference string called “name”
- Folder pointer called “ptrParentFolder”

This is how we'll initialize the private member variables of Folder. Set **m_name** equal to the passed-in parameter, and same for **m_ptrParentFolder**.

Note:
We don't need to de-reference or use address-of operator within this Setup function. Because our **ptrParentFolder** parameter is already a pointer, you can simply set one pointer to another pointer!

Function: GetName
This is a simple function that returns a string. Return **m_name**.

Function: GetParentFolder
This is a function that returns a *pointer to a folder*. This might be a bit foreign since we normally just return normal-variables. It is possible and legitimate to return pointer-variables and reference-variables as well.
Return **m_ptrParentFolder**.

Note:
Here, we also don't need to de-reference or address-of anything, since we're returning A pointer anyway. If we were returning a normal-variable Folder, then we would need to De-reference our pointer before returning!

That should be it for the Folder member functions.

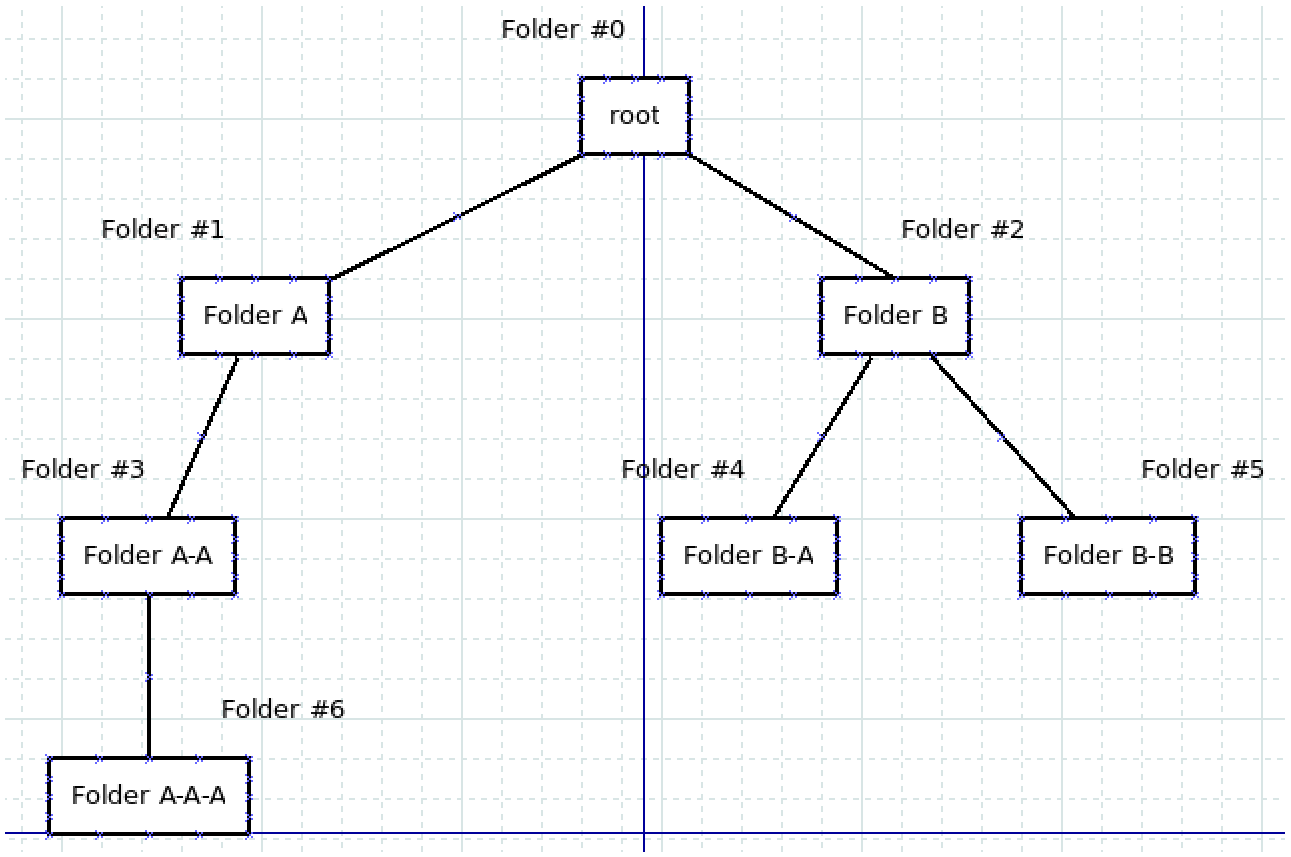
In the file containing **main()**, make sure to **#include “Folder.h”** and make sure the program still builds. If it doesn't, fix the errors before continuing.

<u>Our Folder tree</u>

A computer file-system is one big tree. Each folder can only have one parent, but a folder can have multiple children as well.

It might seem logical to have a Folder store an array of ChildFolders, but in this case we're going to have our Folder simply store one pointer to its ParentFolder.

Here is a diagram of our file-system tree:



We will have a total of seven folders, and each Folder has a parent, except for **root**, which has “NULL” set for its parent pointer.

Within **main()**, create a simple static array of Folders, of size 7.

We are then going to manually initialize each of these folders with our **Setup** function. Remember that the first argument is **name**, and the second is **parent**. Start with folder 0 and work down to folder 6.

Example:

```
     folders[0].Setup( "root", NULL ),
     folders[1].Setup( "FolderA", &folders[0] ),
```

Once you have all of the folders created, build the program just to make sure everything compiles. The next step is creating a menu for the user to choose a folder.

<u>User Interface</u>

Create a for-loop that goes from 0 to 7 (not-inclusive) and increments by one each time. Within this for-loop, we are going to **cout**:

- The current index number (e.g., if you declared "int i", output "i").
- The element at that index in our Folder array.

Run and make sure we have a list of folders, from 0 to 6, matching the names in the Tree diagram.

Next, output a message to the user:
"Which folder do you want to view the path of?"
And store their answer in an **int** variable. Whatever they enter will be the index of a Folder in the folder array.

First, try to just print the name of that folder before moving on:

```
     cout << folders[ choice ].GetName() << endl;
```

If this builds and runs correctly, continue on.

Now, we are going to start at the chosen folder and continue looping until we hit the **root** node. As we go through the chosen Folder and each parent, we will **cout** that folder's name.

Create a new pointer-variable: **currentFolder**. Set it equal to the address-of the chosen Folder.
The chosen Folder is a member of the array we created earlier – where the user's input choice is the element index.

```
     Folder* currentFolder = &folders[ choice ];
```

So we're going to create a while loop to step through each parent. How do we know when to end? We can tell because **root** has NULL set as its parent, and we can request each parent Folder through the **GetParentFolder** function of Folder.

Create a while loop. Its looping condition is while **currentFolder** is not equal to NULL.

Within the loop, output the **currentFolder's** name. Remember that currentFolder is a pointer to a class, and we will either have to de-reference it before accessing members, or use the **arrow-operator**.

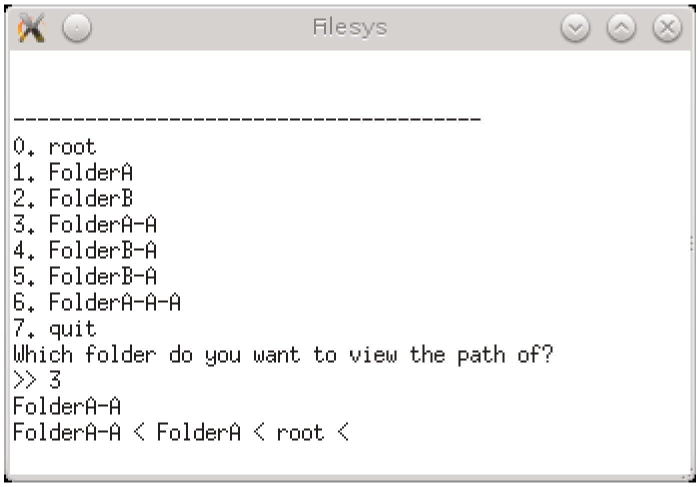
`(*ptrClass).member`	or	`ptrClass->member`

After we've outputted the name of the **currentFolder**, we need to re-assign currentFolder to the parent. If we don't, we will be stuck in this while-loop forever!

After the **cout**, assign a new value to **currentFolder:** its own parent. Use the **GetParentFolder** function to get the next parent.

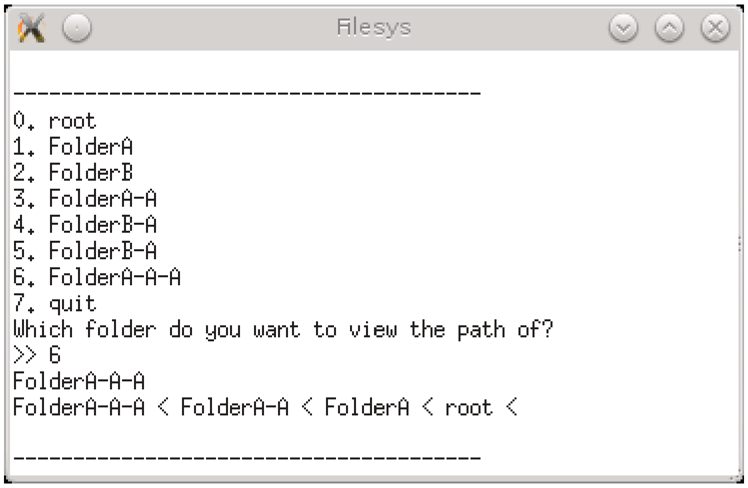
That is all we need to put in our while loop.

Save, build, and run. Make sure that selecting different options gives us the correct parents, according to the File-system table diagram.



```
----------------------------------------
0. root
1. FolderA
2. FolderB
3. FolderA-A
4. FolderB-A
5. FolderB-A
6. FolderA-A-A
7. quit
Which folder do you want to view the path of?
>> 3
FolderA-A
FolderA-A < FolderA < root <
```

Displaying parents of Folder A-A



```
----------------------------------------
0. root
1. FolderA
2. FolderB
3. FolderA-A
4. FolderB-A
5. FolderB-A
6. FolderA-A-A
7. quit
Which folder do you want to view the path of?
>> 6
FolderA-A-A
FolderA-A-A < FolderA-A < FolderA < root <

----------------------------------------
```

Displaying parents of Folder A-A-A

You now have a basic concept of how a file-system works. How do you think file/folder shortcuts are implemented in the operating system? How could we start at a parent-folder and step through all of its children and childrens'-children?
Trees are an important data-structure in computer science, and there are various strategies to traversing them effectively! This will be your next CS class after 201. :)