CS 202 - S22 - Assignment 2

Mr. Piotrowski

Function Overloading and Object Oriented Programming (classes) Intro

Macintosh, Apple, US, 1984
The innovative Macintosh—Apple's second attempt at a GUI-based personal computer, following the failure of the Lisa—was a small, self-contained personal computer with a much-improved, Alto-like graphical desktop. Graphic designers, artists, and educators quickly adopted it.

1 Assignment Introduction

The year is 1984 and all computer output dimensions are fixed size. That means all text is very tiny. The company you work for makes custom terminal programs for people that are looking to automate the mathematical parts of their lives. These people are often financial advisors and accountants.

Lately your products have been receiving complaints from clients who do not have very good sight. The font on the terminal is too small and many are experiencing eye-strain. You have an idea how to make the font bigger. You decide to make a proof of concept for your boss in hopes that the feature will be included in future products.

In this assignment, you will be writing a program that takes string input from files and prints them in a ”bigger” font. Obviously, the terminal default font size is unchangeable, so you will be printing large ascii representations of each letter, number, and punctuation. See below for an example of the word ”Font”.

```
        ########   ######   ##    ##  ########
        ##        ##    ##  ####  ##     ##
        #######   ##    ##  ## ## ##     ##
        ##        ##    ##  ##  ####     ##
        ##        ##    ##  ##   ###     ##
        ##         ######   ##    ##     ##
```

I have given you the file containing the ascii representations so you can easily match my output.

2 How Does this Relate to Object Oriented Programming?

I dare you to implement this as a spaghetti code in main! Let’s say you did, there would be the following disadvantages (*most likely*):

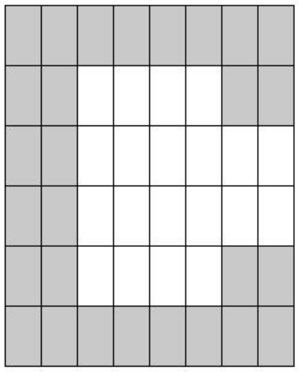
- It would take longer to code.
- Using this program to implement large font in other newer programs would be difficult, as you would have to selectively move sections of the spaghetti code.
- The spaghetti code may be close to unreadable. That just means competent programmers might not understand easily what your code does just by reading it.

Because we want to make this code portable and readable, we should understand that object-oriented programming is the way to go.

2.1 How Will We Package our Data?

2.1.1 Large Letters

We will define a few classes for this assignment. Our most important class is the **largeLetter**! Our program will be representing largeLetters as 2-Dimensional arrays (*a grid with cells. Each cell is either filled or empty - see below*).



#	#	#	#	#	#	#	#
#	#					#	#
#	#						
#	#						
#	#					#	#
#	#	#	#	#	#	#	#

Obviously, the terminal only gives us the power of the ascii table, so each "cell" is filled with a dense symbol like the pound sign (*hashtag for those who don't know what pound is...*).

Because we already have a string class that holds character arrays, we will use these strings to help us make the 2D array. Each letter is represented as a string array of size 6.

```
const int letterHeight = 6;
string rows[letterHeight];
```

Though this array looks only 1D, we know that a string is just an array of characters. Each string is 8 characters wide. To help understand what this looks like, imagine the string array "rows" looks like this:

```
{"########", "##    ##", "##      ", "##      ", "##    ##", "########"}
```

Though it does just look like jargon, if we print each element with an endl in between, it comes out looking like something!

```
{"########"
,"##    ##"
,"##      "
,"##      "
,"##    ##"
,"########"}
```

Now we can write code that looks like this:

```
largeLetter newLetter('F');
```

Obviously there is more code that we will write associated with the above statement, as a constructor will be called and the string array will be initialized, but from a code readers point of view this is very understandable. Consider the case where we do not use classes. All letters would look like this:

```
const int letterHeight = 6;
string newLetter[letterHeight];
```

Now consider the case where we do not use strings! The letters would look like this:

```
const int letterHeight = 6;
const int letterWidth = 8;
string newLetter[letterHeight][letterWidth];
```

Now that is gross. Just from looking at this, I am not sure what I am looking at.

2.1.2 Large Letter Words

Next we will implement a class called **largeLetterWord**! This holds an array of largeLetters that spell out a word. For example, the "Font" word above would be stored in a single **largeLetterWord** object. The largeLetter array would contain 4 letters. By packaging these things into classes, we will be able to write code that looks like this:

```
largeLetterWord newWord("Font");
```

If we did not use a class, all of our largeLetterWords would look like this instead:

```
const int letterHeight = 6;
const int wordLength = 4;
string newWord[wordLength][letterHeight];
```

That is a 3-Dimensional array because the strings are 8 characters each. If only using chars, the largeLetterWord would look like this:

```
const int letterHeight = 6;
const int letterWidth = 8;
const int wordLength = 4;
char newWord[wordLength][letterHeight][letterWidth];
```

Now imagine reading that in some code that you did not write. I am sure you would much rather see "largeLetterWord" instead.

Hopefully I have convinced you the importance of object oriented programming here. As I am sure you can guess, we will be printing these large letters and large words quite a bit. Having them wrapped in a class will allow us to write the printing code only once and bundle it with the objects, making our coding experience nice and easy.

3 Program Flow

This program is only a proof of concept so it will not do very much. The program's job is to open a file and convert all the words in the file to large font words. The program is run using either of the statements below:

```
./a.out inputFile.txt
```

```
./a.out inputFile.txt outputFile.txt
```

If one input file is provided, the program will print all large font words to the terminal using cout. In the case that an input and output file are provided, the program will print all large font words to the output file. Here is an example run using the following text file:

```
This is a test.
```

```
b346-15:S22 - A2 piotrj1$ ./a.out input0.txt
File Opened... Initializing Dictionary...

########  ##    ##  ########  ########
   ##     ##    ##     ##     ##
   ##     ########     ##     ########
   ##     ##    ##     ##           ##
   ##     ##    ##     ##     ##    ##
   ##     ##    ##  ########  ########


########  ########
   ##     ##
   ##     ########
   ##           ##
   ##     ##    ##
########  ########


   ##
  ####
 ##  ##
########
##    ##
##    ##


########  ########  ########  ########
   ##     ##        ##          ##
   ##     #######   ########    ##
   ##     ##              ##    ##
   ##     ##        ##    ##    ##
   ##     ########  ########    ##        ##


b346-15:S22 - A2 piotrj1$
```

And that is it. The program exits.

4 Coding The Assignment

For this assignment you are provided a skeleton code. This means I am giving you some partially completed code. Your only job is to fill in the sections of code that are empty. This style of assignment should be new to you, so feel free to ask questions. Unfortunately, there is not much room for your own creativity here, you will have to follow my instruction.

4.1 Files

There are 3 files in this assignment:

- largeLetter.h
- largeLetter.cpp (**your code here**)
- main.cpp (**your code here**)

You will notice this is a different style than before. Usually, you just have main.cpp with everything in it. In a project that has many classes and functions, having a single file for everything is detrimental. Classes and class sets are often separated into their own files. There are 2 components:

- The header file (*.h or .hpp*)
- The implementation file (*.cpp*)

4.1.1 Header Files

Header files contain the headers/prototypes for a class and/or functions. This is similar to having the function and class prototypes above main. It is easy to see the class/functions and what they do, as they are not cluttered with nasty code.

4.1.2 Implementation Files

The implementation file is where the definitions go. This is a cpp file as it will be compiled every time with main.cpp. The benefit of having the implementation file is as follows. Pretend I am your project manager. I have designed the datatype and I have coded the class and function prototypes for it. Now I can make my grunts (*you*) write the code for those functions. In this way, you are forced to abide by the class structure.

In this assignment I have given you the classes and function prototypes. Your job is to write the functions.

4.1.3 Compiling with Multiple Files

To begin, the proper include statements need to be in each file. Study all 3 files here closely. You will notice that largeLetter.h includes all of the libraries it needs. These are wrapped up with some code that looks like this:

```
#ifndef STRING_H
#define STRING_H
#include <string>
#endif
```

This basically says:

- if this name STRING_H has not been defined before:
  1. define STRING_H
  2. include string library

Now let's say that main.cpp includes both the string and the "largeLetters.h" libraries (*it does!*). If main uses the same #ifndef code, it will not re-include the string library.

You will see that main.cpp include the "largeLetters.h" file. This means that main.cpp has access to all the classes and functions listed in that file.

largeLetters.cpp includes only "largeLetters.h". All of the libraries needed for the code (*like string, fstream*) were included in the header file.

When you are ready to compile:

```
g++ -std=c++11 main.cpp largeLetters.cpp
```

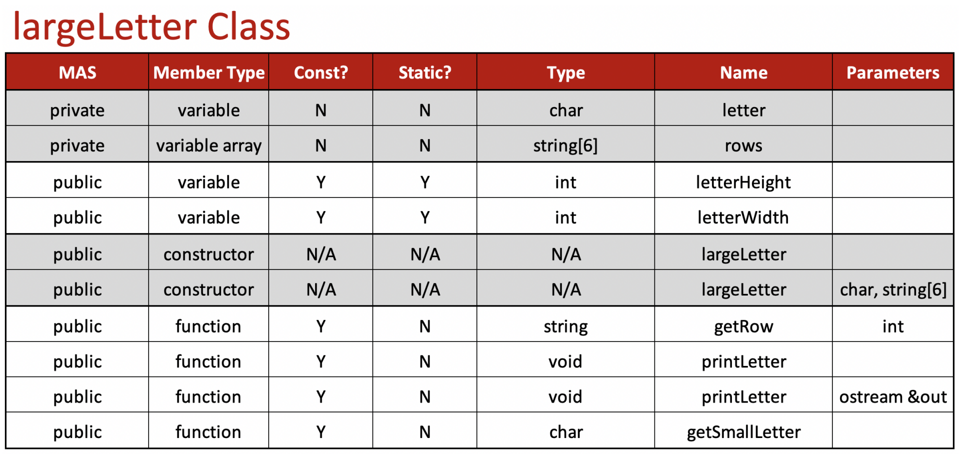
Assuming you have 5 implementation files for some assignment in the future, the compile statement would look like this:

```
g++ -std=c++11 main.cpp file1.cpp file2.cpp file3.cpp file4.cpp
    file5.cpp
```

5 Coding

Note: A better name for these classes might be "largeCharacter" as numbers and punctuations are also included. Please do not be confused by the name.

5.1 largeLetter Class



largeLetter Class

MAS	Member Type	Const?	Static?	Type	Name	Parameters
private	variable	N	N	char	letter	
private	variable array	N	N	string[6]	rows	
public	variable	Y	Y	int	letterHeight	
public	variable	Y	Y	int	letterWidth	
public	constructor	N/A	N/A	N/A	largeLetter	
public	constructor	N/A	N/A	N/A	largeLetter	char, string[6]
public	function	Y	N	string	getRow	int
public	function	Y	N	void	printLetter	
public	function	Y	N	void	printLetter	ostream &out
public	function	Y	N	char	getSmallLetter	

- **static const int letterHeight** - This is the grid height for a largeLetter. For this assignment, each largeLetter is 6 chars tall.
- **static const int letterWidth** - This is the grid width for a largeLetter. For this assignment, each largeLetter is 8 chars wide.
- **char letter** - This is the actual ASCII character that the large letter is representing. For the largeLetter 'F', you would find that 'F' is stored in this member variable. If there is no largeLetter initialized for the object, this char contains NULL character '\0'.
- **string rows[letterHeight]** - This is the array where the ASCII "art" is stored for each largeLetter (*see page 3*). There will be 6 strings in this array. Each string is 8 chars long. When printed back to back with an endl in between each, it will shape out the letter on the terminal.
- **largeLetter()** - This is the default constructor for the class. The constructor is only called if the programmer tries to initialize an empty large letter. You will want to initialize the **letter** member to be null character and each string in the **rows** array to be a string of width 8. It could be done like this:

  ```
  rows[i] = string(8,' '); // 8 is the length of the string
  ```

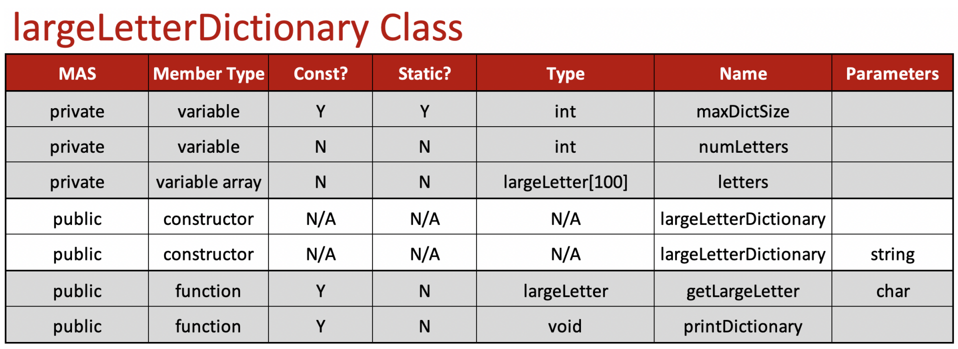
  This string constructor initializes a new string with 8 spaces. You could also write it like this because 32 is the decimal value for the space character.

  ```
  rows[i] = string(8,32); // 8 is the length of the string
  ```

- **largeLetter(char, string[])** - This constructor will initialize the **letter** member with the char parameter. It will also initialize each string in the rows array with each string in the parameter array. You can assume that each string in the parameter array will be of length 8.
- **string getRow(int) const** - This returns the string in the rows array located at the int index parameter. You may wish to include some error checking here to confirm that the int parameter is in bounds.
- **void printLetter(ostream &) const** - This will print a single letter to to the ostream. Just print each row in the rows array with an endline after each row.
- **void printLetter() const** - This will print a single letter to to the screen. Does the same thing as the function above, but it defaults to using cout. Perhaps call the function about and pass cout as a parameter.
- **char getSmallLetter() const** - A getter function that should just return a copy of the **letter** member variable.

5.2 largeLetterDictionary Class

This class was not mentioned before, but we need it to make life easy. This dictionary is going to hold a mapping from the small regular ASCII character to the larger representation of it. With the dictionary, it will be easy to look-up a character in the dictionary to get the larger version of it.


largeLetterDictionary Class

MAS	Member Type	Const?	Static?	Type	Name	Parameters
private	variable	Y	Y	int	maxDictSize	
private	variable	N	N	int	numLetters	
private	variable array	N	N	largeLetter[100]	letters	
public	constructor	N/A	N/A	N/A	largeLetterDictionary	
public	constructor	N/A	N/A	N/A	largeLetterDictionary	string
public	function	Y	N	largeLetter	getLargeLetter	char
public	function	Y	N	void	printDictionary	

- **static const int maxDictSize** - This is the maximum size for a dictionary. In this assignment, the dictionary will only hold up to 100 characters.
- **int numLetters** - This is the number of letters total in the dictionary. For example, the dictionary can hold up to 100 largeLetters, but it may only have 35.
- **largeLetter letters[maxDictSize]** - This is the array of largeLetters. Each letter in the "Letters.txt" file will have a place in this array.
- **largeLetterDictionary()** - The default constructor for the dictionary. Consider initializing the **numLetters** to 0.
- **largeLetterDictionary(string)** - The parameter constructor for the dictionary. This constructor will open the "Letters.txt" or some file that is in a similar format. The "Letters.txt" format is as follows:

```
A
   ##
  ####
 ##  ##
########
##    ##
##    ##
B
#######
##    ##
#######
##    ##
##    ##
#######
C
```

```
########
##    ##
##
##
##    ##
########
```

First, try to open the file using the file name that was passed as a parameter. If the file opens succesfully, print:

```
    "File Opened... Initializing Dictionary..."
```

Now begin reading the file. The first thing you read is the character that is being represented. Let's call this the **char smallLetter**. The following 6 lines after the smallLetter is the large ASCII representation for that smallLetter. So, for every 7 lines, first read the smallLetter then the next 6 rows. You will then want to create a largeLetter object using the data that you just read and store it in the largeLetter array.

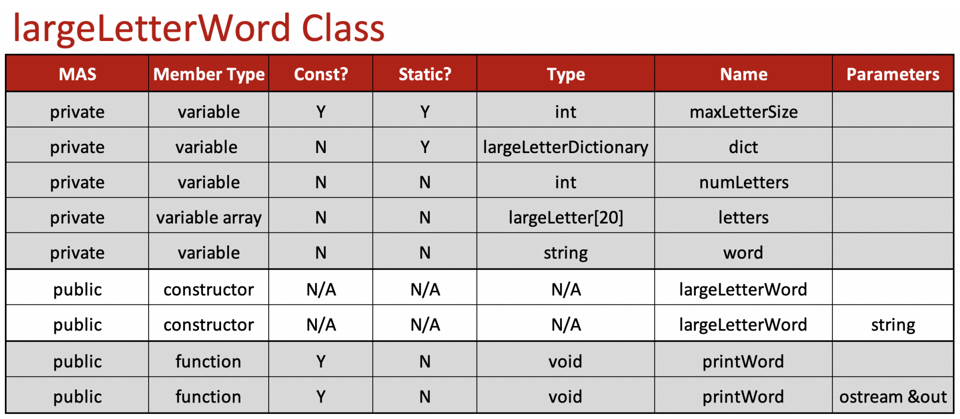
Definitely use getline here as you will also want to grab the spaces following some of the ASCII "art" in order to preserve spacing.

- **largeLetter getLargeLetter(char) const** - This function will return a copy of the largeLetter object corresponding to the char that was passed as a parameter. You will likely have to search the largeLetter array until you find the correct largeLetter. If you do not find it, you can return a default largeLetter (*a largeLetter that was created with a default constructor*).

- **void printDictionary() const** - A function that will print your whole dictionary in this format:

```
--------------
| Letter [R] |
--------------
#######
##    ##
#######
##  ##
##   ##
##    ##
--------------
| Letter [S] |
--------------
########
##
########
      ##
##    ##
########
```

5.3 largeLetterWord Class



largeLetterWord Class

MAS	Member Type	Const?	Static?	Type	Name	Parameters
private	variable	Y	Y	int	maxLetterSize	
private	variable	N	Y	largeLetterDictionary	dict	
private	variable	N	N	int	numLetters	
private	variable array	N	N	largeLetter[20]	letters	
private	variable	N	N	string	word	
public	constructor	N/A	N/A	N/A	largeLetterWord	
public	constructor	N/A	N/A	N/A	largeLetterWord	string
public	function	Y	N	void	printWord	
public	function	Y	N	void	printWord	ostream &out

- **static const int maxLetterSize** - This the he maximum letters that we will allow for any word. After a quick google search, **20+** letters in a word is pretty uncommon, so we will limit our words to 20 letters.
- **static largeLetterDictionary dict** - A static dictionary for all largeLetterWords to help translate regular strings into their large font. This static variable will need to be initialized globally as described in class (*anywhere in the largeLetter.cpp file*). When initializing it use the parameter constructor to make it read from the "Letters.txt" file. You are allowed to hard-code that filename.
- **int numLetters** - The number of letters in the word. For example, if we were storing "Font", the numLetters would be 4.
- **largeLetter letters[maxLetterSize]** - The array to hold the large letter word. For example, if we were to be storing the word "FONT", the letters array would contain a largeLetter F in the 1st spot (*index 0*), a largeLetter O in the second spot (*index 1*), a largeLetter N in the third spot (*index 2*), and a largeLetter T in the fourth spot (*index 3*).
- **string word** - A simple string variable to hold the word being represented by large font. For example, if making a largeLetterWord for "FONT", "FONT" would be stored here.
- **largeLetterWord()** - The default constructor for a largeLetterWord. Initialize numLetters to be 0.
- **largeLetterWord(string)** - The parameter constructor for a largeLetterWord. This constructor should take the string parameter and use it to initialize both the **word** member and the **numLetters** member. Next, it should convert the string parameter into large letters. This implies looking

up each character of the string in the dictionary, getting the largeLetter corresponding to that character, and adding it to the **letters** array.

- **void printWord(ostream out) const -** This function needs to print the word in the below format to the ostream.

```
        ########   ######   ##    ##  ########
        ##        ##    ##  ####  ##     ##
        #######   ##    ##  ## ## ##     ##
        ##        ##    ##  ##  ####     ##
        ##        ##    ##  ##   ###     ##
        ##         ######   ##    ##     ##
```

  This is a little more involved than just printing every letter. You will have to print the first row of every letter, where each row is separated with 2 spaces. Once complete, print an endline and move to the next row. Do this for all 6 rows. Once complete, print two endlines to the out stream to properly space the words apart.

- **void printWord() const -** This does the same thing as above but it will default to using cout only. Perhaps you could just call the function above and pass cout as the parameter.

5.4 main.cpp

I also want you to fill out main. Main will open one or two files provided via the command line. The first file is always the input file that will be translated into large font. The second file, if provided, will be the output file where the large font will go. So, in order of operations, main should:

1. Check the argc for 1 or 2 files. If there are more or less, print an error message and return 0.
2. Next, open the input file. Also open the output file if there was one provided.
3. Begin reading in the file word by word. Using cin will be sufficient here. For each word in the file:
   (a) Convert every character to upper case.
   (b) Make a largeLetterWord using the upper case word.
   (c) Print it either to the terminal or the output file.
4. Once the file is completely read, close it. Close the output file too if needed.
5. Return 0

6 Compiling and Submitting

Compile with:

```
g++ -std=c++11 main.cpp largeLetters.cpp
```

Execute with:

```
./a.out inputFile.txt
```

or

```
./a.out inputFile.txt outputFile.txt
```

7 Comprehension Questions

Please answer these questions through the webcampus quiz.

1. Explain what a static member is. In your own words, why do you think the largeLetterDictionary object in the largeLetterWord class is static and not instanced?
2. Explain what a const member is.
3. Based on the classes described above, list all the function overloads present in this assignment.