COMP 5421 Assignment 1 Due by May 22, 2020

1 Purpose

No matter how hard you try, you just can't completely escape pointers and references in the C++ language because the presence of pointers and references throughout the language is pervasive. The more you understand the inner working of pointers and references in C++, the more equipped you are to fully appreciate and utilize enhanced features of "modern" C++ (currently C++20, which is primarily based on the revolutionary C++11 standard).

This assignment is designed to get you started with C++, giving you practice with arrays, pointers, dynamic memory allocation and deallocation, and writing classes. Most students new to C++ might find this first assignment a bit challenging; however, they will no doubt feel a sense of accomplishment in their own efforts after completing the assignment.

2 Your Task

Implement a class, named `Menu`, that models a menu of options that is typically used in menu-driven text-based interactive programs, where the user is first presented with a list of options to choose from and is then prompted to enter a value corresponding to an option, very similar to menu-based voice interfaces.

In this assignment, the string representation of a `Menu` object looks like the textual pattern shown at right and includes four items:

```
Top message
  1: option one
  2: option two
  3: option three
   ...
  n: option n
Bottom message
??
```

- the string literal "`??`"
- two `Text` objects representing the top and bottom messages
- a dynamic list of `Text` objects representing a list of options on the menu

where `Text` is a class that manages the text (character strings) of the options and messages.

The string literal "`??`" provides the minimal string representation of a `Menu` object regardless of the presence or absence of the other three items.

Since this might be your first encounter with programming in C++, this assignment will provide you with detailed UML class diagrams of classes `Text` and `Menu` as well as presenting sample program runs.

3 Reinventing the wheel

In C++, there are things you might only do once, two of which are implementing a `Text` and a `Menu` classes rather than simply using the highly optimized std::string and std::vector classes, respectively.

The one time that you might implement a `Text` or a `Menu` class is when you are new to C++, being already fluent in another programming language and looking for an effective way to learn and practice the basics of classical C++, including:

- dynamic memory management
- default constructor
- conversion constructor
- the classical "Big Three"
  - copy constructor
  - virtual destructor
  - assignment operator overload
- the insertion operator overload
- member `friend`s of the class
- C-strings (C-style strings of characters)
- using <cstring> supported C-string functions
- dynamic arrays of characters
- dynamic arrays of user defined objects

3.1 C-strings

A C-string is a C-style string of characters.

Although C-strings and the Java string literals look alike, they are fundamentally different.

A C-string is an array of characters that always ends with the null byte character `‘\0’`, the character with ASCII code 0. So looking at the C-string `"abc"`, you should see the character array $\boxed{\texttt{a}}\boxed{\texttt{b}}\boxed{\texttt{c}}\boxed{\texttt{\textbackslash 0}}$. The type of `"abc"` is `const char*`, where the `const` part implies that you cannot change the contents of the character array representing `"abc"`.

4 Class Text

Described by the UML class diagram below[1], class Text represents the text of the messages and options in a menu.

Note that every non-static member function of a class has a this pointer, which points to *this, the object that invoked the function. In other words, every non-static member function of a class has access to *this, “the this object”(even though this is a pointer, not an object.)

Text	The name of this class
− pStore: char *	Points to storage of the C-string in this object
+ virtual ∼Text() :	Releases dynamic storage in use by this object
+ Text() :	Default constructor; equivalent to Text("")
+ Text(txt : const Text &) :	Copy Constructor; deep-copies txt into the object being constructed (i.e., the this object)
+ Text(pCstr : const char *) :	Conversion Constructor (C-string to Text)
+ assign(pCstr : char *) : void	Assigns a C-string to this object, replacing its current contents
+ assign(txt : const Text &) : void	Assigns a Text object to this object, replacing its current contents
+ operator=(txt : const Text &) : Text&	Assignment operator= overload. Assigns the right hand side operand to this object, replacing its current contents
+ append(pCstr : const char *) : void	Appends a C-string to this object, extending its contents
+ append(txt : const Text &) : void	Appends txt's text to this object, extending its contents
+ clear() : void	Erases the contents of the C-string in this object, which becomes an empty C-string
+ length() const : int	Returns the length of this object's text
+ isEmpty() const : bool	Determines whether this object represents a C-string of length zero
+ getCstring() const : const char *	returns pStore as a const pointer (why const?)

[1] Please note that the text outside the UML box is not an UML element and is provided for added information.

4.1 Notes

There's a lot happening in our Text class:

(a) C-strings

(b) Objects of Text each use dynamic memory allocation. Dynamic memory, also called the *heap*, refers to the memory that is is allocated at run time.

(c) The destructor ~Text() is called automatically when an object of its class goes out of scope. The destructor provides a "last call" opportunity for you to deallocate (free) any dynamic memory storage still in use by that object. If you fail to take the opportunity to free dynamic storage, your program will suffer from what is known as "memory leak".

(d) Some member functions can benefit from code sharing, providing an opportunity for you to practice code refactoring. Recall that a constructor can delegate its tasks to another constructor.

(e) Add the following function prototype outside and after the declaration of class Text in the Text.h header file as follows

```
// ostream& operator<<(ostream & sout, const Text& txt); // or simply
ostream& operator<<( ostream &, const Text& ); // no param names required here
```

and implement it in your Text.cpp source file as follows

```
ostream& operator<<(ostream & sout, const Text& txt)
{
   sout << txt.getCstring();
   return sout;
}
```

(f) The <cstring> header file supports several C-string functions. For example, you can use the strcpy() and strcat() functions to copy and append a string to a character array, respectively. For another example, you can use the strlen() function to determine the length of a C-string, and you can use the strcmp() to compare two C-strings.

4.2 Heads Up

Transitioning from the classical C++ to modern C++, future course assignments will prohibit the use of character arrays, and the functions supported in the <cstring> header file.

4.3 Test Drive

Use the following function to test drive your Text class.

```
// Function prototypes (declarations)
void demoText();     // demonstrates Text objects
void demoMenu();     // demonstrates Menu objects

int main()
{  demoText();   //demoMenu();
   return 0;
}
void demoText()
{  Text t1;                      // defalt constructor
   Text t2("quick brown fox");   // conversion constructor
   Text t3{ t2 };                // copy constructor
   cout << "t1: " << t1 << endl; // operator<< overload
   cout << "t2: " << t2 << endl;
   cout << "t3: " << t3 << endl;

   t1.append("The ");            // append a given C-string to t1's C-string
   cout << "t1: " << t1 << endl;
   t1.append(t2);                // append t2's C-string to t1's C-string
   cout << "t1: " << t1 << endl;

   t2 = Text(" jumps over ");    // assignment operator overload
   cout << "t2: " << t2 << endl;

   t3.assign("a lazy dog");      // assign a given C-string to t3's C-string
   cout << "t3: " << t3 << endl;

   t1.append(t2);                // assign t2's C-string to t1's C-string
   cout << "t1: " << t1 << endl;
   t1.append(t3);                // append t3's C-string to t1's C-string
   cout << "t1: " << t1 << endl;
   return;
}
```

The output generated by the function demoTest() should look as follows:

```
t1:
t2: quick brown fox
t3: quick brown fox
t1: The
t1: The quick brown fox
t2:  jumps over
t3: a lazy dog
t1: The quick brown fox jumps over
t1: The quick brown fox jumps over a lazy dog
```

5 Class Menu

Let's see what we expect from the objects of our Menu class.

```
Top message
  1: option one
  2: option two
  3: option three
   ...
  n: option n
Bottom message
??
```

1. Create an empty menu

   ```
   Menu menu;
   ```

   This declaration implies that Menu must have define a default constructor.

2. Print an empty menu

   ```
   cout << menu << endl;
   ```

   This statement implies that the insertion operator<< must be overloaded, either as a friend member[2] of Menu or as a free function.[3]

   Here is how the string representation of an empty menu should look like:

   ```

   ??
   ```

3. Display the menu and read user's input:

   ```
   int choice = menu.read_option_number();
   cout << "you entered: " << choice << endl;
   ```

   At this point the user is expected to enter an an integer. There are two cases:

   a: menu's option list is not empty

   The user must enter a valid option number; otherwise, menu.read_option_number() will reject the input value, repeatedly displaying the menu until the user enters a valid option number.

   b: menu's option list is empty

   Since the option list is empty, the user is allowed to enter any integer value.

   ```

   ?? 1234
   you entered: 1234
   ```

[2] A friend function of class is NOT a member function (i.e., it has no this pointer) but objects of the host class passed to or defined in the friend function can access private members of the host class.

[3] A free function is a stand-alone function that does not belong to any class.

4. Add an option to our menu:

```
menu.push_back("Pepsi");
cout << menu << endl;
```

```

   1: Pepsi
??
```

The top line of a menu display is always preceded by a blank line for better readability.

5. Let's add a couple of more options to our menu:

```
menu.push_back("Apple juice");
menu.push_back("Root beer");
choice = menu.getOptionNumber();
cout << "you entered: " << choice << endl;
```

```

   1: Pepsi
   2: Apple juice
   3: Root beer
?? -1
Invalid choice -1. It must be in the range [1, 3]

   1: Pepsi
   2: Apple juice
   3: Root beer
?? 5
Invalid choice 5. It must be in the range [1, 3]

   1: Pepsi
   2: Apple juice
   3: Root beer
?? 1
you entered: 1
```

6. Let's set the top and bottom messages:

```
menu.set_top_message("Choose your thirst crusher");
menu.set_bottom_message("Enter a drink number");
cout << menu << endl;
```

```

Choose your thirst crusher
   1: Pepsi
   2: Apple juice
   3: Root beer
Enter a drink number
??
```

7. Let's remove the last option and then insert a new option at number 2:

```
menu.pop_back();                        // remove the last option
menu.insert(2, "Iced tea with lemon");  //  this will be option 2
choice = menu.read_option_number();
cout << "you entered: " << choice << endl;
```

```
Option inserted successfully!

Choose your thirst crusher
   1: Pepsi
   2: Iced tea with lemon
   3: Apple juice
Enter a drink number
?? 2
you entered: 2
```

8. The menu object let's you remove an option by option number:

```
menu.pop_back();  // remove the last option
menu.remove(1);   // remove the first option
cout << menu << endl;
```

```

Choose your thirst crusher
   1: Iced tea with lemon
Enter a drink number
??
```

9. The following code segment removes the only remaining option, leaving the menu with an empty option list:

```
menu.pop_back();
cout << menu << endl;
```

```

Choose your thirst crusher
Enter a drink number
??
```

10. Here is our final example:

```
menu.clear_bottom_message();
menu.set_top_message("Who Says You Can't Buy Happiness?\n"
     "Just Consider Our Seriously Delicious Ice Cream Flavors");
menu.set_bottom_message("Enter the number of your Happiness! ");
menu.push_back("Bacon ice cream!");
menu.push_back("Strawberry ice cream");
menu.push_back("Vanilla ice cream");
menu.push_back("Chocolate chip cookie dough ice cream");
choice = menu.read_option_number();
cout << "you entered: " << choice << endl;
```

```

Who Says You Can't Buy Happiness?
Just Consider Our Seriously Delicious Ice Cream Flavors
   1: Bacon ice cream!
   2: Strawberry ice cream
   3: Vanilla ice cream
   4: Chocolate chip cookie dough ice cream
Enter the number of your Happiness!
?? 3
you entered: 3
```

Menu	The name of this class
− option_list : Text*	stores a pointer to dynamic array of `Text` objects managed by `this` object
− capacity : int	The current length of the options list dynamic array
− count : int	The actual number of options on the option list
− top_message : Text	The opening message
− bottom_message : Text	The closing message
− double_capacity() : void	Doubles the current capacity of the options list
+ Menu() :	Default constructor. Initializes top/bottom messages to empty strings, the options list to dynamic array of capacity 1, and count to 0.
+ Menu(mnu : const Menu&) :	Copy Constructor; deep-copies `mnu` into the object being constructed (i.e., the `this` object)
+ virtual ∼Menu() :	Destructor. Releases dynamic storage in use by the options list
+ Menu& operator=(m : const Menu&) :	Assignment operator= overload. Deep copies the right hand side operand into `this` object.
+ insert(index : int , option : Text&) : void	Inserts `option` at position `index`, shifting all options at or past `index` over to the right by one position.
+ push_back(pOption : char *) : void	Adds supplied option to the end of the option list
+ push_back(option : const Text&) : void	Adds supplied option to the end of the option list
+ remove(index : int) : void	Removes an option from the list at given `index`; shifts all options to the right of `index` left by one position.
+ size() const : int	Returns the number of options in the option list.
+ capacity() const : int	Returns the current capacity if the options list
+ pop_back() : void	Removes the last option in the list
+ get(k : int) : Text	Return the k'th option
+ toString() const : Text	Returns a `Text` object storing a string representation of this menu
+ read_option_number() : int	Displays this menu and then reads and returns a valid option number
+ set_top_message(m : const Text&):void	Sets top message to `m`
+ set_bottom_message(m : const Text&): void	Sets bottom message to `m`
+ clear_top_message() : void	sets top message to empty text
+ clear_bottom_message() : void	sets bottom message to empty text
+ isEmpty() const : bool	Determines whether this menu's option list is empty

6 Requirements

Your implementation

- must store the Text objects using raw C-arrays, which must be allocated dynamically using new, and deallocated using delete.
- must use an initial capacity of 1 for the option list in the default constructor. This will speedup testing of your double_capacity() method (1 to 2 to 4 to 8, etc.)
- must overload the insertion operator<<.
- may not use the C++ string class, except for string's c_str() member function,
- may use only the functions strcpy, strcat, strcmp, and strlen from the <cstring> header file,
- may use any "String conversion" functions from <cstdlib> and any function from <cctype>.
- may introduce any number of private member functions of your own to facilitate your tasks.

7 Deliverables

Create a a new folder that contains the files listed below, then compress (zip) your folder, and submit the compressed (zipped) folder *as instructed* in the course outline.

1. Header files: Menu.h and Text.h
2. Implementation files: Menu.cpp, Text.cpp, menuDriver.cpp
3. A README.txt text file (see the course outline).

8 Marking scheme

60%	Program correctness
20%	Proper use of pointers, dynamic memory management, and C++ concepts. No C-style memory functions such as malloc, alloc, realloc, free, etc. No C-style coding.
10%	Format, clarity, completeness of output
10%	Concise documentation of nontrivial steps in code, choice of variable names, indentation and readability of program