1 Objectives

This assignment is designed to get you started with using C++ as an implementation tool, giving you practice with arrays, pointers, dynamic memory allocation and deallocation, and with writing classes.

If you are new to the C++ language, you might find this assignment challenging at first while learning about pointers, references, dynamic memory, etc. However, you will be happy to know that these notions will become second nature to you very quickly, and that you will hopefully feel pleased about the knowledge you will gain through these assignments.

2 Your Task

Implement a class, named `Menu`, that models menus used in text-based menu-driven 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.

The string representation of the `Menu` objects of interest in this assignment look like the textual pattern shown at right.

```
Opening message
Top prompt
   (1) option one
   (2) option two
   (3) option three
   (..)  ...
   (n) option n
Closing message
?-> Bottom prompt
```

A `Menu` object includes of six items:

- the string literal "`?->`",
- four `Text` items representing the top/bottom prompts and the opening/closing messages, and
- a dynamic list of `Text` items representing the menu's list of options,

where `Text` is a class that represents character strings.

The string literal item "`?->`" provides the minimal string representation for a `Menu` object regardless of the presence or absence of the other five items.

Assuming that, for most of you, this might be a 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 Class Text

Represents a character string and provides a few operations including those involved in the following code segment:

```
int demoText()
{
   Text t1("Welcome to C++"); // conversion constructor
   Text t2; // defalt constructor
   Text t3{ t1 }; // copy constructor
   cout << "t1: " << t1 << endl; // operator<< overload
   cout << "t2: " << t2 << endl;
   cout << "t3: " << t3 << endl;

   t2.set(" Programming"); // set t2's text to " Programming"
   cout << "t2: " << t2 << endl;

   t3.set(t1); // set t3's text to t1's text
   cout << "t3: " << t3 << endl;

   t1.append(" Programming"); // append the c-string " Programming" to t1's text
   cout << "t1: " << t1 << endl;

   t3.append(t2); // append t2's text to t3's text
   cout << "t3: " << t3 << endl;

   return 0;
}
```

output

```
t1: Welcome to C++
t2:
t3: Welcome to C++
t2:  Programming
t3: Welcome to C++
t1: Welcome to C++ Programming
t3: Welcome to C++ Programming
```

For this part, you will implement the following UML class diagram for class `Text`:

Text		
–	`*text: char`	points to where the text in this object is stored
+	`virtual ~Text() :`	Destructor. Releases storage in use by this object.
+	`Text() :`	Default Constructor. same as Text("")
+	`Text( t : const Text & ) :`	Copy Constructor.
+	`Text( t : const char * ) :`	Conversion Constructor
+	`set( t : char * ) : void`	Sets this object's text to *t
+	`set( t : const Text &) : void`	Sets this object's text to that of t
+	`operator=( t : const Text & ) : Text &`	Assignment operator= overload.
+	`append( t : const char * ) : void`	appends *t's text to this object's text.
+	`append( t : const Text & ) : void`	appends t's text to this object's text.
+	`length() const : int`	Return the length of this object's text.
+	`isEmpty() const : bool`	Determines whether this object's text is empty.

3.1 Specific Requirements

Your implementation

- must store the character strings using raw C-arrays, which must be allocated dynamically using `new`, and deallocated using `delete`.
- must overload the insertion `operator<<`.

4 Class Menu

For simplicity, this class provides only a default constructor that creates an default menu. The string representation of a default menu is the string literal "`?->`".

Class Menu overloads the `operator<<` to display the string representation of a menu. For example:

```
1 Menu menu; // an empty menu
2  cout << menu << endl;
```

Output

```
1 ?->
```

Obviously, it would be pointless to create a menu object and just display it. The primary purpose of a menu object is to interact with the user: it displays the menu and expect the user to enter an input value.

The range of input values acceptable by a menu object depends on the number of options in a menu's option list. If the menu's option list is not empty, then the user must enter an integer in the range 1 through the highest option number in the list.

However, If the menu's option list is empty, then the user can enter any integer:

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

Output

```
2  ?-> 1234
3  you entered: 1234
```

The `read_option_number()` method displays the menu (the same as `operator<<`) and then reads and returns the integer input. Moreover, if the menu's option list is not employ, `read_option_number()` validates the input values, repeatedly rejecting all out-of range input numbers until the user enters a valid option number.

Both `read_option_number()` and `operator<<` delegate their common task of displaying the menu to another member function named `toString()` that returns a `Text` representation of the menu. Thus, the `cout` statements in the following code segment each display the same output.

```
cout << menu << endl; // operator<< calls toString() internally on menu
choice = menu.read_option_number(); // read_option_number() calls toString() internally
cout << menu.toString() << endl; // menu calls toString() directly
Text t = menu.toString(); // injects menu's string representaion into t, a Text object
cout << t << endl; // same display as all of the above
```

Let's add an option to our menu and then print it:

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

Output

```
4
5     (1) Pepsi
6
7  ?->
```

In a menu with a non-empty option list the display of the option list is preceded and followed by blank lines for better readability.

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;
```

Output

```

   (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

?-> 2
you entered: 2
```

Class `Menu` provides member function to set the prompts in the menu:

```
menu.set_top_prompt("Choose your thirst crusher: ");
menu.set_bottom_prompt("Enter a drink number: ");
cout << menu << endl;
```

Output

```
Choose your thirst crusher:

   (1) Pepsi
   (2) Apple juice
   (3) Root beer

?-> Enter a drink number:
```

The following example shows how to remove the last option and then insert a new option at number 2:

```
14 menu.pop_back(); // remove the last option
15 menu.insert(1, "Iced tea with lemon"); // this will be option 2
16 choice = menu.read_option_number();
17 cout << "you entered: " << choice << endl;
```

Output

```
38 Choose your thirst crusher:
39
40    (1) Pepsi
41    (2) Iced tea with lemon
42    (3) Apple juice
43
44 ?-> Enter a drink number: 2
45 you entered: 2
```

To remove any of the options, the class implements a `remove(int)` member:

```
18 menu.pop_back(); // remove the last option
19 menu.remove(0); // remove the first option (index k indexes option k+1)
20 cout << menu << endl;
```

Output

```
46 Choose your thirst crusher:
47
48    (1) Iced tea with lemon
49
50 ?-> Enter a drink number:
```

The following code segment adds opening and closing messages to the menu:

```
21 menu.set_top_message("Quench your thirst with our fine drinks");
22 menu.set_bottom_message("Time to obey your thirst!");
23 cout << menu << endl;
```

Output

```
Quench your thirst with our fine drinks
Choose your thirst crusher:

   (1) Iced tea with lemon

Time to obey your thirst!
?-> Enter a drink number:
```

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

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

Output

```
Quench your thirst with our fine drinks
Choose your thirst crusher:
Time to obey your thirst!
?-> Enter a drink number:
```

Here is our final example:

```
menu.set_top_message("Who Says You Can't Buy Happiness?");
menu.clear_bottom_message();
menu.set_top_prompt("Just Consider Our Seriously Delicious Ice Cream Flavors for Summer ");
menu.set_bottom_prompt("Enter the number of your Happiness Flavor: ");
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.getOptionNumber();
cout << "you entered: " << choice << endl;
```

Output

```
Who Says You Can't Buy Happiness?
Just Consider Our Seriously Delicious Ice Cream Flavors for Summer

   (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 Flavor: 3
you entered: 3
```

A UML class diagram for class `Menu` is presented as follows:

Menu		The class `Menu`
–	`option_list : Text*`	The dynamically allocated array storing the options list
–	`count : int`	The number of options in the list
–	`max_list_size : int`	The maximum number of options supported by current options list
–	`top_prompt : Text`	The top prompt
–	`bottom_prompt : Text`	The bottom prompt
–	`top_text : Text`	The opening message
–	`bottom_text : Text`	The closing message
–	`double_capacity() : void`	Doubles the current capacity of the options list
+	`Menu() :`	Constructor. Initializes the object and allocates a small options list (See 4.1.)
+	`Menu( otherMenu : const Menu & ) :`	Copy constructor
+	`virtual ~Menu() :`	Destructor. Releases dynamic storage in use by the options list
+	`Menu& operator =(menu : const Menu & ) :`	Overloads the assignment operator
+	`insert( index : int , option : Text & ) : void`	Inserts `option` at position `index`; shifts 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 ) : Text`	Removes an option from the list at given `index`; shifts all options to the right of `index` left by one position. Returns the removed option.
+	`size() const : int`	Returns the number of options in the option list.
+	`capacity() const : int`	Returns the maximum number of options that the menu can support until it needs to expand.
+	`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_prompt( t : const Text & ) : void`	Sets top prompt to the supplied prompt `t`
+	`set_bottom_prompt( t : const Text &):void`	Sets bottom prompt to the supplied prompt `t`
+	`set_top_message( m : const Text & ):void`	Sets opening message to `m`
+	`set_bottom_message( m :const Text &):void`	Sets closing message to `m`
+	`clear_top_prompt() : void`	Removes top prompt
+	`clear_bottom_prompt() : void`	Removes bottom prompt
+	`clear_top_message() : void`	Removes opening message
+	`clear_bottom_message() : void`	Removes closing message
+	`isEmpty() const : bool`	Returns `true` this menu's option list is empty

4.1 Specific 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 the initial capacity of 1 when an object is created by 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<<`.

5 General Requirements

Your implementation

- may not use the C++ `string` class, except for `c_str()` and `data()`,
- may use only the functions `strcpy` 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.

6 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).

7 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