CS 202 - Computer Science II

Project 8

Due date (FIXED): Wednesday, 4/15/2020, 11:59 pm

Objectives: The main objectives of this project are to test your ability to create and use list-based dynamic data structures. A review of your knowledge to manipulate dynamic memory, classes, pointers and iostream to all extents, is also included. You may from now on freely use **square bracket**-indexing, **pointers**, **references**, all **operators**, and the **<cstring>** and **<string>** libraries as you deem proper.

Description:
For this project you will create List classes. There will have to be two separate List-based implementations, one Array-based, and the other Node-based.

Array-based List:
The following header file excerpt is used to explain the required specifications for the class (the actual header ArrayList.h file is provided and accompanies the Project description):

```
...
class ArrayList{
  friend std::ostream & operator<<(std::ostream & os,                    //(i)
                                   const ArrayList & arrayList);
  public:
    ArrayList();                                                         //(1)
    ArrayList(size_t count, const DataType & value);                     //(2)
    ArrayList(const ArrayList & other);                                  //(3)
    ~ArrayList();                                                        //(4)

    ArrayList & operator= (const ArrayList & rhs);                       //(5)


    DataType * front();                                                  //(6)
    DataType * back();                                                   //(7)

    DataType * find(const DataType & target,                             //(8)
                    DataType * & previous,
                    const DataType * after = nullptr);

    DataType * insertAfter(DataType * target,                            //(9)
                           const DataType & value);
    DataType * insertBefore(DataType * target,                          //(10)
                            const DataType & value);
    DataType * erase(DataType * target);                                 //(11)


    DataType & operator[] (size_t position);                            //(12a)
    const DataType & operator[] (size_t position) const;                //(12b)

    size_t size() const;                                                //(13)
    bool empty() const;                                                 //(14)
    void clear();                                                       //(15)
```

```
  private:
    void resize(size_t count);                                    //(16)

    DataType * m_array;
    size_t m_size;
    size_t m_maxsize;
};
…
```

The **ArrayList** Class will contain the following **private** data members:

- **m_array,** a DataType class type Pointer, pointing to the Dynamically Allocated Array data. It is the container for the ArrayList data, and will have to be resized (reallocated) whenever it needs to grow to accommodate more data than it can fit, and possibly whenever it should trim down when it takes up too much space.
- **m_size**, a `size_t`, keeps track of how many DataType elements are currently stored & considered valid inside m_array. *Note*: this has to be properly initialized and updated each time the dynamically allocated memory is changed.
- **m_maxsize**, a `size_t`, denoting how many DataType type objects can fit in total in the currently allocated memory of the m_array. *Note*: this has to be properly initialized and updated each time the dynamically allocated memory is changed, and that generally m_size ≤ m_maxsize.

, will have the following **private** helper methods**:**

(16) resize – will deallocate the dynamic memory pointed to by m_array and then allocate enough total memory to fit the `size_t` count number of elements. Also, the original m_array data should be carried (copied) over to the newly allocated one.
Note A: When enlarging the m_array container, only the valid ArrayList elements (m_size in total) should be copied over, and the rest (m_maxsize-m_size in total) should have the DataType Default ctor value.
Note B: When shrinking down the m_array container, in case the new m_maxsize cannot fit all the m_size elements of the ArrayList, then the last ones are just discarded. If it can, then it copies over only the valid ArrayList elements (m_size in total), and the rest (m_maxsize-m_size n total) should have the DataType Default ctor value.

, and will have the following **public** member functions:

- **(1) Default Constructor** – will instantiate a new list object with no valid data. *Note*: What needs to be initialized in this case?
- **(2) Parametrized Constructor** – will instantiate a new list object, which will hold size_t count number of elements in total, all of them initialized to have the same value as the DataType value parameter. *Note*: Has to properly handle allocation.
- **(3) Copy Constructor** – will instantiate a new ArrayList object which will be a separate copy of the data of the other ArrayList object which is getting copied. *Note*: Remember Deep and Shallow object copies.
- **(4) Destructor** – will destroy the instance of the ArrayList object. *Note*: Any allocated memory pointed-to by m_array has to be deallocated in here.
- **(5) operator=** will assign a new value to the calling ArrayList object, which will be an exact copy of the rhs ArrayList object. Returns a reference to the calling object to be used for cascading operator= as per standard practice. *Note*: Think what needs to happen before allocating new memory for the new data to be held by the calling object.
- **(6) front** returns a pointer to the first (valid) element of m_array, or nullptr if it fails. *Note*: A reason for failing can be that the list is empty.

- **(7) back** returns a pointer to the last (valid) element of m_array, or nullptr if it fails. *Note*: A reason for failing can be that the list is empty.
- **(8) find** returns a pointer to the first (valid) element of m_array, which is found to have the same value as the passed parameter DataType target (the equality operator== as overloaded in class DataType should be used to check that). If it fails (it does not find the value it searched for), it returns a nullptr. Also, it takes in By-Reference a DataType Pointer parameter, and sets it to the Address of the target's predecessor element. If the search fails, or if the target element is found to be the first and has no predecessor, previous should be set to nullptr.
  The method also takes in a DataType Pointer named after, which indicates that the search inside the list should start from after that pointed element. If this is passed as nullptr, it denotes to start searching from the first element of the list. Otherwise, this parameter can be used to resume a 2nd search in case an element exists more times than one (otherwise Find will always return the first element's address).
- **(9) insertAfter** receives a Pointer to DataType, assumed to point to a valid element of the list (or be a nullptr). It inserts after it a new element with the value DataType value. Returns DataType Pointer to the element it just inserted (or nullptr if it failed). *Note*: Try to think through what you are doing, and sketch out how it's going to work. Think of all possible cases, e.g. inserting in the middle, at the end, in the start, what happens if m_array already has a size that fits the element, or if it should be resized to fit the new element, etc.
- **(10) insertBefore** receives a Pointer to DataType, assumed to point to a valid element of the list (or be a nullptr). It inserts before it a new element with the value DataType value. Returns DataType Pointer to the element it inserted (or nullptr if it failed). *Note*: Try to think even more thoroughly what you are doing. This is an Array-based container. Is it feasible to insert an element directly when all you are given in the method is the address of the Node before which you wish to insert? Or do you need to perform some extra step?
- **(11) erase** receives a Pointer to DataType, assumed to point to a valid element of the list (or be a nullptr) which we wish to erase. Returns a DataType Pointer to the element right after the one it just removed (if the last it removed was the last in the list, it should return nullptr). *Note*: Again, think thoroughly about the case of an Array-based container and any extra steps required. Also, think of all possible cases, e.g. removing in the middle, the first element, the last element.
- **(12) operator[]** will allow by-reference accessing of a specific DataType object at index int position within the allocated m_array. *Note*: Should not care if the position requested is more than the m_array size.
- **(13) size** will return the size of the current list. *Note*: This is the m_size of m_array and not its m_maxsize, i.e. it is the number of valid DataType entries inside it.
- **(14) empty** will return a bool, true if the list is empty, and false otherwise.
- **(15) clear** will clear the contents of the list, so after its call it will be an empty list object. *Note*: Does this need to perform memory deallocation?

as well as a friend function:

- **(i) operator<<** will output (to terminal or file depending on the type of ostream& os object passed as a parameter to it) the content of the calling ArrayList object. *Note*: it will do so by traversing the list and calling the insertion operator<< on the valid DataType elements contained within it.

Node-based List:
The following header file excerpt is used to explain the required specifications for the class (the actual header NodeList.h file is provided and accompanies the Project description):
…

```
class NodeList{
  friend std::ostream & operator<<(std::ostream & os,                    //(i)
                                   const NodeList & nodeList);
  public:
    NodeList();                                                          //(1)
    NodeList(size_t count, const DataType & value);                      //(2)
    NodeList(const NodeList & other);                                    //(3)
    ~NodeList();                                                         //(4)

    NodeList & operator= (const NodeList & rhs);                         //(5)

    Node * front();                                                      //(6)
    Node * back();                                                       //(7)

    Node* find(const DataType & target,                                  //(8)
               Node * & previous,
               const Node * after = nullptr);

    Node * insertAfter(Node * target,                                    //(9)
                       const DataType & value);
    Node * insertBefore(Node * target,                                  //(10)
                        const DataType & value);
    Node * erase(Node * target);                                        //(11)


    DataType & operator[] (size_t position);                           //(12a)
    const DataType & operator[] (size_t position) const;               //(12b)

    size_t size() const;                                               //(13)
    bool empty() const;                                                //(14)
    void clear();                                                      //(15)

  private:
    Node * m_head;
};
…
```

The **NodeList** Class will contain the following **private** data members:

- **m_head,** a Node class type Pointer, pointing to the Dynamically Allocated Node object considered as the first element of the list. *Note*: If the list is empty m_head should be nullptr.

,and will have the following **public** member functions:

- **(1) Default Constructor** – will instantiate a new list object with no data (no Nodes). *Note*: What needs to be initialized in this case?
- **(2) Parametrized Constructor** – will instantiate a new list object, which will hold `size_t` count number of elements (Nodes) in total, all of them initialized to hold the same value as the DataType value parameter. *Note*: Has to properly handle allocation.
- **(3) Copy Constructor** – will instantiate a new list object which will be a separate copy of the data of the other NodeList object which is getting copied. *Note*: Remember Deep and Shallow object copies.

- **(4) Destructor** – will destroy the instance of the NodeList object. *Note*: Any allocated memory taken up by elements (Nodes) belonging to the list has to be deallocated in here.
- **(5) operator=** will assign a new value to the calling NodeList object, which will be an exact copy of the rhs NodeList object. Returns a reference to the calling object to be used for cascading operator= as per standard practice. *Note*: Think what needs to happen before allocating new memory for the new data to be held by the calling object.
- **(6) front** returns a Pointer to the first element (Node), or nullptr if the list is empty.
- **(7) back** returns a Pointer to the last element (Node), or nullptr if the list is empty.
- **(8) find** returns a pointer to the first element (Node) of the list, that holds the same value as passed parameter DataType target (the equality operator== as overloaded in class DataType should be used to check that). If it fails (it does not find the value it searched for inside a Node), it returns a nullptr. Also, it takes in By-Reference a Node Pointer parameter named previous, and sets it to the Address of the target Node's predecessor element (also a Node). If the search fails, or if the target element is found within the first Node of the list and has no predecessor, previous should be set to nullptr.
  The method also takes in a Node Pointer named after, which indicates that the search inside the list should start from after that pointed element. If this is passed as nullptr, it denotes to start searching from the first element (Node) of the list. Otherwise, this parameter can be used to resume a 2nd search in case an element exists more times than one (otherwise Find will always return the first element's address).
- **(9) insertAfter** receives a Pointer to Node, assumed to point to a valid element of the list (or be a nullptr). It inserts after it a new element (a Node) that holds the value DataType value. Returns a Node Pointer to the element (a Node) it inserted (or nullptr if it failed). *Note*: Try to think through what you are doing, and sketch out how it's going to work. Think of all possible cases, e.g. inserting in the middle, at the end, in the start, etc.
- **(10) insertBefore** receives a Pointer to Node, assumed to point to a valid element of the list (or be a nullptr). It inserts before it a new element (a Node) that holds the value DataType value. Returns a Node Pointer to the element (a Node) it inserted (or nullptr if it failed). *Note*: Try to think even more thoroughly what you are doing. This is a singly-linked Node-based container. Is it feasible to insert an element directly when all you are given in the method is the address of the Node before which you wish to insert? Or do you need to perform some extra step?
- **(11) erase** receives a Pointer to Node, assumed to point to a valid element of the list (or be a nullptr) which we wish to erase. Returns a Node Pointer to the element (a Node) right after the one it just removed (if the last it removed was the last Node in the list, it should return nullptr). *Note*: Again, think thoroughly about the case of a singly-linked Node-based container and any extra steps required. Also, think of all possible cases, e.g. removing in the middle, the first element, the last element.
- **(12a,12b) operator[]** (const and non-const qualified) will allow by-Reference accessing of a specific DataType object within a Node at an index size_t position within the list. *Note*: Since this is not an Array-based implementation, the size_t position index is a "fake index", just an incremental value such that position=0 corresponds to the first element (a Node) in the list and each subsequent element corresponds to ++position.
- **(15) size** will return the size of the current list. *Note*: Since this is not an Array-based implementation, the function has to traverse the list to find how many elements (Nodes) are contained within it.
- **(16) empty** will return a bool, true if the list is empty, and false otherwise.
- **(17) clear** will clear the contents of the list, so after its call it will be an empty list object. *Note*: Does this need to perform memory deallocation?

as well as a friend function:

- **(i) operator<<** will output (to terminal or file depending on the type of ostream& os object passed as a parameter to it) the content of the calling NodeList object. *Note*: it will do so by traversing the list and calling the insertion operator<< on the valid DataType elements contained within the list's elements (Nodes).

The DataType.h and DataType.cpp files are provided fully implemented. Also, the ArrayList.h and NodeList.h header files are provided, and NodeList.h provides a class Node implementation in it as well. You will create the necessary ArrayList.cpp and NodeList.cpp source files to implement the range of required functionalities You should also create a source file proj8.cpp which will be a test driver for your classes.

Do not forget to initialize pointers and/or set them to nullptr appropriately where needed.
Do not forget to perform allocation, deallocation, deallocation-&-reallocation of dynamic memory when needed!
Memory accessing without proper allocation will cause Segmentation Faults. Forgetting to deallocate memory will cause Memory Leaks!

Use debugging tools, GDB and Valgrind, as you were instructed in your Lab Sections to detect the origin of such errors and memory leaks!

The completed project should have the following properties:

- Your code is required to follow the **file organization structure** demonstrated in your Labs, with subfolders for headers, source files, and a final build products location (generated during build).
- Your project's build should be based on a **CMakeLists.txt** script, which will be included with your deliverables.
- It must compile successfully using the g++ compiler on department machines or the provided Xubuntu VM image.
- The code must be commented and indented properly.
  Header comments are required on all files and recommended for the rest of the program.
  Descriptions of functions commented properly.
- A one page (minimum) typed sheet documenting your code. This should include the overall purpose of the program, your design, problems (if any), and any changes you would make given more time.

Turn in: Compressed file structure (with .cpp and .h files, and your CMakeLists.txt). Also, your project documentation file.

IMPORTANT: Creating a build configuration for Debugging:

- When requiring a build with debug symbols, usually you would specify this using the g++ command with the appropriate flag:
  `g++ -g ...`

  But **CMake** provides a convenient functionality for configuring a "standardized" debug build of your project with the required compiler flags and settings automatically handled by CMake. It does so via a configuration option called: `CMAKE_BUILD_TYPE`.
  If you want to enforce configuring your project build to have debug symbols enabled (because you intend to debug it using `gdb` for instance), you may run the cmake configuration command and specify this option as follows:

  ```
  cmake -D CMAKE_BUILD_TYPE=Debug ..
  ```

  Extra: Other possible values are:
  `-D CMAKE_BUILD_TYPE=Release` which enables recommended compiler optimizations to refactor the compiled code to make it more efficient,
  `-D CMAKE_BUILD_TYPE=RelWithDebInfo` which generates a "Release" (optimized) build but retains debug symbols for debugging (be careful with "Release" builds your code is optimized by the compiler and thus refactored).
- It is recommended to use such a configuration together with `gdb` (as demonstrated in your Labs) to trace any dynamic memory management bugs of your code.

The following are a list of restrictions:

- Your code may use the C++11 standard (or any standard higher or lower).

  Note: Usually, you would specify using the g++ command with some flags:
  `g++ -std=c++11 ...`

  But **CMake** provides the functionality of *autodetecting* your system's C++ compiler and generating the Makefiles to invoke the appropriate commands to be used when you eventually `make` your project.
  If you want to enforce configuring your project build to use a particular standard, you either do so everytime you run the cmake configuration command by:

  ```
  cmake -D CMAKE_CXX_STARDARD=11 ..
  ```

  Or you could put a line like the following inside your **CMakeLists.txt** script:

  ```
  set(CMAKE_CXX_STANDARD 11)
  ```

  You do not need to worry about either of these however, and for now just running the usual `cmake ..` for configuration should do the trick.
- No libraries except **<iostream>** and **<fstream>** and **<string>** or **<cstring>** / **<string.h>** allowed.
- No global variables except `const` ones.
- You are expected to employ code abstraction and reuse by implementing and using functions. The already provided code structure in the project description will be considered sufficient.
- You are expected to implement **const** correctness in your program design. This refers to class method qualifications, function parameter qualifications, etc.

Submission Instructions:

- You will submit your work via WebCampus
- Compress your:
  1. Code file structure (containing Source code files, Header files, CMakeLists.txt)
  2. Documentation

  Do not include executable or library files, nor any build, devel, or other non-required folders.
- Name the compressed folder:
  PA#_Lastname_Firstname.zip
  ([PA] stands for [ProjectAssignment], [#] is the Project number)
  Ex: PA8_Smith_John.zip

Verify: After you upload your .zip file, re-download it from WebCampus. Extract it, compile it and verify that it compiles and runs on the ECC systems.

- Code that does not compile will be heavily penalized –may even cost you your *entire* grade–. Executables that do not work 100% will receive partial grade points.
- It is better to hand in code that compiles and performs partial functionality, rather than broken code. You may use your Documentation file to mention what you could not get to work exactly as you wanted in the given timeframe of the Project.

Late Submission:
A project submission is "late" if any of the submitted files are time-stamped after the due date and time. Projects will be accepted up to 24 hours late, with 20% penalty.

Instructions to remotely test your project configuration and build on the ECC systems:

a) Download your Webcampus submission on your local computer. Let's say you submitted a file named **PAx_Smith_John.zip**, and you now downloaded it into your **Downloads** folder.

b) Navigate to that directory using your terminal, and check the file you downloaded is there.
   a. On **Ubuntu** you can open a terminal with Ctrl+Alt+T and then do:
      ```
      cd Downloads
      ls -al
      ```
   b. On a **Mac** you can open Spotlight and type "Terminal" and hit Enter, then do:
      ```
      cd Downloads
      ls -al
      ```
   c. On **Windows** in your Start Menu type "cmd" and click on Command Prompt, then:
      ```
      cd Downloads
      dir
      ```

c) Remotely copy your submission file from you local Downloads folder to your CSE Ubuntu user account inside its home/$USER folder.
   ```
   scp FILENAME NETID@ubuntu.cse.unr.edu:/nfs/home/NETID/FILENAME
   ```
   For example if the user NetID is **jsmith** and the submission file **PA7_Smith_John.zip**:
```
scp PAx_Smith_John.zip jsmith@ubuntu.cse.unr.edu:/nfs/home/jsmith/PAx_Smith_John.zip
```

d) Login to your CSE Ubuntu user account.
   ```
   ssh NETID@ubuntu.cse.unr.edu
   ```
   For example if the user NetID is **jsmith**:
   ```
   ssh jsmith@ubuntu.cse.unr.edu
   ```

e) Once you are in, check the contents to verify that you have successfully transferred the file:
   ```
   ls -al
   ```

f) Unzip the file into a folder with the same name:

a. If it is a **.zip** file then:
   unzip -o FILENAME.zip -d FILENAME
   Example:
   unzip -o PAx_Smith_John.zip -d PAx_Smith_John
b. If it is a **tar.gz** file then:
   mkdir FILENAME
   tar -xzvf FILENAME.tar.gz -C FILENAME
   Example:
   mkdir PAx_Smith_John
   tar -xzvf PAx_Smith_John.tar.gz -C PAx_Smith_John

g) The above will create a folder with the same name as your submission file, which will contain the unzipped content. Enter the directory and execute the known configuration and build sequence:

```
cd PAx_Smith_John
mkdir build
cmake ..
make
```