1 Project 2

Due: Mar 14 by 11:59p

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in your letter grade for the entire course being reduced by one level.

This document first provides the aims of this project. It then lists the requirements as explicitly as possible. This is followed by a log which should help you understand the requirements. Finally, it provides some hints as to how those requirements can be met.

1.1 Aims

The aims of this project are as follows:

- To give you some exposure to C++ smart pointers.
- To implement a doubly-linked list.
- To give you further exposure to the C++ standard library.

1.2 Background

A problem with the `List` ADT's from the textbook is that the current position `curr` in the list is stored within the list object. In fact, this is identified as a problem in an *alternate version* of the textbook. The issue is that storing the position within the list makes it difficult to have multiple traversals over the list extant at one time.

This project requires you to build a sequence ADT where insertions and removals are only allowed at the beginning and end of the sequence. Additionally, the ADT provides operations for obtaining an iterator positioned at the start or end of the list. The iterator ADT is a **constant** iterator in that it only allows reading the sequence contents and does not allow any modifications to the underlying elements.

So the sequence ADT supports `unshift()` and `shift()` operations for operating at the start of the sequence, `push()` and `pop()` for operating at the end of the sequence (these names are also used by many popular scripting languages like Ruby, Perl and JavaScript). It also supports `clear()` and `size()` operations. Finally, it supports a `cbegin()` and `cend()` operations to obtain the iterators (these names are also used by the STL, the `c` stands for **const**).

The iterator ADT uses operator overloading to make the iterator looks like a pointer with operator `*` returning the current element indexed by the iterator

and `->` returning a pointer to the current element. It supports the pre-increment `++` and pre-decrement `--` operations (it does not support the post-increment and post-decrement operations). It also supports an operator which converts the iterator to a boolean, return true iff the iterator is indexing an element in the underlying sequence.

You are being provided with an implementation of the sequence ADT which uses a fixed size array. You are required to provide an implementation which uses a doubly-linked list. Additionally, you are required to implement a driver program which requires concurrent forward and backward iteration over a sequence.

1.3 Requirements

You must push a `submit/prj2-sol` directory to your `master` branch in your github repository. This directory must contain an implementation `DLinkSeq` of the abstract class `Seq` described in seq.hh which uses a doubly-linked list. Typing `make` within that directory should build a `nums` executable with usage:

```
./nums [-a] INTS_FILE_PATH
```

If the `-a` option, the program should use an `ArraySeq` implementation of the `Seq` abstract class; otherwise it should used a `DLinkSeq` implementation.

The program should read whitespace delimited integers $i_0, i_1, \ldots, i_{n-2}, i_{n-1}$, from `INTS_FILE_PATH`, entering them into the specified `Seq` container. When end-of-file is encountered, the program should use concurrent forward and backward iterators from the `Seq` interface to output the accumulated integers, one per line:

$$i_0, i_{n-1}, i_1, i_{n-2}, i_2, i_{n-3} \ldots, i_{n-3}, i_2, i_{n-2}, i_1, i_{n-1}, i_0$$

i.e. the output should be the first integer, the last integer, the second integer, the second-last integer, ..., the last integer, the first integer.

Note that n can be any non-negative integer. It may be the case that it is greater than the capacity of the chosen sequence container in which case the program can terminate ungracefully.

Non-functional requirements:

- The program should check all command-line arguments and terminate gracefully if they are in error.
- If API assumptions are violated, the program may terminate ungracefully, possibly leaking memory.
- The program should not leak memory on normal termination.

1.4 Sample Log

An edited sample log of the operation of the program is shown below:

```
$ ./nums
usage: ./nums [-a] INTS_FILE_PATH
$ ./nums -x xx
usage: ./nums [-a] INTS_FILE_PATH
$ ./nums  xx
cannot read xx: No such file or directory
$ cat extras/test.data
1
2
3
4
5
6
$ ./nums extras/test.data
1
6
2
5
3
4
4
3
5
2
6
1
$ ./nums -a extras/test.data
1
6
...
#same output as above
...
6
1
$ ./nums extras/overflow-array.data
1
9
2
8
3
7
```

```
4
6
5
5
6
4
7
3
8
2
9
1

#overflows since the ArraySeq has a max capacity of 8.
$ ./nums -a extras/overflow-array.data
nums: arrayseq.hh:54: ...: Assertion ... failed.
Aborted (core dumped)
$
```

1.5 Provided Files

You are being provided with the following files. The files in the prj2-sol directory **must** be submitted along with your project, whether you modify them or not; the files in the extras directory need not be submitted.

seq.hh The `Seq` abstract class you need to implement using a doubly-linked list. You **must not** modify this file.

arrayseq.hh An implementation of the `Seq` abstract class using a fixed-size array. You should not need to modify this file.

README A README file which must be submitted along with your project. It contains an initial header which you must complete (replace the dummy entries with your name, B-number and email address at which you would like to receive project-related email). After the header you may include any content which you would like read during the grading of your project.

test.data and overflow-array.data Test files for the `nums` program. The second file overflows the array used when the `-a` option is specified and the `ArraySeq` is created using the default size.

seq-test.cc, command.hh and command.cc A driver program which can be used for testing both the provided `ArraySeq` and the `DLinkSeq` you are required to write. This program must be linked with with provider classes `ArraySeq` and `DLinkSeq`.

An annotated edited log of using `seq-test` is shown below:

```
#output usage message
$ ./seq-test
usage: ./seq-test ArraySeq|DLinkSeq

# run program again using ArraySeq implementation of Seq
$ ./seq-test ArraySeq
[REPEAT] COMMAND [VALUE]
COMMAND is one of clear|unshift|shift|push|pop|help
>> push 3
[ 3 ]

# use repeat count of 3 to unshift 3 5's
>> 3 unshift 5
[ 5 5 5 3 ]
>> pop
3
[ 5 5 5 ]
>> shift
5
[ 5 5 ]
>> 2 pop
5
5
[ ]

#pop from an empty Seq fails with an error
>> pop
seq-test: arrayseq ...Assertion ... "pop on empty array seq"' failed.
Aborted (core dumped)


# run program again using DLinkSeq implementation of Seq
$ ./seq-test DLinkSeq
[REPEAT] COMMAND [VALUE]
COMMAND is one of clear|unshift|shift|push|pop|help
>> 4 push 22
[ 22 22 22 22 ]
>> pop
22
[ 22 22 22 ]
>> 2 unshift 6
[ 6 6 22 22 22 ]
>> 3 shift
6
```

```
6
22
[ 22 22 ]
>> 2 pop
22
22
[ ]

#shift from an empty Seq fails with an error
>> shift
seq-test: dlinkseq ... Assertion ... "shift() on empty dlist"' failed.
Aborted (core dumped)
$
```

1.6 Hints

The following steps are not prescriptive in that you may choose to ignore them as long as you meet all project requirements.

1. Study the code you have been provided:
   (a) Look at the implementation of the << operator in seq.hh which shows how the iterators are used.
   (b) Understand the code in arrayseq.hh thoroughly as it will give you some idea of how trivially an iterator can be implemented.
   (c) Finally, look at the code in seq-test.cc sufficiently to understand how a reference to the sequence is extracted from the smart pointer returned by `make()`.
2. Make sure your `cs240` repository is up-to-date:
   ```
   $ cd ~/cs240
   $ git pull
   ```
3. Start a new `prj2-sol` branch in your `i240?` project:
   ```
   $ cd ~/i240?
   $ git checkout master
   $ git pull #should not get any updates
   $ git checkout -b prj2-sol
   $
   ```
4. Recursively copy over the files you have been provided with:
   ```
   # assumes your i240? directory is in your prj2-sol branch
   $ cd ~/i240?/submit
   ```

```
$ cp -pr ~/cs240/projects/prj2/prj2-sol .
$ cd prj2-sol


# assumes prj2-sol branch and ~/i240?/submit/prj2-sol dir
$ cp -p ~/cs240/projects/prj2/extras/* .
$
```

5. Commit and push your files to github:

```
# assumes prj2-sol branch and ~/i240?/submit/prj2-sol dir
$ git add .
$ git commit -a -m 'started prj2'
$ git push -u origin prj2-sol  #push changes,
                               #creating remote branch
```

6. Create a `Makefile` set up to build a `seq-test` executable from the files you currently have in your `prj2-sol` directory. Note that you should comment out references to `DLinkSeq` and `dlinkseq.hh` in `seq-test`. Make sure you provide a `clean` target which minimally cleans out all executables and object files.

7. Use your `Makefile` to compile the `seq-test` executable. You should be able to run it using the `ArraySeq` implementation of `Seq`.

8. Start working on your `nums` executable:

   (a) Create a `nums.cc` with a do-nothing `main()`.

   (b) Modify your `Makefile` to build a `nums` executable, linked in with the `ArraySeq` code. Verify that your `Makefile` works.

   (c) Set up validation for the command line arguments for `main()`; simply check the number of arguments and verify that if there is an option specified then it is `-a`.

   (d) Set up a `using` declaration for the type `TestType` of objects you will be storing in your sequences: some flavor of integer.

   (e) Write a function to read integers from the file specified on the command-line. You can do so by simply using `in >> i` where `in` is an `istream` on the file and `i` is a suitably declared integer variable. For now, simply write out the data on to the terminal.

   (f) Create a `Seq<TestType>` using the `ArraySeq`. Note that `ArraySeq::make()` returns a `unique_ptr` to an array sequence (this has been typedef'd to a `SeqPtr`). You can get the raw pointer to a `Seq` by using a `get()` on the value returned by the `make()`. Write code to

perform the extraction of the raw pointer and verify that it does indeed compile.

(g) Now pass the raw sequence pointer over to the function reading the data from the file. Instead of having the function print out the data, have it squirrel away the data into the sequence.

(h) Now write a function to output the data from the sequence. You should use `cbegin()` and `cend()` to obtain iterators positioned at the start and end of the sequence. Once again, those functions return a `unique_ptr ConstIterPtr` which wrap the underlying the iterator pointers. Get hold of the `ConstIter` iterator pointers by simply dereferencing the smart pointers using a `*` dereference operator.

Your code to output the data should now be straightforward: as long as both iterators are not exhausted, output the underlying data and increment/decrement each iterator suitably.

(i) Clean up your code in `nums.cc`. If you have not already done so hook up the `-a` option to use the `ArraySeq`. Make any other changes you feel are necessary. Use valgrind to verify that it does not leak any memory.

You now have a handle on how to use iterators and smart pointers.

9. Start working on your `DLinkSeq` implementation. You can use the code from the textbook or discussed in class as a starting point as well as using the code provided in `arrayseq.hh` for aspects specific to this project. Obviously, you will not be maintaining the current position in the list so you will not need a field like `curr`. Don't bother attempting to manage your own free-list, simply use `new` and `delete` for each list node. Have your class inherit from `Seq` and use two separate `head` and `tail` dummy header nodes set up to point to each other like the code in the textbook.

(a) Set up your class with sufficient dummy code inserted for the operations in the `Seq` interface so that you can compile the code. For example, you can set up your `unshift()` and `push()` as NOPs, `shift()` and `unshift()` to return some fixed dummy value and have your iterator always exhausted (i.e. the **bool** operator always returns **false**). Modify your `nums.cc` to hook in your `DLinkSeq` when `-a` is not specified. Uncomment the lines in `seq-test.cc` you had commented out earlier to hook in your code. Set up your `Makefile` to build both executables.

(b) Start work on the `Seq unshift()` and `push()` operations, using the textbook `insert()` and `append()` code as starting points. Note that the code for `unshift()` and `push()` will be duals in that one can get the code for one from the code for the other by doing things like interchanging variables like `head` and `tail` or `next` and `prev`.

While writing the code above, you should be compiling periodically to get basic sanity checks on your code.

Once you have completed your code, Run the executable, and verify that adding the data to your dlink sequence does not cause any exceptions. If you do get exceptions, debug using gdb. Even if you do not get any exceptions, you may want to use gdb to verify that the data is being entered into the sequence.

(c) Now implement your iterator. It should basically consist of a pointer to the list nodes. You will initialize this pointer to point to the first or last element of the list depending on whether or not the iterator is created by `cbegin()` or `cend()` (if you have set up your linked list correctly, this will do the right thing automatically when the list is empty). The iterator is exhausted if the pointer is pointing to one of the header nodes. The increment/decrement operations should simply involve stepping the pointer within the linked list and returning `*this`.

(d) Once you have implemented your iterator, you should be able to run your `nums` successfully using your `DLinkSeq` implementation. You should also be able to run the `seq-test` executable but without support for the `shift` and `pop` commands.

(e) Now write the code for you `Seq` `shift()` and `pop()` operations using the textbook's code for `remove()` as starting points. Once again, you should find that the code for the two operations are duals of each other.

10. Use valgrind to verify that your program does not leak memory.

11. Iterate until you meet all requirements.

12. Clean up for submission. It is up to you whether or not you submit the files you copied from the `extras` directory, but ensure that typing `make` in your `submit/prj2-sol` directory builds a working `nums` executable without any errors or warnings.

It is a good idea to commit and push your project periodically whenever you have made significant changes. When it is complete please follow the procedure covered in *Lab 0* to merge your `prj2-sol` branch into your `master` branch and submit your project to the TA via github.