CS2201
Spring 2023
Assignment No. 2

Purpose: Gain experience in using and manipulating dynamic arrays, and the Law of the Big 3.

Background on DNA, Restriction Enzymes, and PCR:

This assignment builds upon ou nting a DNA strand by using an array of characters. In this lab, we will enhance our representation ction of having a fixed sized (static) array by replacing it with a dynamic array. This will allow desired and we will add capabilities to grow when necessary. We will also enhance our represent tional capabilities. In particular, we will add a form of splicing that will allow us to insert sequence is meant to more accurately represent what restriction enzymes do. You might want to refer to the l ecification of project #1 for additional information on restriction enzymes and PCR.

The Assignment:

Just as in the prior assignment, we are going to represent a DNA strand by using a partially-filled array of characters, only this time the array will be dynamically allocated. This will allow us to represent very large DNA strands and also allow us to increase the size of the strands during execution if necessary. We are also enhancing the class by adding additional capabilities.

Each DNA_Strand object will now contain three data fields:

- `maxDNA` – the size of the dynamically allocated array
- `mySize` – the size of the DNA strand being stored in the array (this is the same as in project #1)
- `myDNA` – a pointer to the dynamically allocated array

The functional specification below will lead you through the steps necessary to convert your DNA_Strand class from using a static array to using a dynamic array. After that we will add additional functionality to the class, now that we have the ability to grow the size of the array at runtime.

The DNA_Strand.h file, which describes all the functions to be implemented, is available with this project specification.

Functional Specifications:
You will be supplied the class declaration file: **DNA_Strand.h.** The file contains the declaration of a set of functions to manipulate arrays representing DNA.

Here are the new methods that you need to implement (there are some changes to existing methods too – those will be noted later):

```
// Alternate constructor.
// Dynamically create an empty DNA_Strand of a given array size.
DNA_Strand (size_t size);   // alt ctor

// The copy constructor.
DNA_Strand (const DNA_Strand & s);

// Destructor
// Clean up the DNA_Strand (e.g., delete dynamically allocated memory).
~DNA_Strand ();

// Assignment operator performs an assignment by making a copy of
// the contents of parameter <rhs>
const DNA_Strand& operator= (const DNA_Strand & rhs);
```

```
// Returns the size of the array, which is also the max size of a strand we can represent
size_t maxSize () const;

// countEnzyme -- overloaded
// string parameter version
// Counts non-overlapping instances of the target Enzyme
// Eg, the eznyme "AAA" appears 3 non-overlapping times in the DNA "AAAAAAAAAAA"
size_t countEnzyme (const std::string & target) const;


// grow
// This method will allo                    the size of the dynamically allocated
// array by allocating                      e desired size, copying the data from
// the old array to the                     hen releasing the old array.
// If the newSize is le                     to the current size, then no actions
// are taken.
void grow (size_t newSize);

// append (accepting a string parameter)
// Append the characters of the parameter to the end of the current DNA,
// growing the array if necessary.
// Example: if myDNA contained ACTTGA and "ACCTG" was received as a parameter,
// then afterward myDNA will contain ACTTGAACCTG
void append (const std::string & rhs);

// append (accepting a DNA_Strand parameter)
// Append the characters of the parameter to the end of the current DNA,
// growing the array if necessary.
// Example: if myDNA contained ACTTGA and ACCTG was received as a parameter,
// then afterward myDNA will contain ACTTGAACCTG
void append (const DNA_Strand & rhs);

// splice (accepts 2 Strings representing sequences)
// finds first pair of targets in current DNA strand and replaces
// the sequence between the end of the first target through the end of the
// second with the insertSequence, growing the array if necessary.
// If two instances of the target are not found, then no changes are made.
// See project spec for note on efficiency.
void splice (const std::string & target, const std::string & insertSequence);

// splice
// instead of starting from the beginning of the strand, this version
// starts from a given index, and returns the position *after* the splice,
// returns -1 if no changes are made.
int splice (size_t pos, const std::string & target, const std::string & insertSequence);
```


To get started on this project, you are provided with a project2.zip file. This project2.zip file contains a CLion project that has a DNA_Strand.cpp file whose method bodies contain temporary junk code that you need to replace (just like project #1). The project also contains a DNA_Strand.h file that has several changes from project #1. The project as provided compiles and runs, though it does not produce the correct results since all the method bodies contain temporary junk code. The following instructions lead you through the process to create a correctly working class. These instructions will incrementally add functionality to the project, a bit at a time. Making the changes in this fashion will allow you to compile and test your intermediate steps – a key to success with this and any large, complex project.

Process:

1. Review the grading of your project #1 submission and fix your code correspondingly. If your project #1 submission has not been graded yet, you should continue with the following instructions but be sure to incorporate fixes to your old code before submitting your project #2 solution.
2. Unzip the provided project2.zip, and open the resulting project in the CLion IDE. Please see the section titled “Opening a CLion project” in the project #0 spec and ensure that your project #2 CMakeLists.txt file has all the

correct compiler flags. This is important to ensure your code compiles when it is graded. Also make sure that your project settings/preferences specify the use of the clang compiler under the CMake options (as per the CLion/clang installation instructions posted to Brightspace).

3. Once the project is opened, wait for all progress bars to complete before proceeding. The code as provided will compile and run, though the test code will report errors since only the class methods only hold temporary junk code. Replace the provided DNAtest.cpp with the DNAtest.cpp from your project #1. The code should continue to compile and run, though all the tests should continue to fail since the necessary functionality is still missing.
4. Open up DNA_Strand.h and DNA_Strand.cpp and review the provided code. Note the major changes in DNA_Strand.h; which are:
    a. The constant MAX[illegible]ed with maxDNA, a private instance variable of type size_t. This instance variable w[illegible]k of the size of the dynamically allocated array for each object.
    b. The constant DEF[illegible] been defined. It is type "const size_t" and has the value 50.
    c. The myDNA array [illegible] rather than static array. I.e., it is declared as a `char*`.
    d. Note that mySize s[illegible]nue to hold the size of the DNA strand in the myDNA array.
    e. There are a few mc[illegible]clared. We will address those later.
5. Open the DNA_Strand.cpp file provided with this project and note that all the method bodies contain temporary junk code. Next open the DNA_Strand.cpp file from your project #1. One by one, copy-n-paste the **bodies** of the methods from your project #1 file over to the methods in the project #2 file (do not overwrite the new method header comments). Skip over any of the methods that are new for project #2. Take your time and make sure you copy all the methods from your project #1 file. As you do this copying, ignore for now the red squiggly lines or messages that deal with MAX_DNA.
6. After you have copied the method bodies, do a global find-replace in DNA_Strand.cpp changing all instances of MAX_ DNA to maxDNA. Again, maxDNA is an instance variable that will keep track of the size of the dynamically allocated array.
7. Change the default constructor to initialize an empty DNA_Strand with a dynamically-allocated array of size DEFAULT_DNA_SIZE (don't forget to correctly initialize maxDNA and mySize too). The constructor should use the base member initialization list when possible.
8. Change the alternate ctor that takes a string as a parameter to allocate an array of the needed size (the size of the parameter string), and initialize the strand appropriately. The ctor should use the base member initialization list as much as possible. This ctor will allow us to represent DNA strands of any size.

At this point, you have a class that should behave *almost* identical to your project #1 class, except that this class uses dynamic arrays rather than static arrays. Compile and run all your tests in DNAtest.cpp (that you copied from project #1). Note: some students have stated that their code would not compile without defining a destructor, but I have not found that to be the case – if your compiler requires you to add the destructor then please add it. When you run your code the only tests that may fail would be those that depend upon the existence of a copy ctor or an assignment operator, or a test depending upon the initialization of a DNA strand to be limited to 50 characters. You should alter your test program to properly test the new alternate constructor since it no longer limits DNA strands to 50 characters.

If all your testing is successful (besides the noted exceptions), you are ready to start adding new functionality to your project. At this point, it would be good to review the grading feedback that you received on project #1 and address all the issues identified. You will likely not have the feedback yet when you start this project, but you should come back and complete this step as soon as the feedback is available.

If your testing is unsuccessful, make sure you address all the problems before you proceed or you may end up wasting a lot of your time (you may want to add your Big-3 first before you spend much time debugging). As you add the following functionality to your class, compile and test each method as you go. If you want, you can add private helper methods to your class to support the work that you need to do (though that is not required, and I did not find it necessary).

9. Add the other alternate ctor that takes an initial array size but still creates an empty DNA. The ctor should use the base member initialization list as much as possible.
10. Add the destructor to the class.
11. Add the copy ctor to the class. The ctor should use the base initialization list as much as possible.
12. Add the assignment operator to the class.
13. Add the `maxSize()` method. This method reports the size of the dynamically allocated array, rather than the size of the DNA strand.

14. Add the overloaded `countEnzyme()` method that takes a string parameter rather than a char parameter. This method will count ***nonoverlapping*** occurrences of the target string.
15. Add the `grow()` method. This method can be used to allocate a larger dynamic array. Data must be copied from the original array to the new array one element at a time. Be sure to free the old array.
16. Add the two `append()` methods. Some careful planning will allow you to define one in terms of the other.
17. Add the `splice()` methods. Again, define one in terms of the other. **Note:** to receive full credit on these functions, they must be "*efficient*" in that they do not move data elements of the array twice. It is tempting to write these functions by performing a cleave and then inserting the new data – but that means data is shifted down in the array during the cleave and then shifted back up during the insert [such a solution works but will only receive partial credit].
18. Okay, if you have not done so already, start working on fixing of your project #1 submission and incorporate fixes accordingly into your project #2 code.
19. As a final step, make sure that the DNA_Strand.h that is being compiled in your project is the same as the DNA_Strand.h file distributed with this assignment. The only differences would be private helper methods that you added (again, none are needed, and no changes to any implementation).

Other details:
Here are a few notes that might be helpful:
1. You are free to add helper methods to the private section of your class.
2. To emphasize the point one more time: you are not allowed to treat your `myDNA` array as a cstring (a null-terminated, character array). Rather than marking the end of the DNA strand with a null terminator, we are using the `mySize` instance variable to keep track of how many characters the array is holding.
3. As with project #1, you are not allowed to convert your DNA_Strand to a string object so that you can search or edit the string. Rather you are to implement the methods of this assignment by operating directly on the char array.
4. You are expected to do your work on the dynamic myDNA array directly whenever possible. You should not create auxiliary arrays and copy data in & out of them. You should only create new arrays when it is required.

Final write-up: When you have completed the assignment, create a **README** document (a .txt text file or a .doc Word document), and in it answer the following questions (be sure to name the file simply **README**):
1. State your name and email address.
2. After reviewing this spec and the .h file, please estimate/report how many hours you think it will take you to complete this project. [This is just an estimate and does not affect your grade.]
3. How many hours did you actually spend total on this assignment? [This information will not affect your grade.]
4. Did you encounter any impediments that prevented you from making progress on this assignment?
5. What did you like or hate about this assignment?
6. Do you have any suggestions for improving this assignment?

Submission for grading:
When you have completed your work on this assignment, please submit the three source files for grading: **DNA_Strand.cpp, DNA_Strand.h,** *and* your test driver (likely named **DNAtest.cpp**), plus your final **README** write-up document. Submit ONLY the four files – please do not submit a zip file containing your entire project. You can submit the files by visiting the assignment page in Brightspace (click on the assignment name), scroll down to the "Submit Files" section, and add the files by clicking on the "Add a file" button and finding the files to attach.

After submitting your homework, it is good practice to verify your submission. Revisit the assignment page in Brightspace and make sure that your files were successfully submitted. Then click on each file to open it up so that you can verify that you submitted the correct file, as opposed to some older version of the file. It is your responsibility to insure the correctness of your submission.

If you need to resubmit any of the files, you need to do a full resubmission of ***all*** the files, as only your last submission is saved.

Grading:
This project is worth 50 points. Your grade on this project will be based on the following:
1. The correctness of your methods implemented in the DNA_Strand.cpp file.

2. The use of good programming style. No lines of code past column 100, proper indentation, proper use of curly braces, etc. See the style document posted to Brightspace under Course Resources.
3. Eliminating redundancy in your code. If you fail to utilize other methods that you have written and have repeated functionality in multiple methods, you will lose points.
4. The thoroughness of your testing performed in the DNATest.cpp file. Note: the thoroughness of your testing often has a great impact on the correctness of your code (see item #1 above).
5. Appropriate responses to all questions in the README file.

You should also review the syll alties for late programming assignments.

Please post questions to Piazza s document or you feel something is missing or unnecessarily confusing.

Acknowledgements:
This assignment is loosely based o achan at Duke University which is in the collection of ACM SIGCSE Nifty Assignments.