CSC 2430 – Data Structures 1
Homework #6 – ADT "DynString" Object
Due: Wednesday, March 2, 2016

Design and implement an ADT "**DynString**" that represents a variable-length, dynamically allocated character-string value. DynString is also designed to be able to hold the contents of an entire text file (e.g., a single long string that may contain numerous text lines, each delimited by an embedded '\n' newline character).

In many ways, DynString is similar to the SimpleString and DynArrayQ classes from the class lecture notes. The primary differences are:

- Fixed Size Buffer vs. Dynamically sized and allocated buffer
  - SimpleString utilizes a fixed-size array buffer of characters in the private data area.
  - DynString contains a dynamic array pointer (pBuff) to a Cstring character string array buffer whose value must be dynamically allocated and deleted in the various DynString methods, based on the actual string data value the an object represents. As always with CStrings, remember that the actual size of the allocated pBuff buffer for a particular string data value always requires one extra position to hold the '\0' sentinel at the end of a string value.
    E.g., "Hello" has a curLength = 5, and a pBuff dynamic array allocation of 6 bytes.
- In terms of dynamic memory allocation and management, DynString is fairly similar to the DynArrayQ example, with Copy Constructors, Destructors, and an operator= assignment operator overload. However, unlike DynArrayQ, your DynString objects must always have an allocated buffer, even for a zero-length string (that is, even the string value "" which has curLength = 0 must have a buffer of size 1 allocated, to hold the string's sentinel '\0'; pBuff must never be NULL).

For this lab, create a project that utilizes the filenames "DynString.h" and "DynString.cpp" for the ADT class (because the following main test program utilizes these specific filenames for DynString).

- Attached to this lab is a copy of DynString.h, that specifies all of the constructor and method prototypes.
- Also attached is a stub of DynString.cpp named "DynStringStub.cpp". It contains a draft version of the implementation code for the DynString class methods. You must rename the stub file "DynString.cpp" and then modify it to complete the implementation each of the specified methods.

As you are developing your implementation, you should implement a small main.cpp test program to perform the "unit tests" of the various methods. This code will create various DynString instances and invoke the various methods to verify that proper functionality has been achieved.

Once you are confident that your DynString implementation is complete, then delete your testing main.cpp module from the project and instead add the "DynStringMain.cpp" main application program to the project (copy is attached to lab assignment). This program creates a DynString variable, reads in an entire file as it's character-string value, and then produces an output report that prints out all lines in the string-file that contain the specific substrings "DynString" and "pBuff". DynStringMain.cpp uses your own version of the **DynString.cpp** implementation file as the test data (store a copy of your DynString.cpp file in the solution Debug subdirectory so it can be read in and analyzed).

To summarize, for this lab you must create a project with 3 source-code files (plus a temporary main.cpp for preliminary unit testing purposes):

- the **DynString.h header declaration file** (as supplied with this lab assignment)
- the **DynStringMain.cpp main program** that uses and tests out your class implementation (supplied with lab).
- **DynString.cpp** - your **class implementation file** (stubbed version supplied with this lab assignment)

Turn in:

1. **DynString.cpp** - your source code listing for the implementation of the class methods.
2. **an example execution run** of the DynStringMain.cpp program that demonstrates the use of DynString objects. This example execution uses a copy of your own "DynString.cpp" file to read and analyze.

Printout listings for the **DynString.h** header and the **DynStringMain.cpp** files are **not** required.
You should not modify them for this lab without checking with your instructor first.

```
// DynStringMain.cpp   CSC2430 Homework Dynamic Strings      Mike Tindall

#include <iostream>
using namespace std;

#include "DynString.h"

// Print report of all lines in s that contain instances of substring look
void findAll(const DynString& s, const DynString& look)
{
    cout << "Find lines containing " << look.toString() << ":" << endl;

    char ch = look.getChar(0);       // starting character of look

    // Look through s character by character to find substrings that match
    int pos = 0;
    do {
        pos = s.findChar(ch, pos);   // Find next possible position
        if(pos >= 0)
        {
            // See if substring of s matches the look string
            if(look.compare(s.substr(pos, look.length())) == 0)
            {
                // Output the entire line from s that contains look

                // First, determine the linenumber that corresponds to pos
                int lnum = s.findLineNumber(pos);

                // Now, retrieve the lnum line as a substring from s
                DynString ds = s.findLine(lnum);

                // Output a line in the report
                cout.width(4);
                cout << lnum << ": " << ds.toString() << endl;
            //  cout << lnum << ": " << ds << endl;  // using << overload
            }

            ++pos; // Move on to the next position
        }
    } while(pos >= 0);

    cout << endl;
}

int main()
{
    DynString s;

    if( !s.readFile("DynString.cpp") )
      cout << "Cannot open file: DynString.cpp" << endl;

    findAll(s, DynString("DynString"));
    findAll(s, DynString("pBuff"));

    return(0);
}
```

```
// DynString.h         CSC 2430 Homework
// A variable-sized dynamic array implementation of a string ADT
#include <iostream>

class DynString {
public:
     // Construct a new DynString, initialized to substring of s
     //   starting at position 0, of length len
     // If len < 0, utilize entire string s
     // If len > length of s, then utilize entire string s
     // Both s and len have default parameter values
     DynString(const char s[]="", const int len=-1);

     // Copy Constructor.  Construct new (deep) copy of s
     DynString(const DynString &s);
     // Assignment operator= override.  Make new (deep) copy of s.
     DynString& operator=(const DynString &s);
     // Destructor.  Delete allocated string-value
     ~DynString();

     // Return current string-value length
     const int length() const;

     // Return ASCIIZ char* string pointer
     const char* toString() const;

     // Read next input line into string
     // Assumes that incoming input line has at most 1000 characters
     // Returns true if a line is successfully input.
     // Returns false and sets the string value to "" if the end-of-file is reached.
     bool readLine(std::istream& in = std::cin);

     // Read entire input file into string as single, long string-value
     //   with embedded \n newline characters marking the ends-of-lines.
     // Returns true if the filename is successfully opened and read.
     // Returns false and sets the string value to "" if a file error occurs.
     bool readFile(const char filename[]);

     // Concatenate s to end of string-value
     void concat(const DynString &s);

     // Compare string-value to s.
     // Return (-), 0, or (+) depending on result of comparison
     const int compare(const DynString &s) const;

     // Retrieve string[position].  Return 0 if out of range
     const char getChar(const int position) const;

     // Find position of ch in string.  Return -1 if not found
     // Begin searching with [startoffset] position
     const int findChar(const char ch, const int startoffset=0) const;

     // Return new DynString with value that is substring of original
     DynString substr(const int start, const int len=-1) const;

     // Return substring of original corresponding to linenum (1 .. n)
     // Do not include '\n' at end of line
     DynString findLine(const int linenum) const;

     // Return line number 1 .. n corresponding to position in string
     int findLineNumber(const int position) const;

private:
     char *pBuff;               // The actual string buffer array
     int   curLength;           // current length of string-value

     // Added extra -- not required for lab assignment. Implementation provided.
     // Overload << operator for DynStrings
     // Note: friend function, not a class method
     friend std::ostream &operator<<(std::ostream& out, const DynString &s);
};
```

Notes (read and consider carefully in your implementation):

- You may use any of the standard library "str…( )" functions that are useful (such as strcpy_s, strcat_s, strcmp, etc.).
- char *pBuff   is the pointer to the actual string value. It is always allocated with code similar to:
  ```
        pBuff =  new  char [ actualstringlength + 1 ];
  ```
  leaving room for the '\0' character at the end of the ASCIIZ string value. Note that pBuff should never be set to NULL. Even the "" 0-length string requires a 1-byte allocation, to hold the '\0' sentinel.
- When a DynString is created and constructed, pBuff is allocated as shown above, and curLength is set to the string length. However, several of the other methods (e.g., copy-constructor, assignment operator=, readLine, readFile, concat) cause the string-value to be changed. To avoid memory leaks, in any method in which an existing string value is being replaced, it is important to delete the original pBuff value, before allocating the new array value:
  ```
        delete [ ] pBuff ;
        pBuff =  new char [ actualstringlength + 1 ];
  ```
- Note that pBuff is always pointing to some string value, and is never NULL.
- As with all classes in which memory is allocated dynamically, it is important to include a destructor method (to deallocate the data), and both a copy constructor and an overloaded assignment operator= method to make a "deep" copy of an existing value.
- Note that the readLine(istream& in) method takes an I/O stream variable (such as cin) as a by-reference parameter. The readLine method implementation can just use "in >> value" or "in.getline(…)" to read from the stream.
- If the readFile(filename) method is invoked, it must open the specified filename and read in the contents of the file, storing the data as a single, long string-value. If the incoming file contains several distinct 'lines' (that is, with '\n' newline characters), then the long string-value must also contain the newline characters. One strategy for accomplishing this is to simply read in each incoming line using the in.getline(. . .) mechanism, and then concatenate each line to the end of the current DynString. It will also be necessary to explicitly concatenate the "\n" string after each line.
- The string-value of a DynString object may contain several '\n' delimited text lines.
  You can easily search for the '\n' characters using the findchar(. . .) method.
- The findLineNumber(pos) method converts a string-position 'pos' into a corresponding line-number.
  Start at the beginning of the string-value as line #1, and then scan through the string-value incrementing the line-number until you get to the 'pos' position.
- The findLine(linenumber) method must determine the starting position and length of the specified line, and extract out a DynString substring that contains just a copy of that line.
  Start at the beginning of the string-value as line #1, and then scan through the string-value incrementing the line-number until you reach the specified line. From this position in the string-value that corresponds to the start of the specified line, continue scanning to locate the end of the line '\n' position. Subtracting the two positions gives the length of the line, and a substr(…) operation can extract the line.

- Note that the constructor for DynString is a very powerful operation, which is able to construct a new DynString object of a specified length from an arbitrary array of characters. This is equivalent to constructing a new DynString object as a substring of an existing string. A working implementation of this method is included in the stub file.
  For example,
  ```
       char  buff[50] = "Hi There";
       DynString tmp  = DynString( buff, 2);            // tmp   = "Hi"
       DynString tmp2 = DynString( &buff [3], 3 );      // tmp2 = "The"
  ```

- The friend function **operator<<**(ostream& out, const DynString &s) overloads the << operator for any ostream. Its implementation can use the normal "out << s.toString();" insertion operator and manipulators to write to the stream. It is not a class 'method', because the second parameter specifies a DynString s reference. A 'friend' function is allowed to have access to the internal private parts of the class and implementation, even though it is not an actual method.