COMSC-165 Lab 15
Advanced Linked Structures

This lab introduces binary tree data structures, by using them in an artificial intelligence program that learns from its mistakes. Binary trees enable you to organize data records in a way that lets you sort through them with a series of binary choices, leading from a *root node* through the tree until an endpoint is reached. You will write a Win32 console application program called **Animal.cpp**, in which the computer invites the user to think of an animal, and then proceeds to guess that animal by asking a series of yes/no questions.

The program starts by inviting the user to think of an animal and to indicate when he is ready to proceed. Immediately, the computer guesses that the animal is an elephant. Presumably this guess is incorrect, so the user tells the computer what animal he was thinking of. The computer learns from its mistake by asking the user to teach it a yes/no question that it can use in the future, to tell the difference between the incorrectly guessed animal, and the correct one. It "remembers" this, and in its enthusiasm invites the user to start another **cycle** by thinking of another animal, and thus the process continues. Download the the ZIP containing Windows 32-, 64-bit, and Mac versions, for an example of how the program is to work. (For the Mac version, before running in Terminal, for security reasons, you may have to run this command: `chmod a+x AnimalMac`.)

After a while, the computer "learns" enough so that it can pretty much figure out any animal that any user might think of. This type of artificial intelligence program is called an *expert system*, in which an expert "trains" the computer and the results are used by others as a diagnostic tool.

The program does *not* include *persistence*, so it unfortunately "forgets" what it learns every time it is run.

As you complete this assignment, post the required file(s) to the COMSC server. For assignments that involve more than one file, you may post them all at once, or one-at-a-time as you complete them, as you prefer. For assignments that involve more than one separate "lab" (like lab 1a, 1b, and 1c), the individual labs are graded in order, starting with "a", which must be fully correct before "b" is graded, and so on. This assignment is worth 50-points, to be awarded after all labs are successfully completed. Use the "Submit your FA2014 work" link on the class website to post your file for this lab, using the **lab15** folder provided.

LAB 15: An Artificial Intelligence Program [`Animal.cpp`]
This lab does not contain step-by-step instructions for rewriting the **Animal** program -- instead it presents a specification, which you are to follow in any way you wish. Read the entire specification before doing any coding or planning. Once you fully understand the specification, begin by planning the different parts of the program that you know you will need. Start coding *after* you have the program as planned out as you can get it.

SPECIFICATION

1. Use this struct definition for the binary tree nodes:

```
struct animal
{
  char text[100]; // a yes/no question OR an animal name, 99 characters with room for a null
  animal* yes; // pointer to next node if question is answered YES
  animal* no; // pointer to next node if question is answered NO
}; // if "yes" and "no" are 0, then "text" is an animal name
```

Animal objects are used to store either a yes/no question or an animal name. All the nodes at the extremities of the binary tree contain animal names, and their `yes`/`no` pointers are 0, because they are endpoints of different lines of questioning. The tree "grows" when an endpoint is reached, and the animal name in that node is *incorrect*. In this case, the node "splits" -- you add two new nodes, and connect them to the tree with the `yes`/`no` pointers of the node containing the incorrect animal name. Now, you replace the incorrect animal name in the `text` string buffer with a question, and put animal names in the two new nodes.

2. Fill-in this pseudo code outline to create your **cpp** file:

```
// program and programmer identification
// includes
// struct definition
// function prototype
void deallocateTree(animal*); // write this!
int main()
{
  // initialize a "root" pointer named "animal* root"
  // set "root" to a newly created node; set its text to "elephant", pointers to 0
  // start a "while" loop that runs each cycle of the program
    // invite the user to think of an animal which it will try to guess
    // await the user's response, and break out of the loop if he declines
    // declare a pointer "p" to traverse the tree, and initialize it to "root"
    // start a loop to traverse the binary tree
      // if p->yes is 0...
        //...print p->text as the guessed animal
        // ask user if this is correct
        // if correct, brag and break from loop
        // ask user what animal he was thinking of...
        //...store in "char a[100]"
        // ask what yes/no question differentiates "p->text" from "a"...
        //...store in "char q[100]"
        // ask which response is correct for "a" -- yes or no...
        //...store in "char yesNo"
        // create two new nodes, names "y" and "n"
        // if the correct response for "a" is yes...
          // copy "a" into y->text
          // copy p->text into n->text
        // else if the correct response is no...
           // copy "a" into n->text
           // copy p->text into y->text
        // copy "q" into p->text
        // set y->yes, n->yes, y->no, and n->no to 0
        // set p->yes to y and p->no to n
        // break from loop
      // else if p->yes is not 0
        // print p->text as a question
        // ask for a yes/no reply...
        //...store in "char yesNo"
        // if "yes", set p to p->yes
        // else set p to p->no
  // reclaim memory
  deallocateTree(root);
}
```

3. Write the missing `deallocateTree` function, to deallocate the memory for each node in the tree (using the `delete` keyword). Refer to your lecture notes in order to write this recursive function.

4. Run through at least 5 cycles to test the code and to train the program.

Be sure to initialize the root node's `yes` and `no` pointers, as well as its animal "guess". Be sure to use `strcpy()` to set string values. Be sure to use `cin.ignore();` *between* each use of a `cin >>` that is followed by a `cin.getline`. Be sure to include matching open and close braces for your loops and `if` statements.

Be sure to properly indent, space, and align your code. Code readability *will be judged* as part of the scoring of this exercise. Copy your code to NotePad, to make sure there are no tabs to misalign your code when it is viewed outside of your code editor.

*Note: **Animal.cpp** will be modified and improved in lab 16.*

*Post **Animal.cpp** to the COMSC server.*

GRADING POLICY
Lab work will be returned for you to fix and resubmit for any of the following reasons, or for not following stated specifications, unless directed otherwise by your instructor, Point penalties will be assessed per the syllabus for each time a lab is returned for redo for any listed reason below or specification above. Points will be awarded for an assignment upon final, successful completion of all of its labs. Note that you will NOT receive a list of corrections to make, because to do so would remove any incentive for students to do quality work. Instead, grading of your work will stop upon discovery of the first mistake. Grading will not continue until the mistake is corrected. So to avoid multiple cycles of correction and resubmission, and the cumulative point penalties, make sure that your work is as correct as you can make it before submitting it.

Basic checklist, even if your compiler "forgives" doing otherwise:

1. Spell and case filenames *exactly* as specified. To avoid errors, be sure to configure your Windows computer to NOT hide file name extensions -- refer to the Burns Intro to Programming text, chapter 2.
2. Submit files to the specified folder. Do NOT submit archive files, like ZIP, JAR, or RAR.
3. Include identifying information as the first lines in the file AND as the first lines of your output in main.
4. Do NOT use the statement `system("PAUSE");` or any other system-specific code.
5. Do NOT try to access the value stored in an uninitialized variable.
6. CPP and H files must be free of careless typing mistakes that would prevent compilation.
7. Put an `endl` as the last item in the last-executed `cout` statement before the closing curly brace of `main`.
8. Do NOT forget to use the `cin.ignore` statements for non-string console input, unless specifically directed otherwise.
9. Do NOT forget to use the `fin.ignore` statements for non-string file input, unless specifically directed otherwise.
10. For file I/O, NEVER include a path designation. ALWAYS use the "working folder" for files.
11. Use the proper `#includes` and `usings` as listed in the C++ Library PDF.
12. Do NOT #include C-libraries such as `math.h` -- include their C++ equivalents instead, like `cmath`.
13. Do NOT #include libraries that you do not use.
14. Do NOT #include "**stdafx.h**"; do NOT use `pragma`.
15. Do NOT use C/C++ constructs or library items that are not specifically taught in this class, such as `goto`, `scanf` or `fscanf`, `fixed`, `strcmp_c`, `stricmp`, etc.
16. Do NOT use `void main()` -- use `int main()` always.
17. Do NOT use `return 0;` as the last statement in `int main()`, because it is not required by ANSI Standard C++.
18. If you use output statements in your program for debugging purposes, be sure to delete them before submitting your work.
19. Do *not* use "global variables".

Advanced checklist, even if your compiler "forgives" doing otherwise:

1. `NULL` is ONLY a pointer value -- do NOT use it for integer or other numeric values, even if your compiler allows it.
2. Do NOT declare a static array with a variable size specification, such as `int n =...; int a[n];`, even if your compiler allows it.
3. NEVER #include a CPP file, even if the textbook includes examples of such.
4. For console input and text file input with `>>`, use `.ignore` as explained in the Burns Intro to Programming text, chapter 5, unless specifically directed otherwise.
5. NEVER use `cin >>` to read directly into a numeric variable, like `int` or `double` or `float`. Read as a string (C or C++) and convert to a number using `atoi` or `atof`. Include `cstdlib`!
6. Do NOT use the `inline` keyword. To make class member functions inline, write their definitions inside the class definition.
7. Once the `const` keyword is introduced in this course, use it where applicable. Note that class "getter" functions are not really getter functions unless they have a trailing `const` keyword to flag them as such.
8. Make sure that all value-returning functions return a valid value under any logical circumstance.
9. Exit functions with `return` statements, and NOT `exit` -- do NOT use `exit` under any circumstances.

10. In every H and CPP file, ALWAYS #include libraries and H files for ALL functions, classes, and identifiers used in that file. Exception: use a class "forward declaration" instead of #including the class' H file when only pointers and reference variables appear.
11. Include the `cstring` library for char-array-based string ("C string") *functions*, and the `string` library for STL-based-strings ("C++ strings").
12. When using the random number generator seed function `srand`, call it ONCE ONLY in your program, and "sacrifice" the first value, like this: `srand(time(0)); rand();` at the top of main. Be sure to #include both `ctime` and `cstdlib`.
13. In COMSC 210, use `using namespace std;`. In 165 and 200, use `using std::` statements, properly placed under the #includes of the libraries to which they pertain.
14. `std::string` and `std::getline` belong to the `string` library. In 165 and 200, use `using std::` statements for these. Note that `cin.getline` is a different thing, and only requires `using std::cin;`.
15. Do NOT use `using std::` for anything in the C libraries. Use it for items from C++ libraries, only.
16. Test drivers MUST at least include ifndef testing, object copy testing with assignment after declaration, and const object testing with assignment upon declaration.
17. Test drivers must NOT include console or file input, unless directed otherwise.
18. Do NOT allow memory leaks, unless specifically instructed otherwise.