CSCI 104 - Fall 2015 Data Structures and Object Oriented Design

- Due: Wednesday, September 23, 11:59pm
- Directory name for this homework (case sensitive): `hw3`
  - This directory should be in your `hw_username` repository
  - This directory needs its own `README.md` file
  - You should provide a `Makefile` to compile and run the code for your tests/programs in problems 2, 3, 4, and 5. See instructions in each problem for specific rules.

Skeleton Code

Some skeleton code has been provided for you in the `hw3` folder and has been pushed to the Github repository `homework-resources`. If you already have this repository locally cloned, just perform a `git pull`. Otherwise you'll need to clone it.

```
$ git clone git@github.com:usc-csci104-fall2015/homework-resources
```

Problem 1 (Review Material)

Carefully review Array Lists, Queues, Stacks, Amortized Runtime, Operator Overloading, Copy Constructors, and C++ STL

Problem 2 (Array Lists, Copy Constructors, and Operator Overloading, 35%)

Write an integer array-based list class, `AListInt` in two files alistint.h (provided for you) and `alistint.cpp` (which you will create and implement yourself). Notice it has roughly the same list interface (public member functions) as your linked list implementation. However, it should also contain a

- Copy constructor
- Assignment operator
- A private member function `resize()` that will increase the capacity of the list by **doubling** its size and be called if the list becomes full and the user requests another insertion.
- Operator+ to produce the concatenation of two lists
- Operator[] to access an element at a given location (much like you can do with a normal C++ array)

When you have completed `alistint.h` and `alistint.cpp`, you should write a Google Test-based unit test program named `alisttest.cpp` (using your knowledge of unit-testing) that ensures each of the member functions work and also use `valgrind` to check that there are no memory leaks. Add a rule `alisttest` to your `Makefile` to compile all the needed code and your test program. We should be able to compile your program by simply typing `make alisttest`. Be sure you test the case where the list is resized.

Problem 3 (Stacks, 10%)

Use your ArrayList implementation from Problem 2 to create a Stack data structure for variables of type `int`. Download and use the provided stackint.h as is (do NOT change it). Notice the stack has an ArrayList as a data member. This is called **composition**, where we compose/build one class from another, already available class. Essentially the member functions of the `StackInt` class that you write should really just be wrappers around calls to the underlying linked list.

You should think **carefully** about efficiency. **All operations (other than possibly the destructor) should run in (amortized) O(1)** Failure to meet this requirement may result in the loss of half of the available points on this problem.

Problem 4 (Amortized Analysis, 15%)

In Question 3, you implemented a Stack using a dynamic array. Hence, under the hood of your stack implementation, whenever there was no more space left, you doubled the size of allocated memory and copied stuff over.

Part (a)

If, when there was no more space left, you had increased the size of allocated memory to allow for **10** more values to be pushed, what would be the amortized worst-case running-time for push?

Part (b)

If, when there was no more space left in an array of (current) size n, you had increased the size of allocated memory to allow for `sqrt(n)` more values to be pushed, what would be the amortized worst-case running-time for push? (This one is a bit trickier than the `sqrt(n)` question on HW2, because you take the square root of the current size, rather than of the "final" array size.)

Part (c)

With each pop, you did not reallocate the array. One might be worried that this wastes space. If we pop enough elements, maybe we should also allocate a smaller array, and copy stuff over.

Our first attempt may be the following rule: if the array is less than half full, then allocate an array of half the size and copy everything over, deleting the old array. What is the worst-case sequence of pops and pushes for your implementation? Was this a good idea? (In other words, what can you say about amortized time now?)

Part (d)

An alternative solution to part (c) is the following: if the array is less than a quarter full, then allocate an array of a quarter the size and copy everything over, deleting the old array. (When the array is full, you still **double** the array, though.) Analyze the amortized worst-case running time of push and pop in this implementation. Is this solution appreciably better?

Problem 5 (Boolean Expression Parser and Evaluator, 40%)

Boolean expressions consist of variables that can take on a value from the set {F,T} (or {0,1}), the operators AND (&), OR (|), and NOT (~), along with parentheses to specify a desired order of operations. Your task is to write a program that will read Boolean expressions from a file, and a set of variable assignments from a second file, and evaluate and show the output of the given Boolean expressions using the variable assignment.

Boolean Expressions are defined formally as follows:

1. Any non-negative integer is a Boolean expression, namely a variable. (Think of "3" as representing the variable "x3".)
2. The constants "T" and "F" are Boolean expressions, denoting "True" (1) and "False" (0), respectively.
3. If Y1, Y2, ..., Yk are Boolean expressions then the following are Boolean expressions:
    - ~Y1
    - (Y1|Y2|Y3|...|Yk)
    - (Y1&Y2&Y3&...&Yk)

Notice that our format rules out the expression 1&2, since it is missing the parentheses. It also rules out (1&2|3) which would have to instead be written ((1&2)|3), so you never have to worry about precedence. This should make your parsing task significantly easier. Whitespace may occur in arbitrary places in Boolean formulas. Each expression will be on a single line.

Examples (the first two are valid, the other three are not):

```
(14 &(2|3 ))
~~(2 & 7& ( ~5000000000 |~0))
((1&2)    // missing parenthesis
(1&2|3)   // mixing operators
(&1&2)    // extra &
```

The variable assignment will be stored in a separate file. The file format will simply be the variable number followed by an F (for false) or T (for true). The variable number and assignment will be separated by whitespace, with one assignment per line. All variables will be represented as non-negative numbers though not necessarily sequential {0 to n-1}.

Example:

```
1 T
2 F
7 F
3 T
0 F
50000000 T
14 T
```

Your program should take two parameters at the command line: the first parameter is the file in which the formulas are stored, the second parameter is the file in which the variable assignment is stored. Your program should take these two files and for each expression output to `cout`, one per line, one of the options:

- `Malformed` if the formula was malformed (did not meet our definition of a formula) and then continue to

the next expression.

- `Unknown Variable` if the formula references a variable that was not present in the assignment file (i.e. `(1 & 6)` where `6` was not defined in the assignment file) and then continue to the next expression.
- `T` if the expression was well-formed, contained only known variables, and evaluated to **true**
- `F` if the expression was well-formed, contained only known variables, and evaluated to **false**

Each expression will be on a single line by itself so that you can use getline() and then parse the whole line of text that is returned to you. If you read a blank line, just ignore it and go on to the next. The second file may contain a few extra unused variables. The variable names/numbers will always fit into `int` types, but as you can see from the example, they can be pretty large. You **MUST** also use an STL Map data structure to store your variable mappings from the 2nd file and to then access those values when you evaluate an expression.

While this may be contrary to your expectation of us, you must **not** use recursion to solve this problem. Instead keep a stack on which you push pieces of formula. **Use your `StackInt` class** from Problem 3 for this purpose. Push open parenthesis '(', variables, truth values {T,F}, and operators onto the stack. When you encounter a closing parenthesis ')', pop things from the stack and evaluate them until you pop the open parenthesis '('. Now --- assuming everything was correctly formatted --- compute the value of the expression in parentheses, and push it onto the stack as a truth value {T,F}. When you reach the end of the string, assuming that everything was correctly formatted (otherwise, report an error), your stack should contain exactly one value (F or T), which you can output.

In order to be able to push all those different things (parentheses, operators, variable numbers, truth values) onto the stack, you will need to represent each item with an integral value. It is your choice how to do this mapping. One option is to store special characters (parentheses, operators, truth values) as special numbers that you reserve specifically for these purposes. It might make your code more readable to define the mapping of special characters to integers by declaring them as const ints as in:

```
const int OPEN_PAREN = -1;
```

That way, your code can use `OPEN_PAREN` wherever you want to check for that value.

Chocolate Problem (Logic Puzzle Solver, 2 chocolate bars)

We will periodically have "chocolate problems" on homework assignments. These are significantly harder than regular problems. There are no "points" for them, that is, solving chocolate problems will not affect your grade in any way. The reward for chocolate problems is literally chocolate. For this problem, if you get it all correct, you will receive two chocolate bars (of your choice, within reason). Partially correct solutions may receive partial chocolate allocations. You are welcome to solve chocolate problems in teams, in which case you share the chocolate you receive. If you submit a chocolate problem, you should do so directly by e-mail to your instructor, attaching all relevant files. The TAs and graders will not evaluate your chocolate problem solutions.

As a first step for this homework's chocolate problem, expand your solution to the Boolean evaluator to also **find** assignments that make the formula true. Recursion and backtracking may be helpful here. Since there could be many assignments making a given formula true, you should implement a functionality whereby you can test if a particular variable must be true (or false) for **all** assignments that make the formula true.

Now for the real fun part. Write a parser that parses English descriptions of facts about the world, and generates Boolean formulas that are then fed into your Boolean solver to learn new facts about the world. We suggest a format as follows (but feel free to modify):

- First specify the types of facts you are interested in. For instance "human", or "student", or "smart", or "at USC". Internally, these will turn into Boolean variables. (Hint: a map may be a useful data structure here.)
- Now specify facts about the world, like the following examples:
    - Every student is human.
    - If x is at USC and a student, then x is smart.
    - Whenever y is not human, y is not smart.
    - I am a student.
- Your program should be able to parse some similar set of patterns about the world. You don't have to parse the full English language (that would be a lot), but you should understand a small collection of "reasonable" phrases and grammatical patterns.
- As a hint, steer clear of operators that involve more than one object. Things like "x likes y" are much harder to reason about than just "x is a human/student/...". The latter are just true/false variables for single entities. If you want to implement full functionality with more complex relations and functions, you'll turn this into a semester-long project.
- Your program should now output every fact that can be inferred about the world from the given facts. The functionality of the Boolean solver should come in handy here.
- Demonstrate how cool your program is by giving us some non-trivial puzzles about the world where your program infers something that would take a human at least 30 seconds to figure out.
- As always with chocolate problems, you have some leeway in file formats, what gets passed where, how to interact, and what exactly is the way to parse.