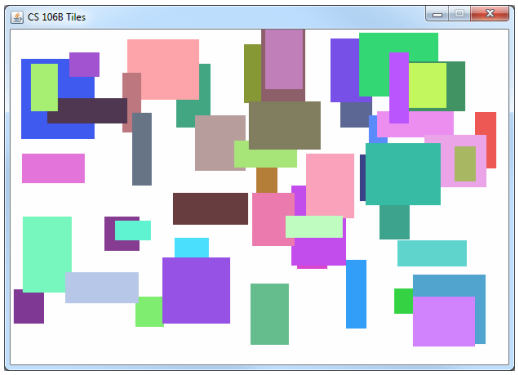
Stanford University, CS 106B, Autumn 2013
Homework Assignment 3: Tiles, Assassins

due Friday, October 25, 2013, 2:00pm

Thanks to Mike Clancy for the Tiles assignment idea and Stuart Reges for the Assassins assignment.

This two-part assignment practices implementing collections using a growable unfilled array and using a linked list.

Part A: Tiles



- **Files:** Turn in files **TileList.h** and **TileList.cpp**.
  Do not modify any of the other provided files.

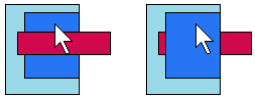
In Part A of this assignment you will write the logic for a graphical program that allows the user to click on rectangular tiles. You'll write a class that stores a list of tiles using an array as internal storage. The program is graphical, but you do not directly draw any graphics yourself; all of the graphical interface code has already been written for you, and your code just has to call it.

The main client program is provided for you as `tilemain.cpp`. When it runs, it will create a graphical window on the screen displaying a list of tiles. (Initially the panel shows 50 tiles, but you can add more by typing N.) Each tile is represented as a structure of the provided type `Tile`. Each tile's position, size, and color are randomly generated by the main program. The overall list of tiles should be stored and maintained by your code in the `TileList` class you will write.

In the screenshot to the right, notice that some tiles overlap and occupy some of the same (x, y) pixels in the window. In such a case, the tile created later is "on top" of prior tiles and may partially cover them on the screen. You should think of the overall list of tiles as having an ordering, where tiles stored earlier in the list are closer to the "bottom" and ones stored later in the list are closer to the "top". This is sometimes called a *z-ordering*.

The graphical user interface ("GUI") displays the tiles and allows the user to click on them. Depending on the kind of click, one of five different actions occurs:

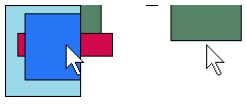
- If the user clicks the **left mouse button** while the mouse cursor points at a tile, that tile is moved to the very *top* of the z-ordering (the end of the tile list).



- If the user clicks the **left mouse button** while holding the **Shift or Alt key**, while the mouse cursor points at a tile, that tile is moved to the very *bottom* of the z-ordering (the start of the tile list).

- If the user clicks the **right mouse button** while the mouse cursor is pointing at a tile, that tile is removed from the tile list and disappears from the screen.

- If the user clicks the **right mouse button** while holding the **Shift or Alt key**, *all tiles that occupy that pixel* are removed from the tile list and disappear from the screen.



- If the user types the **N keyboard key**, a new randomly positioned tile is created and added to the screen.

If you use a computer with a 1-button mouse, you can simulate a right-click with a Ctrl-click (or a two-finger tap on the touch pad on some laptops). If the user clicks the window at a pixel not occupied by any tiles, nothing happens.

If the user clicks a pixel that is occupied by more than one tile, the top-most of these tiles is used. (Except if the user did a Shift-right-click, in which case it deletes all tiles touching that pixel, not just the top one.)

Note that **your code does not need to directly detect mouse clicks or key presses**. Our provided code detects the clicks / presses and responds by calling various methods on your `TileList` object, as described on the next page.

Part A Implementation Details:

Your `TileList` class stores a list of tiles internally as an **array of `Tile` structures**. It is your job to grow the array as needed if it becomes too full to fit a tile, and also to shift elements left/right as needed to reorder tiles. Write the following public members; stay within the Big-Oh runtime noted. **Assume that every parameter passed is valid.**

`TileList()`	In this constructor you initialize a new empty list of tiles with capacity 10.
`~TileList()`	In this destructor you should free all dynamically allocated memory used by your list.
`addTile(`***`tile`***`)`	In this method you should add the given tile to the end (top) of the tile list. O(1).
`drawAll(`***`window`***`)`	In this method you are passed a reference to a `GWindow` object, and you should draw all tiles in your list on that window in the proper order using each tile's `draw` function. Tiles earlier in the list should appear "below" tiles later in the list. O(*N*).
`indexOfTopTile(`***`x, y`***`)`	In this method you should return the 0-based index of the topmost (last) tile in the list that touches the given (x, y) position. If no tile touches it, you should return -1. O(*N*).
`raise(`***`x, y`***`)`	Called by the provided GUI when the user left-clicks the given x/y coordinates. If these coordinates touch any tiles, you should move the topmost (last) of these tiles to the very top (end) of the list, shifting as needed. O(*N*).
`lower(`***`x, y`***`)`	Called by the provided GUI when the user shift-left-clicks the given x/y coordinates. If these coordinates touch any tiles, you should move the topmost (last) of these to the very bottom (beginning) of the list, shifting as needed. O(*N*).
`remove(`***`x, y`***`)`	Called by the provided GUI when the user right-clicks. If these coordinates touch any tiles, delete the topmost (last) of them from the list, shifting as needed. O(*N*).
`removeAll(`***`x, y`***`)`	Called by the provided GUI when the user Shift-right-clicks. If these coordinates touch any tiles, you should delete *all* such tiles from the list, shifting as needed. O(N^2).

Each `Tile` structure in your list has the following public members. See the provided `Tile.h` for more details.

`x`, `y`, `width`, `height`	The top/left (x, y) coordinate of the tile and its width and height in pixels.
`color`	The color of the tile, specified as an RGB string such as `"#ff00ff"`.
`contains(`***`x, y`***`)`	Returns `true` if this tile touches the given (x, y) pixel position.
`draw(`***`window`***`)`	Draws the tile onto the given `GWindow`.
`toString()`	Returns a text representation of the tile, which can be useful for debugging.

We have provided you a partially complete version of your header file `TileList.h` that declares the preceding members. You will need to modify the file to complete it. In particular, you will need to do the following:

- **Add comments** to `TileList.h` and `TileList.cpp`. (Comment headers on public members go in the `.h` file. The implementations of those members in the `.cpp` file do not require any header comments.)
- **Declare any necessary private members** in `TileList.h`, such as private instance variables needed to implement behavior. Your inner data storage *must* be an array of `Tile`s; do not use other data structures.
- **Add `const`** as appropriate to any public members that do not modify the state of the list.
- **Implement the bodies** of all member functions and constructors in `TileList.cpp`. Several operations are similar. As appropriate, avoid redundancy in your code by creating additional "helper" functions in your `TileList` to help reduce redundancy. (Declare them `private`, so outside code cannot call them.)

Please do not make any changes to `TileList.h` that modify the public constructor or methods' names or parameter types. Our client code should be able to call the public methods on your tile list successfully without modification.

We suggest coding basic operations first, such as `addTile` and `drawAll` only. Then write `raise`, and then the rest. To figure out if a tile touches a given x/y pixel, call the`Tile`'s `contains` function. Delay worrying about resizing the array by changing the main program to use fewer rectangles (say, 5) at first. Do not use `GRect` on this program.

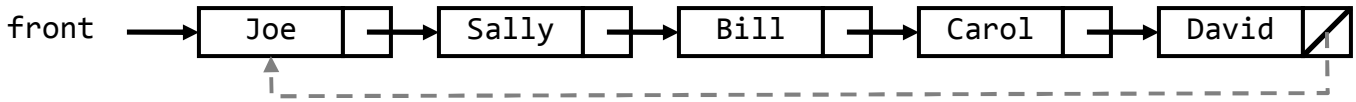
Part B: Assassins

- **Files:** Turn in **AssassinsList.h** and **AssassinsList.cpp**. Do not modify any of the other provided files.

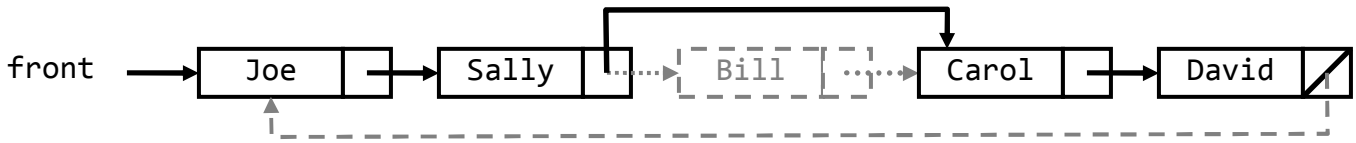
Part B of the program focuses on implementing linked lists using pointers and dynamic memory allocation.

"Assassins" is a real-life game played every year at Stanford. Each person playing has a particular target that he/she is trying to "assassinate." Assassinating a person often means finding them on campus in public and acting on them in some way, such as saying, "You're dead," or squirting them with a water gun, or tagging them. One of the things that makes the game fun to play in real life is that initially each person knows only who they are assassinating; they don't know who is trying to assassinate them, nor do they know whom the other people are trying to assassinate.

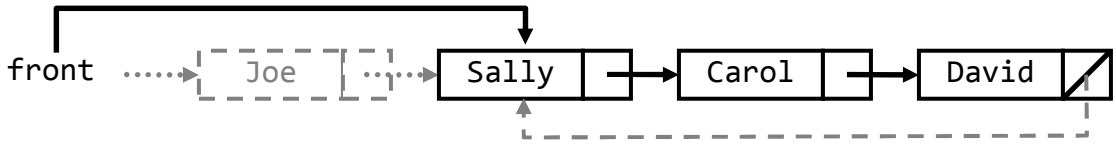
The game is played as follows: You start with a group of people; say we have five people playing named Joe, Sally, Bill, Carol, and David. A circular chain of assassination targets (called the "game ring" in this program) is established. For example, we might decide Joe should target Sally, Sally should target Bill, Bill should target Carol, Carol should target David, and David should target Joe. (In the actual linked list that implements this game ring, David's `next` pointer would be `NULL`. But conceptually we can think of it as though the next person after David is Joe, the front person in the list.) Here is a picture of this "game ring":



When someone is assassinated, the links need to be changed to "skip" that person. For example, suppose that Bill is eliminated by Sally. Now Sally's new target will be Bill's former target: Carol. The game ring becomes:



If the first person in the ring is eliminated, the list's front must adjust. If David eliminates Joe, the list becomes:



You will write a class `AssassinsList` that keeps track of who is targeting whom and who eliminated whom. Internally your class maintains two **linked lists**: a list of people who are currently alive (the "game ring") and a list of people who are now dead (the "graveyard"). As people are eliminated, you must move them from the game ring to the graveyard by rearranging links between nodes. The game ends when only one node remains in the game ring: the winner.

A client program called `assassinsmain` has been written for you. It reads a file of names and constructs an object of your list class. This main program then asks the user for the names of each victim to eliminate until there is just one player left alive (at which point the game is over and the last remaining player wins).

Your program's output should match the partial log of execution at right. See the course web site for complete example output files. Your program should exactly reproduce this output format and behavior.

```
Current game ring:
  Joe is targeting Sally
  Sally is targeting Bill
  Bill is targeting Carol
  Carol is targeting David
  David is targeting Joe
Current graveyard:

next victim? bill

Current game ring:
  Joe is targeting Sally
  Sally is targeting Carol
  Carol is targeting David
  David is targeting Joe
Current graveyard:
  Bill was eliminated by Sally

next victim? JOE

Current game ring:
  Sally is targeting Carol
  Carol is targeting David
  David is targeting Sally
Current graveyard:
  Joe was eliminated by David
  Bill was eliminated by Sally
```

Part B Implementation Details:

For this assignment we specifying exactly what fields you can have in your `AssassinsList` class. You must have *exactly* the following two fields; you are not allowed to have any others:

- a pointer to the front node of the game ring
- a pointer to the front node of the graveyard (`null` if empty)

```
struct Node {
   string name;   // my person's name
   string killer; // who eliminated me
   Node*  next;   // ptr to next node
   Node(string name, Node* next){...}
};
```

You must use our `Node` structure, found in `AssassinsList.h`, which has the implementation provided at right. Your list class must have the following public members. See the provided `AssassinsList.h` file for more details.

`AssassinsList(`***`file`***`)`	In this constructor you initialize a new list over the people in the given file. The names appear one per line. (Names could have spaces, so read the file a line at a time.) Build a game ring of linked nodes in the *opposite order* of how they appear in the file. If the file lines are `"John"`, `"Sally"`, and `"Fred"`, the game ring would store that Fred is targeting Sally who is targeting John who is targeting Fred (in that order). Assume that the file is readable, that the names are non-empty strings, and there are no duplicates. O(*N*). You should **throw** a string exception if the list file does not contain any names.
`~AssassinsList()`	In this destructor you should free all memory used by your list nodes.

`printGameRing()`	In this method you print the names of the people in the game ring, one per line, indented by two spaces, as shown. It should produce no output if the game is over. O(*N*). For the names on the first page, the initial output would be:
	`Joe is targeting Sally`
	`Sally is targeting Bill`
	`Bill is targeting Carol`
	`Carol is targeting David`
	`David is targeting Joe`

`printGraveyard()`	In this method you should print the names of the people in the graveyard, one per line, with each line indented by two spaces, with output of the form "***name*** was eliminated by ***name***". It should print the names in the opposite of the order in which they were eliminated (most recently eliminated first, then second most recently eliminated, and so on). It should produce no output if the graveyard is empty. O(*N*). For example, from the previous names, if Bill is eliminated, then David, then Carol, the output is:
	`Carol was eliminated by Sally`
	`David was eliminated by Carol`
	`Bill was eliminated by Sally`

`isAlive(`***`name`***`)`	You should return `true` if the given name is in the game ring and `false` otherwise. O(*N*). **Ignore case** in comparing names; `"salLY"` should match a node for `"Sally"`.
`isDead(`***`name`***`)`	You should return `true` if the given name is in the current graveyard and `false` otherwise. O(*N*). **Ignore case** in comparing names; for example, `"CaRoL"` should match a node for `"Carol"`. Note that `isAlive` and `isDead` are not exact opposites; a string that isn't a name of a player in the game is neither alive nor dead.
`isGameOver()`	You should return `true` if the game ring has just one person in it. O(1).
`winner()`	You should return the name of the winner of the game, or `""` if game is not over. O(1).
`eliminate(`***`name`***`)`	In this method you record the elimination of the person with the given name, moving them from the game ring to the front of the graveyard. Other than the removed person, this should not change the relative order of the game ring (i.e., the links of who targets whom). **Ignore case** in comparing names. A node remembers who eliminated the person in its `killer` field; it is your code's responsibility to set that field's value. O(*N*). **Throw** a string exception if the game is over or the name is not part of the game ring.

Every method's runtime should follow the Big-Oh restriction given, where *N* is the number of people in your lists. Don't forget to use the course web site's **Output Comparison Tool** to help check your output for various test cases.

As in Part A, we have provided you a partially complete version of your header file `AssassinsList.h` that declares the preceding members. You will need to modify the file to complete it. As in Part A, you will need to add comments, declare any private members needed, apply `const` as appropriate to members and parameters.

Part B Development Strategy and Hints:

This is meant to be an exercise in linked list manipulation. As a result, you must adhere to the following rules:

- You must use our `Node` structure for your lists. You are not allowed to modify it.
- You may not construct any arrays, `Vector`s, `ArrayList`s, stacks, queues, sets, maps, STL containers, or other data structures from the libraries; you *must* use linked nodes.
- If there are *N* names in the list of `string`s passed to your constructor, you should create exactly *N* new `Node` structures in your constructor. As people are eliminated, you have to move their node from the game ring to the graveyard by changing pointers, without creating any new node objects.

Your constructor creates the initial game ring nodes, and then your class may not create any more new `Node` objects for the rest of the program. You are allowed to declare as many local variables of type `Node*` (pointers to a node) as you like. Declaring a `Node*` variable does not create a new object and therefore doesn't count against the limit.

The `eliminate` method is the hardest one, so **write it last**. Use debug print statements and/or the debugger liberally to find problems in your code. You will likely have a lot of errors related to `NULL` pointers, infinite loops, etc. and will have a very hard time tracking them down unless you are comfortable with debugging techniques.

You may want to write your own **testing code**. The provided main program is not exhaustive; it requires every method in order to compile, and it never generates any of the exceptions you have to handle. You may share testing code with others on the message forum so long as the code does not give away details of your program solution. If you have bugs or exceptions in your code, print the state of your list and each `Node` object with temporary **debug print statements to `cout`**. You can do the same for the tiles in Part A. The provided `Tile` and `Node` structures both have `toString` and `<<` members for easy printing. (Though Part A is a graphical program, text output to `cout` does still appear on the console.) Note that printing a pointer to a node is not the same as printing a node.

A word of caution: Some students try to store the game ring in a **"circular" list**, with the list's final element storing a `next` reference back to the front element. But we discourage you from implementing the program in this way; we strongly suggest that you follow the normal convention of having `NULL` in the `next` field of the last node. Most novices find it difficult to work with a circular list; it is easy to end up with infinite loops or other bugs. There is no need to use a circular list, because you can always get back to the front via the fields of your `AssassinsList`. If you feel strongly that you want to use a circular list, you are allowed to do so, but it is likely to make the program harder.

Style Guidelines and Grading:

All items mentioned in the "Implementation and Grading" from the previous assignment(s) specs also apply here. Please refer to those documents as needed. Note the instructions in the previous assignments about procedural decomposition, variables, types, parameters, value vs. reference, and commenting. Don't forget to **cite any sources** you used in your comments. Refer to the course **Style Guide** for a more thorough discussion of good coding style.

Part of your grade will come from appropriately utilizing the data structures we ask for, namely a growable array in Part A and a pair of linked lists of nodes in Part B. **Redundancy** is another major grading focus; some methods are very similar, and you should avoid repeated logic as much as possible. Do not use **recursion** on this assignment.

For full credit, your code should have no **memory leaks**. Free any allocated memory in each class's destructor.

As for **commenting**, place a descriptive comment header on each file you submit. In your .h files, place detailed comment headers next to every member explaining its purpose, parameters, what it returns, any exceptions it throws, assumptions it makes, etc. You don't need to put any comment headers on those same members in the corresponding **.cpp** file, though you should place inline comments as needed on any complex code in the bodies.

Please remember to follow the **Honor Code** on this assignment. Submit your own work; do not look at others' solutions. Do not give out your solution; do not place a solution on a public web site or forum.