Assignment 5b: Tiles

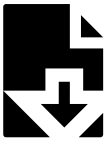
Assignment by Cynthia Lee, based on an assignment by Marty Stepp and Victoria Kirst. Originally based on a nifty assignment idea from Mike Clancy of UC Berkeley.

Outline:

Description Implementation Style FAQ Extras

This is a **pair assignment**. You are allowed to work individually or work with a single partner. If you work as a pair, **comment both members' names** on top of every submitted code file. Only one of you should submit the assignment; do not turn in two copies.

Files and Links:



Starter Code

We provide a ZIP archive with a starter project that you should download and open with Qt Creator. You will edit and turn in only the following file. The ZIP contains other files/libraries; do not modify these. Your code must work with the other files unmodified. If you want to declare function prototypes, declare them at the top of your **.cpp** file, not by modifying our provided **.h** file.

- **tilelist.cpp**
- **tilelist.h**

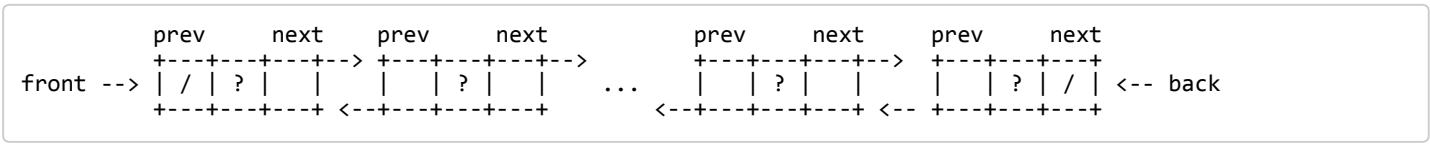
demo JAR

Problem Description:

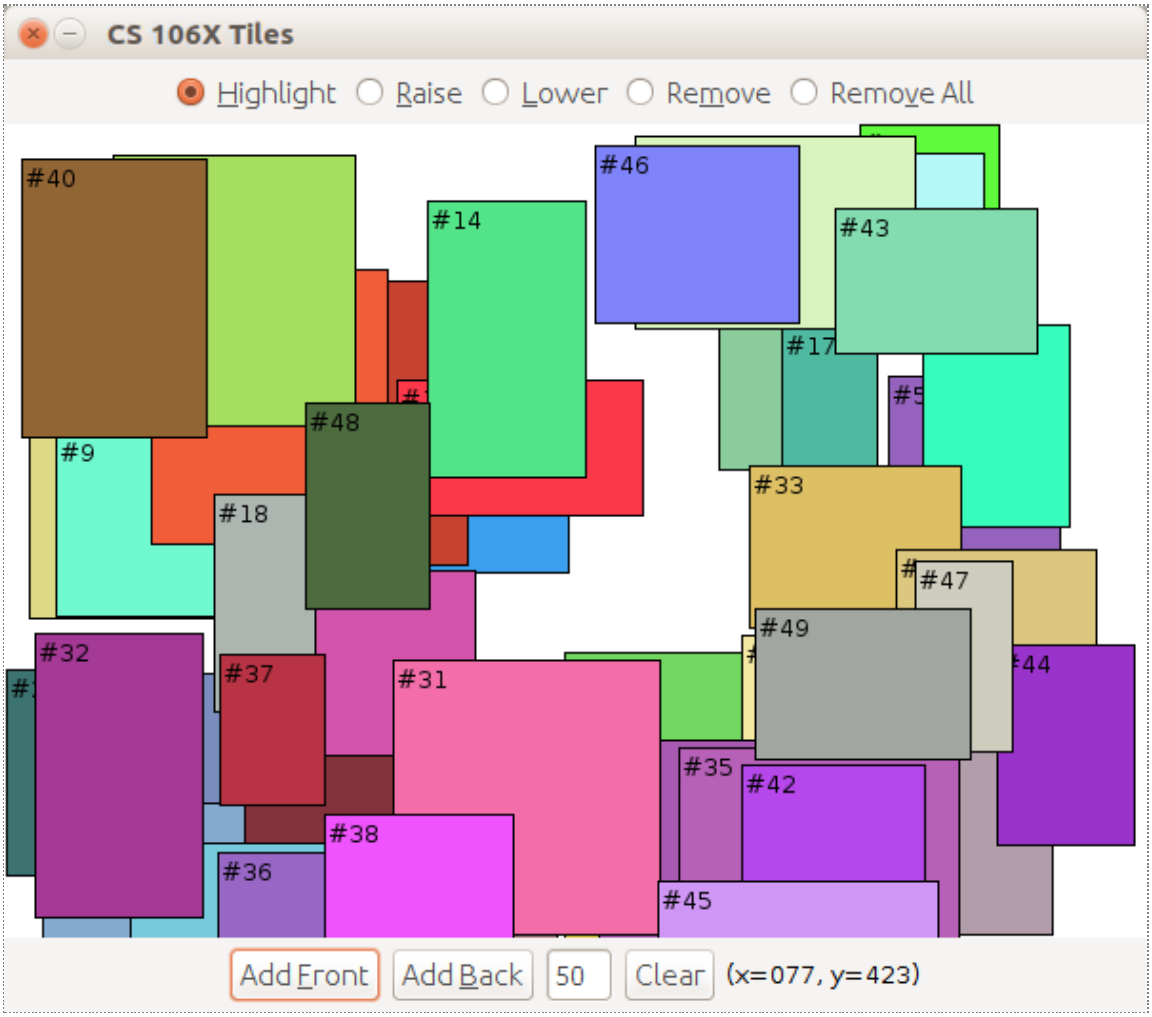
In this problem you will write the logic for a graphical program that allows the user to click on rectangular tiles. This program is a bit like the window manager of your operating system. When the program runs, several random rectangular "tiles" will appear in the window. Some tiles overlap and occupy the same (x, y) pixels. We think of the tiles as having a "z-ordering" where the tile created/added later is "on top" of prior tiles and is able to partially cover them on the screen.

Your job is to write a **class `TileList`** that implements the collection of tiles using a **doubly-linked list**. Your class maintains a pointer to the **front** of your linked list, which represents the "top" tile, and each tile afterward is below it in the z-ordering. Because this is a doubly-linked list, you also maintain a pointer to the **back** of your linked list, which represents the "bottom" tile, the last one in the z-ordering. Each individual node contains data about a tile as well as two pointers: a link to the **next** tile (the

one below it in the z-ordering) and **previous** tile (the one above it in the z-ordering). This allows you to traverse your linked list in either direction, which will be useful for the various operations we want to implement. In general a doubly-linked list looks something like the following diagram:

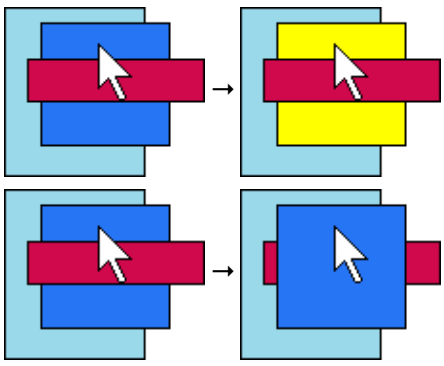


This program is graphical, but you do not directly draw any graphics yourself; all of the graphical interface code has already been written for you, and your code just has to manage the tiles and keep them in the proper order. The main client program and graphical user interface are provided for you as **tilegui.cpp/h**. When the program runs, it will create a graphical window that allows you to add tiles to your tile list at the front (top) or back (bottom). Each tile's position, size, and color are randomly generated by the main program. You must write the code to place the tiles into a linked list as they are pushed, to draw the tiles on the screen, and to perform operations on tiles when the user clicks on them.

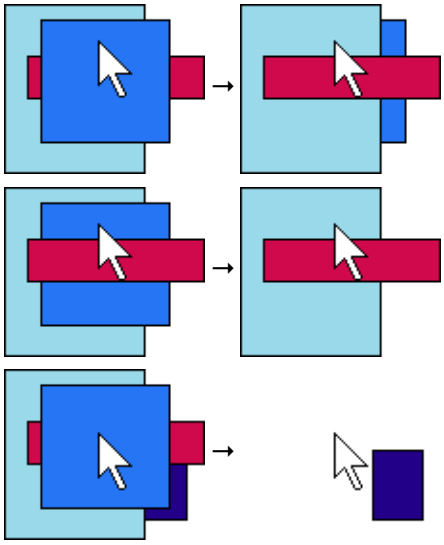


Depending on the user's action, one of several different **actions** occurs:

- **Highlight:** If the user clicks a tile while the "Highlight" radio button is selected, that tile's color is changed to yellow. (To do this, set the color member variable of the tile node to the string "yellow".) Its x/y position and z-ordering do not change.
- **Raise:** If the user clicks a tile while the "Raise" radio button is selected, that tile is moved to the very top of the list's z-ordering (the front of the linked list).
- **Lower:** If the user clicks a tile while the "Lower" radio button is selected, that tile is moved to the very bottom of the list's z-ordering (the back of the linked list).



- **Remove:** If the user clicks a tile while the "Remove" radio button is selected, that tile is removed from the tile list and disappears from the screen.
- **Remove All:** If the user clicks an (x, y) pixel while the "Remove All" radio button is selected, all tiles that touch that (x, y) position are removed from the tile list and disappear from the screen.
- **Add to Front/Back:** If the user clicks one of the Add buttons, one or more new random tiles are created and added to the front or back of the list.
- **Clear:** If the user clicks the Clear button, all tiles are removed from the list and screen.



If the user clicks a pixel that is occupied by more than one tile, the top-most of these tiles is used. If the user clicks the window at a pixel not occupied by any tiles, nothing should happen.

Your code does not need to directly detect **mouse clicks**. Our provided code detects the clicks and responds by calling various member functions on your **`TileList`** object, as described below.

Implementation Details:

Provided TileNode Structure:

Your **`TileList`** class stores its list of tiles as a linked list of **`TileNode`** structures. A **`TileNode`** stores data about a single rectangular tile, along with links to a previous and next node to facilitate building a doubly-linked list. The node structure is provided to you in the files **TileNode.h/cpp**; do not modify those files. Each node has the following public members. Each member is assumed to have a Big-Oh of O(1).

Member Name	Description
`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"`** or a color name such as **`"red"`**.
`prev`, **`next`**	Pointers to neighboring nodes; set to **`NULL`** if there is no next structure.
`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.
`operator <<`	Allows you to print a node to the console for debugging. (Remember to print the node itself and not a pointer to a node.)

*members of provided **`TileNode`** structure*

Here is a brief example of declaring and using a single tile node structure:

```
// example of declaring and using a tile node
TileNode* node = new TileNode(x, y, w, h);
node->prev = NULL;
node->next = NULL;
if (node->contains(x, y)) {
    node->draw(window);
}
cout << *node << endl;   // print the node for debugging
```

You are required to have **exactly two private member variables** in your class: (1) a pointer to the **front** node in the list (the top of the z-ordering), and (2) a pointer to the **back** node in the list (the bottom of the z-ordering). When a new TileList is created, initially the tile list is empty, meaning that the front and back pointers are null. You are not allowed to have any other member variables in your class. For example, you are not allowed to keep a member variable pointing to any other nodes in the linked list, nor are you allowed to keep an `int` for the list's size/length.

TileList Members:

You must write the following public members in your `TileList` class. Each member must run within the Big-Oh runtime noted; the *N* in this context means the length of the linked list. You may assume that every parameter passed is valid unless otherwise specified. See the next page for some general restrictions about proper linked list usage and style.

We provide you a partially complete version of your header file **TileList.h** that declares the following members. You can add other member functions (such as helper functions) as long as they are `private`.

Member Name	Description	Big-Oh
`TileList()`	In this constructor you initialize a new empty list of tiles.	O(1)
`~TileList()`	In this destructor you must free all dynamically allocated memory used by your nodes.	O(*N*)
`list``.debug();`	This member function is **optional**. It is called when the user clicks the Debug button in the GUI. You can put anything you want in this function, such as a loop to print your list or any other information to the console. Or you can leave it completely blank; it is up to you.	O(?)
`list``.addFront(`*`x, y, w, h, color`*`);`	In this member function you should add a new tile with the given position, size, and color to the front of your linked list (the top of the z-ordering).	O(1)
`list``.addBack(`*`x, y, w, h, color`*`);`	In this member function you should add a new tile with the given position, size, and color to the back of your linked list (the bottom of the z-ordering).	O(1)
`list``.drawAll(`*`window`*`);`	In this member function you are passed a reference to a `GWindow` object, and you should draw all tiles in your list on that window in bottom-to-top (back-to-front) order using each tile node's `draw` function.	O(*N*)
`list``.highlight(`*`x, y`*`);`	Called by the provided GUI when the user clicks the given x/y coordinates when the Highlight button is selected. In this member function, if these coordinates touch any tiles, you should set the topmost (closest to the front) of these tiles to have a color member value of "yellow". The tile should not move at all in the z-ordering, and no other tiles should change color. If no tile touches these x/y coordinates or they are out of bounds, nothing should happen.	O(*N*)

`list.raise(x, y);`	Called by the provided GUI when the user clicks the given x/y coordinates when the Raise button is selected. In this member function, if these coordinates touch any tiles, you should move the topmost (closest to the front) of these tiles to the front of the list (top of the z-ordering). All other tiles should remain at their original relative z-ordering. If no tile touches these x/y coordinates or they are out of bounds, nothing should happen.	O(*N*)
`list.lower(x, y);`	Called by the provided GUI when the user clicks the given x/y coordinates when the Lower button is selected. In this member function, if these coordinates touch any tiles, you should move the topmost (closest to the front) of these tiles to the back of the list (bottom of the z-ordering). No other tiles should be modified; all other tiles should remain at their original relative z-ordering. If no tile touches these x/y coordinates or they are out of bounds, nothing should happen.	O(*N*)
`list.remove(x, y);`	Called by the provided GUI when the user clicks the given x/y coordinates and the Remove button is selected. In this member function, if these coordinates touch any tiles, you should delete the topmost (closest to the front) of them from the list. No other tiles should be modified; all other tiles should remain present in the list and at their original relative z-ordering. If no tile touches these x/y coordinates or they are out of bounds, nothing should happen.	O(*N*)
`list.removeAll(x, y);`	Called by the provided GUI when the user clicks the given x/y coordinates and the Remove All button is selected. In this member function you should delete all tiles that touch or contain this x/y point from the list. (See screenshots earlier in this document.) No other tiles should be modified; all other tiles should remain present in the list and at their original relative z-ordering. If no tile touches these x/y coordinates or they are out of bounds, nothing should happen.	O(*N*)
`list.clear();`	In this member function you should remove all tiles from the list and free their associated memory.	O(*N*)

members you must write in ***`TileList`*** *class*

You will need to modify the provided **TileList.h/cpp** files to complete them. In particular, you will need to do the following:

Add descriptive comments to TileList.h/cpp. Both files should have top-of-file headers; one file should have header comments atop each member function, either the .h or .cpp; and the .cpp should have internal comments describing the details of each member function's implementation.

Declare your necessary two private member variables in TileList.h, along with any private member functions you want to help you implement the required public behavior. Your inner data storage *must* be a linked list of tiles; do not use any other collections or data structures in any part of your code.

Implement the bodies of all member functions and constructors in TileList.cpp. Notice that several operations are similar or have common aspects. 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.)

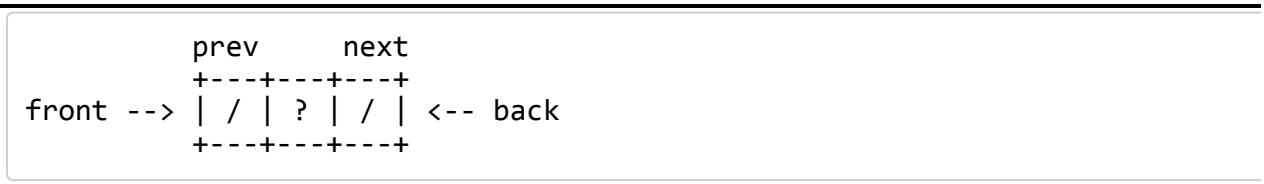
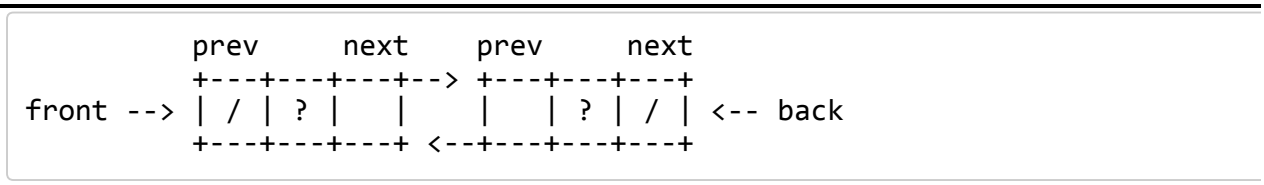
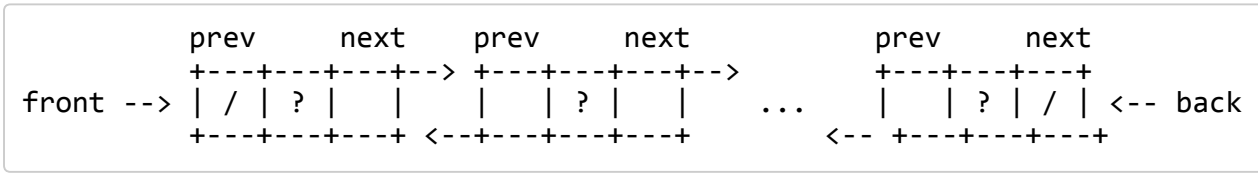
Development Strategy and Hints:

We suggest coding basic operations first, such as **`addFront`** and **`drawAll`** only. Then write **`highlight`**, because it has some similar elements to other member functions like **`raise`** and **`remove`** but without needing to modify the pointers in the linked list. Then write the rest of the list-modifying functions. To figure out if a tile touches a given x/y pixel, call the **`TileNode`**'s **`contains`** member function. Do not use **`GRect`** or **`GRectangle`** on this program, nor the **`LinkedList`** library class.

Draw pictures: When manipulating linked lists, it is often helpful to draw pictures of the linked list before, during, and after each of the operations you perform on it. Manipulating linked lists can be tricky, but if you have a picture in front of you as you're coding it can make your job substantially easier.

Strongly recommended helpers: While you should decide for yourself how best to decompose the problem, we strongly recommend you write helper functions for common code between operations. One common operation is the task of figuring out which tile was clicked on at a given x/y position. Another common operation is to extract/remove a node from its place in the doubly-linked list so that it can be moved elsewhere (to the front/back) or removed entirely. Helper functions like these can greatly simplify your code and help you avoid redundancy and bugs.

List states: When processing a doubly-linked list, you should consider the following states and transitions between them in your code as you add and remove elements: You should also, for each state below, think about the effect of adding a node in the front, middle, and back of the list.

zero nodes	
one node	
two nodes	
N nodes	

Don't forget that you can put some code in your **`debug`** member function to print information about the current state of your list.

Style Details:

As in other assignments, you should follow our **Style Guide** for information about expected coding style. You are also expected to follow all of the general style constraints emphasized in the Homework 1-4 specs, such as the ones about good problem decomposition, parameters, redundancy, using proper C++ idioms, and commenting. The following are additional points of

emphasis and style contraints specific to this problem. (Some of these may seem overly strict or arbitrary, but we need to constrain the assignment to force you to properly practice pointers and linked lists as intended.)

Redundancy and helpers: If your implementation has a common operation, make a private helper function and call it multiple times in that file.

Commenting: Add descriptive comments to your .h and .cpp files. Both files should have top-of-file headers. One file should have header comments atop each member function (either the .h or .cpp; your choice). The .cpp file should have internal comments describing the details of each function's implementation. Also put a comment atop every member function that states its Big-Oh. For example, the **`addFront`** operation runs in constant time, so you should put a comment on that function that says "O(1)."

Restrictions on pointers: The whole point of this assignment is to practice pointers and linked lists. Therefore, do not declare or use any other **collections** in any part of your code; all data should be stored in your linked list of nodes. There are some C++ "smart pointer" libraries that manage pointers and memory automatically, allocating and freeing memory as needed; you should not use any such libraries on this assignment.

Restrictions on private member variables: As stated previously, the only member variables (a.k.a. instance variables; private variables; fields) you are allowed to have are a **`TileNode*`** pointer to the front and back of your list. You may not have any other member variables. Do not declare an **`int`** for the list size. Do not declare members that are pointers to any other nodes in the list. Do not declare any collections or data structures as members.

Restrictions on modifying function headers: Please do not make modify the **`TileList`** class's public constructor or public member functions' names, parameter types, or return types. Our client code should be able to call public member functions on your list successfully without any modification.

Restrictions on creation and usage of nodes: The only place in your code where you should be using the **`new`** keyword is in the **`addFront`** and **`addBack`** functions. No other members should use **`new`** or create new nodes under any circumstances. You also should not be modifying or swapping nodes' data values, such as their x, y, width, and height. In other words, you should implement all of the linked list operations like **`raise`** and **`remove`** by manipulating node pointers, not by creating entirely new nodes and not by modifying "data" of existing nodes.

You also should not create any more **`TileNode`** structures than necessary. For example, if the client has called **`addFront`** 6 times, your code should have created exactly 6 total **`TileNode`** objects; no more, no less. You shouldn't, say, have a seventh empty "dummy" node at the front or back of the list that serves only as a marker, and you shouldn't accidentally create a temporary node object that is lost or thrown away. You can declare as many local variable *pointers* to nodes as you like.

Restrictions on traversal of the list: Any member function that modifies your tile list should do so by traversing across your linked list a single time, not multiple times. For example, in functions like **`drawAll`**, **`highlight`**, **`raise`**, **`lower`**, **`remove`**, and **`removeAll`**, do not make one traversal to find a node of interest and then a second traversal to manipulate/modify that node. Do the entire job in a single pass.

Since your program uses a doubly-linked list, you can traverse it in a forward or backward direction as desired. Your code should always choose an appropriate direction in which to traverse the list. For example, if you are looking for something near the back of the list, or if you need to visit nodes in back-to-front order, start from the back and loop backward. Otherwise, start at the front and loop forward.

Regardless of direction, do not traverse the list farther than you need to. That is to say, once you have found the node of interest, do not unnecessarily process the rest of the list; break/return out of code as needed once the work is done.

Avoiding memory leaks: This item is listed under Style even though it is technically a functional requirement, because memory leakage is not usually visible while a program is running. To ensure that your class does not leak memory, you must delete all of the node objects in your tile list whenever data is removed or cleared from the list. You must also properly implement a destructor that deletes the memory used by all of the linked list nodes inside your **`TileList`** object.

Frequently Asked Questions (FAQ):

For each assignment problem, we receive various frequent student questions. The answers to some of those questions can be found by clicking the link below.

Tiles FAQ (click to show)

Possible Extra Features:

Here are some ideas for extra features that you could add to your program:

- **Changeable tile colors:** Make it so that tiles can change colors, perhaps by right-clicking them, or perhaps automatically on a timer.
- **Draggable tiles:** You'd have to modify the provided GUI for this one. Make it so that if you press and hold the mouse on a tile and then drag it, the tile moves by changing its x/y position. maybe something fun with the tile colors (right-click to change tile color?).
- **Resizable tiles:** Make it so that tiles can be resized by clicking and dragging their bottom-right corners. (Tricky!)
- **Other:** If you have your own creative idea for an extra feature, ask your SL and/or the instructor about it.

Indicating that you have done extra features: If you complete any extra features, then in the comment heading on the top of your program, please list all extra features that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can look at their code easily).

Submitting a program with extra features: Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your feature(s) cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit two versions of your program file: a first one with the standard file name without any extra features added (or with all necessary features disabled or commented out), and a second one whose file name has the suffix **-extra.cpp** with the extra features enabled. Please distinguish them in by explaining which is which in the comment header. Our turnin system saves every submission you make, so if you make multiple submissions we will be able to view all of them; your previously submitted files will not be lost or overwritten.

Honor Code Reminder:

You are expected to follow the **Stanford Honor Code**.

- In your file's comment header, list your name (and your partner's name, if you worked in a pair); also cite *all* sources of help, including books, web pages, friends, section leaders, etc.
- Do not consult any assignment solutions that are not your (pair's) own.
- Do not attempt to disguise any code that is not your (pair's) own.
- Do not give out your assignment solution to another student.
- Do not post your homework solution code online. (e.g. PasteBin, DropBox, web forums).
- Please take steps to ensure that your work is not easily copied by others.

Remember that we run **similarity-detection software** over all solutions, including this quarter and past quarters, as well as any solutions we find on the web.

If you need help solving an assignment, we are happy to help you. You can go to the LaIR, or the course message forum, or email your section leader, or visit the instructor/Head TA during their office hours. **You can do it!**