Lab 4 V3

Bonus writeup is complete. Worth 2 points.

Dates

Prototypes demo by Friday 1-March-2024

Early Tuesday 5-March-2024

On-Time: Thursday 7-March-2024

Late until Friday 8-March-2024

Take a break over spring break.

Contents

Initializations

The init sequence is more complex. See the section on command line arguments. Before the sim is run, the arguments have to be right. Init checks for the presence of arguments but doesn't validate the

filename – that's input's job. So master init has 2 parts,. Check arguments first before any attempt at calling btp_initialize, there is no reason to kick into graphics mode if the sim isn't going to run.

Linked List

In lab 4 you will write the 4 functions that we use in the linked list library:

- insert
- iterate
- any
- sort
- deleteSome

Signatures

Your code **must** use the exact signatures used by the library functions; same name, same parameters, same return type.

Your code must use the following declaration for a node:

```
typedef struct Node{
        struct Node *next;
        void *data;
        }Node;
```

The node declaration should **not** be visible outside your linkedlist.c file. This is the same declaration that the library code uses. If you use it, you can mix calls into your code and calls into the library code freely. If you use anything else, you risk going insane trying to track down mysterious pointer bugs in code that looks perfect.

Files

All of your linked list code should be in your linkedlist.c file. All supporting functions must be declared static (internal linkage), leaving only the functions listed above visible outside the file. The headers.sh script should work correctly (it doesn't create declarations for internal linkage functions).

Dynamic Memory

As before, allocation and free calls should be inside a function that has a static int to count the number of allocations and frees. So you might want an "allocate_node()" and "free_node()" function with appropriate parameters and return types. They must be in linkedlist.c, they must have internal linkage. Your code should print diagnostics in text mode:

```
DIAGNOSTIC: 1 nodes allocated.
DIAGNOSTIC: 1 nodes freed.
```

Your list code must tolerate allocation failures. If insert fails to allocate a node, return false and let the application deal with it. This *will* happen when you implement the unreliable memory code (See Unreliable).

Visibility

The list code must **not** #include the structs.h or equivalent file. In short, the list code must not know in any way what the data is. The list deals in void pointers and only void pointers for data.

No code outside linkedlist.c is allowed to know what a node is.

Only the functions that provide the public interface to the list are allowed to be visible outside the list code.

Plan of Attack

Until you have all the required functions completed, retain the **`-llinkedlist`** item in your makefile rules. This lets you use your code (linkedlist.c) as well as the library (for functions you haven't written yet). Do the easy ones first:

1. iterate
2. any
3. insert
4. deleteSome
5. sort

You don't have to do any dynamic memory until you get to insert and deleteSome. You could get 4 prototypes done with the list code alone. Recycle some of your lab 3 list prototypes and point them at your code.

Have your list functions issue a debug print message so that you know when your code is being used and when the library is being used.

Once you have the 4 required functions written, **remove** the **`-llinkedlist`** item in your makefile and test to be sure that your code is being used.

There is a more elaborate plan of attack in the L4_00_Dates document.

Terse

You are allowed to use the terse insert code from the slides, properly modified, for your insert. If you use terse insert, you are **required** to use terse delete. Terse insert is quite short. Terse delete is something like 8 lines.

Ten-line Limit

If you use the long method for insert, your insert does not have to meet the ten-line limit. If you use the long form of deleteSome, that function does not have to meet the ten-line limit.

Sort

If you implement sorting, do not change nodes, swap data instead. I suggest bubble sort. Sort is not easy. Your sort must not infinite loop. You might want to code it in a way that deals with equality comparisons gracefully.

I/O

In lab 4, all objects will be read from a file whose name is given on the command line. Some input might be read directly off the keyboard for a bonus point.

The command line must have the right number of arguments

The initialization checks in lab 4 are more extensive than lab 3.

- If argc is less than 2, the lab should give an error and decline to run.
- If argc is 3 and bonus mode is not present in the code, the lab should give an error and decline to run.
- If argc is 3 and the lab is in TEXT mode, the lab should give an error and decline to run regardless of whether bonus code is on-board or not.
- If argc is 3, the lab is in GRAPHICS modes, and the bonus code is on-board, the lab must run in bonus mode if all other initializations pass.

The output below is for when no file name is given on the command line (the first bullet point above):

```
[kirby.249@cse-sl1 lab4]$ lab4
ERROR: insufficient args - no filename given
Returning 1
Total run time is 0.000042677 seconds.
[kirby.249@cse-sl1 lab4]$
```

The output below is the same for the second and third cases given in the bullet points above:

```
[kirby.249@cse-sl1 lab4]$ lab4 x7c.btp bonus
ERROR: Bonus code is not present.
Returning 1
Total run time is 0.000033617 seconds.
[kirby.249@cse-sl1 lab4]$
```

The file must open for reading

The input filename is given as the first argument to the lab4 command. If just the file is given, the code will attempt to run the lab in non-bonus mode. If the file cannot be opened for reading, the lab should decline to run the simulation. If the file opens, a DIAGNOSTIC will be printed if in TEXT mode:

```
[kirby.249@cse-sl1 lab4]$ lab4 xf.btp
DIAGNOSTIC: Successfully opened xf.btp for reading.
```

If a file that cannot be opened is given on the command line, an ERROR message must be printed (if the lab is in TEXT mode). Not being able to open the file counts as bad input and the input routine should return false so that main knows that there was bad input.

```
[kirby.249@cse-sl1 lab4]$ lab4 this_file_does_not_exsist
ERROR: Unable to open this_file_does_not_exsist for reading.
Returning 2
Total run time is 0.000491619 seconds.
[kirby.249@cse-sl1 lab4]$
```

Use the **fopen** function to open the file. Google that function and read up on file I/O. Open it in read-only mode using "r" as the mode.

After reading, the file must be closed

After input completes, but certainly before the lab ends, close the input file pointer using **fclose** and print a DIAGNOSTIC message (if in TEXT mode).

```
DIAGNOSTIC: Input file closed.
```

Reading data from the file

Use **fscanf** instead of **scanf** to read mascots and coins.

Reading user input – bonus

If the bonus flag is set in the sim structure, your code will loop performing non-sleeping reads one character at a time using **btp_getch()** which is similar to **getchar()**. This call always returns immediately regards of whether a typed-in character awaits on the input stream or not. If a character was waiting, that character will be returned. If no character was waiting, it returns ERR, a macro defined in <curses.h> as having a value of -1. It may be best to add the following line to input.c instead of including curses.h.

```
#define ERR     (-1)
```

If a value other than ERR is returned, record that character and then read again. If ERR is returned, ignore it and the loop is done. ERR means that the user hasn't pressed any keys since we read that last one. We will read all waiting characters so that we can do all of the teams at the same time.

The input.c file will need to include btp.h to get access to **btp_getch**.

Testing

You should get the same numbers for text mode as lab 3 if the data file lacks any objects from team 1-3. A test file that lacks those teams will be provided.

Bonus mode should be tested with xBonus.btp to put the three playable mascots into play. Team 3 is hard to play without a numeric keypad.

Unreliable Allocation

Dealing with allocation failures

In lab 4, your code will be subject to allocation failures. Instead of calling malloc or calloc, it will call modified versions found in yet more libraries. The reliable library works as expected. The unreliable library fails at "random" times, causing objects and nodes to fail to allocate. Your code must gracefully deal with these events as described here.

You will need to download altmem.zip.

Recovery

Your code should not quit on memory failure. If a mascot or coin fails to allocate, return to the input loop and carry on. If a node fails to allocate, insert returns false and the lab 4 code will have to free the

allocated object since the list doesn't have a node to hold it. This is *a* recovery strategy and not *the only* recovery strategy. It presumes that our code took measures to make future allocations work and it presumes that rolling over and playing dead at the first sign of difficulty is not acceptable. The Apollo 11 Eagle spacecraft went into computer overload on descent to the moon but the software was built to keep the important stuff alive even if it meant the other stuff starved. One of the questions to ask when you are given a programming assignment is, "what is our error recovery strategy?"

Details

Code Changes

If a mascot fails to get inserted onto the linked list, it must be freed.

Change every malloc, calloc, and free call to use the "alternative" equivalents instead.

Note that the "libraries" are not actual libraries. They are .o files that have been compiled with –g so that you can run gdb without issues. Here is the altmem.h header file found in the altmem.zip archive in piazza:

```
/* alternative memory calls (both reliable and unreliable use these) */
void *alternative_calloc(size_t nmemb, size_t size);
void *alternative_malloc(size_t size);
void alternative_free(void *ptr );
```

You will want to

```
#include <stdlib.h>
#include "altmem.h"
```

in the 2 files you have that deal in dynamic memory. Any file that includes altmem.h must *first* include stdlib.h. Note also that other header files might require stdio.h. When including files, put standard headers <stdio.h> before custom headers "altmem.h" [note the <> versus ""].

There will be point deductions if any of the non-prototype code you turn in directly calls malloc, calloc, or free.

Your dynamic memory code should be restricted to one file on the application side (this is a lab 3 mandate). Likewise your linkedlist.c file should be the only file that deals in dynamic memory for nodes.

What follows is example makefile lines as guidance for lab4. It builds the lab4 target – a mandatory target – using the reliable memory allocation code. The lab4**u** target builds against the unreliable memory allocation code. Note the highlighted items. Modify the dependency list to suit, but leave either reliable or unreliable in there (but not both).

```
# a lab 4 makefile prior to writing linked list code might resemble this

lab4u: lab4.o n2.o memory.o (other .o files here) unreliable.o
        gcc -g -o $@ $^  -lm -L. -lbtp -lncurses –llinkedlist

lab4: lab4.o n2.o memory.o (other .o files here) reliable.o
        gcc -g -o $@ $^  -lm -L. -lbtp -lncurses –llinkedlist
```

Prototypes for altmem

The prototypes below are merely tests if you use the entire lab code base.

Suggested prototype / test: Take your existing code; add the word `alternative_` in front of all calloc, malloc, and free calls. Then adjust your makefile to add reliable.o to the rule that builds that code. Then make and test.

Suggested prototype / test: Create a nearly identical target to the prototype above in your makefile. Change reliable.o to **un**reliable.o and test. Use this to test code you wrote in lab 3 that never got tested because malloc and calloc never failed. Get those bugs out early. The unreliable allocator fails on the first call, the third call, and the sixth call… So you should be able to test the 2 test cases: fails to allocate either a mascot or coin and fails to allocate a node after a successful allocation of a mascot or coin. The pattern is fail, one success, fail, two successes, fail, three successes, fail… This means that we get a pseudo-random failure chain that should test everything but lets us get on with things if we have enough stuff.

Initial testing for allocation failures

Here is debug mode for xf.btp with unreliable allocation in memory.c using the linked list library (which always allocates nodes successfully and does not have DEBUG messages).

```
DEBUG: sim: trying to allocate Red mascot
ERROR: memory: failed to allocate 56 bytes.
DEBUG: sim: unable to allocate Red mascot

DEBUG: sim: trying to allocate Red coin
DIAGNOSTIC: allocation #1 allocated 24 bytes
DEBUG: memory: allocated pointer is 0x23df010
DEBUG: sim: trying to insert Red coin
DIAGNOSTIC: 1 nodes allocated.

DEBUG: sim: trying to allocate White mascot
ERROR: memory: failed to allocate 56 bytes.
DEBUG: sim: unable to allocate White mascot

DEBUG: sim: trying to allocate Green coin
DIAGNOSTIC: allocation #2 allocated 24 bytes
DEBUG: memory: allocated pointer is 0x23df050
DEBUG: sim: trying to insert Green coin
DIAGNOSTIC: 2 nodes allocated.

DEBUG: sim: trying to allocate Cyan mascot
DIAGNOSTIC: allocation #3 allocated 56 bytes
DEBUG: memory: allocated pointer is 0x23df090
DEBUG: sim: trying to insert Cyan mascot
DIAGNOSTIC: 3 nodes allocated.

DEBUG: sim: trying to allocate Yellow coin
ERROR: memory: failed to allocate 24 bytes.
DEBUG: sim: unable to allocate Yellow coin
```

```
DEBUG: sim: trying to allocate Blue coin
DIAGNOSTIC: allocation #4 allocated 24 bytes
DEBUG: memory: allocated pointer is 0x23df0f0
DEBUG: sim: trying to insert Blue coin
DIAGNOSTIC: 4 nodes allocated.

DEBUG: sim: trying to allocate Magenta coin
DIAGNOSTIC: allocation #5 allocated 24 bytes
DEBUG: memory: allocated pointer is 0x23df130
DEBUG: sim: trying to insert Magenta coin
DIAGNOSTIC: 5 nodes allocated.

DEBUG: sim: trying to allocate Cyan coin
DIAGNOSTIC: allocation #6 allocated 24 bytes
DEBUG: memory: allocated pointer is 0x23df170
DEBUG: sim: trying to insert Cyan coin
DIAGNOSTIC: 6 nodes allocated.

DEBUG: sim: trying to allocate White coin
ERROR: memory: failed to allocate 24 bytes.
DEBUG: sim: unable to allocate White coin
scanf returned -1
```

Here is debug mode for xf.btp with unreliable allocation in memory.c using unreliable linked list code. Note that it requires both coins and mascots to be freed when the list cannot insert them. It also has coins and mascots failing to allocate in the first place.

```
DEBUG: sim: trying to allocate Red mascot
ERROR: memory: failed to allocate 56 bytes.
DEBUG: sim: unable to allocate Red mascot

DEBUG: sim: trying to allocate Red coin
DIAGNOSTIC: allocation #1 allocated 24 bytes
DEBUG: memory: allocated pointer is 0x1871010
DEBUG: sim: trying to insert Red coin
ERROR: linkedlist: Failed to malloc a Node
DEBUG: sim: freeing Red coin
DEBUG: memory: about to free 0x1871010
DIAGNOSTIC: 1 objects freed

DEBUG: sim: trying to allocate White mascot
DIAGNOSTIC: allocation #2 allocated 56 bytes
DEBUG: memory: allocated pointer is 0x1871010
DEBUG: sim: trying to insert White mascot
DIAGNOSTIC: 1 nodes allocated.
DEBUG: linkedlist: allocated pointer is 0x1871050

DEBUG: sim: trying to allocate Green coin
ERROR: memory: failed to allocate 24 bytes.
DEBUG: sim: unable to allocate Green coin
```

```
DEBUG: sim: trying to allocate Cyan mascot
DIAGNOSTIC: allocation #3 allocated 56 bytes
DEBUG: memory: allocated pointer is 0x1871070
DEBUG: sim: trying to insert Cyan mascot
DIAGNOSTIC: 2 nodes allocated.
DEBUG: linkedlist: allocated pointer is 0x18710b0

DEBUG: sim: trying to allocate Yellow coin
DIAGNOSTIC: allocation #4 allocated 24 bytes
DEBUG: memory: allocated pointer is 0x18710d0
DEBUG: sim: trying to insert Yellow coin
ERROR: linkedlist: Failed to malloc a Node
DEBUG: sim: freeing Yellow coin
DEBUG: memory: about to free 0x18710d0
DIAGNOSTIC: 2 objects freed

DEBUG: sim: trying to allocate Blue coin
DIAGNOSTIC: allocation #5 allocated 24 bytes
DEBUG: memory: allocated pointer is 0x18710d0
DEBUG: sim: trying to insert Blue coin
DIAGNOSTIC: 3 nodes allocated.
DEBUG: linkedlist: allocated pointer is 0x18710f0


DEBUG: sim: trying to allocate Magenta coin
DIAGNOSTIC: allocation #6 allocated 24 bytes
DEBUG: memory: allocated pointer is 0x1871110
DEBUG: sim: trying to insert Magenta coin
DIAGNOSTIC: 4 nodes allocated.
DEBUG: linkedlist: allocated pointer is 0x1871130

DEBUG: sim: trying to allocate Cyan coin
ERROR: memory: failed to allocate 24 bytes.
DEBUG: sim: unable to allocate Cyan coin

DEBUG: sim: trying to allocate White coin
DIAGNOSTIC: allocation #7 allocated 24 bytes
DEBUG: memory: allocated pointer is 0x1871150
DEBUG: sim: trying to insert White coin
DIAGNOSTIC: 5 nodes allocated.
DEBUG: linkedlist: allocated pointer is 0x1871170
scanf returned -1
```

Reference code issues

It's quite possible that the reference code doesn't deal with allocation failures even if it detects them. Be very sure that your code does properly deal with these issues. Check the DIAGNOSTIC numbers at the end to make sure. Check every insert call in the reference code as well as the mascot and coin allocation sequence.

Output Changes for Lab 4

The biggest output change is that the color of the object will be printed as a string in most of the places where the program does output.

Strings

Your program will need to have a function that will return a string when given a color number. Consider an array of strings. Use the following chart to map color numbers to color names. Your strings will not be in all capital letters, just have the first letter capitalized.

```
/* curses basic colors:
 * BLACK   0
 * RED     1
 * GREEN   2
 * YELLOW  3
 * BLUE    4
 * MAGENTA 5
 * CYAN    6
 * WHITE   7
 */
```

Formatting

Use %7s to print the color name. in tabular output (This is the main print for mascots. This code uses a function called "team" to transform a color number into a string).

```
printf("%3d    %7s    (%8.5lf, %8.5lf)    (%9.5lf, %9.5lf)\n",
    get_score(brutus), team(brutus->color),
    brutus->x_position, brutus->y_position,
    brutus->x_velocity, brutus->y_velocity);
```

You don't have to match the reference output in terms of whitespace but it is nicer for the graders. Example output below. See output files for complete examples.

Messages

Retain all existing messages and **update** the text mode ones with the color name. Use %s (not %7s) to format the color names.

```
White Brutus jumps at  1.50000,  0.50000
White Brutus hits the ceiling at 12.87500,  9.50000
White Brutus makes it to the flag!
Red Brutus hits the floor at 10.25000,  0.50000
Cyan Brutus finds loot at  9.00000,  0.50000
```

Tabular output

Update for less spacing on the main table and add color name to the regular and final tables. Here is the final output of xFast.btp:

```
Brutus makes it to the flag!

Pts       Team     (__X_____, __Y_____)     (__VX_____, __VY_____) ET= 4.40625
 22      White     (13.31250,  9.49512)     ( 14.00000,  -0.31250)
 18       Cyan     (13.25000,  4.29688)     (  3.00000,   8.25000)
```

```
 16        Red    (13.25000,  3.19824)    (  4.00000,   8.18750)

   Team     (__X_____, __Y_____)
  White     (10.50000,  0.50000)
Magenta     ( 8.50000,  0.50000)
   Blue     ( 7.50000,  0.50000)
 Yellow     ( 6.50000,  0.50000)
  Green     ( 5.50000,  0.50000)
    Red     ( 4.50000,  0.50000)
```

Diagnostics & Debug

See the linked list and unreliable sections for information and examples. The list code gets diagnostics similar to the dynamic memory code for objects. **Both get new mandates for debug messages**.

Memory.c gets these two mandate debugging messages. The first comes after a successful allocation and the diagnostic message that goes with it. The second one comes just before a free call. Your pointer values will change from run to run and might not match these.

```
DEBUG: memory: allocated pointer is 0xf64010
DEBUG: memory: about to free 0xf64210
```

The value of seeing these pointers is if free blows up you can tell what kind of values should have been passed to it. Add similar messages to the linked list code:

```
DEBUG: linkedlist: allocated pointer is 0xf64190
DEBUG: linkedlist: about to free 0xf642d0
```

You might want to add the debug messages in “deal_with_coin” and “deal_with_mascot” that are marked as coming from sim in the sample debug output in the section on unreliable memory. That are not mandated but they do give good visibility into what is going on.

Here are a repeat of the message needed by file IO

```
DIAGNOSTIC: Successfully opened xf.btp for reading.
ERROR: Unable to open this_file_does_not_exsist for reading.
ERROR: failed to close Input file.
DIAGNOSTIC: Input file closed.
```

Bonus Mode

In bonus mode, 3 of the colors are under manual control. You will need to extend the mascot structure to so that it keeps one more value, movement velocity. It will be set to the regular x velocity when the mascot is initialized. You will need to extend the simulation structure to hold a buffer of read in characters as well as the count of characters currently in the buffer. That count is initially zero. Your implementation **must** use two tables of function pointers, one to deal with team differences and one to deal with keystrokes.

Integration with Existing Code – Input

Inside the simulation loop, just before the move everyone call and right after the update to elapsed time, the code should call a function to read any keyboard input. That function:

- Sets the count of characters currently in the sim structure's buffer to zero
- Loops, reading a single character at a time until btp_getch returns ERR or the buffer size is reached
    - Puts each read in character into the next open spot in the buffer
    - Increments the count of characters currently in the buffer

Do not overflow your buffer – your code must guard against too much input. The buffer size must be a #define that lives in structs.h. Use 12 to start with.

Commands

You will need to implement 4 commands:

- Jump: If VY is zero, set it to the jump velocity. Generate a jump message as usual.
- Go left: Set x velocity to negative of movement velocity.
- Go right: Set x velocity to movement velocity.
- Stop moving: Set x velocity to zero.

Key Mappings

Color 1 uses wads characters to invoke those commands. (w is jump, a is left, d is right, s is stop.)
Color 2 uses ijlk to invoke those commands (similar order: i is jump…)
Color 3 uses 8465 characters to invoke those commands (similar order: 8 is jump…)

Integration with Existing Code - Commands

Go locate the call to "maybe_jump" in the move_brutus function in the physics.c file in the lab 3 reference code. Your code must always do the regular motion control on all mascots aside from jumping; basic motion, adjusting for the terrain, dealing with floors and ceilings. If your code is running in regular (non-bonus mode), call maybe_jump or your equivalent. In bonus mode, call a new function "take_action" instead.

Function Pointers – Take Action

The take action function will use the mascot color number to access a table of 8 function pointers. It will make a call using the correct function pointer. The table will be block scope and local to the take action function. The table will be initialized as follows:

- At subscript 0, a NULL value (Color 0 is invalid)
- At subscript 1,2, and 3, "react_to_input" (a function you will write)
- At subscripts 4-7 "maybe_jump" (or your existing equivalent)

This implies that the new function you write will share the same signature as maybe jump. A well crafted typedef might make this table easier to declare. It may be wise (it is not mandated) to check the color value and complain via btp_status if a bad color value is found. The take action function probably needs to live in physics.c since it calls maybe_jump.

Function Pointers – React to Input

The function that reacts to input will need a table of 256 actions. An action has 2 parts; the color it applies to and the function that will get invoked. An array of structs might be appropriate. The

following declaration creates such an array and initializes the whole thing to zero (a color we don't expect to see) and NULL (for the function pointers).

```
struct Action responses[256] = { {0, NULL} };
```

Each such action needs to be tied to a particular key. The following line might be illuminating:

```
responses['w'] = (struct Action) {1, jump};
```

This ties the lower-case w key to color number 1 and a function named "jump". You need 12 such lines, and they don't count against the 10-line limit. The cast allows us to assign many values to the fields of the struct in a single line the same initializations on a declaration work.

The function will loop through each character in the simulation structure's buffer and check if the entry for that character matches the color of the mascot that got passed in. If there is a color match, invoke the function, passing it the current mascot. These functions are very short. The four functions are shared by all 3 colors that are subject manual control:

```
responses['i'] = (struct Action) {2, jump};
```

The react to input function and the four functions it will directly call should be in their own file, perhaps called actions.c. The jump function will need to call the jump message output function. To debug this code, make use of btp_status to gain access to the status lines. You may need the help of sprint – go look it iup.

Scoring Details

TBA

No Global Variables! Code Must Compile!

Global variables is a -10 penalty. Errors or warnings in compilation means no credit for the lab, though any credit for prototypes you demo will remain.

Submission

Effectively: Same as before. Your zip file needs to contain README_LAB4, **all** the code to be graded, and a makefile sufficient to build all of the targets that are to be graded. **Those targets include lab4 and lab4u.** Do not include any of your .o files. **Do** include supplied .o files such as n2.o, reliable.o, unreliable.o. Any file you edited by hand (other than test data) probably needs to be included. Also include the libraries (*.a) in your zip.

Your lab is counted late if the makefile doesn't self-test the zip

All files that you edit by hand must have your name in them. Machine generated headers need your name in them.

Any code that comes from the instructor's lab 2 o 3 must retain proper attribution.

Defects in the submission can cause a zero score for the lab or worse.

README_LAB4 text:

THIS IS THE README FILE FOR LAB 4

BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I HAVE PERFORMED ALL OF THE WORK TO DETERMINE THE ANSWERS FOUND WITHIN THIS FILE MYSELF WITH NO ASSISTANCE FROM ANY PERSON OTHER THAN THE INSTRUCTOR OF THIS COURSE OR ONE OF OUR UNDERGRADUATE GRADERS.

The readme should contain your **name**, the number of **hours** you worked on the lab, and any comments you want to add. Of particular interest are what was hard or easy about the lab or places where programming reinforced what we went over in class.