# Lab 3

Structures, pointers, dynamic memory, function pointers, multi-file development.

## Version

V2. Added Table of Contents and Plan of Attack

V3. Changed “traverse” to “iterate”

## Dates

Early: Mon 28-Sept-2020

ON TIME: Thu 1-Oct-2020

Cutoff: Fri 2-Oct-2020

## Contents

## Intro

Lab 3 takes the basic simulation of lab 2 and runs all of the balls in the file at the same time. Unlike lab 2, our balls will change bounce off the side rails and slow down. If they go off the table, it will be into one of the six pockets. We'll store the balls in a linked list. We won't write the linked list in lab 3, it will be provided. When there are no balls on the table or if all of the balls on the table have stopped rolling then the program pauses for two seconds and gracefully shuts down, freeing any dynamic memory it still holds.

**Items in bold** are typically graded for.

## Changes

We read all of the balls before running the simulation. We store them all in dynamically allocated structs and put those in a linked list. Our simulation loop runs as long as there are balls on the table and at least one of them is moving. After moving the balls, delete the ones that made it off the table and into a pocket before iterating the simulation loop again.

## Functional layout

The following is not required, but can be taken as a guide to program structure for lab 3. It walks through "peeling the onion" in terms of functionality. It is given as a function name and its main purpose. This gives a way – not the only way – to organize from the top down. Left out are numerous low-level functions that you probably already have from lab 2. Left out are functions that are similar to functions listed. Read through these for implementation hints even if you don't use it.

Main: Time how long it takes to initialize, run, and teardown the sim.

Attempt sim: I need to read balls and if I get any I run the table

Read balls: As long as I can read balls, I add balls to the sim

Add Balls: I take read-in data, copy it to dynamic memory, and put that dynamic memory ball on the list in ball number order.

Allocate Ball: I am the only code that can call malloc. I keep track of all allocations.

Run the table: as long as the list has moving balls, I output them, I have the list move them, and I have the list delete the ones that made it off the table into a pocket. Somewhere in there I bump the clock.

Output: I draw balls, I print balls, and if we had Siri I'd have her talk to you about the balls.

Draw balls: I set up the graphics for one frame, have the list draw each ball, and then toss it all on screen and let you see it.

Draw A Ball: I can take the ball data and put it on the screen.

Print balls: I print out the header, have the list print each ball, and then toss on a blank line.

Print A Ball: I can take ball data and print it.

Move ball: Update X and Y as before using any helper functions required. If the ball hits a rail, I reflect it off and cut the speed to half. Then I apply friction. This basic description hides a lot of detail, which means it should involve a lot of short functions.

Basic Move: I move the ball the same way as lab 2

Bounce: If the new X value tells me the ball is across from a rail, I call Y reflect handler. If the new Y value tells me the ball is across from a rail, I call X reflect handler.

X reflect handler: If the ball is within a radius of the left edge, I call reflect left. If it is within a radius of the right edge, I reflect right.

Reflect Left: I give the ball a new X value and flip the sign of VX. Then I cut both velocities in half.

Apply friction: I get the friction percent for a ball and apply it to both velocities

Friction Percent: I compute the percentage of velocity remaining for a ball.

## The Table

The dimensions of the table have not changed. But we do need to know how big the pockets are.

- A side pocket is 4.5 inches wide and is centered on the long sides of the table.
- The corner pockets extend 3.1819805 inches in both X and Y.

These values belong in dimensions.h and you might consider computing where in X the side pocket starts and ends and putting those numbers in dimensions.h as well.

## Data Types

Like lab2, X, Y, VX, and VY data will be stored in doubles and computed as doubles. Ball number is still an int . You will store more data than in lab 2 – see the structs section for more info.

## Movement

In addition to moving the ball in X and Y as mentioned above, the movement code must also change. You will need to check for hitting a rail. You will need to add friction – the ball slows down as it rolls.

### Reflection

Because the ball never moves more than half of its radius in any one update, we can reliably detect when it hits a rail at the sides of the table. And since the corner pockets are larger than a ball diameter, no ball can hit two different side rails on the same update. [*Design note: computing deltaT the way we*

*do save us a ton of complexity*]. The rails only cover certain regions of the perimeter and the gaps between them form the pockets.

An example might be in order. Consider a ball moving left. The X value is approaching zero. The ball will either hit the left rail or it will go into either the upper or lower of the left corner pocket. The Y value tells us if the ball is approaching the rail or one of the pockets. When the X value is less than the ball radius, the ball has either hit the rail or it is about to drop into a pocket. If the ball has hit the rail, we reflect it.

To reflect off the left rail, we change the ball's X position and X velocity. The ball should have bounced when the edge of the ball contacted the rail. So however much the new position has the edge of the ball intruding into the rail is how much we want the edge of the ball to be away from the rail. Since the ball bounced off, it reverses direction in X, so we change VX to –VX. After we do that we cut in half both VX and VY – it takes energy to reverse the direction the ball is rolling.

If the ball is heading for a pocket, we leave it alone. When the center of the ball leaves the table, other code will delete that ball.

Reflection involves computing how far the edge of the ball has moved off the table (into the rail) and moving the ball so the edge of the ball is that far away from the rail. *Note that the position of the ball is the position of the center of the ball.* After we move the ball, we reverse one of the velocity components. After we do all that we cut both velocity components in half.

## Friction

After a ball has been moved, including reflection, apply friction. Friction is expressed in terms of acceleration. Acceleration is change in velocity over time. The value is based on coefficient of rolling friction "mu", the force of gravity, and the amount of time it is applied for. The equation amounts to:

dV = MU_ROLLING * G_IPS2 * dT

Where

mu = 0.01

G_IPS2= 386.08858267716 "gravity in inches per second per second"

dT is the value computed after all of the balls are read but none of them have been moved. It is computed once and does not change.

The dV obtained is the applied to the overall velocity of the ball , not VX or VY separately. The magnitude of the verall velocity can be obtained by

V = sqrt( VX * VX + VY * VY)

The sqrt function is in the math library. #include <math.h> and put –lm on the makefile rule that links all of the code together.

If dV is greater or equal to V, then the ball stops. If not compute the percentage that dV is of V. If you then compute 1- this percent, you get the percentage remaining. Multiply each of VX and VY by that remaining percent to get the new velocities.

*Suggested prototypes:* A function that takes VX, VY and returns a remaining percent might be in order. A function that applies friction using that percent might be handy, too. Use a calculator and numerous debug messages to make sure that this crucial math is correct.

The values for mu and gravity belong in a header file. **Your code should have no "magic" numbers – constants belong in #define or const variabls.** 0.0 is not considered a magic number.

## Graphics

How we use graphics will only change slightly. Drawing the balls will be a two part operation. Draw balls function will have to do the eb_clear and then ask the list to draw each ball. When that completes, Draw balls then calls eb_refresh and millisleep. Asking the list to draw each ball means calling iterate (see list section) and passing in a function pointer to code that will make an eb_ball call.

## Text output

See the various output files for formatting. **The balls must come out in ball number order.** Like Draw Balls, Print Balls will have two parts. One part prints the header and blank line. Between those operations it has the list print each ball. Like with graphics, this involves passing an action function to traverse.

You do not create node messages, the list machine does. You DO create ball allocation/free messages. Counts must all match.

Here are various startup and shutdown messages

```
DIAGNOSTIC: 1 balls allocated.
DIAGNOSTIC: 1 nodes allocated.
read_one: scanf returned -1

At time  0.0000:
Ball # 8 is at (12.500, 12.500) moving (  22.5000,    0.0000)


...

At time  5.8500:
Ball # 8 is at (78.343, 12.500) moving (   0.0000,    0.0000)

Simulation ends at sim elapsed time of  5.8500 seconds.
DIAGNOSTIC: 1 balls freed.
DIAGNOSTIC: 1 nodes freed.
Total run time is 0.000636101 seconds.
```

Here is some of the tabular output.

```
At time  0.0000:
Ball # 0 is at (10.500, 10.500) moving (-12.000, -12.000)
Ball # 1 is at (50.000,  6.500) moving (  0.000, -20.000)
```

```
Ball # 2 is at (94.500,  5.500) moving ( 20.500, -20.500)
Ball # 3 is at (94.500, 44.500) moving ( 30.500,  30.500)
Ball # 4 is at (50.000, 39.000) moving (  0.000,  35.000)
Ball # 5 is at (12.000, 38.000) moving (-50.000,  50.000)
Ball # 6 is at (25.000, 25.000) moving (-18.000,   0.000)
Ball # 7 is at (27.000, 26.250) moving (  0.000,  21.500)
Ball # 8 is at (27.000, 23.750) moving (  0.000, -21.500)
Ball # 9 is at (75.000, 25.000) moving ( 24.000,   0.000)
Ball #10 is at (73.000, 26.250) moving (  0.000,  32.000)
Ball #11 is at (73.000, 23.750) moving (  0.000, -32.000)
Ball #12 is at (95.000, 40.000) moving ( 12.500,  25.000)
Ball #13 is at (95.000, 40.000) moving ( 12.500,  25.000)
Ball #14 is at (95.000, 40.000) moving ( 12.500,  25.000)
Ball #15 is at (95.000, 40.000) moving ( 12.500,  25.000)
```

## Diagnostics

Anytime a diagnostic message is called for, it will have DIAGNOSTIC: at the beginning of the message. Details of the messages will be in the full write-up marked by **diagnostic**. You do not have to worry about printing the node messages in lab 3, the supplied linked list code will print them. You do have to worry about the counts being right – see dynamic memory below.

```
DIAGNOSTIC: 1 balls allocated.
DIAGNOSTIC: 1 balls freed.
```

## Two Structures

In this lab, you should never need to pass a structure – doing so is probably going to create bugs and malfunctions. You *will* need to pass pointers to structures. You have to be careful about the order in which you list your two structures in the header file since one references the other.

### Ball Structure

Instead of passing 5 data items around everywhere, we will put all of the data for a ball in a structure. The declaration for that structure will be in a header file. Any file of C code that needs to deal with balls will include that header file.

```
#include "structs.h"
```

Our ball structures will carry all of the data in lab 2 as well as:

- a pointer back to the simulation structure this ball lives in (see below)

This pointer gives each ball access to simulation-wide data. For example, the action function move ball needs to know what dT is. Move ball gets passed a single ball that is in the list. (Move ball will be a function that you write).

*Suggested prototype*: Create and initialize a structure and then pass the address of the structure to a function that outputs the ball data from the structure. We will need to print numeric output similar to lab 2, so you can start with your lab 2 print code and turn it into lab 3 print code. This prototype gives you experience with structures and pointers to structures.

### Simulation Structure

Your code will also need a single instance of a structure for the simulation level variables. These include elapsed time, delta time, and a storage pointer. The storage pointer is a pointer to void used to hold the linked list that will hold the balls. Set it to NULL and let the linked list code manipulate its value. Sometimes you will pass this pointer and other times you will pass the address of this pointer since C is call by value – see comments in the list header below. Your code can also check to see if the pointer is NULL. Do no other operations on this pointer.

*Suggested prototype*: Declare such a structure, initialize it, and pass it via pointer to another function.

## Dynamic Memory

**Your code can have a single statically allocated ball structure in the function that reads in balls, but no other statically allocated ball structures**. If your code successfully reads a ball, it should get a dynamically allocated ball structure from `malloc()` or `callloc()` and copy the data to the dynamic space. You can pass the address of the static structure to a function that deals in dynamic memory, but other than that, the statically allocated structure must not be visible outside the function that owns it. Once you get the dynamic memory squared away, be sure to insert the ball on the list.

The messages below are only printed in text mode.

- Each time your code allocates a ball it must increment the count of allocations and print a **diagnostic** message with that count.
- The code that allocates must check allocation failures. If allocation fails, it must print an **error** message and return NULL.
- Each time your code frees a ball, it must increment a count of frees and print a **diagnostic** message with that count.

Code that allocates and frees ball structures **must be located in a separate C code file** that only deals in allocating and freeing balls.

*Suggested prototype*: Write a function to test using a static storage block scope variable. It would increment each time the function is called and print that count.

*Suggested prototypes*: The first prototype tests the allocation function and checks the return value for NULL. It might put data into the returned ball structure and print it. The second prototype tests the function that frees ball structures that came from the allocator.

Calls to deleteSome (see List Machine) will need a callback to free the ball structure for deleted balls. This happens for balls that go off the table into a pocket and at the very end when the remaining balls have all stopped moving.

## Multi-File Development

Your makefile is going to get a workout with this lab. At a minimum, you will write 3 C code files. More is fine if it helps. There will at least be a file that houses main, another for table-related code, and another for ball allocation and recycling. A potential fourth is all of the functions that will be fed as function pointer parameters to the linked list (such as action functions). Your makefile will need to have correct dependencies so that if you touch one C code file, other files to not recompile. Look again at

how .o files are made and how .o files are linked to become executables. Feel free to ask for and give help with makefiles. Unlike C code, makefiles can be shared. ***C code cannot be shared!*** Your executable will need to link the linked list code, which will be released soon.

The following is from my makefile. See the slide deck on makefiles if you do not understand how it works.

**DO NOT COPY/PASTE ANY OF THIS! WORD HAS ALTERED IT IN BAD WAYS. Copy by hand.**

```
#this line is designed to save you from zero points on your lab (same as lab1)
%.o:%.c *.h
    gcc -ansi -pedantic -Wimplicit-function-declaration -Wreturn-type -g -c $< -o $@

#example line to link lab 3 code.  –lm is math, others are familiar.
#you might have fewer files to link
lab3: lab3.o table.o dt.o functions.o io.o mem.o
    gcc -g -o $@ $^  -L. -l8ball -llinkedlist -lncurses –lm
```

## Dealing with the List Code ("The List Machine")

Your code will never iterate through the ball structures, it will ask the list to do so. Just call iterate and tell the list what action you want to happen to each ball. When you call deleteSOme, you can tell it how to determine what balls need to be deleted and tell it what to do with any ball it decides to delete.

The storage pointer in the simulation structure is how we hold a reference to the linked list that will hold the balls. Look at the header file for the linked list code shown below. Take a close look at the typedefs for function pointers. The actual .h file is on piazza.

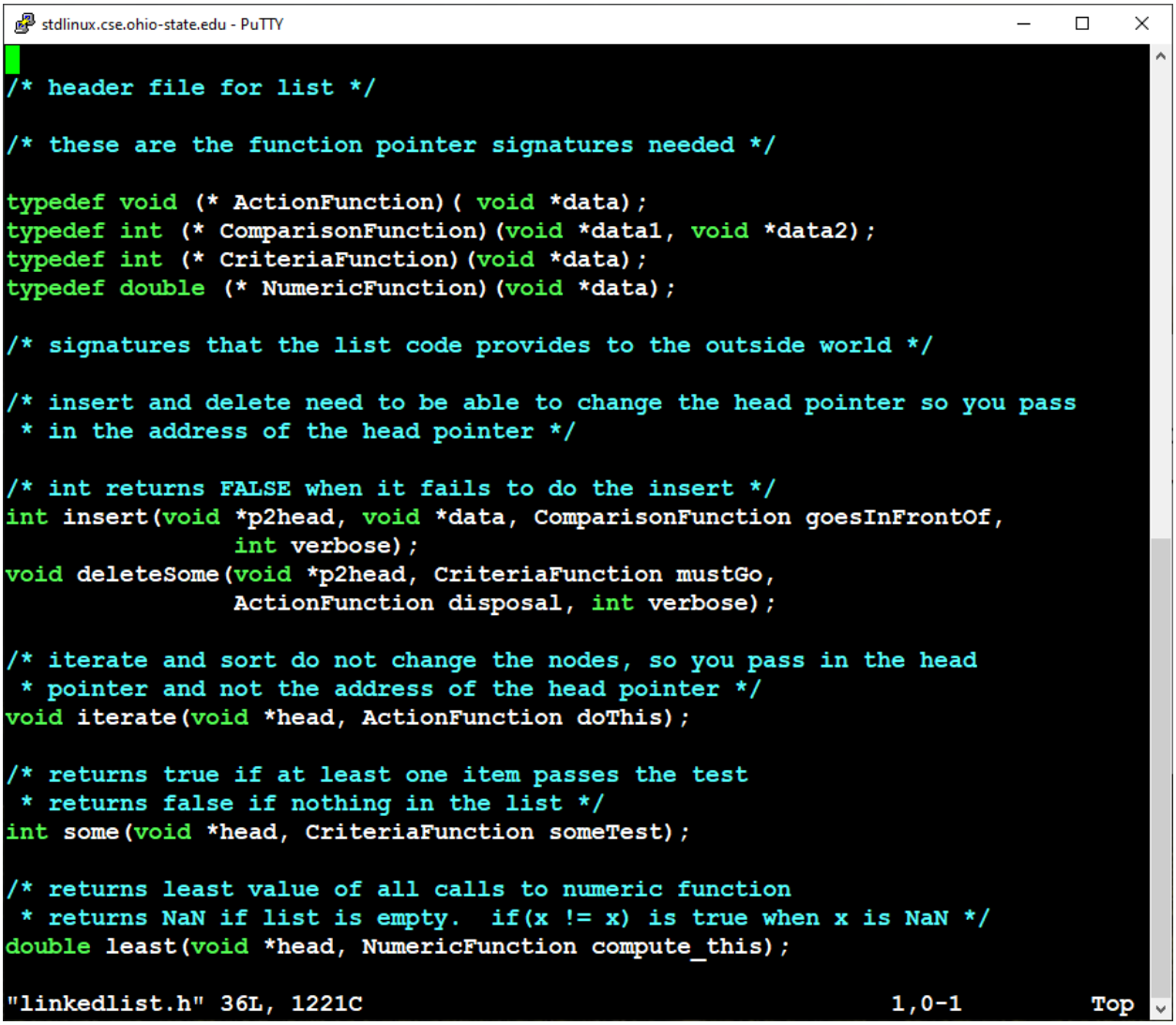
There are 5 functions that manipulate the list:

- insert – be sure to check its return value, needs a pointer to a pointer
- deleteSome – also needs a pointer to a pointer
- iterate – do some action to every data item
- some – do some of the items pass the test?
- least – run a numeric function on each ball and give back the lowest number returned

The first two need the address of the storage pointer and the last two need the value of the storage pointer. If you have a pointer to your simulation called "sim" and the simulation structure has a storage pointer called "storage" then insert and deleteSome want **`&sim->storage` as** the first parameter. Note the **&**. Likewise iterate and sort want `sim->storage` as their first parameter.

**Your code must check the return value of insert and respond if a FALSE value is returned.** The list code uses dynamic memory to allocate nodes. You can see this in the diagnostics listed above. If insert fails to allocate a node to hold the ball, it will return FALSE. Your code must then deal with the ball it just allocated that it now cannot store. It does so by recycling the dynamically allocated ball structure and continuing from there.

For insert and deleteSome, pass **TEXT** as the last parameter, not VERBOSE.




Note that all of the code is heavily based on pointers and function pointers. *Please note the comments regarding how the list itself is passed in.* A void pointer is assignment compatible with *ANY* type of pointer. The iterate and sort functions do not change the pointer passed in, so call by value works just fine and you pass in the storage pointer held by the simulation structure. The insert and deleteSome functions *DO* change the pointer used to access the list, so you pass in the address of the storage pointer in the simulation structure. All of those calls have pointer to void as the type. Think of the storage pointer as "pointer to a node" or as the "head pointer."

So the void pointer type can hold the value that comes from a pointer to a node or it could hold the value that comes from taking the address of a pointer to a node. The same type – void pointer – can hold either kind of pointer because void pointers are assignment compatible with *any* pointer type. We don't need a special type of void pointer to hold a pointer to a pointer.

*Why all of the void pointers?*

We use void pointers for the data so that the linked list code doesn't need to be compiled with structs.h. If the list code knew that it held ball structures, it would need to know what ball structures were. Using void pointers on the data side means the list code is generic.

Using void pointers for the data means the list has to use function pointers to do anything with the data. Comparison functions exist so that the list can know how to order the data. Criteria functions exist so that the list can make judgements about the data, such as whether or not to delete a node. Action functions exist so that the list code can do something to the data when visiting a node during a traversal. Our functions that print balls or draw them graphically will be invoked (possibly indirectly) as action functions.

We use void pointers for the head of the list so that the linkedList.h file doesn't get built into all of the lab code. The storage pointer in the simulation structure acts as the head pointer, and the storage structure is going to be known by nearly all of the application code. Your code in lab 3 doesn't need to know how the list works or if it is even a list at all. If the internals of the list change, your code does not have to change. It is possible that the library is using realloc() and an array, but your code can't tell the difference.

Are there downsides to all of those void pointers?

Yes. Your code has to know and be exact about type. Balls are data and they are passed in void pointers. The storage pointer will point to a node in the list. It is also passed as a void pointer. Mix them up and you get no warnings or errors from the compiler, only hard to find bugs. The list itself gets passed the address of the storage pointer sometimes and the value of the storage pointer the rest of the time. Mix them up and it will compile, but it will not work.

Prototypes come in a variety of flavors. First you test the typedef'd function pointer types. Then – much later – you test list functions.

*Suggested prototypes*: For each typedef'd function pointer type, write a function that fits the signature and call it via a function pointer. Those functions need pointers to some data, and *I suggest strings for testing.*

- An action function might take a void pointer, know it is actually a string, and print the string.
- A comparison function might take 2 strings and compare the first character in each string and return the result of that comparison.
- A criteria function might return isupper() on the first character of the string.

It is very easy to simply toss a string literal at this code as your data, possibly after coercing the type to a void pointer. Get all 3 function types working before going after the list itself. The test the list code using your functions from the prototypes.

*Suggested Prototypes*: Test insert, iterate, and deleteSome in that order. Then tackle some and least.

- Test insert using a comparison function based on the first character and insert one string that begins with an upper case letter. Then call iterate and pass an action function that prints the string.
- Copy that to a new prototype that does all of the above and then does another insert using a string with a lower case first letter. Copy the iterate call and call again after the second insert.
- Copy all of the above and then do a delete. Pass a criteria function that selects based on the case of the first character. Use your print action function as the disposal function so that you can tell what gets deleted. Then prove it via one more iterate call as before.
- For some, load the list with three things, print the list with iterate. Then call some with various criteria functions that prove it works.

- For least, pull up your earlier tests with strings. For your numeric function, use strlen to get a number and cast it to double.

Once you have proven that you can use the list code without seg faults, you are ready to use it.

Here is an example set of functions that might be passed to the list machine:

```
/* comparisons */
int numeric_order( void *lower, void *higher);

/* actions */
void action_motion(void *data);
void print_ball(void *data);
void dispose_ball(void *data);
void draw_ball(void *data);

/* criteria */
int always_true(void *data);
int are_moving(void *data);

/* numeric */
double frame_time(void *data);
```

## Submission

In general, the similar to lab 2. Change the 2's in the file names to 3's. [This text will be updated later]

### How to avoid getting a zero:

1. Be sure to download your submission out of Carmen and see that it builds in an empty folder.
2. Note: The following command must build your submitted lab without warnings or errors if you want credit.

   $`make lab3`

## Plan of Attack

The lab is certainly long, but not impossible.

- Prototypes are your friend
- Complexity is your enemy

In piazza the file p1.c has an example of a set of main functions for prototypes to test the individual parts of ball motion separately and then altogether. The functional layout section gives you a roadmap and list of functions that you might need. The low level one should be tested as prototypes.

Get structs and pointers to structs under control first. The motion control prototypes in p1.c show you how to get started with declaring a single struct and passing a struct pointer.

Motion control is next. You already have basic motion in lab 2 code. Port that to use structs. Then start on the rest of motion control (friction, reflection). Once that is done, you are ready to write an action function that calls all of them. You now have a working, tested action function that you will need to

feed the list when you start putting everything together. While you are here, do a prototype for on table. You can also do one for computing dT.

Dynamic memory comes next. Use prototype code from motion control to create a statically allocated ball (easier than using scanf). Ask you dynamic allocator “allocate ball” for some memory. Copy the static struct to the dynamic memory. Call dump ball and pass it the dynamic pointer to prove it holds the data. Test. Now add to that code. Call your code to recycle the dynamic memory. This tests the 2 main dynamic memory functions. Your real ball reader will want to allocate and the recycler will be needed when a ball is disposed of.

Now do the basic exercise of the linked list machine. These are given above. You can use strings as your data type because they are easy to work with. In your real code, it will be ball structs.

Once you have tested all of the subsystems, integrate.