Data in Lab 3 SP23

The main data in lab 3 will be stored in structs. Do **not** pass any struct by value.

Contents

Ball Data

For the ball data, we will stop using an array of doubles and use a struct instead. The struct needs numerous members:

- bits
- optionally it could have color
- X, Y, VX, VY, theta, and force
- A pointer to the simulation

Reusing two slots in the array in lab 2 was a questionable design decision – feel free to note that in your critique of lab 2 code. We certainly won't do that with structs because the members have unique names. Putting color in the struct once early on means that we won't have to make calls into bits.c in the output functions. It also means that we have two places that think they define the color of the ball.

Do **not** use an array of doubles for lab 3. Every single thing about a ball goes into the ball structure.

We need a pointer to the simulation so that if we get handed a ball, we can find out things about the simulation it lives in.

Your input code can have a statically allocated ball, but the rest of the code will be dealing with dynamically allocated balls. Use the one statically allocated ball as a place to read in one ball's worth of data. Then allocate a dynamically allocated ball and copy to that. We'll keep the balls on one of two linked lists.

Simulation Data

The sim needs a few fields that are not tied to any one ball:

- Elapsed time
- Score (an integer)
- A pointer that is the head of the in-play linked list
- A pointer that is the head of the off-table linked list

There will be exactly one simulation in our code and we will pass a pointer to it to the high level routines such as master_output. It carried everything else, directly (score) or indirectly (balls on the lists). Please do not dynamically allocate the simulation.

The two pointers are void pointers. The in-play list will be kept in **Y order**. Since balls on the table move, you will need to call sort before doing text output on the in-play list. The sort call uses the same comparison function that insert to that list uses.

The off-table list will be kept in **VY order**. (See output – VY is used to get the score for a ball. The faster a ball goes off the table, the more points it scores.) Since these balls never change, we don't need to call sort on the list; insert will put any new ball in the right place.

Interactions Between Structs and the Lists

The list stores incoming data in void pointers. The list does not know about ball structures. When the list wants to give a ball back to your code, it will pass the void pointer it holds. **Never cast a void pointer.** As soon as possible, assign that void pointer value to a strongly typed pointer. The list will call your action functions, comparison functions, and criteria functions and they will get passed void pointers.

Your code will have action functions that look like:

```
void draw_ball(void *data)
{
        struct Ball *bp = data;
          /* rest of function goes here.
          * Expect a lot of bp-> expressions */
}
```

All of your action functions should follow this template. Name your structures as you like, the list will only see void pointers. We know they are pointer to ball structures, so we assign them that way. That lets us use -> to get to the members.

The sim structure and the list

Each void pointer in the sim structure is a list head. **C is pass by value.** *We will never pass the pointer to the sim structure itself to the list.* We will be passing the list head to the list, either a copy of it or its address. Inside the list, those pointers are actually pointer to node but our code doesn't know what a node is because that's not our lab code's business. The list hides nodes from the application. Likewise, the application hides bits from the list. Neither side knows the internal data type of the other side, this is a truly generic list.

Functions that change the list heads

The **insert** and **deleteSome** functions both need to be able to *change* the head pointer, so we pass the **address** of the head pointer to those functions. The type we pass is pointer to pointer to void. The list uses a simple void pointer to receive that since *a void pointer can hold the value from any pointer type*. Inside the list internals, it will assign that void pointer to a strongly typed pointer to a pointer variable.

The point here is that you have to be able to take a pointer to a structure, use arrow to get a member of the structure, and then & to get the address of that member of that structure. Pass that address.

Functions the don't change the list heads

The **iterate** and **sort** functions do not add or remove nodes, so the head will not need to change. We pass a *copy* (pass by value is fine here) of the list head to iterate and the sort function. Again, the list code receives this in a void pointer and the list knows what strong type to use internally. That happens to be pointer to node, but our lab code doesn't know and shouldn't care.

So your code will take a struct pointer, use the arrow operator to get to a member of the struct and pass it without taking its address.

Header Files

Your structures must be declared in a header file. It will be #included by nearly every file. In your C code files, put the #include for this header above all #include that carry function declarations. It goes below any #include that use < > delimiters.

Plan of attack - structures

The finer details may take more than a few reads to fully understand. So start out with the members. You know what the members are and what types they need to be. Create the header file and write a prototype that uses it. You may already have a prototype that does this.