Contents

Version 2

Changes/additions:

1. If malloc/calloc returns NULL, your code in memory.c **must** print an error message if the code is in text mode. Here is an example of such a message.

   ```
   ERROR: allocate_thing: failed to allocate 24 bytes.
   ```

2. The insert function returns a bool. Your code needs to check that return value. If insert returns false, it did not put your pointer on the list. (This usually means that malloc failed when insert tried to allocate a node.) This means that your program will leak memory if you do not free the dynamic data that you were trying to put on the list. Insert will print an error message, so your code doesn't have to. Your code is already supposed to print a diagnostic message when it frees memory, so it will be clear what happened.

Dates

Pretotypes due Thu 19-Sep-24

Early by Tue 24-Sep-24

On Time by Friday 27-Sep-24

Late until Sat 28-Sep-24

Words in **bold** are likely to be mentioned in the rubric as something that gets graded.

Introduction

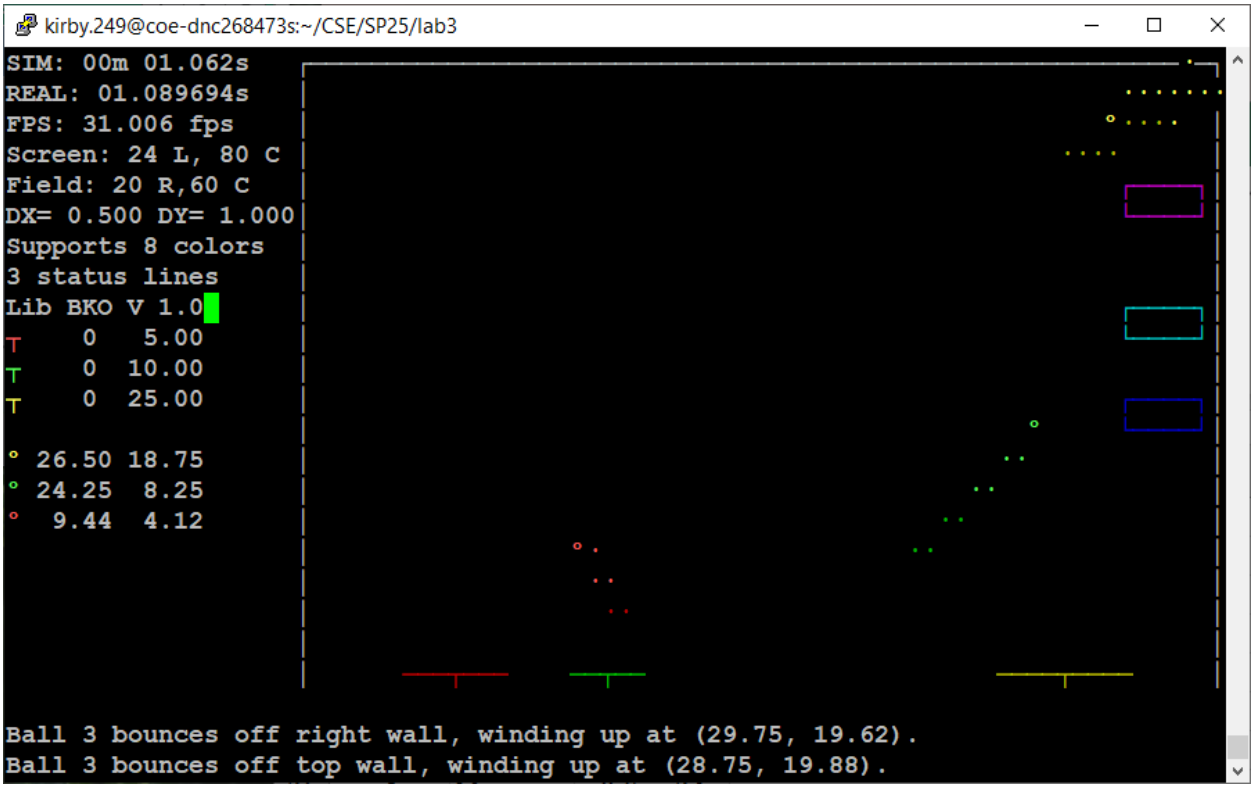
Lab three involves the following:

- Large program development
- Dynamic memory
- Pointers
- Structs
- Integrating with the linked list library
- Void pointers and anonymous data types
- Use of the static keyword on functions and variables

While the basic simulation is the same, lab 3 involves having more things to keep track of. Notably there will be multiple balls, blocks, and paddles in the game. We will use two linked lists and an array to keep track of them. Portions of our lab 2 code will have to change to accommodate these changes.

Added functionality includes scoring. When a ball hits a block, the block will be removed and points will be awarded. The amount of points awarded is the block color * 10. The points are awarded to the paddle that has the same color as the ball.

Functions that are localized minions will need to be marked with the **static** keyword. (See the output section, but this applies to sim and input most notably.)



Design by Comment

(Some functions are left out but should be familiar from lab 2)

Main: if we init, do everything and teardown. Time how long it takes and return what happened.

Do everything: create an empty sim structure, see if good input can fill it successfully and if so run the sim. Whether or not we run the sim, clear the lists when we are done. Tell main if we had good or bad input.

Run simulation: Set the clock to zero and run the sim as long as we have balls on the field. Sort them, output everything, update the clock, update the balls, and delete any balls that left the field. When the sim ends, do final output.

Clear lists: delete any remaining balls and print how many were removed. Delete any remaining blocks and print how many were removed.

Update balls: iterate the ball list, updating each one.

Update ball: do basic motion, delete any blocks we hit, hit walls, hit paddles.

Hit walls: given a ball, adjust for each of the walls.

Hit paddles: given a ball, loop the paddles and check to see if it hit the paddle.

Hit paddle: given a ball and a paddle, see if it hit and if so adjust the ball and give a bounce message.

Deleting blocks we hit: use deleteSome on blocks, deleting if hit_block says to do so and being sure to free the block in that case.

Hit_block: if this ball is inside this block, work the collision cases and then score points for the correct paddle. Return true if we hit.

Various output functions – change so that they iterate the ball or block lists. So print_blocks checks to see if there are blocks on the list and if so it prints the heading and then calls iterate on the block list, passing print_a_block as the action function. If no blocks are on the list, it prints that there are no blocks in play. Print_a_block knows how to print a single block. Likewise for printing balls. Likewise for drawing balls and blocks.

Structs & Pointers

You will need many structures for this lab. Most of them are found in the structs.h file given in the lab 2.5 code used in the critique homework. These are for the balls, blocks, and paddles. You will need to add an additional one, the simulation structure, to hold everything.

The sim structure will hold:

- The elapsed time (a double)
- The head pointer for the linked list that holds blocks
- The head pointer for the linked list that holds balls
- An array of 8 paddle structures

The number of paddles in the array needs to be a #define manifest constant that lives in structs.h with the rest of the structs. Everything in the sim structure needs to be initialized to zero before use. Do **not** use dynamic memory to allocate the sim structure or for the paddle array.

You will need to update the ball structure so that it also hold a pointer to the sim structure. This allows functions that are only passed a ball to have access to the sim structure. Be sure to set that pointer in your input routine since it is not a value that will be read. You won't be able to delete blocks if the ball structure doesn't have a valid sim pointer.

The individual balls, blocks, and paddles will be stored in structs, not arrays. Do **not** pass actual structs in lab 3, only pointers to structs.

Dynamic Memory

Both balls and blocks are allocated dynamically. They can be allocated by the same function if you like. They also need to be freed. All functions that call malloc, calloc, or free must be located in a separate file, such as memory.c.

Diagnostic & Debug Messages

All allocations and frees must print diagnostic messages when in TEXT mode. Your memory.c file is allowed to test TEXT for permission to print the diagnostics. Additional printing is required in DEBUG mode. See the examples below:

```
DIAGNOSTIC: allocate_thing: 48 bytes allocated for object #1
DEBUG: allocate_thing: returning 0x9922b0

DIAGNOSTIC: allocate_thing: 24 bytes allocated for object #6
DEBUG: allocate_thing: returning 0x993480

DEBUG: free_thing: freeing 0x199c880
DIAGNOSTIC: free_thing: 6 objects freed
```

The above output shows the diagnostic message for allocating a ball and a block and then later freeing what could be either a ball or a block. The allocation function diagnostic message needs to print how many bytes it allocated and the number of successful allocations it has made so far, including this one. The allocation debug message needs to print the pointer value returned, which is very valuable when debugging. Recall that these are the only allowable pointer values that can be given back via a free call.

The code that frees has to print a diagnostic showing how many things have been freed so far including this one. The debug message shows the pointer that is about to be freed *prior to actually calling free.* When your code is in debug mode, it should call **`fflush(stdout);`** directly after it does the debug print of that pointer value to be freed. This makes sure that the printf output makes it to the terminal before the potentially fatal call to free with a bad pointer value.

To keep track of how many allocations or frees have been made consider the following line of code

```
static int objects = 0;
```

This line is inside the allocation function and another just like it goes in the function that calls free.. Being static, the int retains its value for the life of the program. When the program loads, it gets set to zero, not each time the function is entered. If allocation succeeds, increment the static int and print what happened. The int will stay incremented until the next call when you increment it again. Have the same line in the function that calls free and use it to track the number of objects freed. (Technically we don't have objects in C, but structs represent objects).

General Dynamic Memory Requirements

Your code **must** free all memory that it allocates. (When the sim completes, you will need to clear off any remaining balls and blocks.) There should not be any balls left, but clear that list anyways.

The dynamic allocation code will need to use malloc or calloc to create space for the structure that holds the object. Such code will **always** check the return value to make sure that space was allocated. It will return a pointer to the allocated space and that pointer itself might be NULL so the code that asked for allocation will need to check that it got back usable space.

If allocation fails, the code must not crash. The error recovery strategy we will use is that we will be unable to do anything with the object we read in, but we will go back to reading as usual. Our code won't attempt to change the situation. It will attempt to go on with the next object if there is one and it will attempt to run the sim if any objects made it into the game. Real-world production code needs error handling strategies better than simply crashing. Our method is a hand wave to the idea that we are doing error recovery. As simple as it may be, code that does not crash always beats code that sometimes crashes.

The Linked List Allocations and Diagnostics

The list allocates and frees node structures. In TEXT mode it also generates diagnostic messages with the counts. All of our dynamic objects (balls and blocks) will consume 1 node while they are on the lists.

```
DIAGNOSTIC: allocate_thing: 48 bytes allocated for object #2
DEBUG: allocate_thing: returning 0x8f7320
DIAGNOSTIC: 2 nodes allocated.

DEBUG: free_thing: freeing 0x63b780
DIAGNOSTIC: free_thing: 6 objects freed
DIAGNOSTIC: 6 nodes freed.
```

You do not have to write any code to generate linked list diagnostic messages, they are built in to the library code of insert and deleteSome. See the bold messages in the output just above. This output comes from x2.bkf which has 3 balls and 3 blocks. The first messages show the reading of the second ball and subsequent insert into the linked list. The second group shows the freeing of the last ball after the other balls have already left the field and all of the blocks have been removed by collisions.

Using the Linked List Library

Your code will manage 2 linked lists, one for balls and one for blocks. (Paddles will be kept in an array.)

Your sim structure keeps the head pointers. The sim structure uses void pointers for this because the lab 3 code does not know the internal data type used by the list. *Even though node.h is in the zip, you may **not** use it in lab 3!* The supplied library implements the list operations for you in lab 3.

The list provides 4 operations that we use:

- insert (to add an object to a list)
- deleteSome (to delete zero or more objects from a list)
- iterate (to do something with every object, one at a time)
- sort (a list using some comparison)

Details on these are in their own sections.

Head Pointer Operations

Your code may do the following to the head pointer that the sim structure holds:

- Set it to NULL (do this once when you initialize the sim structure)
- Compare it to NULL (a NULL value indicates an empty list)
- Pass it to a list function such as iterate that doesn't need to change them (iterate, count, and sort)
- Pass its address to list functions that do need to change them such as insert and deleteSome

Your lab 3 code is **not** allowed to manipulate the head pointer in any other way.

Data on the list

The list will hold any kind of data that you can get the address of. Your code hands a pointer to your data to the list. The list stores that pointer in a void pointer because the list doesn't know what kind of data you want to keep on the list. You can put strings on the list. You can put structs on the list. You should never put different kinds of data on one list at the same time. Your code should always know what data type is on a particular list. In lab 3, that will be pointer to block structures or pointer to ball structures. In your prototypes, I suggest using strings; they are natively pointer based, they declare without use of malloc, and they are easy to print.

Callback functions

Your code will give the list callback functions. Those callback functions will receive one or more void pointers. Your code **cannot** cast the void pointers, it must assign them to a pointer of the correct type. An example of this is given in the section *Interactions Between Structs and the Lists.*

The callbacks below are described in terms of blocks or balls. For prototypes, you can have a different set of callbacks that work with strings. Note that you will *never* need to declare any variable to be of type ActionFunction even though you could. You can pass the name of any function fitting that signature to the list as your action function.

Do **not** declare any variable of the typedef'd types:

- **`ActionFunction`**
- **`ComparisonFunction`**
- **`CriteriaFunction`**

The list is NOT re-entrant safe! For example, your code calls iterate. The iterate function calls your action function "foo." The foo function must never call any list function on the *same* list. In lab 3, when we collect blocks we start by calling iterate on the *ball* list. The action function called with each ball then deletes from the *block* list, so we are ok.

Action Functions

```
typedef void (* ActionFunction)( void *data);
```

Action functions are used to do things to the data you stored on the list. The action function is given a pointer to an object structure. An example action function would be one that draws something. Another use of an action function is to deal with the data held by a node that is about to be deleted. In that case, it should be freed. (In other scenarios it would be OK to insert it onto a different list.)

Comparison Function

```
typedef bool (* ComparisonFunction)(void *data1, void *data2);
```

A common comparison function takes two pointers to object and returns a Boolean indicating true when the first object belongs earlier in the list than the second object. You will write comparison functions and pass their address to parameters of this type. Insert needs two of these to put new objects into the lists in the proper order. Sort needs them as well.

Criteria Functions

```
typedef bool (* CriteriaFunction)(void *data, void *helper);
```

A criteria function is used to give back a Boolean value. The data value will be a pointer to an object structure. The "always true" criteria function will ignore its parameters. Use "always true" when clearing a list at the end. Other criteria functions will do things like decide if ball is colliding with a block. These more complex criteria functions evaluate objects with the help of some other data item given in the helper parameter. You will need a function that expects a block as the data and a ball as the helper. If the ball has collided with the block, the ball data needs to be fixed, points scored, and the function returns true so that the block is deleted. (We would prefer not to do that much in a criteria function, but it is the only place where both the ball and the block are together.)

Library functions

The following functions are provided in the linked list library. See the zip fine on piazza. The header file is in the zip as well. Do not implement these in lab 3.

Insert

```
/* int returns FALSE when it fails to do the insert */
bool insert(void *p2head, void *data, ComparisonFunction goesInFrontOf, int text);
```

Because of call-by-value, the first parameter is the address of the head pointer of the list you want to insert the alien on. The data parameter will be a pointer to an object. When inserting into the ball list, use a comparison function that uses Y values. The text parameter will always be passed TEXT.

deleteSome

```
int deleteSome(void *p2head, CriteriaFunction mustGo, void *helper,
               ActionFunction disposal, int text);
```

This function is used to remove nodes from a linked list. It gets passed the address of a head pointer since it may need to alter that pointer. The criteria function passed yields true if the object needs to be removed. The helper pointer will get passed to the criteria function to help it make a decision about each object. The text parameter will always be passed TEXT.

One use of deleteSome is to remove a block that has been hit by a ball. The criteria function decides if block needs to leave the list. If so, the action function will be called to dispose of the block. Other uses of deleteSome are to clear a list at the end.

deleteSome returns the total number of nodes deleted.

Iterate

```
void iterate(void *head, ActionFunction doThis);
```

Iterate walks through the list, passing each data item on the list in turn to the action function. In lab 3 code, the action function does whatever it does to the structure and then goes on to the next node in the list. (In prototypes, the action function does whatever it does to the data type on the list, typically strings.) Since iterate does not cause any nodes to enter or leave the list, the first parameter is a head pointer *not the address of the head pointer like insert and deleteSome need.*

sort

```
void sort(void *hptr, ComparisonFunction cf);
```

Sort does what you expect using the comparison function that you give. It bubble sorts the list comparing adjacent items using the comparison function. Sort moves data around, but it leaves the links alone. Since sort does not cause any nodes to move, enter, or leave the list, the first parameter is a head pointer *not the address of the head pointer like insert and deleteSome need.* We will need to sort the ball list just prior to calling master output.

Other Functions

There are other functions in the linked list library, but we won't use them.

Building with the list library

Your makefile might need something like this:

```
LIBS=-lbko -llinkedlist -l:libncurses.so.5 -lm
```

Prototypes and the list

The list really doesn't care what type it holds, as long as your code coughs up a pointer to that data. The easiest way to do prototypes with the list – strongly suggested – is to use strings. They are pointer friendly and print easily and recognizably. Consider this code:

```
void *list = NULL;
rval = insert ( &list, “Go Bucks!”, first_letter, 1);
```

Here list is a pointer to void, first_letter is a comparison function that compares the first character of two strings and returns the result of numerically comparing them. Then call

```
iterate(list, print_string);
```

Here print_string is an action function that prints a string (pointer to char) that came in as a pointer to void. It is all of two lines long; a definition of a char pointer and a printf statement.

Don’t bother with structures when prototyping the list. Make sure that you can use all of the required list functions. Start with insert and use iterate to test it. Then do a few insertions and delete one thing from the list and use iterate to print what’s left. You might insert stirrings that start with both upper and lower case characters and you delete strings that start with a lower case letter. You could test sort by using a comparison function that compares the strlen of the two strings and then using iterate to print the newly sorted list.

Interactions Between Structs and the Lists

The list stores incoming data in void pointers. The list does not know about object structures. When the list wants to give a object 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 looks like:

```
void draw_a_block(void *data)
{
        struct Block *block = data;
          /* rest of function goes here.
          * Expect to see block-> 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 block or 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 a 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, any** 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, count 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.

Input

In lab 3, we read and keep all of the balls, block, and paddles. Your read_balls function will do inserts on the ball linked list. Your read_blocks function will do inserts on the blocks linked list. Your read_paddles function will populate data in the paddle array (held by the sim structure).

You might want to read into non-dynamic actual structs that are declared in each of the three main read routines. So you would declare a ball structure variable, have scanf fill it, and then do a struct – struct assignment to the dynamic memory once that gets successfully allocated. It's OK if you have functions that add a ball or block to their list in input.c. The ten line limit strongly suggests that the act of reading a ball is different than the act of adding a ball, as does the difference in what those two are about.

If your input functions find bad input they will return a false value to the caller as before.

All input functions except good_input (or your equivalent) will be marked **static** so that they cannot be called from outside of the input.c file. You will need to order them in the file so that they are defined before they are called.

Ball Input

The ball data will go on the ball list. The head of the ball list is held by the sim structure. It is best to use a locally declared ball structure as the place where scanf places the data it reads. If all is right with the ball, allocate dynamic memory for the ball, copy the local struct data to the dynamic memory, and

attempt to insert the dynamic memory on the ball list. If allocation fails, continue the input loop. Our code will do its best to cope with failures. If insert fails, free the dynamic memory and continue the loop.

Ball data is sorted by Y value (biggest Y value first in the list).

Block Input

This is done similarly to balls.

Blocks are sorted by Y value primarily. If the two blocks have different Y values, the lowest block is first. If the two blocks have the same value for Y, the block with the lowest X value is first.

Paddle Input

Paddles get some new validations.

- Paddle size may not be negative
- Color may not be less than or equal to zero
- Color may not be greater than 7

Be sure to check for this in the function that reads paddle data, much the same way previous code validated VY. You will want to output a bad paddle message and return false.

If the paddle data is valid, overwrite the correct element of the paddle array with the new data. Use the color number as the subscript in the paddle array. (This is also how points will be added to the correct paddle; use the ball color as the subscript.)

Output

Before calling master_output, the sim loop should sort the ball list in ball (Y) order. Blocks don't move, so they stay in the order they were inserted in even after some of them leave. **All** output functions, even ones you are given, must use const on all incoming pointers. Retrofitting this will cause numerous function headers to change. Start with master_output and final_output and then the compiler will help you find all of the functions that they call that also need const. Then do this in the message functions.

Paddles are only output if they have a non-zero color.

The fragment of the text output for x2.bkf is given below, which shows the two sort orders.

```
Elapsed time:  0.00000
Balls:
Ball 3 is at (25.00000, 17.00000) with velocity ( 8.00000,  4.00000).
Ball 2 is at (20.00000,  4.00000) with velocity ( 4.00000,  4.00000).
Ball 1 is at (10.50000,  2.00000) with velocity (-1.00000,  2.00000).
```

```
Blocks:
Block 4 is at (27.00000,  8.00000).
Block 6 is at (27.00000, 11.00000).
Block 5 is at (27.00000, 15.00000).
Paddles:
Paddle 1 is at  5.00000,  size 1.50 with score 0.
Paddle 2 is at 10.00000,  size 1.00 with score 0.
Paddle 3 is at 25.00000,  size 2.00 with score 0.
```

The text below shows a fragment of xAll.debug output which makes block sorting clear. The blocks in the row at Y=12 come before the block with Y=14. Ties in the row of blocks at Y=12 are broken by the X values.

```
Blocks:
Block 4 is at ( 0.00000, 12.00000).
Block 4 is at ( 3.00000, 12.00000).
Block 4 is at ( 6.00000, 12.00000).
Block 4 is at ( 9.00000, 12.00000).
Block 4 is at (12.00000, 12.00000).
Block 4 is at (15.00000, 12.00000).
Block 4 is at (18.00000, 12.00000).
Block 4 is at (21.00000, 12.00000).
Block 4 is at (24.00000, 12.00000).
Block 4 is at (27.00000, 12.00000).
Block 5 is at ( 0.00000, 14.00000).
```

Iterating

Some of the output functions will need to use iterate since there is more than one block and more than one ball. So print_balls will use an iterate call to to print all of the balls, passing in an action function that knows how to print a single ball. You will need to refactor that output this way.

Paddles are iterated with a for loop. If a paddle has color value of zero, that means it is not being used and won't be printed.

Parameters

Parameters will change throughout the output system compared to lab 2. Many high level calls will be passed the sim structure. Certain low level functions will get turned into action functions. Your function that draws a single ball will take a void pointer and swiftly turn it into a ball struct pointer and draw with that. All high level functions (notably those that take the pointer to the sim structure) must use const. Callbacks will *not* have const void pointers.

Final Output

Here is the final output for x2.debug showing that two of the paddles scored points. Also shown at the end are the results of clearing the lists.

```
Elapsed time: 25.78125
```

```
No Balls in play
No Blocks in play
Paddles:
Paddle 1 is at  5.00000,  size 1.50 with score 0.
Paddle 2 is at 10.00000,  size 1.00 with score 100.
Paddle 3 is at 25.00000,  size 2.00 with score 50.

removed 0 balls
removed 0 blocks
Returning 0
Total runtime is 0.004687071 seconds
```

Note that when there are no blocks or no balls, a different header is printed. The paddle header is always printed since they don't go away.

File Considerations

If you want, you can combine text, graphics, and output into a unified output.c file. This is not mandatory. You must use the **static** keyword on all output functions unless they are called from outside the output system. So master_output will not be static, but master_draw would be if that function is in the same file as the function that call it. A draw_balls function that would be called by master_draw clearly must be static. You will need to order your file so that the static functions come before the functions that call them since static functions don't appear in the header file. The lab 2.5 code has an integrated output system (output, graphics, text all in one file), so most of them will be static. There are plenty of examples of how static gets used in the supplied code. The lab 2.5 output.h shows the only functions that will not carry the static keyword: the three messages , master_output, and final_output. All other functions in output.c carry the static keyword.

Submission

Effectively: Same as prior labs. Your zip file needs to contain README_LAB3, **all** the code to be graded, and a makefile sufficient to build the lab. Do not include any .o files other than n2.o or the lab3 executable. Any file you edited by hand (other than test data) probably needs to be included. Aside from header files, files generated by the compiler should not be included.

All files that you edit by hand must have your name in them.

README_LAB3 text:

```
THIS IS THE README FILE FOR LAB 3.

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.

No Global Variables! Code Must Compile!

Global variables is a -10 penalty. Errors or warnings in compilation means no credit for the lab, though you might get 2 points if your prototypes are there and compile cleanly.

Zip files and makefiles

Your makefile rules that build your zip file need to self-test the lab (build the lab) or the lab is **late** regardless of when it was turned in.

Consider this fragment of a makefile that should be familiar to you:

```
#self-test what I want graded
make –C install –r lab3
```

The above lines are an example.