Contents

Version 0.4

Added the required functions section.

V0.2 OR not and in the bit manipulations

V0.3 0x3F in the description of setting the bits; it 3 shifted and the bits flipped, used to clear the old status bits before setting the new bits.

V.04: Updating the bits section has ~~strikethrough~~ on text for things removed from the lab. We only need 2 functions, one to set the status to in play and another to give back color.

Data in Lab 2

There are two important types of data in lab 2, the floating point data and the bit field stored in an unsigned char.

Floating point data

Creating and accessing the array

Use an array of doubles to store the 4 data items needed for a ball:

```
double ball[4];
```

Create some #define symbols to use in place of numeric subscripts:

```
ball[THETA] += 5.0;
ball[VY] += GRAVITY * DELTA_T;
```

For a ball on the launcher, you'd need **`X, Y, THETA, FORCE`** to have values 0-3.

We'll re-use those last two for a ball in play to mean something else, giving us symbols **`X, Y, VX, VY`** for values 0-3. That means VY and FORCE have the same value, namely 3.

Commentary on that bit of design

It's not the best idea on earth to re-use half of the array that way. It is expedient. You could create an array of 6 items if you wanted to. We do get around giving the same variable name different meanings. Properly done, we'd create a struct for the balls, but we won't have structs in time. Engineering is about tradeoffs. The pain point here are keeping track of whether the ball is on the launcher and we are talking about theta and force or the ball is in play and we are talking about VX and VY. In fact, upon launch, we use theta and force to compute vx and vy. Once we have vx and vy, we don't need theta and force.

Please don't iterate the array. Instead, access the elements individually as if they were separate variables with funny names.

Passing the Array

The array gets us around pass by value. We will pass the array name, which is a pointer, to functions that will get a copy of that pointer. When we subscript that copy of the pointer, we access the array elements. We don't get a copy of the elements, we got a copy of where the elements live in memory. This will give us code such as this function call where we pass the array:

```
update_position(ball);
```

And functions that get passed the array would look like:

```
void update_position(double ball[])
{
      ball[X] += //code you have to write goes here
```

In both cases, the underlined part is the important part of passing the array. If you change ball[0] in a function, you have changed it in the calling function as well.

The bitfield

What do the bits mean?

As part of the input, each ball has a value that is the result of encoding a number of very small values together and storing them in an unsigned char. Your code must be able to extract values from the bits in that unsigned char.

Assume that the 8 bits of the unsigned char are as follows:

STUV WXYZ

In this case Z is the low order bit and S is the most significant bit.

The bits ST, treated as a number, give the status of the ball:

0 is not used
1 is off the table
2 means the ball is in play
3 means the ball is on the launcher

The bits UVW, treated as a number, give the color of the ball. This will be a number ranging from 0-7.

Bits XWZ are not used in lab 2.

Required Functions

The simplifications to the lab mean that your bits code only needs to provide two functions:

- One that extracts the color and returns it as in int
- One that takes the bits and returns mostly the same bits with the status set to on the table

Please keep that in mind when reading about the bit manipulations below.

Extracting the bits

Extracting the bits involves shifting and masking. Shifting moves the bits a certain number of bit positions. Masking means bitwise anding the bits with a mask. The mask will have a certain number of 1 bits all in a row.

To get the status:

- Use a copy of the bits
- shift the copy to the right by 6 places
- bitwise and the copy with the value 0x03

To get the color:

- Use a copy of the bits
- shift the copy to the right by 3 places
- bitwise and the copy with the value 0x07

Separation of concerns

The amount we shift by the values we use to mask must be contained in one file and not made available to any of the rest of the code. Instead, a set of functions will be provided to extract values and to change values in the bitfield. Perhaps a source code file named bits.c would work. Only bits.c knows how to take the bitfield and deliver a color or a status value. Only bits.c knows how to take a bitfield and return a bitfield with the status bits modified. This file will need to provide 2 functions. ~~We need two extraction functions and three functions that set the status field.~~ The functions we need are one to give back the color and another that changes the status field to in play.

The magic numbers needed for shifting and masking need to be #define symbols placed at the top of the C code file. The magic numbers for status are 6 and 0x03. The magic numbers for color are 3 and 0x07. We don't put these numbers in a header file because we want to restrict them to being used only in one file.

Updating the bits

We need to be able to change the ball's status as it goes through the simulation. ~~When we read in the ball, we need to force the status to 3, meaning it is on the launcher.~~ To put the ball in play, we need to

force the status to 2. ~~When the ball goes off the table, we set the status to 1.~~ We need a function that will take a given set of bits and return an updated set of bits in order to put the ball in play.

To set the status we need to clear out the old status and then set the new status. The first 4 steps below get rid of the old status value. The last two steps put the new status value into the bit field.

To change the status:

1. Set an unsigned char to the mask value for status (0x03)
2. Shift that left by the shift amount for status (6 places)
3. Unary negate that to flip the 1 and 0 bits to get the value 0x3F
4. And that with the original bits, which clears the upper two bits and keep the value
5. Take the new value for status and shift it left by the shift amount (6 places)
6. OR that with the result from step 4 to get the new bits and return that

If bit manipulations don't make sense to you, you can get rid of the first three steps by setting a #define constant to 0x03F. The cost to doing this is that this value has to be kept in lock step with the shift amount and mask. (You have two things and one is the opposite of the other; change one and you have to change the other. The method above says given one, compute the other each time and that's always right even if things change.)

The file Masking.pdf is available in the lab2 area of Piazza for more information.