Exercise 3

It is important to realize that computers work with the binary number system, and similar to how it is easy for humans to multiply or divide by ten by shifting a decimal point, it is easy (fast) for computers to multiply or divide by two. In C, this is done by making use of the shift instructions:

```
int a, b, c;

a = 200;
b = a >> 3; // Shift a 3 bits right =>  divide by 2^3 => b = 200 / 8   = 25
c = a << 4; // Shift a 4 bits left => multiply by 2^4 => c = 200 * 16 = 3200
```

Note that this does not work for floating point numbers! In fact, many microcontrollers are not at all optimized for floating point operations which means that any operation involving `floats` or `doubles` take a considerable amount of time to process. Because of this there exist several methods to avoid the use of floating point operations. One of the more common ones is the use of fixed point math instead of floating point math.

As an example: say you want to divide 2 by 3. You can do this using integers if you instead divide 2000 by 3 resulting in 667 which should then be interpreted as 0.667. Now, you need to make sure you program interprets this correctly, which may seem a bit annoying, however, since this is many times faster than using floats, fixed point math is often used in timing critical operations. When working with 16 or 32-bit integers it is convenient to have 8 or 16 bits reserved for decimals, but you could in principle also use 4, 12, or how many bits you want for decimals. In the coming exercises we will be using both 16.16 and 2.14 (whole number.decimals) fixed point data types. If you want to know more about this topic, there is a reference on DTU Inside by Yates, 2007.

As hinted on earlier, to convert an integer to a fixed-point variable, you basically just multiply it by a constant. It is preferable to have the constant be a power of 2, since this allows you to do the multiplication using shift operations.

```
a = n1 * const;
b = n2 * const;
```

Addition and subtraction using fixed point numbers are straightforward:

```
c = a + b; // = n1 * const + n2 * const = (n1 + n2) * const
d = a - b; // = n1 * const - n2 * const = (n1 - n2) * const
```

You have to be more careful when doing multiplication and division since these give you a constant factor too much and too little, respectively:

```
e = a * b; // =  n1 * const   *  n2 * const   = (n1 * n2) * const^2
f = a / b; // = (n1 * const) / (n2 * const) = (n1 / n2)
```

A possible solution to this could be:

```
#define FIX14_SHIFT 14

#define FIX14_MULT(a, b) ( (a)*(b) >> FIX14_SHIFT )
#define FIX14_DIV(a, b)  ( ((a) << FIX14_SHIFT) / b )
```

Note that FIX14_MULT and FIX14_DIV are not functions but macros. When you compile your code any call to the macro is replaced by the macro itself, which means that using macros is faster than using functions, although it does result in your program taking up a bit more space.

A class of mathematical functions that is especially time consuming to calculate is the set of trigonometric functions (i.e. sin, cos, tan, ...). For these functions it is common to set up Look-Up Tables (LUT) which consist of pre-calculated values that your program can look up as required. This speeds up the computations significantly, but it comes at the expense of higher memory requirements. Therefore, you have to make a trade-off between accuracy, computational speed, and memory usage when you design a LUT.

In this exercise you will implement the cos and sin functions using LUTs. Sine and cosine have a number of properties that can be utilized to minimize the size of the LUT:

$$\cos(a) = \sin(a + 90°) \qquad (1)$$

$$\cos(-a) = \cos(a) \qquad (2)$$

$$\cos(a + 180°) = -\cos(a) \qquad (3)$$

$$\sin(-a) = -\sin(a) \qquad (4)$$

$$\sin(a + 180°) = -\sin(a) \qquad (5)$$

$$\sin(a) = \sin(a + n \cdot 360°) \qquad (6)$$

Equation (6) states that the sin and cos functions are repetitive, meaning that we only have to store 360 values if we want a resolution of 1°. Equations (1) means that we only need one LUT for both cos and sin which saves us 50% memory! You can save even more memory by using the remaining equations, but this comes at the expense of more computations and will therefore slow down the look-up process. In this exercise we will only be using equations (1) and (6).

Exercise 3.1 - Creating a Look-up Table

Creating a LUT from scratch can be quite tedious, but fortunately a guy named Jasper Vijn (aka Cearn) has been so kind as to make an Excel spreadsheet that will do it for you. It is even free to use!

- Download excellut from DTU Inside. Alternatively, you can get an updated version at http://www.coranac.com/projects/excellut/, but be aware that the syntax is quite different.
- Make sure you enable macros in Excel when you open the file.

- Use the following parameters to generate a sine look-up table:
  bytes/num: 2 (each constant will be 2 bytes)
  fixed point: 14 (use a 2.14 fixed point format)
  size: 512 (the LUT will contain 512 values)
- In cell I2 make sure to choose column c, and press Column Calc to calculate the LUT data.
- Remove the tick in cell E5 since we don't want to create a look-up table for division. Also, make sure that there is a tick in cell C5.
- Press Export to create .c and .h files containing the LUT.

You should now have a LUT containing the following information:

Index	**Value (hex)**	**Value (dec)**		
0	0x0000	0.0000	sin(0)	cos(3π/2)
1	0x00C9	0.0123		
.	.	.		
127	0x3FFF	0.9999		
128	0x4000	1.0000	sin(π/2)	cos(0)
129	0x3FFF	0.9999		
.	.	.		
255	0x00C9	0.0123		
256	0x0000	0.0000	sin(π)	cos(π/2)
257	0xFF37	-0.0123		
.	.	.		
383	0xC001	-0.9999		
384	0xC000	-1.0000	sin(3π/2)	cos(π)
385	0xC001	-0.9999		
.	.	.		
510	0xFE6E	-0.0245		
511	0xFF37	-0.0123		

The reason that the table should contain 512 constants rather than 360 is that some computations are much easier (and faster) when the number of constants is a power of 2.

One problem you will run into when using fixed point number is how to print them correctly. We have therefore provided the following function:

```
void printFix(int32_t i) {
// Prints a signed 16.16 fixed point number
    if ((i & 0x80000000) != 0) { // Handle negative numbers
        printf("-");
        i = ~i + 1;
    }
    printf("%ld.%04ld", i >> 16, 10000 * (uint32_t)(i & 0xFFFF) >> 16);
    // Print a maximum of 4 decimal digits to avoid overflow
}
```

The numbers in the sin LUT are in the 2.14 format and must be converted to 16.16 before you can use the above function to print them:

```
int32_t expand(int32_t i) {
// Converts an 18.14 fixed point number to 16.16
    return i << 2;
}
```

Exercise 3.2 - Sin and Cos Functions

Now that you have the LUT, it is time to create sin and cos functions that can utilize it!

- Create a function that calculates sin by using the LUT.

To simplify things, you should let the function operate on circles divided into 512 steps rather than 360 degrees. You have to implement the function such that the repetitive nature of sin is obtained (equation (6)) - without using any loops!

- Test the function by calculating:

	Expected Result
$\sin(0)$	0.0000
$\sin(45°) = \sin(45/360 * 512)$	0.7070
$\sin(-78°)$	-0.9783
$\sin(649°)$	-0.9456

Since we want to avoid multiplication and division as much as possible, you should calculate 45°/360*512 etc. on a calculator and write sin(64) in your program. As a programmer you are able to do your calculations in 512 steps per 360°, however, the situation is a little different if a user has to input an angle, but let's not worry about that for now. To keep yourself (and anyone looking at your code) sane, you should ALWAYS comment on which format you are using in your code! Example:

```
a = sin(64); // Calculate sin(45 degrees)
```

The next step is to create a cos function. Remember that cos(a) = sin(a + 90°), and that 90° = 90/360*512 = 128 = 2^7.

- Implement a cos function using the same LUT as before.
- Test the function by calculating:

	Expected Result
$\cos(0)$	1.0000
$\cos(45°) = \cos(45/360 * 512)$	0.7070
$\cos(-78°)$	0.2070
$\cos(649°)$	0.3253

Exercise 3.3 - Vector Rotations

Now that we have cos and sin implemented as ready to use functions it is time to apply them to something. How about manipulating vectors? Rotations of 2D vectors can be done using the following formulas:

$$x' = x \cdot \cos(\theta) - y \cdot \sin(\theta) \qquad (7)$$

$$y' = x \cdot \sin(\theta) + y \cdot \cos(\theta) \qquad (8)$$

But before we can rotate vectors, we need a way to store them in memory. For this we can use structures:

```
struct vector_t {
    int32_t x, y;
}
```

Unfortunately, a C function cannot return a structure, you will therefore need to store the output of a function in the same memory location as the input:

```
void initVector(struct vector_t v) {
    v.x  = 10;
    v.y  = 20;
}

...
initVector(vec); // Note that vec will NOT be changed by initVector()
...
```

If you try running this piece of code, you will find that it does not work at all! The parameter `v` is stored on the stack and will not be copied back to the caller. You will therefore have to use pointers to get this to work.

Note: for this exercise we recommend that you do all the calculations in 18.14 format instead of 16.16 as this reduces the risk of getting overflows when doing multiplications. But remember that you must convert everything back to 16.16 before you can print the numbers.

- Create a function that takes a vector and an angle (in 512 steps for a full circle, not 360°) and rotates the vector. You can call the function rotate(...).
- Test the function by calculating:

	Expected Result
`rotate`((1,2), 180°) = `rotate`((1,2),180/360*512)	(-1.0000, -2.0000)
`rotate`((6,4), -10°)	(6.5955, 2.9154)
`rotate`((-4,-4), 900°)	(4.0000, 4.0000)
`rotate`((-4,2), -35°)	(-2.1186, 3.9383)

Not it is time to do some more manual bit manipulation.

Exercise 3.4 - Bit Manipulation Questions Continued

This exercise is meat for training the use of fixed point arithmetic and it should therefore be done individually.

In the following assume 8.8 notation.

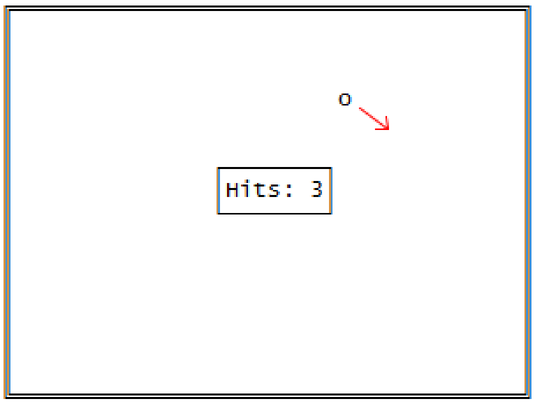
0x17 How do you convert a char into 8.8 notation? ____________________
0x18 How do you convert an 8.8 value into a char? ____________________
0x19 What is the smallest, positive, non-zero value you can store? ____________________

0x1A What happens if you add two 8.8 values? ____________________
0x1B What happens if you multiply two 8.8 values? ____________________
0x1C What happens if you multiply an 8.8 value with a char? ____________________

0x1D What about rounding when you convert an 8.8 value to a char? ____________________
0x1E How do you make it round correctly? ____________________

Exercise 4

In this exercise you will create a small project putting together everything that you've learned so far. You will create an animation of a ball bouncing around between a set of walls and counting how many times the ball strikes the edges.



Exercise 4.1 – Graphics

We will start by drawing the walls:

- Based on exercise 2.1, create a new function to draw the outer walls. You have to decide on a window size that fits your terminal environment.
- Use the same function to draw a box in the center of the window.

The next step is to create the ball. For this step you should start out discussing in your group how to tackle the problem and then divide the work between you, for example, having each group member write one of the necessary functions.

- Create a structure for the ball containing position and velocity. Note: remember to use fixed point variables and NOT floats.
  Create a function that updates the position of the ball based on its velocity. Hint: pointers are your friends! Additional hint: A simple update function could be (x = x + vx * k, y = y + vy * k).
- Create a function that draws a lowercase 'o' (or any symbol you prefer) at the ball's position.
- In your main loop use these functions to first clear the screen and draw the walls, then update the ball's position, and finally draw the ball.

When you run your code, depending on the size of your window, you will probably notice that it takes a good while to draw everything. This will not do! Let's try increasing the baud rate of the serial connection as this will allow for faster data transmission. A good starting point would be 115200 baud, just remember to change it in both PuTTY and your code. Now you should have a much smoother animation, however, you might run into a new challenge: the ball moving too quickly. To remedy this, you can add a counter and only run the update function when the counter is at certain values. Later, we will be dealing with this in a more elegant fashion.

Exercise 4.2 – Collisions

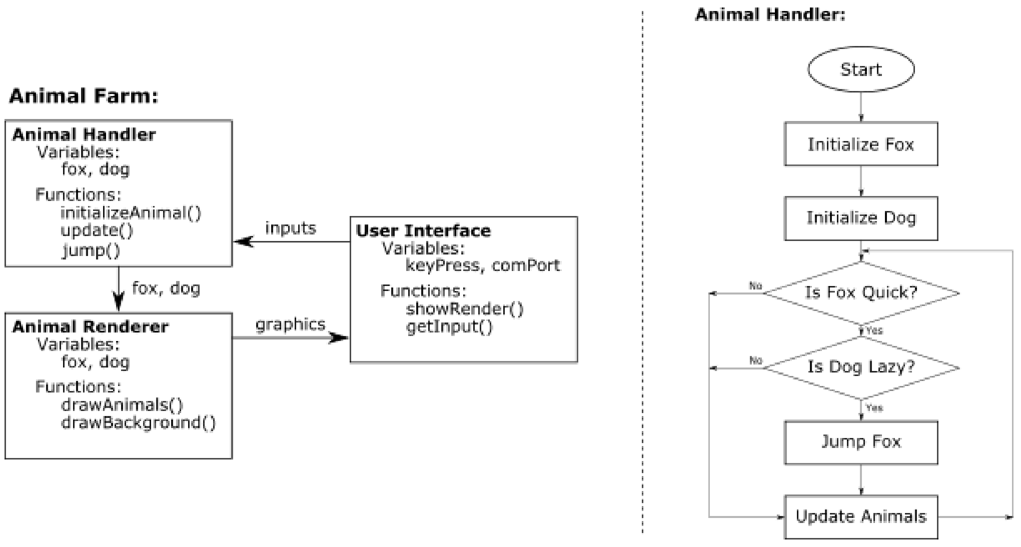
The next step is to handle collisions:

- Create a function that checks if the ball is overlapping any of the edges of the window.
- Whenever this is the case, the velocity vector of the ball should be reflected relative to the normal of the wall it hits.
- Create a counter that is incremented every time a collision occurs.
- Draw the collision counter in the center rectangle of the screen.

If everything went well, you should now have a procedurally generated animation running on your screen! But we are not quite done yet; when working on larger programming projects there is always documentation to be made.

Exercise 4.3 – Documentation

For this mini-project you will have to create a flowchart and a block diagram describing the program as well as descriptions of the functions you've made. The image below shows an example of how a block diagram for a program called Animal Farm and a flow chart for a subcomponent called Animal Handler may look:



Similarly, a function description for a function Jump may look like:

Jump	Jumps fox over dog
Syntax:	`void Jump(fox* f, dog* d);`
Parameters:	`f`: A fox. Should be quick, preferably brown.
	`d`: A dog. Should be lazy.

The function takes a fox and a dog and jumps the fox over the dog.
If the fox is not quick an error will be thrown.

- Create a block diagram for you program. It will be very simple.
- Create a flowchart of your main loop.
- Create function descriptions for the functions you've created.