LAB 04:
Structs, States, Text, Arrays, and Swap

Provided Files

- gba.c, gba.h
- duck.h
- font.c, font.h
- text.h
- print.c, print.h

Files to Add/Edit

- .vscode
  - tasks.json
  - settings.json
- main.c
- text.c
- duck.c

Instructions

In this lab, you will create a state machine, draw text to the screen, and swap duck structs.

Make sure to copy over your .vscode/tasks.json and settings.json from one of your previous assignments.

TODO 1

For the first part of this lab, we will implement a very basic state machine containing two states. Our states will begin as black backgrounds but will eventually contain more. The state machine for this assignment will have the following states:

- ➔ REST
  - ◆ Pressing START calls `goToAnimate()` and takes you to the ANIMATE state.
- ➔ ANIMATE
  - ◆ Pressing START calls `goToRest()` and takes you to the REST state.

The following TODOs will walk you through setting up this behavior.

TODO 1.0

- In the `main()` while loop, create the switch-case for the state machine.
    - The `state` variable that holds the current state and the enum containing the states have already been written for you.
    - You should have a case for each state. For now, each case will just contain a `break` statement. Soon, you will add your state transition behavior.

TODO 1.1

- At the top of `main.c`, make the function prototypes for `goToRest()` and `goToAnimate()`.

TODO 1.2

- Below the already-written `initialize` function, make the state transition functions for `REST` and `ANIMATE`.
    - These should be the two separate functions, `goToRest()` and `goToAnimate()`.
    - For now, each of these functions should just update your `state` variable to match the state you are transitioning to.

TODO 1.3

- Call your `goToRest()` function in the `initialize()` function so that when the game starts, the `REST` state is first.

TODO 1.4

- In the switch cases you made in TODO 1.0, transition to the other state if the user presses START.
    - Pressing START in the `REST` state should take you to the `ANIMATE` state.
    - Pressing START in the `ANIMATE` state should take you to the `REST` state.
    - To change states, you should call the **state transition function** for the state you want to change to.

Compile and run. You will not notice any new changes on the emulator screen, but you should fix any errors that prevent compilation.

TODO 2

It's time to add text so we can label each state.

TODO 2.0

- In `text.c`, complete the `drawChar()` and `drawString()` functions.
    - **`Hint:`** `drawString()` should make use of `drawChar()`, calling it

inside of a loop

TODO 2.1

- In `main.c`, update your *state transition* (`goTo...`) functions to draw text to the upper part of the screen (make sure it doesn't clip past boundaries, and that the color is visible!)
    - For `REST`, draw "Resting Ducks" to the screen in a color of your choice.
    - For `ANIMATE`, draw "Animated Ducks!" in a different color of your choice.

TODO 2.2

- Before your `drawString()` call in each state transition function, you should erase the area where previous text may have been.
    - You should call `drawRectangle()` and use an area that is just big enough to hide the text from the previous state. You will want to pass in the background color (`BLACK`).

Compile and run. You should be able to press START to toggle between your REST and ANIMATE states. Each state should be labeled accordingly and text from the previous state should not be visible. If this is not the case, fix your code before moving forward.

TODO 3

TODO 3.0

- In `duck.c`, initialize the `duckBitmaps` array. The array should contain the already defined bitmaps: `bitmap1`, `bitmap2`, `bitmap3`, `bitmap4`, and `bitmap5`.
    - `bitmap1`, `bitmap2`, etc. are arrays of integers defining how each duck is drawn. `duckBitmaps` is an array of integer pointers. It should hold the memory addresses of each of the bitmap arrays.

TODO 3.1

- In `duck.c`, find `drawDuck()` and complete the switch statement. In this switch statement, we are setting pixels to a particular color based on the retrieved value from the duck bitmap. Each duck bitmap has 6 possible values, 0, 1, 2, 3, 4, or 5. Create a case for values 1, 2, 3, 4, and 5. The 0 case can be handled by the default statement, as we will not set the pixel any color if the value of the bitmap is 0. For each case, you'll be setting a pixel at `x + i, y + j`.

1. For case 1, set the color of the pixel to dark gray.
    - `DARKGRAY` is a macro defined for you in `gba.h`.
2. For case 2, set the color of the pixel to the duck's `color` property.

- Make use of the **duck pointer** that is passed into the function, and access that duck struct's color member.
- You can use the arrow operator (`->`) to access members of struct *pointers* like this:
  `ptr->member` (returns the value)
  `ptr->member = value` (assigns a new value)

3. For case 3, set the color of the pixel to yellow.
   - `YELLOW` is a macro defined for you in `gba.h`.
4. For case 4, set the color of the pixel to black.
   - `BLACK` is a macro defined for you in `gba.h`.
5. For case 5, set the color of the pixel to orange.
   - `ORANGE` is a macro defined for you in `gba.h`.

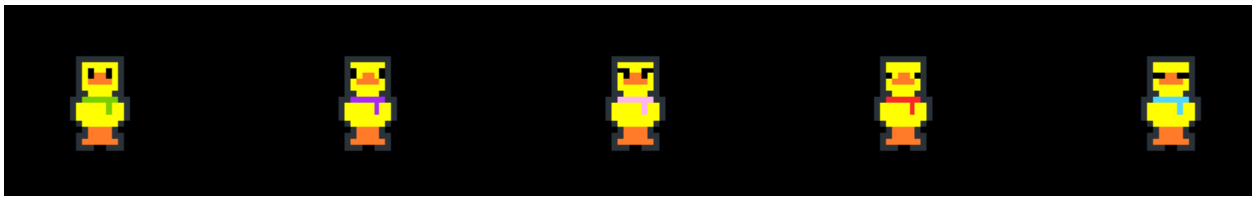
TODO 3.2

Now we need to update the state machine so we can actually draw the duck bitmaps. Each state should be updated with the following functionality (new changes are bolded):

- ➔ `REST`
  - ◆ Calls `waitForVBlank()` and `drawGame()` **in that order**
  - ◆ Pressing START calls `goToAnimate()` which takes you to the `ANIMATE state`
- ➔ `ANIMATE`
  - ◆ Calls `updateGame()`, `waitForVBlank()`, and `drawGame()` **in that order**
  - ◆ Pressing START calls `goToRest()` which takes you to the `REST state`

Add the appropriate function calls to each state.

Compile and run. In addition to your text, you should see the following in each state. If you do not, fix your code before moving forward. If your text overlaps with the duck bitmaps, you should fix this as well.



TODO 4

In our `ANIMATE` state, we want the ducks to constantly swap. You will implement the `swap()` and `reverseDucks()` functions to accomplish this.

TODO 4.0

- At the top of main.c, add the *function prototype* for a function called `swap()`.

TODO 4.1

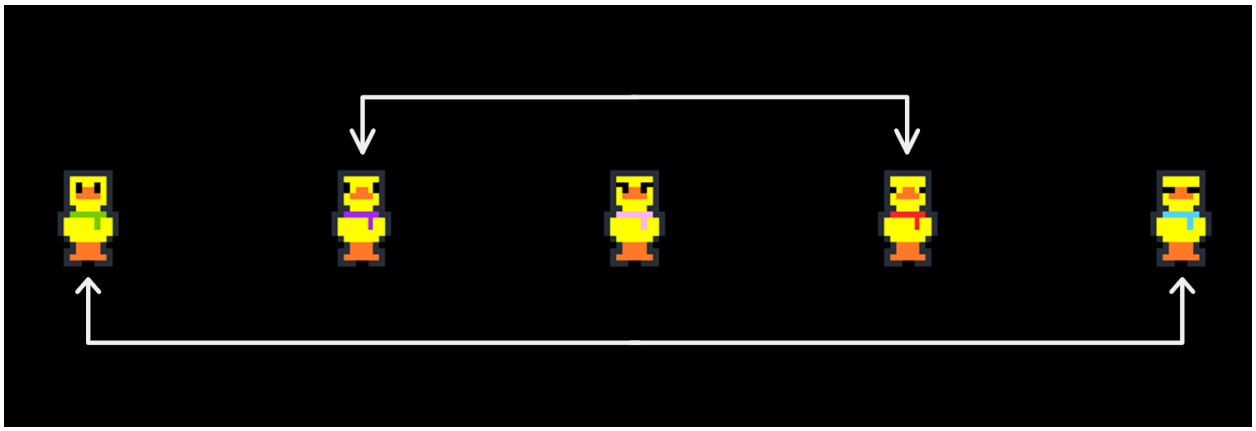
- At the bottom of `main.c`, write the `swap()` function which will ***swap two duck structs***. Refer to lecture material to get a refresher on how to implement swap.

TODO 4.2

- In `main.c`, implement the `reverseDucks()` function. To do this, you will write an ***in-place array reversal*** for the `ducks` array. *In-place* means that you may not use an additional array or data structure to reverse the contents of the array. This function should swap the entire array, not each index at a time. Thus, you will need to call the `swap()` method you wrote to perform the in-place array reversal. It should be able to swap any size array, not just the array of ducks.
    - **Hint**: you only need to iterate through half of the array to swap the elements.
    - **Hint**: there's no `array.length` in C! Make use of the `DUCKCOUNT` macro from `duck.h` instead.

Compile and run your lab. When you enter the `ANIMATE` state, the ducks should continuously reverse their order. If you return to the `REST` state, the ducks should stay still in the most recent order from the `ANIMATE` state.

When in the ANIMATE state, the ducks should be swapping like this:



Checklist

Ensure that:

- You can press START to toggle between the REST and ANIMATE states.
- Each state is labeled with the correct text label at the top of the screen.
- The ducks are drawn properly in each state.
- While in the REST state, the ducks remain stationary.
- While in the ANIMATE state, the ducks swap back and forth.

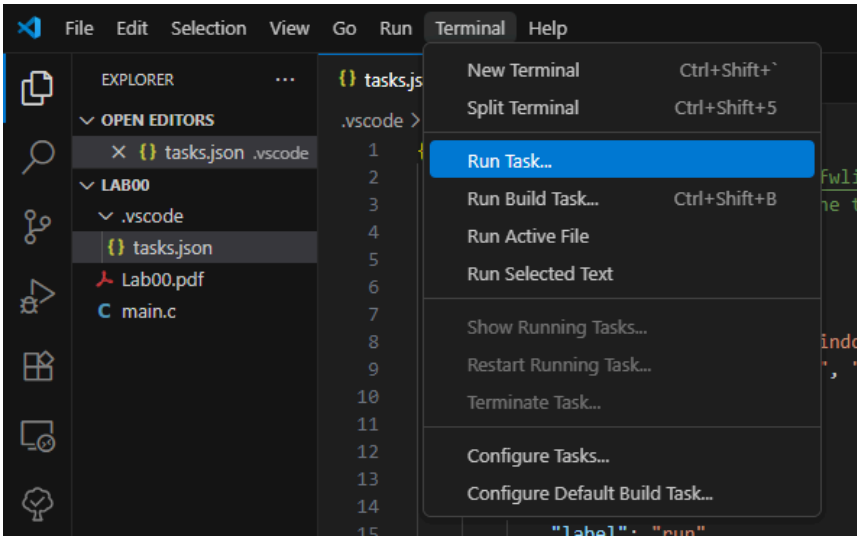
To see what a completed version of the project looks like, see `Example.gba`.

Tips

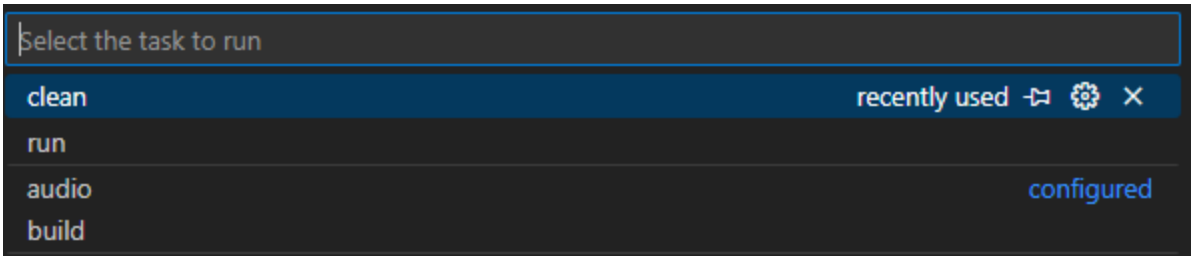
- Review lecture materials for how to implement swap.
- Draw out what your logic for `reverseDucks()` actually does. This will help you ensure you've implemented the code correctly!
- Follow each TODO in order, and only move forward if everything is correct.

How to Clean

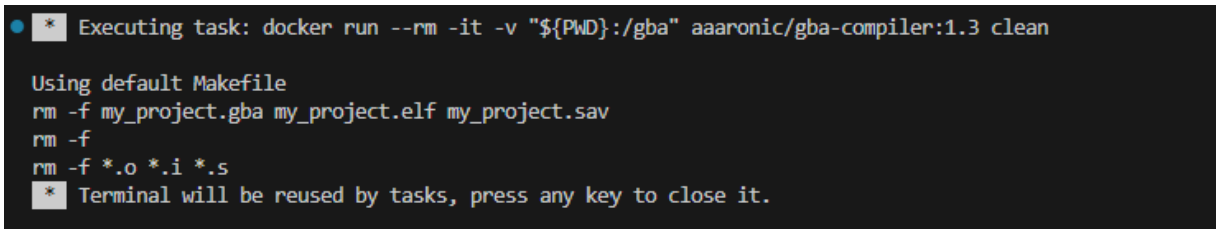
Navigate to the **Terminal** option at the top of your screen. Click on it, and then select **Run Task…**, as shown in the image below.



A dropdown menu will appear with the possible tasks to perform. Next, select **clean** from the options. Note that your options might be in a different order than listed here, but the task should be there.



After selecting clean, something similar to the following should appear in your terminal tab (bottom of the screen).



If so, success! You have cleaned. You can now do **cmd/ctrl + shift + B** to build and run.