CS2261 Media Device Architecture: Arrays, Structs, Struct Pointers, and Pooling

Announcements and Clarifications:

* HW2 due today
* Quiz next Monday
* Lab 4 on Thursday

Notes:

Arrays:

* Similar to Java, but lacks the .length feature or even an enforced size of array (you can index out of bouds)
  + There are no “objects” in C
  + Declared like:
    - int a[15]
      * Finds a spot in memory big enough for 15 integers and “zeros” it out (makes space)
  + Assigning
    - a[4] = 3;
      * a[4] works like pointer arithmetic. “a” is similar to a pointer and it points to the start of the array. Using brackets moves it 4 spaces in memory (varies based on the type. In this case, int). Moves to store the number 3 at index 4.
  + Declared and assigned
    - int b[] = {3,1,-2,5}
  + Indexed like
    - int c = a[7];
* Arrays in C are actually **const pointers** with preallocated memory after them
* **Const** is a keyword that means you can’t change the value after initialization (like final in Java)
* Ex. int \*a = (int \*) 0x6000000;

a = a + 1; // a will now point to 0x6000004, one int’s

length away

int b[10];

b = b + 1; // throws a compiler error

* When you create an array, enough memory is preallocated to hold all of it. However, there are no restrictions in C like in Java. Java gives you an IndexOutOfBounds Exception should you go too far, but C never stops you, so you end up going out of the allocated space and writing in places in memory where there could possibly be stored data.
* Ex. int a = 9;

int \*b = &a;

b[1] = 6; // You just wrote over…something

int c[5];

c[1] = 6; // You just wrote over the second array element.

* c[9] = 7; // You just wrote over…something
* Determining length to avoid going over or if you want to iterate through the array has two possible solutions:
  + Create a macro that you can use to set the size and that you can use as the upper bound when iterating through the array
    - Disadvantages include that the number of ways to declare the array are limited, and any changes have to manually updated throughout the program
  + Instead, use **sizeof** keyword
* **sizeof** – a keyword based in the processor, will return how many bytes something is
  + return how many BYTES allocated to array
    - int a[50]; // the sizeOf is 200, because ints take 4 byts, 4\*50 = 200 bytes total
    - int c = sizeof a[0]; //first element in the array, so 4 (one int)
    - int d = (sizeOf a); / / (sizeof c); //d will equal 50, this formula will always return the length of the array
  + useful for making code cross compatible
    - The size of variables is not set in stone in C (it is determined by the processor)
    - Since we know the GBA sizes, this point isn’t that important for us.

Structs

* No objects in C, so to organize related data (all the variables related to a thing, like the row or col), we use structs
* Not objects but like objects. Used for organization of related data
* Syntax:
  + Keyword name {
    - Fields (unlimited)
  + }; //don’t forget this semicolon
  + Struct ball {
    - int row;
    - int col;
    - Unsigned short color;
  + };
* Its recommended to put structs in a .h file, because structs are only declarations and not memory-altering code
* sizeOf – will give you entire size of the struct
  + for ball, you would get 10 (4 from the int row, 4 from the int col, and 2 from the unsigned short)
* similar to arrays
* declaring assigning:
  + struct ball a;
  + struct ball b;
    - type if struct ball, not “ball”
    - members are all initialized to 0
    - initializing:
      * a.row = 75
      * a.col = 120
      * a.color = RED
      * can also copy data
        + b = a;
        + data is copied, NOT ponting to sam reference. Changing b will NOT change a
        + be sure to declare b beforehand
* typedef lets you make an alias for a type
  + typedef currentypename newtypename
  + typedef unsigned short u16
    - u16 color = GREEN;
    - can still use original name too
  + only works in the file where you made the alias
    - best to include it in a .h file to make sure all files can access it when they include the .h file
* struct and typedef
  + typedef type {
    - fields
  + } NAME;
  + Example:
    - typedef struct {
      * int row;
      * int col;
      * Unsigned short color;
    - } BALL;
  + This is also acceptable:
    - typedef struct ball {
      * int row;
      * int col;
      * Unsigned short color;
    - } BALL;
      * Just means it can be called ball and BALL
* Anonymous structs are those without a name, and usually only useful for when you use them once
* Where to define structs:
  + Only define them in the file you are going to use (main.c not myLib.c)
* Struct Pointers
  + Struct functions
    - Often we will want a function to operate on a struct
    - On the slide, the function won’t do anything because your changes only exist in the function and also only exist on a copy of the data (not actual data), and as soon as the function ends, the changes end
    - In Java, objects are pointers. But C doesn’t have pointers so it won’t work on the actual data but a COPY of the data
* (\*a).row = a->row
  + Only use the -> pointer when you have a struct pointer
  + If you have the struct only, use the “.”
* Operator Precedence
  + (\*a).row += (\*a).rdel;
  + \*a.row += \*a.rdel;
    - This line doesn’t work.
    - The \* operator has a lower precedence than the . operator
    - So the second line tries to get the row and rdel of an address, then dereference those values.
  + **Solution: the -> operator**
  + Dereferences a struct pointer, and then gets a member
    - Ex. a->row += a->rdel;
* Bringing it all together
* void updateBall(BALL \*a) {
* if (a->row > 0 && a->row + a->height < SCREENHEIGHT) {
* a->row += a->rdel;
* }
* // etc.
* }
* Use it like so:
* BALL b;
* b.row = 10;
* // etc.
* updateBall(&b);
* Make sure you only use -> when you have a struct pointer
* If you have the struct itself, use .
* Pooling
  + What if you have multiple of the same thing, that behave the same?
  + Make an **array of structs**
    - Ex. The player can shoot bullets. They appear when fired, and disappear when they hit something.
  + Adding to and removing from an array is hard to code (and takes forever to compute)
  + The Solution: Pooling
    - Create an array of items just longer than the max number you will need at one time (called the “pool”).
    - Set all items in the pool to inactive.
    - When you an item to exist, find an inactive item in the pool, set it to active, and initialize its variables.
    - Every frame, update the active items in the pool.
    - you need an item to cease to exist, set it inactive.
  + Setting up the pool

typedef struct {

int row;

int col;

int cdel;

int rdel;

int oldRow;

int oldCol;

unsigned short color;

int active;

} BALL;

* Taking an item from the pool
  + When you need a new item, take the first inactive item in the pool.

if (BUTTON\_PRESSED(BUTTON\_A)) { // If we need a new ball

for (int i = 0; i < 20; i++) { // Loop through the pool

if (!balls[i].active) { // Find the first inactive

initBall(&balls[i]); // Initialize it somehow

balls[i].active = 1; // Set it active

break; // End the loop

}

}

}

* Using items in the active pool
  + Every frame, you want to update and draw the active items in the pool.
  + Adding items back to the inactive pool
  + When you don’t need an item any more, add it back to the pool of inactive items by setting it inactive.
  + Probably a part of the item’s update function
    - if (balls[i].row > SCREENHEIGHT) {
    - balls[i].active = 0;
    - }
  + If the item is still onscreen when this happens, you may need to erase it the next time draw() is called.
  + Useful to make a .erased member of the struct for this
  + Ex. if (!balls[i].active && !balls[i].erased)
  + // erase the ball, then balls[i].erased = 1;
* Size of the pool
  + If the pool is too small, you may need an item when no inactive items are available.
  + So nothing happens, and your player keeps firing but no bullets come out
  + If the pool is too big, you waste memory and computation time when looping through all of them.
  + If your pool is for items that move in one direction on the screen, you can calculate a good number
  + distance they travel / (frames between item spawn \* pixels moved per frame)
  + Useful for bullets, falling items, etc.