CS2261 Media Device Architecture

Announcements:

* Quiz next Monday, covering material between last quiz and Monday

Notes:

What is the register value if…

* You press all the buttons at once?
  + The register value will show all 0’s
* You press the down button then the right button?
  + The register bits corresponding to right and down will turn to 0
  + If you want to have an event occur when two buttons are pressed, be sure to change your code to reflect that the down button and the right button are being pressed using the (&&) operator
  + Switching their order does the same thing (down then right)

Programming multiple similar things (like lots of rectangles falling):

* It’s difficult to maintain all their variables and constantly be updating them.
* Use arrays and loops to keep track of them, to efficiently update their variables and positions, instead of having multiple objects with separate variables you manually must edit.
  + To accomplish this, there is a useful construct calls **structs**
  + Structs are like objects without methods (but not really objects because there are no objects in C)
  + There are several ways to declare them:

struct <name> {

<variable dec>

…

}

* Creating a struct like above will make a new type called ‘struct <name>’
* Examples:

struct moveObj {

int bRow;

int bCol;

int bOldRow;

int bOldCol;

int bRDel;

int bCDel;

int bHeight;

int bWidth;

u16 color;

}; //the name of this struct is moveObj, and that’s what is used for calling the variables inside

* Note that structs don’t necessarily need a name (anonymous structs):

typedef struct {

int bRow;

int bCol;

int bOldRow;

int bOldCol;

int bRDel;

int bCDel;

int bHeight;

int bWidth;

u16 color;

} ; //anonymous

* Use typedef to create aliases for structs (MOVEOBJ alias for struct moveObj):

typedef struct {

int bRow;

int bCol;

int bOldRow;

int bOldCol;

int bRDel;

int bCDel;

int bHeight;

int bWidth;

u16 color;

} MOVEOBJ;

struct movObj {

int bRow;

int bCol;

int bOldRow;

int bOldCol;

int bRDel;

int bCDel;

int bHeight;

int bWidth;

u16 color;

} rect;

* Below, rect is a struct of an anonymous type (bc no name after “struct”). Legal but discouraged:

struct {

int bRow;

int bCol;

int bOldRow;

int bOldCol;

int bRDel;

int bCDel;

int bHeight;

int bWidth;

u16 color;

} rect; //anon type

* Initializing structs can be done in main. Below, the variables are matched by order, and the variables not receiving a value are set to 0 :

int main() {

…

MOVOBJ rect = {ROWINIT, COLINIT, RDELINIT, CDELINIT, SIZE}

}

* Another way to initialize, access and use:
  + “.” + name
  + Example:
    - rect.row = INIT;
* If an unassigned int defaults to 0, then pointers will default to null (like in Java, how objects default to null)

Arrays:

* Contiguous collection of elements of the same type
  + In Java, polymorphism allowed for certain objects and their subclasses to considered essentially the same object and be entered in the same array
* C implementation is different from Java
  + In Java, arrays have a length aspect, but in C arrays don’t know anything about their size
  + Can go beyond allocated space
    - In Java, going beyond the allocated length would throw a IndexOutOfBoundsException
    - C doesn’t enforce this, allows you to go over, and this brings the risk of overwriting valuable data
* Array declaration:
  + <type> <name>[<size>]
  + Example: int x [20]
    - Compiler allocates space for 20 integers (80 bytes total from the 4 bytes per integer multiplied by the 20 ints)
      * (4\*20 = 80)
* What if…

int x[20]; //integer array

int \*p; //integer pointer

p = x; // is this legal?? What happens?

* + - Both x and p are pointers, and so this is legal.
  + x = p; //THIS is illegal though
    - not legal, because we would lose the array.
    - The compiler actually generates an address for x and allocates a certain amount of memory for the array. X stores the address of the array.
    - X is known as a constant pointer that cannot be changed after initialization, so x = p is illegal
  + p = x;
  + p++; // is this legal? Yes it is, Since p is an integer pointer and you are incrementing by one, the compiler moves over one integer, or 32 bits. Recall pointer arithmetic.

DEMO (ALSO UPLOADED)

Notes (ctnd.):

* int ia[6];
  + allocates consecutive spaces for 6 integers
  + how much space?
  + There is a method called **sizeof**(type) that returns what the size of the type on that particular computer
  + 6 \* sizeof(int)
* int ia[6];
* Note: ia === &ia[0]
  + Can reach a position in the array in a way similar to array indexing, but it’s not indexing. Instead you are using pointer arithmethic
    - ia[4]
      * multiples the size of the type by what’s in the brackets and adds it to the address, then dereferences, landing you in the 4h “index” of ia
        + sizeOf(int) \* 4 + iaMemLoc
* Array name points to the first in the array
* Pointers and arrays are NOT the same thing
  + Array names are constant pointers and unchangeable
* sizeOf
  + evaluated at compile time
  + two forms
    - sizeof object
    - sizeof (type name)