CS 2261: Media Device Architecture - Week 6

Reminders

- Quiz 2: Wednesday (bring Buzz card again)
- HW3: Due 9/27
- Quiz 3: Oct. 10th

Overview

- More on Structs
- Pooling
- DrawChar
- DMA

Structs

We can declare our own types!

Structs are a chunk of memory that can contain multiple different types (with named fields, even!).

As close to OO as you can get in C.

- No inheritance. No polymorphism.
- Weak Encapsulation. Some abstraction.

C Structs

Define one (sort of):

```
struct tag_name {
    type member1;
    type member2;
    /* declare as many members as desired, but the entire
structure size must be known to the compiler. */
};
```

C Structs

More useful definition:

```
typedef struct tag_name {
   type member1;
   type member2;
} StructAlias;
```

e.g.:

```
typedef struct student {
   unsigned char age;
   char name[128];//size must be known! max here is 127
} Student;

Now you can declare one either way:
struct student s1;
Student s2;
```

Struct Initialization

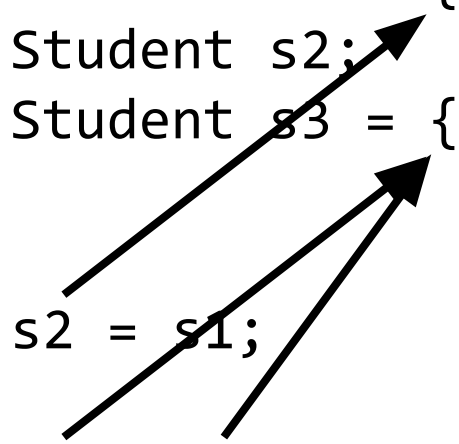
```
Student s1 = { 20, "George Burdell" };
Student s2 = { .age = 20, .name = "George Burdell" };
Student s3 = s1;


Struct s0; // what does this do?
```

Depends on where you are.

- Static, it would set everything to "zero" (NULL, '\0', 0, etc.).
- Dynamically, it isn't guaranteed to clear anything, just says "this space is yours for now.", with whatever was already in it.

Unlike arrays… You can assign them to other structs (copying ensues)

```
Student s1 = { 20, "George Burdell, Jr." };
Student s2;
Student s3 = { 57, "George Burdell, Sr." };
```

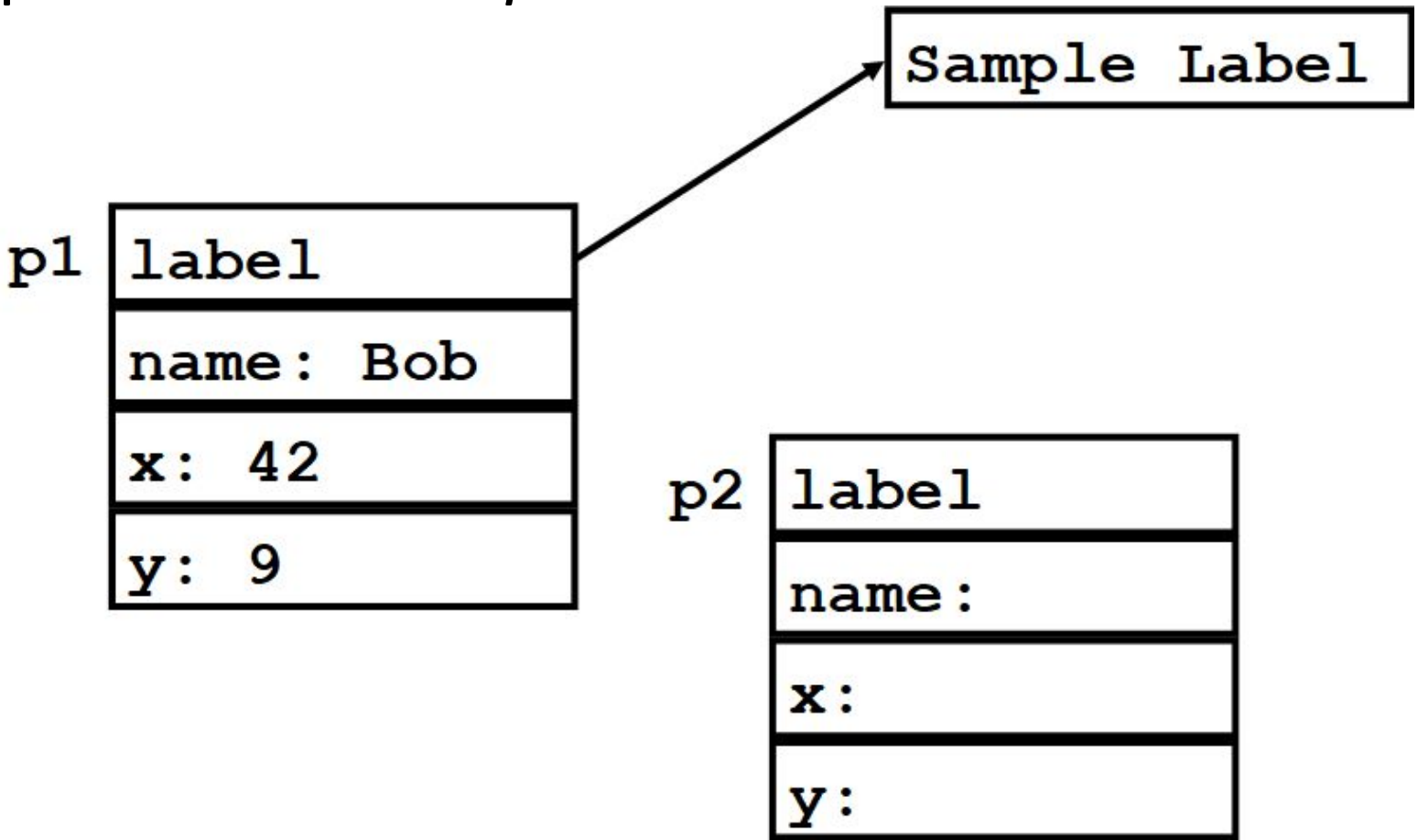


```
s2 = s1;

s1 = s3;
```

What's what now?

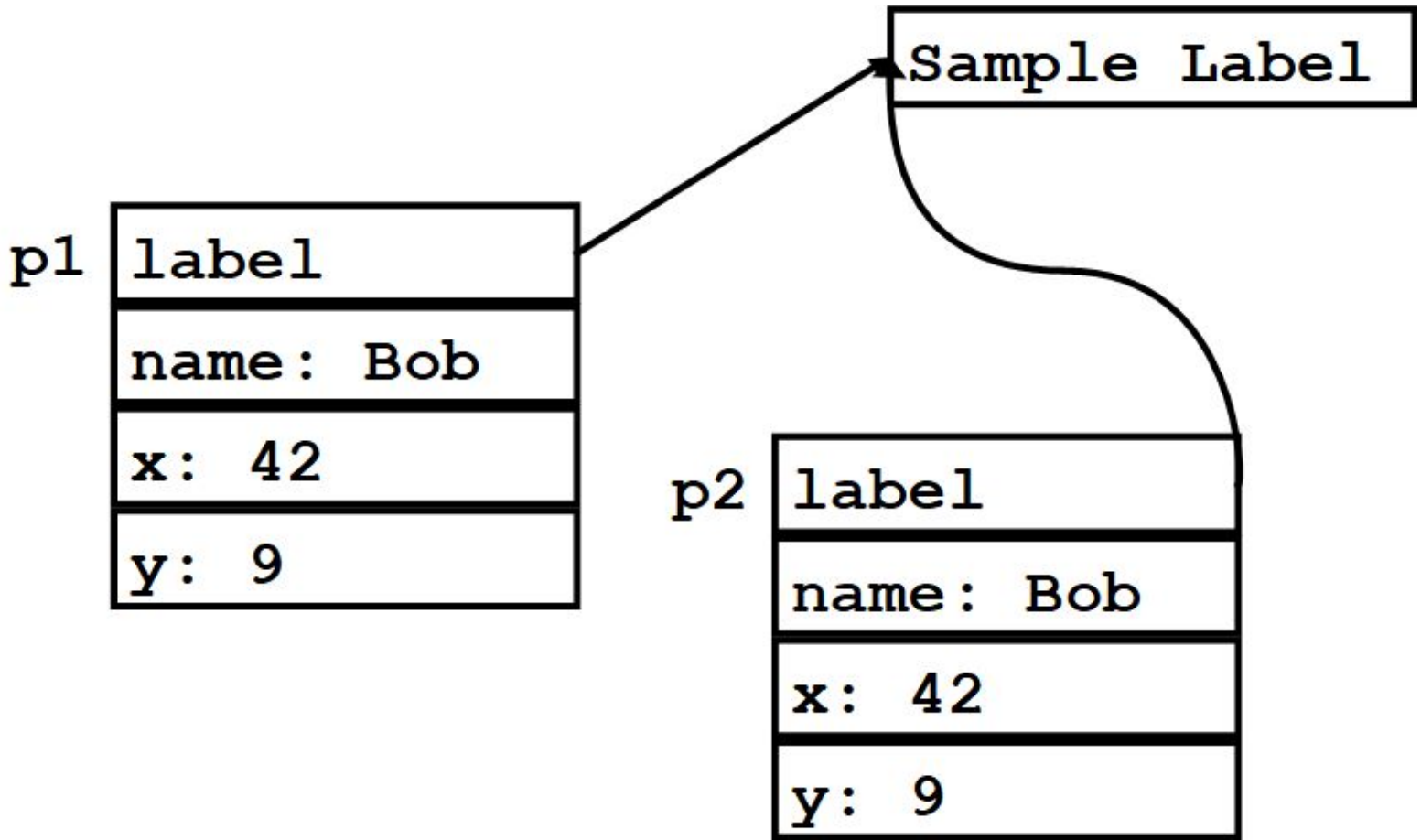
But be careful!

The pointers are copied to contain the same address, so copies are not *deep*.



But be careful!

p1 = p2;



Accessing Data Members

```
Student s1 = { 20, "George Burdell, Jr." };

s1.age;
s1.name;
```

Pointers to Structs

```
Student s1 = { 20, "George Burdell, Jr." };
Student *s1ptr = &s1;

Access via the arrow operator:
s1ptr->name;
s1ptr->age;


This is the same as (*s1ptr).name, (*s1ptr.age), etc.

Why not *s1ptr.name and *s1ptr.age?
```

Dereferencing

typedef struct { int a; } MyStruct;

- Assuming p is a pointer (`MyStruct *`)
- We now have 3 ways dereferencing p.

```
*p   // type: (MyStruct) simple dereference
p[4] /* type: (MyStruct)
        the MyStruct 4*sizeof(MyStruct) later in
        memory from the one p is pointing to
        (valid or not… could explode at runtime)
      */
p->a /* type: int
        dereference and access a within struct
        pointed to by p */
```

Of Note:

```
struct foo {
   int a;
   char b;
} alpha, beta;

struct {
   int a;
   char b;
} gamma;
alpha = beta;  /* Okay */
alpha = gamma;  /* Illegal! */
```

Also...

```
struct {  /* "foo" is missing */
   int a;
   char b;
} alpha, beta;

struct {
   int a;
   char b;
} gamma;
alpha = beta;  /* Okay */
alpha = gamma;  /* Illegal! */

// Not even casting will help:
alpha = (struct foo)gamma;   /* Illegal */
```

Sizeof?

```
typedef struct student {
    unsigned char age;
    char name[128];//size must be known! max here is 127
} Student;
sizeof(Student); // ???
```

What do we expect?

It's 129 (1 char, plus an array of 128 chars)

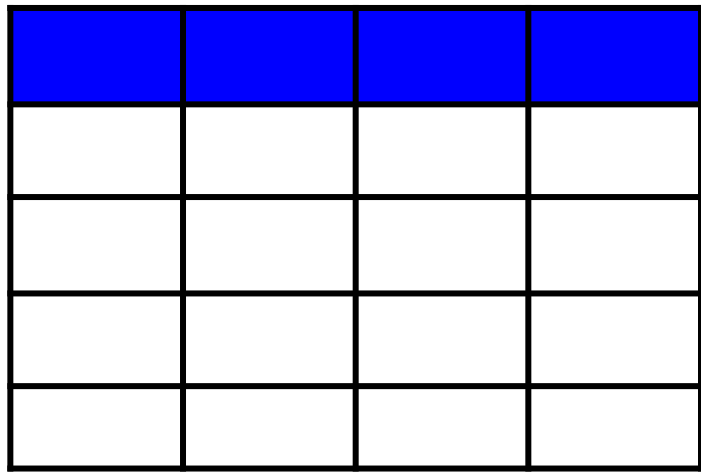
Sizeof?

What do we expect here?

```
typedef struct mystruct {
   int myint;
   char mychar;
   int myint2;
} MyStruct;

sizeof(int);  // 4
sizeof(char);  // 1
sizeof(MyStruct); // ?
```

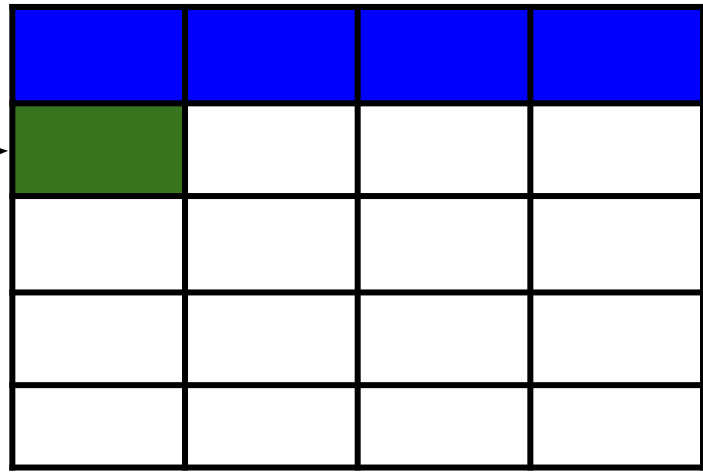
What Happens in Memory?

```
typedef struct mystruct {
   int myint;
   char mychar;
   int myint2;
} MyStruct;
```



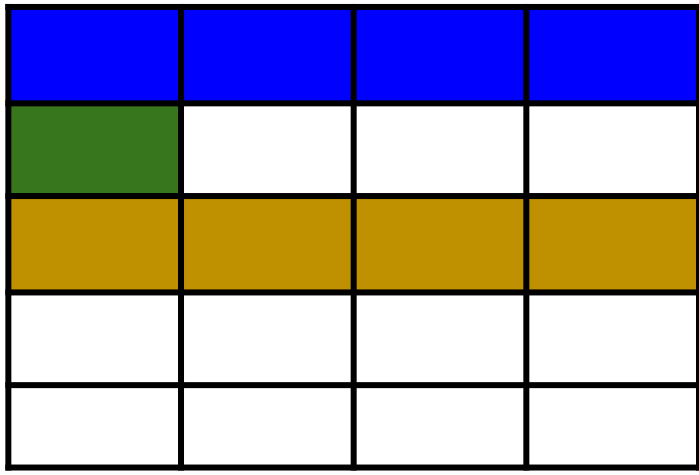
What Happens in Memory?

```
typedef struct mystruct {
   int myint;
   char mychar;
   int myint2;
} MyStruct;
```



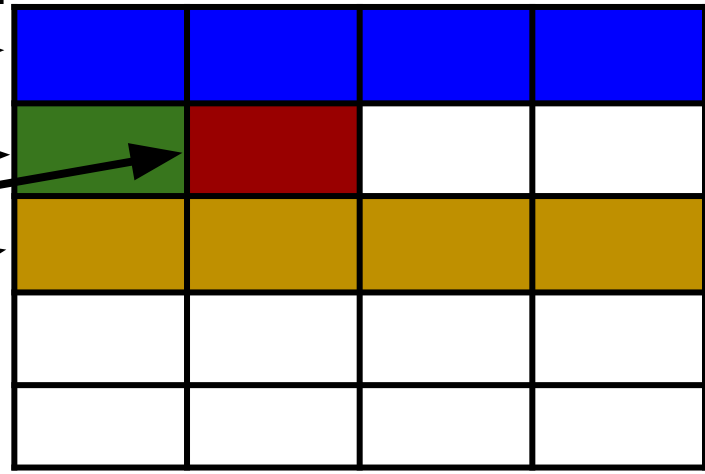
What Happens in Memory?

```
typedef struct mystruct {
   int myint;
   char mychar;
   int myint2;
} MyStruct;

sizeof(MyStruct);  // 12
```



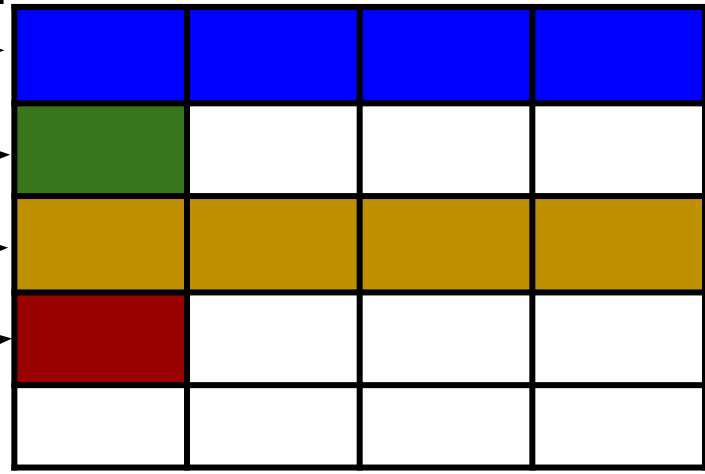
MyStruct2

```
typedef struct mystruct2 {
   int myint;
   char mychar;
   char mychar2;
   int myint2;
} MyStruct2;

sizeof(MyStruct2);  // still 12
```



MyStruct3

```
typedef struct mystruct3 {
   int myint;
   char mychar;
   int myint2;
   char mychar2;
} MyStruct3;

sizeof(MyStruct3);  // 16!?
```



Sizeof for Structs

- `sizeof(MyStruct)` = Σ(sizes of all elements) + ?
- Compilers often allocate more space due to alignment issues (space vs processor time).
  - Grabbing entire words on the word boundary is often much more efficient than individual bytes.
  - Interestingly, the compiler is not allowed to reorder the contents, even though it gets to choose alignment thereof.
- Takeaway: `sizeof(MyStruct)` is the only way to know how big MyStruct is in memory for the specific platform you've compiled to.

Not recursive

- Struct cannot contain another struct of the same type.
  - How big would on need to be if it could contain an entire copy of itself?!?!

- But it ***can*** contain a pointer to another struct of the same type.
  - This enables self-referential structures (like trees and linked lists).

Simple Linked-List

```
typedef struct {
   int data;
   ListNode *next;
} ListNode;
```

Doubly-Linked-List

```
typedef struct {
   int data;
   ListNode *prev;
   ListNode *next;
} ListNode;
```

Structs

- May:
  - be copied or assigned
  - have their address taken with &
    - Their members/properties can also have their addresses taken with &
  - have their members accessed
  - be passed as arguments to functions
  - be returned from functions

- May Not:
  - be compared (not even to itself).

Object Pooling

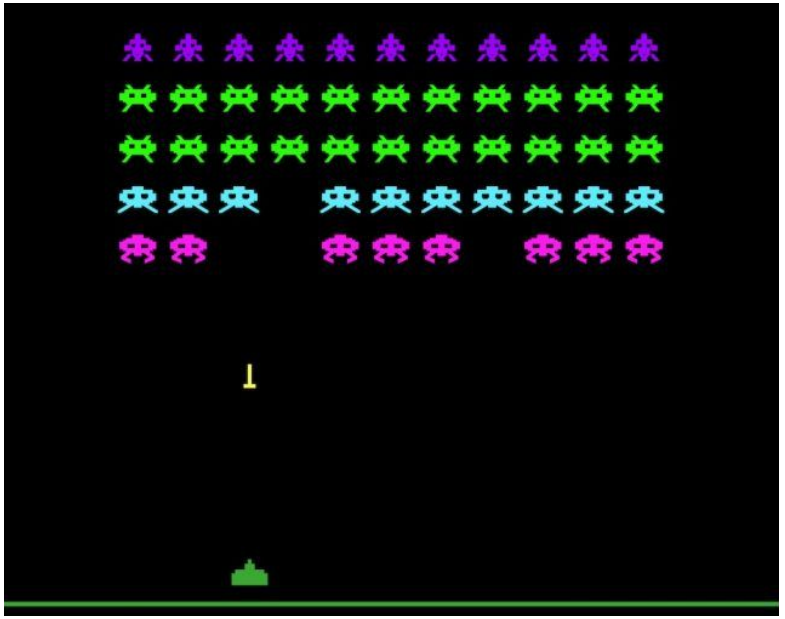
- Creating and destroying structs can be work-intensive (also, we'd need to be careful not to create too many -- as well as make sure we).
- A common technique for managing game objects, is to create all of them in advance (hello loading bars before levels…), then to loop over all of them and decide if they are active or not, and draw / react to them accordingly.
- This set of objects (for us, C structs), is referred to as the "pool".
  - You'll want one pool per type of thing (fireballs, goombas, koopas, bricks, etc.)

Simple C pooling

- Add an "active" property to the struct type you want a pool of.
- Create a static array of the type of that struct, with a length equal to the most of that kind of thing your game could ever need. (Bonus: as a static array, "active" will default to `0`.)
- When you would ordinarily create a new struct, instead find the first inactive one and initialize it (setting it to active, and setting **all** of its state to what you want it to be).
- All of your game logic should only with with active structs.
- When you're done with it, set it to inactive.

Applied (somewhat)

- Let's say I'm making Space Invaders
  - Only 1 ship. No need for a pool there.
  - I can only have one shot in motion at a time (also no pool).
  - There's a boatload of aliens on the screen!
    - Pool!
      I'm going to ignore that they look different for now.



Space Invaders Enemy Pool

```
#include <time.h>
#include <stdlib.h>
typedef struct {
  int x;
  int y;
  int prev_x;
  int prev_y;
  u16 color;
  int active;
} Enemy;

srand(time(NULL));
Enemy enemies[55]; // 5 rows of 11, in the original game;
Enemy *enemyToErase;  // keep track of the most recent hit to remove it

int main(){
  while(1) {
    initGame();  // set initial x/y positions for invaders and
                 // mark all of them active (I want to use all initially)
    updateGameState();
    waitForVBlank();
    drawGame();
  }
}
```

```
void updateGameState(){
  updateBullet();
  updateDefender();
  randomEnemyReadded();
  for(int i; i < 55; i++) {
    Enemy e = enemies[i];
    if (e.active) {
      checkForBulletCollision(&e);
      checkForReverseDirection(&e);
    }
    updateEnemy(&e); // need to keep updating
  }
}
void checkForBulletCollision(Enemy *e){
  e.active = 0;
  enemyToErase = &e; // need to erase me next draw.
}
void drawGame(){
  if (enemyToErase) { eraseEnemy(enemyToErase); }
  for(int i; i<55, i++){
    if (enemies[i].active) {
      drawEnemy(&e);
    }
  }
  // Draw bullet and defender here.
}
```

```
void randomEnemyReadded() {
  if (rand() < (RAND_MAX >> 10)){
    for(int i; i < 55; i++){
      if (!enemies[i].active){
        enemies[i].active = 1;
      }
    }
  }
}
```

Benefits

- Don't have to try to figure out how to add to / remove elements from a C array (no fun)
- Don't have to create objects on the stack and try to figure out how to keep them around (copying, mostly...)
- Finding the first inactive struct in the pool is often faster than creating an entirely new one.

Pooling pitfalls

- Remember to set the state on *everything* inside the struct when grabbing it from the pool (including setting it active). You wouldn't want it acting like it was the last time you were using it.

- 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.

More Advanced Pooling (Conceptually)

- Generally speaking, most object pools "manage" themselves.
- They usually have an interface for requesting an object from the pool, and for returning it back to the pool (which also removes the stale state).
  - The request can be O(1), as opposed to our current O(n).
- Such advanced pools can also dynamically expand to create more members, if the pool has no currently free members.
- These are mostly overkill for C.

Drawing Characters to the Screen (finally!!!)

- Up until now, we've been unable to print anything to the screen… Which makes debugging kind of difficult.
- We can't just printf
- We need something that tells us how to draw pixels for a particular character value.
- We need a font!

We need a font

- What if I could give you a 2-D array (let's say 8x6) with values 1 (draw a pixel) or 0 (don't draw a pixel), for each letter (stored as chars, for relative space efficiency)?
- Then your job would be to iterate through that array, drawing pixels according to those zeros and ones.
- This is the essence of a font (in this case, fixed-size), and it's what we provide in a font.c for you at some point soon.

font.c

```
u8 fontdata[] = {
// ... non-printable stuff, whitespace, capital characters
/* num: 97 */
0,0,0,0,0,0,
0,0,0,0,0,0,
0,0,1,1,1,0,
0,0,0,0,0,1,
0,0,1,1,1,1,
0,1,0,0,0,1,
0,0,1,1,1,1,
0,0,0,0,0,0,
/* num: 98 */
0,1,0,0,0,0,
0,1,0,0,0,0,
0,1,1,1,1,0,
0,1,0,0,0,1,
0,1,0,0,0,1,
0,1,0,0,0,1,
0,1,1,1,1,0,
0,0,0,0,0,0,
// ... more ASCII stuff
}
```

font.c

```
u8 fontdata[] = {
// ... non-printable stuff, whitespace, capital characters
/* num: 97 */
0,0,0,0,0,0,
0,0,0,0,0,0,
0,0,1,1,1,0,
0,0,0,0,0,1,
0,0,1,1,1,1,
0,1,0,0,0,1,
0,0,1,1,1,1,
0,0,0,0,0,0,
/* num: 98 */
0,1,0,0,0,0,
0,1,0,0,0,0,
0,1,1,1,1,0,
0,1,0,0,0,1,
0,1,0,0,0,1,
0,1,0,0,0,1,
0,1,1,1,1,0,
0,0,0,0,0,0,
// ... more ASCII stuff
}
```

Given font.c

If each letter is 8x6, the length of each is then 48.

To get the set of 48 values for a given character, you need to multiply the ASCII value of the character you want by 48.

fontdata[48 * 'a'] is the offset to the 48 chars (0/1) you want for the letter a.

```
#include "mylib.h"
#include "font.h"

void drawChar(int x, int y, char ch, u16 color){
  for(int i=0; i<6; i++) {
    for(int j=0; j<8; j++) {
      if (fontdata_6x8[48*ch + i + 6*j]) {
        SetPixel(x + i, y + j, color);
      }
    }
  }
}
int main() {
  REG_DISPCNT = MODE3 | BG2_ENABLE;
  u16 color1 = RGB(31,0,31);
  u16 color2 = RGB(0,31,0);
  drawChar(0, 0, 'H', color1);
  drawChar(6, 0, 'e', color2);
  drawChar(12, 0, 'l', color1);
  drawChar(18, 0, 'l', color2);
  drawChar(24, 0, 'o', color1);
  drawChar(6, 8, 'W', color1);
  drawChar(12, 8, 'o', color2);
  drawChar(18, 8, 'r', color1);
  drawChar(24, 8, 'l', color2);
  drawChar(30, 8, 'd', color1);
  drawChar(36, 8, '!', color2);
  while (1);
}
```


drawChar - VisualBoyAdvance-M 2.1.0
File Emulation Options Tools Help
Hello
World!
98%(0, 58 fps)

Now you could draw a String

Remembering that Strings are just a sequence of chars ending in `'\0'`

I'm leaving that as an exercise to the reader.

Direct Memory Access (DMA)

- What makes a program slow?

```
for(int i=0; i<34800; i++)
{
  VIDEO_BUFFER[i] = 0;
  // super slow clear screen
}
```

Is there a solution?

- Processor is slow
- Moving blocks of memory or filling blocks of memory is common and slow

Custom Circuits

- Can do a few things really well
- Used for basic operations
- If things become complicated must revert to general purpose processor

Enter DMA

- DMA = Direct Memory Access

- Hardware supported data copy
  - Up to 10x as fast as array copies
    - Especially for larger arrays
  - You set it up, the CPU is halted, data is transferred, and CPU gains back control
  - It's completely blind to the contents/type of the data being copied (so we'll use void pointers with it).

DMA Channels

The GBA has 4 DMA channels: (we'll only use 3 for now)

- 0
  - Highest Priority
  - Time Critical Operations
  - Only works with IWRAM
- 1 & 2
  - Transfer sound chunks to sound buffer
- 3
  - Lowest Priority
  - General purpose copies, like loading tiles or bitmaps into memory

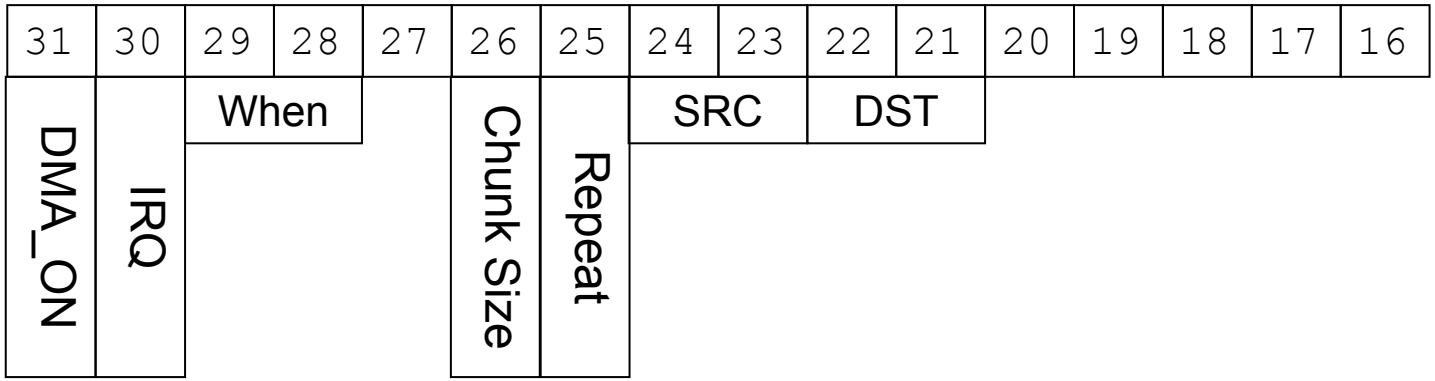
Using DMA

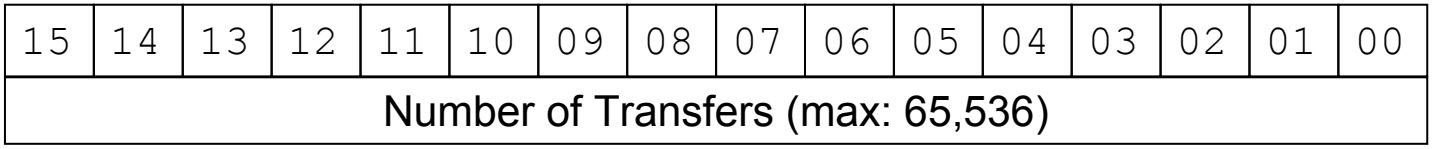
- Source
  - REG_DMAxSAD (0x040000B0 for channel 0)
    (where x is the channel number: 0, 1, 2, 3)
  - The address of the data that will be copied
- Destination
  - REG_DMAxDAD (0x040000B4 for channel 0)
  - The address to copy the data to
- Amount
  - REG_DMAxCNT (DMA control) (0x040000B8 for channel 0)
  - How much to copy plus options

REG_DMAxCNT

- Lower 16 bits contain amount to transfer
- Upper 16 bits contain other options
  - Turn on a DMA channel
  - When to perform the DMA
  - How the copy source and destination behave
  - How much to copy at a time
  - Whether or not to throw an interrupt on completion
  - Repeat or don't repeat on finish
- Can be treated as one 32 bit register, or two 16 bit registers
  - They're all bits, after all!

DMA Control bits





This is the number of *chunks* to copy. Not the number of bytes.
Zero here means do it 2^16 times (since zero made no sense to be an option here)

DMA Control Bits explained

bits	name	bits description
0-15	N	Number of transfers (where 0 means 2^16).
16-20	NOT USED	
21-22	Destination Adjustment bits	
	DMA_DST_INC	00: increment after each transfer (default)
	DMA_DST_DEC	01: decrement after each transfer
	DMA_DST_FIXED	10: none; address is fixed
	DMA_DST_RESET	11: haven't used it yet, but apparently this will increment the destination during the transfer, and reset it to the original value when it's done.
23-24	Source Adjustment bits	
	DMA_SRC_INC	00: increment after each transfer (default)
	DMA_SRC_DEC	01: decrement after each transfer
	DMA_SRC_FIXED	10: none; address is fixed
	DMA_SRC_RESET	11: "forbidden" for source
25	DMA_REPEAT	If set, repeats the copy at each VBlank or HBlank if the DMA timing has been set to those modes.

DMA Control Bits explained

bits	name	bits description
26	CS (Chunk size)	0: Copy by half-word (16 bits) 1: Copy by word (32 bits)
27	NOT USED	
28-29	TM (Timing Mode)	00: Start immediately (still a small delay before it takes over) 01: Start at VBlank 10: Start at HBlank 11: Start at Refresh - advanced and probably not what you expect. Ignore for now (or forever?)
30	I (Interrupt Request)	If set, Raise an interrupt when finished.
31	En (Enable)	Enable the DMA transfer for this channel (turn it on!)

Source/Dest Adjustment

1. Increment Source and Dest:
   - Both set to defaults (00)
   - This copies source to dest
   - Ex: drawImage, drawFullScreenImage
2. Source Fixed, increment Dest:
   - Source set to (10), Dest set to (00)
   - Fills dest with one value from source
   - Ex: drawRect, fillScreen

All the Dest. fixed options probably make no sense at all (lots of overwriting). Decrementing just lets you do things backwards (or reverse things, if src and dest are going in opposite directions).

Chunk Size and Number

- Ex. Copy an array of 43 shorts
  - chunk size 16, copy 43 chunks
- Ex. Copy an array of 103 ints
  - chunk size 32, copy 103 chunks
  - chunk size 16, copy 206 chunks (half an int each time)
- Ex. Copy an array of 82 chars
  - chunk size 16, copy 41 chunks (two chars each time)
  - can't chunk by 32, because then you'd want to copy 20.5 chunks

Memory

X

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

```
char *cp = X;
```

Memory

X

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

```
cp++;
```

Memory

X	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

```
u16 *sp = X;
```

Memory

X

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

```
sp++;
```

CS2261

Memory

X	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

```
int *intptr = X;
```

CS2261

Memory

X

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

```
intptr++;
```

Memory

X

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

```
typedef struct
{
   const volatile void *src;
   volatile void *dst;
   volatile u32 cnt;
} DMAREC;

DMAREC *dma = X;
```

Memory

X

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

```
dma++;
```

Why are we doing this?

DMA control registers are all consecutive in memory!

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

```
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile u32 cnt;
} DMAREC;

DMAREC *dma = (DMAREC *)0x040000B0;
```

Memory


0x040000B0
00000000 00000000 00000000 00000000
DMA0SAD
DMA0DAD
DMA0CNT
DMA1SAD
DMA1DAD
DMA1CNT
DMA2SAD
DMA2DAD
DMA2CNT
DMA3SAD
DMA3DAD
DMA3CNT
dma[0];
dma[1];
dma[2];
dma[3];

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

```
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile u32 cnt;
} DMAREC;

DMAREC *dma = (DMAREC *)0x040000B0;
```

```
dma[3].src;
```

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

```
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile u32 cnt;
} DMAREC;

DMAREC *dma = (DMAREC *)0x040000B0;
```

```
dma[3].dst;
```

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

```
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile u32 cnt;
} DMAREC;

DMAREC *dma = (DMAREC *)0x040000B0;
```

```
dp[3].cnt;
```

DMA Setup

- Map a struct array over the DMA registers

```
typedef struct
{
    const volatile void *src;
    volatile void *dst;
    volatile u32 cnt;
} DMAREC;

#define DMA ((volatile DMAREC*)0x040000b0)
```

Filling the Screen

```
void fillScreen(volatile u16 color){
  DMA[3].cnt = 0;  // clear it first!
  DMA[3].src = &color;
  DMA[3].dst = VIDEO_BUFFER;
  DMA[3].cnt = 1 << 31 | // turn it on!
               1 << 26 | // set chunk size to 32 bits
               1 << 22 | // set src as fixed
               16200;    // 32400 / 2
}

int main(){
  REG_DISPCNT = MODE3 | BG2_ENABLE;
  u16 i = 0;
  while (1) {
    fillScreen(i);
    i+= 127;
  }
}
```

Filling a rectangle

Fill row-by-row using DMA for each row.

As long as each row is >10 pixels, it should still be faster than manually drawing every pixel.

When to DMA?

- Copying/filling a lot of data (more than ~20 pixels) with NO LOGIC to the copy.
  - drawRect, fillScreen
  - drawImage
  - arrayCopy
  - arrayReverse

- drawChar with DMA - nope, since you want logic (though you could copyChar with DMA)