

# **CS 2261: Media Device Architecture - Week 5**

# Announcements

- Quizzes are graded -- average was pretty good
  - Will be handed back next class meeting, after excused make-ups are resolved.
  - Next quiz *will* be tougher (that survey was pretty generous)
- Quiz 2: September 26th
- Quiz 3: October 10th
  - Spoiler Alert: I won't be here, but the show must go on!

# Announcements Continued

- Milestones / Project Demo Schedule (more details coming)
  - M1: 11/01
  - M2: 11/08
  - M3: 11/15
  - M4: 11/29
  - Volunteer Demos: 12/04 & 12/05 ("Final Instructional Days")
    - Present this day -- in front of the whole class -- and skip the final exam period.
  - Science Fair Demo Period:  
(????) Wed., Dec 12 2:40 PM - 5:30 PM (????)
    - I'm still confirming this last one -- our class meets at odd times according to OSCAR...

# Overview

- Pointer as variables (*mostly* review)
- Pointers as function arguments
- **SWAP**
- C Arrays
  - Nested Arrays
  - Arrays vs Pointers
  - Arrays as function arguments

# Pointers as variables

```
// some static variables
int foo;
int *bar;
int **baz;

int main(){
    ...
}
```

## Variable Table

- These variables will be in the static section of memory. Let's just call that 0xF0, for example purposes only.

| Name | Address | Value |
|------|---------|-------|
| foo  | 0xF0    | 0     |
| bar  | 0xF4    | NULL  |
| baz  | 0xF8    | NULL  |

### Note:

- The addresses here increase. When I tested it locally, they *decreased*. They could technically be in completely different places, despite being all static vars. "Implementation-specific..."

# Pointers as variables


```
// some static variables
int foo;
int *bar;
int **baz;

int main(){
    // set bar to address of foo
    bar = &foo;
}
```

## Variable Table

- These variables will be in the static section of memory. Let's just call that 0xF0, for example purposes only.

| Name | Address | Value |
|------|---------|-------|
| foo  | 0xF0    | 0     |
| bar  | 0xF4    | 0xF0  |
| baz  | 0xF8    | NULL  |



# Pointers as variables

```
// some static variables
int foo;
int *bar;
int **baz;

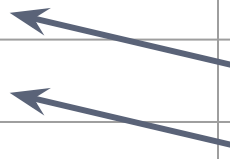
int main(){
    // set bar to address of foo
    bar = &foo;

    // set baz to address of bar
    baz = &bar;
}
```

## Variable Table

- These variables will be in the static section of memory. Let's just call that 0xF0, for example purposes only.

| Name | Address | Value |
|------|---------|-------|
| foo  | 0xF0    | 0     |
| bar  | 0xF4    | 0xF0  |
| baz  | 0xF8    | 0xF4  |



# Pointers as variables

```
// some static variables
int foo;
int *bar;
int **baz;

int main(){
    // set bar to address of foo
    bar = &foo;

    // set baz to address of bar
    baz = &bar;

    // double-dereference baz
    // to alter foo
    **baz = 3;
}
```

## Variable Table

- These variables will be in the static section of memory. Let's just call that 0xF0, for example purposes only.

| Name | Address | Value |
|------|---------|-------|
| foo  | 0xF0    | 3     |
| bar  | 0xF4    | 0xF0  |
| baz  | 0xF8    | 0xF4  |

```
graph LR
    bar[bar] --> foo[foo]
    baz[baz] --> bar[bar]
```



# Whiteboard from class:

```
int main() {
```

```
    int foo;
```

```
    int * bar;
```

```
    int ** baz;
```

```
}
```

```
    baz = &bar;  
    bar = &foo;  
    ** baz = 3;
```

0xF0

foo (int)

00000000

0xF4

bar (int\*)

0xF0

0xF8

baz (int\*)

0xF4

bar = 0 (=null)

# Notes

- NULL, not null or Null
- `int *ptr = NULL;` and `int *ptr = 0;` compile to the same thing.
  - Which, again, may not actually be a pointer with a value of zero. It definitely isn't in some cases.
- `int *ptr;` is not the same as `int *ptr = NULL;` for a *dynamic* variable. (It *is* the same for one with static storage duration).
- There are also pointers "without a type":
  - `void *voidPointer;`  
`// more later (memory allocation)`

# Pointers and Functions

- ~~Next~~ This class we will pass pointers as arguments to functions.
  - This is how you accomplish passing "by reference" in C
    - C does not pass by reference, but Java does for objects.
    - C actually passes by value, it's just that you're providing a pointer *as* the value.
- To be continued... on the next slide!

# Famous Thoughts on Pointers

"Pointers are like jumps, leading wildly from one part of the data structure to another. Their introduction into high-level languages has been a step backwards from which we may never recover." -- Tony Hoare

"You can either have software quality or you can have pointer arithmetic, but you cannot have both at the same time." -- Bertrand Meyer

"Pointers are cool!" -- Jim Greenlee

# Pointers

- Powerful and dangerous
- No runtime checking (for efficiency)
  - Easy to get out of bounds (Segmentation Fault!)
- Bad reputation (in fairness, the syntax is confusing)
  - `int foo; int *ptr = &foo; // Good`
  - `int foo; int *prt; *ptr = &foo; // BAD!`
- Java attempts to remove the features of pointers that cause many of the problems hence the decision to call them references
  - No "address of" operators
  - No dereferencing operator (always dereferencing)
  - No pointer arithmetic

# Pointers as Function Arguments

```
// contrived_example.c
#include "mylib.h"

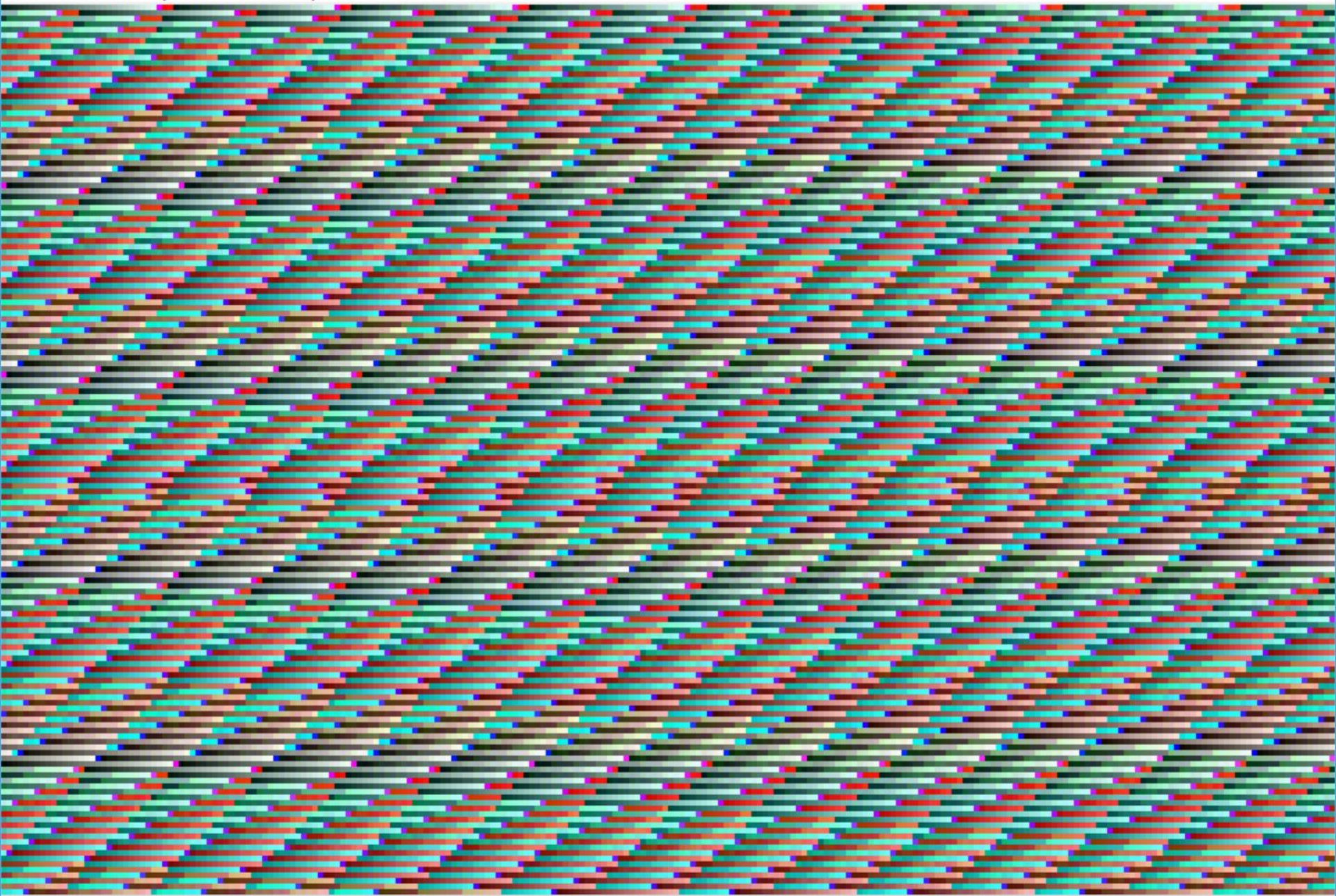
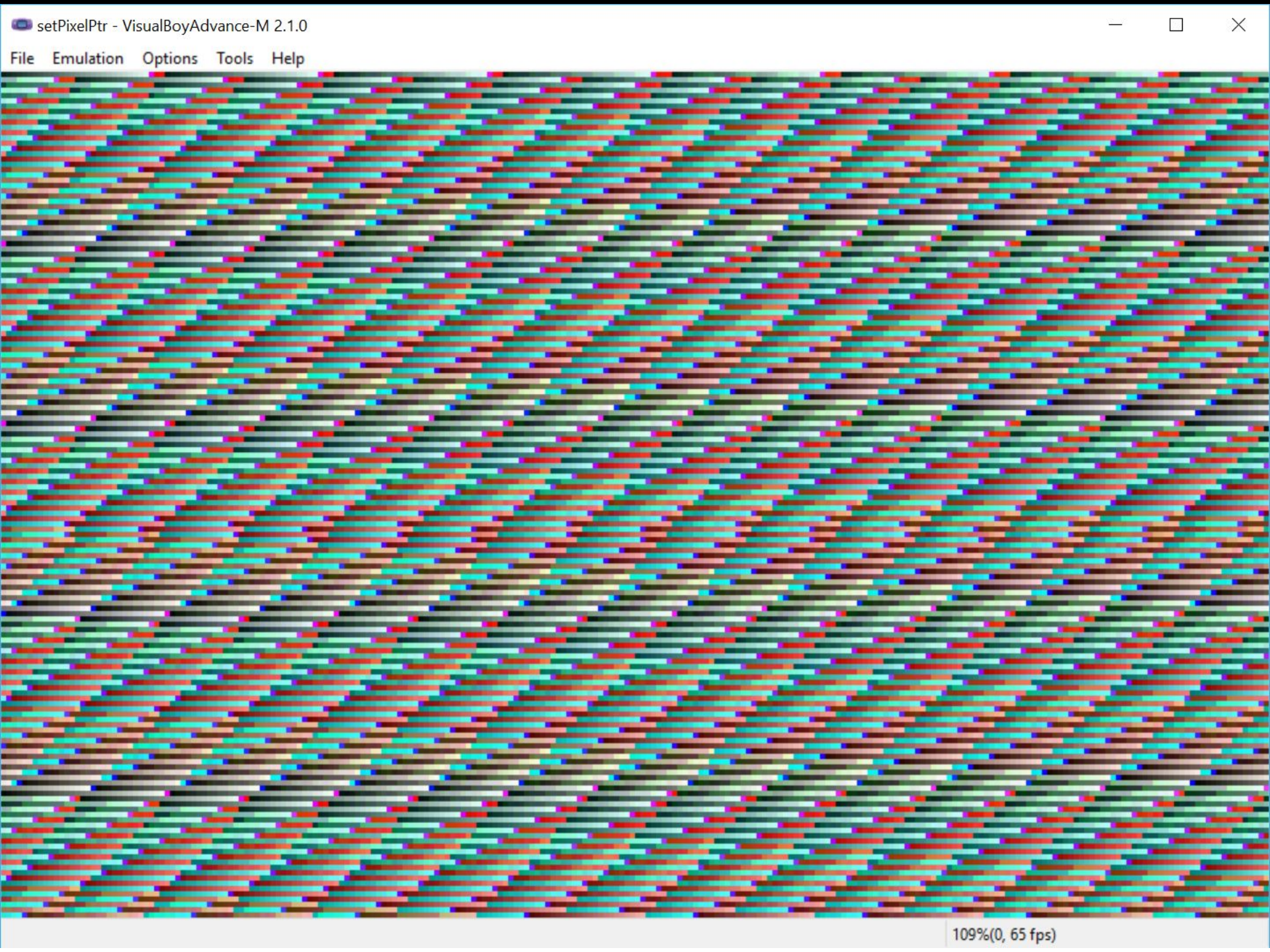
void setPixel(u16 *pixel, u16 color){
    *pixel = color;
}

int main() {
    REG_DISPCNT = MODE3 | BG2_ENABLE;

    for(int i=0; i<86400; i++){
        setPixel(VIDEO_BUFFER + i, RGB(i % 255, i % 127, 127 + i %
127));
    }

    while (1);
}
```





# Better Example

- What if I wanted to swap two pixels?
  - `u16* pixel1; u16* pixel2;`  
`pixel1 = pixel2;`  
`pixel2 = pixel1; // Good?`



# Better Example

- What if I wanted to swap two pixels?

- `u16* pixel1; u16* pixel2;`  
`pixel1 = pixel2;`  
`pixel2 = pixel1; // Good? Not even close!`

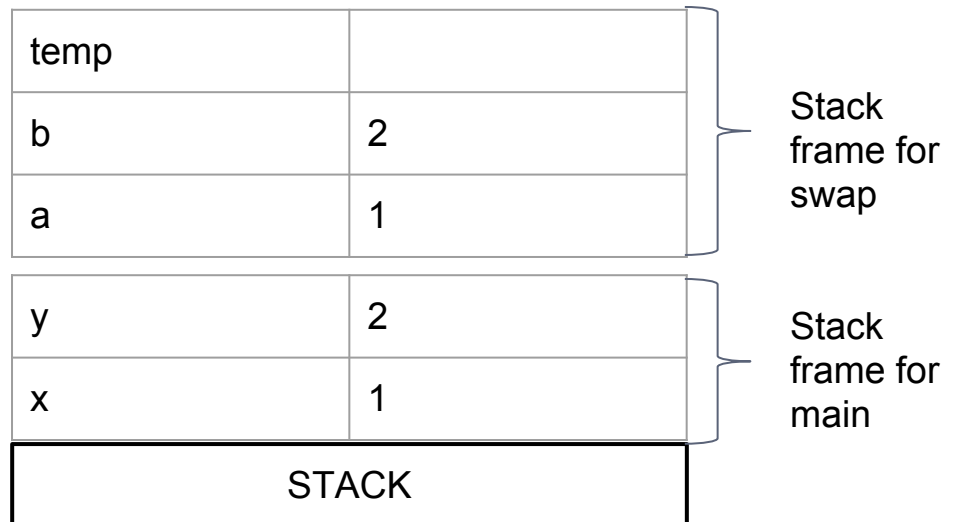
- Get closer:

- `// add a temp var`  
`u16* pixel1; u16* pixel2; u16* temp;`  
`temp = pixel1;`  
`pixel1 = pixel2;`  
`pixel2 = temp; // Better? To the board!`

# BAD Swap Function

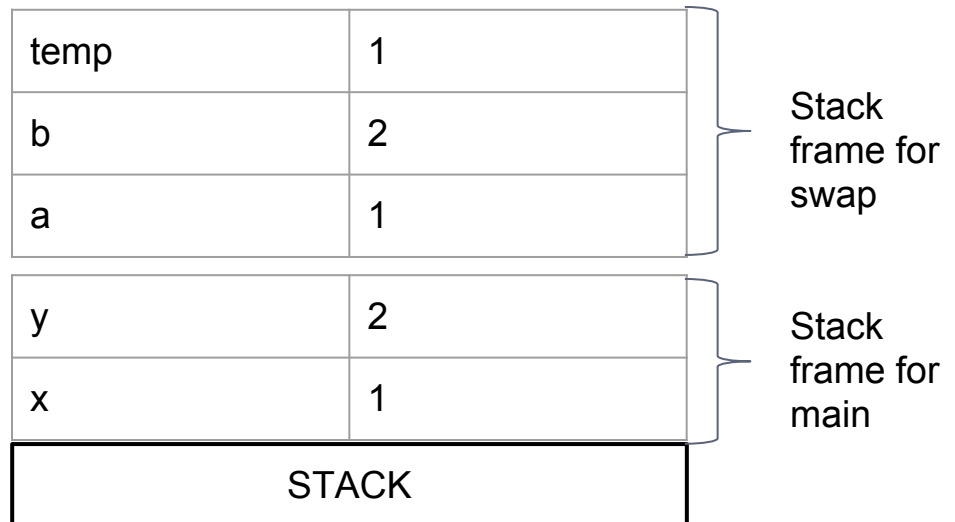
```
void swap(u16 a, u16 b){
    u16 temp = a;
    a = b;
    b = temp;
}

int main() {
    u16 x = 1;
    u16 y = 2;
    swap(x, y);
}
```



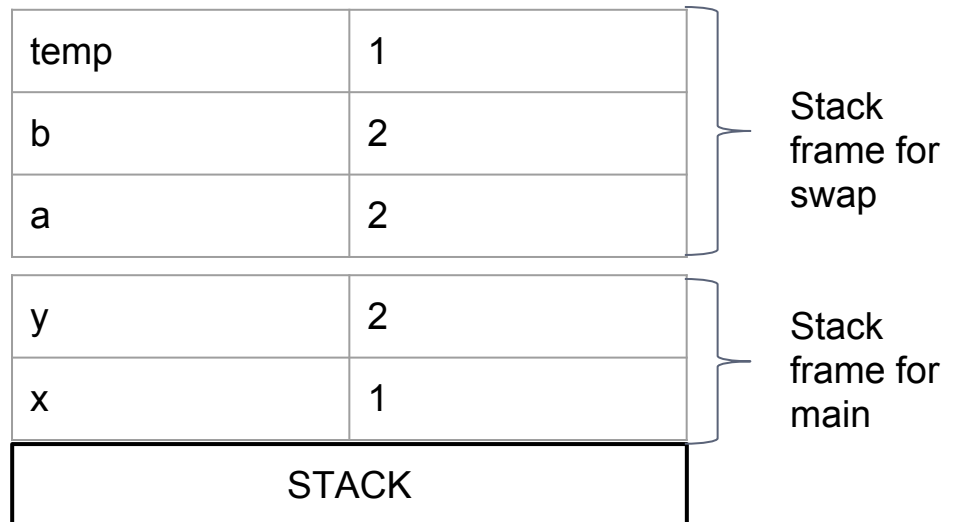
# BAD Swap Function

```
void swap(u16 a, u16 b){  
    u16 temp = a;  
    a = b;  
    b = temp;  
}  
  
int main() {  
    u16 x = 1;  
    u16 y = 2;  
    swap(x, y);  
}
```



# BAD Swap Function

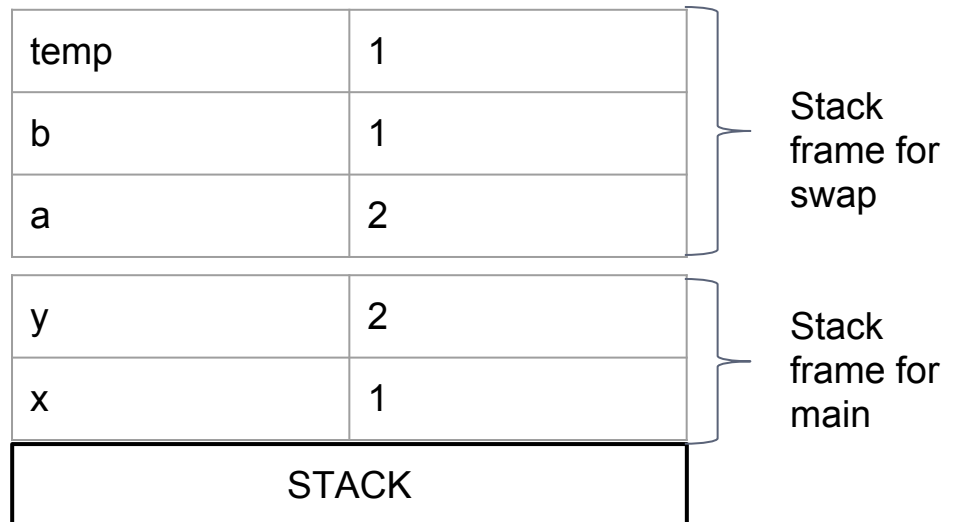
```
void swap(u16 a, u16 b){  
    u16 temp = a;  
    a = b;  
    b = temp;  
}  
  
int main() {  
    u16 x = 1;  
    u16 y = 2;  
    swap(x, y);  
}
```



# BAD Swap Function

```
void swap(u16 a, u16 b){
    u16 temp = a;
    a = b;
    b = temp;
}

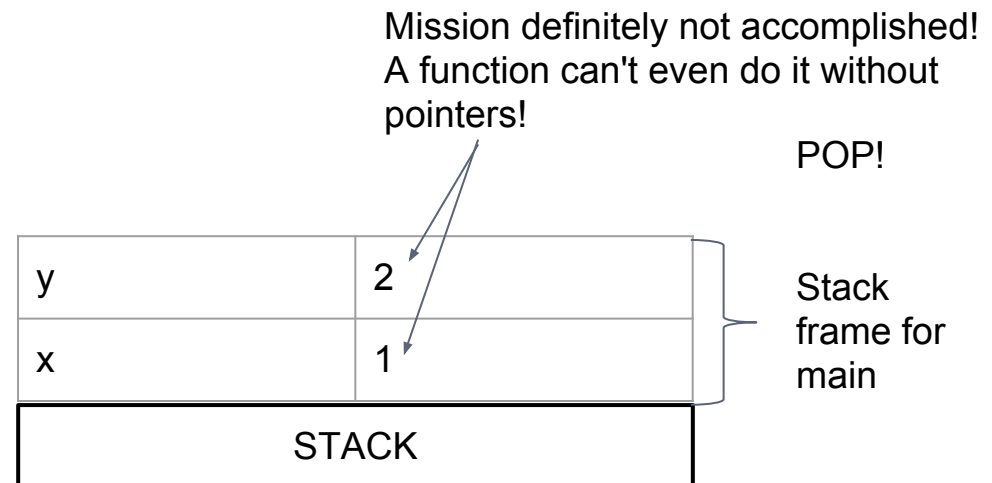
int main() {
    u16 x = 1;
    u16 y = 2;
    swap(x, y);
}
```



# BAD Swap Function

```
void swap(u16 a, u16 b){
    u16 temp = a;
    a = b;
    b = temp;
}

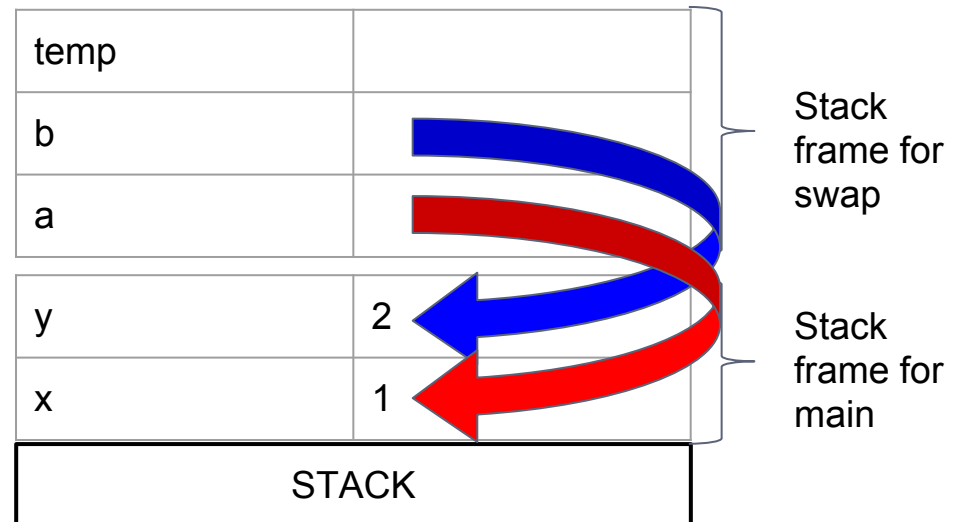
int main() {
    u16 x = 1;
    u16 y = 2;
    swap(x, y);
}
```



# Swap Function

```
void swap(u16 *a, u16 *b){
    u16 temp = *a;
    *a = *b;
    *b = temp;
}

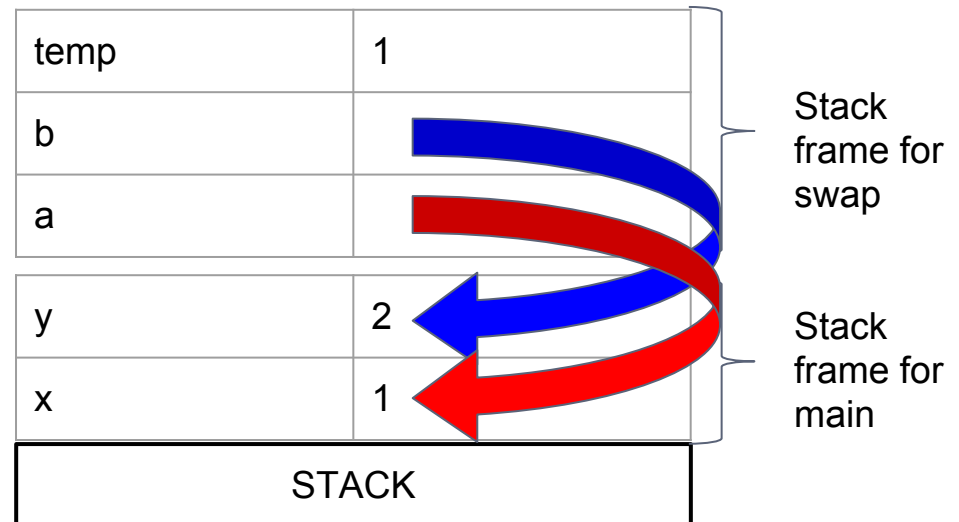
int main() {
    u16 x = 1;
    u16 y = 2;
    swap(&x, &y);
}
```



# Swap Function

```
void swap(u16 *a, u16 *b){
    u16 temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    u16 x = 1;
    u16 y = 2;
    swap(&x, &y);
}
```

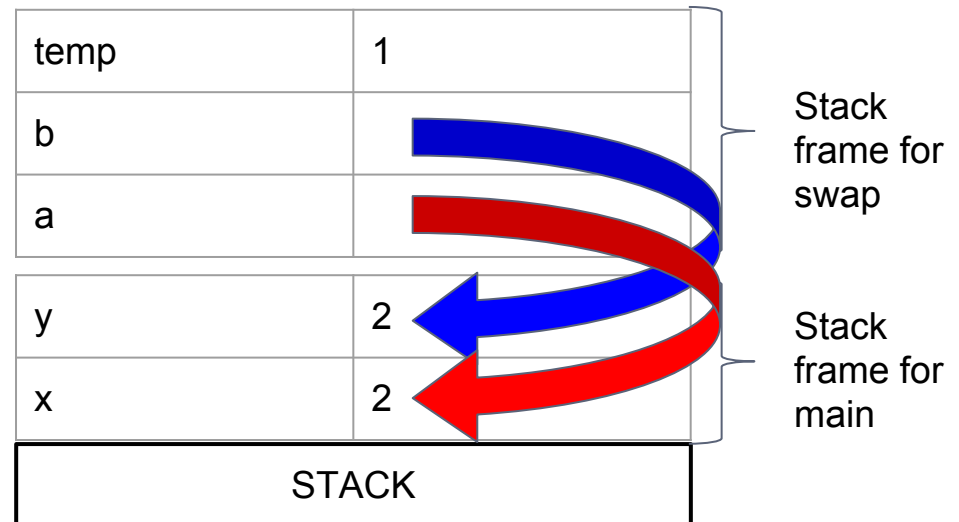




# Swap Function

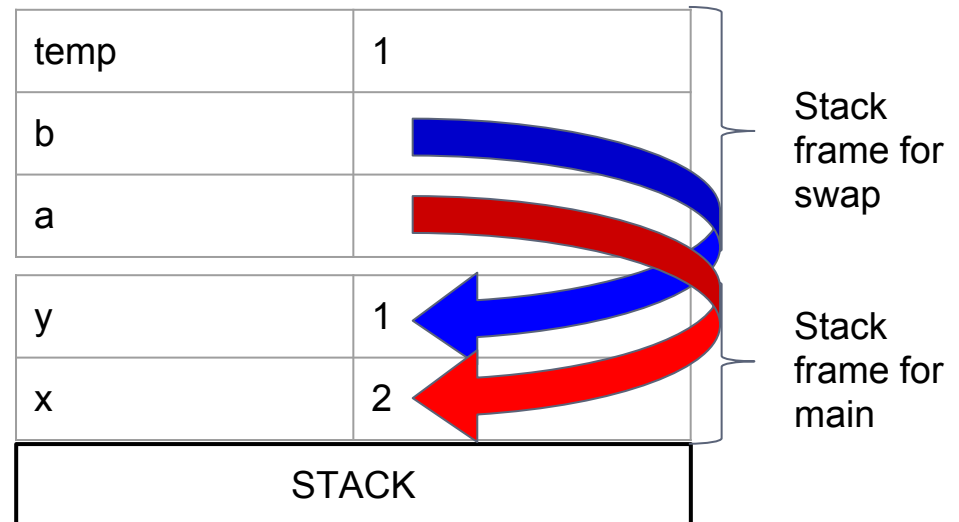
```
void swap(u16 *a, u16 *b){
    u16 temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    u16 x = 1;
    u16 y = 2;
    swap(&x, &y);
}
```



# Swap Function

```
void swap(u16 *a, u16 *b){  
    u16 temp = *a;  
    *a = *b;  
    *b = temp;  
}  
  
int main() {  
    u16 x = 1;  
    u16 y = 2;  
    swap(&x, &y);  
}
```

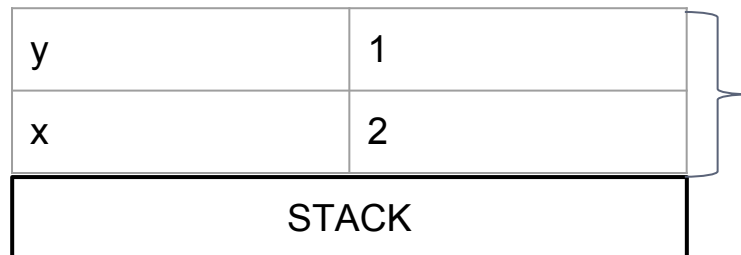


# Swap Function

```
void swap(u16 *a, u16 *b){  
    u16 temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

```
int main() {  
    u16 x = 1;  
    u16 y = 2;  
    swap(&x, &y);  
}
```

POP!

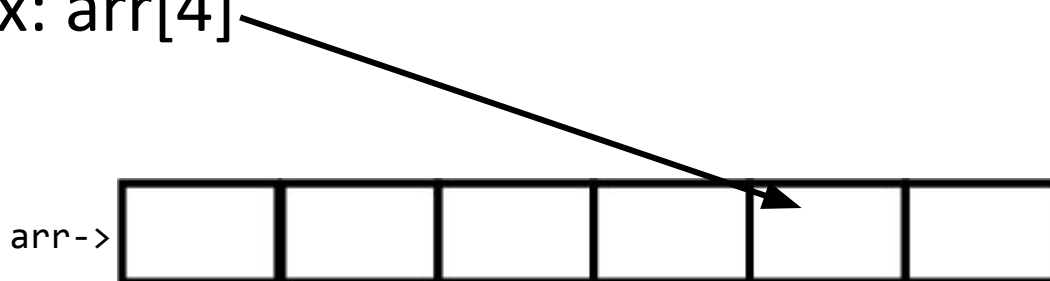


Stack  
frame for  
main

# Arrays

## ■ `int arr[6];`

- Allocates 6 consecutive int-sized spaces in memory.
  - How much is that? `6 * sizeof(int) // 24 on GBA`
  - The type of `arr` is `int[6]`;
- `arr` is effectively a constant pointer to the first member of the integer array
  - `arr` is `&arr[0]`
- Indexing into `arr` uses pointer arithmetic via the array syntax: `arr[4]`



# Array Syntax

```
int arr[] = {1,2,3}; // array literal  
int arr[3] = {1,2,3}; // same
```

```
int arr[3] = {1,2,3,4,5}; // Technically legal (warning).  
// No bounds checking when creating or assigning to arrays in C
```

```
int arr[3];  
arr[17] = 2; // Buffer overflow (but legal in C)
```

# Arrays vs Pointers

```
int arr[6] = {0, 1, 2, 3, 4, 5}; // instantiated with vals
int *ptr;
```

```
// arr[0] => *(arr + 0*sizeof(int))
// Pointer Arithmetic is why array indexing begins at zero
```

```
ptr = arr // okay
arr = ptr // Nope! arr as a pointer is fixed.
```

```
sizeof(ptr) // 4 on GBA (32-bit)
sizeof(arr) // 24 (6 * sizeof(int))
```

```
&ptr; // address of ptr, itself (not where it points)
&arr; // address of the first element in arr: &arr[0]
// arr is &arr is &arr[0].
// you don't get to ask where arr as a variable is.
```

# Pointer Arithmetic

```
int i;  
int ia[MAX];  
for(i = 0; i < MAX; i++)  
    ia[i] = 0;
```

addition

addition

```
int *ip;  
int ia[MAX];  
for(ip = ia; ip < ia + MAX; ip++)  
    *ip = 0;
```

addition

no addition

Sometimes pointer arithmetic  
is faster than array manipulation

# Arrays as Function Arguments

- Arrays only half-remember their size.
  - `void myFunction(int arr[]){}`
  - the same as:
  - `void myFunction(int *arr){}`

Inside of BOTH, `arr` no longer knows its own size.  
C is not an OO language. `arr.length` is not a thing.



# The Length Problem

```
void myFunction(int arr[8]){  
}
```

- This function will be compile-time enforced to only allow you to call it with arrays of length 8.
  - We can't go writing one of these for every possible array length
    - Also, what would we do with arrays we create at runtime?

# A Fix?

```
void myFunc(int arr[]){  
    int length = sizeof(arr) / sizeof(arr[0]);  
    for (int i=0; i<length; i++) {  
        // do something now with the array  
    }  
}
```

# This does not work!

```
void myFunc(int arr[]){  
    int length = sizeof(arr) / sizeof(arr[0]);  
    for (int i=0; i<length; i++) {  
        // do something now with the array  
    }  
}
```

sizeof is a compile-time function. sizeof(arr) in this context is the same as sizeof(int \*)

```
int length = sizeof(int*) / sizeof(int); // 1 on GBA
```

# The Length "Fix" :(

- Pass the length as an argument. Always!

```
void myFunc(int arr[], size_t length){  
}
```

```
/*
```

size\_t is a special type...

It's actually the type returned by sizeof().

It's an implementation-specific alias to one of the unsigned integer types, and guaranteed to be at least 16 bits.

```
*/
```

# Modifying the Members of an Array

```
int arr = {1,2,3,4,5};

void doubleMembers(int arr[], size_t length){
    for(int i; i<length; i++){
        arr[i] = 2*arr[i];
    }
}
```

# Arrays of Arrays

```
int disp[2][4] = {  
    {10, 11, 12, 13},  
    {14, 15, 16, 17}  
};
```

OR

```
int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};
```

disp[row][col] translates to:  
 $*(disp + row * row\_size + col)$

# Functions that return Arrays

Arrays are a way of asking for memory, so could we use them to create new memory chunks for us to use dynamically?

```
int *func(int length) {  
    int a[length]; // var-length array C99+  
    return a;  
}
```

Why won't this work (even if C let you do it)?

Tiny Demo.

We will get to the actual way to dynamically request memory later. Dynamic arrays are too ephemeral.