CS 2261: Media Device Architecture - Week 4

HANSEN – Sep. 10, 2018 – Georgia Institute of Technology

Reminder

- Quiz Wednesday!
  - Bring your Buzz Card!
  - Be here! (email me now if you're going to miss it -- or ASAP is you have a legitimate emergency)
  - There will be lecture after it (albeit probably a half-length one).
  - Review class notes to study for it! HW / Lab will help some, but all of lecture is fair game.
- Homework 1 due Friday!
- Milestones schedule coming out soon (apologies for the delay)

Grades & TA assignments

- TA assignments will be communicated directly with each of you shortly (Canvas is being resistant to how I would have done this with T-Square).

- Grades for Lab0 and Lab1 should be entered into Canvas this week. (let me know if yours haven't been by Saturday Morning!)

Overview

- How Can We Reuse Code?
  - Program Layout / Code Organization in C
    - .c & .h files
    - definition vs declaration
      - prototype / signature
- Inputs Example
  - BreakoutRevisited
- variables, functions, and scope
  - globals
  - extern
  - static variables

- Macros for good (and bad) -- Note: punted until next lecture.

We're building up a lot of things before main.

```c
#define REG_KEYINPUT (*(volatile u16*)0x04000130)

#define KEY_A        0x0001
#define KEY_B        0x0002
#define KEY_SELECT   0x0004
#define KEY_START    0x0008
#define KEY_RIGHT    0x0010
#define KEY_LEFT     0x0020
#define KEY_UP       0x0040
#define KEY_DOWN     0x0080
#define KEY_R        0x0100
#define KEY_L        0x0200

#define KEY_DOWN_NOW(key) (~(REG_KEYINPUT) & key)

#define RGB(R, G, B) ((R) | (G) << 5 | (B) << 10)
#define REG_DISPCNT (*(unsigned short *)0x04000000)
#define MODE3 3
#define BG2_ENABLE (1<<10)
#define VIDEO_BUFFER ((u16*)0x06000000)

typedef unsigned short u16;
typedef unsigned char u8;

#define SetPixel(x, y, val) (VIDEO_BUFFER[(x) + (y)*240] = val)
```

```c
volatile u16* scanlineCounter = (u16*) 0x04000006;

void drawSquare(u8 x, u8 y, u8 size, u16 color){
  for (u8 i=0; i<size; i++){
    for (u8 j=0; j<size; j++){
      SetPixel(x+i, y+j, color);
    }
  }
}

void waitForVBlank() {
  while (*scanlineCounter >= 160);
  while (*scanlineCounter < 160);
}

void drawHorizontalLine(u8 x, u8 y, u8 size){
  // ...
}

void drawVerticalLine(u8 x, u8 y, u8 length){
  // ...
}
```

Sure would be nice to be able to reuse some of this quickly/easily between projects.

Let's start mylib.c

- We have a bunch of code that could work for any GBA program using mode 3.

- #include is a preprocessor directive that copy-pastes the code into your own.

- Throw all that code in there, then just #include "mylib.c" in main.c (with mylib.c in the same directory)
  - Right?

Let’s start mylib.c

- We have a bunch of code that could work for any GBA program using mode 3.

- #include is a preprocessor directive that copy-pastes the code into your own.

- Throw all that code in there, then just #include "mylib.c" in main.c (with mylib.c in the same directory)
  - Right?
    - Not quite.

Let’s start `mylib.c`

- In C, there are two kinds of files. "Source files" (ending in .c), and "header files" (ending in .h).
    - Header files get included into many files.
        - They don't contain much actual code.
    - Source files include header files, and get linked together after compilation.
        - Source files aren't included in other source files*
            - Including them can be tricky and messy.

*-- except when they are for optimization reasons

What belongs in `mylib.h` vs `mylib.c`?

- mylib.h
  - Preprocessor macros
  - typedefs
  - declarations
    - function prototypes (aka "signatures")
    - global variable declarations (without assignment)

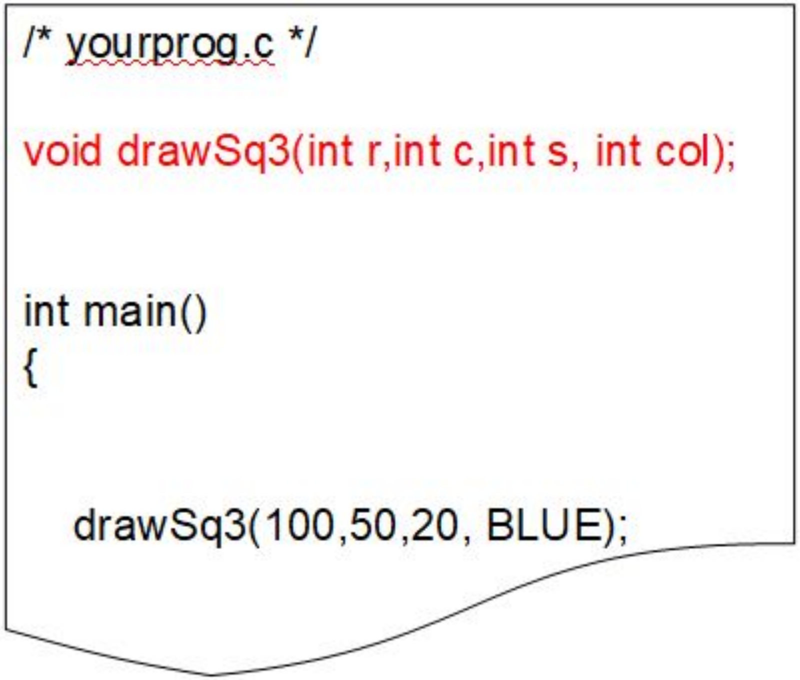
- mylib.c
  - `#include "mylib.h"  // include own header as first line!`
    - This is just a convention, but it's a well-established one.
  - Actual function definitions
  - Global variable assignments (you intend to make available to main.c)

Function Prototypes

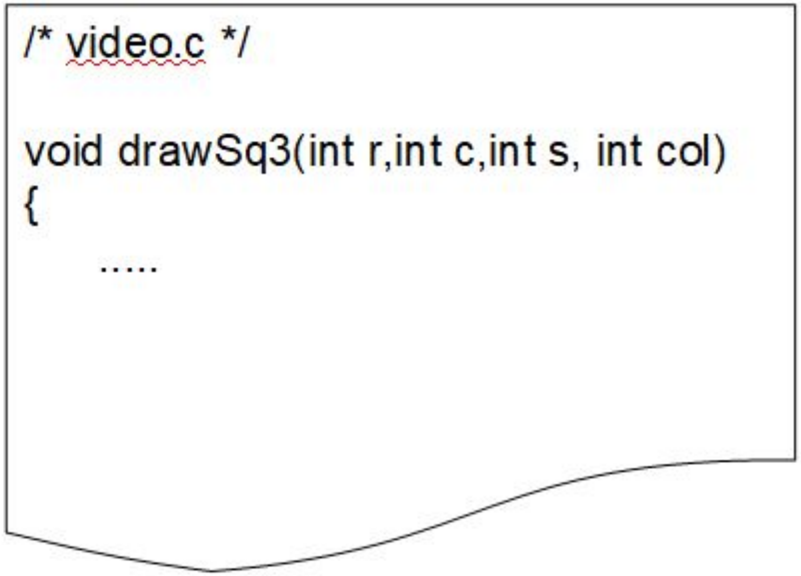
```
void foo();

int main();

void drawRect(int x, int y, int height, int width);
```

- Just the return type, method name, and the arguments list.
  - This is all the compiler needs to know for other .c files to be able to compile. It assumes the method will come from some file, and will find it at the linking step.
  - These are just like methods in Java interfaces.
  - In principle, you can compile a single file with only function prototypes present.

Without Header Files



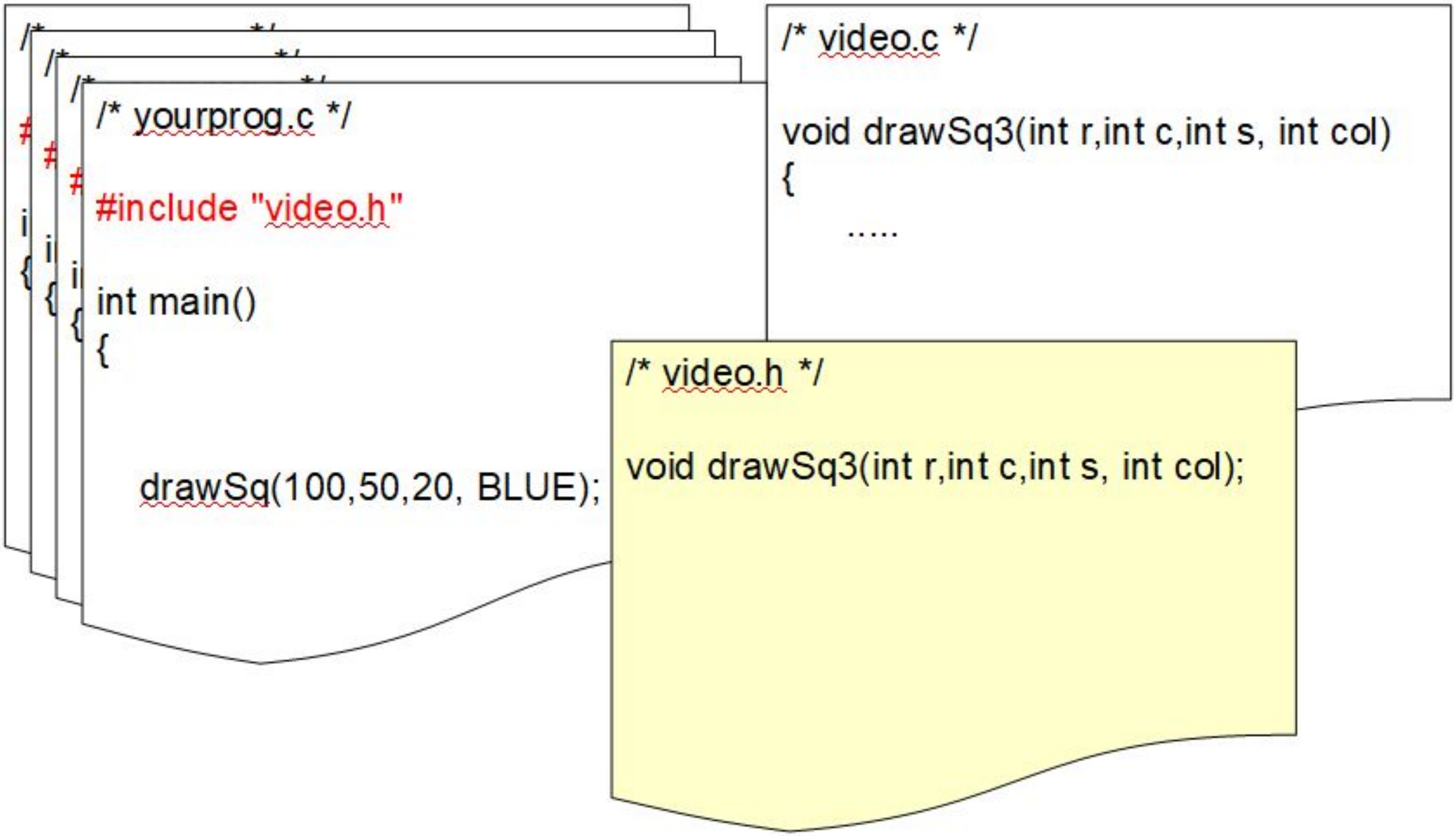
```
/* yourprog.c */

void drawSq3(int r,int c,int s, int col);


int main()
{


   drawSq3(100,50,20, BLUE);
```



```
/* video.c */

void drawSq3(int r,int c,int s, int col)
{
      .....
```

This isn't too bad, but what if I had dozens of files?

Thank you, header files!



```
/* yourprog.c */

#include "video.h"

int main()
{


    drawSq(100,50,20, BLUE);
```

```
/* video.c */

void drawSq3(int r,int c,int s, int col)
{
      .....
```

```
/* video.h */

void drawSq3(int r,int c,int s, int col);
```

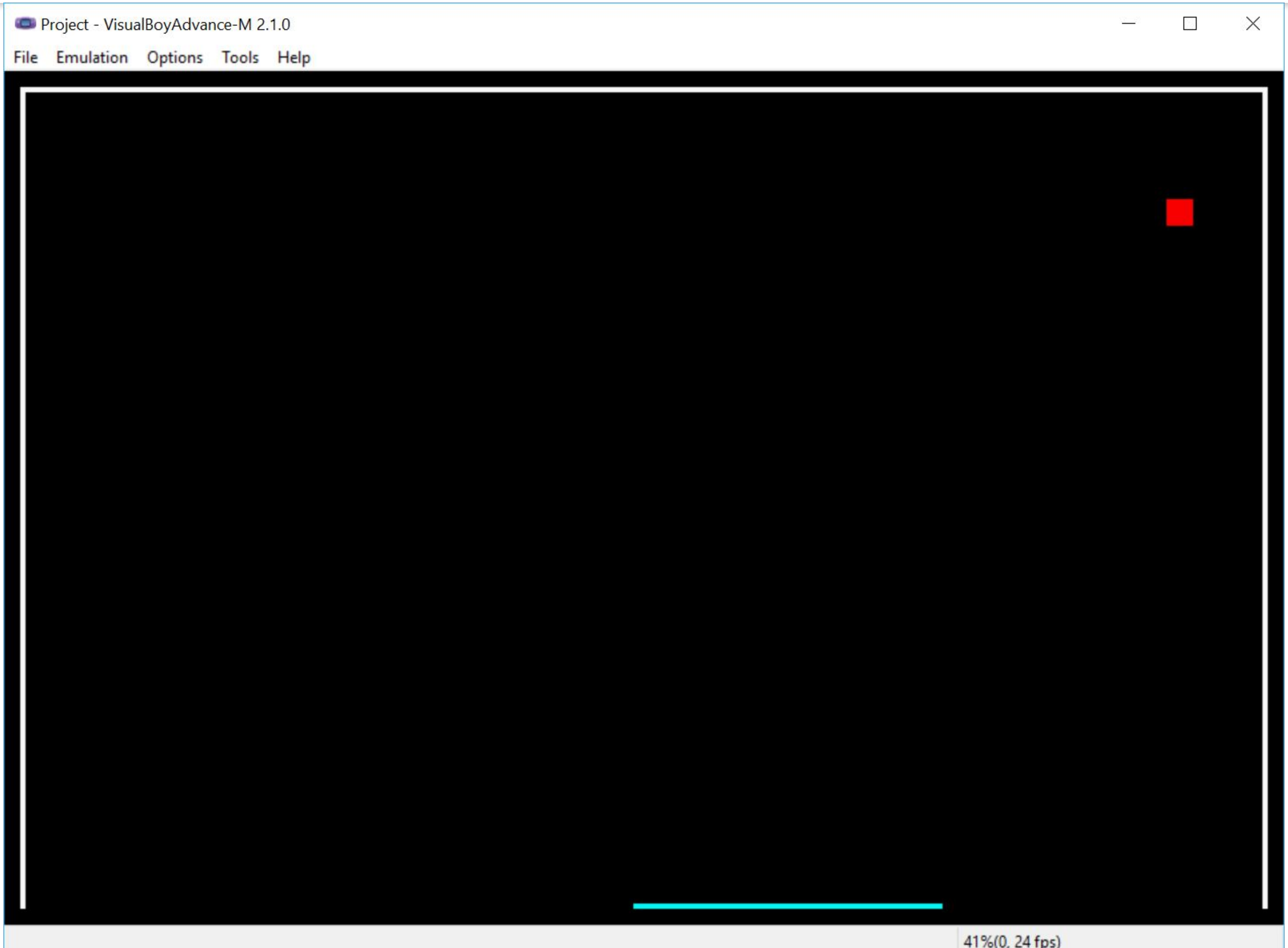
Using `mylib.c` with `main.c`

- Add `#include "mylib.h"` to the top of `main.c`
  - Make sure it's at the top of mylib.c too!
- Add mylib.c to the list of sources in your Makefile
  - SOURCES = main.c mylib.c
- Each .c file is compiled *individually*, leading to multiple .i and .s files. Then the linker combines them into a single executable.

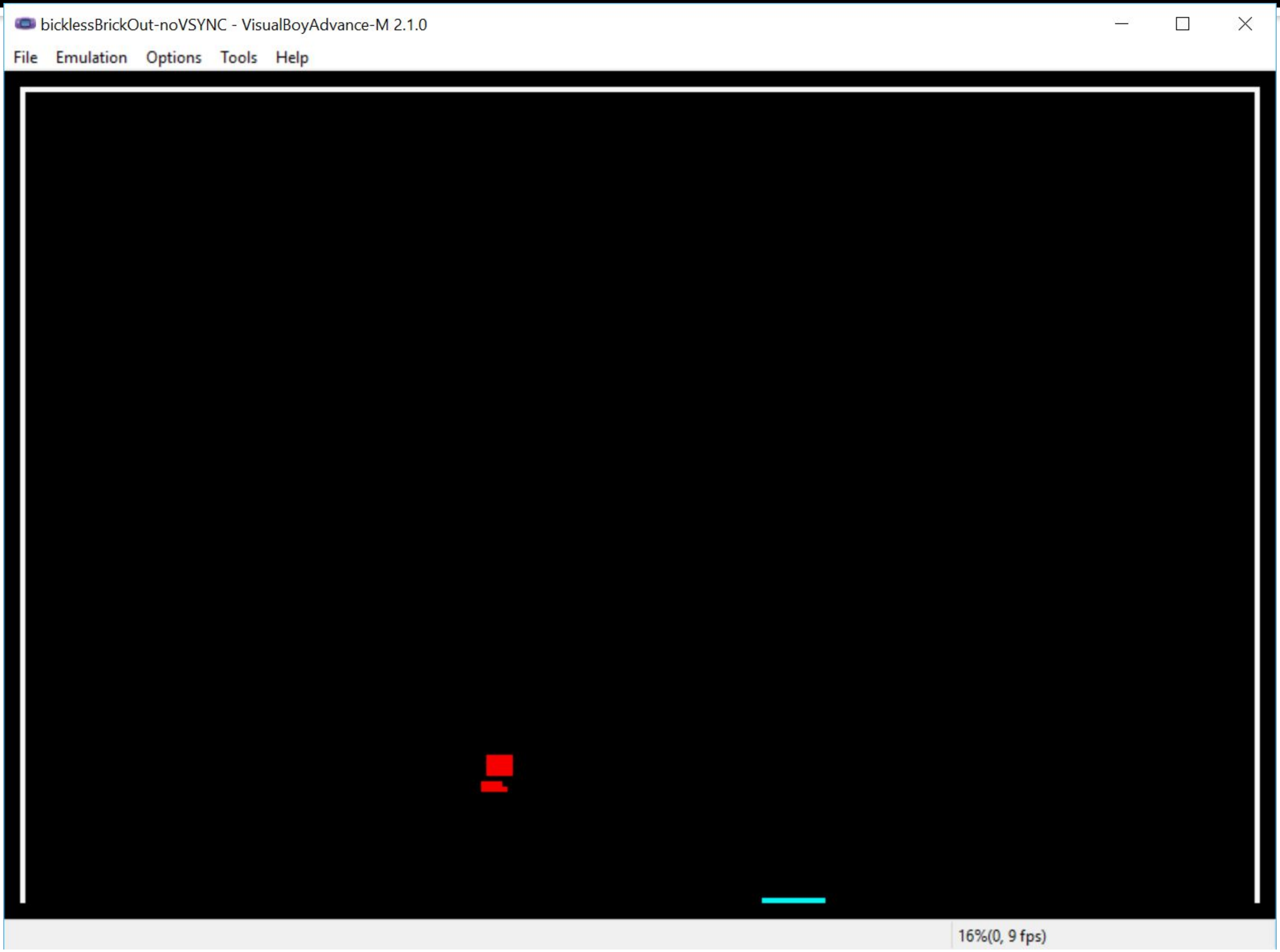
Using `mylib.c` with `main.c`

- Gotchas:
  - C compilers don't check header files for change every compilation, by default. (They do with .c files).
    - If you want to catch these changes, you need to either
      - Edit all of the .c files that use the header files
        - Fast recompile, but tedious and allows for human error (boo!).
      - Recompile everything using make clean build
        - This is the safest option.
        - This can be *really slow* for a large project (also boo!).

Let's Demo Working Brickless Brickout



Demo without Vsync



bicklessBrickOut-noVSYNC - VisualBoyAdvance-M 2.1.0

16%(0, 9 fps)

Flicker

- I talked about screen tearing before, however:
- Flicker is the more dominant issue in Mode 3
- Things alternate between shown and unshown, turn up in two places at once, get ripped apart, etc.
- Main takeaway: Vsync is definitely still your friend. The cpu is faster than the display hardware. Use that ~1/200 s (83776 cpu cycles) of VBlank to the fullest.

```c
/* mylib.h */
#define RGB(R, G, B) ((R) | (G) << 5 | (B) << 10)
#define REG_DISPCNT (*(unsigned short *)0x04000000)
#define MODE3 3
#define BG2_ENABLE (1<<10)
#define VIDEO_BUFFER ((u16*)0x06000000)
#define SetPixel(x, y, val) (VIDEO_BUFFER[(x) + (y)*240] = val)
#define V_COUNT (*(volatile u16*)0x04000006)

#define REG_KEYINPUT (*(volatile u16*)0x04000130)
#define KEY_A        0x0001
#define KEY_B        0x0002
#define KEY_SELECT   0x0004
#define KEY_START    0x0008
#define KEY_RIGHT    0x0010
#define KEY_LEFT     0x0020
#define KEY_UP       0x0040
#define KEY_DOWN     0x0080
#define KEY_R        0x0100
#define KEY_L        0x0200
#define KEY_DOWN_NOW(key) (~(REG_KEYINPUT) & key)

typedef unsigned short u16;
typedef unsigned char u8;

// definitions are in mylib.c
void drawVerticalLine(u8 x, u8 y, u8 length, u16 color);
void drawHorizontalLine(u8 x, u8 y, u8 length, u16 color);
void drawSquare(u8 x, u8 y, u8 size, u16 color);
void waitForVBlank();
```

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

void drawVerticalLine(u8 x, u8 y, u8 length, u16 color) {
  for(u8 i=0; i<length; i++){
    SetPixel(x, y + i, color);  // uses macro from mylib.h
  }
}

void drawHorizontalLine(u8 x, u8 y, u8 length, u16 color) {
  for(u8 i=0; i<length; i++){
    SetPixel(x + i, y, color);
  }
}

void drawSquare(u8 x, u8 y, u8 size, u16 color){
  for (u8 i=0; i<size; i++){
    for (u8 j=0; j<size; j++){
      SetPixel(x+i, y+j, color);
    }
  }
}

void waitForVBlank() {
  while (V_COUNT >= 160);  // wait until current VBlank ends
  while (V_COUNT < 160);  // wait until next VBlank starts
}
```

```c
/* main.c */
#include "mylib.h"

int time = 0;
int ballSize, ballX, ballY, ball_Vx, ball_Vy;
int prevBallX, prevBallY;
int padding, screenWidth, screenHeight;
int paddleSize, paddleX, prevPaddleX, paddleY;

void updateBallPosition() {
  int timestep = 3;
  prevBallX = ballX;
  prevBallY = ballY;
  if (time % timestep == 0 && time != 0) {
    ballX += ball_Vx;
    ballY += ball_Vy;

    if (ballX <= padding){
      ballX = padding + (padding -ballX) + 1;
      ball_Vx = -ball_Vx;
    }
    if (ballY <= padding){
      ballY = padding + (padding - ballY) + 1;
      ball_Vy = -ball_Vy;
    }
    if (ballX + ballSize >= 239 - padding) {
      ballX -= ballX + ballSize - (239 - padding);
      ball_Vx = -ball_Vx;
    }
    if ((ballY + ballSize >= paddleY) &&
        (ballX + ballSize >= paddleX) &&
        (ballX < paddleX + paddleSize)) {
      ballY -= ballY + ballSize - paddleY - 1;
      ball_Vy = -ball_Vy;
    }
  }
}
```

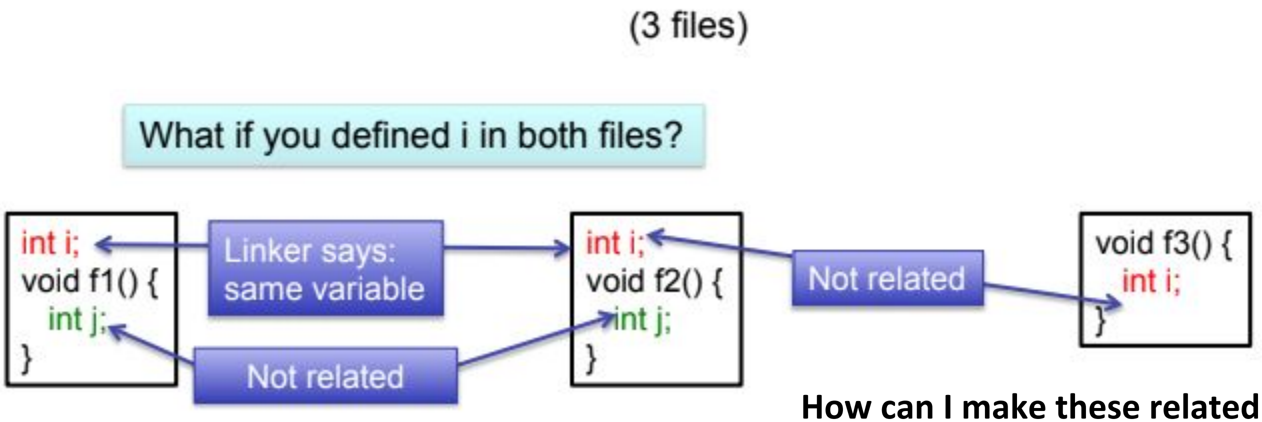
```c
void updatePaddlePosition(){
  int paddleSpeed = 1;
  prevPaddleX = paddleX;
  if (KEY_DOWN_NOW(KEY_LEFT)) {
    paddleX -= paddleSpeed;
  }
  if (KEY_DOWN_NOW(KEY_RIGHT)) {
    paddleX += paddleSpeed;
  }

  if (paddleX <= padding){
    paddleX = padding + 1;
  }
  if (paddleX + paddleSize >= (239 - padding)){
    paddleX = 239 - padding - paddleSize;
  }
}

void checkReset(){
  if (ballY > paddleY) {
    ballX = screenWidth / 2;
    ballY = screenHeight / 2;
  }
}
```

```
int main() {
  REG_DISPCNT = MODE3 | BG2_ENABLE;

  padding = 3;
  ballSize = 5;
  screenWidth = 240 - 2*padding;
  screenHeight = 160 - 2*padding;
  paddleSize = (239 - 2*padding) / 4;
  paddleX = ((screenWidth - paddleSize) / 2) + padding;
  paddleY = screenHeight + padding - 1;
  prevPaddleX = paddleX;
  ballX = screenWidth / 2;
  ballY = screenHeight / 2;
  ball_Vx = 1;
  ball_Vy = 2;

  drawHorizontalLine(padding, padding, screenWidth, RGB(31, 31, 31));
  drawVerticalLine(padding, padding, screenHeight, RGB(31, 31, 31));
  drawVerticalLine(screenWidth + padding - 1, padding, screenHeight, RGB(31, 31, 31));

  while (1) {
    updateBallPosition();
    updatePaddlePosition();
    checkReset();
    waitForVBlank();
    drawSquare(prevBallX, prevBallY, ballSize, RGB(0, 0, 0));
    drawSquare(ballX, ballY, ballSize, RGB(31, 0, 0));
    drawHorizontalLine(prevPaddleX, paddleY, paddleSize, RGB(0, 0, 0));
    drawHorizontalLine(paddleX, paddleY, paddleSize, RGB(0, 31, 31));
    time++;
  }
  return 0;
}
```

C Scopes Summary

- Global: Visible to everywhere in the program
- File: Not link-able. Only visible to code in the current ".c" file.
- Function: Only visible inside the function
- Block: Only visible inside the block (and sub-blocks)

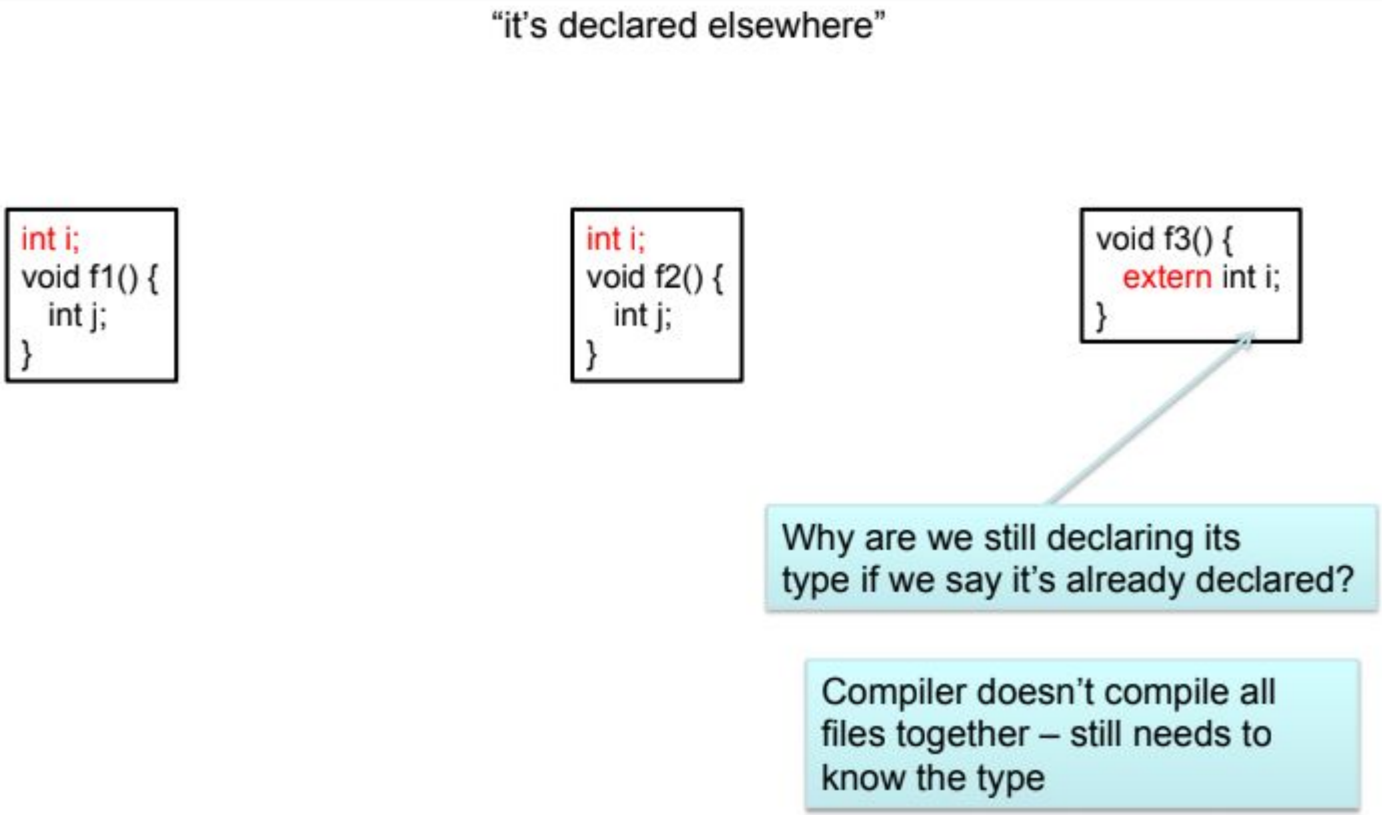
Basic Example



How can I make these related?

- The two globals are considered the same in this case.
  - * -- They have static storage duration
    - i.e. they exist "before" the program is run, until the end.
    - They're not typically called "static", because… (wait a couple slides)

The extern Keyword



“it’s declared elsewhere”

```
int i;
void f1() {
   int j;
}
```

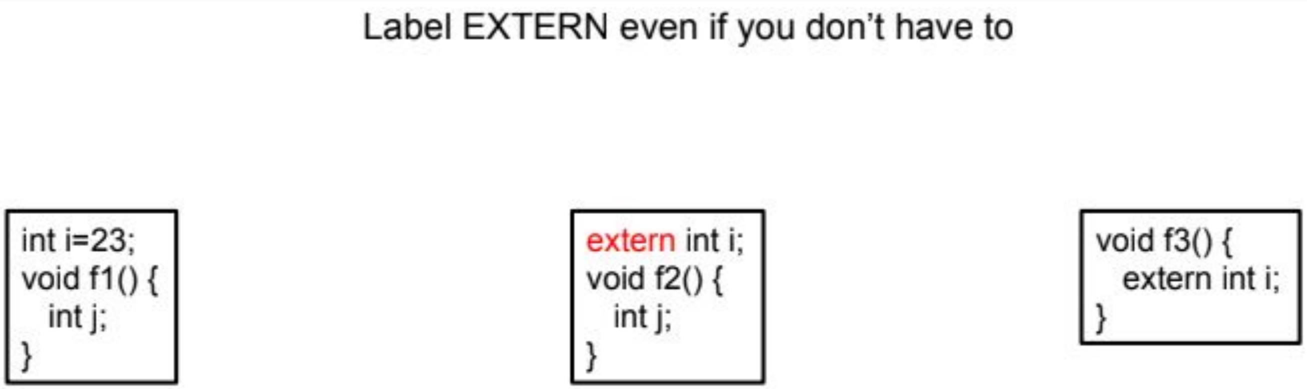
```
int i;
void f2() {
   int j;
}
```

```
void f3() {
   extern int i;
}
```

Why are we still declaring its type if we say it’s already declared?

Compiler doesn’t compile all files together – still needs to know the type

Be pedantic



Label EXTERN even if you don't have to

```
int i=23;
void f1() {
   int j;
}
```

```
extern int i;
void f2() {
   int j;
}
```

```
void f3() {
   extern int i;
}
```

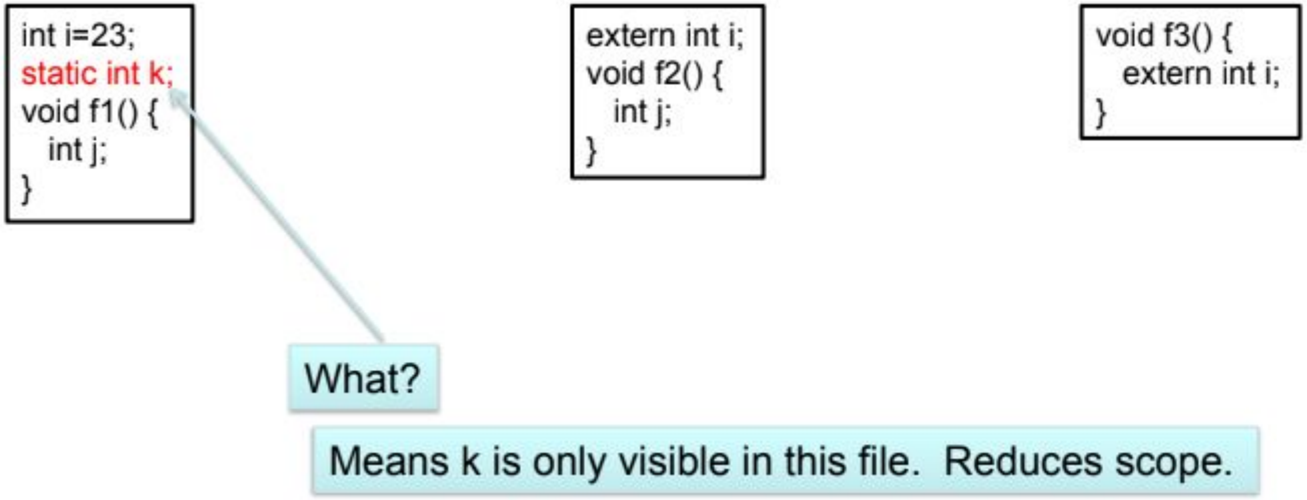
Static Variables

- Static?
  - Exist throughout program execution
  - Allocated before your program runs ("pre-allocated")
    - Not dynamically allocated / deallocated like automatic variables (local function variables)

- Static keyword is related (esp. in one case)
  - Comes in 3 basic flavors

The static keyword

On a "global" variable



```
int i=23;
static int k;
void f1() {
   int j;
}
```

```
extern int i;
void f2() {
   int j;
}
```

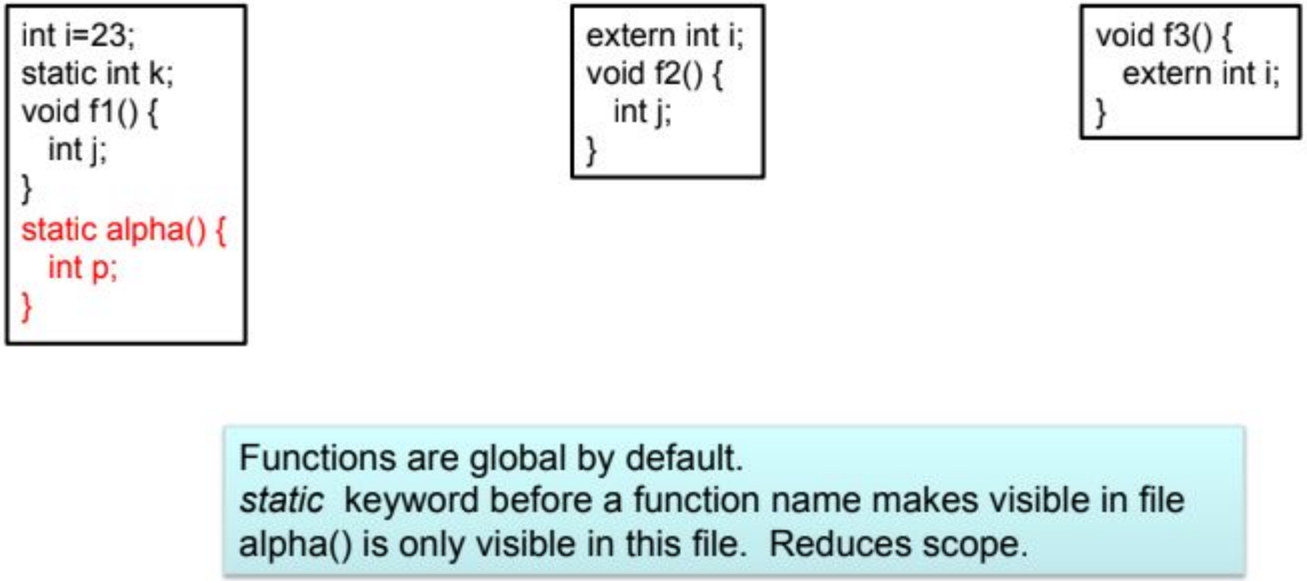
```
void f3() {
   extern int i;
}
```

What?

Means k is only visible in this file. Reduces scope.

The static keyword

On a "global" function



```
int i=23;
static int k;
void f1() {
   int j;
}
static alpha() {
   int p;
}
```

```
extern int i;
void f2() {
   int j;
}
```

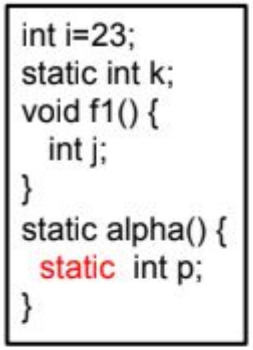
```
void f3() {
   extern int i;
}
```

Functions are global by default.
static keyword before a function name makes visible in file
alpha() is only visible in this file. Reduces scope.

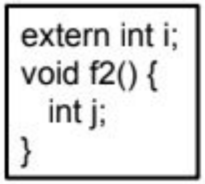
static on globals reduces them to file scope by disallowing linking to them from other files

The static keyword

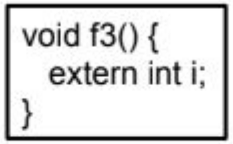
Within a function, on a "local" variable



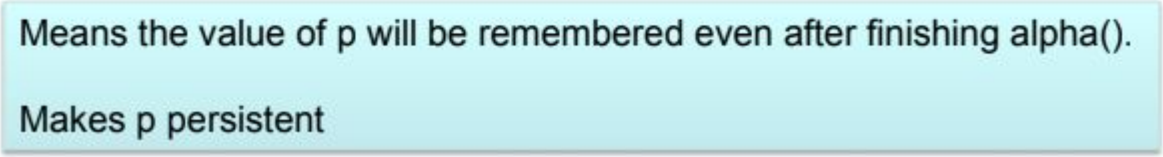
```
int i=23;
static int k;
void f1() {
   int j;
}
static alpha() {
  static  int p;
}
```



```
extern int i;
void f2() {
   int j;
}
```



```
void f3() {
   extern int i;
}
```



Means the value of p will be remembered even after finishing alpha().

Makes p persistent

- This allows `alpha()` to keep state across many calls (it could count how many times it's been called, for example).
  - Note: p now has static storage duration. It's just only accessible by name within the function alpha()

C Scopes Summary

- Preallocated / static
- Dynamically Allocated

- Global: Visible to everywhere in the program
  - anything outside main(), by default
- File: Not link-able. Only visible to code in the current ".c" file.
  - static vars / functions outside main()
- Function: Only visible inside the function
  - vars inside functions -- including main()
  - functions inside functions
  - static things inside a function
- Block: Only visible inside the block (and sub-blocks)

```
void foo(int bar){
  int i = 1;
  if (bar % 2) {
    int i = 2;
    printf("%d", i);  // prints 2 -- it's a different i than the function-level one.
  }
  printf("%d", i);  // prints 1
}
```

Stacks vs Queues

- Stacks
  - Last in, first out (LIFO)
  - Code execution works as a stack, where each function can call functions inside of it. Those inner functions have to finish before the outer function can.

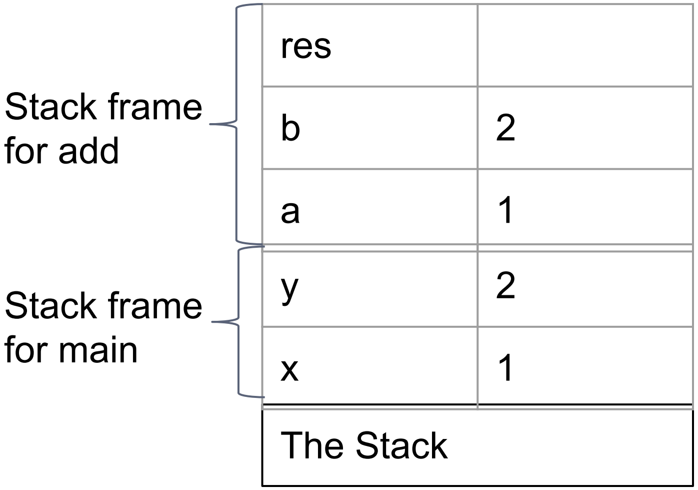
- Queues
  - First in, first out (FIFO)
  - This is typically how we process lists of data (especially streaming data).

Simplified Call Stack

Every function call adds to the call stack.

```
int add(int a, int b){
   int res = a + b;
   return res
}

int main() {
   int x = 1;
   int y = 2;
   return add(x, y);
}
```

Note: The stack frame has more machinery than this



Stack frame for add

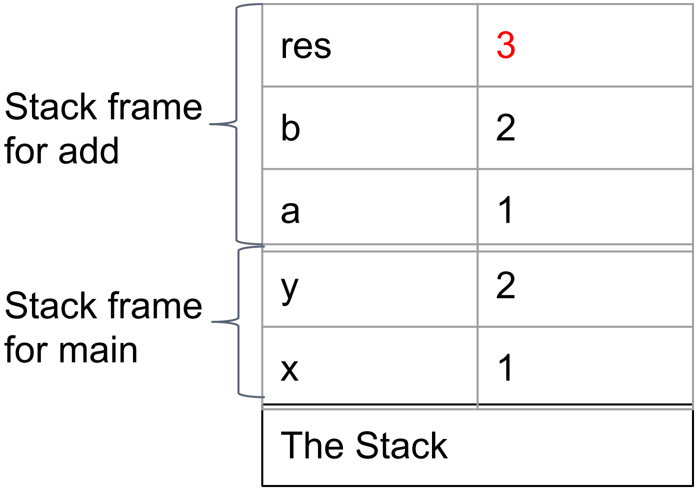
Stack frame for main

res	
b	2
a	1
y	2
x	1
The Stack	

Simplified Call Stack

Every function call adds to the call stack.

```
int add(int a, int b){
  int res = a + b;
   return res
}

int main() {
   int x = 1;
   int y = 2;
   return add(x, y);
}
```



Stack frame for add

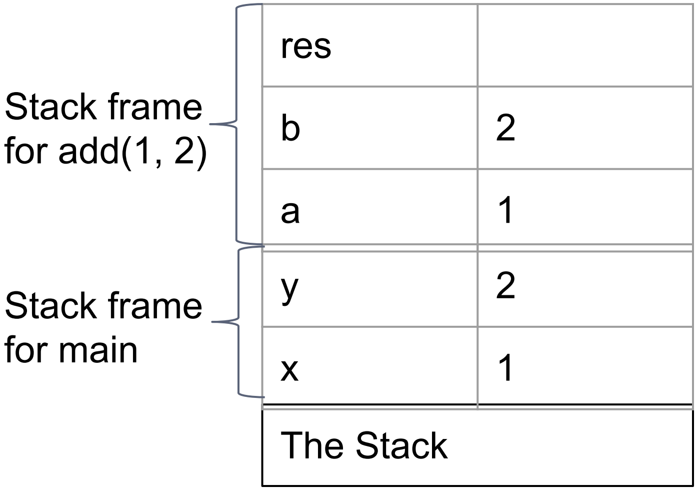
Stack frame for main

res	3
b	2
a	1
y	2
x	1
The Stack	

Simplified Call Stack

Every function call adds to the call stack.

```
int add(int a, int b){
  int res = a + b;
   return res
}

int main() {
   int x = 1;
   int y = 2;
   return add(x, add(x, y));
}
```



Stack frame for add(1, 2)

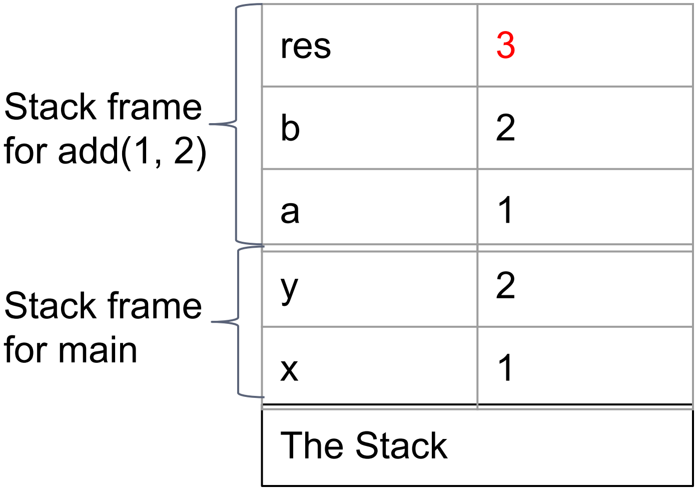
Stack frame for main

res	
b	2
a	1
y	2
x	1
The Stack	

Simplified Call Stack

Every function call adds to the call stack.

```
int add(int a, int b){
  int res = a + b;
   return res
}

int main() {
   int x = 1;
   int y = 2;
   return add(x, add(x, y));
}
```



Stack frame for add(1, 2)

Stack frame for main

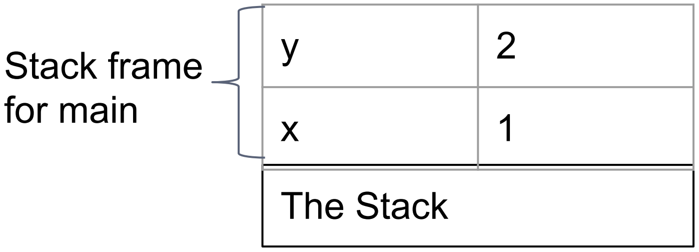
res	3
b	2
a	1
y	2
x	1
The Stack	

Simplified Call Stack

Stacks pop off when they resolve

```
int add(int a, int b){
  int res = a + b;
   return res
}

int main() {
   int x = 1;
   int y = 2;
   return add(x, add(x, y));
}
```

3 is temporarily stored somewhere hardware-dependent, typically a register



Stack frame for main

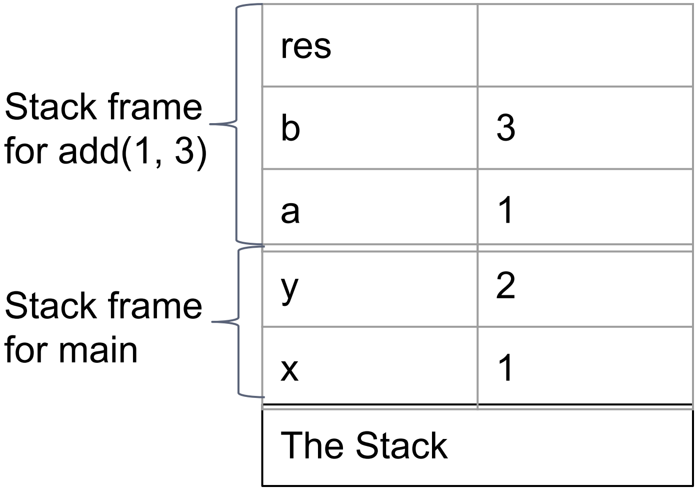
y	2
x	1

The Stack

Simplified Call Stack

New stack frame for second call.

```
int add(int a, int b){
  int res = a + b;
   return res
}

int main() {
   int x = 1;
   int y = 2;
   return add(x, add(x, y));
}
```

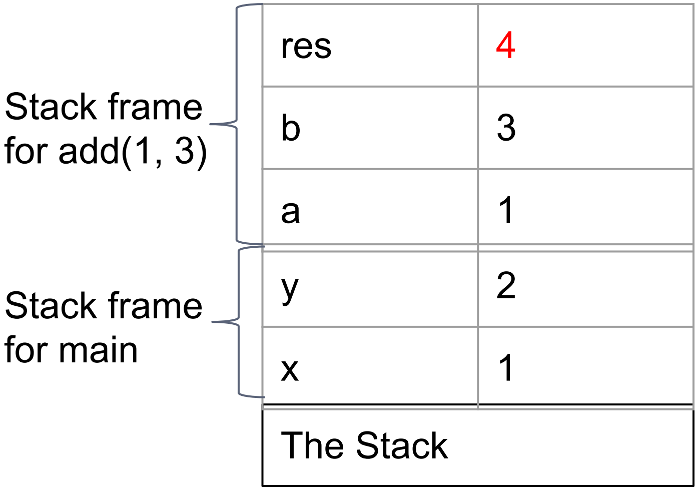


Stack frame for add(1, 3)	res	
	b	3
	a	1
Stack frame for main	y	2
	x	1
	The Stack	

Simplified Call Stack

Every function call adds to the call stack.

```
int add(int a, int b){
   int res = a + b;
   return res
}

int main() {
   int x = 1;
   int y = 2;
   return add(x, add(x, y));
}
```



Stack frame for add(1, 3)

Stack frame for main

res	4
b	3
a	1
y	2
x	1
The Stack	

Suppose...

Lecture ended here! The rest is in the next lecture as well.

- We need the function max(a,b)

- We need it for several different types
  - ints
  - floats
  - unsigned
  - etc,

- Should we write a function for each?
  - int max(int a, int b)
  - float max(float a, float b)
  - etc.

The Macro Solution

- We can write a single macro which will work for different types!

```
#define max(a, b) a >= b ? a : b
int x = 7;
int y = 8;
float p = 78.6;
float q = 29.2;
```

A Macro Gotcha

```
#define SQUARE(x) ((x)*(x))

int x = 2;
int z = SQUARE(x++);
```

What’s the correct answer?

```
z = x^2                 // 4
z = (x + 1)^2           // 9
z = x * (x + 1)         // 6
z = (x + 1) * (x + 2)   // 12
```

Even more importantly what has happened to the value of x???

A Macro Gotcha

```
#define SQUARE(x) ((x)*(x))

int x = 2;
int z = SQUARE(x++);
```

What’s the correct answer?

```
z = x^2               // 4
z = (x + 1)^2         // 9
z = x * (x + 1)       // 6
z = (x + 1) * (x + 2) // 12
```

Even more importantly what has happened to the value of x???

What happens?

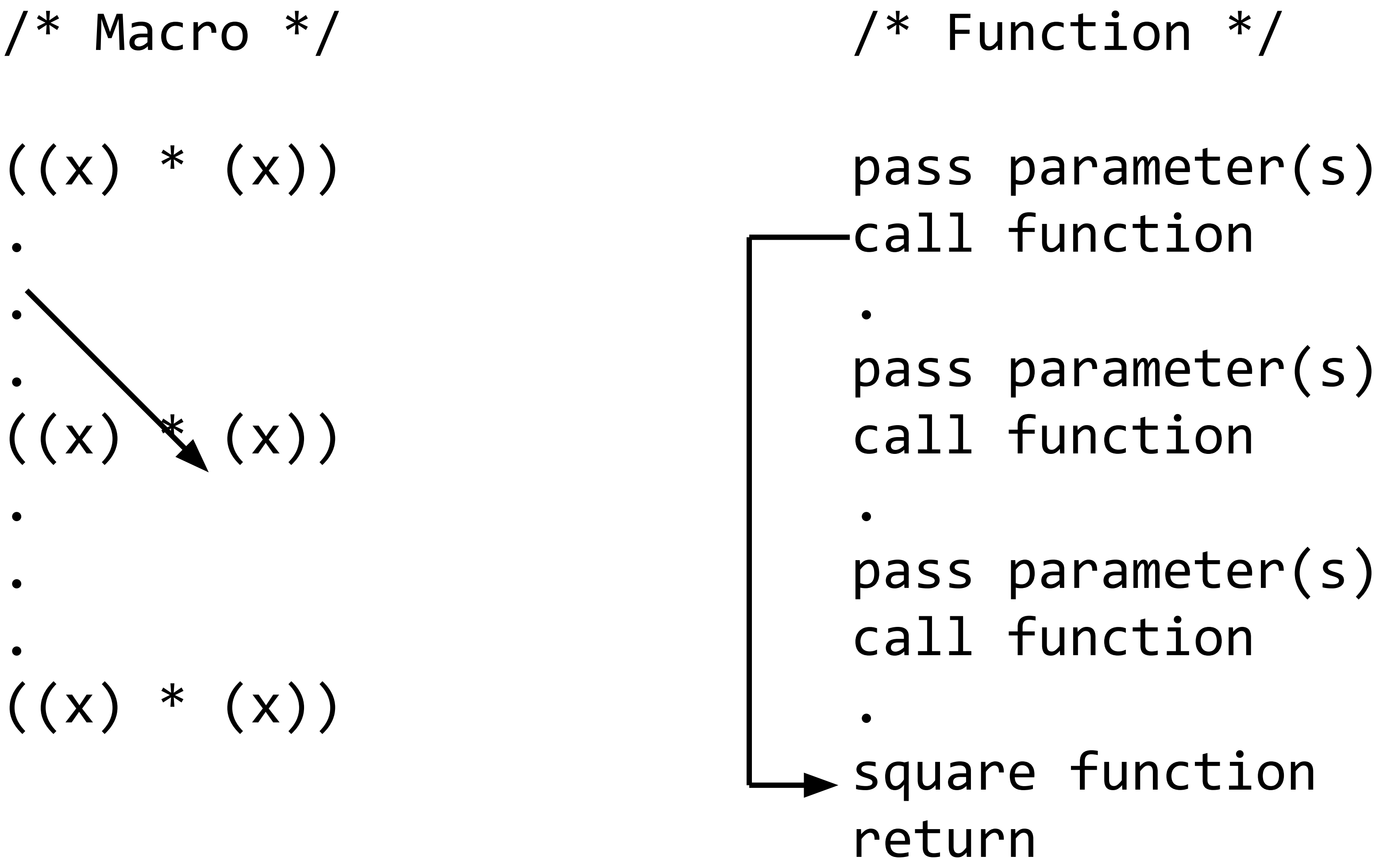
```
/* Macro */

SQUARE(x)
.
.
.
SQUARE(x)
.
.
.
SQUARE(x)
```

```
/* Function */

square(x)
.
.
.
square(x)
.
.
.
square(x)
```

What happens?

```
/* Macro */

((x) * (x))
.
.
.
((x) * (x))
.
.
.
((x) * (x))
```

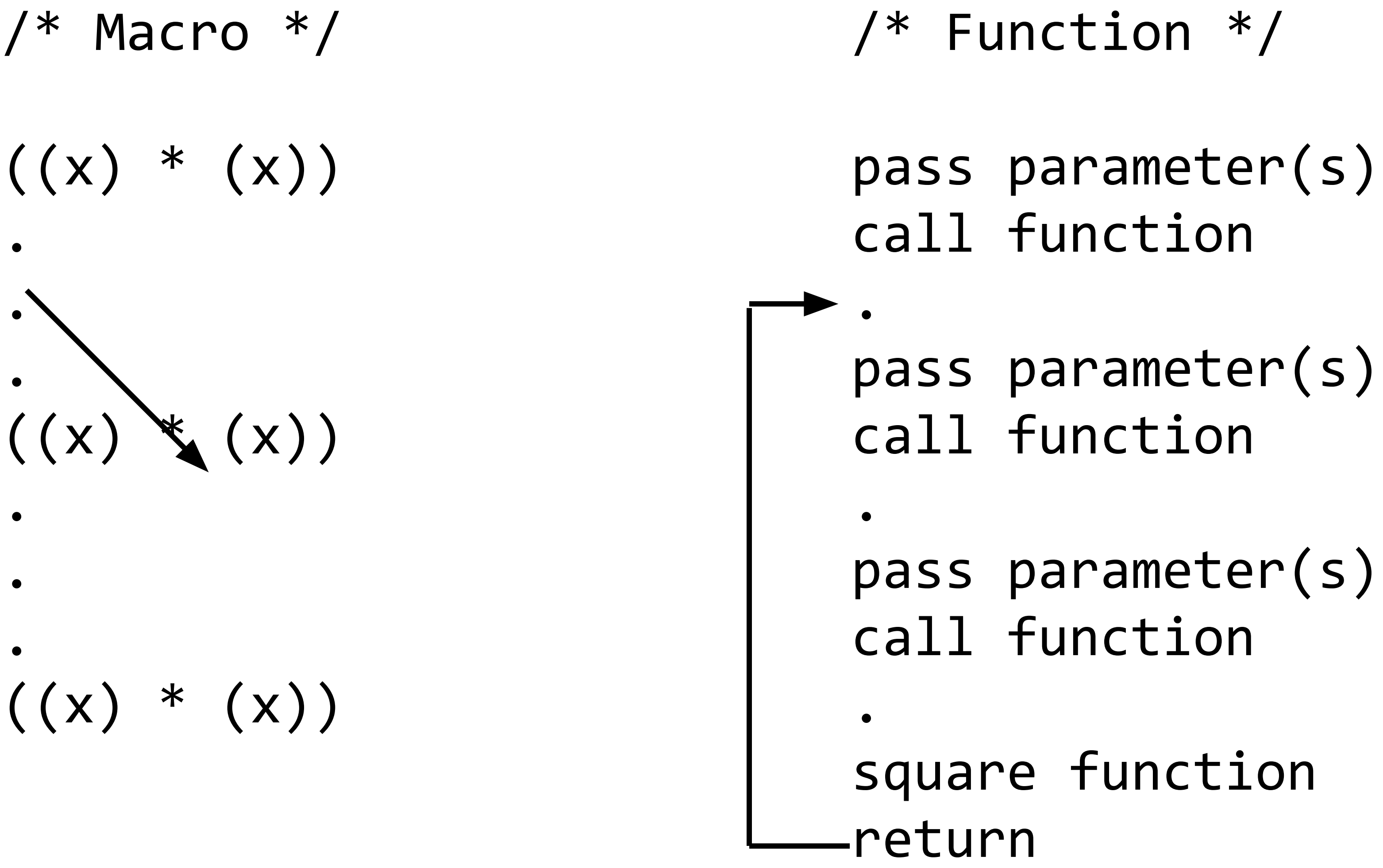
What happens?



```
/* Macro */

((x) * (x))
.
.
.
((x) * (x))
.
.
.
((x) * (x))
```

```
/* Function */

pass parameter(s)
call function
.
pass parameter(s)
call function
.
pass parameter(s)
call function
.
square function
return
```

What happens?



```
/* Macro */

((x) * (x))
.
.
.
((x) * (x))
.
.
.
((x) * (x))
```

```
/* Function */

pass parameter(s)
call function
.
pass parameter(s)
call function
.
pass parameter(s)
call function
.
square function
return
```

Macros vs. Functions

- Macros
  - Text substitution at Translation (compile) time
  - May have problems: e.g. square(x++)
  - Will work with different types due to operator overloading
    - floats, doubles, ints, …
  - Difficult to implement if complex
  - [illegible]

- Functions
  - Separate piece of code
  - Overhead of passing arguments and returning results via stack
  - Fixes ambiguity problems: e.g. square(x + y) or(x++)
  - Function optimizes for space. Why?

Macros vs Functions

- If the goal is not clearly optimization for speed or time but rather somewhere in-between it’s difficult to know exactly which choice is correct.

- In any event: Don’t try and outwit the compiler!

- A better algorithm is more of an improvement that trying to write tricky code!!!