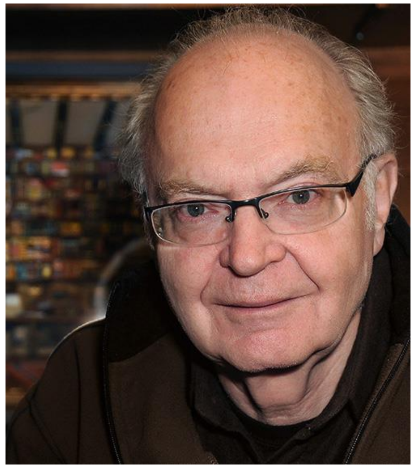
CMPSC 311 - Introduction to Systems Programming

Introduction to C

Professor Abutalib Aghayev

(Slides are mostly by Professor Patrick McDaniel)

But first, emacs or IDE?



Emacs users

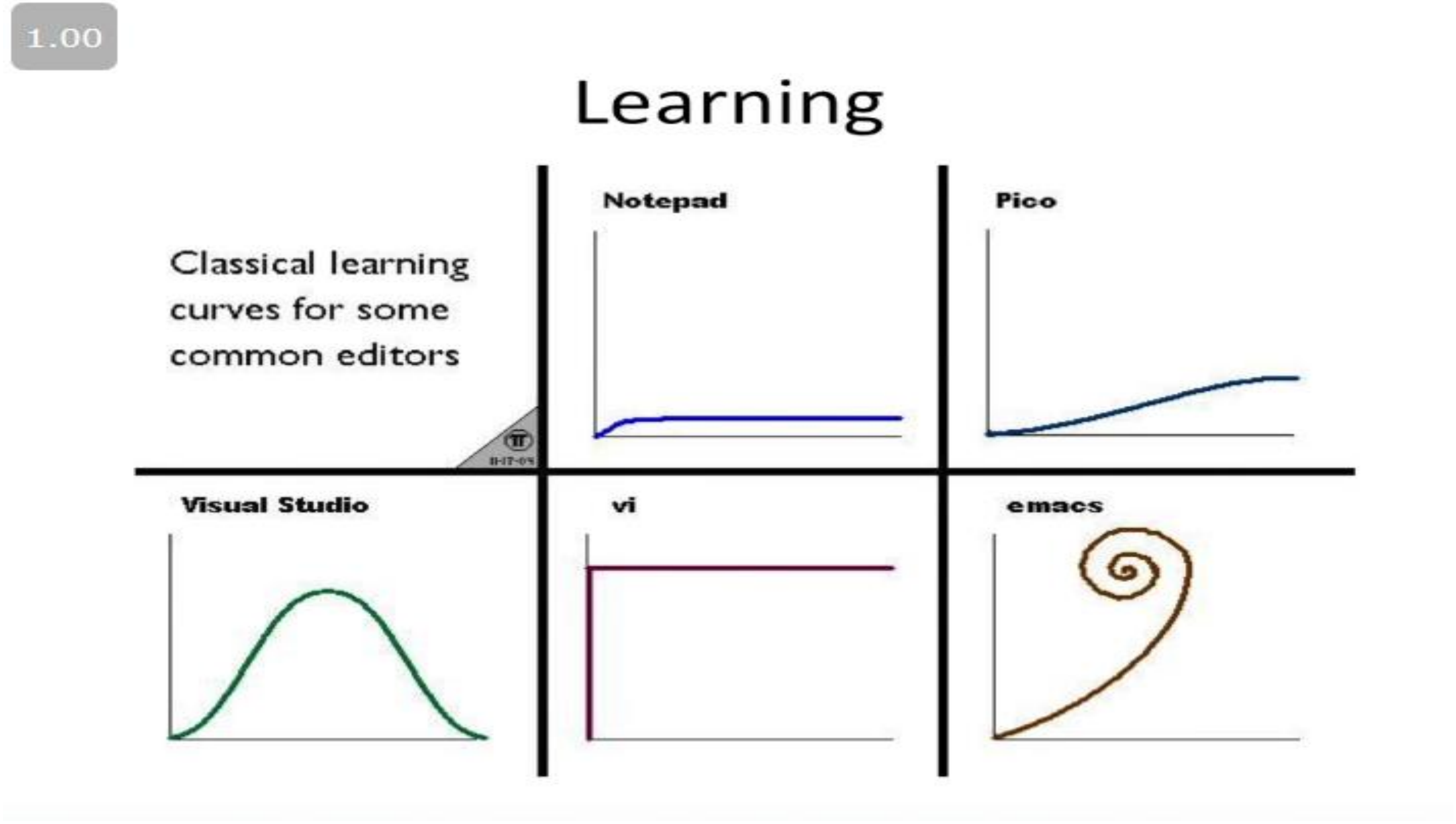
Visual Studio user

Emacs in the movies..

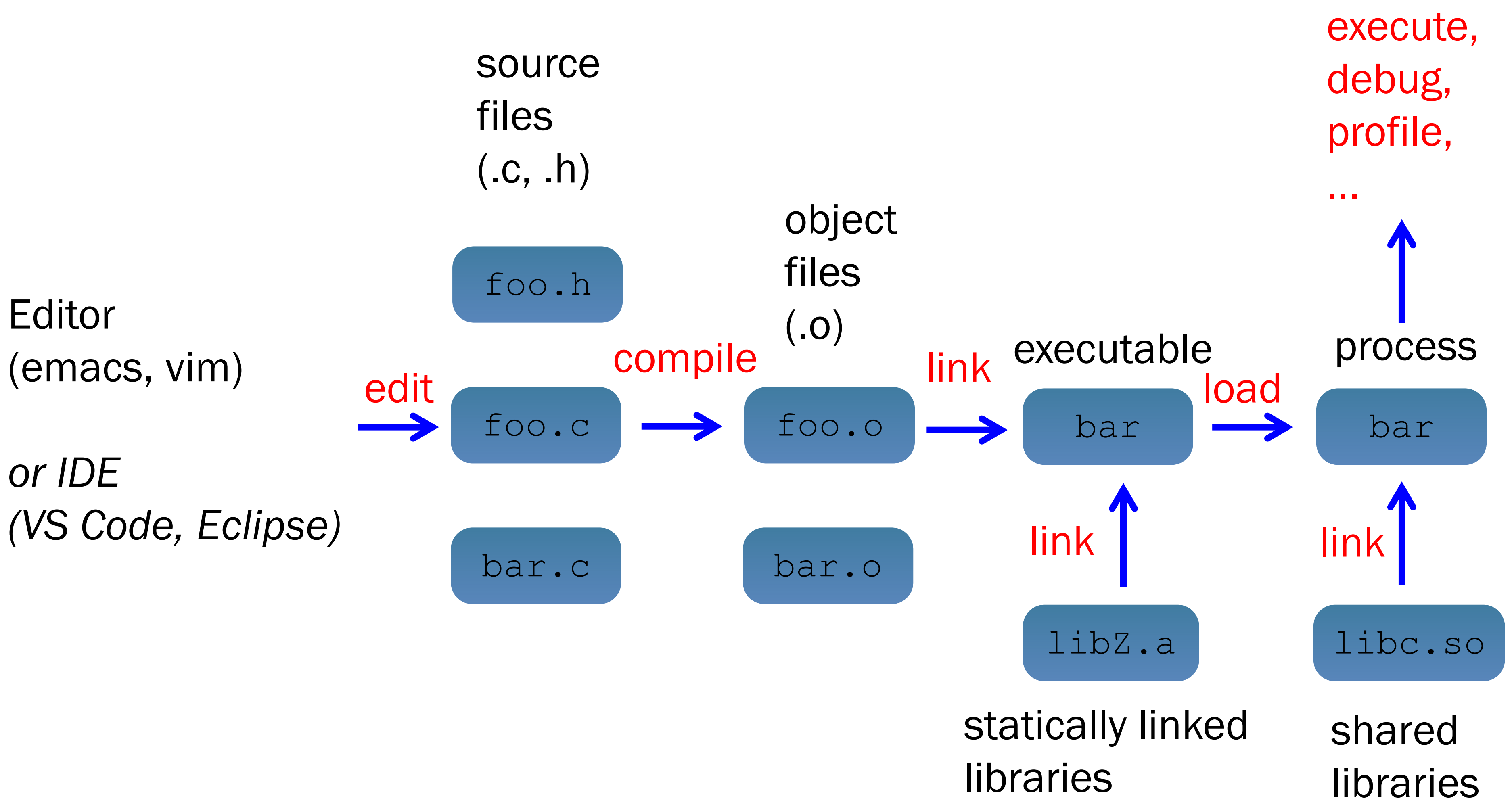
Emacs in the movies..

What makes emacs powerful?

- It's programmable with a full-fledged functional programming language, LISP



C workflow


Editor
(emacs, vim)
or IDE
(VS Code, Eclipse)
edit
source files (.c, .h)
foo.h
foo.c
bar.c
compile
object files (.o)
foo.o
bar.o
link
executable
bar
link
libZ.a
statically linked libraries
load
process
bar
link
libc.so
shared libraries
execute, debug, profile, …

Defining a function

returnType name(type name, ..., type name)

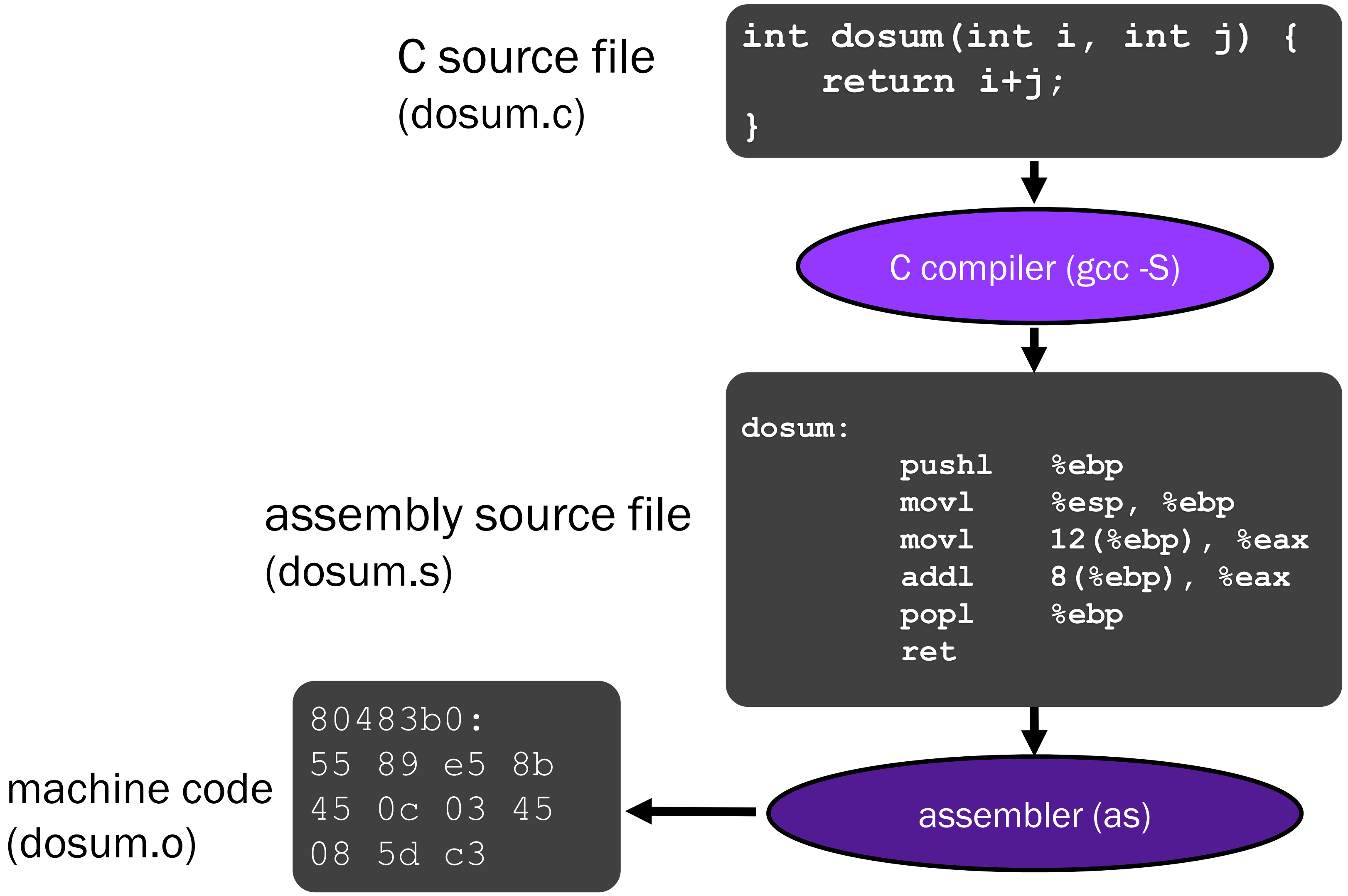
{

statements;

}

```
// sum integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i=1; i<=max; i++) {
    sum += i;
  }
  return sum;
}
```

From C to machine code



C source file
(dosum.c)

```
int dosum(int i, int j) {
    return i+j;
}
```

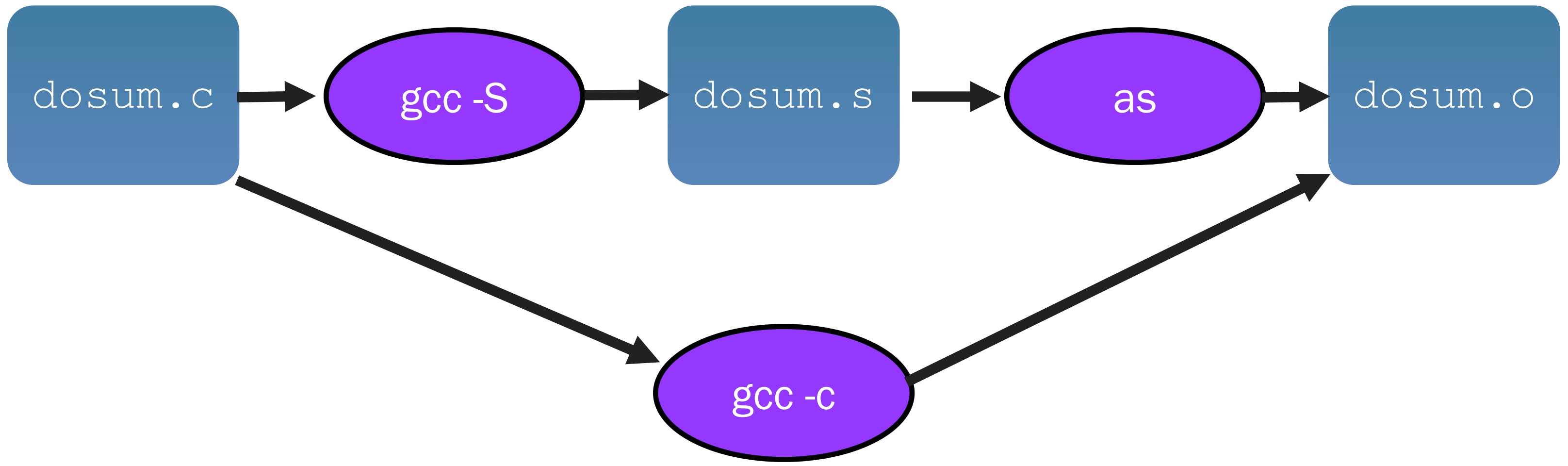
assembly source file
(dosum.s)

```
dosum:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        popl    %ebp
        ret
```

machine code
(dosum.o)

```
80483b0:
55 89 e5 8b
45 0c 03 45
08 5d c3
```

- Most C compilers generate object “.o" files directly
  - i.e., without actually saving the readable .s assembly file



Note: *Object code* is is re-locatable machine code, but generally cannot be executed without some manipulation (e.g., via a linker)

Anatomy of a C program …

```c
#include <stdio.h>

int myfunc(int i) {
    printf("Got into function with %d\n", i);
    return 0;
}

int main(void) {
    myfunc(10);
    return 0;
}
```

All C programs start with the “main()” function …

Anatomy of a C program …

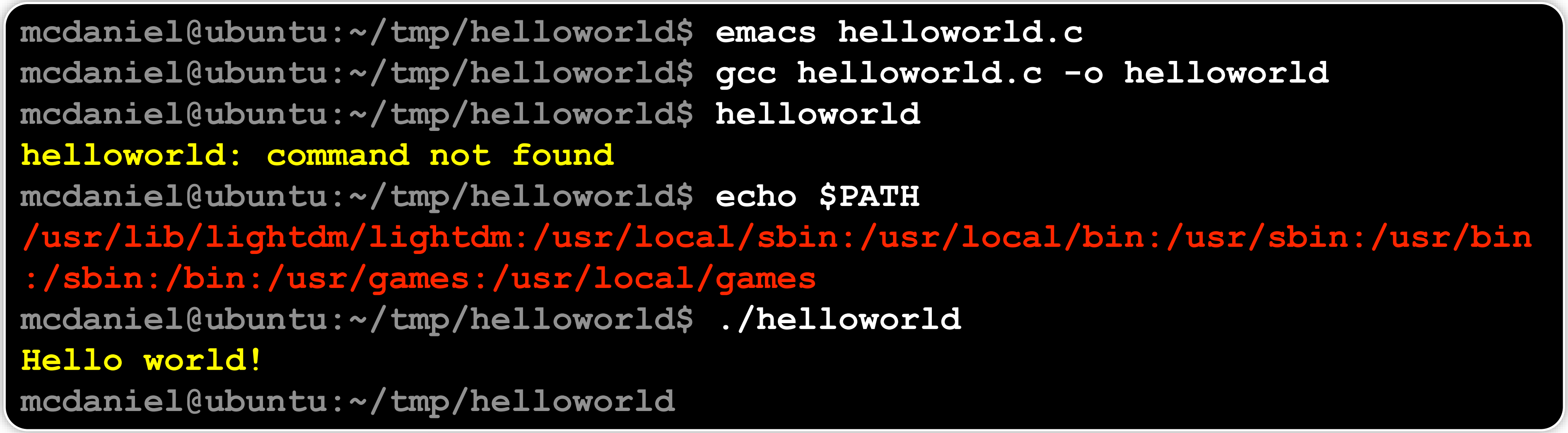
```c
#include <stdio.h>

int myfunc(int i) {
    printf("Got into function with %d\n", i);
    return 0;
}

int main(void) {
    myfunc(10);
    return 0;
}
```

Compile and link

Running the program

```
% gcc -g -Wall main.c -o main
% ./main
Got into function with 10
%
```

Running a program



```
mcdaniel@ubuntu:~/tmp/helloworld$ emacs helloworld.c
mcdaniel@ubuntu:~/tmp/helloworld$ gcc helloworld.c -o helloworld
mcdaniel@ubuntu:~/tmp/helloworld$ helloworld
helloworld: command not found
mcdaniel@ubuntu:~/tmp/helloworld$ echo $PATH
/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
:/sbin:/bin:/usr/games:/usr/local/games
mcdaniel@ubuntu:~/tmp/helloworld$ ./helloworld
Hello world!
mcdaniel@ubuntu:~/tmp/helloworld
```

- UNIX looks for a program in all of the directories listed by the `PATH` environment variable, or locally of prepended by "`./`"
  - to add to search path just add more "`:`" separated paths,

```
export PATH=$PATH:/new/path
```

Multi-file C programs

C source file
(dosum.c)

```
int dosum(int i, int j) {
    return i+j;
}
```

this “prototype” of dosum( ) tells gcc about the types of dosum’s arguments and its return value

C source file
(sumnum.c)

```
#include <stdio.h>

int dosum(int i, int j);

int main(int argc, char **argv)
{
  printf("%d\n", dosum(1,2));
  return 0;
}
```

dosum( ) is implemented in dosum.c

Multi-file C programs

C source file
(dosum.c)

```c
int dosum(int i, int j) {
    return i+j;
}
```

C source file
(sumnum.c)

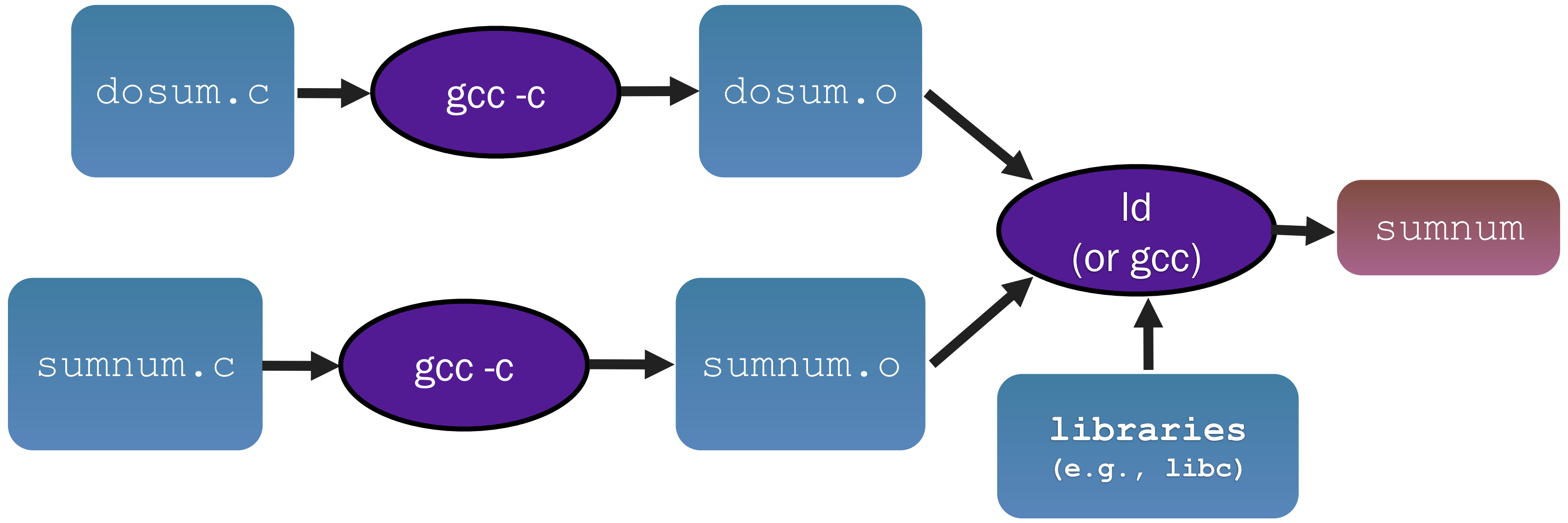
```c
#include <stdio.h>

int dosum(int i, int j);

int main(int argc, char **argv)
{
  printf("%d\n", dosum(1,2));
  return 0;
}
```

why do we need this #include?

where is the implementation of printf?

Compiling multi-file programs

- Multiple object files are *linked* to produce an executable
  - standard libraries (libc, crt1, …) are usually also linked in
  - a library is just a pre-assembled collection of .o files



Object files revisited …

- sumnum.o, dosum.o are object files
  - each contains machine code produced by the compiler
  - each might contain references to external symbols
    - variables and functions not defined in the associated .c file
    - e.g., `sumnum.o` contains code that relies on `printf()` and `dosum()`, but these are defined in `libc.a` and `dosum.o`, respectively
  - linking resolves these external symbols while smooshing together object files and libraries

Lets dive into C itself

- Things that are the same as Java
- syntax for statements, control structures, function calls
- types: int, double, char, long, float
- type-casting syntax: float x = (float) 5 / 3;
- expressions, operators, precedence

  + - * / % ++ -- = += -= *= /= %= < <= == != > >= && || !

- scope (local scope is within a set of { } braces)
- comments: /* comment */ or // comment *to EOL*

Primitive types in C

- integer types
  - `char, int`
- floating point
  - `float, double`
- modifiers
  - `short [int]`
  - `long [int, double]`
  - `signed [char, int]`
  - `unsigned [char, int]`

type	bytes (32-bit)	bytes (64-bit)	32-bit range	printf
char	1	1	[0, 255]	%c
short int	2	2	[-32768,32767]	%hd
unsigned short int	2	2	[0, 65535]	%hu
int	4	4	[-214748648, 2147483647]	%d
unsigned int	4	4	[0, 4294967295]	%u
long int	4	8	[-2147483648, 2147483647]	%ld
long long int	8	8	[-9223372036854775808, 9223372036854775807]	%lld
float	4	4	approx [10^{-38}, 10^{38}]	%f
double	8	8	approx [10^{-308}, 10^{308}]	%lf
long double	12	16	approx [10^{-4932}, 10^{4932}]	%Lf
pointer	4	8	[0, 4294967295]	%p

C99 extended integer types

- Solve the conundrum of “how big is a long int?”

```
#include <stdint.h>

void foo(void) {
  int8_t  w;    // exactly 8 bits, signed
  int16_t x;    // exactly 16 bits, signed
  int32_t y;    // exactly 32 bits, signed
  int64_t z;    // exactly 64 bits, signed

  uint8_t w;    // exactly 8 bits, unsigned
  ...etc.
}
```

Similar to Java...

- variables
  - must declare at the start of a function or block (not required since in C99)
  - need not be initialized before use (gcc -Wall will warn); ALWAYS INITIALIZE YOUR VARS

```
#include <stdio.h>

int main(void) {
  int x, y = 5;    // note x is uninitialized!
  long z = x+y;

  printf("z is '%ld'\n", z); // what's printed?
  {
    int y = 10;
    printf("y is '%d'\n", y);
  }
  int w = 20;  // ok in c99
  printf("y is '%d', w is '%d'\n", y, w);
  return 0;
}
```

Similar to Java…

- `const`
  - a qualifier that indicates the variable's value cannot change
  - compiler will issue an error if you try to violate this
  - why is this qualifier useful?

```
#include <stdio.h>

int main(void) {
  const double MAX_GPA = 4.0;

  printf("MAX_GPA: %g\n", MAX_GPA);
  MAX_GPA = 5.0;  // illegal!
  return 0;
}
```

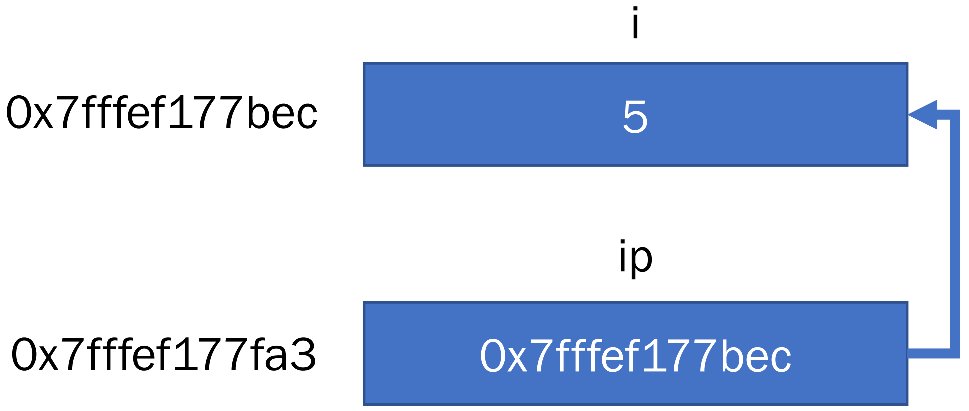
Similar to Java...

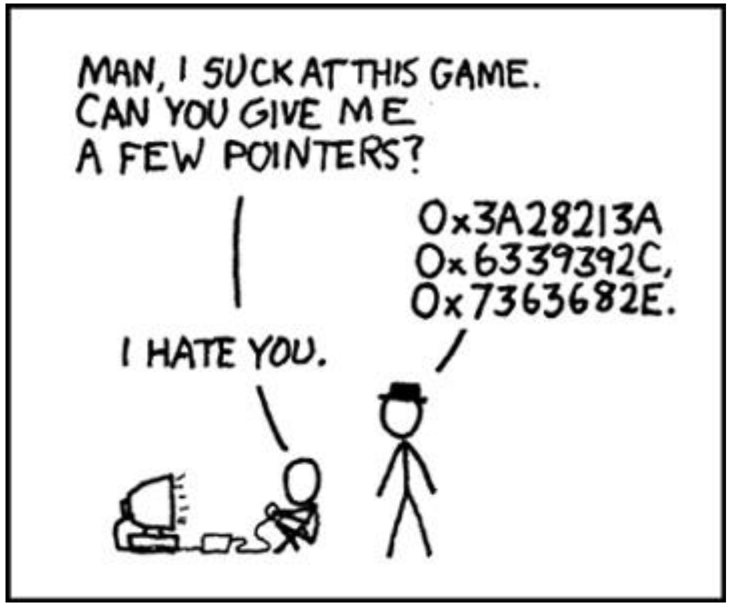
- `for` loops
  - can't declare variables in the loop header (changed in c99)
- `if/else`, `while`, and `do while` loops
  - no boolean type (changed in c99: `#include <stdbool.h>`)
  - any type can be used; 0 means **false**, everything else **true**

```c
int i;

for (i=0; i < 100; i++) {
  if (i % 10 == 0) {
    printf("i: %d\n", i);
  }
}
```

Pointers

```c
#include <stdio.h>

int main(void) {
  int i = 5;
  int *ip = &i;

  printf("%d\n", i);
  printf("%p\n", ip);
  *ip = 42;
  printf("%d\n", i);
  printf("%d\n", *ip);
}
```





MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

Key concepts:

- Taking address of a variable: &
- Dereferencing a pointer: *
- Aliasing: *ip is an alias for i

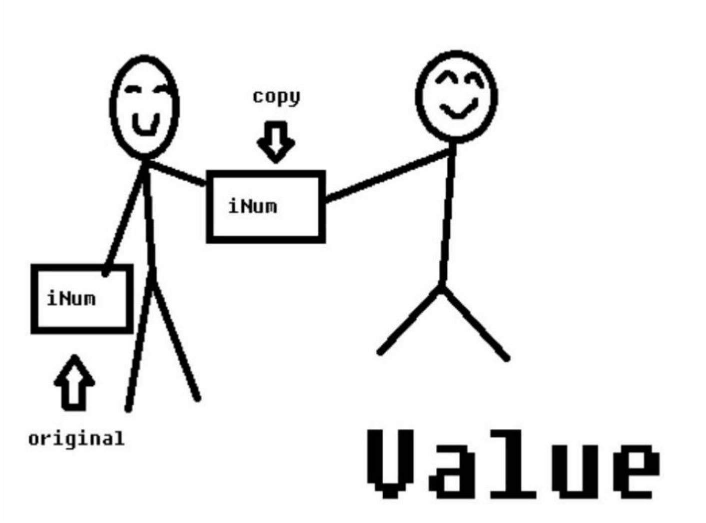
Similar to Java

- C always passes arguments by value
  - value is "copied" into function
  - any local modification change is not reflecting in original value passed
- pointers let you pass by reference
  - pass "memory location" of variable
  - more on these soon
  - least intuitive part of C
  - very dangerous part of C

```
void add_pbv(int c) {
  c += 10;
  printf("pbv c: %d\n", c);
}

void add_pbr(int *c) {
  *c += 10;
  printf("pbr *c: %d\n", *c);
}

int main(void) {
  int x = 1;
  printf("x: %d\n", x);
  add_pbv(x);
  printf("x: %d\n", x);
  add_pbr(&x);
  printf("x: %d\n", x);
  return 0;
}
```

Pass-by-value

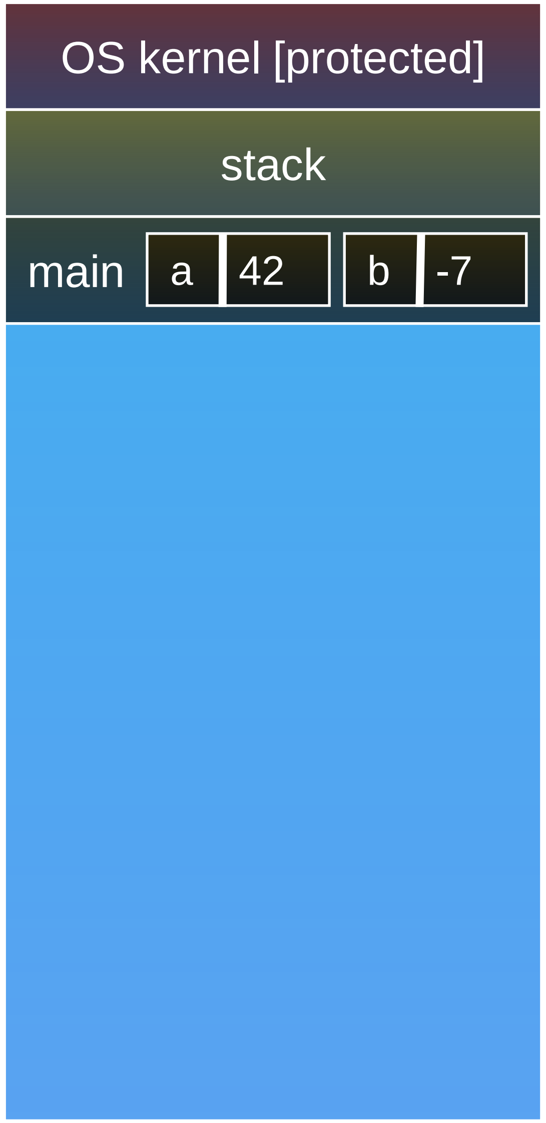
- C passes arguments by value
  - callee receives a copy of the argument



```c
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(a, b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

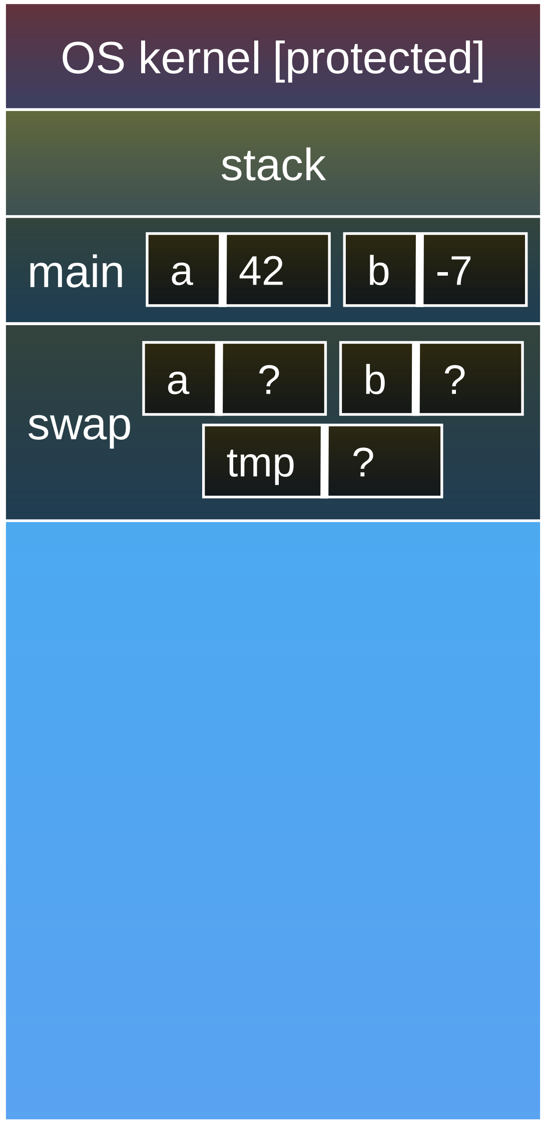
  - if the callee (function that is called) modifies an argument, caller's copy isn't modified

Pass-by-value



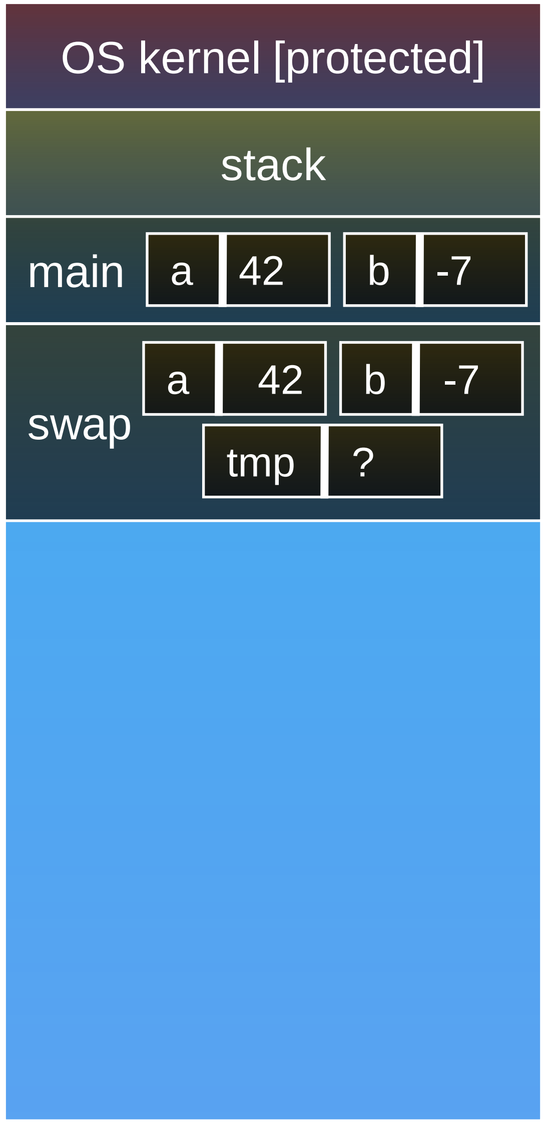
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(a, b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-value



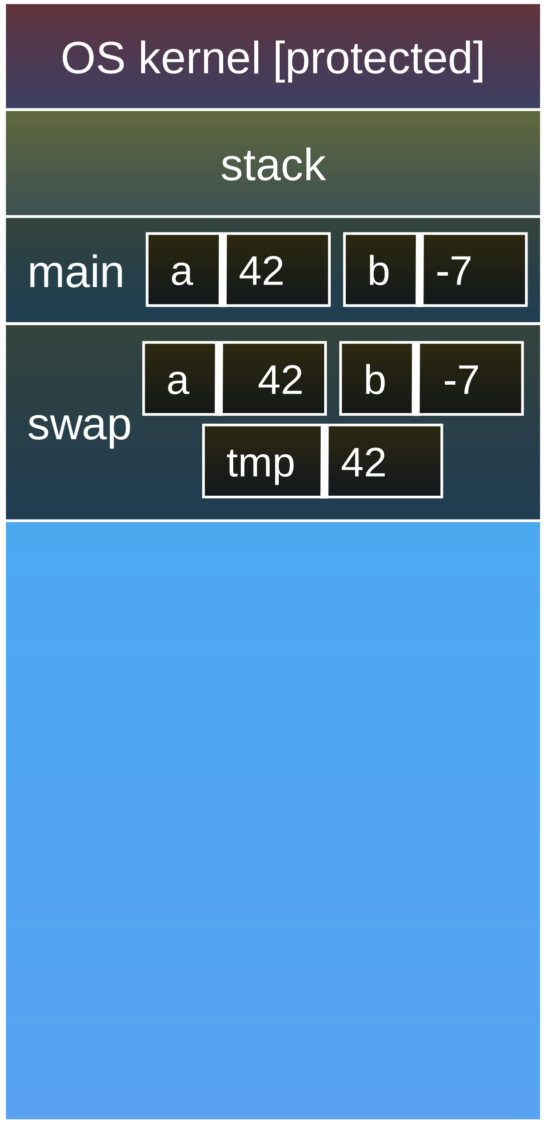
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(a, b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-value



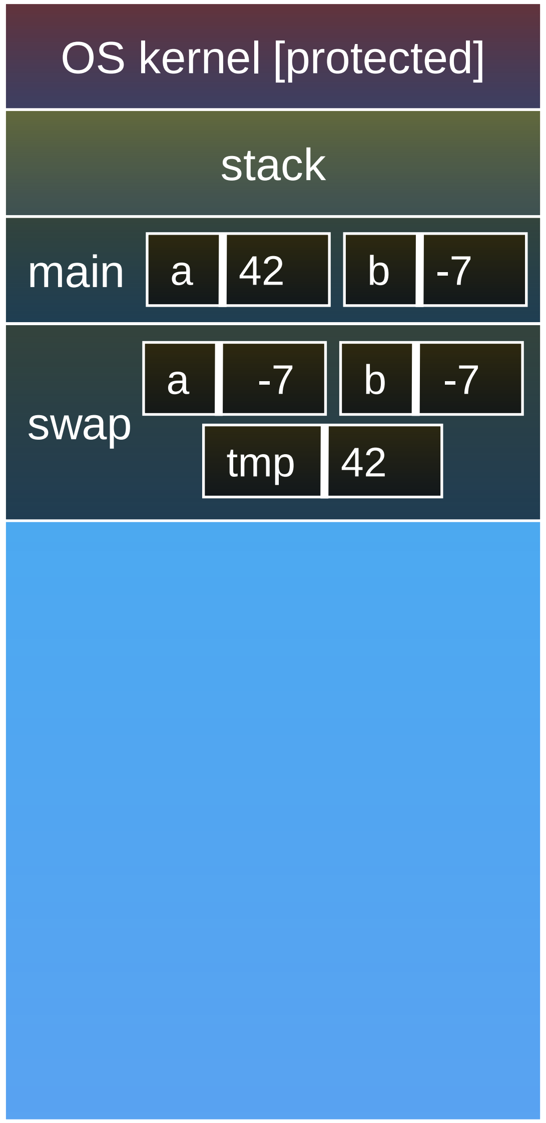
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(a, b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-value



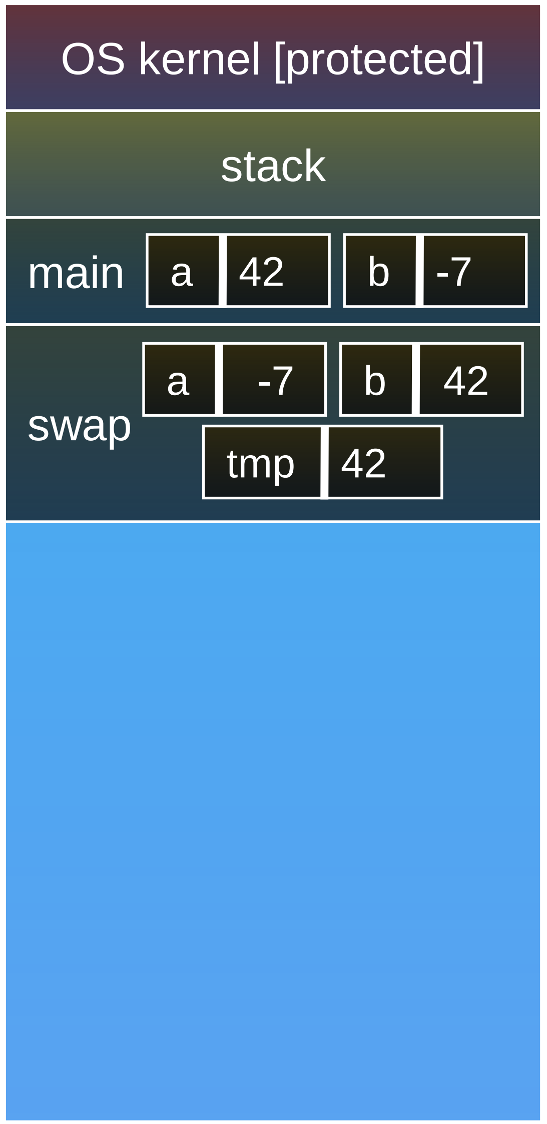
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(a, b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-value



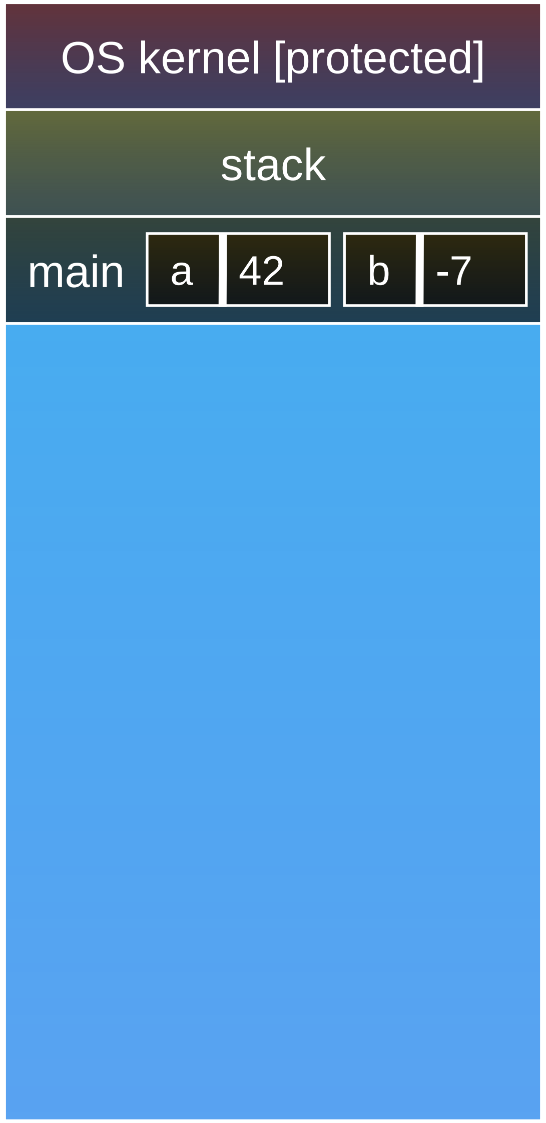
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(a, b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-value



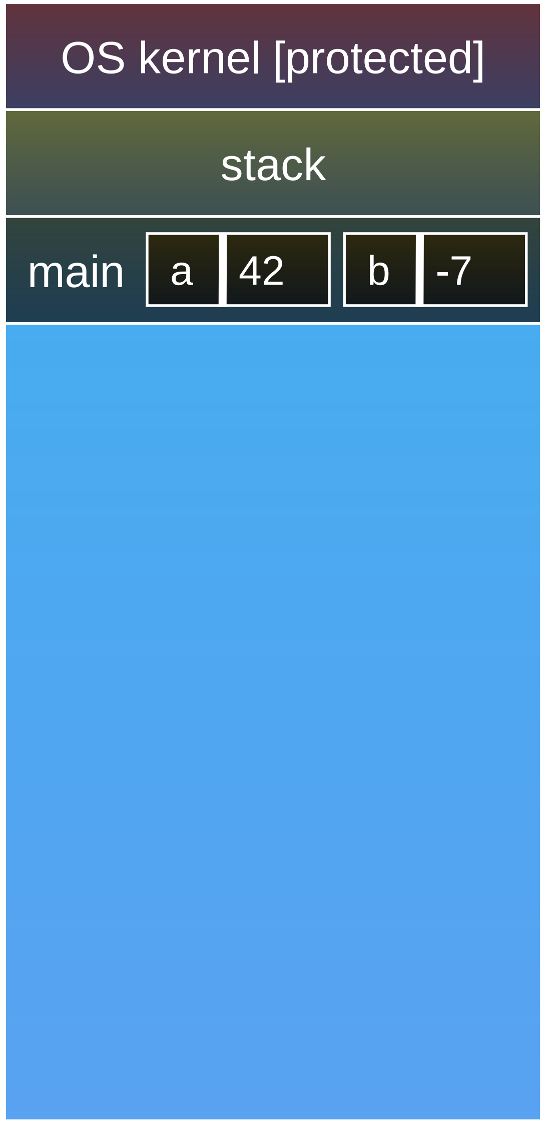
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(a, b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-value



```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(a, b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

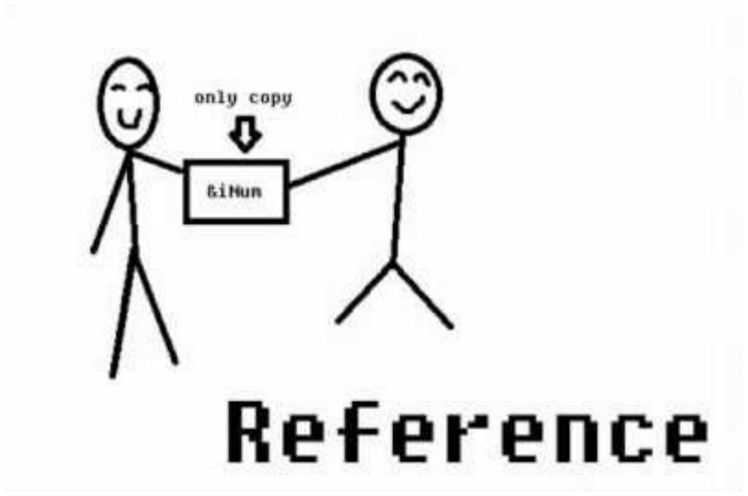
Pass-by-value



```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(a, b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-reference

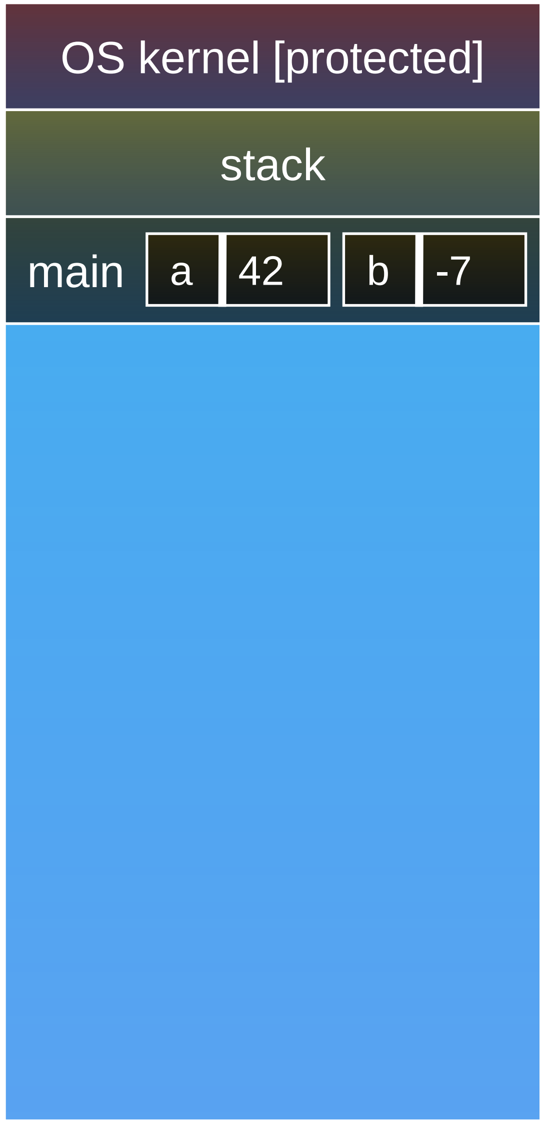
- You can use pointers to pass by reference
  - callee still receives a copy of the argument
  - but, the argument is a *pointer**
  - the pointer's value points-to the variable in the scope of the caller
  - this gives the callee a way to modify a variable that's in the scope of the caller



```c
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(&a, &b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

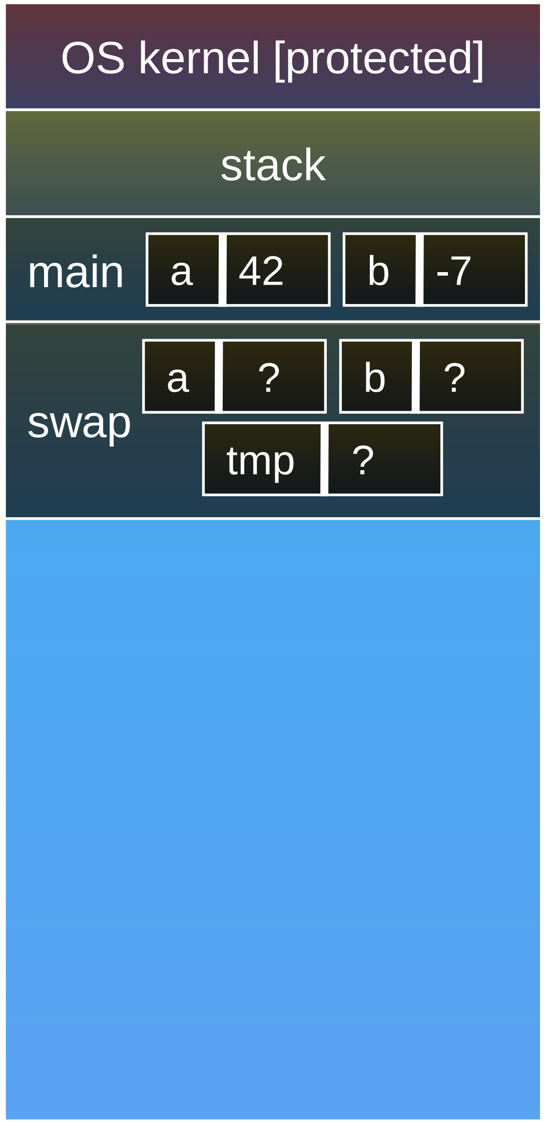
* The key to C (and languages like it) is getting good at using pointers.

Pass-by-reference



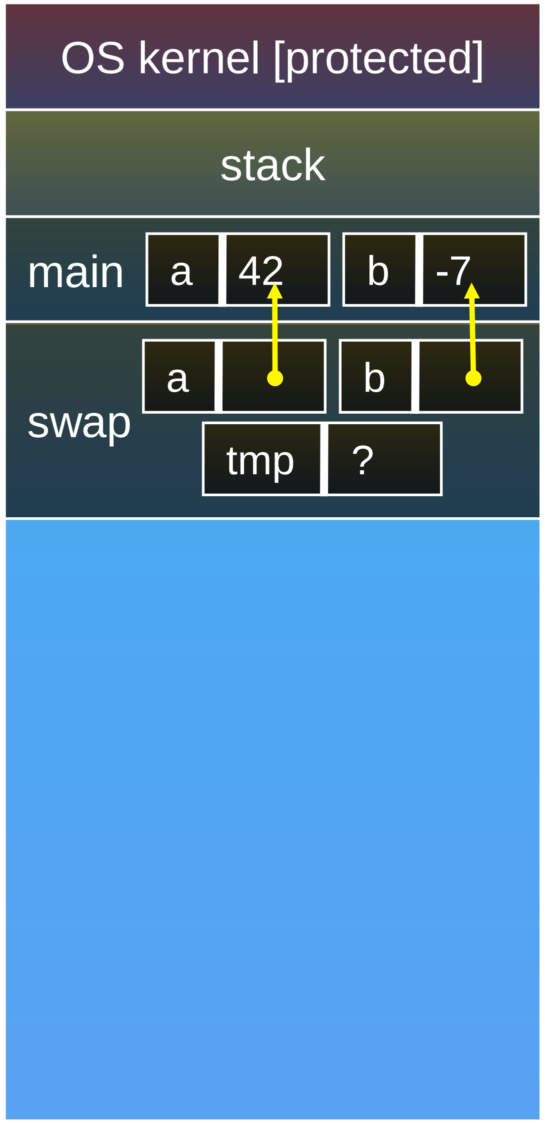
```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(&a, &b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-reference



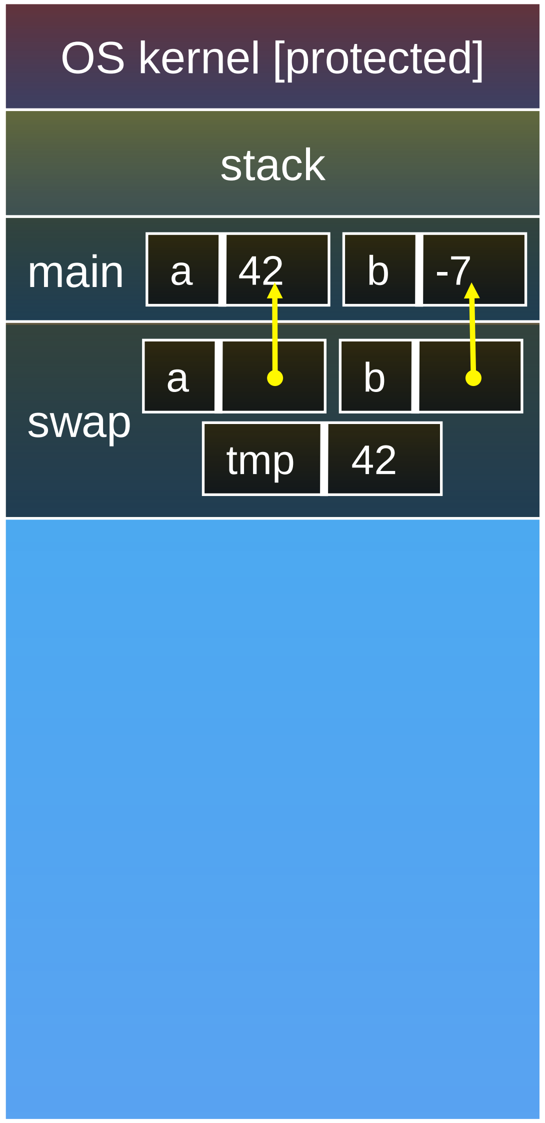
```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(&a, &b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-reference



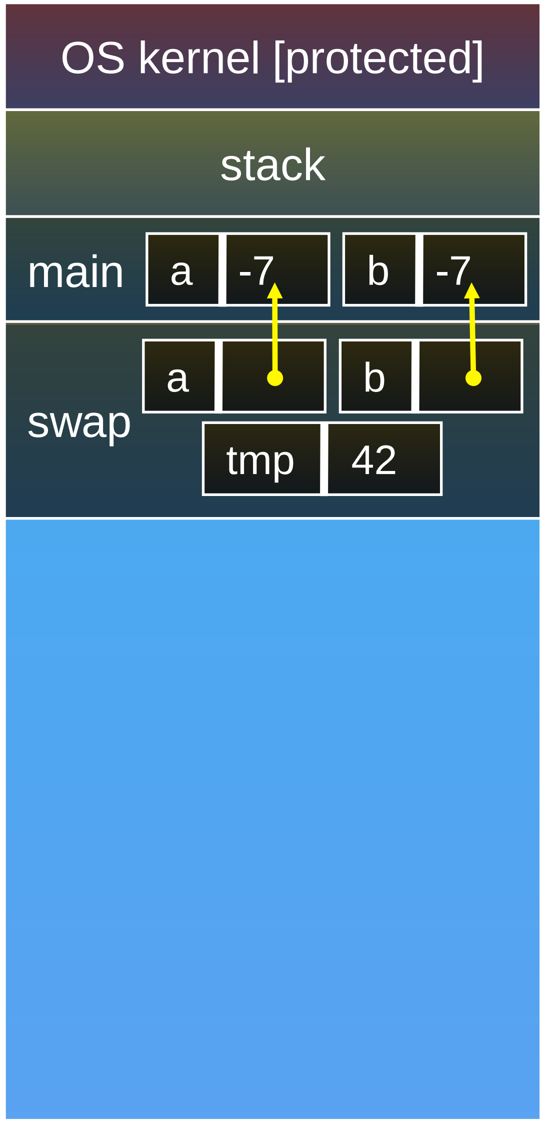
```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(&a, &b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-reference



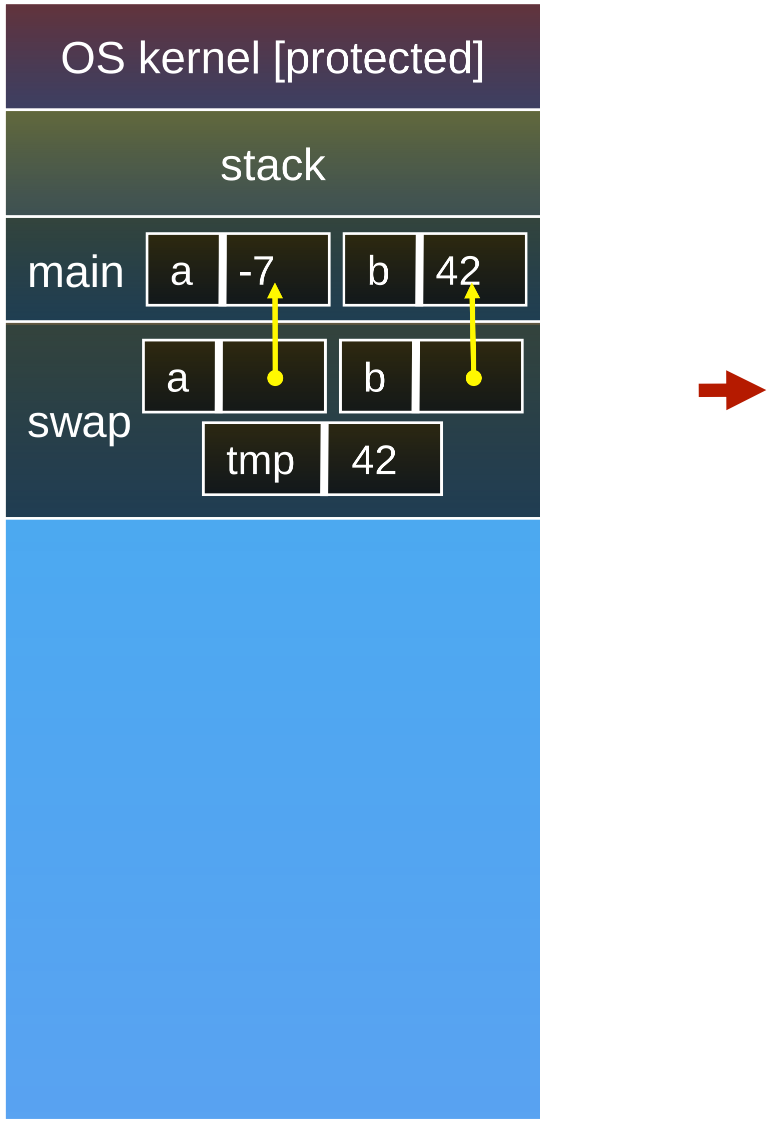
```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(&a, &b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-reference



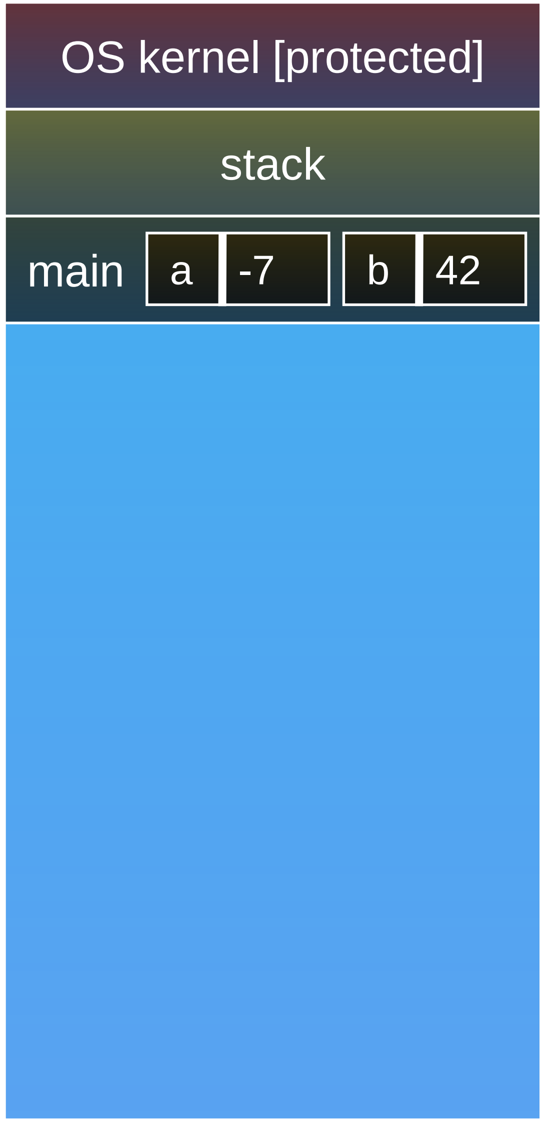
```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(&a, &b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-reference



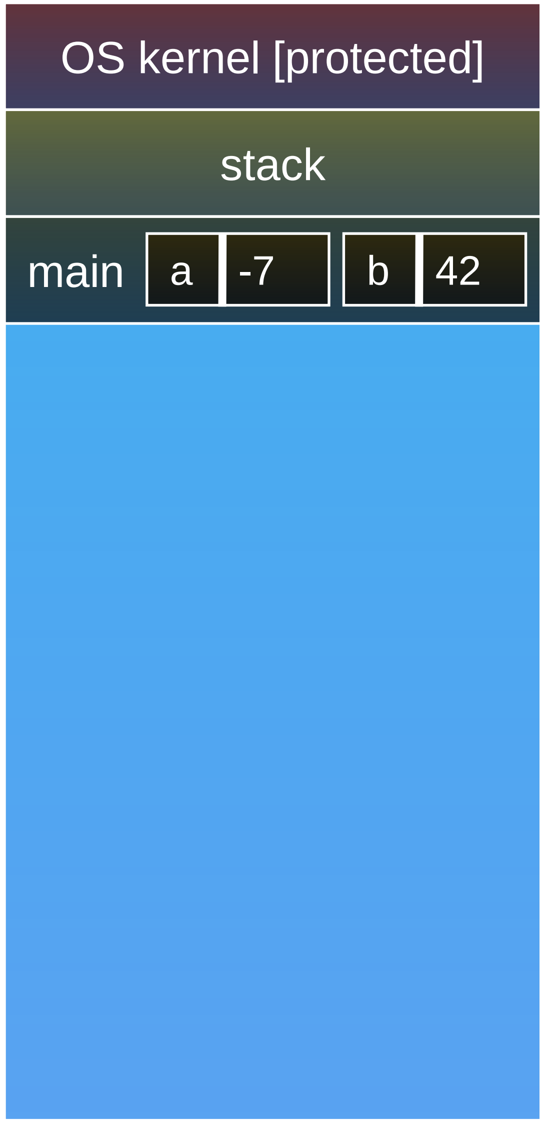
```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(&a, &b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-reference



```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(&a, &b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Pass-by-reference

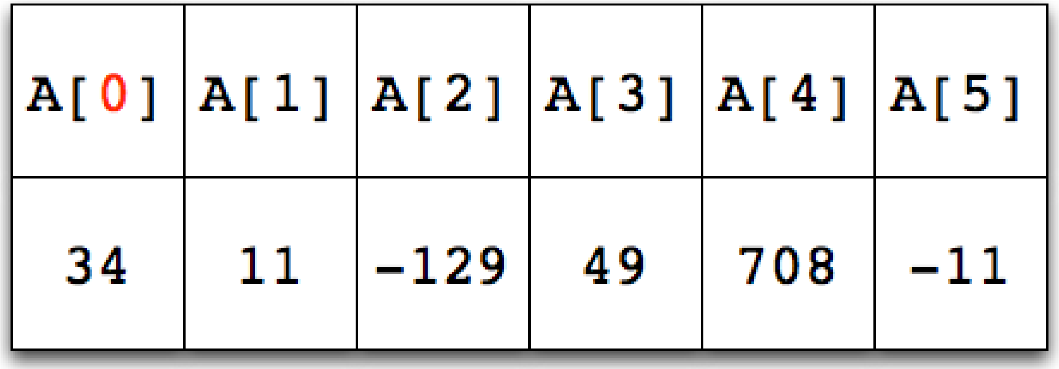


```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(void) {
  int a = 42, b = -7;

  swap(&a, &b);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

Very different than Java

- arrays
  - just a bare, contiguous block of memory of the correct size
  - array of 6 integers requires 6 x 4 bytes = 24 bytes of memory
- arrays have no methods, do not know their own length (no bounds checking)
  - C doesn’t stop you from overstepping the end of an array!
  - many, many security bugs come from this (buffer overflow)

```
int x[6];
```



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
34	11	-129	49	708	-11

Very different than Java

- arrays
  - just a bare, contiguous block of memory of the correct size
  - array of 6 integers requires 6 x 4 bytes = 24 bytes of memory
- arrays have no methods, do not know their own length (no bounds checking)
  - C doesn’
  - many, many security bugs come from this (buffer overflow)

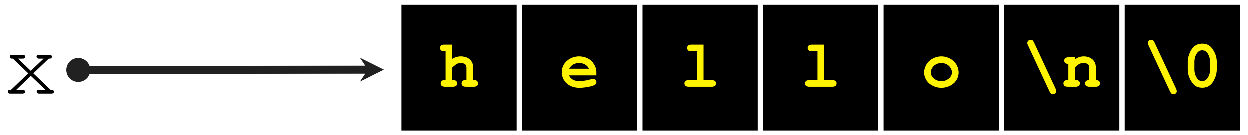
```
X[7] = 45; // Legal C, but can cause memory fault!!!!
```

```
int x[6];
```

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
34	11	-129	49	708	-11

Very different than Java

- strings
  - array of `char`
  - terminated by the `NULL` character `'\0'`
  - are not objects, have no methods; `string.h` has helpful utilities (see strings lecture coming soon!)



```
char *x = "hello\n";
```

Very different than Java

- errors and exceptions
  - C has no exceptions (no try / catch)
  - errors are returned as integer error codes from functions
  - sometimes makes error handling ugly and inelegant
  - some support from OS using signals (end of semester)
- crashes
  - if you do something bad, you'll end up spraying bytes around memory
  - hopefully causing a "segmentation fault" and crash
- objects
  - there aren't any; struct is closest feature (set of fields)

Very different than Java

- memory management
  - there is no garbage collector
  - anything you allocate you have to free (memory leaks)
  - local variables are allocated off of the stack
  - freed when you return from the function
  - global and static variables are allocated in a data segment
  - are freed when your program exits
  - you can allocate memory in the heap segment using malloc()
  - you must free malloc'ed memory with free()
  - failing to free is a leak, double-freeing is an error (hopefully crash)

Very different than Java

- console I/O
  - C library (libc) has portable routines for reading/writing, e.g., `scanf(),printf()`
- file I/O
  - C library has portable routines for reading/writing
    - `fopen(), fread(), fwrite(), fclose()`, etc.
    - does buffering by default, is blocking by default
  - OS provides system calls
    - we'll be using these: more control over buffering, blocking
    - Low level binary reads and writes, e.g., `read(), write(), open(), close()`

Very different than Java

- network I/O
  - C standard library has no notion of network I/O
  - OS provides (somewhat portable) routines
  - lots of complexity lies here
  - errors: network can fail
  - performance: network can be slow
  - concurrency: servers speak to thousands of clients simultaneously

Note: most of these topics will be covered in detail over the semester.

Very different than Java

- Libraries you can count on
    - C has very few compared to most other languages
    - no built-in trees, hash tables, linked lists, sort , etc.
    - you have to write many things on your own
    - particularly data structures
    - error prone, tedious, hard to build efficiently and portably
    - less productive language than Java, C++, python, or others

Problem: ordering

- Don’t call a function that hasn’t been declared yet:

```
#include <stdio.h>

int main(void) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}

// sum integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i=1; i<=max; i++) {
    sum += i;
  }
  return sum;
}
```

Problem: ordering

- Solution 1: reverse order of definition

```
#include <stdio.h>

// sum integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i=1; i<=max; i++) {
    sum += i;
  }
  return sum;
}

int main(void) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}
```

Problem: ordering

- Solution 2: provide function declaration
    - teaches the compiler the argument and return types of the function that will appear later

```
#include <stdio.h>

// this function prototype is a
// declaration of sumTo
int sumTo(int);

int main(void) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}

// sum integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i=1; i<=max; i++) {
    sum += i;
  }
  return sum;
}
```

- The body-less function declaration is called a *functional prototype*.

UNIX Std*

- There are three predefined streams provided to all UNIX programs
  - Standard input (`stdin`)
  - Standard output (`stdout`)
  - Standard error (`stderr`)

- printf(“this is printed to standard output\n”);
- fprintf(stdout, “this is printed to standard output as well\n”);
- fprintf(stderr, “this is printed to standard error\n”);