Systems and Internet Infrastructure Security

Institute for Networking and Security Research
Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA

Introduction to C (Part 1)

Devin J. Pohly <djpohly@cse.psu.edu>
(adapted from original slides by Steve Gribble at UWash)

Assignment #2

- Getting your feet wet with C
- See website
- Due February 7
- Any major problems – talk to me!

Unix utilities: tar

- `tar` collects multiple files and directory data in a single file
  - Archive, like Zip or RAR
  - `tar` just puts the files together
  - It uses other programs to compress
- The resulting single file is frequently called a tarball.
  - Common extensions: .tar.gz, .tgz, .tar.bz2, .tbz

tar usage

`tar -x -v -z -f file.tar.gz`

- `-x`: extract from an archive
- `-v`: verbose (show info/filenames)
- `-z`: uncompress the tarball with gzip
- `-f`: read the archive from this file
- PROTIP: combine short options `-xvzf` and save keystrokes! (see below)

`tar -cvzf file.tar.gz <files>`

- `-c`: create an archive
- `-v`: verbose
- `-z`: compress the tarball with gzip
- `-f`: output the archive to this file
- `<files>`: list of files and directories to store

C workflow

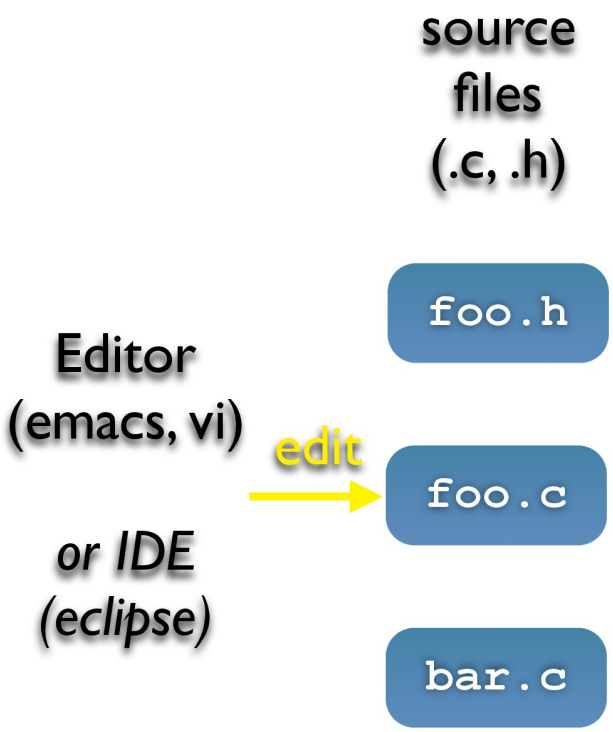
PENNSTATE

1 8 5 5

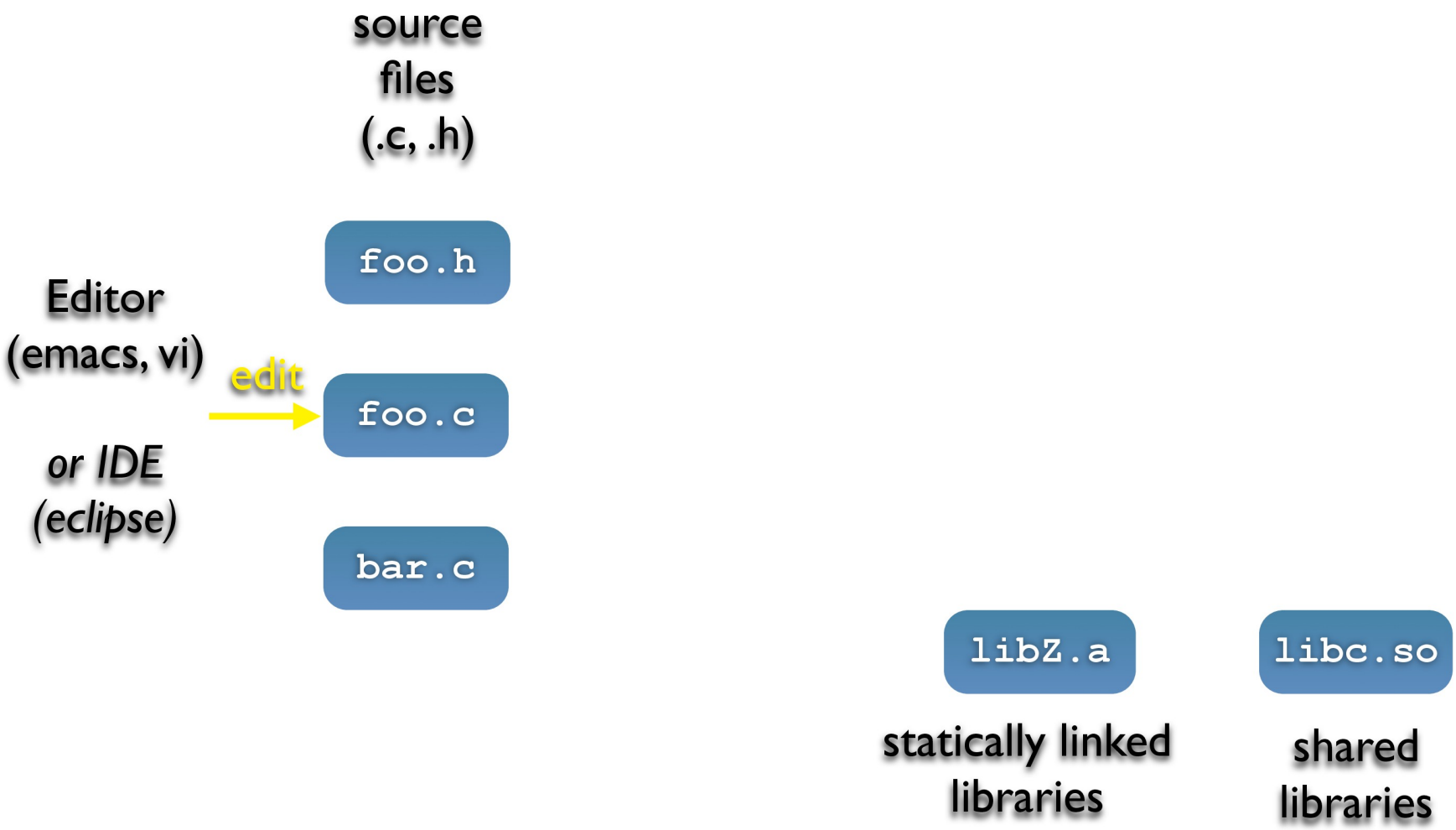
Editor
(emacs, vi)

or IDE
(eclipse)

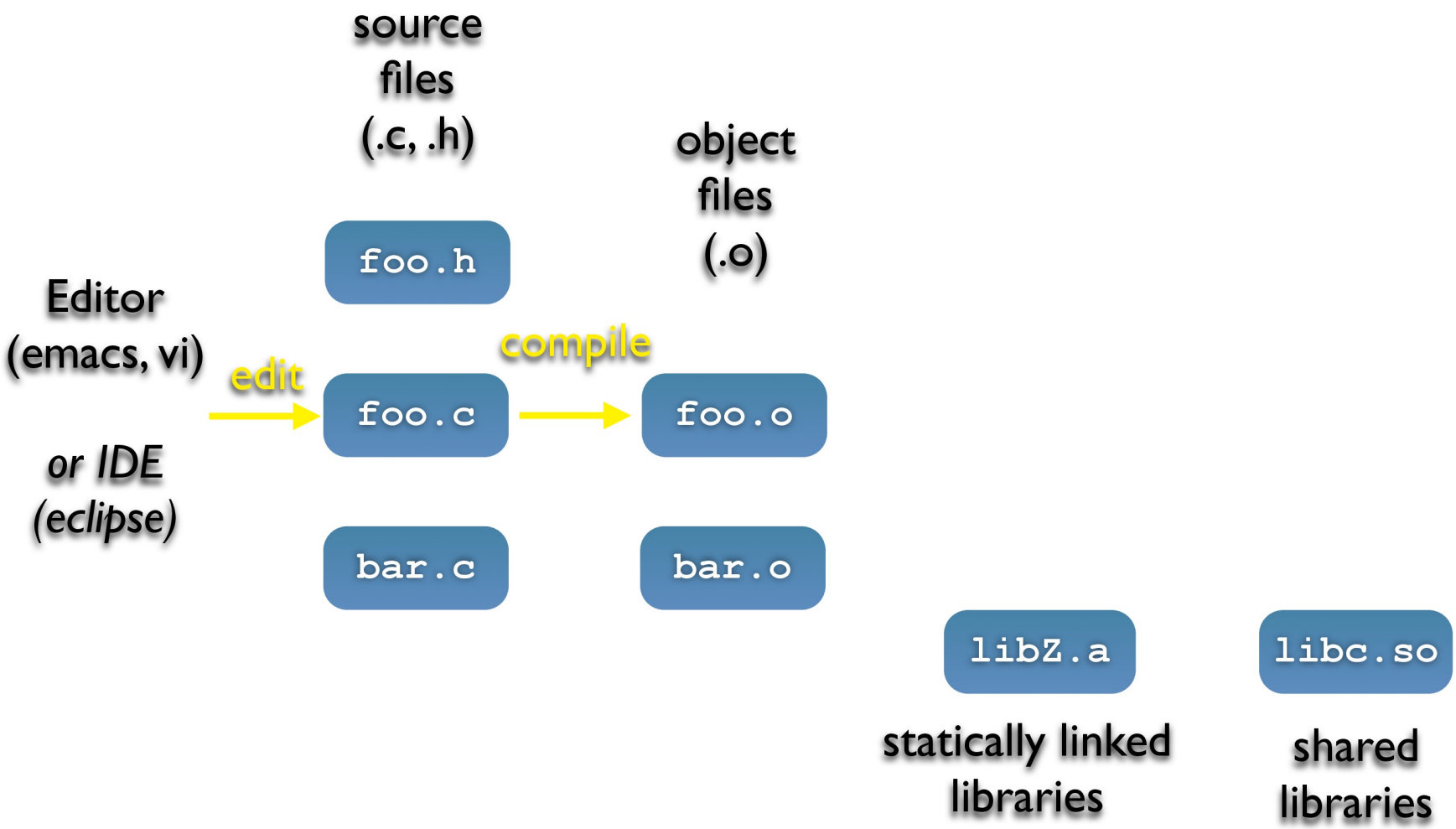
C workflow


source
files
(.c, .h)
foo.h
Editor
(emacs, vi)
edit
foo.c
or IDE
(eclipse)
bar.c

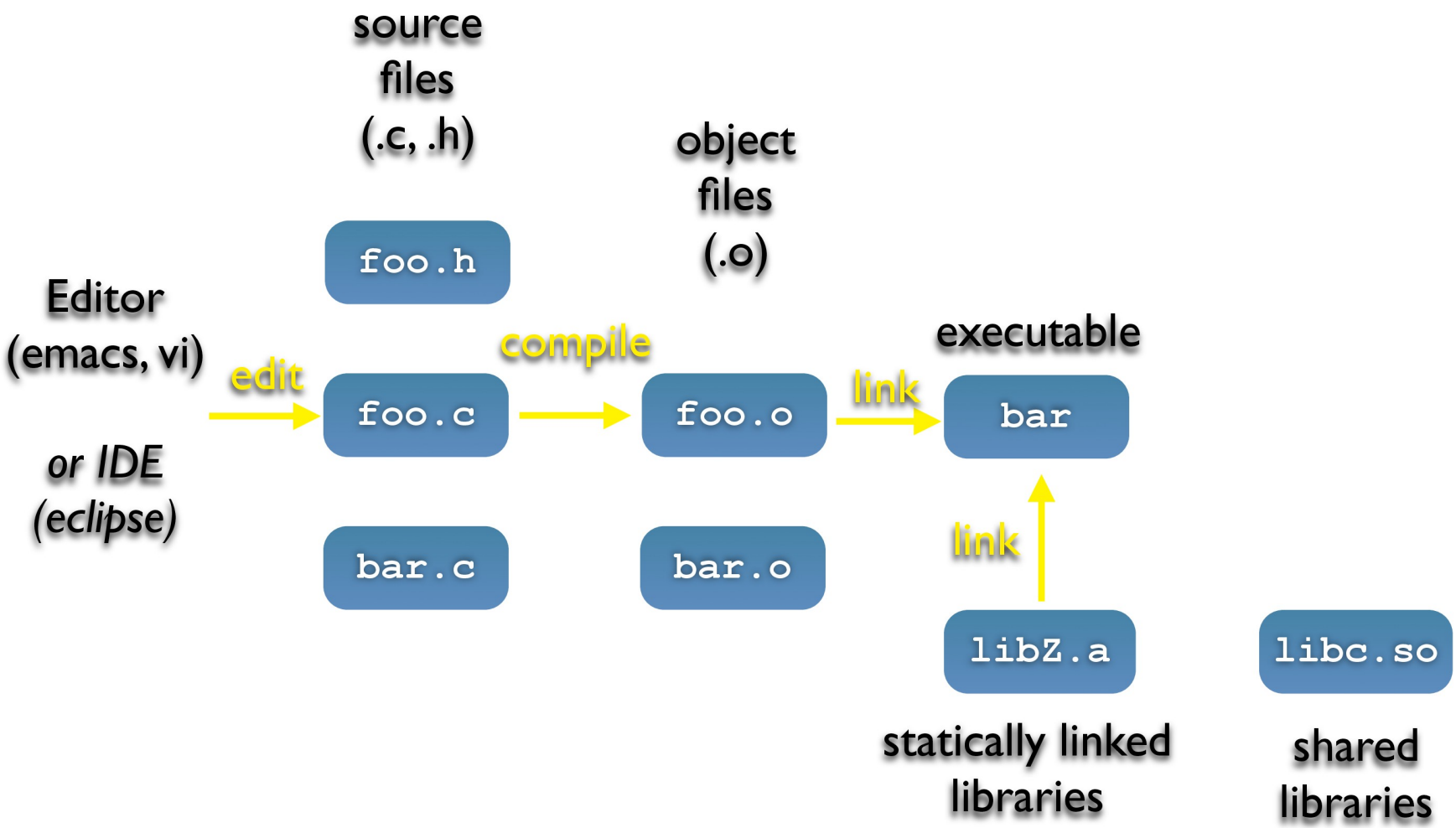
C workflow


source
files
(.c, .h)
foo.h
Editor
(emacs, vi)
edit
foo.c
or IDE
(eclipse)
bar.c
libZ.a
libc.so
statically linked
libraries
shared
libraries

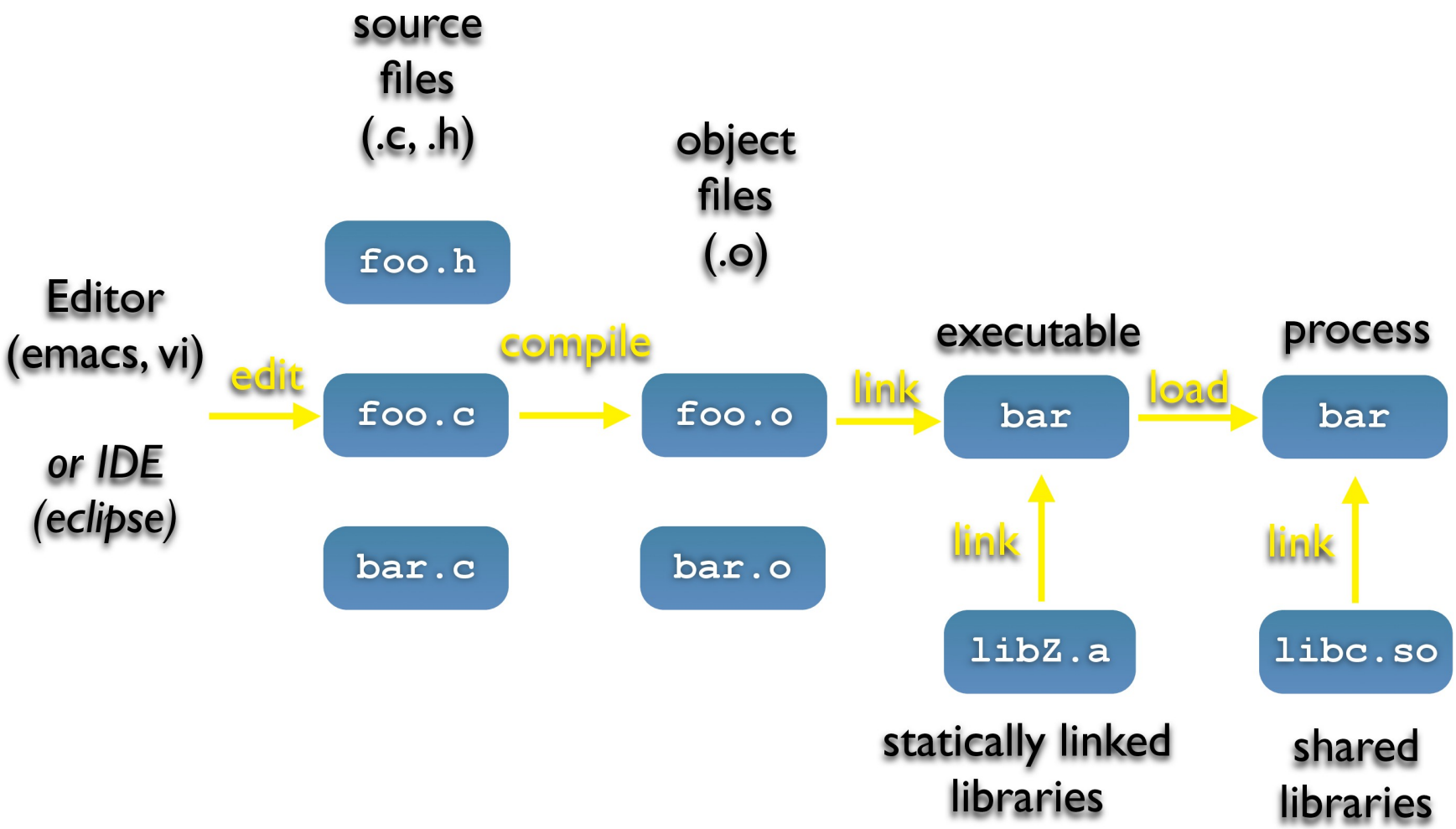
C workflow


source
files
(.c, .h)
object
files
(.o)
foo.h
Editor
(emacs, vi)
compile
edit
foo.c
foo.o
or IDE
(eclipse)
bar.c
bar.o
libZ.a
libc.so
statically linked
libraries
shared
libraries

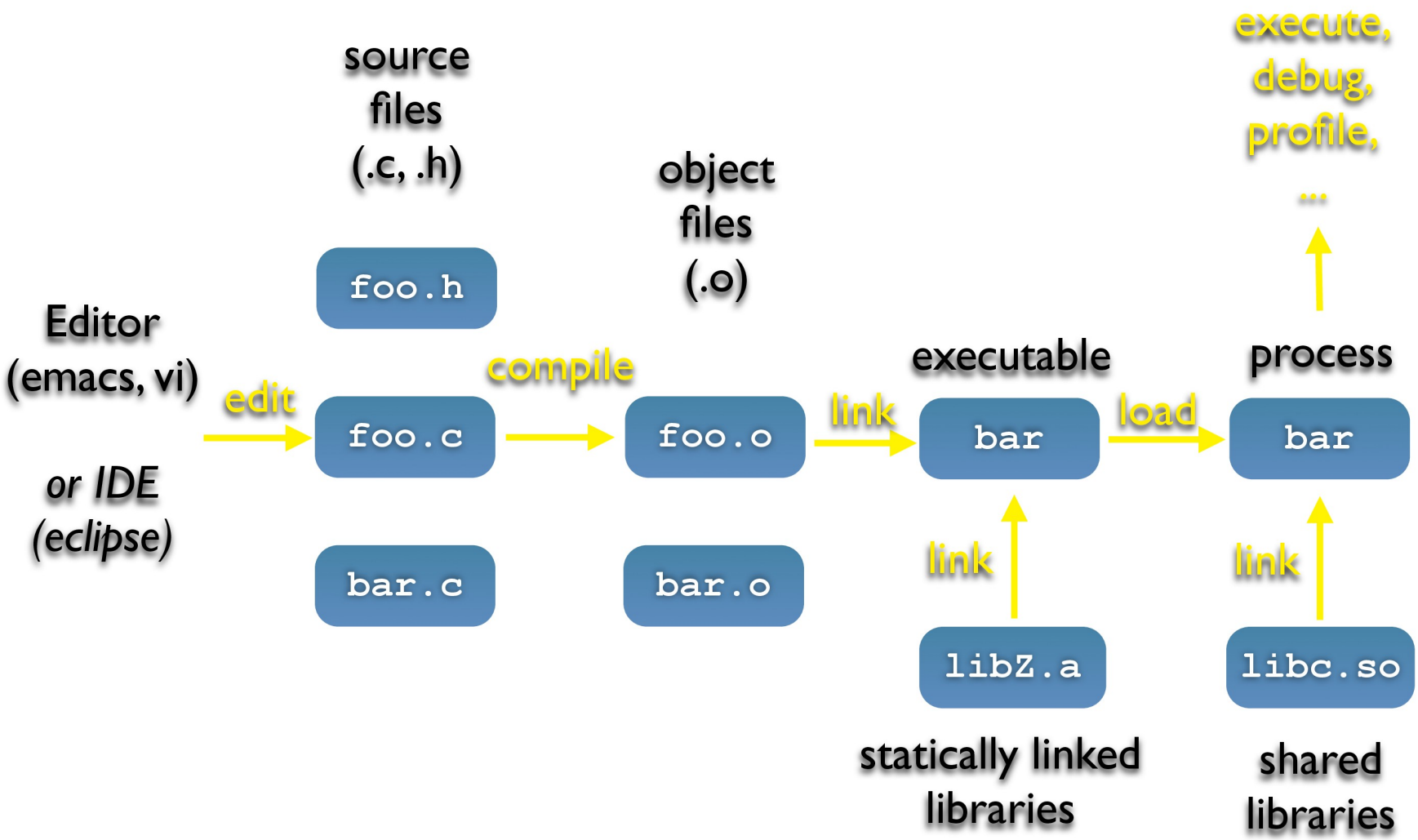
C workflow


source
files
(.c, .h)
object
files
(.o)
foo.h
Editor
(emacs, vi)
edit
compile
executable
link
foo.c
foo.o
bar
or IDE
(eclipse)
link
bar.c
bar.o
libZ.a
libc.so
statically linked
libraries
shared
libraries

C workflow


source
files
(.c, .h)
object
files
(.o)
foo.h
Editor
(emacs, vi)
edit
foo.c
compile
foo.o
link
executable
bar
load
process
bar
or IDE
(eclipse)
bar.c
bar.o
link
link
libZ.a
libc.so
statically linked
libraries
shared
libraries

C workflow


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

From C to machine code

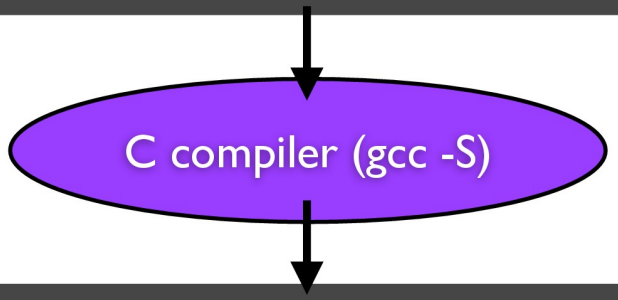
C source file
(dosum.c)

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

From C to machine code

C source file
(dosum.c)

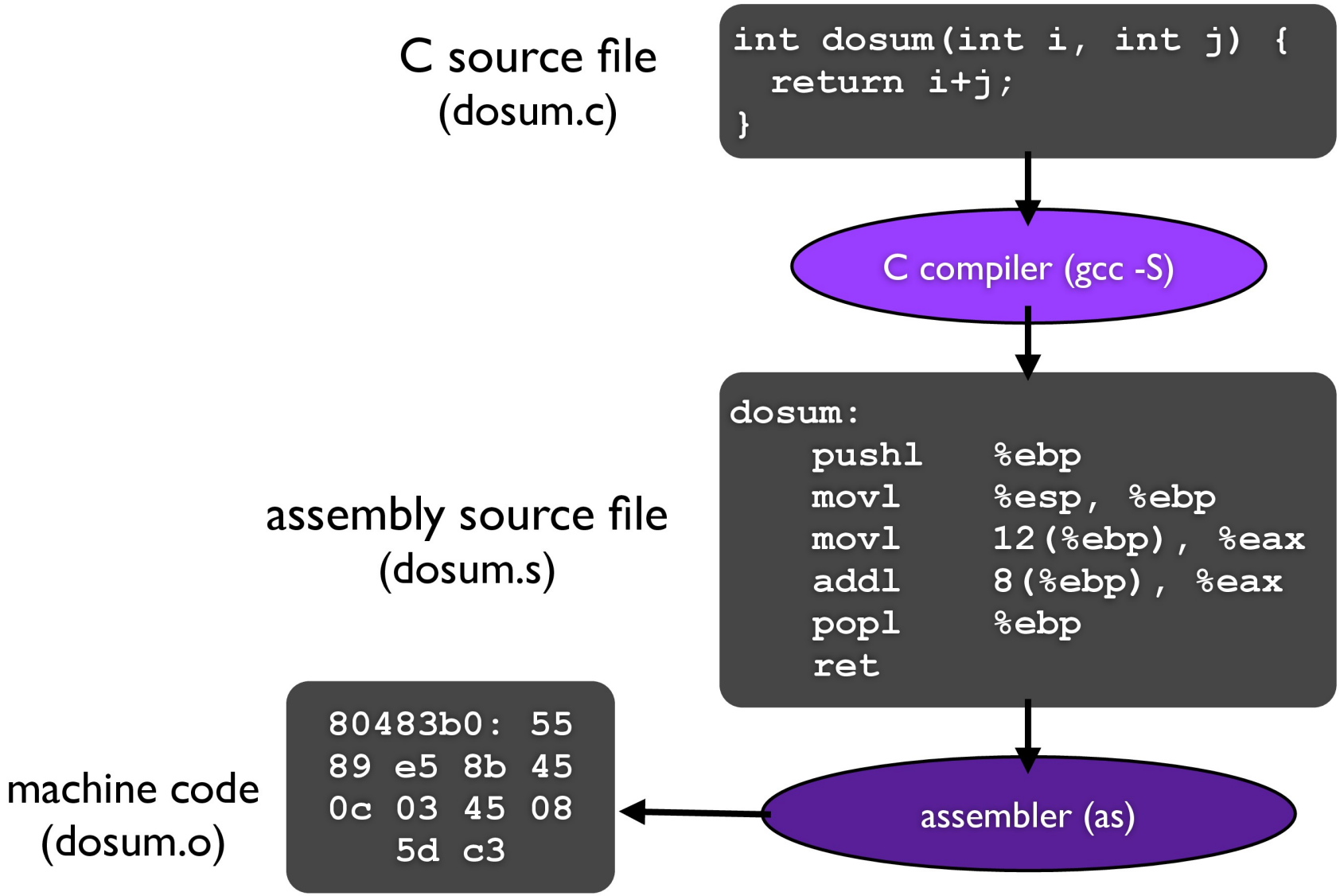
```
int dosum(int i, int j) {
  return i+j;
}
```



assembly source file
(dosum.s)

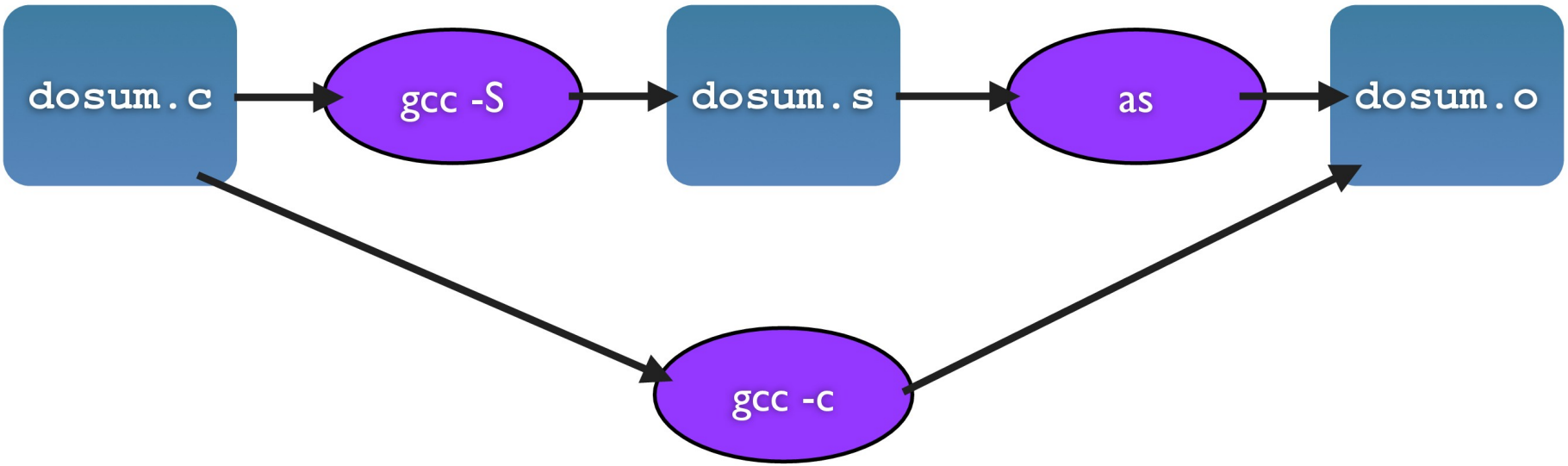
```
dosum:
    pushl   %ebp
    movl    %esp, %ebp
    movl    12(%ebp), %eax
    addl    8(%ebp), %eax
    popl    %ebp
    ret
```

From C to machine code



Skipping the ASM

- Most C compilers generate .o files (machine code) directly
  - I.e., they don't actually save the readable .s file
  - If you're curious, pass the `-S` option to `gcc`.



Multi-file C programs

C source file (dosum.c)

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

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;
}
```

Multi-file C programs

C source file (dosum.c)

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

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 not implemented in sumnum.c

Multi-file C programs

C source file (dosum.c)

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

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;
}
```

this "prototype" of dosum( ) tells gcc about the types of dosum's arguments and its return value

dosum( ) is not implemented in sumnum.c

Multi-file C programs

C source file (dosum.c)

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

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;
}
```

Multi-file C programs

C source file (dosum.c)

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

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;
}
```

where is the implementation of printf?

Multi-file C programs

C source file (dosum.c)

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

C source file (sumnum.c)

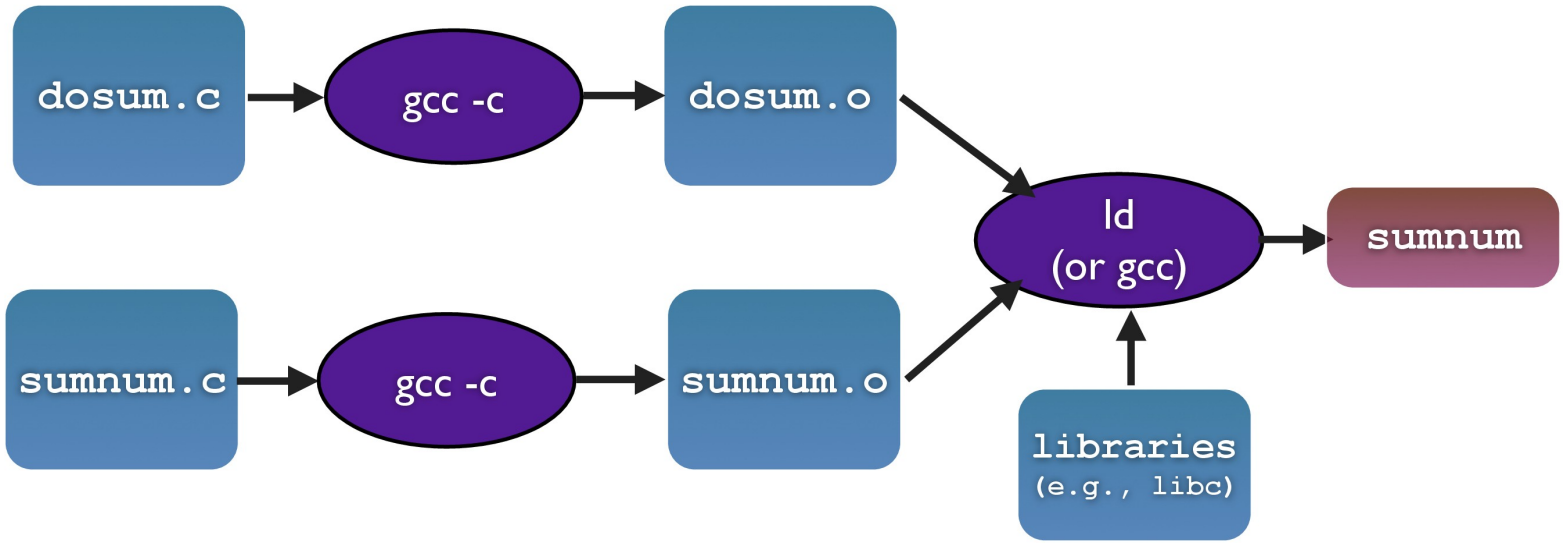
```
#include <stdio.h>

int dosum(int i, int j);

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

why do we need this #include?

where is the implementation of printf?

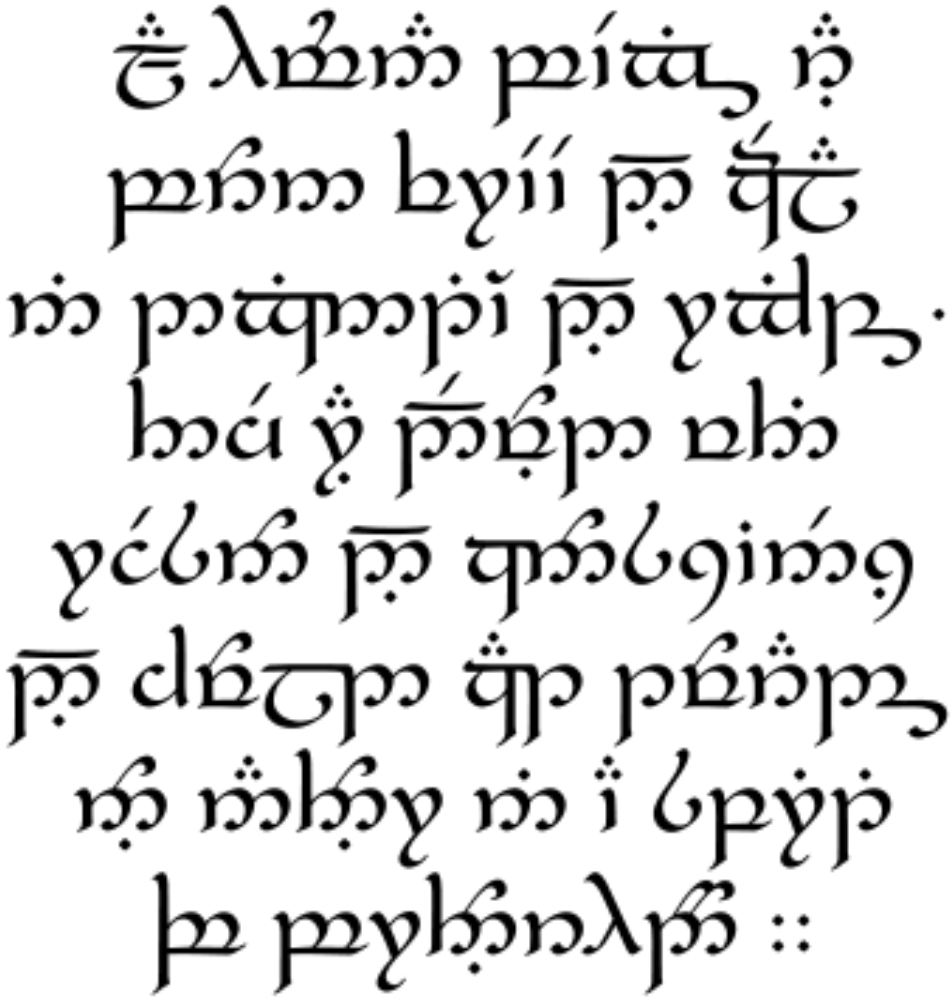
Compiling multiple files

- Multiple object files are *linked* to produce an executable
  - Standard libraries (libc, crt1, …) are usually also linked in
  - Library are just pre-assembled collections of .o files



Object files

- sumnum.o and 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., the code in sumnum.o relies on printf() and dosum(), but these are defined in libc.a and dosum.o, respectively
  - Linking combines the needed object files and libraries to resolve these external symbols



The C language

PENNSTATE 1855

- Things that are the same as Java and C++:
  - Syntax of statements, control structures, function calls
  - Primitive types: `int`, `double`, `char`, `long`, `float`
  - Type-casting syntax: `double x = (double) 5 / 3;`
  - Expressions, operators, precedence
    - `! ++ -- * / % + - < <= > >= == != && || = += -= *= /= %=`
  - Static scope: local scope within `{curly braces}`
  - Comments: `/* comment */` or `// comment`

Primitive types in C

- Integer types
  - char, int
- Floating point types
  - float, double
- Modifiers
  - short [int]
  - long [int, double]
  - long long [int]
  - 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 fixed-width int types

- What type can I use to get exactly 32 bits?

```
#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/C++

- Variables
  - Declared at the start of a function or block (changed in C99)
  - Need not be initialized before use (`gcc -Wall` will warn)

```c
#include <stdio.h>

int main(int argc, char **argv) {
  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/C++

- `const`
  - 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?

```c
#include <stdio.h>

int main(int argc, char **argv) {
  const double MAX_GPA = 4.0;

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

Similar to Java/C++

- `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)
  - Any type can be used: 0 and NULL are false, everything else true

```
int i;

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

Similar to Java/C++

- C always passes arguments by value
  - Value is *copied* into function
  - Any local change is not reflected in the original value that was passed
- Pointers let you pass by reference
  - Copy the memory location of the variable and use that to change the original
  - Most tricky and dangerous feature 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(int argc, char **argv) {
  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;
}
```

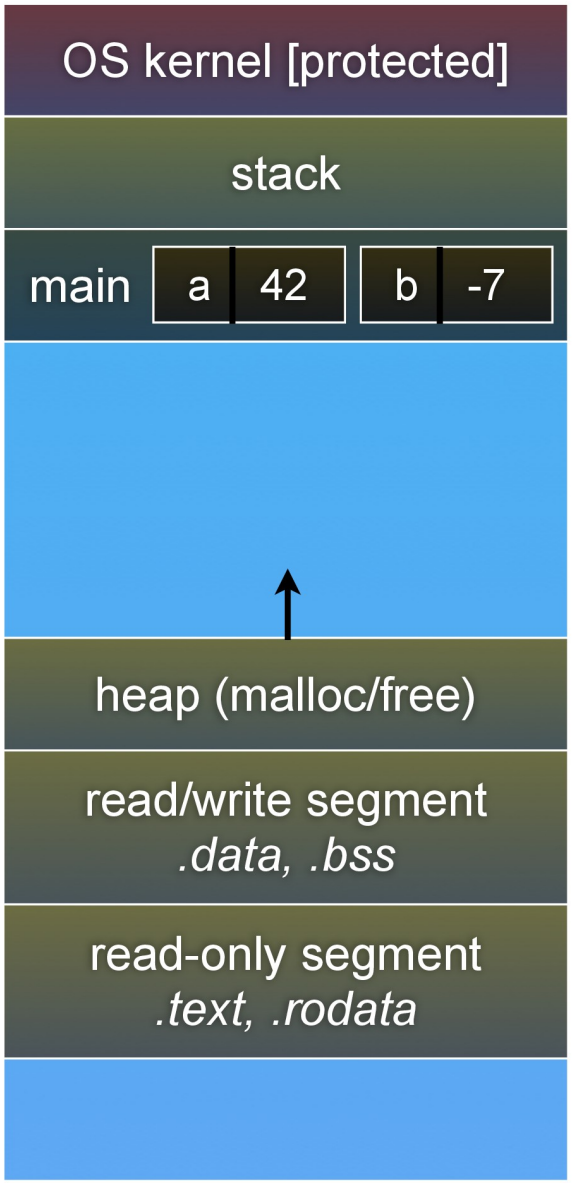
Pass-by-value

- Passing arguments by value
  - Callee receives its own copy of the argument
  - If the callee modifies its copy, the caller's copy isn't modified

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

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

^ Does not work!

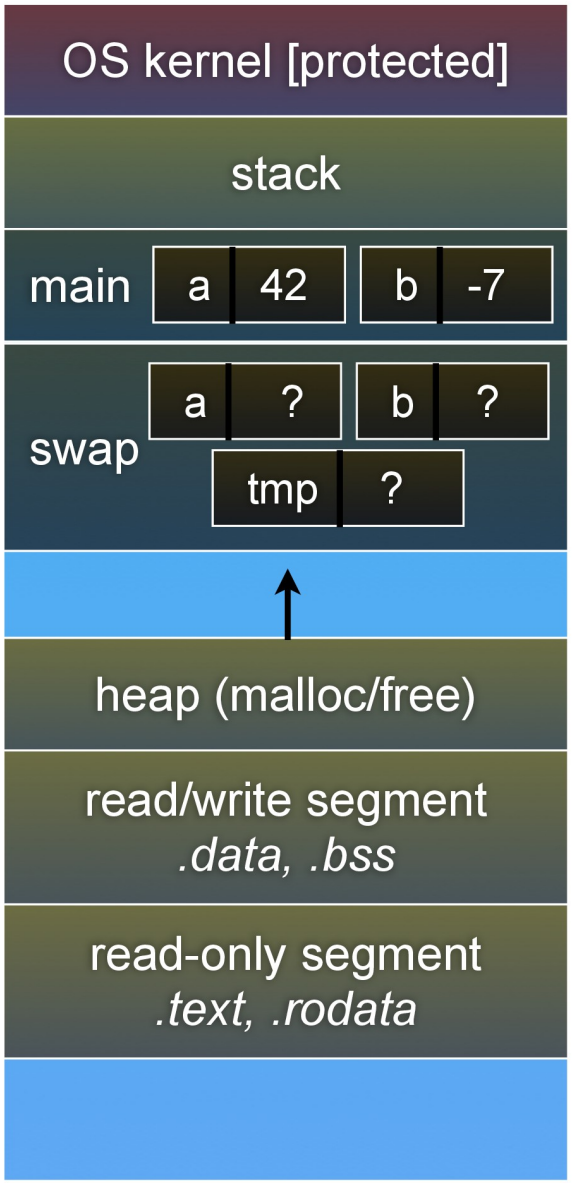
Pass-by-value




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

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

^ Does not work!

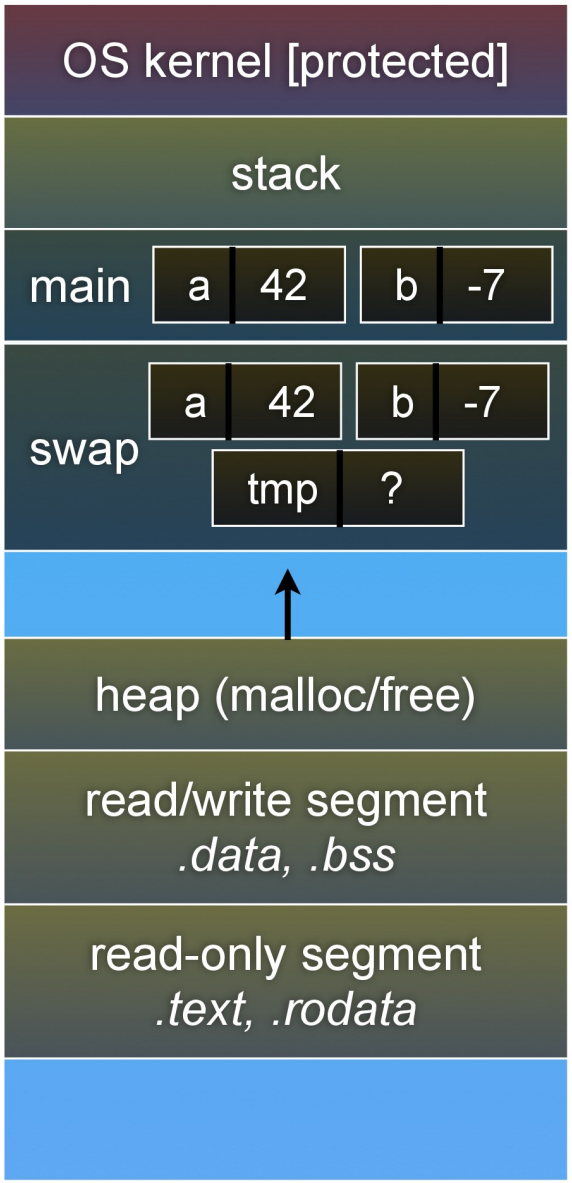
Pass-by-value




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

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

^ Does not work!

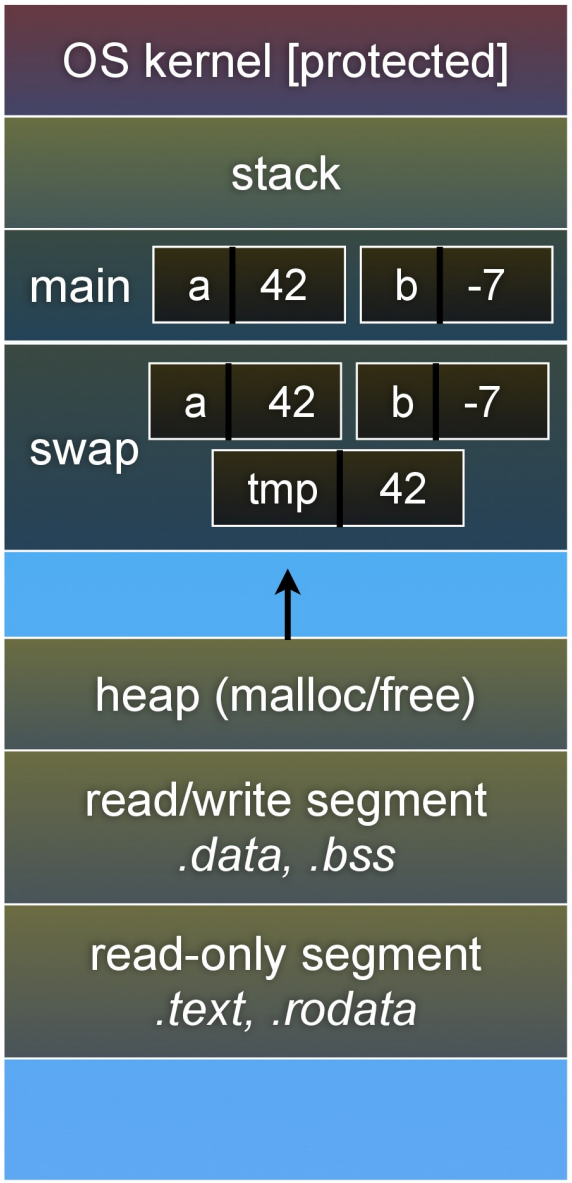
Pass-by-value




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

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

^ Does not work!

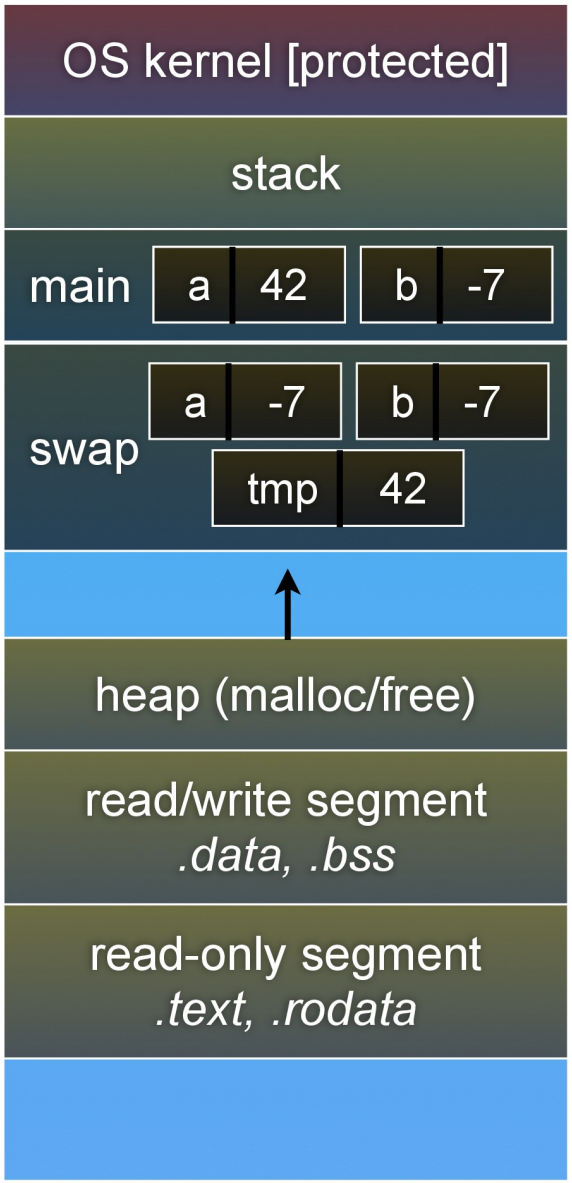
Pass-by-value




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

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

^ Does not work!

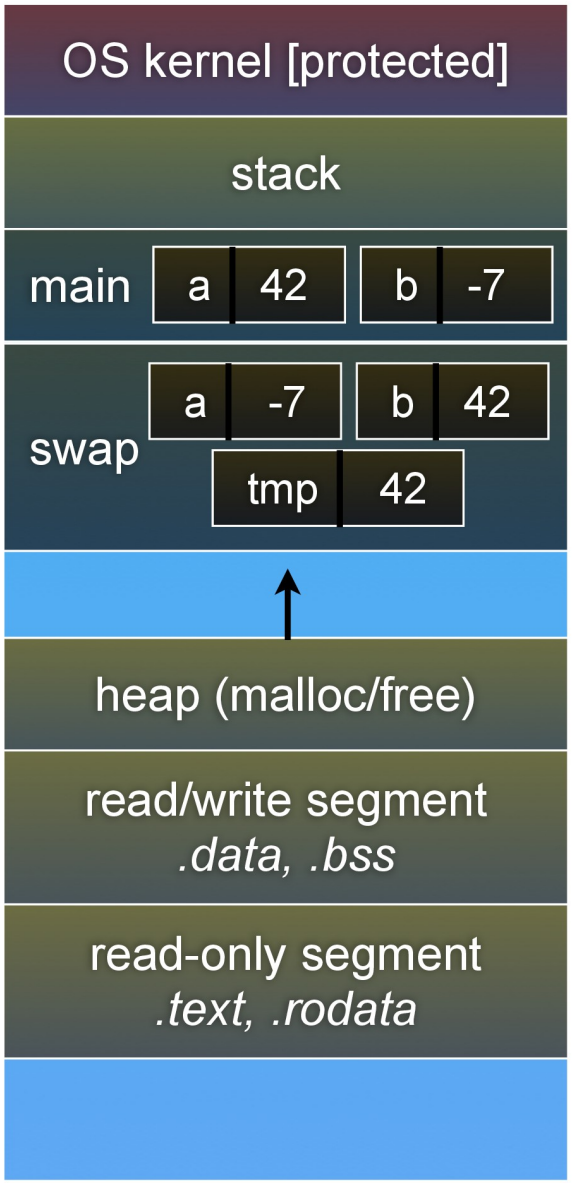
Pass-by-value




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

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

^ Does not work!

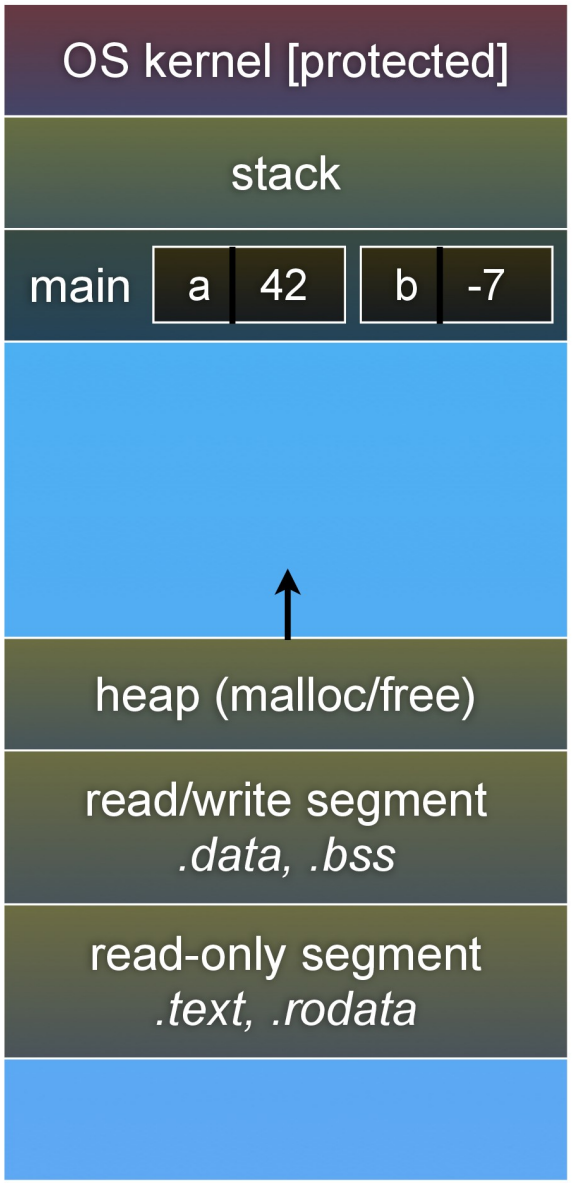
Pass-by-value




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

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

^ Does not work!

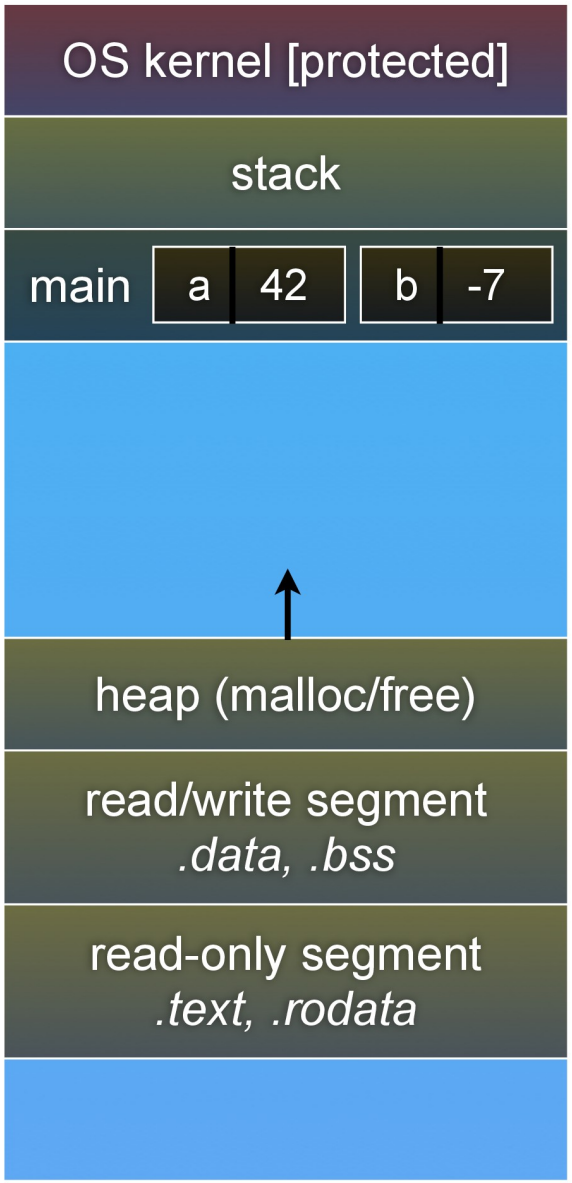
Pass-by-value




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

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

^ Does not work!

Pass-by-value




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

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

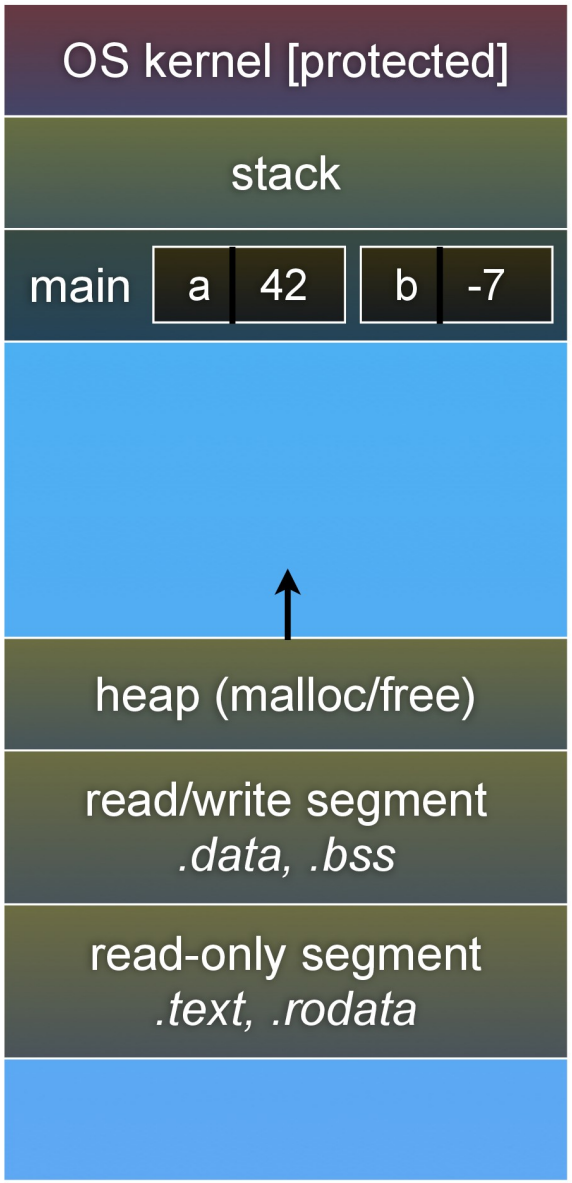
^ Does not work!

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 is an address that points to the variable
  - This gives the callee a way to modify a variable that's in the caller's scope.

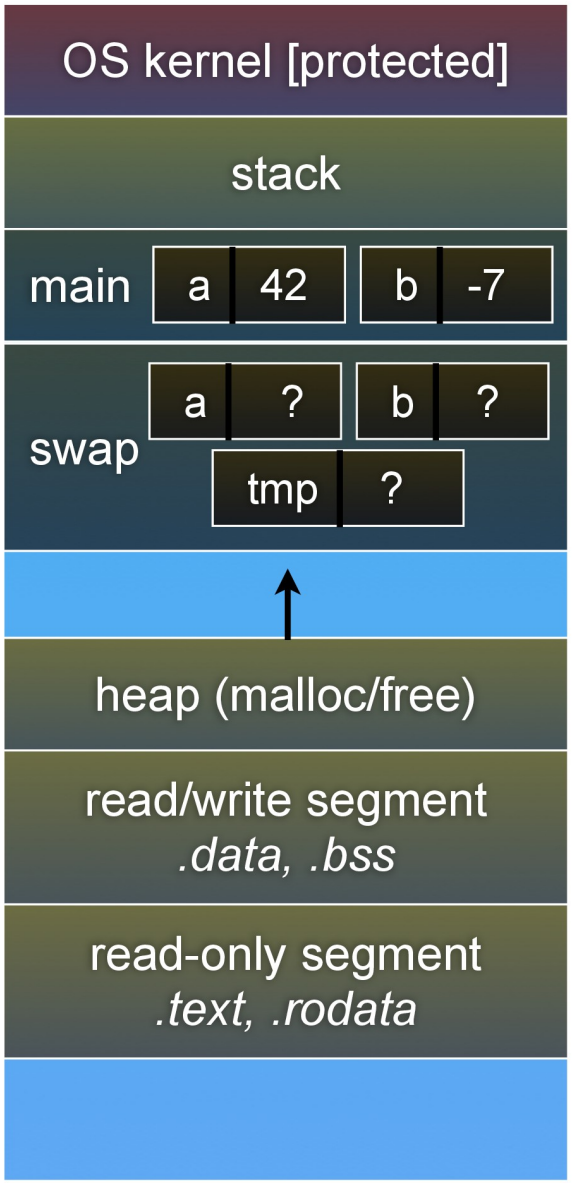
```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

Pass-by-reference



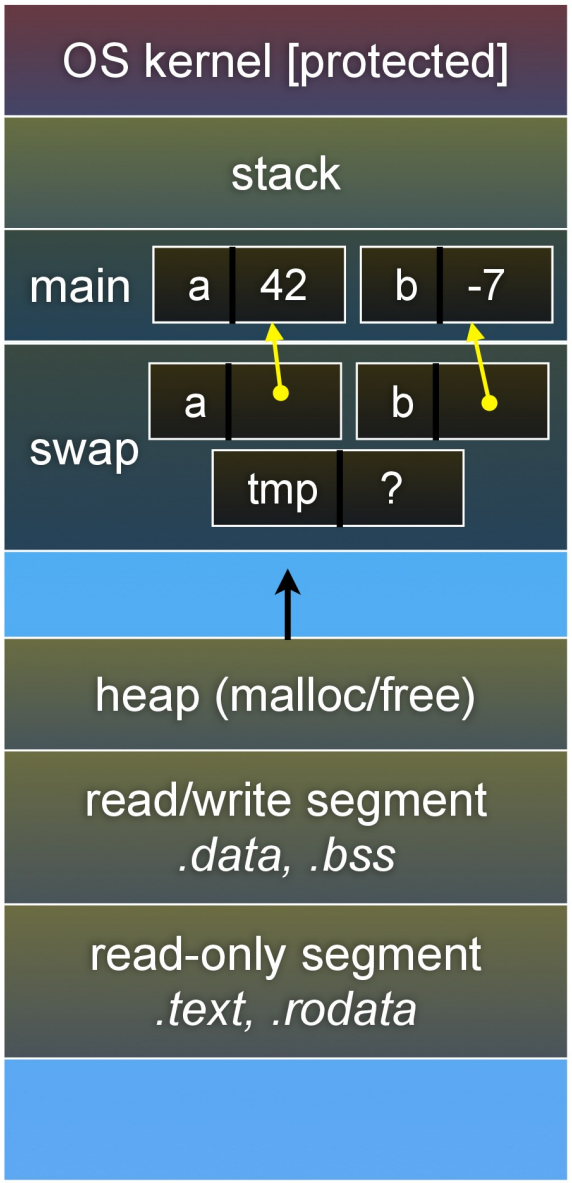

```c
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

Pass-by-reference



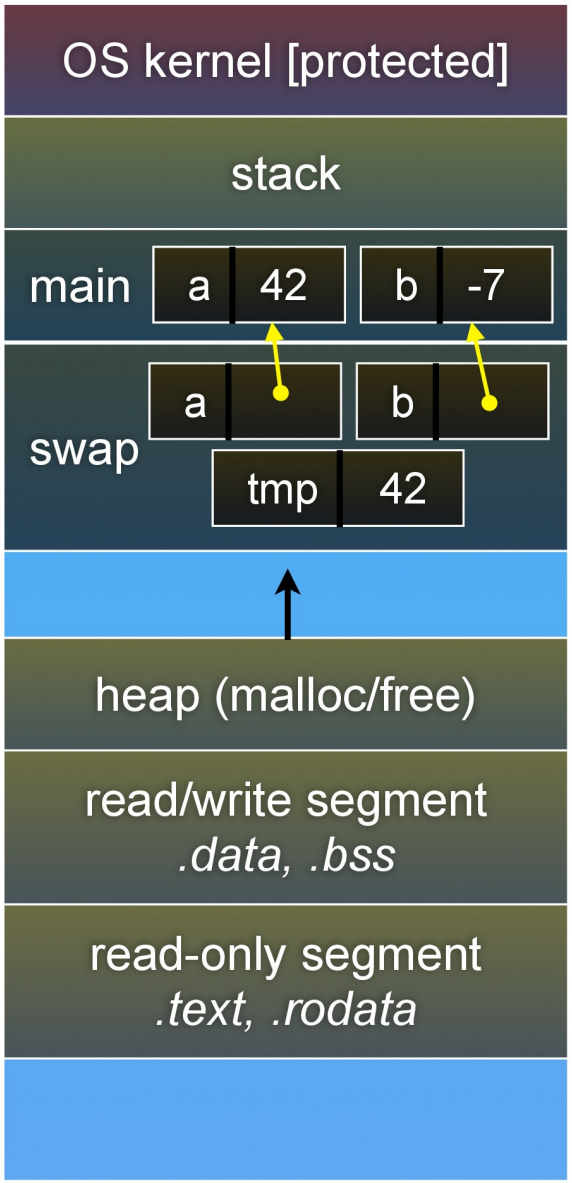

```c
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

Pass-by-reference



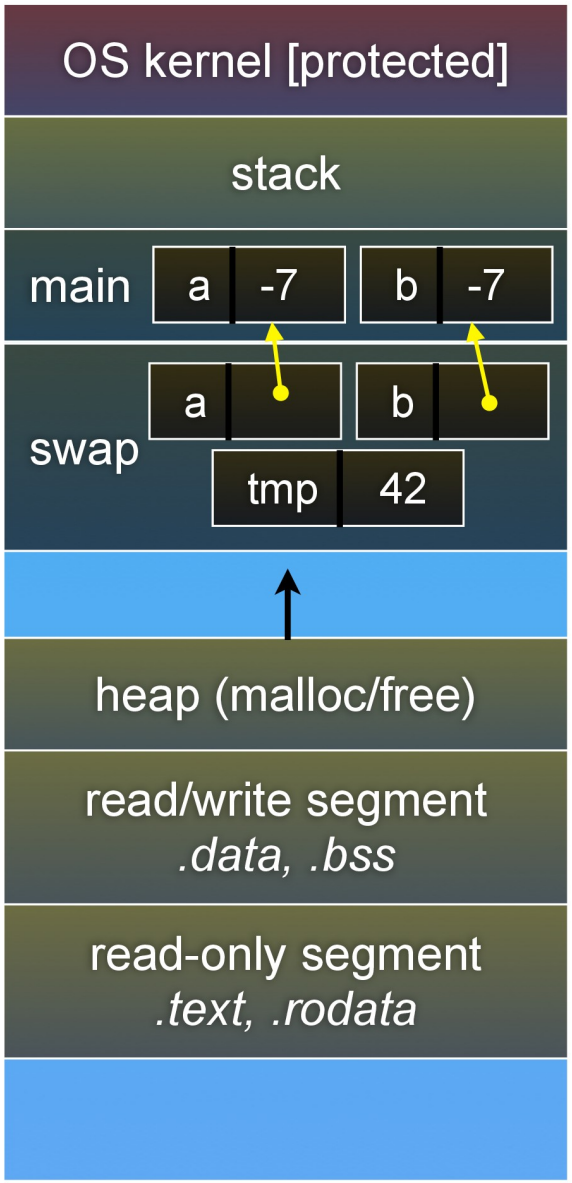

```c
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

Pass-by-reference



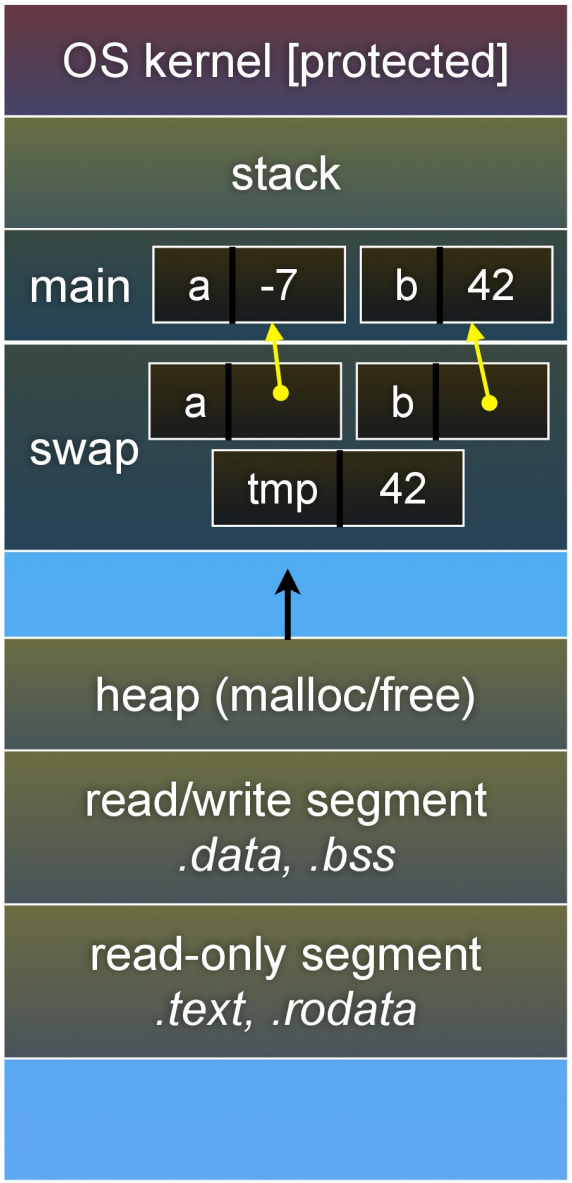

```c
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

Pass-by-reference



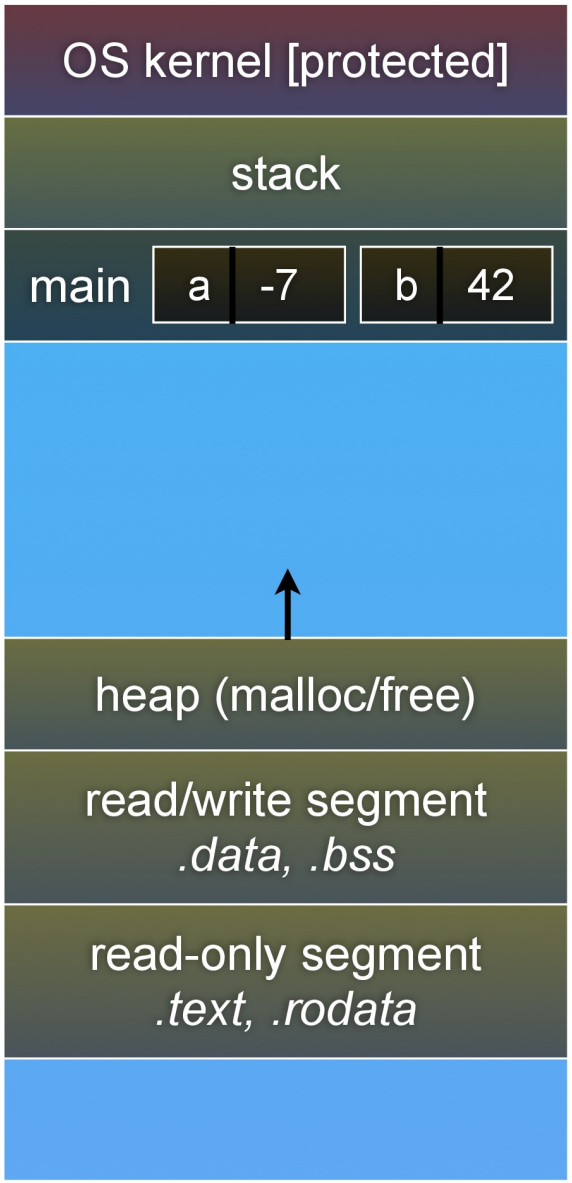

```c
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

Pass-by-reference



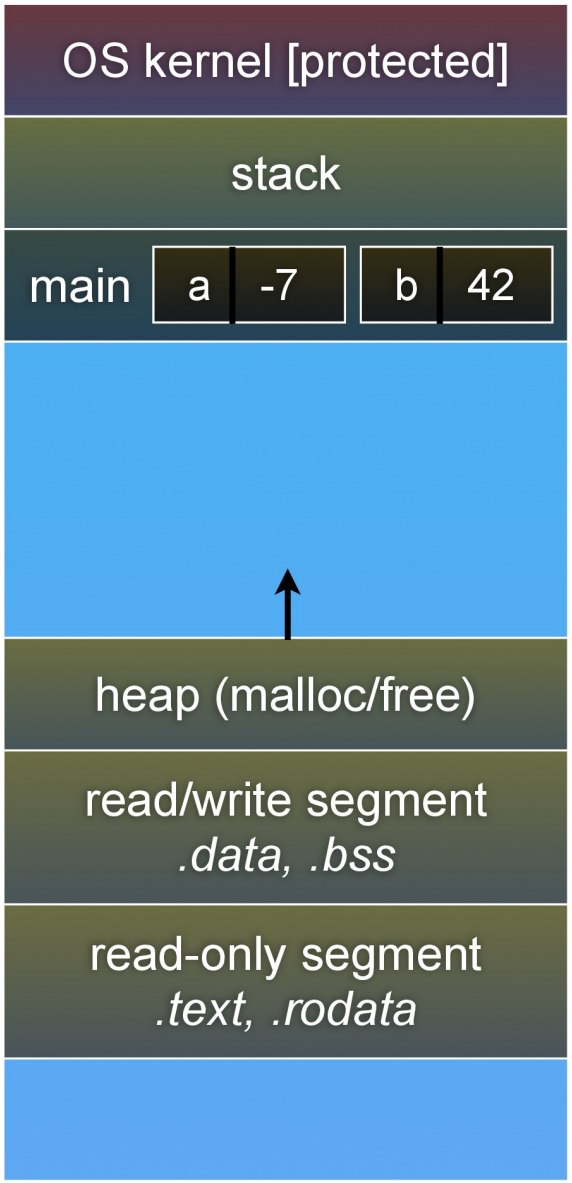

```c
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

Pass-by-reference




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

int main(int argc, char **argv) {
  int a = 42, b = -7;

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

Pass-by-reference




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

int main(int argc, char **argv) {
  int a = 42, b = -7;

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