

S

Compilers: Part 1

Assembly, Linkers, and

Loaders

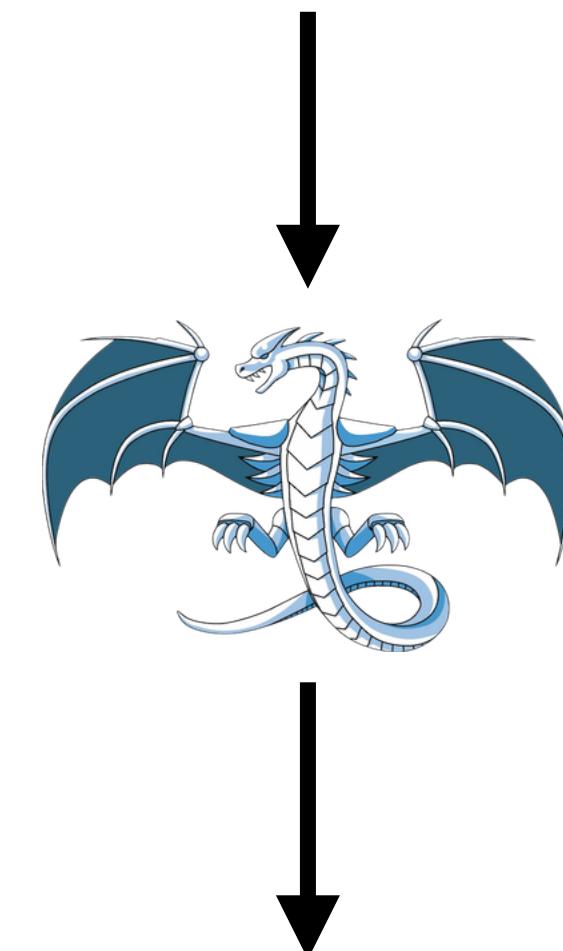
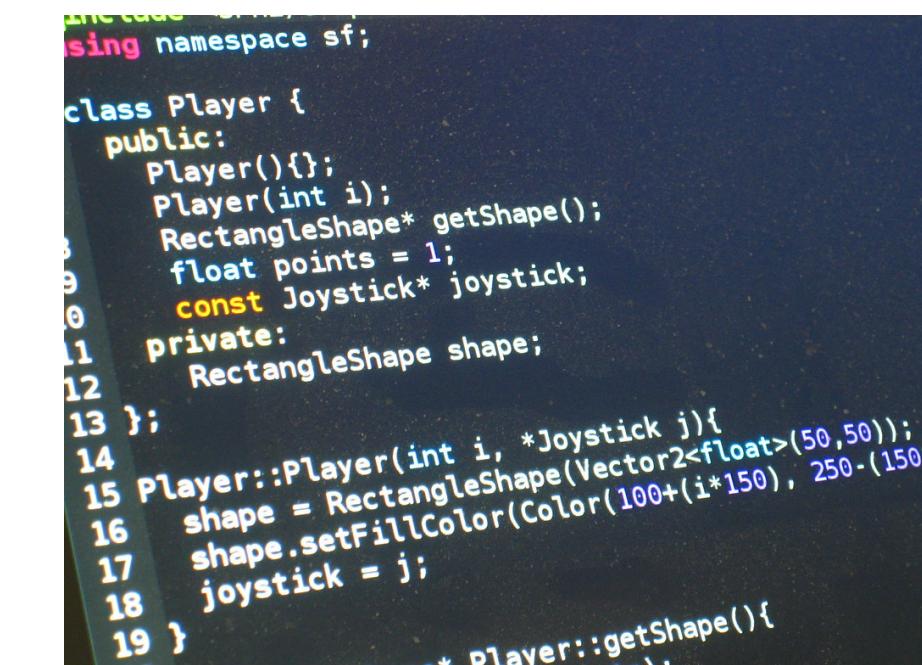
CIS352 — Spring 2024

Kris Micinski

Compilers

C++

- As you probably know, the processor
 - Translate “high-level” language to “object” language
 - Typically, the object language is a **binary**, though other examples exist (e.g., JVM bytecode).
 - Parsing binary formats can be done very efficiently
 - The precise format of the object file is largely determined by the OS linker / loader
 - E.g., Windows Portable Executable (PE binaries), Mac Mach-o, Executable and Linkable (ELF)



LLVM/ Clang

ELF/...



Assembly Language and ISAs

- The computer executes very, very simple *instructions* on a clock.
- Assembly language is the human-readable version of the binary language ultimately spoken by the processor.
- The processor ultimately reads, decodes, and executes instructions in a specific language called its **Instruction Set Architecture (ISA)**
 - This is the “native” language that your processor knows how to execute.
 - Common examples you may have heard of: Pentium x86, x86-64, ARM

```
section .data
    int_format db "Hello, world.",10,0
    global _main
    extern _printf

section .text

_main:
    push rbp
    mov rbp, rsp ; move the stack pointer into the base pointer

    ; Set up for calling printf
    lea rdi, [rel int_format] ; Load address of format string into rdi
    mov rax, 0 ; Zero rax to indicate no floating-point arguments are passed
    call _printf ; Call printf

    ; Clean up and return
    leave
    ret
```

```
section .data
    int_format db "Hello, world.",10,0
    global _main
    extern _printf

section .text
_main:
    push rbp
    mov rbp, rsp ; move the stack pointer into the base pointer

    ; Set up for calling printf
    lea rdi, [rel int_format] ; Load address of format string into rdi
    mov rax, 0 ; Zero rax to indicate no floating-point arguments are passed
    call _printf ; Call printf

    ; Clean up and return
    leave
    ret
```

Different **sections** of the file. Common segments include data (read only, BSS, ...) and **.text**, which is where the **code** gets put

Focusing just on the _main function

_main:
Initialization

```
push rbp  
mov rbp, rsp ; move the stack pointer into the base pointer
```

```
; Set up for calling printf  
lea rdi, [rel int_format] ; Load address of format string into rdi  
mov rax, 0 ; Zero rax to indicate no floating-point arguments are passed  
call _printf ; Call printf
```

```
; Clean up and return  
leave  
ret
```

Call printf

Return

Assembly Progression

- Programming in assembler could easily take a whole course; tons of nuanced concepts, which differ widely depending on the OS/ABI/compiler/linker/...
- I will show **x86-64** (i.e., AMD 64-bit assembler, extending and compatible with Pentium x86)
 - Possible to cross-compile x86-64 to run on M2 Mac (I have one!) using Rosetta, will see how
 - ARM Assembly is also common
- I will show (mostly) **NASM** (Netwide assembler) syntax, though I may occasionally mess up
 - There are many different types of assemblers, MASM, GAS, NASM,

Registers: Blazing-Fast Variables

- **Registers:** the main data structures over which instructions operate
- All modern laptops are 64-bit: this means that registers are 64 bits.
- Registers are used as **pointers** in C, and thus 64-bit machines may address up to 2^{64} bytes of memory; if you do the math 2^{32} bits is only around 4GB of RAM, 2^{64} is a big improvement!
- Instructions will take inputs in registers (sometimes literals are allowed) and store the output to a result register

Originally, 8-bit registers: al, bl, cl, dl

Traditionally, x86 architectures only had **four** 16-bit general purpose registers: ax, bx, cx, dx

Also other registers: bp, sp, di, si

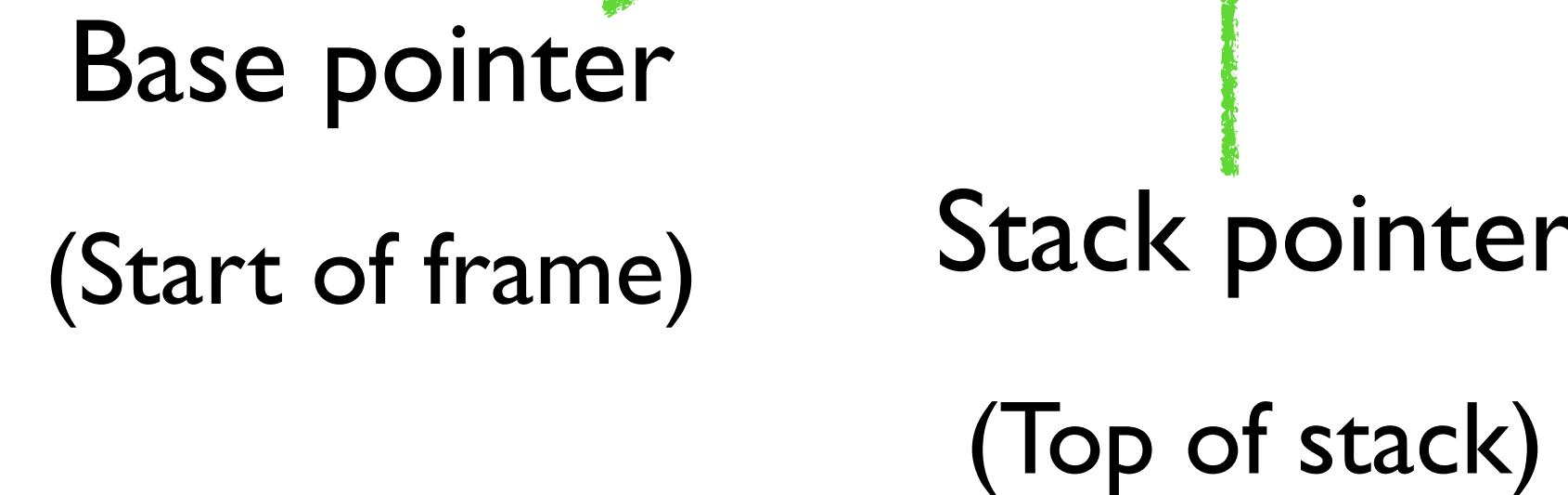
Base pointer
(Start of frame)

A diagram illustrating the relationship between the Base pointer (bp) and the Stack pointer (sp). The Base pointer is labeled 'Base pointer' with the subtitle '(Start of frame)' and is shown pointing to the left. The Stack pointer is labeled 'Stack pointer' with the subtitle '(Top of stack)' and is shown pointing upwards. Two green arrows connect these labels to their respective points on the diagram.

Originally, 8-bit registers: al, bl, cl, dl

Traditionally, x86 architectures only had **four** 16-bit general purpose registers: ax, bx, cx, dx

Also other registers: bp, sp, di, si



IP: instruction pointer

Points at current instruction,
incremented after each instruction

FLAGS: holds flags

Set on subtraction, comparison, etc..

Traditionally, x86 architectures only had **four** 16-bit general purpose registers: ax, bx, cx, dx

Also other registers: bp, sp, di, si

As time progressed, also added 32-bit registers: eax,
ebx, ecx, edx

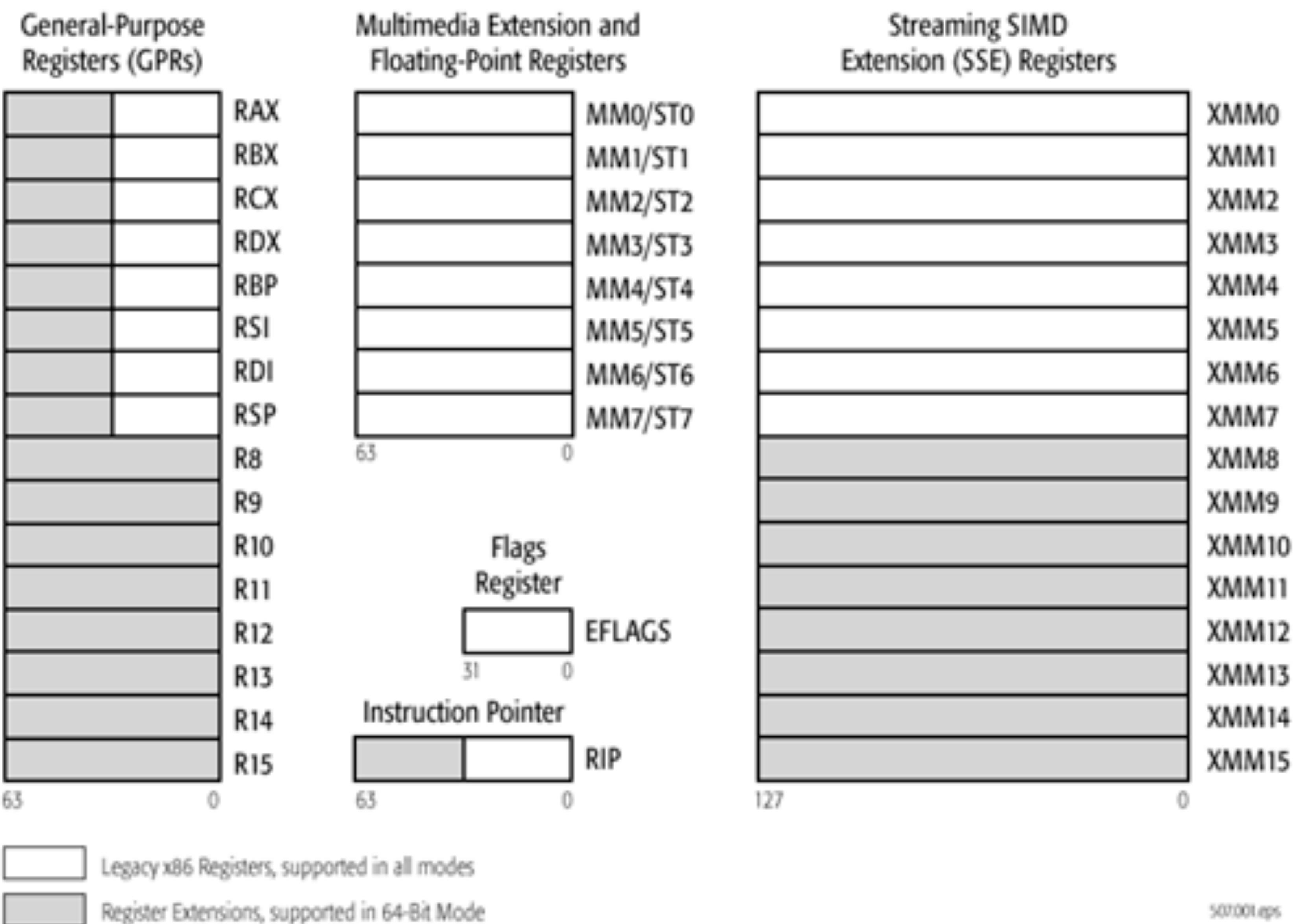
In past few years, 64-bit registers: rax, rbx, rcx, rdx
(Also 64-bit versions: rip, etc..)

We'll pretty much exclusively use
64-bit registers!

Note RAX is an **extension** of EAX



If you change EAX, you change lower 32 bits of RAX

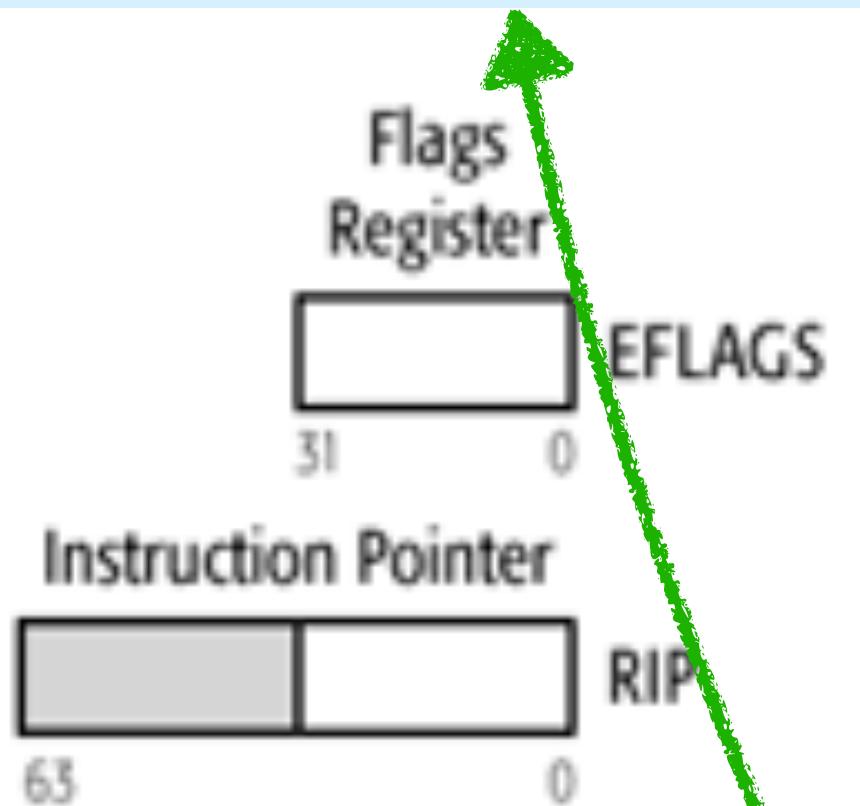


General-Purpose Registers (GPRs)

	RAX
	RBX
	RCX
	RDX
	RBP
	RSI
	RDI
	RSP
63	R8
63	R9
63	R10
63	R11
63	R12
63	R13
63	R14
63	R15
0	0

Multimedia Extension and Floating-Point Registers

MM0/ST0
MM1/ST1
MM2/ST2
MM3/ST3
MM4/ST4
MM5/ST5
MM6/ST6
MM7/ST7



Streaming SIMD Extension (SSE) Registers

XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM14
XMM15
0



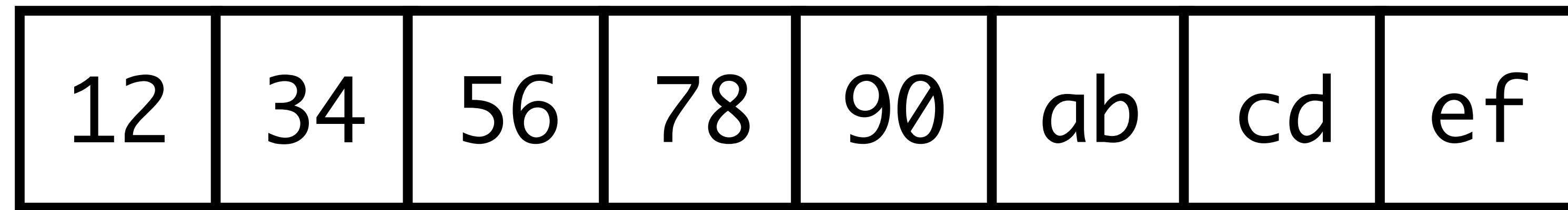
Legacy x86 Registers, supported in all modes



Register Extensions, supported in 64-Bit Mode

Special regs: floating-point / matrix ops

To represent `0x1234567890abcdef`



Most Significant Byte

Least Significant Byte

x86 is a **little-endian** architecture

If an n-byte value is stored at addresses a to $a+(n-1)$ in memory,
byte a will hold the **least significant byte**

0x1234567890abcdef

Exercise with partner

Instructions

Binary code is made up of giant sequences of “instructions”

Modern Intel / AMD chip has hundreds of them, some very complex

Moving memory around

Arithmetic

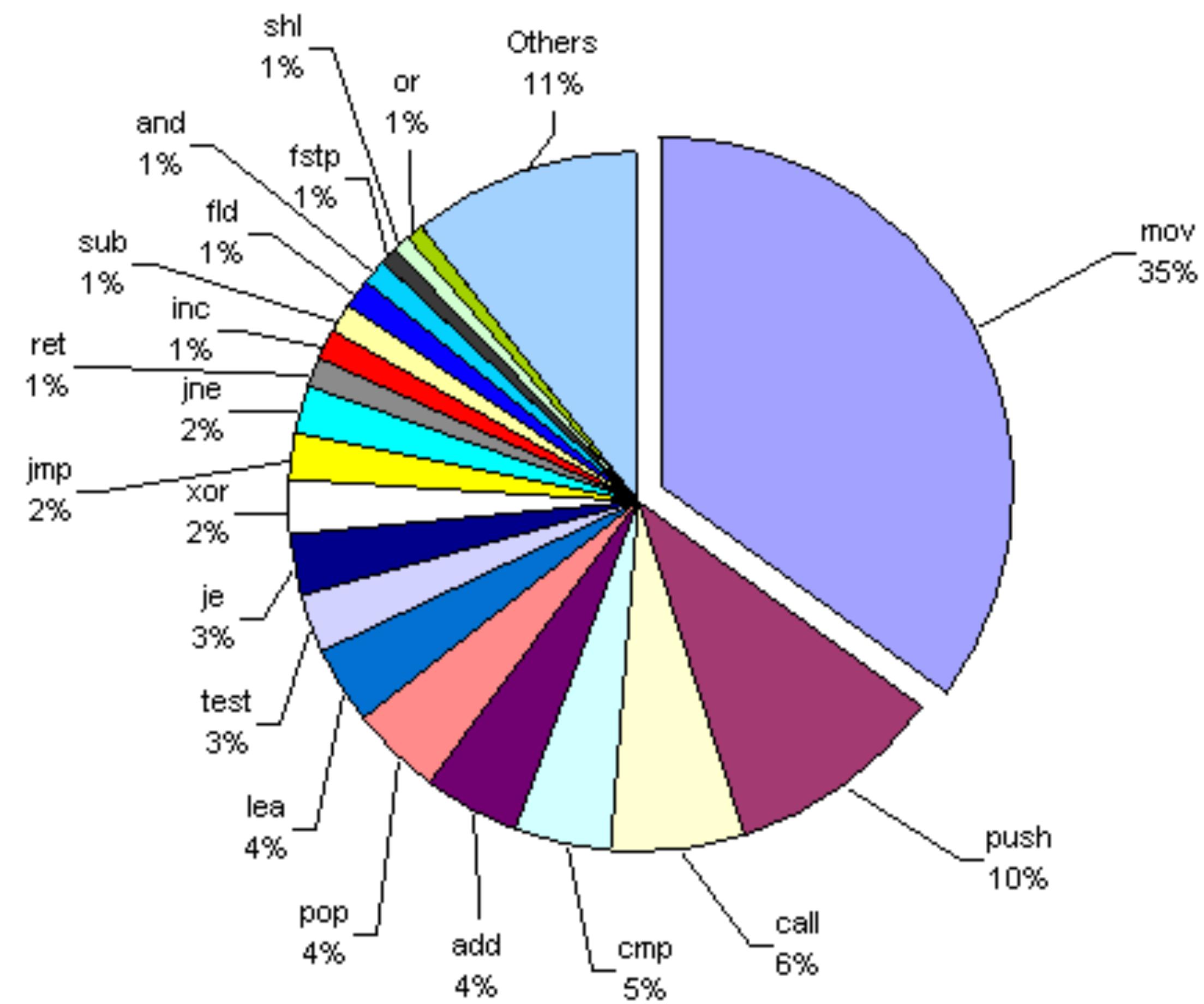
Branch / If

Matrix operations

Atomic-Instructions

Transactional memory instructions

Top 20 instructions of x86 architecture



Plurality of instructions
are **movs**

Then **push**

Then **call**

Intraprocedural Instructions

- Today, we'll learn a few types of instructions:
 - **mov** — move values around / load / store
 - **Arithmetic / logical operators** — operate on registers
 - **Comparison instructions** — loads EFLAGS register
 - **(Un)conditional jumps** — to labels, based on EFLAGS
- Next lecture, we'll look more closely at functions, stack frames, function calls, and calling conventions.

Several addressing modes

Addressing modes allow us to speak about where data is: we can load data from other registers, from constants (immediate), or from other memory.

“Move the value from register rbx into the register rax”

	Destination	Source
mov	rax,	rbx
Opcode name		

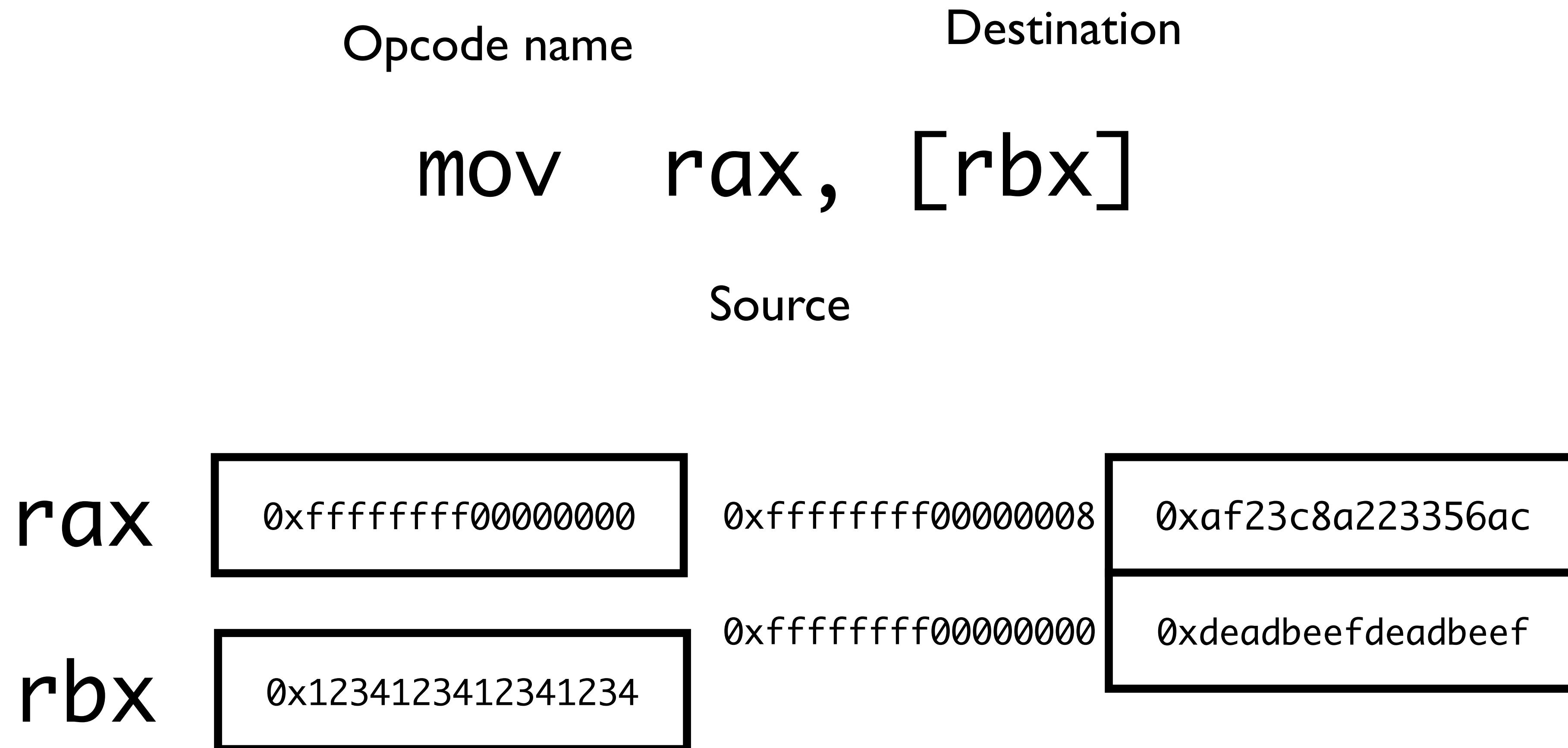
Memory: a **giant chunk of bytes**

You can load from and store to it using **pointers**

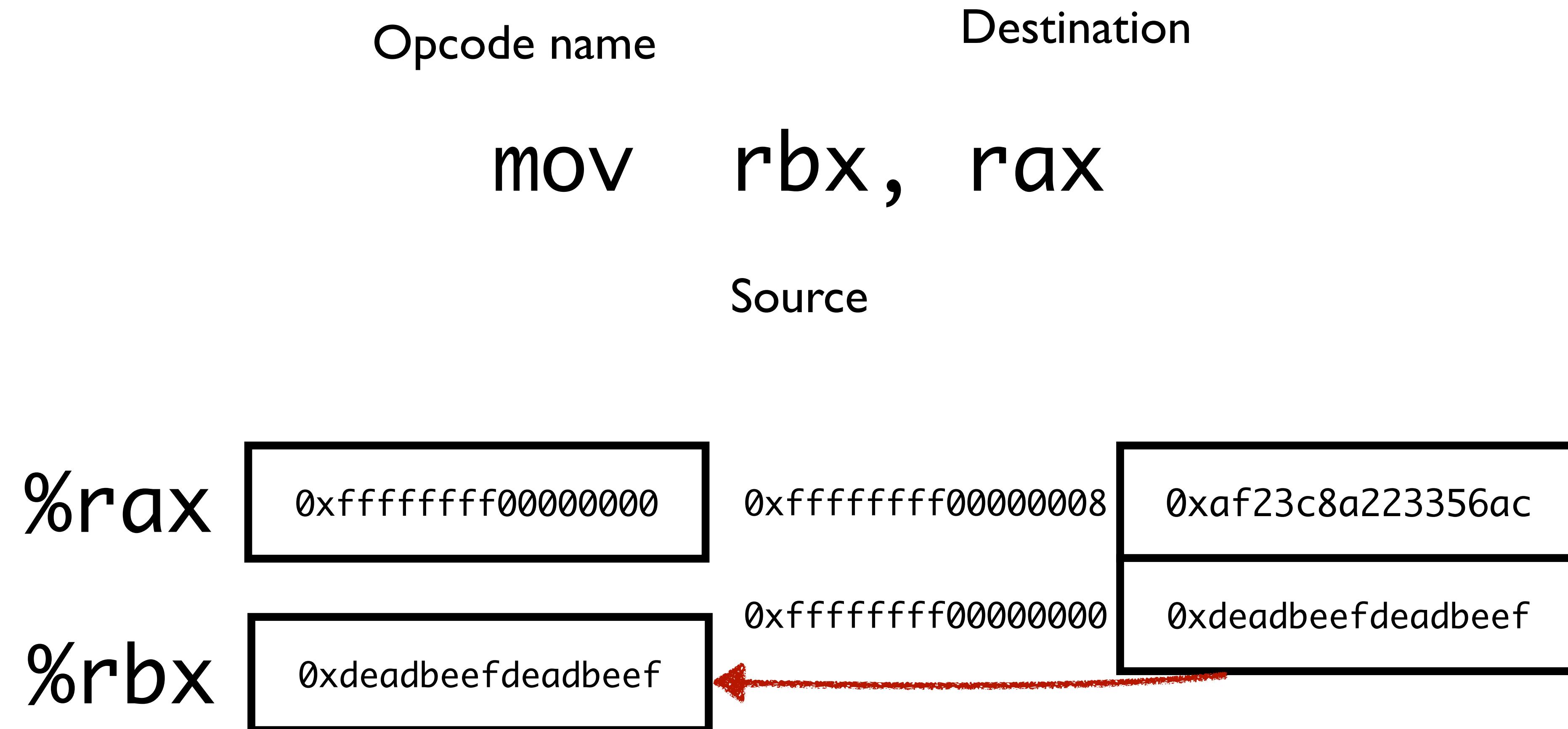
```
mov rax, [rbx]
```

“Move the 64-bit value stored at the location pointed to by rbx into tax”

“Move the value **at address rbx** into register **rax**”



“Move the value **at address** rax into register rbx”



Memory: a **giant chunk of bytes**

You can load from and store to it using **pointers**

```
mov rax, [rax + rdi*8 + 500]
```

You **can't** move memory-to-memory

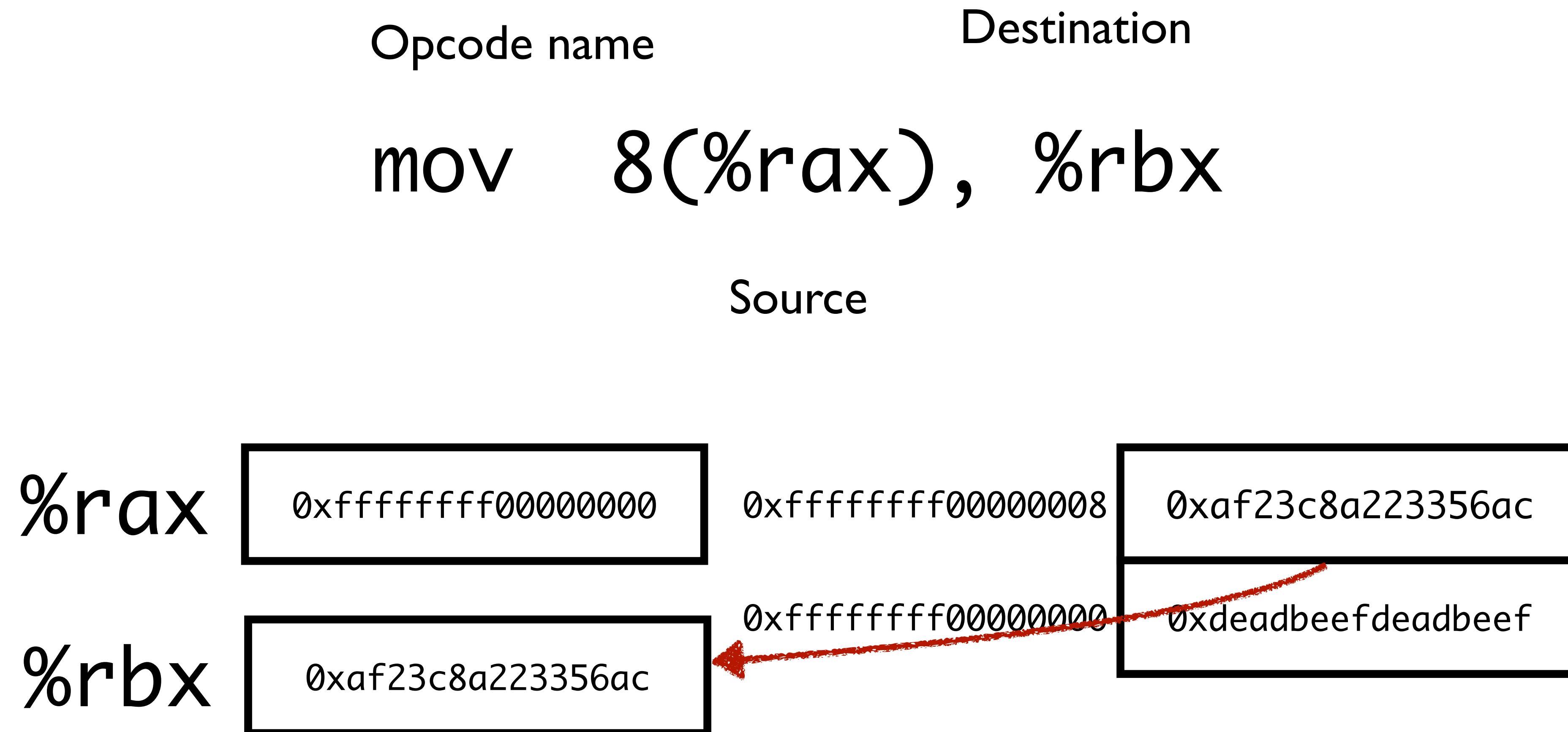
WRONG!

mov [rbx], [rax]

First, load into an intermediary register

```
mov rcx, [rax]  
mov [rbx], rex
```

“Move the value **at address** %rax+8 into register %rbx”



A few other more complicated ones that allow
you to add registers, offsets, etc...

Different instructions allow different addressing-modes

“Move the value from register rbx into the register rax”

	Destination	Source
mov	rax,	rbx
	Opcode name	

Arithmetic operations

In NASM, written destination-first, source-last

	Destination	Source
add	rax,	rbx

Semantics is:
 $rax += rbx$

List of arithmetic / logic instructions

add, sub, imul, idiv, inc, dec, neg, ...

Bitwise Logic Operations

and, or, xor, not, shl, shr, sal,
sar, rol, ror, ...

Comparison operators

The comparison instructions **cmp** and **test** set the FLAGS register, which will subsequently influence how conditional jump instructions (jg, jz, jge, ...) behave

```
_start:
    mov rax, 5 ; First number to compare
    cmp rax, 3 ; Compare first number with second number

    ja .greater ; Jump if above (unsigned comparison)
    jmp .less_or_equal

.greater:
    ; Print message_gt
    mov rdi, message_gt ; Address of the message
    call print_string
    jmp .exit

.less_or_equal:
    ; Print message_le
    mov rdi, message_le ; Address of the message
    call print_string
    jmp .exit
```

Conditional jumps such as **jz** (“jump if last comparison was **zero**,” i.e., equal), or **jge** (“jump if last comparison was **greater than or equal to**”).

jmp — **Unconditional** jump

je / jz — jump if zero (equal) flag is set

jne / jnz — jump if not zero (equal)

js — jump if sign

jg — jump if greater

jl — jump if less

jle — jump if less than or equal to

An **unconditional jump** jumps to a label unconditionally.

```
section .text
global _start

_start:
    ; Do something
    JMP somewhere_else ; Jumps to the label "somewhere_else"

somewhere_else:
    ; Execution continues here after the jump
    ; Do something else
```

From Instructions to Functions

- Putting instructions together forms functions.
- Instructions execute one-after-another, in absence of (un)conditional jumps.
- Now, we want to study how to use multiple instructions to build **computations**
- One obvious challenge: **registers are limited!**
- A big computation might require us to be very careful with how we use registers.

Setup

```
_main:
```

```
    push rbp  
    mov rbp, rsp
```

```
    mov rax, 3  
    mov rbx, 5  
    imul rax, rbx  
    mov rbx, 4  
    add rax, rbx
```

```
; Clean up and return
```

```
leave
```

```
ret
```

Return

The main part of the program has five instructions:

- ▶ Move 3 into rax
- ▶ Move 5 into rbx
- ▶ Multiply rbx by rax, leave result in rax
- ▶ Move 4 into rbx
- ▶ Add rax and rbx, leave result in rbx

Exercise:

- ▶ Load 10 into rax
- ▶ Load 20 into rbx
- ▶ Load 15 into rcx
- ▶ Shift rbx by 2 (use shr)
- ▶ Multiply rcx by rbx, leave result in rcx
- ▶ Add result to rax, leave result in rax

_main:

```
push rbp  
mov rbp, rsp  
; ; YOUR CODE HERE
```

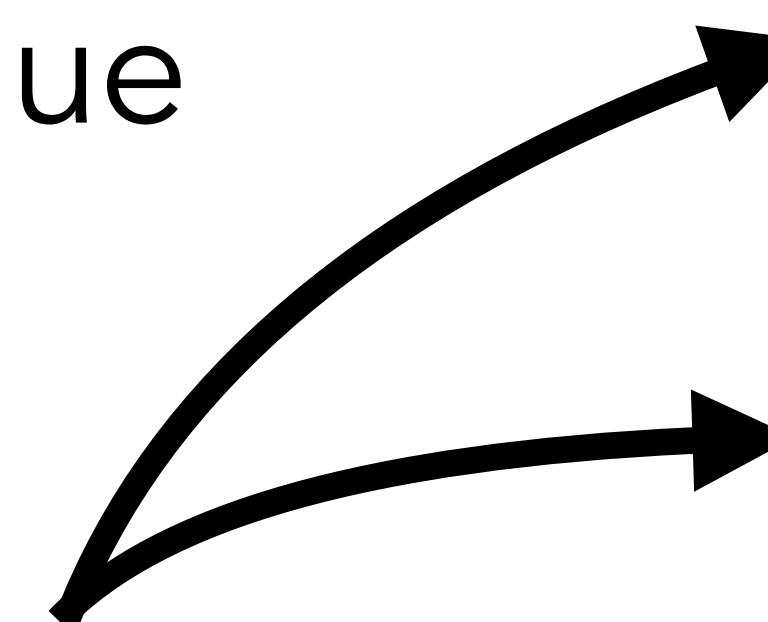
```
leave  
ret
```

Possible to drive control-flow by
using **jnz/jmp/...**

Example: using **cmp** to compare
a register to a specific value

Notice: tag branches with **labels**

```
_main:  
    push rbp  
    mov rbp, rsp  
  
    mov rax, 5  
    sub rax, 8  
    cmp rax, 0  
    jnz not_zero  
    jmp zero  
  
zero:  
    mov rax, 15  
    jmp done  
  
not_zero:  
    mov rax, 20  
  
done:  
    leave  
    ret
```



Compiling Complex Expressions

An issue: x86-64 instructions don't allow **nesting**, expressions like $(x + 5) * (y - 2)$ must be broken down into sequences of instructions:

```
// assume x in rax, y in rbx
mov rcx, 5
add rax, rcx // rax := x + 5, notice, rax changed!
mov rcx, 2
sub y, rcx // rbx := y - 2
imul rax, rbx // result in rax
```

- Unfortunately, instructions like add **trash registers**
 - Can't use value of x ever again!
 - Registers availability can get **very tight**, but not so much anymore (modern CPUs tend towards more general-purpose registers)
 - May need to be smart about **register allocation**

Solution: Virtual Registers

- **Assume** we have enough registers
- Each subexpression assigned a virtual register
- Map virtual registers to actual registers later
- All arguments to functions are **atoms**
- (Typically) values assigned exactly once (SSA, more on this later)

(+ (* x (+ y 2))
(+ z (- y x)))

ANF Conversion
r3)



(let* ([r0 (+ y 2)]
[r1 (* x r0)]
[r2 (- y x)])
[r3 (+ z r2)])

```
(let* ([r0 (+ y 2)]  
      [r1 (* x r0)]  
      [r2 (- y x)]  
      [r3 (+ z r2)])  
  r3)
```

To “virtual”
assembly



```
mov r0, y  
add r0, 2  
mov r1, x  
imul r1, r0  
mov r2, y  
sub r2, x  
mov r3, z  
add r3, r2
```

```
mov r0, y  
add r0, 2  
mov r1, x  
imul r1, r0  
mov r2, y  
sub r2, x  
mov r3, z  
add r3, r2
```

Register allocation
turns this into
actual x86-64



```
// For example, if...  
// y = [rdx]  
// x = [rdx+8]  
// z = [rdx+16]  
mov rax, [rdx]  
add rax, 2  
mov rbx, [rdx+8]  
imul rbx, rax  
mov rcx, [rdx]  
sub rcx, [rdx+8]  
mov rcx, [rdx+16]  
add rcx, rbx
```

If we run out of space in registers (common), we'll need to store values somewhere else.

To do this, we use RAM, typically via the stack / heap.

We move values into / out of registers—the values we're working with have to be shuffled into and out of RAM via **mov** instructions. More on this **next time**

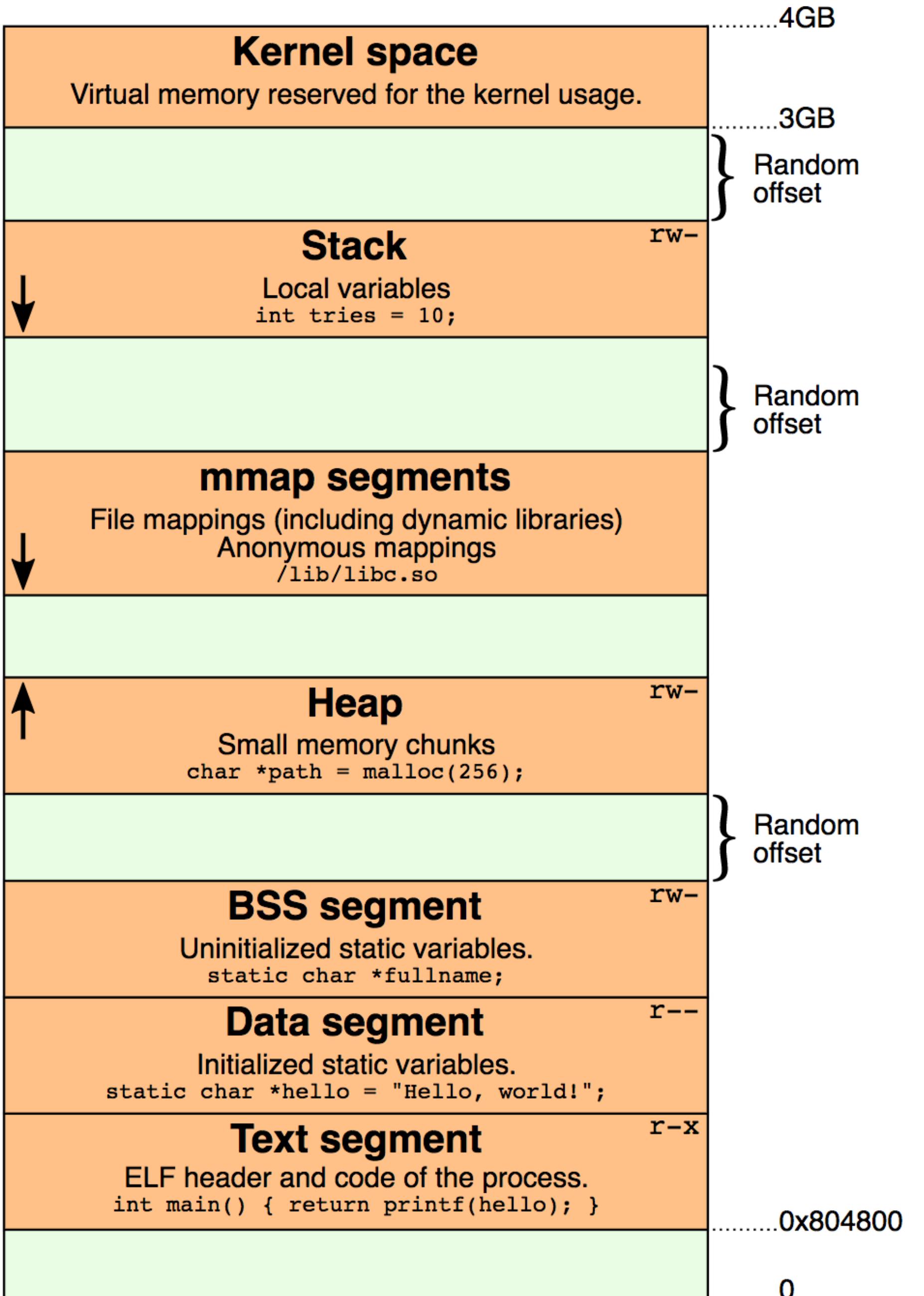
We'll wrap up today by looking at how OS loads the program

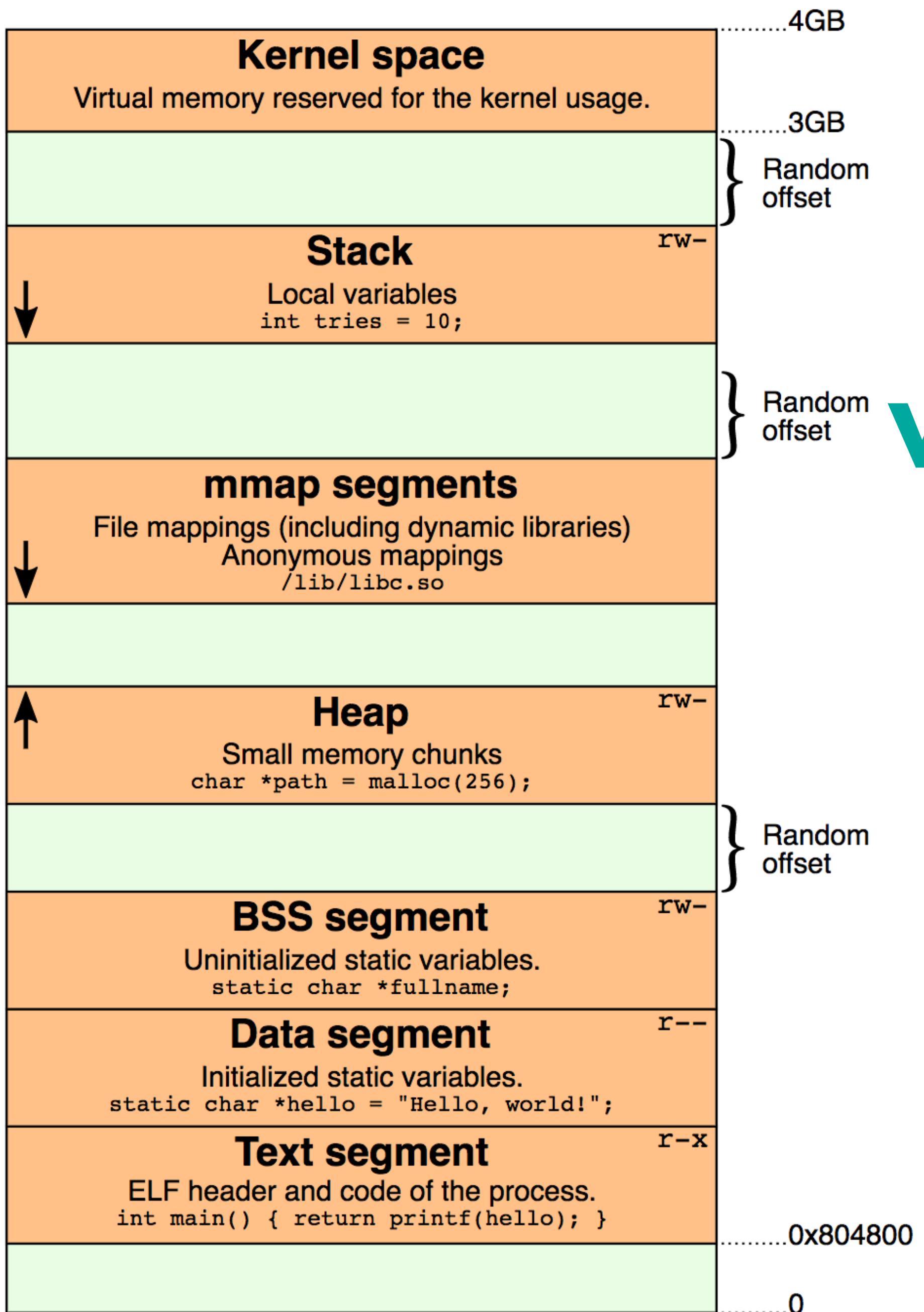
Binary specifies a number of **sections** which describe the program's data (icons, strings, resources, etc...) along with its code.

At runtime, the OS separates these into different **segments**

Kernel memory

Your OS uses it

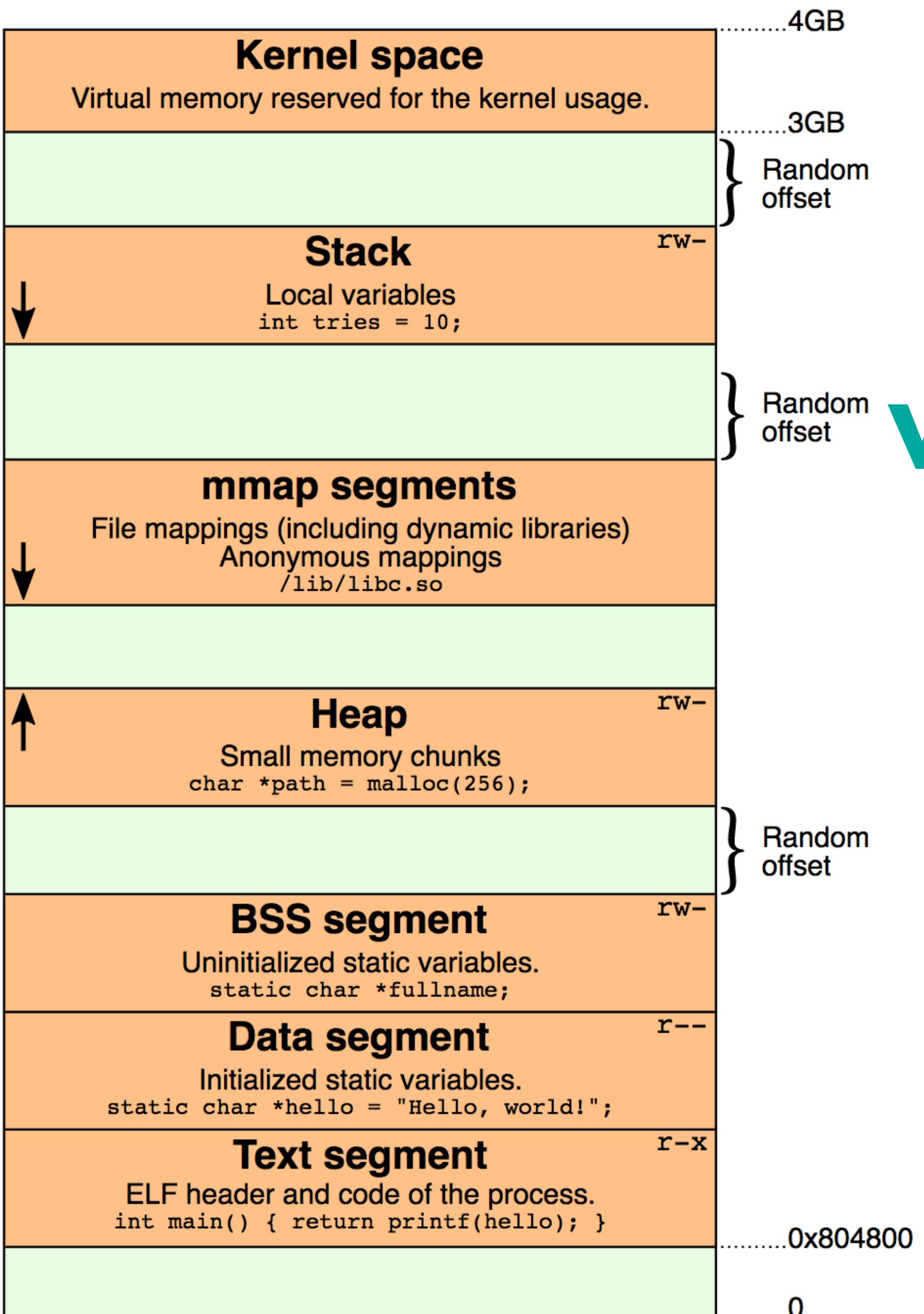




Stack: push / pop

Very important (security):

The stack grows **down**

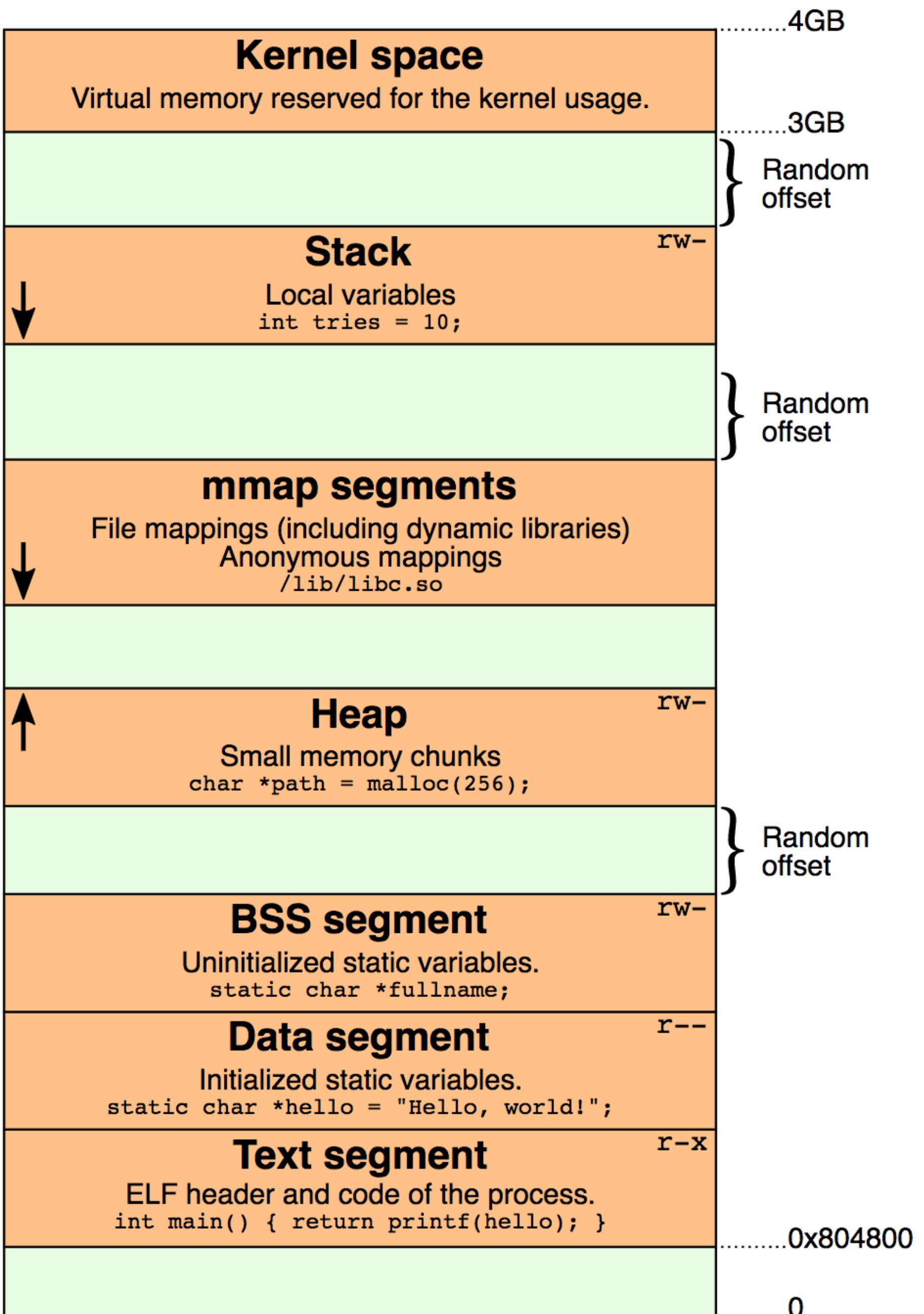


Stack: push / pop

Very important (security):

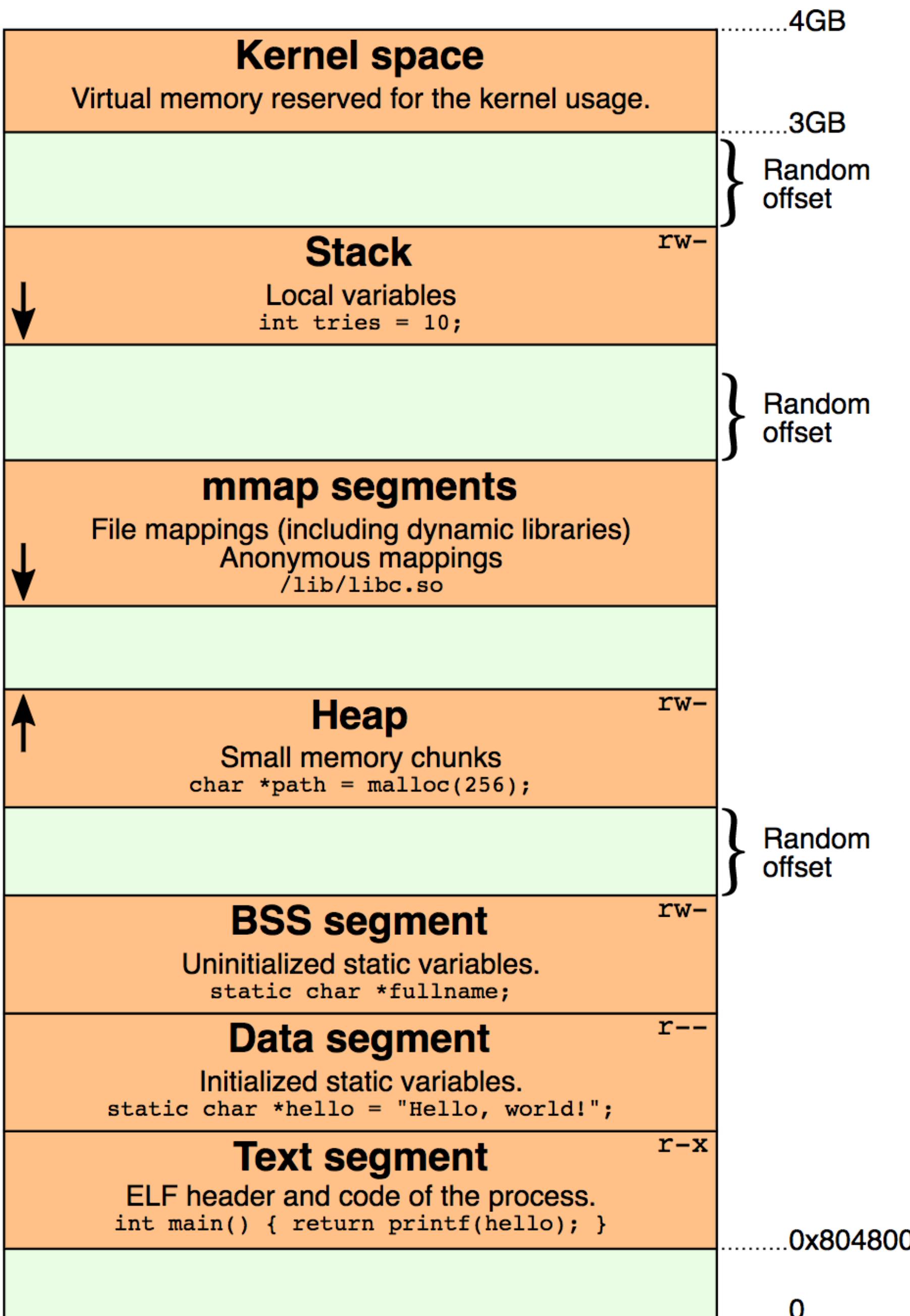
The stack grows **down**

- Stack used for **local variables**
- Stack also **return points** for invoked functions



mmap segments

Allows you to **map** a file to memory

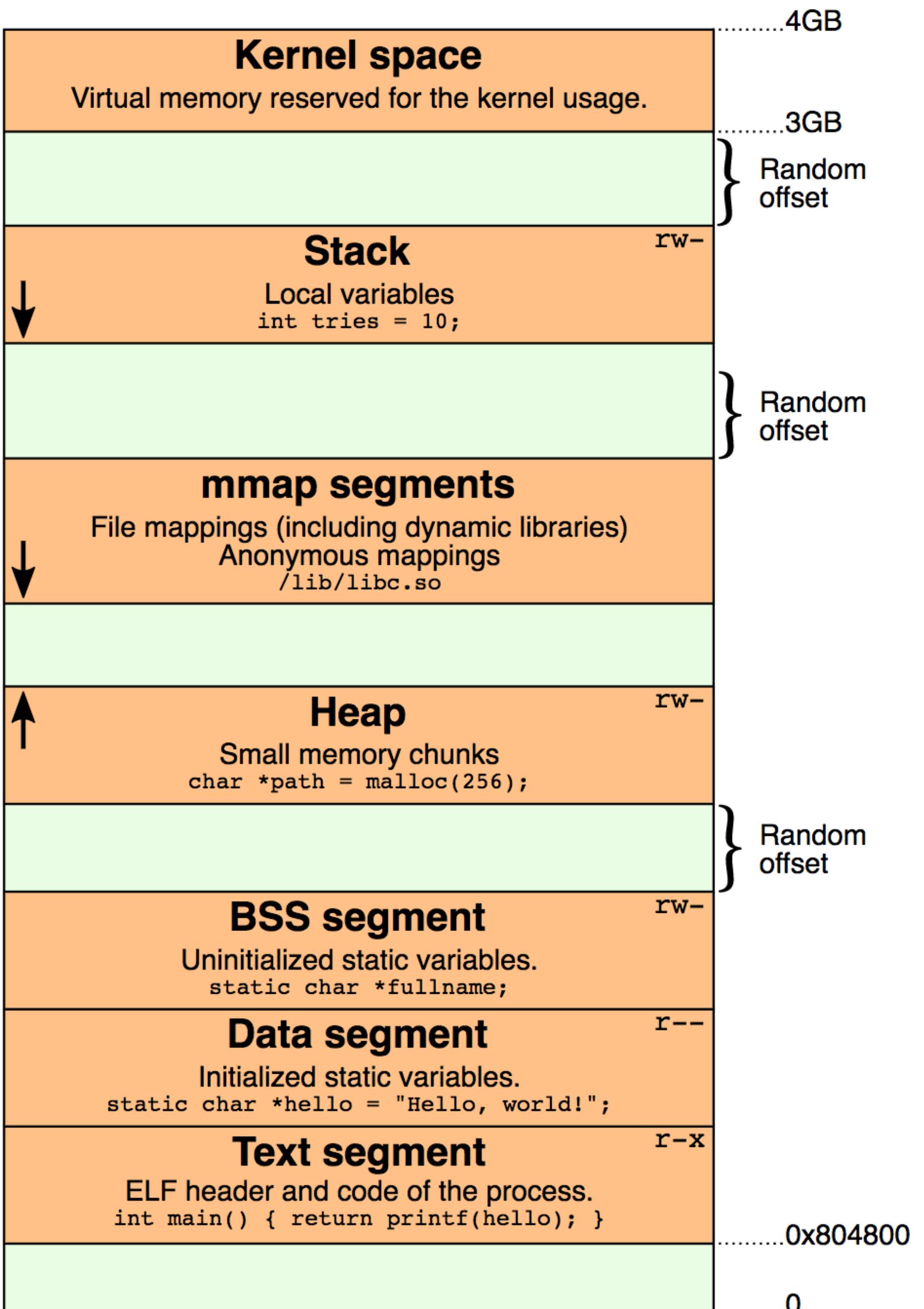


Lots of big objects live on the heap,
especially in modern languages
(Python, Java, Racket, C#, ...)

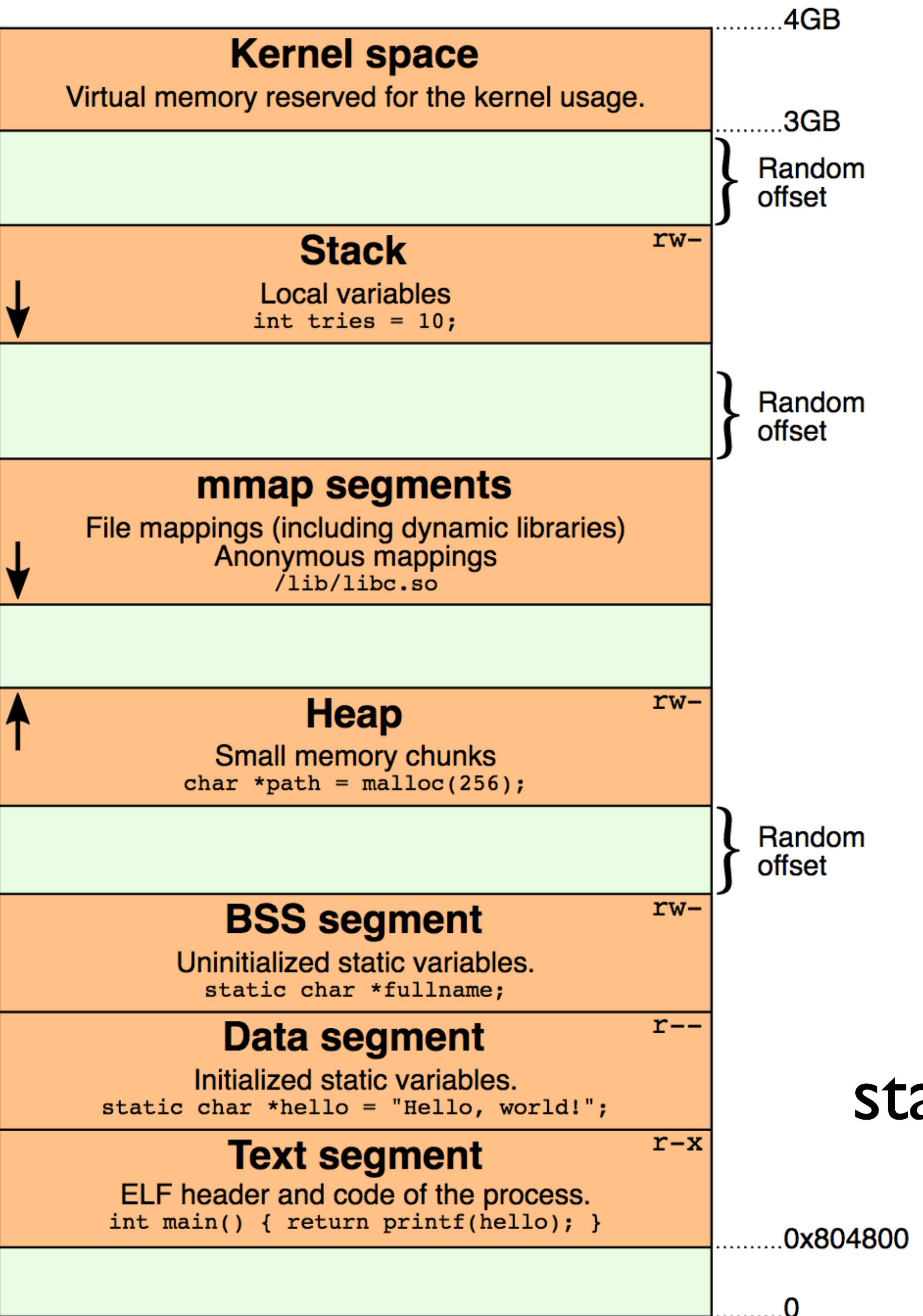
Heap: dynamic allocation

C++: New / delete

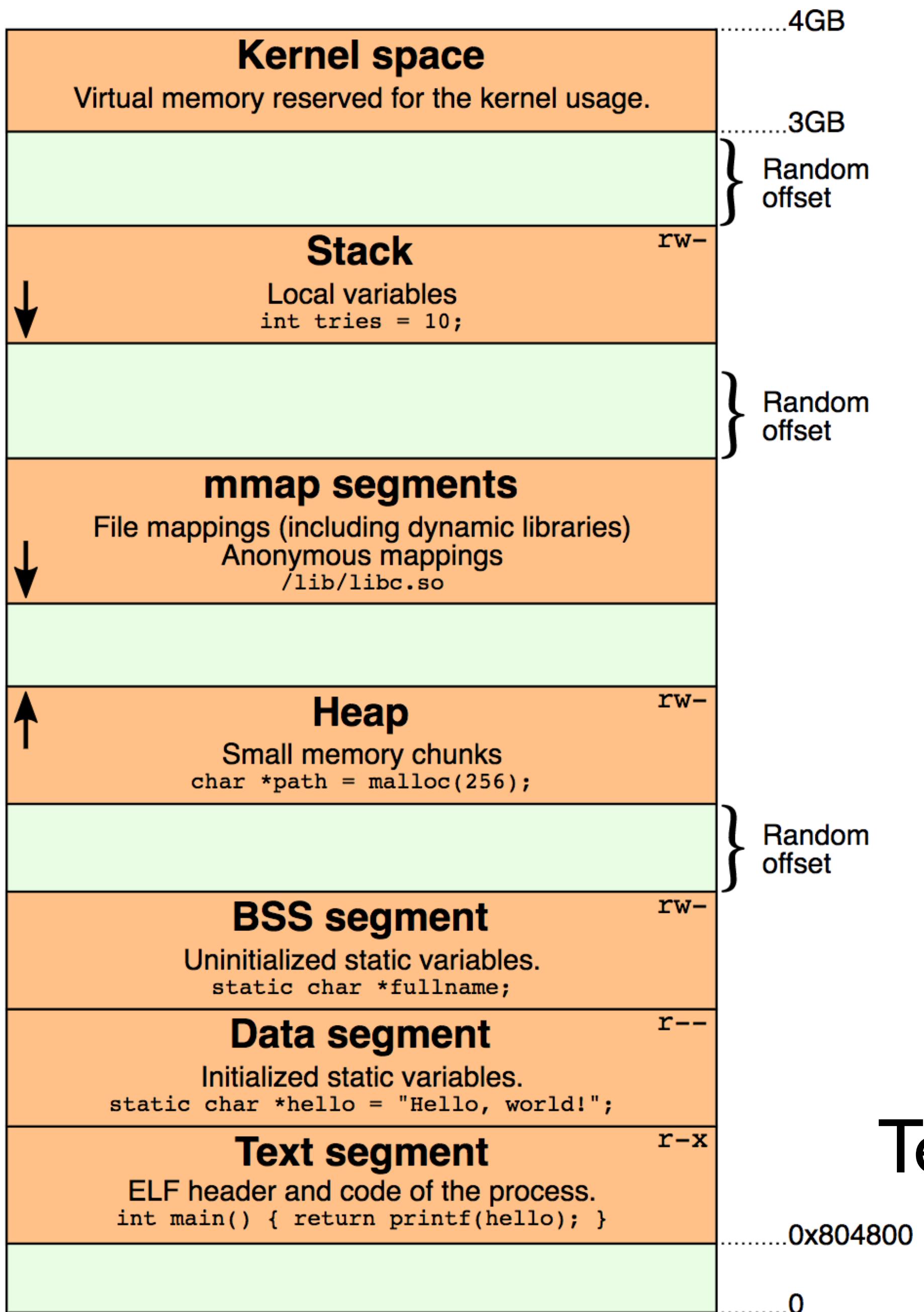
C: Malloc / free



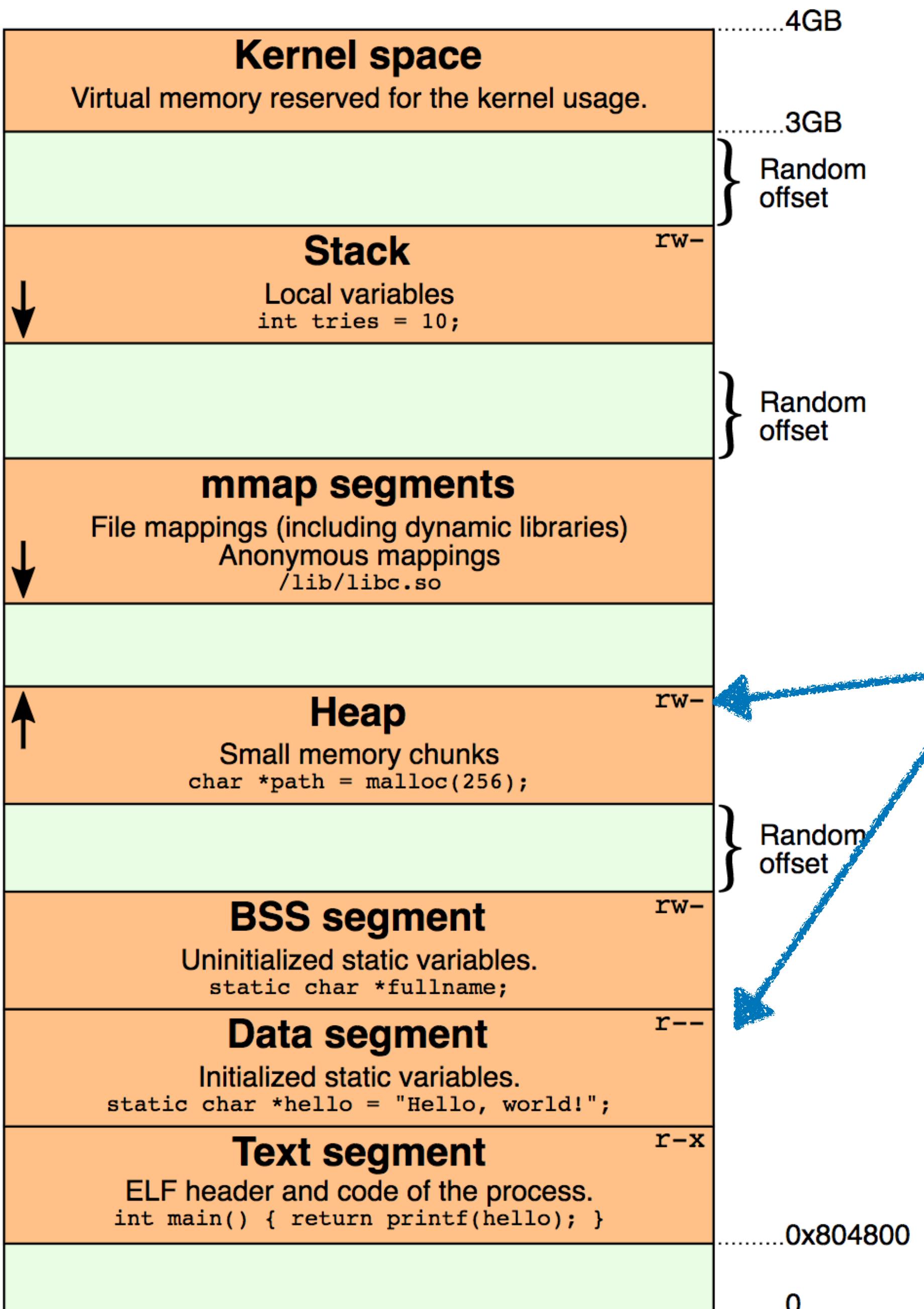
BSS: Uninitialized static vars (globals)



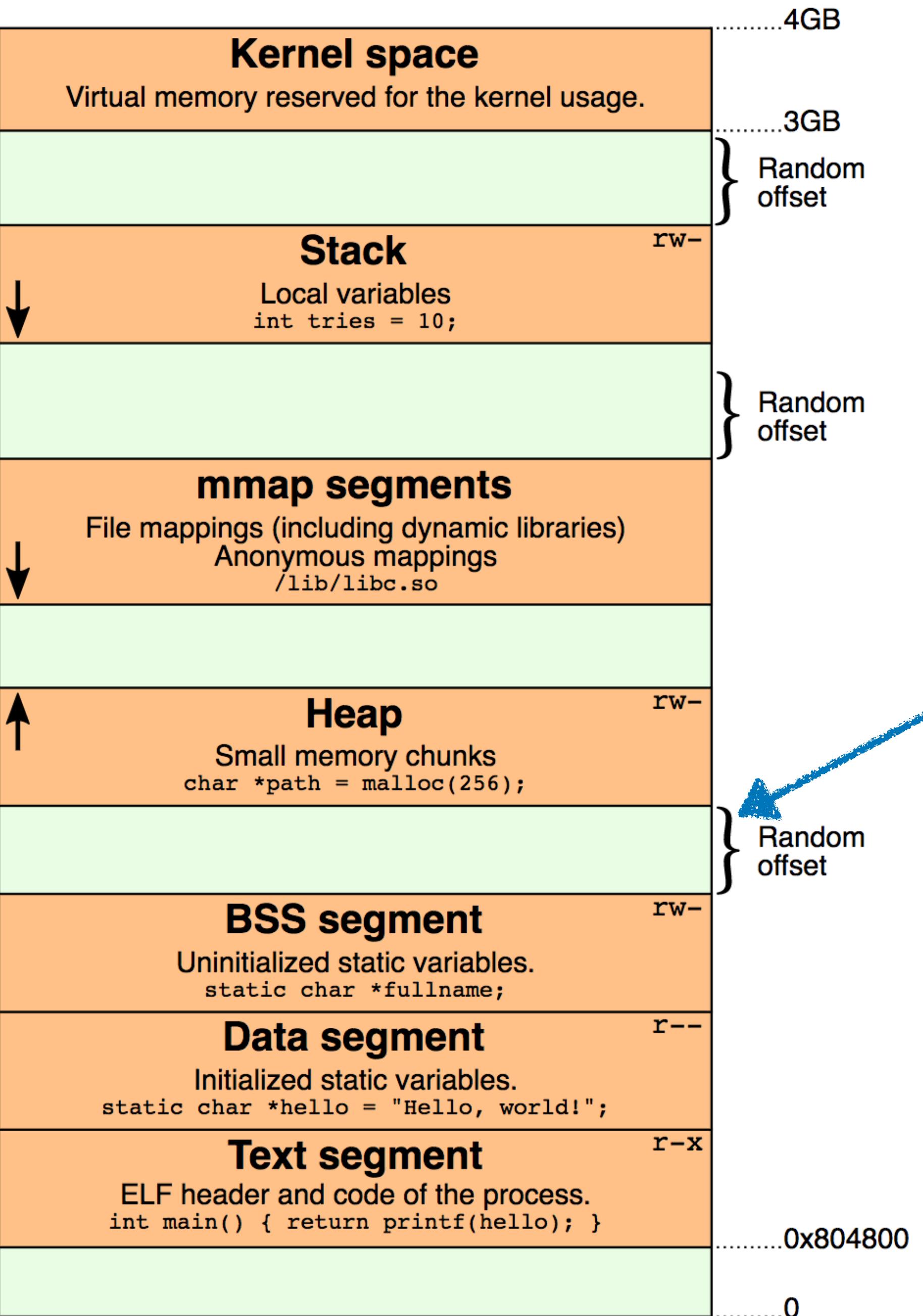
Data segment: initialized statics—e.g., constant strings



Text segment: program code



Note the permissions



This **random offset** really security feature