Lecture 10
Virtual Memory, Pointers, and 1D Arrays

Comp_Sci 213

Introduction to Computer Systems

Wednesday, Oct. 28, 2026

Kate Smith – Fall 2026

Slides adapted from:
Ghena, St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Notes

- Exams will be graded by Friday Oct. 30
  - ***Friday, Oct 30 is the last day to drop a class***
- ***From Syllabus: You need to have a passing grade (64% or higher) averaged between the two exams to pass the course, regardless of your performance in the homework and lab assignments***
- ***BOMB LAB NOW DUE FRIDAY, OCT. 30!***
- We will go over the Short answer/depth exam on Monday, Nov. 2
  - Likely go through T/F today

Topics

- C Memory Layout
- Pointers
- One-dimensional Arrays

Topics

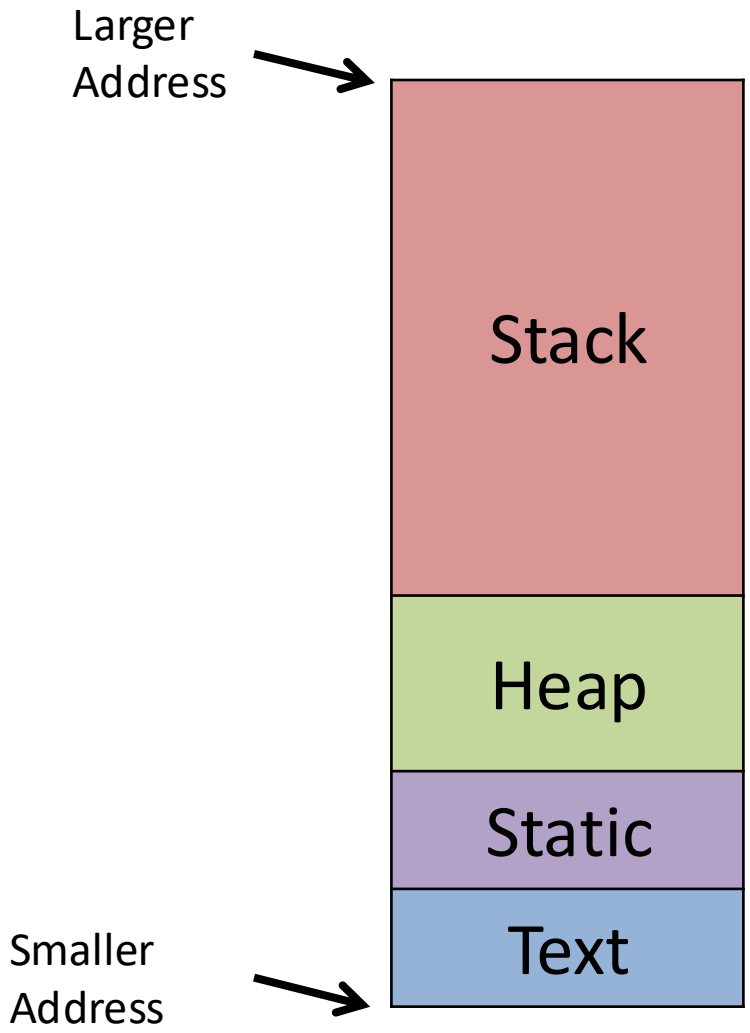
- **C Memory Layout**
- Pointers
- One-dimensional Arrays

Virtual Memory

- ***Virtual memory*** is an abstraction that provides each process with the illusion that it has exclusive use of main memory
  - Each process has the same uniform view of memory known as its ***virtual address space***
- The virtual address space consists of well-defined areas:
  - **Stack** – memory used for local variables within functions and information needed to go to and return from functions. This region expands and contracts during program execution (grows downward).
  - **Heap** – memory used for dynamic memory allocation during program runtime. This region expands and contracts during program execution (grows upward).
  - **Static (Data)** – all data allocated statically outside of functions (global and static variables)
  - **Text** – where code lives, and some variables that are declared with the `const` keyword. This segment cannot be written to by the program.

C memory layout

- Stack Section
  - Local variables (that aren't in registers)
  - Function arguments
- Heap Section
  - Memory granted through `malloc()`
- Static Section (a.k.a. Data Section)
  - Global variables
  - Static function variables
  - String literals
- Text Section (a.k.a Code Section)
  - Program code

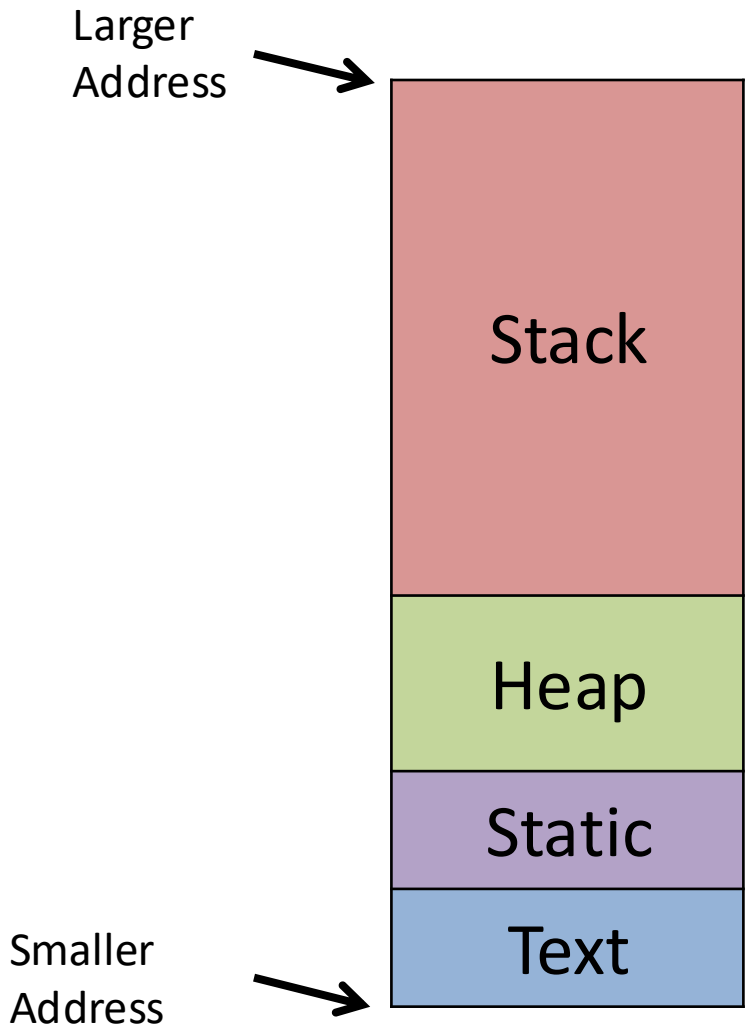


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

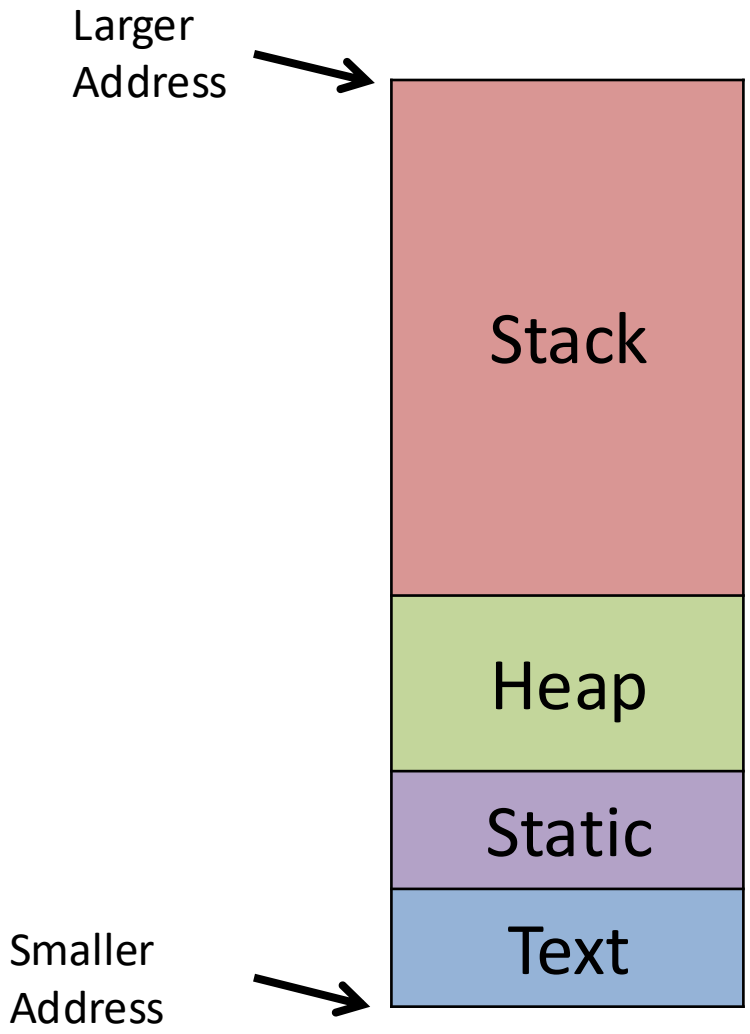


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

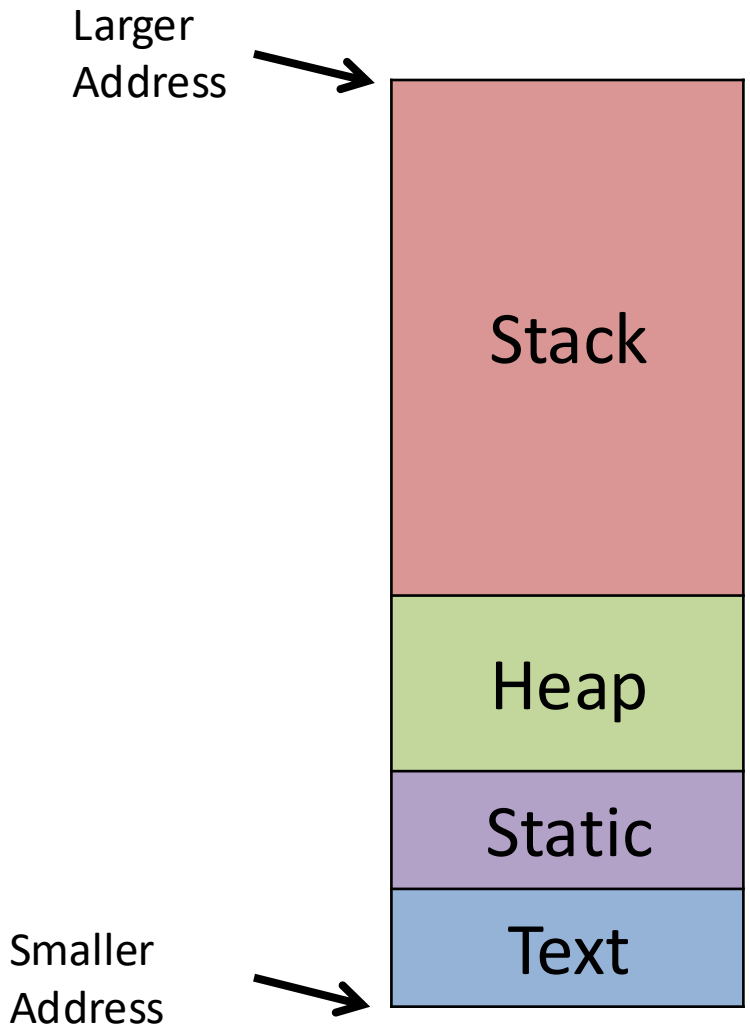


C memory layout

```
char glob_str[80] = {0};
void func short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

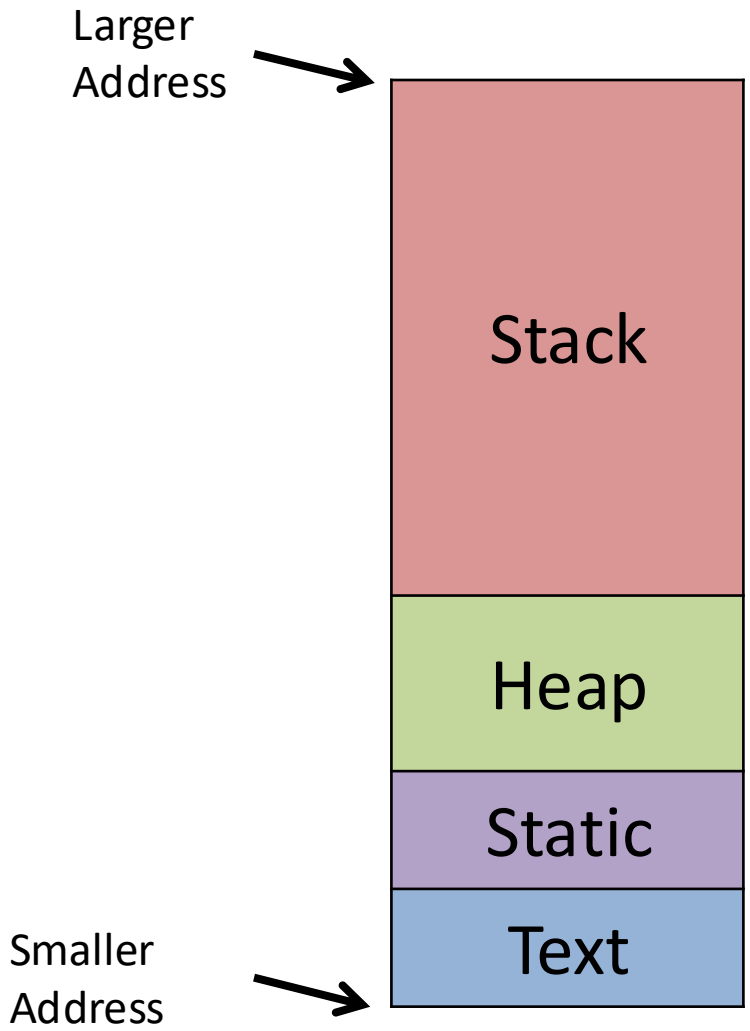


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

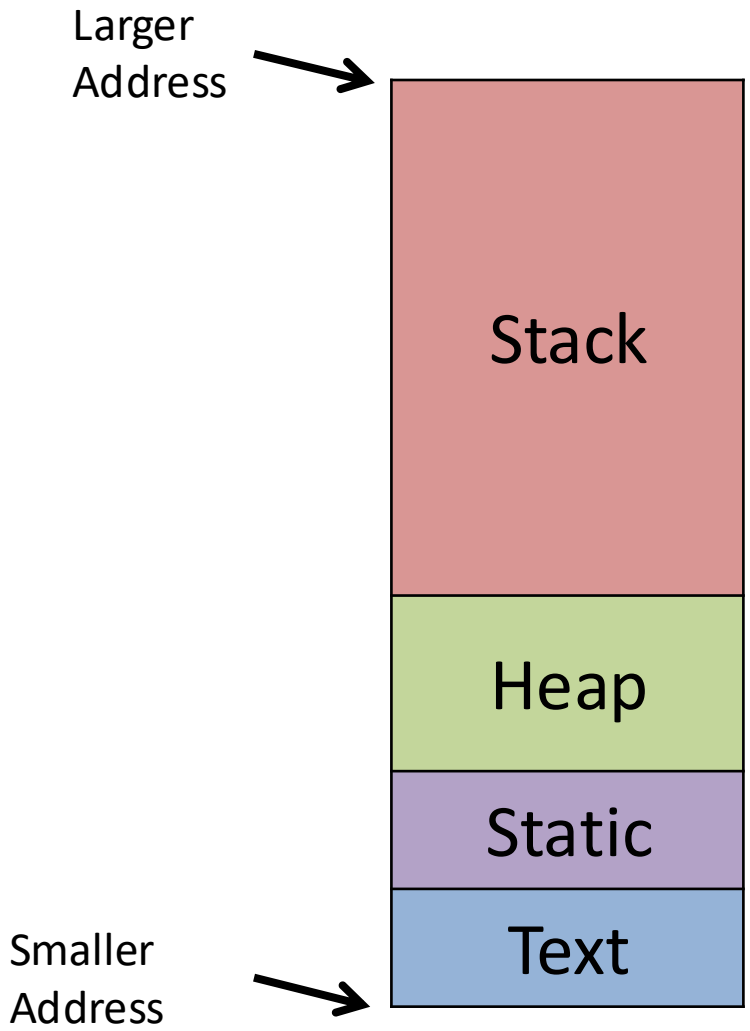


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

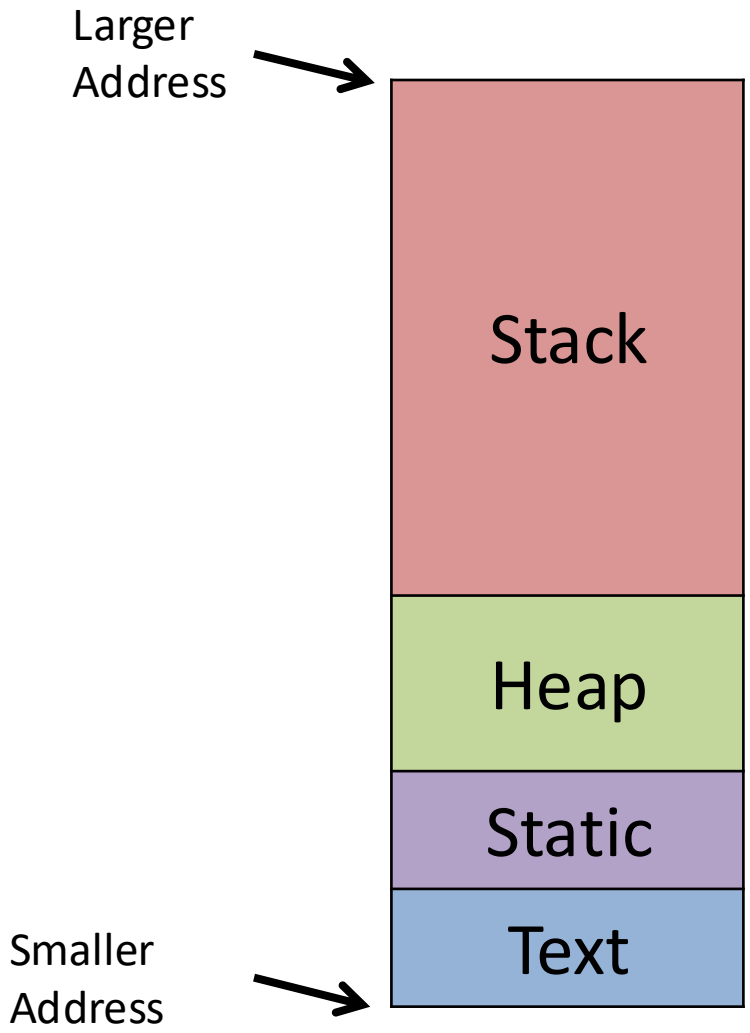


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

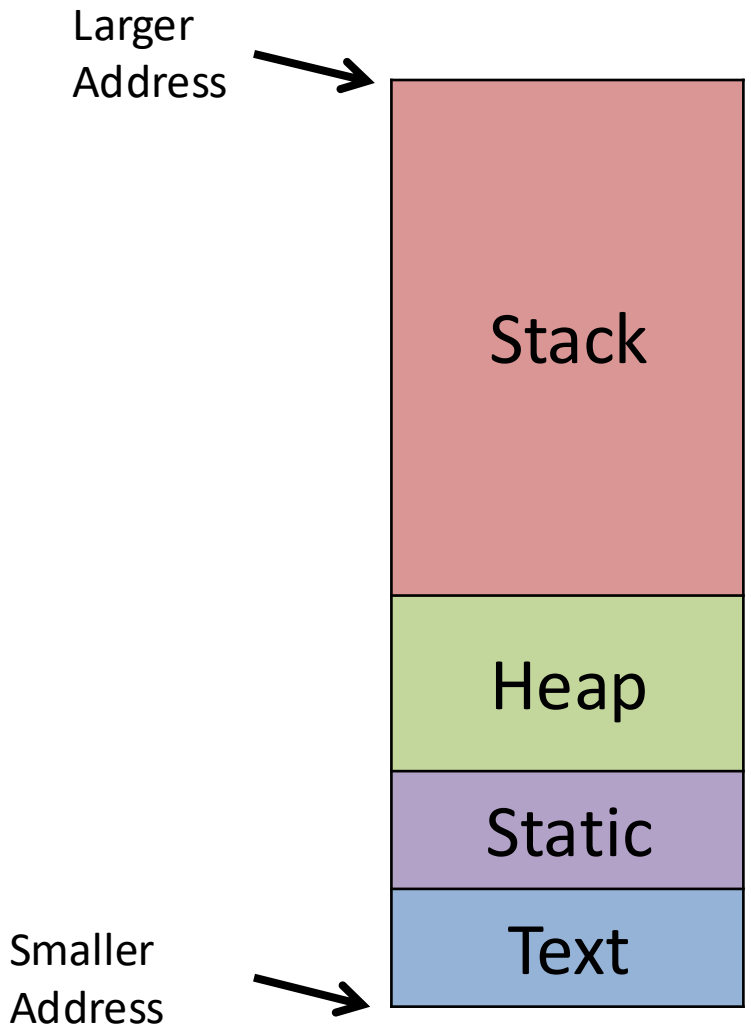


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

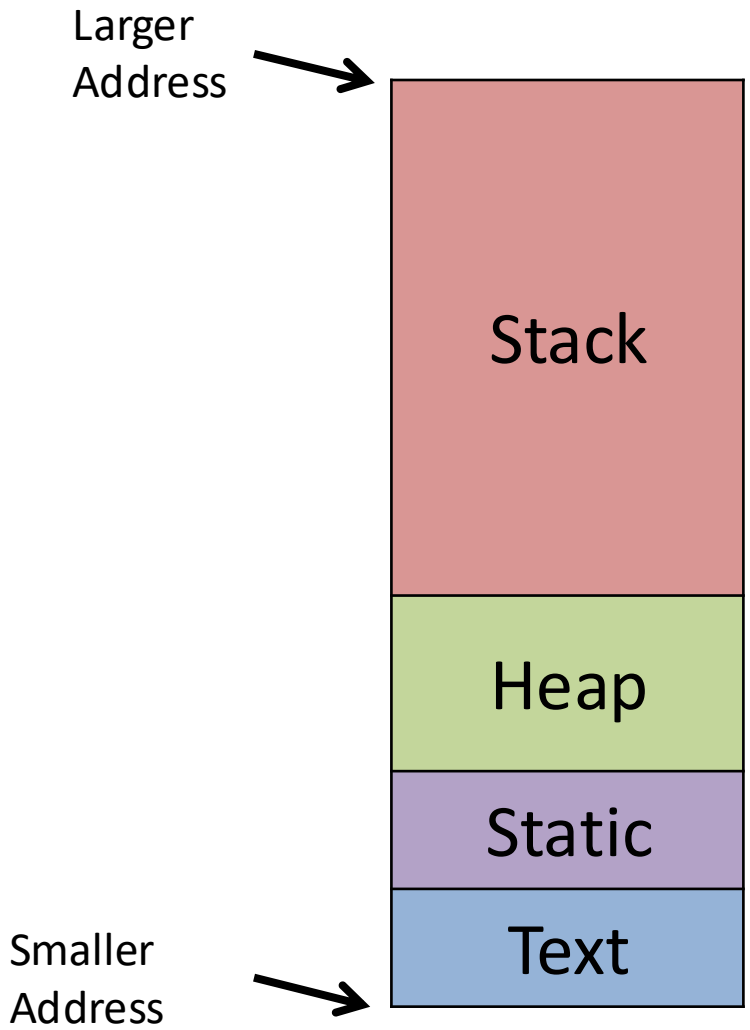


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

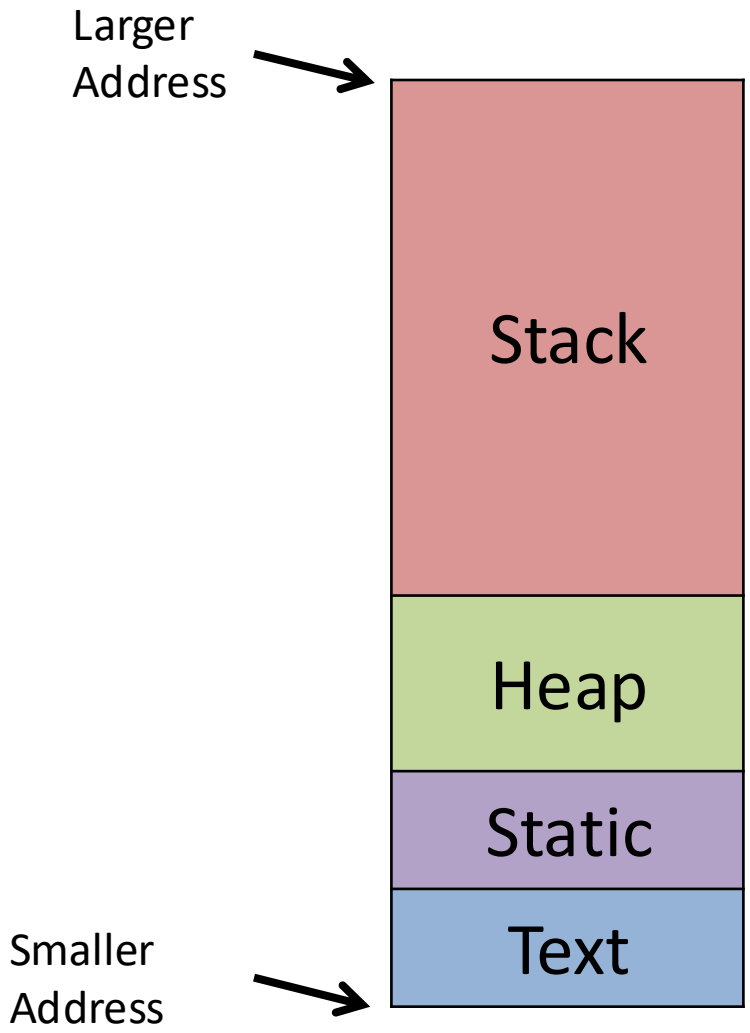


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

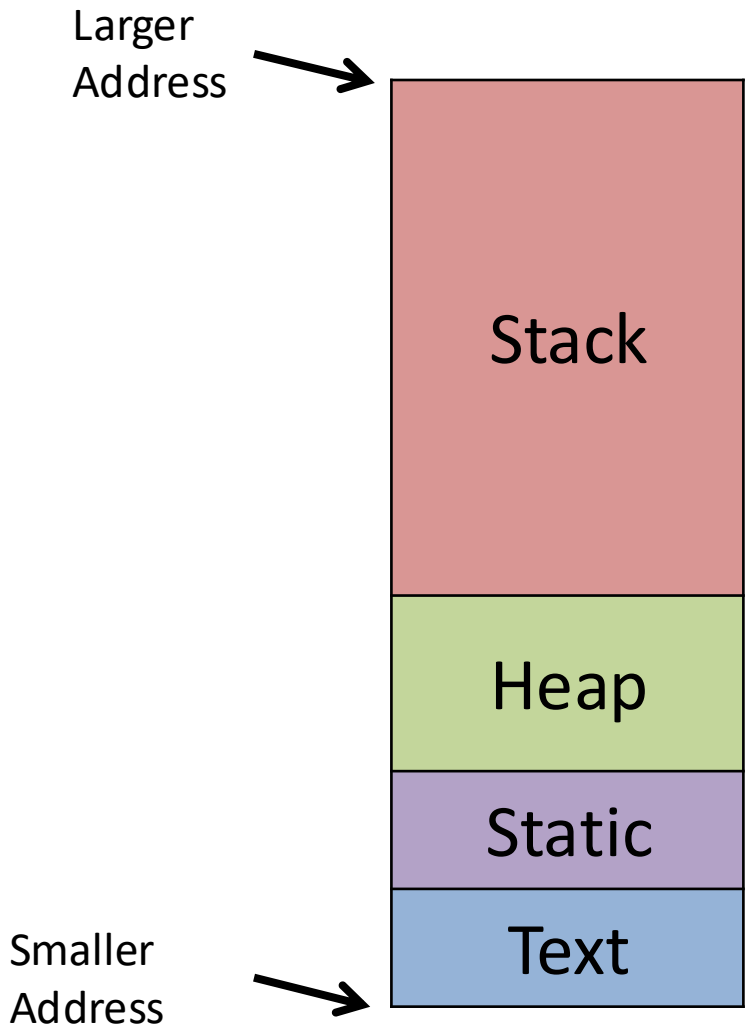


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```



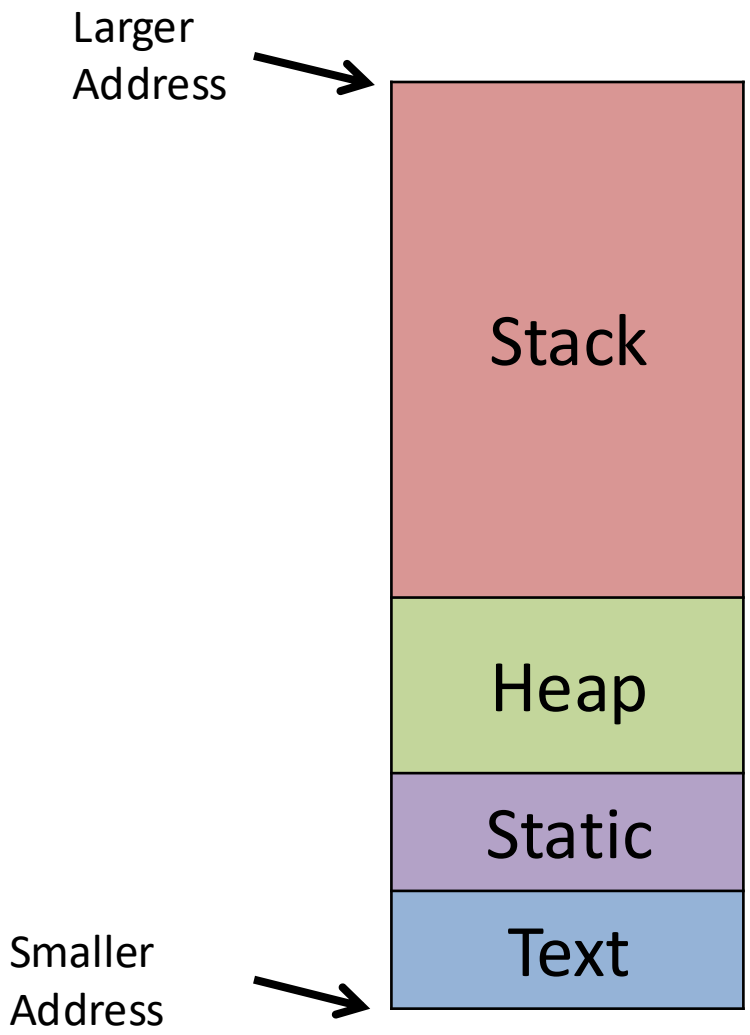
C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

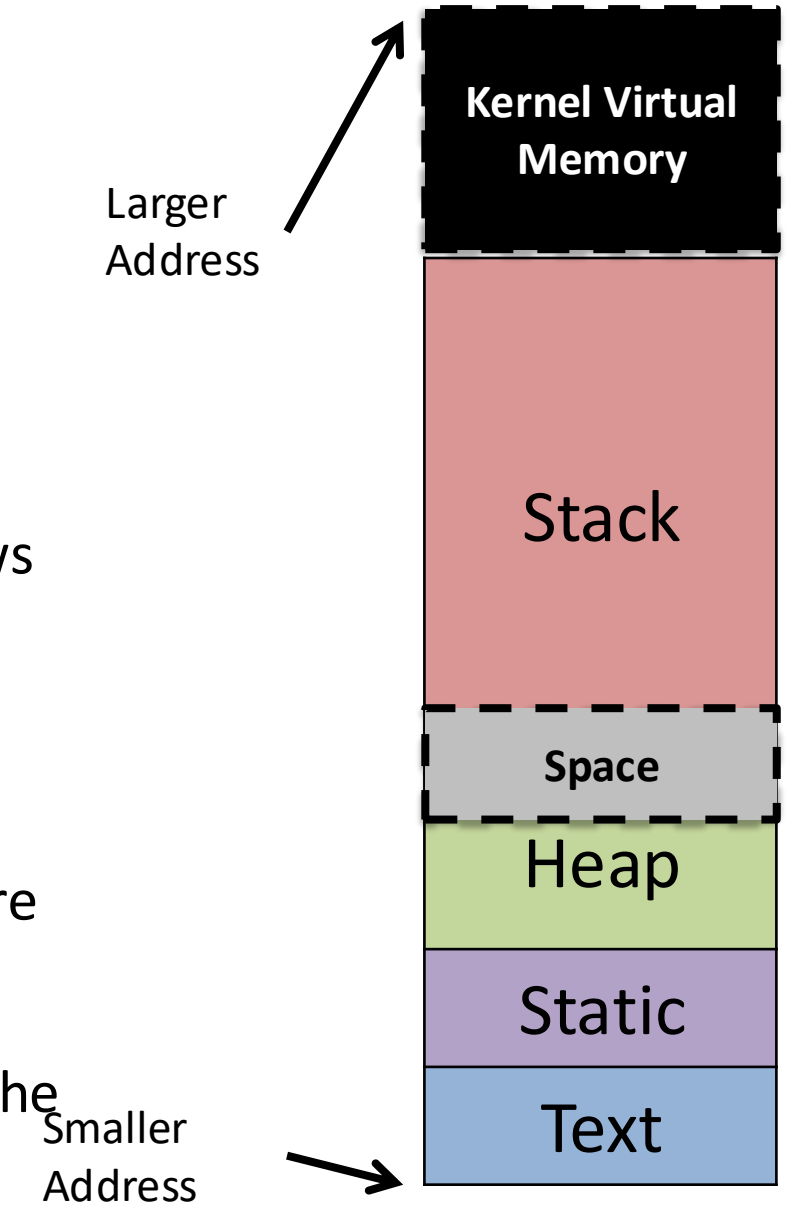
    printf("Hello CS213\n");
}
```

Assembly instructions generated by this function go in the Text section



Important notes!

- Illustration to the right gives relative locations for memory allocation of a C program – not exact, and not to scale
  - Stack and heap grow dynamically (down and up respectively)
  - There is actually a region of space in between that allows expansion / contraction (and holds other stuff too)
- ***Kernel virtual memory*** is found at the top region of the address space (invisible to user code)
  - ***Kernel*** – the portion of the OS that is always resident in memory. It is privileged code that runs with full hardware access, managing the CPU, memory, devices, and processes.
  - When an application program requires some action by the OS (e.g. read/write to file), a special ***system call*** instruction is performed that transfers control to kernel. Kernel performs action and returns to the program.



Interacting with data sections in assembly

- Stack
  - Stack pointer is saved in **`%rsp`** (remember, this was a special register saved for a special purpose!) and can be moved as needed
- Heap
  - C library (malloc) handles this above the machine level
  - i.e. from the machine point of view, there is no heap
- Static
  - Arbitrary pointers to memory can be created and used
    - With memory addressing instructions
    - Might references global values
- Text
  - Assembly code is placed here automatically
  - Labels are just addresses within the Text section

Question

Which sections are absolutely required, and which aren’t?

- Text
- Static
- Heap
- Stack

Question

Which sections are absolutely required, and which aren't?

- Text: necessary! Holds the code!
- Static: only necessary if using global variables or strings
- Heap: only necessary if you heap-allocate (using malloc or automatically while using other languages)
- Stack: necessary if you use variables or call functions
  - Probably always needed unless you write in assembly….

Pushing and Popping Stack Data

- A stack is a data structure where values can be added (***push***) or deleted (***pop***), but only through the "Last in, first out" discipline
  - Holds variables, function calls…
- Popped value will be a value that was most recently pushed and is still on the stack
- Can be implemented as an array where we always insert and remove elements from one end of the array (***top*** of the stack)
  - Stack top has lower address (stack grows downward)
  - Stack bottom has higher address
- The stack pointer `%rsp` hold the address of the top stack element

If we wanted to push `%rbp` to the stack:

```
subq $8, %rsp   (decrement stack pointer by 8 for quad word)
movq %rbp, (%rsp)
```

Instruction	Effect	Description
pushq *S*	R[%rsp] ← R[%rsp] − 8; M[R[%rsp]] ← *S*	Push quad word
popq *D*	*D* ← M[R[%rsp]]; R[%rsp] ← R[%rsp] + 8	Pop quad word

Figure 3.8 Push and pop instructions.

Push and pop instructions take a single operand

Illustration of Stack Operation

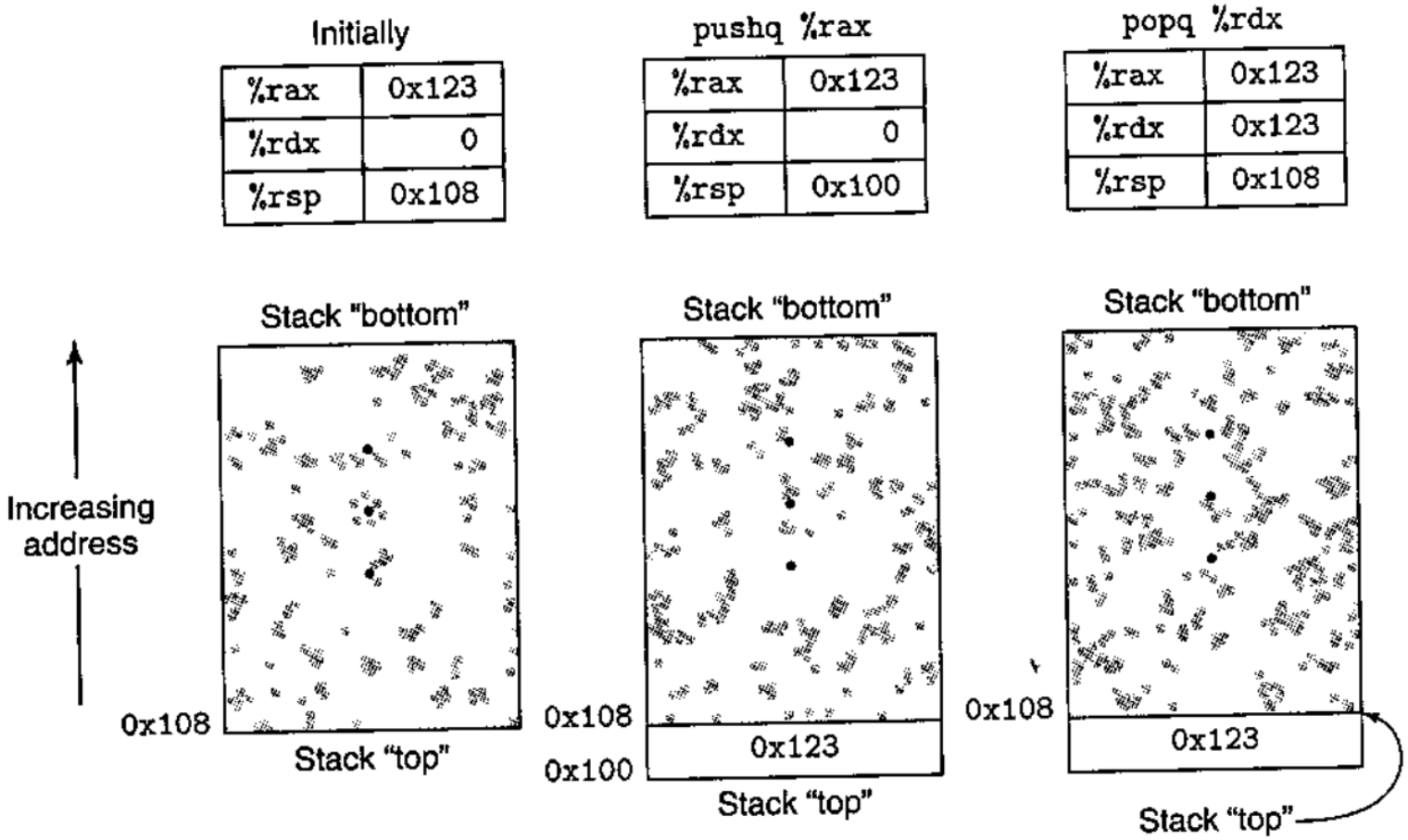


Figure 3.9 Illustration of stack operation. By convention, we draw stacks upside down, so that the "top" of the stack is shown at the bottom. With x86-64, stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register %rsp) and storing to memory, while popping involves reading from memory and incrementing the stack pointer.

Topics

- C Memory Layout
- **Pointers**
- One-dimensional Arrays

Basic Data Types

- Integers
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on instructions used, e.g.:
    - `imulq` – signed full multiply
    - `mulq` – unsigned full multiply

Intel	ASM	Bytes	C
byte	**`b`**	1	**`[unsigned] char`**
word	**`w`**	2	**`[unsigned] short`**
double word	**`l`**	4	**`[unsigned] int`**
quad word	**`q`**	8	**`[unsigned] long`**

Floating point data

- Won't be focusing on floating point
  - Has changed much more than integer types across updates
  - Not all x86-64 machines have the same capabilities here
- Registers %xmm0 - %xmm15
  - 128-bit registers
  - On newest machines refer to as %zmm0-%zmm31 (512-bit registers)
- Instructions
  - `addss` (add scalar single-precision)
  - `addsd` (add scalar double-precision)
  - `addpd` (add packed double-precision, two doubles at once)

More complex data types

- Pointers and Arrays

```
int* a  = &v;

int list[2] = {15, 27};
```

- Structs

```
typedef struct {
 int a;
 char b;
 int* c;
} mystruct_t;
```

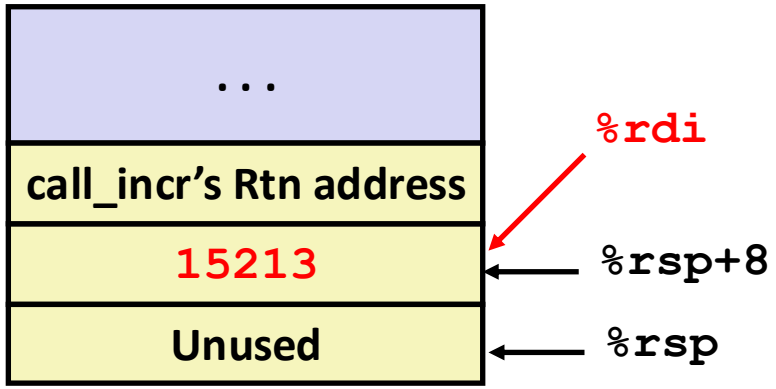
Example pointer code: calling `incr`

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movq    $3000, %rsi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

- Pointers are addresses
- `v1` must be stored on stack
  - Why? need to create pointer to it
- Compute pointer as `8(%rsp)`
  - Use `leaq` instruction

Memory (stack)



Register	Use(s)
%rdi	&v1
%rsi	3000

Pointers to global variables

```
int global_var = 15;

int* myfunc(void) {
  global_var += 2;
  return &global_var;
}
```

```
  .text
  .globl myfunc
  .type myfunc, @function
myfunc:
  addl $2, 0x2f1f(%rip)
  mov $0x404028, %eax
  ret
```

Two different mechanisms shown here: rip-relative and absolute

- Usually the compiler picks one to use

```
.globl global_var
  .data
  .align4
  .type global_var, @object
  .size global_var, 4
global_var:
  .long 15
```

Naming constants

These two are the same code.
One just uses a name for the constant.

```
   .text
   .globl myfunc
   .type myfunc, @function
myfunc:
   addl $2, 0x2f1f(%rip)
   mov $0x404028, %eax
   ret
   .globl global_var
   .data
   .align4
   .type global_var, @object
   .size global_var, 4
global_var:
   .long 15
```

```
   .text
   .globl myfunc
   .type myfunc, @function
myfunc:
   addl $2, global_var(%rip)
   mov $global_var, %eax
   ret
   .globl global_var
   .data
   .align4
   .type global_var, @object
   .size global_var, 4
global_var:
   .long 15
```

Topics

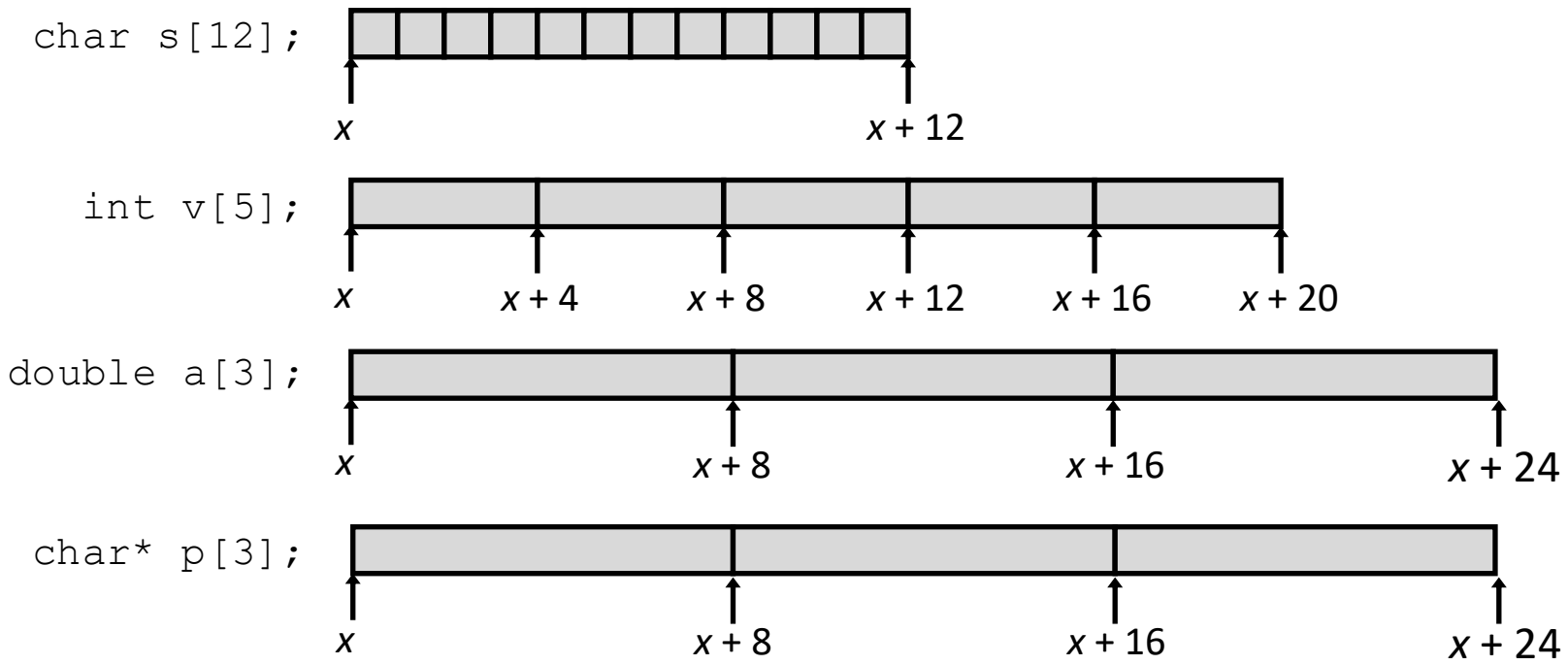
- C Memory Layout
- Pointers
- One-dimensional Arrays

One-Dimensional Array Allocation

- Basic Principle

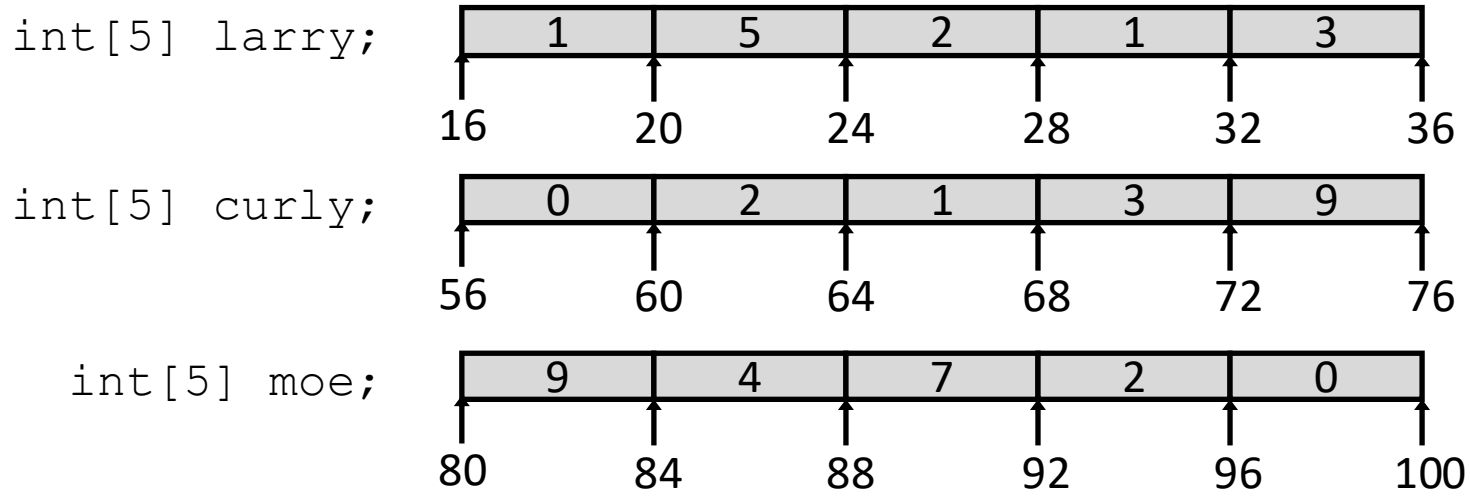
  T **`A[`***L***`];`** **`// e.g., int A[4];`**

  - Array of data type *T* and length *L*
  - Contiguously allocated region in memory of *L* * **`sizeof`**`(`*T*`)` bytes



Placing arrays at addresses

```
int[5] larry = { 1, 5, 2, 1, 3 };
int[5] curly = { 0, 2, 1, 3, 9 };
int[5] moe   = { 9, 4, 7, 2, 0 };
```



Numbers below arrays are addresses in decimal.

I picked some random starting address for each

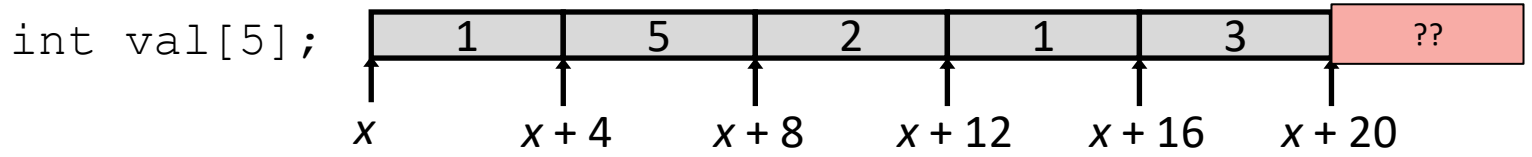
- Each array is allocated in contiguous 20 byte blocks
  - But no guarantee that `curly[]` will be right after `larry[]`!

Array Access and Pointer Arithmetic

- Basic Principle

  T **`A[`***L***`];`**

  - Identifier **`A`** can be used as a pointer to array element 0: **`A`** is of type *T**
  - ***Warning***: in C arrays count number of elements, but in assembly we count number of bytes!

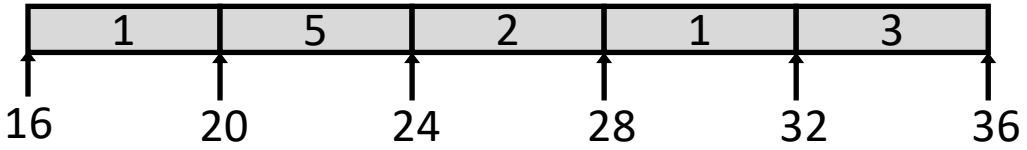


- Reference

Type	Value	
`val[4]`	**`int`**	3
`val`	**`int*`**	*x*
`val+1`	**`int*`**	*x* + 4
`&(val[2])`	**`int*`**	*x* + 8
`val[5]`	**`int`**	?? **No array bounds checking!!!**
`*(val+1)`	**`int`**	5
`val + i`	**`int*`**	*x* + 4 *i*

One-Dimensional Array Accessing Example

```
int[5] larry;
```




```
int get_digit(int[5] larry, size_t digit)
{
  return larry[digit];
}
```

```
get_digit:
  # %rdi = larry
  # %rsi = digit
  movl (%rdi,%rsi,4),%rax    # z[digit]
  retq
```

`%rdi` -> starting address of array

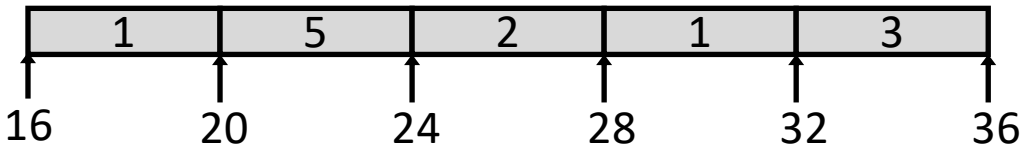
`%rsi` -> array index

- Desired digit at `%rdi + 4*%rsi`
- Use memory addressing! `(%rdi,%rsi,4)`

- This is why memory accesses have a scale! `D(Rb, Ri, s)`
  - Scale 1, 2, 4, or 8 -> type sizes

What happens if you access past the end of an array?

```
int[5] larry;
```




```
int get_digit(int[5] larry, size_t digit)
{
  return larry[digit];
}
```

```
get_digit:
  # %rdi = larry
  # %rsi = digit
  movl (%rdi,%rsi,4),%rax    # z[digit]
  retq
```

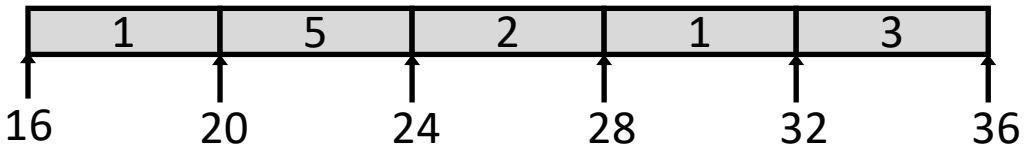
- Consider at the assembly generated for `get_digit`

1. There is no bounds checking at all

2. Accesses past the end of the array instead read from nearby memory!

Rust version of the same code

```
int[5] larry;
```




```
int get_digit(int[5] larry, size_t digit)
{
  return larry[digit];
}
```

```
get_digit:
  # %rdi = larry
  # %rsi = digit
  cmpq $4, %rsi
  ja .PANIC_BOUNDS_CHECK
  movl (%rdi,%rsi,4),%rax    # z[digit]
  retq
```

- Rust does bounds checking of array accesses
  - PANIC_BOUNDS_CHECK
- Downside: performance is slower

One-Dimensional Array Loop Example

```
void zincr(int *z) {
  size_t i;
  for (i = 0; i < 4; i++)
    z[i]++;
}
```



```
zincr:
  # %rdi = z
  movl    $0, %eax            #   i = 0
  jmp     .L3                 #   goto middle
.L4:                          # loop:
  addl    $1, (%rdi,%rax,4)   #   z[i]++
  addq    $1, %rax            #   i++
.L3:                          # middle:
  cmpq    $4, %rax            #   i:4
  jbe     .L4                 #   if i<=4, goto loop
  retq
```

Example

```
z -> %rdi
i -> %rax
addl $1, (%rdi,%rax,4) #Source: z[i]++ (int z[])
```

- What changes if z is instead an array of:
  - short
  - char
  - bool
  - char*
  - unsigned int

Example

```
z -> %rdi
i -> %rax
addl $1, (%rdi,%rax,4) #Source: z[i]++ (int z[])
```

- What changes if z is instead an array of:
  - short `addw $1, (%rdi,%rax,2)`
  - char
  - bool
  - char*
  - unsigned int

Example

```
z -> %rdi
i -> %rax
addl $1, (%rdi,%rax,4) #Source: z[i]++ (int z[])
```

- What changes if z is instead an array of:
  - short `addw $1, (%rdi,%rax,2)`
  - char `addb $1, (%rdi,%rax,1)`
  - bool
  - char*
  - unsigned int

Example

```
z -> %rdi
i -> %rax
addl $1, (%rdi,%rax,4) #Source: z[i]++ (int z[])
```

- What changes if z is instead an array of:
  - short `addw $1, (%rdi,%rax,2)`
  - char `addb $1, (%rdi,%rax,1)`
  - bool `addb $1, (%rdi,%rax,1)`
  - char*
  - unsigned int

Example

```
z -> %rdi
i -> %rax
addl $1, (%rdi,%rax,4) #Source: z[i]++ (int z[])
```

- What changes if z is instead an array of:
  - short `addw $1, (%rdi,%rax,2)`
  - char `addb $1, (%rdi,%rax,1)`
  - bool `addb $1, (%rdi,%rax,1)`
  - char* `addq $1, (%rdi,%rax,8)`
  - unsigned int

Example

```
 z -> %rdi
 i -> %rax
 addl $1, (%rdi,%rax,4) #Source: z[i]++ (int
z[])
```

- What changes if z is instead an array of:
  - short `addw $1, (%rdi,%rax,2)`
  - char `addb $1, (%rdi,%rax,1)`
  - bool `addb $1, (%rdi,%rax,1)`
  - char* `addq $1, (%rdi,%rax,8)`
  - unsigned int Nothing. Still 4 bytes. add works the same on sign/unsigned

Discuss Exam T/F (see video)