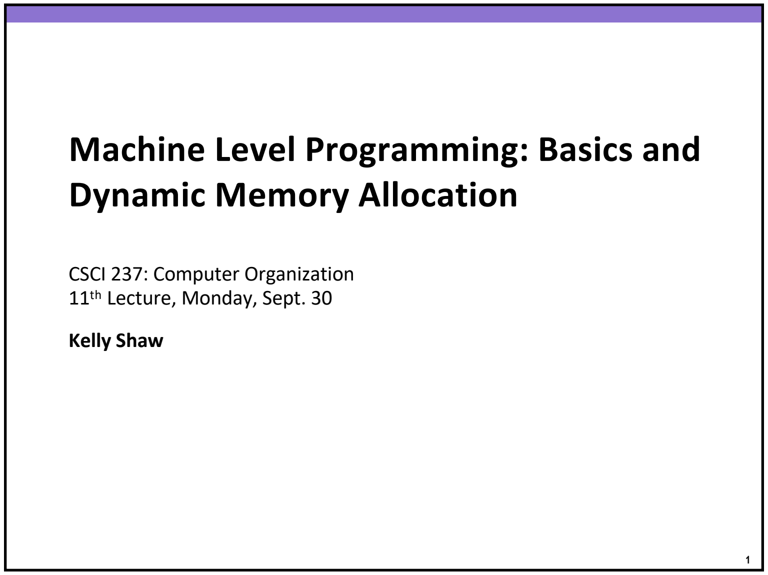
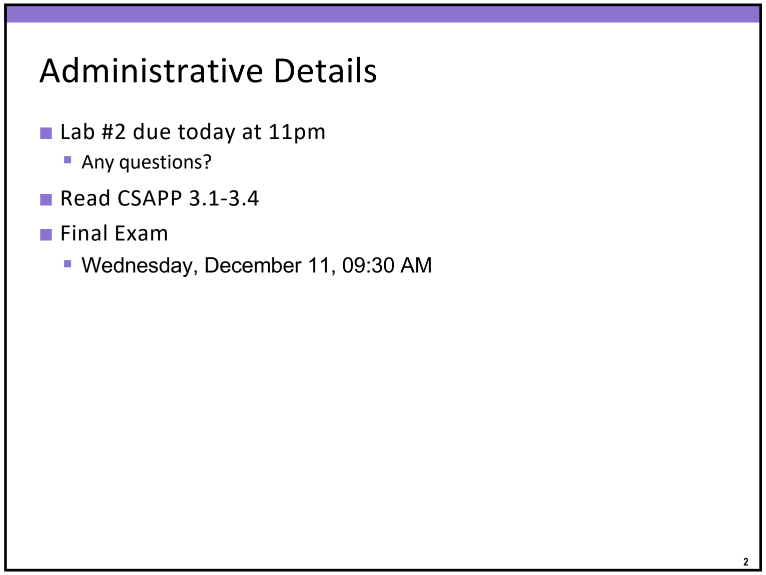


Machine Level Programming: Basics and Dynamic Memory Allocation

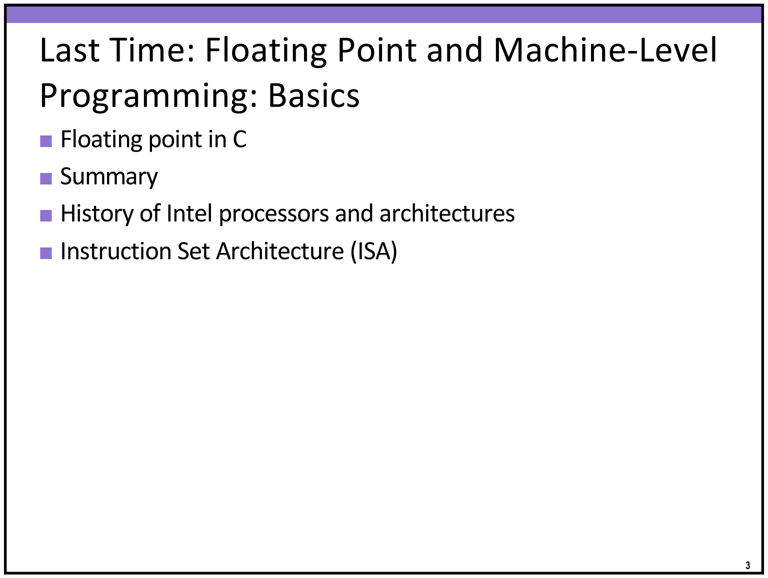
CSCI 237: Computer Organization
11th Lecture, Monday, Sept. 30

Kelly Shaw



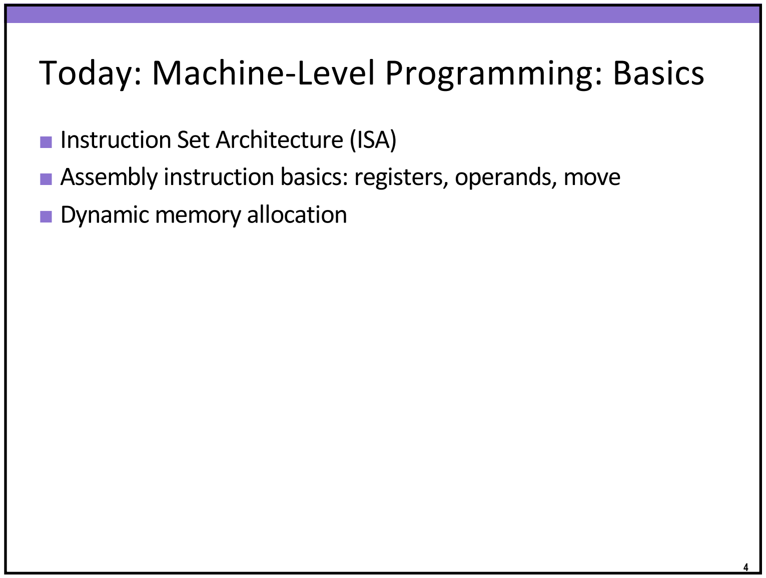
Administrative Details

- Lab #2 due today at 11pm
  - Any questions?
- Read CSAPP 3.1-3.4
- Final Exam
  - Wednesday, December 11, 09:30 AM



Last Time: Floating Point and Machine-Level Programming: Basics

- Floating point in C
- Summary
- History of Intel processors and architectures
- Instruction Set Architecture (ISA)



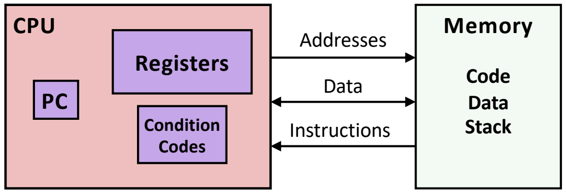
Today: Machine-Level Programming: Basics

- Instruction Set Architecture (ISA)
- Assembly instruction basics: registers, operands, move
- Dynamic memory allocation

Definitions

- Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand for writing assembly/machine code.
  - Examples: instruction set specification, registers
- Microarchitecture: Implementation of the architecture
  - Examples: cache sizes and core frequency
- Code Forms:
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code
- Example ISAs:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones

Assembly/Machine Code View



Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

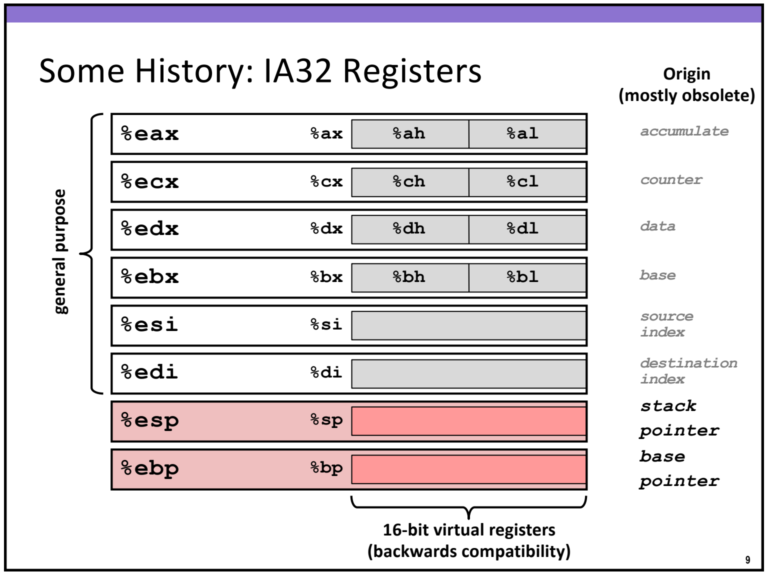
Assembly Characteristics: Data Types

- “Integer” data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)



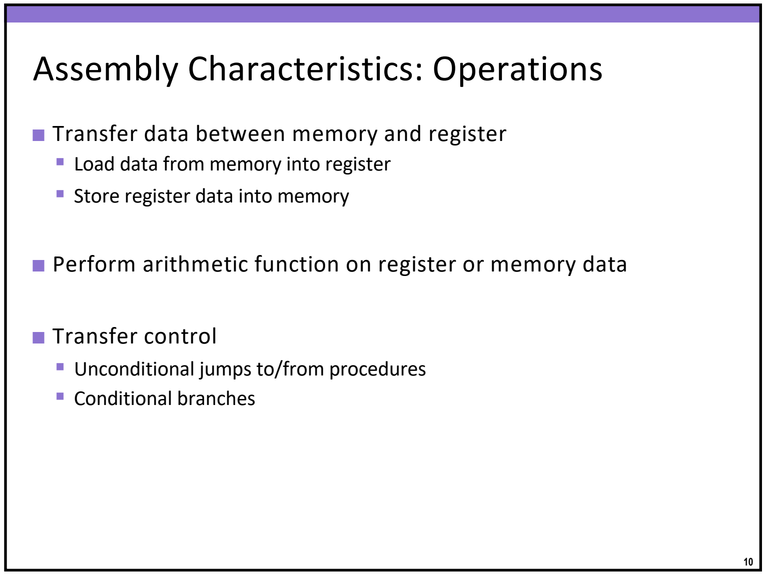
Some History: IA32 Registers

general purpose					Origin (mostly obsolete)
	%eax	%ax	%ah	%al	accumulate
	%ecx	%cx	%ch	%cl	counter
	%edx	%dx	%dh	%dl	data
	%ebx	%bx	%bh	%bl	base
	%esi	%si			source index
	%edi	%di			destination index
	%esp	%sp			stack pointer
	%ebp	%bp			base pointer

16-bit virtual registers (backwards compatibility)

9

9

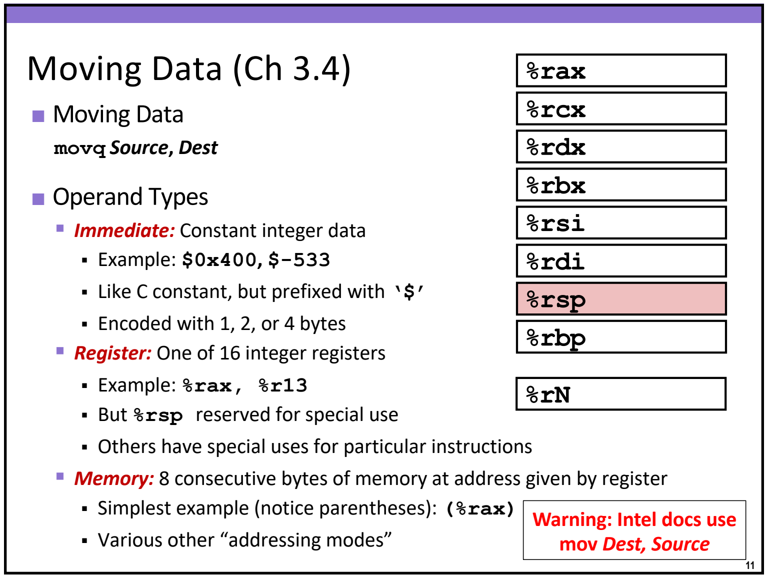


Assembly Characteristics: Operations

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Perform arithmetic function on register or memory data
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

10

10



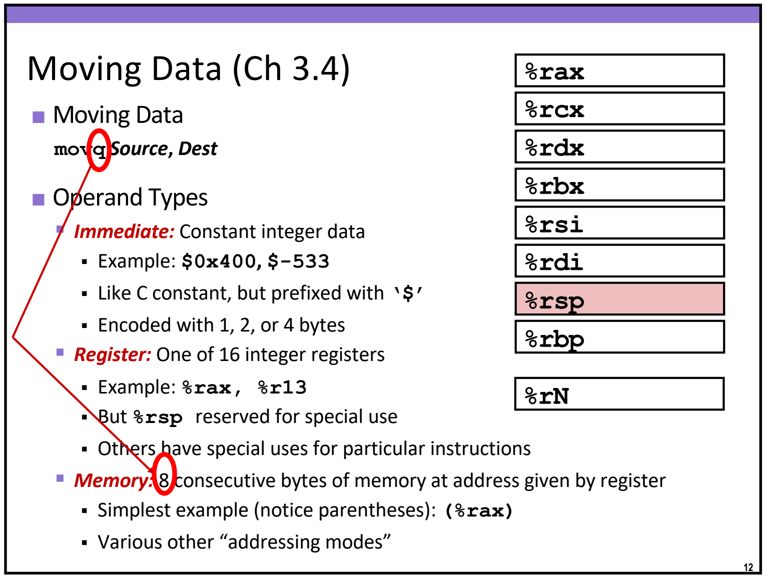
Moving Data (Ch 3.4)

- Moving Data
  `movq` ***Source*, *Dest***
- Operand Types
  - ***Immediate:*** Constant integer data
    - Example: `$0x400, $-533`
    - Like C constant, but prefixed with `'$'`
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 16 integer registers
    - Example: `%rax, %r13`
    - But `%rsp` reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example (notice parentheses): `(%rax)`
    - Various other "addressing modes"

Warning: Intel docs use mov *Dest, Source*

11

11

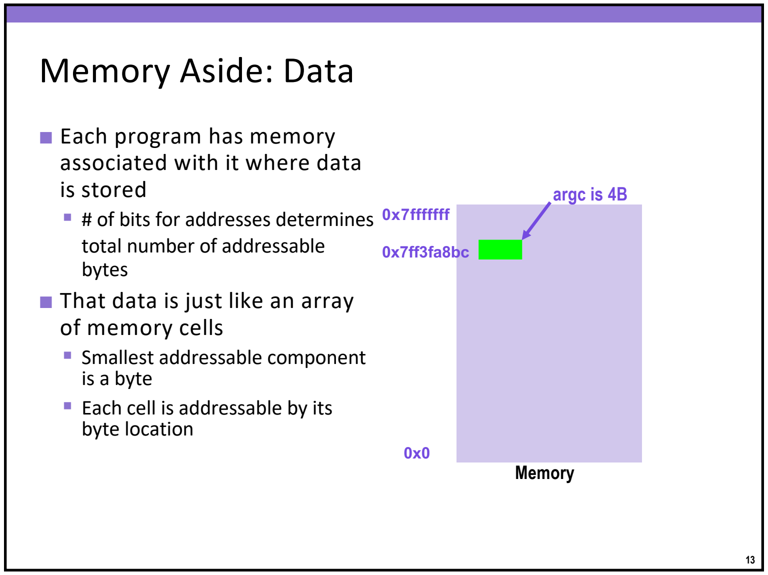


Moving Data (Ch 3.4)

- Moving Data
  `movq` ***Source*, *Dest***
- Operand Types
  - ***Immediate:*** Constant integer data
    - Example: `$0x400, $-533`
    - Like C constant, but prefixed with `'$'`
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 16 integer registers
    - Example: `%rax, %r13`
    - But `%rsp` reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example (notice parentheses): `(%rax)`
    - Various other "addressing modes"

12

12

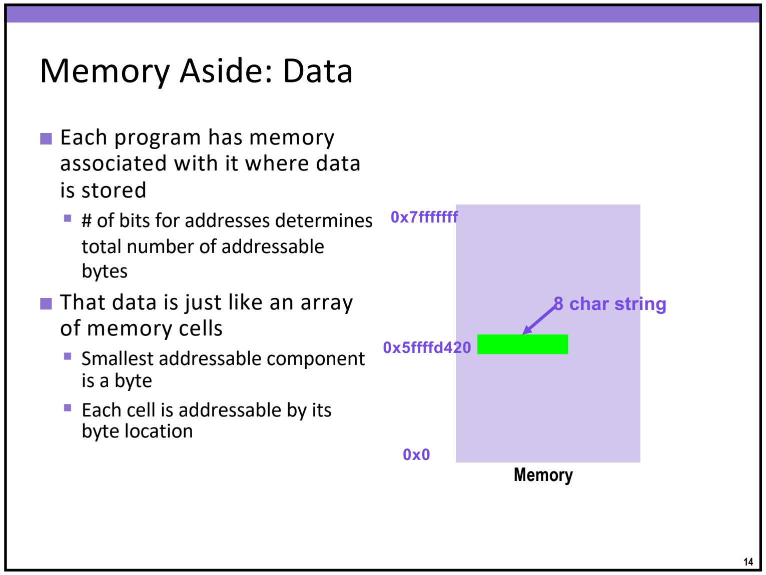


Memory Aside: Data

- Each program has memory associated with it where data is stored
  - # of bits for addresses determines total number of addressable bytes
- That data is just like an array of memory cells
  - Smallest addressable component is a byte
  - Each cell is addressable by its byte location

argc is 4B
0x7fffffff
0x7ff3fa8bc
0x0
Memory

13

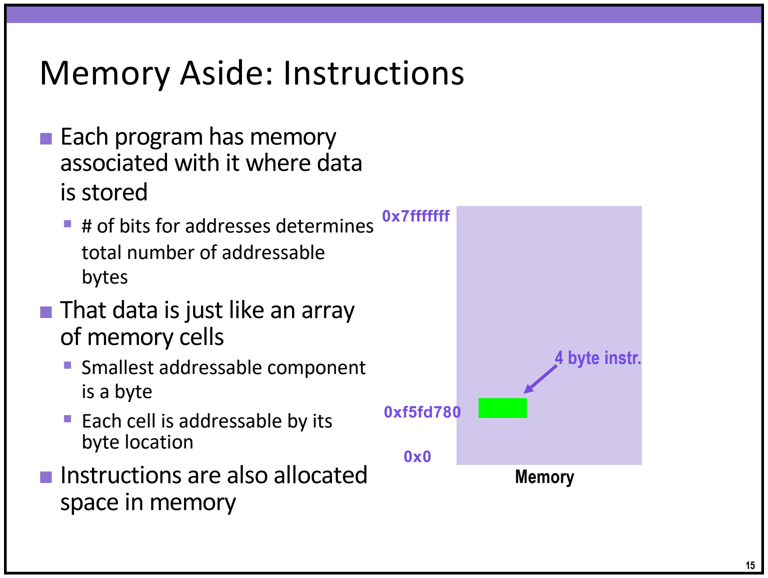


Memory Aside: Data

- Each program has memory associated with it where data is stored
  - # of bits for addresses determines total number of addressable bytes
- That data is just like an array of memory cells
  - Smallest addressable component is a byte
  - Each cell is addressable by its byte location

0x7fffffff
8 char string
0x5ffffd420
0x0
Memory

14

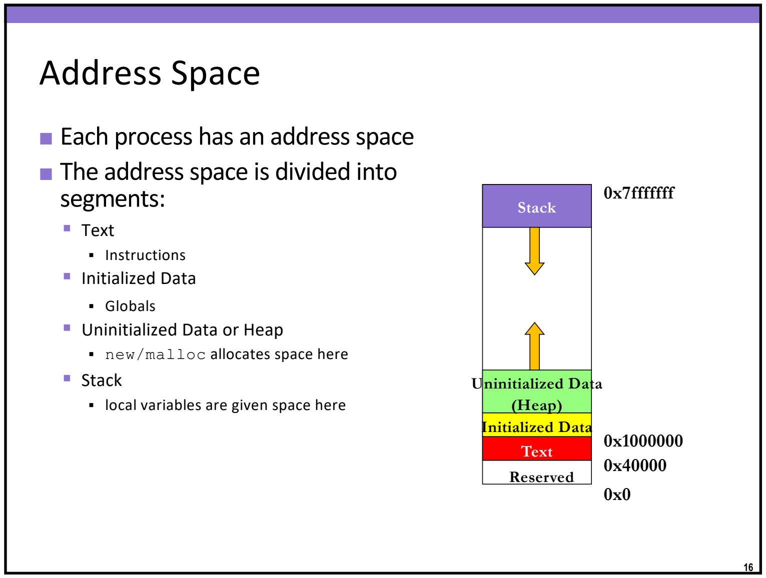


Memory Aside: Instructions

- Each program has memory associated with it where data is stored
  - # of bits for addresses determines total number of addressable bytes
- That data is just like an array of memory cells
  - Smallest addressable component is a byte
  - Each cell is addressable by its byte location
- Instructions are also allocated space in memory

0x7fffffff
4 byte instr.
0xf5fd780
0x0
Memory

15

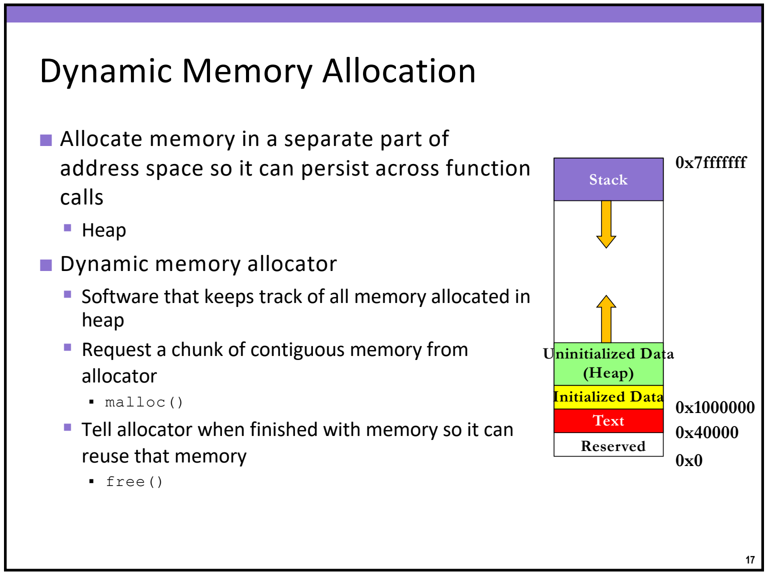


Address Space

- Each process has an address space
- The address space is divided into segments:
  - Text
    - Instructions
  - Initialized Data
    - Globals
  - Uninitialized Data or Heap
    - `new/malloc` allocates space here
  - Stack
    - local variables are given space here

Stack
0x7fffffff
Uninitialized Data
(Heap)
Initialized Data
Text
Reserved
0x1000000
0x40000
0x0

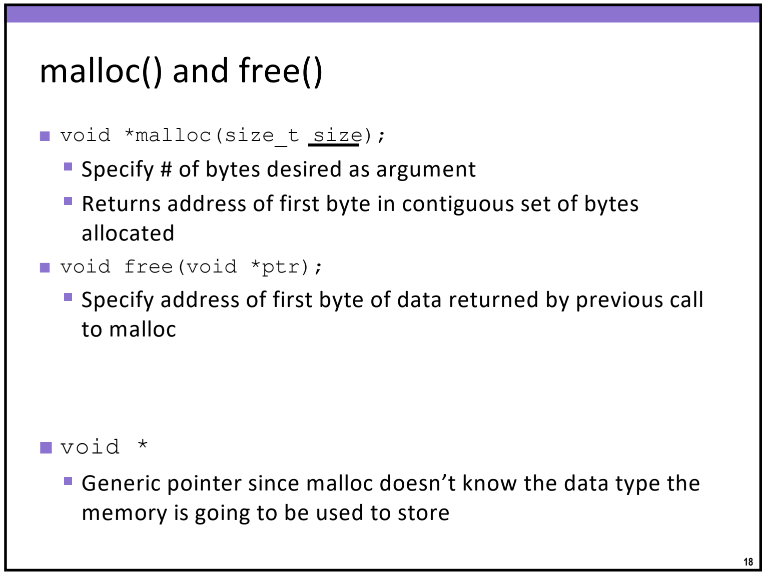
16



Dynamic Memory Allocation

- Allocate memory in a separate part of address space so it can persist across function calls
  - Heap
- Dynamic memory allocator
  - Software that keeps track of all memory allocated in heap
  - Request a chunk of contiguous memory from allocator
    - `malloc()`
  - Tell allocator when finished with memory so it can reuse that memory
    - `free()`

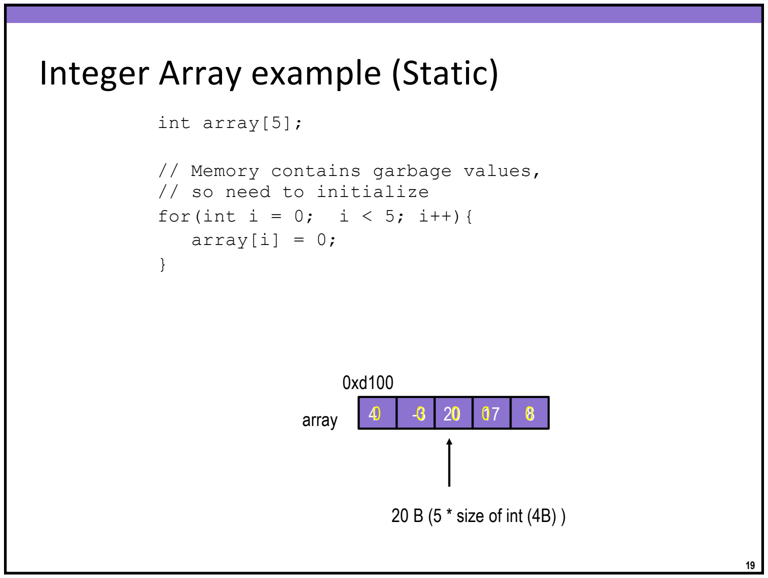
17



malloc() and free()

- `void *malloc(size_t size);`
  - Specify # of bytes desired as argument
  - Returns address of first byte in contiguous set of bytes allocated
- `void free(void *ptr);`
  - Specify address of first byte of data returned by previous call to malloc
- `void *`
  - Generic pointer since malloc doesn't know the data type the memory is going to be used to store

18

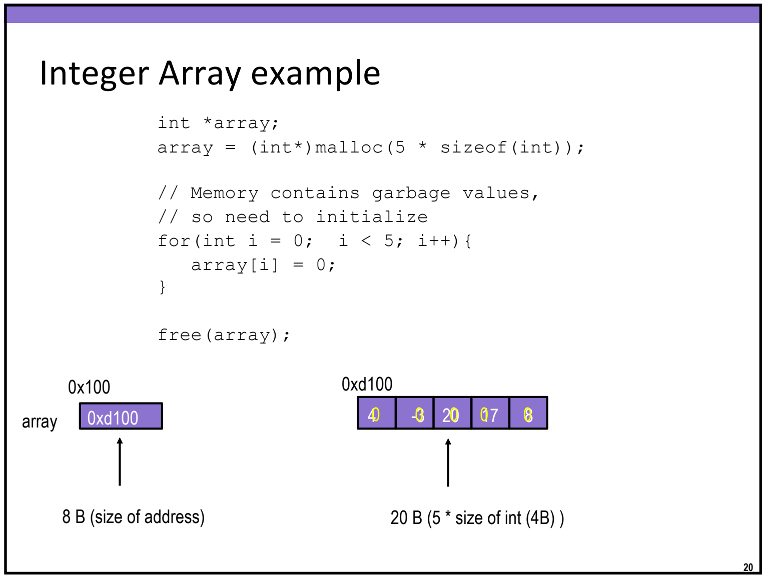


Integer Array example (Static)

```
int array[5];

// Memory contains garbage values,
// so need to initialize
for(int i = 0;  i < 5; i++){
   array[i] = 0;
}
```

19



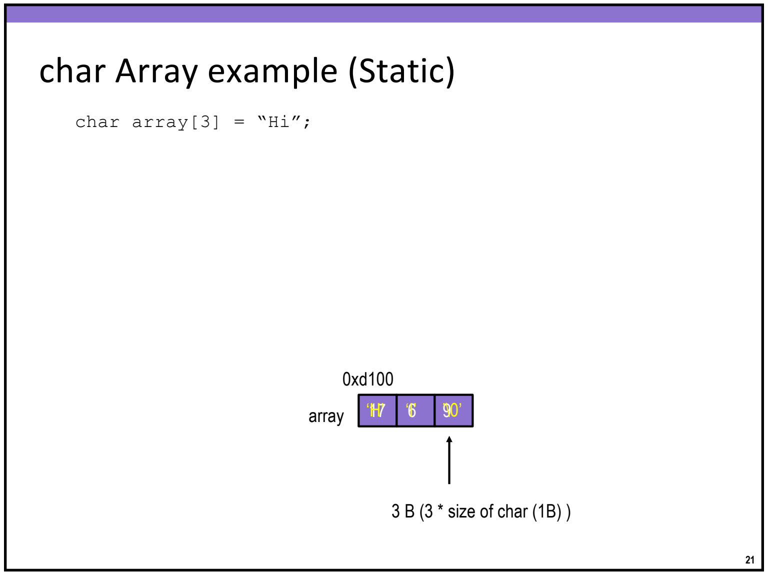
Integer Array example

```
int *array;
array = (int*)malloc(5 * sizeof(int));

// Memory contains garbage values,
// so need to initialize
for(int i = 0;  i < 5; i++){
   array[i] = 0;
}

free(array);
```

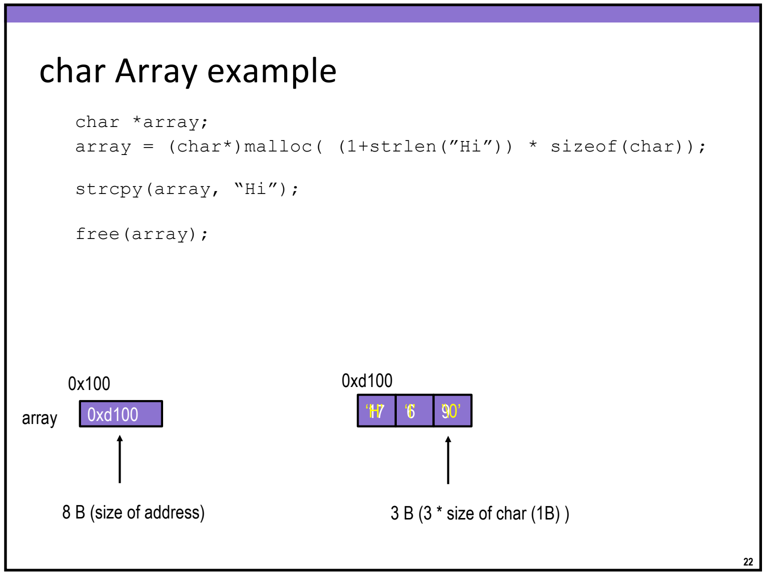
20



char Array example (Static)

```
char array[3] = “Hi”;
```

21



char Array example

```
char *array;
array = (char*)malloc( (1+strlen(”Hi”)) * sizeof(char));

strcpy(array, “Hi”);

free(array);
```

22