Oregon State University

COLLEGE OF ENGINEERING | School of Electrical Engineering and Computer Science

CS 161

Introduction to CS I

Lecture 15

- How does memory work in a C++ program?

About Me

- 6th year at OSU, got my Bachelor Degree in Spring, 2018
- Involved in CS 16X since Fall 2017
- Taught CS 161 last term

Week 6 Tips

- Lab 6 – posted
  - Revisit pass by reference
  - Practice on pass by
    - Implementing Hangman
  - Memory model
- Study session this week
  - Thursday 6-7pm at LINC 268
  - Worksheet 6 is posted on the website

Assignment 4: Text Surgeon

- Read in a line of text from the user, and perform analysis and manipulation of that string
- Provides practice with
  - String functions
  - 1-dimensional arrays
  - C-style strings
  - Dynamic memory allocation
- Design Document is due Feb. 16 – go for it!

Review: References and Pointers

- Declare variables:
  - Reference: `int& z = n;      /* z is an alias to n */`
  - Pointer: `int* p = &n;     /* p is the address of n */`
- Operators (perform actions):
  - &: address-of
    - `p = &n;`
    - `&n = 5234; /* not allowed! (what would it mean?) */`
  - *: dereference (value-of): access the value at memory address
    - `int g = *p;      /* read */`
    - `*p = 27;         /* write/change */`

References versus Pointers

- Do not confuse "reference" (a data type) with "pass by reference" (something that happens when you call a function)
- Reference: an alias to some variable (permanent)
  - `int& r = s;`
  - Can assign new values to `r` (which is `s`), but cannot make `r` be an alias to another variable later
  - Must be initialized when declared
- Pointer: stores the address of some variable
  - `int* p = &s;`
  - Can change what address `r` contains (where it points to) anytime
  - Can be declared, then initialized later

Pointer activity

```
1. int r = 17;
2. int s = -10;
3. int* q = NULL;
```

```
&r = Addr1
&s = Addr2
```

Line	r	s	q	*q
3				
4				
5				
6				
7				
8				

Pointer activity

```
1. int r = 17;
2. int s = -10;
3. int* q = NULL;
4. q = &r;
```

```
&r = Addr1
&s = Addr2
```

Line	r	s	q	*q
3	17	-10	0	X
4				
5				
6				
7				
8				

Pointer activity

```
1. int r = 17;
2. int s = -10;
3. int* q = NULL;
4. q = &r;
5. r = -5;
```

```
&r = Addr1
&s = Addr2
```

Line	r	s	q	*q
3	17	-10	**0**	**X**
4	17	-10	**Addr1**	17
5				
6				
7				
8				

Pointer activity

```
1. int r = 17;
2. int s = -10;
3. int* q = NULL;
4. q = &r;
5. r = -5;
6. *q = 42;
```

```
&r = Addr1
&s = Addr2
```

Line	r	s	q	*q
3	17	-10	**0**	**X**
4	17	-10	**Addr1**	17
5	**-5**	-10	Addr1	**-5**
6				
7				
8				

Pointer activity

```
1. int r = 17;
2. int s = -10;
3. int* q = NULL;
4. q = &r;
5. r = -5;
6. *q = 42;
7. q = &s;
```

```
&r = Addr1
&s = Addr2
```

Line	r	s	q	*q
3	17	-10	**0**	**X**
4	17	-10	**Addr1**	17
5	**-5**	-10	Addr1	**-5**
6	**42**	-10	Addr1	**42**
7				
8				

Pointer activity

```
1. int r = 17;
2. int s = -10;
3. int* q = NULL;
4. q = &r;
5. r = -5;
6. *q = 42;
7. q = &s;
8. s++;
```

```
&r = Addr1
&s = Addr2
```

Line	r	s	q	*q
3	17	-10	**0**	**X**
4	17	-10	**Addr1**	17
5	**-5**	-10	Addr1	**-5**
6	**42**	-10	Addr1	**42**
7	42	-10	**Addr2**	-10
8				

Pointer activity

```
1. int r = 17;
2. int s = -10;
3. int* q = NULL;
4. q = &r;
5. r = -5;
6. *q = 42;
7. q = &s;
8. s++;
```

```
&r = Addr1
&s = Addr2
```

Line	r	s	q	*q
3	17	-10	**0**	**X**
4	17	-10	**Addr1**	17
5	**-5**	-10	Addr1	**-5**
6	**42**	-10	Addr1	**42**
7	42	-10	**Addr2**	-10
8	42	**-9**	Addr2	**-9**

Passing pointers into functions

```
• int v = 3;                 int* p = &v;
• void fn1(int w);           void pfn1(int* q);
• void fn2(int& w);          void pfn2(int*& q);
```

- Pass by value: make a copy: `fn1(v);`
  - <u>Same for pointers</u>: make a <u>copy</u> of the <u>address</u> <u>inside</u> the pointer variable; changes to `q` do not change `p` `pfn1(p);`
- Pass by reference: pass the address of the variable : `fn2(v);`
  - <u>Same for pointers</u>: pass the <u>address</u> <u>of</u> the pointer variable; changes to `q` DO change `p` `pfn2(p);`

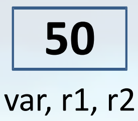
Challenge questions

- What if you made a pointer (p2) that points to a pointer (p1) that points to an int (x)?
  - What would the picture look like?
  - Write the code for this picture.

- Can you make this same picture for references?
  - What if you had two references, r1 and r2?

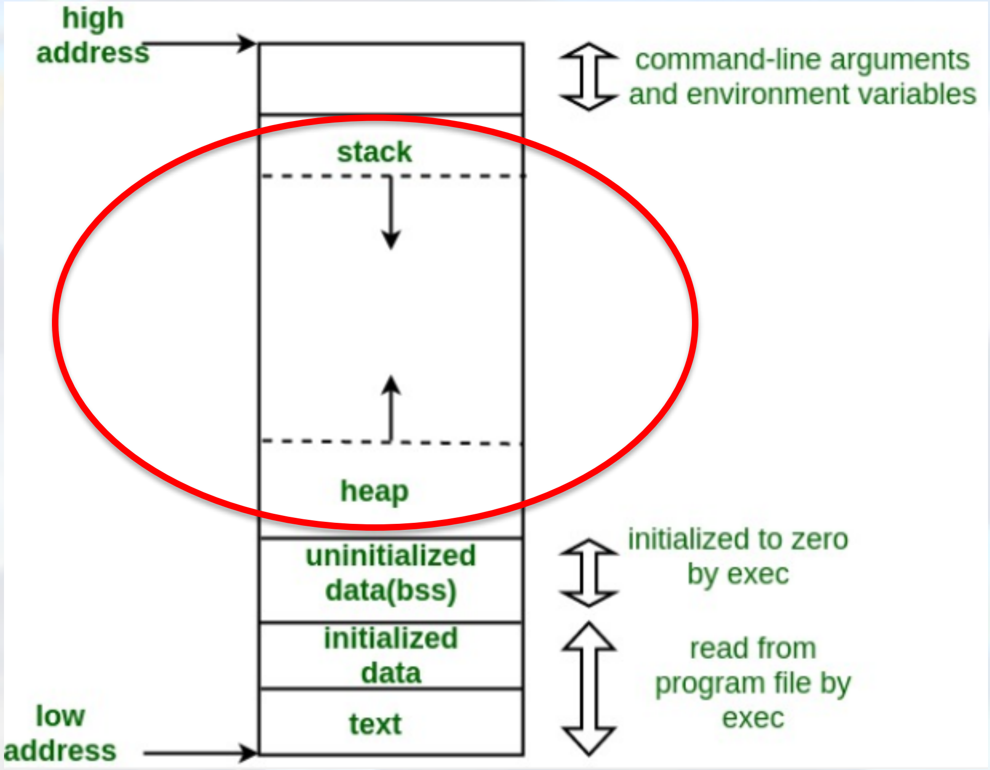
```
int var = 50;
int &r1 = var;
int &r2 = var;
```

You cannot say: int &&r2 = var;



Oregon State University
College of Engineering

Memory Model/Layout


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

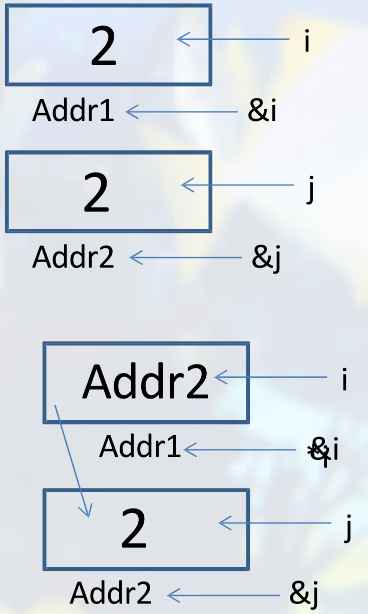
What we have seen so far: Variables vs. Pointers

- Value
  - Values stored directly
  - Copy of value is passed

```
int i, j=2;
i=j;
```

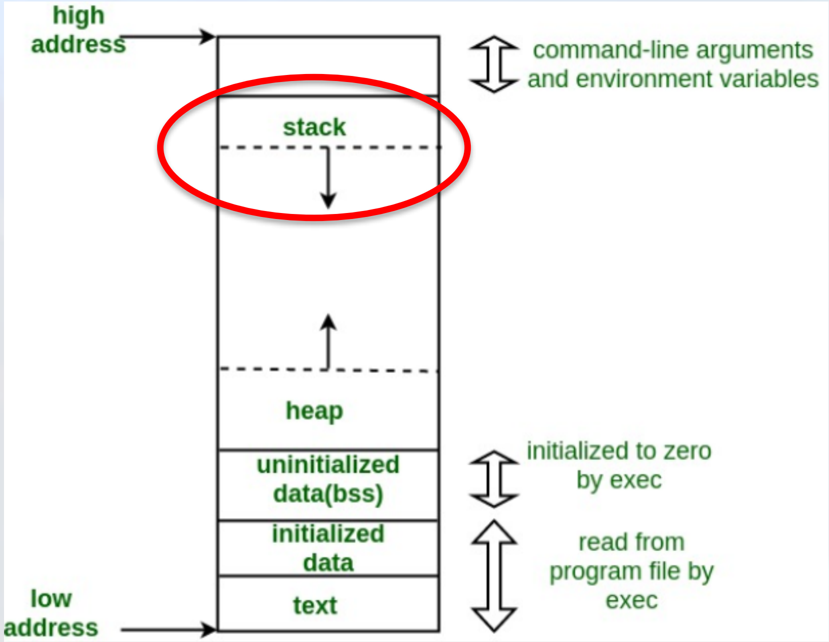
- Pointer
  - Address to variable is stored
  - Copy of address is passed

```
int *i = NULL, j=2;
i=&j;
```



Stack – Static Memory

- Stack
  - Variables known in advance (global/local variables, constants), always allocated **at compile time**
  - Functions have their own stack frame
  - When a function ends, the stack frame collapses and cleans up the memory for you


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

What if we don’t have the j?

- We need to create the address space
- How do we do this?

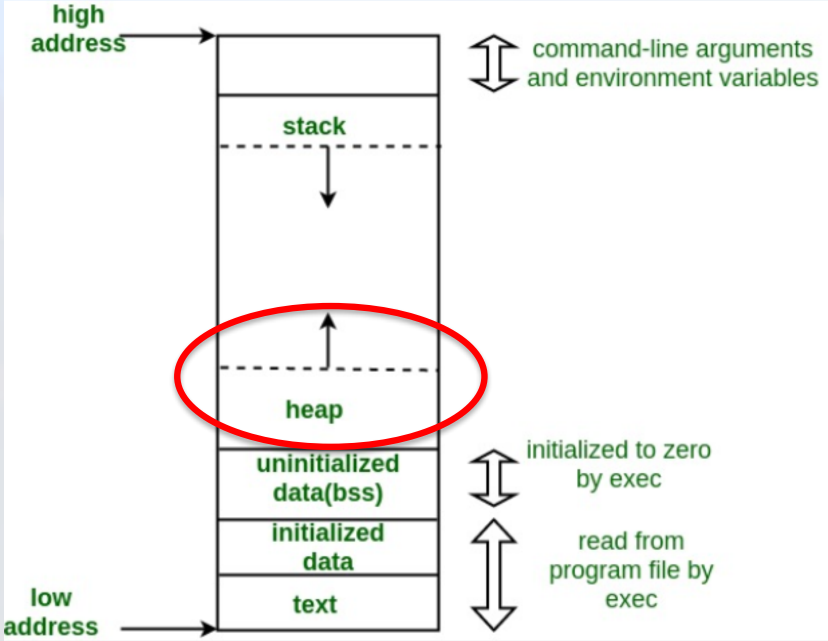
  - **new** type;
- For example:

```
int *i = NULL;
i = new int; //new returns an address
*i = 10;
```

Oregon State University
College of Engineering

Heap – Dynamic Memory

- Heap
  - Variables defined **at runtime** (use **new** keyword), do not need to be known in advance
  - Variables declared dynamically in a function do not disappear when the function ends as they are on the heap and not the function stack
  - Need to free dynamic memory when done with it, otherwise memory leaks


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized data
text
low address
initialized to zero by exec
read from program file by exec

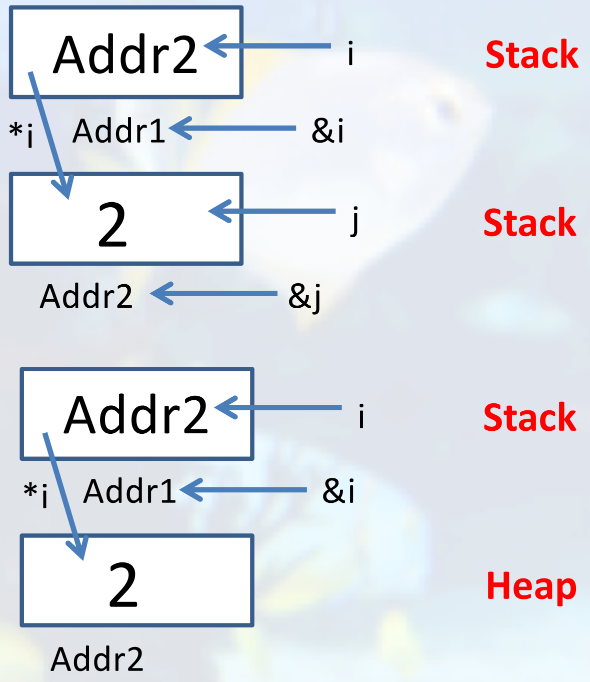
Static vs. Dynamic

- Static
  - Assign address of variable

```
int *i=NULL, j=2;
i=&j;
```

- Dynamic
  - Create memory
  - Assign memory to pointer

```
int *i=new int;
*i=2;
```



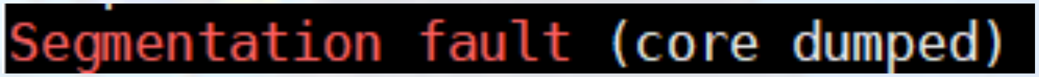
How to avoid Memory Leaks?
A: Deleting items from the heap

- Delete operator: `delete`
- (delete does not clear the memory contents, just lets it be reused)

For example:

```
int main () {
   int *i = NULL;
   i = new int;
   *i = 2;
   delete i;
   i = NULL; // set the pointer back to NULL
   return 0;
 }
```

Segmentation Fault (aka segfault)



```
Segmentation fault (core dumped)
```

- Something that causes programs to crash
- Often caused by program trying to read or write an illegal memory location

For example, what’s wrong with this:

```
int main () {
   int *i = NULL;
   i = new int; //if forget this, segfault
   *i = 2;
   delete i;
   i = NULL; // set the pointer to NULL
   return 0;
 }
```

Memory allocation tips

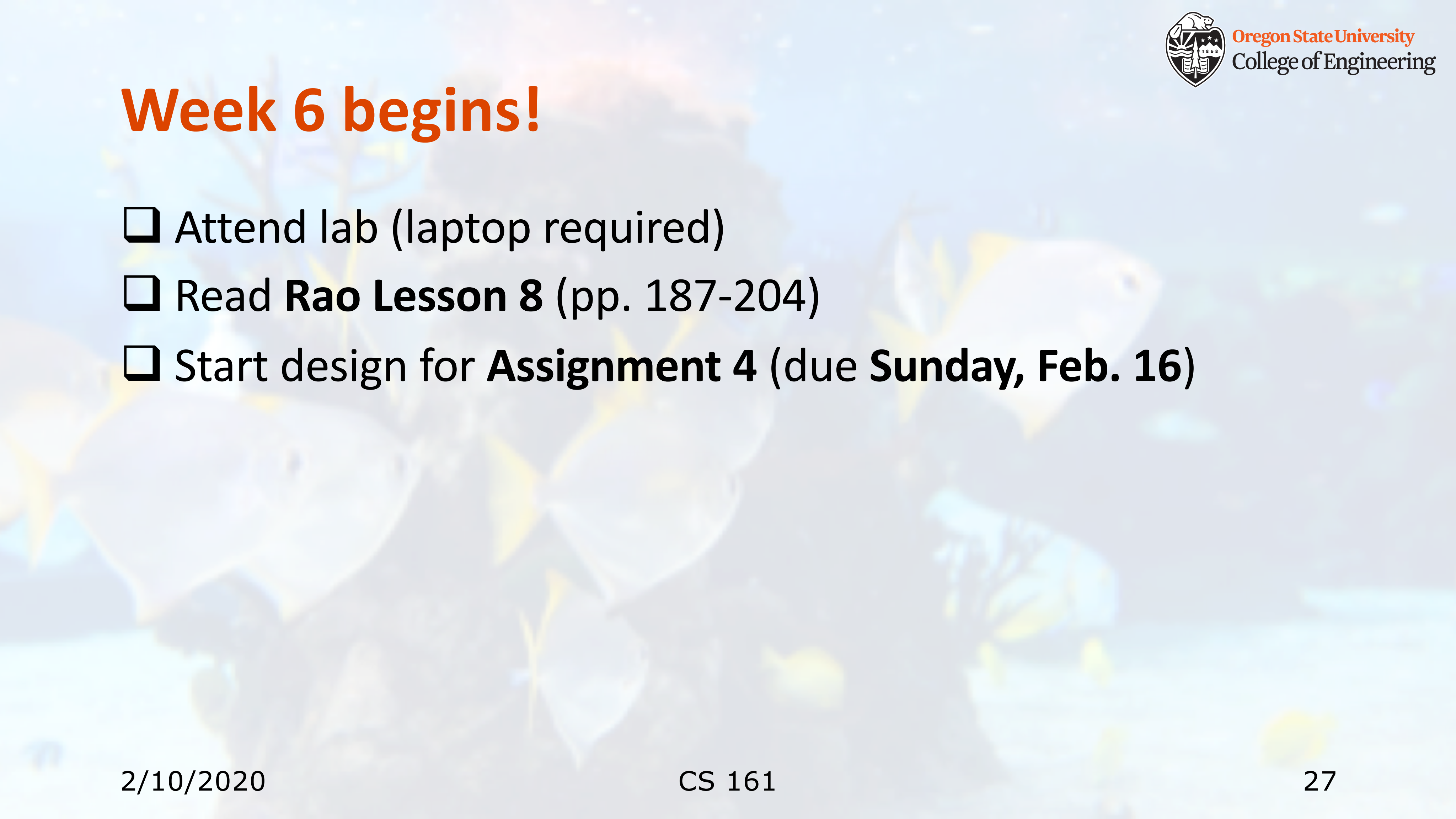
- new can fail – throws exception
- after delete, set your ptr to NULL (explicitly)
- you can delete a NULL ptr with no adverse effects
- Gotchas:
  - forget to delete: memory leak
  - forget to set to NULL: dangling pointers
- tool: valgrind

What vocabulary did we learn today?

- Static memory
- Dynamic memory
- Stack
- Heap
- Segmentation fault
- Dynamic memory operators: new and delete
- Memory leak
- Dangling pointer

What ideas and skills did we learn today?

- Memory model: where the stack and the heap are
- How to dynamically allocate memory
- How to delete dynamic memory
- How to check for memory leaks (valgrind)



Week 6 begins!

□ Attend lab (laptop required)

□ Read **Rao Lesson 8** (pp. 187-204)

□ Start design for **Assignment 4** (due **Sunday, Feb. 16**)