



Oregon State
University

COLLEGE OF ENGINEERING

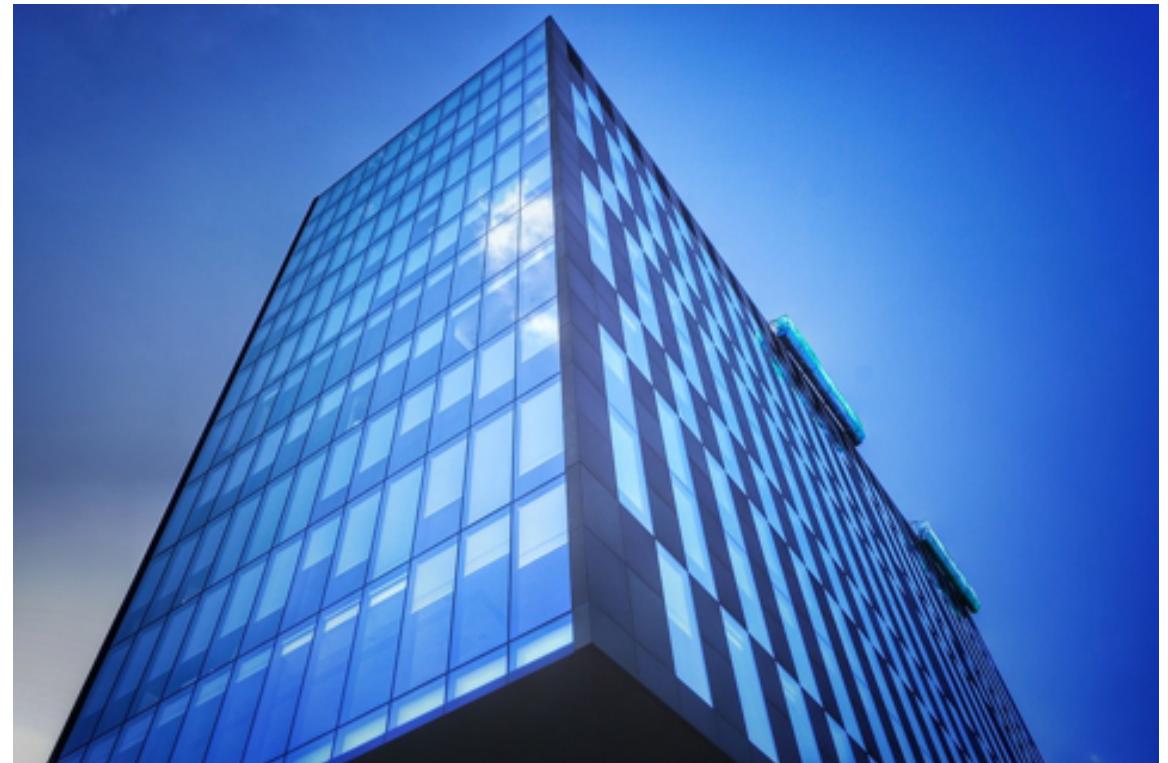
| School of Electrical Engineering
and Computer Science

CS 161

Introduction to CS I

Lecture 20

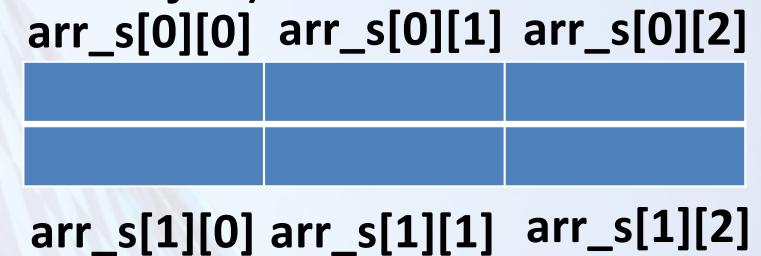
- Multidimensional arrays
- Structs: create your own data types!



Review: Create 2D arrays

- Stack (static, one block of memory, row-major)

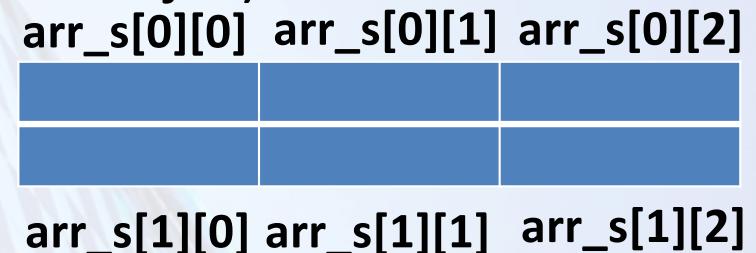
```
1. char arr_s[2][3];
```



Review: Create 2D arrays

- Stack (static, one block of memory, row-major)

```
1. char arr_s[2][3];
```



- Heap (dynamic, pointers to pointers)

```
1. char** arr_d = new char*[2];
2. for (int i=0; i<2; i++)
3.     arr_d[i] = new char[3];
```



Passing static 2D arrays to functions

- Static: must include the size of both dimensions, or at least the final dimension
 - So that we know where each row starts

```
1. int main() {  
2.     int array[3][3];  
3.  
4.     pass_2Darray_1(array);  
5.     pass_2Darray_2(array);  
6.  
7.     return 0;  
8. }
```

```
1. void pass_2Darray_1(int a[3][3]) {  
2.     cout << a[0][0] << endl;  
3. }  
4. /* OR */  
5. void pass_2Darray_2(int a[][3]) {  
6.     cout << a[0][0] << endl;  
7. }
```

Passing dynamic 2D arrays to functions

- Dynamic: no sizes need to be specified, because the row pointers indicate where each row starts

See lec20-pass-2D-arrays.cpp

```
1. int main() {  
2.     int** array;  
3.     /* allocate array */  
4.     pass_2Darray_3(array);  
5.     pass_2Darray_4(array);  
6.     /* free array */  
7.     return 0;  
8. }
```

2/21/2020

```
1. void pass_2Darray_3(int* a[]) {  
2.     cout << a[0][0] << endl;  
3. }  
4. /* OR */  
5. void pass_2Darray_4(int** a) {  
6.     cout << a[0][0] << endl;  
7. }
```

CS 161

5

Useful: include array dimensions

- Just as with 1D arrays, if you want to iterate over items in an array, pass the sizes as arguments

```
1. void pass_static_2Darray(int a[3][2], int rows, int cols);  
2. void pass_static_2Darray(_____, int rows, int cols);  
  
3. void pass_dyn_2Darray(_____, int rows, int cols);  
4. void pass_dyn_2Darray(int** a, int rows, int cols);
```

Useful: include array dimensions

- Just as with 1D arrays, if you want to iterate over items in an array, pass the sizes as arguments

```
1. void pass_static_2Darray(int a[3][2], int rows, int cols);  
2. void pass_static_2Darray(int a[][2], int rows, int cols);  
  
3. void pass_dyn_2Darray(int* a[], int rows, int cols);  
4. void pass_dyn_2Darray(int** a, int rows, int cols);
```

Create and return a 2D char array in a function?

_____ create_2D_array(_____) ;

1. Heap or stack?
2. What return type?
3. What input arguments?

Create and return a 2D char array in a function?

_____ create_2D_array(_____) ;

1. Heap or stack?

- Must be **heap** – stack is freed when function returns

2. What return type?

- **char****

3. What input arguments?

- **int n_rows, int n_cols**

Create and return a 2D int array in a function

Let's do it!

```
1. char** create_2D_array(int r, int c) {  
2.     char** ttt = new char*[r]; /* row pointers */  
3.     for (int i=0; i<r; i++)  
4.         ttt[i] = new char[c]; /* row arrays */  
5.     return ttt;  
6. }
```

Your favorite collectible item

- Anime body pillows
- Military coins
- Books
- Porcelain dolls
- Money
- ...

Let's collect planets

- We want to store, for each planet:
 - Name
 - Radius (in km)
 - Number of moons
 - Color

```
1. string p1_name;  
2. string p2_name;  
3. string p3_name;  
4. ...
```

```
1. string name[9];  
2. float radius[9];  
3. int n_moons[9];  
4. ...
```

Mars



Jupiter



(Not to scale)

Define your own data structure ("struct")

- Package info into one data structure for each item

See lec20-structs.cpp

Members

```
1. struct planet {  
2.     string name;  
3.     float radius;  
4.     unsigned short n_moons;  
5.     string color;  
6. } ;
```

```
1. planet mars;  
2. mars.name = "Mars";  
3. mars.radius = 3389.5;  
4. mars.n_moons = 2;  
5. mars.color = "red";
```

- Much more readable and easier to manipulate

Your turn: Define your structure

- What item do you like to collect?
- What attributes do you want to store?
 - Think of at least 3
 - Choose appropriate data types for each member

```
1. struct planet {  
2.     string name;  
3.     float radius;  
4.     unsigned short n_moons;  
5.     string color;  
6. };
```

Functions on structs

See `lec20-structs.cpp`

```
1. /* Return the name of the largest planet */
2. string largest_planet(planet p1, planet p2) {
3.     if (p1.radius >= p2.radius)
4.         return p1.name;
5.     else /* p2.radius > p1.radius */
6.         return p2.name;
7. }
```

- Access members with the `.` operator (e.g., `p1.radius`)
- Your turn: Think of a function you would like for your struct

Functions on structs

See `lec20-structs.cpp`

- The previous function passed structs by value (made a copy)
- As structs get larger, it is better to **pass by reference**
- If you want to modify values, you must pass an address
(pass by reference, or pass a pointer)

Functions on structs: pass by reference

See lec20-structs.cpp

- 1. /* We discovered a new moon for this planet! */
2. void add_moon_r(**planet&** p) {
3. p.n_moons++;
4. }

```
1. int main() {  
2.   planet jupiter;  
3.   jupiter.n_moons = 79;  
4.   add_moon_r(jupiter);  
5.   cout << jupiter.n_moons << endl;  
6.   return 0;  
7. }
```

Functions on structs: pass a pointer

See lec20-structs.cpp

- 1. /* We discovered a new moon for this planet! */
2. void add_moon_p(planet* p) {
3. (*p).n_moons++;
4. }

```
1. int main() {  
2.   planet jupiter;  
3.   jupiter.n_moons = 79;  
4.   add_moon_p(&jupiter);  
5.   cout << jupiter.n_moons << endl;  
6.   return 0;  
7. }
```

The arrow operator (->)

See lec20-structs.cpp

- 1. /* We discovered a new moon for this planet! */
2. void add_moon_p(planet* p) {
3. p->n_moons++;
4. }

```
1. int main() {  
2.     planet jupiter;  
3.     jupiter.n_moons = 79;  
4.     add_moon_p(&jupiter);  
5.     cout << jupiter.n_moons << endl;  
6.     return 0;  
7. }
```

Your turn: Create an array of 9 structs (planets)

```
1. _____ my_planets[____]; /* stack */
```

```
2. _____ my_planets = new _____ [____]; /* heap */
```

Your turn: Create an array of 9 structs (planets)

```
1. planet my_planets[9]; /* stack */
```

```
2. planet* my_planets = new planet[9]; /* heap */
```

Delete the array of structs off the heap

```
1. planet my_planets[9]; /* stack */  
  
2. planet* my_planets = new planet[9]; /* heap */  
3. delete [] my_planets;
```

Struct initializer

- One member at a time:

```
1. planet mars;  
2. mars.name = "Mars";  
3. mars.radius = 3389.5;  
4. mars.n_moons = 2;  
5. mars.color = "red";
```

- All members at once (in order):

```
1. planet mars = { "Mars", 3389.5, 2, "red" };
```

What vocabulary did we learn today?

- Struct
- Member
- . ("dot") operator: access a member
- -> ("arrow") operator: dereference pointer and access member

What ideas and skills did we learn today?

- How to pass 2D arrays to functions
- How to define your own data structures (structs)
 - e.g., planet
- How to access and update the members of a struct
 - e.g., p.radius
- How to pass structs to a function
- How to create an array of structs on the stack or heap
- How to initialize structs

Week 7 nearly done!

- Attend lab (laptop required)
- Read <http://www.cplusplus.com/doc/tutorial/structures/>
- Assignment 4 (due Sunday, Feb. 23)

See you Monday!