Initial code setup

```
$ cd eecs211
$ curl $URL211/lec/05pointer.tgz | tar zx
...
$ cd 05pointer
```
Road map

- What’s a pointer?
- What can it do?
- What’s the point?
What is a pointer?
int main()
{
    int a = 5, b = 10;
    a = 12;
}

• Variables name objects, which contain values
• Assignment changes the value in an object
• Each object has an address
Review: variables, objects, values

```c
int main()
{
    int a = 5, b = 10;
    a = 12;
}
```

- Variables name objects, which contain values
Review: variables, objects, values

```c
int main()
{
    int a = 5, b = 10;
    a = 12;
}
```

- Variables name objects, which contain values
- Assignment changes the value in an object

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>
Review: variables, objects, values

```c
int main()
{
    int a = 5, b = 10;
    a = 12;
}
```

- Variables name objects, which contain values
- Assignment changes the value in an object
- Each object has an address
Memory is a huge array, and addresses are indices into it.
Memory is a huge array,
and addresses are indices into it.

Array of **chars**: (hexadecimal)

<table>
<thead>
<tr>
<th>...</th>
<th>100</th>
<th>101</th>
<th>102</th>
<th>103</th>
<th>104</th>
<th>105</th>
<th>106</th>
<th>107</th>
<th>108</th>
<th>109</th>
<th>110</th>
<th>111</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>48</td>
<td>65</td>
<td>6C</td>
<td>6C</td>
<td>6F</td>
<td>20</td>
<td>77</td>
<td>6F</td>
<td>72</td>
<td>6C</td>
<td>64</td>
<td>00</td>
<td>...</td>
</tr>
</tbody>
</table>

Array of **shorts**: (little endian)

<table>
<thead>
<tr>
<th>...</th>
<th>100</th>
<th>102</th>
<th>104</th>
<th>106</th>
<th>108</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>48</td>
<td>6C</td>
<td>6C</td>
<td>6F</td>
<td>20</td>
<td>...</td>
</tr>
</tbody>
</table>

Array of **ints**: (big endian)

| ... | 100 | 104 | 108 | ... |
|-----|-----|-----|-----|
| 48  | 65  | 6C  | 6C  |

Mixed!

**double** and **4 chars**:

<table>
<thead>
<tr>
<th>...</th>
<th>100</th>
<th>108</th>
<th>109</th>
<th>110</th>
<th>111</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td>6C</td>
<td>6C</td>
<td>6F</td>
<td>20</td>
<td>77</td>
<td>...</td>
</tr>
</tbody>
</table>
Memory is a huge array, and addresses are indices into it.

Array of **chars**: (hexadecimal)

<table>
<thead>
<tr>
<th>...</th>
<th>100</th>
<th>101</th>
<th>102</th>
<th>103</th>
<th>104</th>
<th>105</th>
<th>106</th>
<th>107</th>
<th>108</th>
<th>109</th>
<th>110</th>
<th>111</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>48</td>
<td>65</td>
<td>6C</td>
<td>6C</td>
<td>6F</td>
<td>20</td>
<td>77</td>
<td>6F</td>
<td>72</td>
<td>6C</td>
<td>64</td>
<td>00</td>
<td>...</td>
</tr>
</tbody>
</table>

Array of **shorts**: (little endian)

<table>
<thead>
<tr>
<th>...</th>
<th>100</th>
<th>102</th>
<th>104</th>
<th>106</th>
<th>108</th>
<th>110</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6548</td>
<td>6C6C</td>
<td>206F</td>
<td>6F77</td>
<td>6C72</td>
<td>0064</td>
<td>...</td>
</tr>
</tbody>
</table>
Memory is a huge array, and addresses are indices into it.

<table>
<thead>
<tr>
<th>Array of <strong>chars</strong>:</th>
<th>(hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>... 100 101 102 103 104 105 106 107 108 109 110 111 ...</td>
<td></td>
</tr>
<tr>
<td>... 48 65 6C 6C 6F 20 77 6F 72 6C 64 00 ...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array of <strong>shorts</strong>:</th>
<th>(little endian)</th>
</tr>
</thead>
<tbody>
<tr>
<td>... 100 102 104 106 108 110 ...</td>
<td></td>
</tr>
<tr>
<td>... 6548 6C6C 206F 6F77 6C72 0064 ...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array of <strong>ints</strong>:</th>
<th>(big endian)</th>
</tr>
</thead>
<tbody>
<tr>
<td>... 100 104 108 ...</td>
<td></td>
</tr>
<tr>
<td>... 48656C6C 6F20776F 726C6400 ...</td>
<td></td>
</tr>
</tbody>
</table>
Memory is a huge array, and addresses are indices into it.

<table>
<thead>
<tr>
<th>Array of <strong>chars</strong>:</th>
<th>(hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>... 100 101 102 103 104 105 106 107 108 109 110 111 ...</td>
<td></td>
</tr>
<tr>
<td>⋯ 48 65 6C 6C 6F 20 77 6F 72 6C 64 00 ⋯</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array of <strong>shorts</strong>:</th>
<th>(little endian)</th>
</tr>
</thead>
<tbody>
<tr>
<td>... 100 102 104 106 108 110 ...</td>
<td></td>
</tr>
<tr>
<td>⋯ 6548 6C6C 206F 6F77 6C72 0064 ⋯</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array of <strong>ints</strong>:</th>
<th>(big endian)</th>
</tr>
</thead>
<tbody>
<tr>
<td>... 100 104 108 ...</td>
<td></td>
</tr>
<tr>
<td>⋯ 48656C6C 6F20776F 726C6400 ⋯</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Mixed! double and 4 chars</strong>:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>... 100 ...</td>
<td></td>
</tr>
<tr>
<td>⋯ 1.56C6C6F20776Fp+135 72 6C 64 00 ⋯</td>
<td></td>
</tr>
</tbody>
</table>
Let’s see some real addresses

We can get the address of a variable using the & operator, and format it with printf’s "%p" (after casting it to the “universal” pointer type void*):

```c
int main()
{
    int a = 5, b = 7, c = 9;

    printf("a: \%d\n", a);
    printf("b: \%d\n", b);
    printf("c: \%d\n", c);

    printf("&a: \%p\n", (void*) &a);
    printf("&b: \%p\n", (void*) &b);
    printf("&c: \%p\n", (void*) &c);
}
```
Output from previous slide

$ build/addresses
a: 5
b: 7
c: 9
&a: 0x7ffee536816c
&b: 0x7ffee5368168
&c: 0x7ffee5368164

Note that the addresses (in hexadecimal) are 4 bytes apart, which must by sizeof(int) on my system.
Output from previous slide

$ build/addresses
a: 5
b: 7
c: 9
&a: 0x7ffee536816c
&b: 0x7ffee5368168
&c: 0x7ffee5368164

Note that the addresses (in hexadecimal) are 4 bytes apart, which must by `sizeof(int)` on my system.
Pointers

- We can store the address of one object in another object.
Pointers

- We can store the address of one object in another object
- A object containing an address is called a pointer
Pointers

- We can store the address of one object in another object
- A object containing an address is called a *pointer*
- A pointer to an object of any type $T$ has type $T*$
Pointers

- We can store the address of one object in another object
- A object containing an address is called a *pointer*
- A pointer to an object of any type $T$ has type $T*$

```c
int main()
{
    int a = 5, b = 7;
    int* ip;
    ip = &a;
    ip = &b;
}
```
Pointers

- We can store the address of one object in another object
- A object containing an address is called a pointer
- A pointer to an object of any type $T$ has type $T*$

```c
int main()
{
    int a = 5, b = 7;
    int* ip;
    ip = &a;
    ip = &b;
}
```

`a @100`  `b @104`

`5`  `7`
Pointers

• We can store the address of one object in another object
• A object containing an address is called a pointer
• A pointer to an object of any type $T$ has type $T*$

```c
int main()
{
    int a = 5, b = 7;
    int* ip;
    ip = &a;
    ip = &b;
}
```

```
a @100  b @104  ip @108
5       7
```
Pointers

- We can store the address of one object in another object
- A object containing an address is called a pointer
- A pointer to an object of any type $T$ has type $T*$

```c
int main()
{
    int a = 5, b = 7;
    int* ip;
    ip = &a;
    ip = &b;
}
```

```
5 100 7 104 100
```
Pointers

- We can store the address of one object in another object.
- A object containing an address is called a pointer.
- A pointer to an object of any type $T$ has type $T^*$.

```c
int main()
{
    int a = 5, b = 7;
    int* ip;
    ip = &a;
    ip = &b;
}
```

<table>
<thead>
<tr>
<th>a</th>
<th>@100</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>@104</td>
<td>7</td>
</tr>
<tr>
<td>ip</td>
<td>@108</td>
<td>104</td>
</tr>
</tbody>
</table>
Pointers

- We can store the address of one object in another object
- A object containing an address is called a pointer
- A pointer to an object of any type $T$ has type $T*$

```c
int main()
{
    int a = 5, b = 7;
    int* ip;
    ip = &a;
    ip = &b;
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>@100</td>
<td>5</td>
</tr>
<tr>
<td>@104</td>
<td>7</td>
</tr>
<tr>
<td>@108</td>
<td></td>
</tr>
</tbody>
</table>
Pointers

- We can store the address of one object in another object
- A object containing an address is called a pointer
- A pointer to an object of any type $T$ has type $T^*$

```c
int main()
{
    int a = 5, b = 7;
    int* ip;
    ip = &a;
    ip = &b;
}
```

```
<table>
<thead>
<tr>
<th></th>
<th>a @100</th>
<th>b @104</th>
<th>ip @108</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>5</td>
<td>7</td>
<td>[]</td>
</tr>
</tbody>
</table>
```
Pointers

- We can store the address of one object in another object
- A object containing an address is called a pointer
- A pointer to an object of any type $T$ has type $T*$

```c
int main()
{
    int a = 5, b = 7;
    int* ip;
    ip = &a;
    ip = &b;
}
```

```
a @100  b @104  ip @108
5 7
```
What’s with the syntax?

```c
int* p;
```
What’s with the syntax?

```c
int* p;
int *p;
```
What’s with the syntax?

```c
int* p;
int *p;
int * p;
int * p;
int*p;
```
What’s with the syntax?

```c
int* p;
int *p;
int * p;
int*p;
int*p;
    int

    *

    p ;
```
What’s with the syntax?

```c
int* p;
int *p;
int * p;  // don’t
int*p;    // don’t
          int

*  
p    
```

// o_o
What’s with the syntax?

int* p;
int *p;
int * p;  // don’t
int*p;     // don’t
        int
            // o_o
*               
            
p ;
What’s with the syntax?

```c
int* p;       // “p is an int*”
int *p;       // “*p is an int”
int * p;      // don’t
int*p;        // don’t
int
   // o_o
*  

p ;
```
Beware!

What does this mean?

```c
int* p, q;
```
Beware!

What does this mean?

\texttt{int\,*\,p,\,q;} \equiv \texttt{int\,*\,p,\,q;}
What does this mean?

\[ \text{int* p, q; } \equiv \text{ int *p, q; } \equiv \text{ int *p; int q;} \]
Beware!

What does this mean?
\[
\text{int* } p, q; \equiv \text{int* } p, q; \equiv \text{int* } p; \text{ int } q;
\]

So you gotta write:

\[
\text{int* } p; \\
\text{int* } q;
\]
Beware!

What does this mean?

\[ \texttt{int* p, q;} \equiv \texttt{int *p, q;} \equiv \texttt{int *p; int q;} \]

So you gotta write:

\[ \texttt{int* p;} \]
\[ \texttt{int* q;} \quad \text{or} \quad \texttt{int *p, *q;} \]
Beware!

What does this mean?

\[
\text{int* } p, q; \equiv \text{ int } *p, q; \equiv \text{ int } *p; \text{ int } q;
\]

So you gotta write:

```c
int* p;
int* q; \text{ or int } *p, *q; \quad (but \text{ please not int* } p,* \text{ q;})
```
What can it do?
What can you do with a pointer?

You can dereference (or “follow”) it, using the * operator:

```c
int main()
{
    int y = 5, z = 7;
    int* ip = &y;       // referent is y
    z = *ip + 1;        // use value of referent
    *ip = 9;            // assign to referent
}
```
What can you do with a pointer?

You can dereference (or “follow”) it, using the * operator:

```c
int main()
{
    int y = 5, z = 7;
    int* ip = &y;         // referent is y
    z = *ip + 1;          // use value of referent
    *ip = 9;              // assign to referent
}
```

```
y @100  z @104
5   7
```
What can you do with a pointer?

You can dereference (or “follow”) it, using the * operator:

```c
int main()
{
    int y = 5, z = 7;
    int* ip = &y;      // referent is y
    z = *ip + 1;       // use value of referent
    *ip = 9;           // assign to referent
}
```

```
y @100      z @104      ip @108
5           7           
```
What can you do with a pointer?

You can dereference (or “follow”) it, using the `*` operator:

```c
int main()
{
    int y = 5, z = 7;
    int* ip = &y;  // referent is y
    z = *ip + 1;   // use value of referent
    *ip = 9;       // assign to referent
}
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>5</td>
</tr>
<tr>
<td>z</td>
<td>6</td>
</tr>
<tr>
<td>ip</td>
<td>9</td>
</tr>
</tbody>
</table>
What can you do with a pointer?

You can dereference (or “follow”) it, using the \* operator:

```c
int main()
{
    int y = 5, z = 7;
    int* ip = &y;    // referent is y
    z = *ip + 1;    // use value of referent
    *ip = 9;        // assign to referent
}
```

y @100  z @104  ip @108

9  6  0

Try on C Tutor ➔
Can a struct contain a struct?

Can a struct contain an array?

Can a struct contain a pointer?

Can you have an array of structs?

Can you have an array of arrays?

Can you have an array of pointers?

Can you have a pointer to a struct?

Can you have a pointer to an array?

Can you have a pointer to a pointer?

Can you have a pointer to a field of a struct?

Can you have a pointer to an element of an array?

Can you have a pointer to a field of struct which is an element of an array which is a field of a struct?

Yes.

† Yes, but declaring it looks weird.

‡ Can you not have a pointer to an array?
FAQ

Can a struct contain a struct?

Yes.
FAQ

Can a struct contain a struct?  Can a struct contain an array?
FAQ

Can a struct contain a struct? Can a struct contain an array? Can a struct contain a pointer?
Can a struct contain a struct?  
Can a struct contain an array?  
Can a struct contain a pointer?  
Can you have an array of structs?  
Can you have an array of arrays?  
Can you have an array of pointers?
FAQ

Can a struct contain a struct?  Can a struct contain an array?  Can a struct contain a pointer?  Can you have an array of structs?  Can you have an array of arrays?  Can you have an array of pointers?  Can you have a pointer to a struct?  Can you have a pointer to an array?  Can you have a pointer to a pointer?
FAQ

Can a struct contain a struct?  Can a struct contain an array?  Can a struct contain a pointer?  Can you have an array of structs?  Can you have an array of arrays?  Can you have an array of pointers?  Can you have a pointer to a struct?  Can you have a pointer to an array?  Can you have a pointer to a pointer?  Can you have a pointer to a field of a struct?
FAQ

Can a struct contain a struct?  Can a struct contain an array?  Can a struct contain a pointer?  Can you have an array of structs?  Can you have an array of arrays?  Can you have an array of pointers?  Can you have a pointer to a struct?  Can you have a pointer to an array?  Can you have a pointer to a pointer?  Can you have a pointer to a field of a struct?  Can you have a pointer to an element of an array?
FAQ

Can a struct contain a struct?  Can a struct contain an array?  Can a struct contain a pointer?  Can you have an array of structs?  Can you have an array of arrays?  Can you have an array of pointers?  Can you have a pointer to a struct?  Can you have a pointer to an array?  Can you have a pointer to a pointer?  Can you have a pointer to a field of a struct?  Can you have a pointer to an element of an array?  Can you have a pointer to a field of struct which is an element of an array which is a field of a struct?
FAQ

Can a struct contain a struct?* Can a struct contain an array?* Can a struct contain a pointer?* Can you have an array of structs?* Can you have an array of arrays? Can you have an array of pointers?* Can you have a pointer to a struct?* Can you have a pointer to an array? Can you have a pointer to a pointer?* Can you have a pointer to a field of a struct?* Can you have a pointer to an element of an array?* Can you have a pointer to a field of struct which is an element of an array which is a field of a struct?*

* Yes.
Can a struct contain a struct?* Can a struct contain an array?*
Can a struct contain a pointer?* Can you have an array of structs?* **Can you have an array of arrays?**† Can you have an array of pointers?* Can you have a pointer to a struct?* Can you have a pointer to an array?  Can you have a pointer to a pointer?* Can you have a pointer to a field of a struct?* Can you have a pointer to an element of an array?* Can you have a pointer to a field of struct which is an element of an array which is a field of a struct?*

* Yes.
† Yes, but declaring it looks weird.
FAQ

Can a struct contain a struct?* Can a struct contain an array?* Can a struct contain a pointer?* Can you have an array of structs?* **Can you have an array of arrays?**† Can you have an array of pointers?* Can you have a pointer to a struct?* **Can you have a pointer to an array?**‡ Can you have a pointer to a pointer?* Can you have a pointer to a field of a struct?* Can you have a pointer to an element of an array?* Can you have a pointer to a field of struct which is an element of an array which is a field of a struct?*

* Yes.
† Yes, but declaring it looks weird.
‡ Can you not have a pointer to an array?
Everything is compositional

typedef struct { short h, k; } entry;
typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6];
    entry *some_entry;
    short *some_subentry;
    entry *some_entries[12];
    entry (*some_row)[6];
    entry **some_ptr;
}
Everything is compositional

typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6]; // array of 3 arrays of 6 structs
    entry *some_entry;
    short *some_subentry;
    entry *some_entries[12];
    entry (*some_row)[6];
    entry **some_ptr;
}

typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6];       // array of 3 arrays of 6 structs
    entry *some_entry;      // pointer to struct
    short *some_subentry;
    entry *some_entries[12];
    entry (*some_row)[6];
    entry **some_ptr;
}

Everything is compositional

typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6]; // array of 3 arrays of 6 structs
    entry *some_entry; // pointer to struct
    short *some_subentry; // pointer to field of struct
    entry *some_entries[12];
    entry (*some_row)[6];
    entry **some_ptr;
}
Everything is compositional

typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6];       // array of 3 arrays of 6 structs
    entry *some_entry;      // pointer to struct
    short *some_subentry;   // pointer to field of struct
    entry *some_entries[12]; // array of 12 pointers to structs
    entry (*some_row)[6];   // array of 12 pointers to structs
    entry **some_ptr;
}
Everything is compositional

typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6]; // array of 3 arrays of 6 structs
    entry *some_entry; // pointer to struct
    short *some_subentry; // pointer to field of struct
    entry *some_entries[12]; // array of 12 pointers to structs
    entry (*some_row)[6]; // pointer to array of 6 structs
    entry **some_ptr;
}

Everything is compositional

typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6];            // array of 3 arrays of 6 structs
    entry *some_entry;           // pointer to struct
    short *some_subentry;        // pointer to field of struct
    entry *some_entries[12];     // array of 12 pointers to structs
    entry (*some_row)[6];        // pointer to array of 6 structs
    entry **some_ptr;            // pointer to pointer to struct
}
typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6];         // array of 3 arrays of 6 structs
    entry *some_entry;        // pointer to struct
    short *some_subentry;     // pointer to field of struct
    entry *some_entries[12];  // array of 12 pointers to structs
    entry (*some_row)[6];     // pointer to array of 6 structs
    entry **some_ptr;         // pointer to pointer to struct
} m;
Everything is compositional

typedef struct { short h, k; } entry;

struct matrix
{
  entry data[3][6];                // array of 3 arrays of 6 structs  
  entry *some_entry;              // pointer to struct        
  short *some_subentry;           // pointer to field of struct 
  entry *some_entries[12];        // array of 12 pointers to structs  
  entry (*some_row)[6];           // pointer to array of 6 structs  
  entry **some_ptr;               // pointer to pointer to struct 
} m;

m.data[2][5].h = 6;
typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6]; // array of 3 arrays of 6 structs
    entry *some_entry; // pointer to struct
    short *some_subentry; // pointer to field of struct
    entry *some_entries[12]; // array of 12 pointers to structs
    entry (*some_row)[6]; // pointer to array of 6 structs
    entry **some_ptr; // pointer to pointer to struct
} m;

m.some_entry = &m.data[row][col];
m.some_subentry = &m.data[row][col].k;
Everything is compositional

typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6];            // array of 3 arrays of 6 structs
    entry *some_entry;            // pointer to struct
    short *some_subentry;         // pointer to field of struct
    entry *some_entries[12];      // array of 12 pointers to structs
    entry (*some_row)[6];         // pointer to array of 6 structs
    entry **some_ptr;             // pointer to pointer to struct
} m;

m.some_entry = &(((m.data)[row])[col]);
m.some_subentry = &((((m.data)[row])[col]).k);
typedef struct { short h, k; } entry;

struct matrix {
  entry data[3][6]; // array of 3 arrays of 6 structs
  entry *some_entry; // pointer to struct
  short *some_subentry; // pointer to field of struct
  entry *some_entries[12]; // array of 12 pointers to structs
  entry (*some_row)[6]; // pointer to array of 6 structs
  entry **some_ptr; // pointer to pointer to struct
}
m;

m.some_entry = &(m.data[row][col]);
m.some_subentry = &(m.data[row][col].k);
typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6]; // array of 3 arrays of 6 structs
    entry *some_entry; // pointer to struct
    short *some_subentry; // pointer to field of struct
    entry *some_entries[12]; // array of 12 pointers to structs
    entry (*some_row)[6]; // pointer to array of 6 structs
    entry **some_ptr; // pointer to pointer to struct
} m;

m.some_entry->k = 7;
*m.some_subentry = 7;
Everything is compositional

typedef struct { short h, k; } entry;

struct matrix
{
  entry data[3][6]; // array of 3 arrays of 6 structs
  entry *some_entry; // pointer to struct
  short *some_subentry; // pointer to field of struct
  entry *some_entries[12]; // array of 12 pointers to structs
  entry (*some_row)[6]; // pointer to array of 6 structs
  entry **some_ptr; // pointer to pointer to struct
} m;

m.some_entry->k = 7;
*(m.some_subentry) = 7;
typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6]; // array of 3 arrays of 6 structs
    entry *some_entry; // pointer to struct
    short *some_subentry; // pointer to field of struct
    entry *some_entries[12]; // array of 12 pointers to structs
    entry (*some_row)[6]; // pointer to array of 6 structs
    entry **some_ptr; // pointer to pointer to struct
} m;

m.some_entries[1] = &m.data[1][2];
m.some_entries[1]->h = 8;
Everything is compositional

typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6]; // array of 3 arrays of 6 structs
    entry *some_entry; // pointer to struct
    short *some_subentry; // pointer to field of struct
    entry *some_entries[12]; // array of 12 pointers to structs
    entry (*some_row)[6]; // pointer to array of 6 structs
    entry **some_ptr; // pointer to pointer to struct
} m;

m.some_row = &m.data[row];
(*m.some_row)[col].h = 9; // necessary parentheses!
Everything is compositional

typedef struct { short h, k; } entry;

struct matrix
{
    entry data[3][6]; // array of 3 arrays of 6 structs
    entry *some_entry; // pointer to struct
    short *some_subentry; // pointer to field of struct
    entry *some_entries[12]; // array of 12 pointers to structs
    entry (*some_row)[6]; // pointer to array of 6 structs
    entry **some_ptr; // pointer to pointer to struct
} m;

m.some_ptr = &m.some_entries[cur];
*m.some_ptr = m.some_entry;
Okay, but why?
What’s the point?

- “Talk about” objects
- Avoid copying
- They’re super general
- Unnamed objects (next time)
Let’s talk about objects

```c
void swap(int* ip, int* jp)
{
    int temp = *ip;
    *ip = *jp;
    *jp = temp;
}

int x = 5, y = 7;
swap(&x, &y);
```
void swap(int* ip, int* jp) {
    int temp = *ip;
    *ip = *jp;
    *jp = temp;
}

int x = 5, y = 7;
swap(&x, &y);

x @100 y @104
5 7
Let’s talk about objects

```c
void swap(int* ip, int* jp) {
    int temp = *ip;
    *ip = *jp;
    *jp = temp;
}
```

```c
int x = 5, y = 7;
swap(&x, &y);
```

(caller’s stack frame)

(ip @200)
(100)
(jp @208)
(104)

(temp @216)
(5)

(y @104)
(7)

(x @100)
(5)
Let’s talk about objects

```c
void swap(int* ip, int* jp)
{
    int temp = *ip;
    *ip = *jp;
    *jp = temp;
}

int x = 5, y = 7;
swap(&x, &y);
```

(x @100  y @104
  5 7
ip @200  jp @208  temp @216
  100 104 5)

(caller’s stack frame)

(temp @216
  5)

(swap’s stack frame)
Let’s talk about objects

```c
void swap(int* ip, int* jp)
{
    int temp = *ip;
    *ip = *jp;
    *jp = temp;
}

int x = 5, y = 7;
swap(&x, &y);
```

(x @100  y @104)

7 7

(ip @200  jp @208  temp @216)

100 104 5

(caller’s stack frame)

(swap’s stack frame)
Let’s talk about objects

```c
void swap(int* ip, int* jp)
{
    int temp = *ip;
    *ip = *jp;
    *jp = temp;
}
```

```c
int x = 5, y = 7;
swap(&x, &y);
```

(caller’s stack frame)

(ip @200 100 100 100
  5 5 5 5
  7 7 7 7)

(jp @208 104 104 104
  5 5 5 5
  7 7 7 7)

(temp @216 5 5 5 5
  7 7 7 7
  100 100 100)

(caller’s stack frame)
Let’s talk about objects

```c
void swap(int* ip, int* jp)
{
    int temp = *ip;
    *ip = *jp;
    *jp = temp;
}
```

```c
int x = 5, y = 7;
swap(&x, &y);
```

(caller’s stack frame)
Avoiding copying

#define N  1024

struct intvec
{
    size_t count;
    int    data[N];
};

void push(struct intvec r, int v)
{
    r.data[r.count] = v;
    ++r.count;
}
Avoiding copying

#define N 1024

struct intvec
{
    size_t count;
    int    data[N];
};

struct intvec push(struct intvec r, int v)
{
    r.data[r.count] = v;
    ++r.count;
    return r;
}
Avoiding copying

```
#define N 1024

struct intvec
{
    size_t count;
    int data[N];
};

void push(struct intvec* r, int v)
{
    ++(*r).count;
}
```
Avoiding copying

#define N 1024

struct intvec
{
    size_t count;
    int data[N];
};

void push(struct intvec* r, int v)
{
    r->data[r->count] = v;
    ++r->count;
}

Syntactic sugar: \langle ptr \rangle \rightarrow \langle field \rangle \ means \ (\ast \langle ptr \rangle) . \langle field \rangle
Arrays decay to pointers

```c
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]); // ⇒ 0x7ffee5c6e2f0
put_ptr(a);
put_int(a[0]);
put_int(*a);
```

Arrays decay to pointers

```c
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);          // ⇒ 0x7ffee5c6e2f0
put_ptr(a);              // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);           // ⇒ 0x7ffee5c6e2f0
put_int(*a);             // ⇒ 0x7ffee5c6e2f0
```

for (int i = 0; i < 5; i++) {
    put_int(a[i]);
    put_int(*(a + i));
    put_size(sizeof a);
    put_size(sizeof (a + i));
}
Arrays decay to pointers

```c
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]); // ⇒ 0x7ffee5c6e2f0
put_ptr(a); // ⇒ 0x7ffee5c6e2f0
put_int(a[0]); // ⇒ 2
put_int(*a);
```

Arrays decay to pointers

```c
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);                // ⇒ 0x7fffee5c6e2f0
put_ptr(a);                    // ⇒ 0x7fffee5c6e2f0
put_int(a[0]);                 // ⇒ 2
put_int(*a);                    // ⇒ 2
```

Try on C Tutor ➔
Arrays decay to pointers

```c
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]); // ⇒ 0x7ffee5c6e2f0
put_ptr(a);     // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);  // ⇒ 2
put_int(*a);    // ⇒ 2

put_ptr(&a[1]);
put_ptr(a + 1);
put_int(a[1]);
put_int(*(a + 1));
```
Arrays decay to pointers

```c
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]); // ⇒ 0x7ffee5c6e2f0
put_ptr(a); // ⇒ 0x7ffee5c6e2f0
put_int(a[0]); // ⇒ 2
put_int(*a); // ⇒ 2

put_ptr(&a[1]); // ⇒ 0x7ffee5c6e2f4
put_ptr(a + 1);
put_int(a[1]);
put_int(*(a + 1));
```
Arrays decay to pointers

```c
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]); // => 0x7ffee5c6e2f0
put_ptr(a); // => 0x7ffee5c6e2f0
put_int(a[0]); // => 2
put_int(*a); // => 2

put_ptr(&a[1]); // => 0x7ffee5c6e2f4
put_ptr(a + 1);
put_int(a[1]);
put_int(*(a + 1));
```
Arrays decay to pointers

```c
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]); // ⇒ 0x7ffee5c6e2f0
put_ptr(a);     // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);  // ⇒ 2
put_int(*a);    // ⇒ 2

put_ptr(&a[1]); // ⇒ 0x7ffee5c6e2f4
put_ptr(a + 1); // ⇒ 0x7ffee5c6e2f4
put_int(a[1]);  // ⇒ 3
put_int(*(a + 1));
```

put_size(sizeof a); // ⇒ 20
put_size(sizeof (a + 0)); // ⇒ 8
Arrays decay to pointers

```c
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]); // ⇒ 0x7ffee5c6e2f0
put_ptr(a); // ⇒ 0x7ffee5c6e2f0
put_int(a[0]); // ⇒ 2
put_int(*a); // ⇒ 2

put_ptr(&a[1]); // ⇒ 0x7ffee5c6e2f4
put_ptr(a + 1); // ⇒ 0x7ffee5c6e2f4
put_int(a[1]); // ⇒ 3
put_int(*(a + 1));
```

20
Arrays *decay* to pointers

```c
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]); // ⇒ 0x7ffee5c6e2f0
put_ptr(a); // ⇒ 0x7ffee5c6e2f0
put_int(a[0]); // ⇒ 2
put_int(*a); // ⇒ 2

put_ptr(&a[1]); // ⇒ 0x7ffee5c6e2f4
put_ptr(a + 1); // ⇒ 0x7ffee5c6e2f4
put_int(a[1]); // ⇒ 3
put_int(*(a + 1)); // ⇒ 3
```
Arrays decay to pointers

```c
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);       // ⇒ 0x7ffee5c6e2f0
put_ptr(a);           // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);        // ⇒ 2
put_int(*a);          // ⇒ 2

put_ptr(&a[1]);       // ⇒ 0x7ffee5c6e2f4
put_ptr(a + 1);       // ⇒ 0x7ffee5c6e2f4
put_int(a[1]);        // ⇒ 3
put_int(*(a + 1));    // ⇒ 3

put_size(sizeof a);
put_size(sizeof (a + 0));
```
Arrays decay to pointers

```c
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]); // ⇒ 0x7ffee5c6e2f0
put_ptr(a); // ⇒ 0x7ffee5c6e2f0
put_int(a[0]); // ⇒ 2
put_int(*a); // ⇒ 2

put_ptr(&a[1]); // ⇒ 0x7ffee5c6e2f4
put_ptr(a + 1); // ⇒ 0x7ffee5c6e2f4
put_int(a[1]); // ⇒ 3
put_int(*(a + 1)); // ⇒ 3

put_size(sizeof a); // ⇒ 20
put_size(sizeof (a + 0)); // ⇒ 8
```
Array indexing is pointer arithmetic

\[
\langle aexpr \rangle[\langle iexpr \rangle] \quad \text{means} \quad *(\langle aexpr \rangle + \langle iexpr \rangle)
\]
Array indexing is pointer arithmetic

\[ \langle aexpr \rangle [\langle iexpr \rangle] \] means \( \ast (\langle aexpr \rangle + \langle iexpr \rangle) \)

\&\langle aexpr \rangle [\langle iexpr \rangle] \] means \( \langle aexpr \rangle + \langle iexpr \rangle \)
Strings are arrays of chars

#include <stdio.h>

int main()
{
    char mystery[] = {
        71, 111, 32, 39, 67, 97, 116, 115, 33, 0
    };

    printf("%s\n", mystery);
}

Strings are arrays of chars

```c
#include <stdio.h>

int main()
{
    char mystery[] = {
        71, 'o', 32, 39, 67, 97, 116, 115, 33, 0
    };

    printf("%s\n", mystery);
}
```
#include <stdio.h>

int main()
{
    char mystery[] = {
        71, 'o', 32, 39, 67, 'a', 116, 115, 33, 0
    };

    printf("%s\n", mystery);
}

Strings are arrays of chars
Strings are arrays of chars

```c
#include <stdio.h>

int main()
{
    char mystery[] = {
        71, 'o', 32, 39, 67, 'a', 't', 115, 33, 0
    };

    printf("%s\n", mystery);
}
```
Strings are arrays of chars

```c
#include <stdio.h>

int main()
{
    char mystery[] = {
        71, 'o', 32, 39, 67, 'a', 't', 's', 33, 0
    };

    printf("%s\n", mystery);
}
```
Strings are arrays of chars

```c
#include <stdio.h>

int main()
{
    char mystery[] = {
        71, 'o', 32, 39, 67, 'a', 't', 's', '!', 0
    };

    printf("%s\n", mystery);
}
```
Strings are arrays of chars

```c
#include <stdio.h>

int main()
{
    char mystery[] = {
        71, 'o', 32, 39, 67, 'a', 't', 's', '!', '\0'
    };

    printf("%s\n", mystery);
}
```
Strings are arrays of chars

```c
#include <stdio.h>

int main()
{
    char mystery[] = {
        71, 'o', 32, '\', 67, 'a', 't', 's', '!', '\0'
    };

    printf("%s\n", mystery);
}
```
How long is a C string?

```c
int main()
{
    const char* cptr = "12345";

    //
```

23
How long is a C string?

```c
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr); // ⇒ ?
    printf("%zu\n", sizeof *cptr); // ⇒ ?
    printf("%zu\n", sizeof(const char*)); // ⇒ ?
    printf("%zu\n", sizeof(const char)); // ⇒ ?
    const char carray[] = "12345";
    printf("%zu\n", sizeof carray); // ⇒ ?
    printf("%zu\n", sizeof(const char[6])); // ⇒ ?
    for (size_t i = 0; i < sizeof carray; ++i)
        printf("%d \n", (int) carray[i]); // ⇒ ?
}
```
How long is a C string?

```c
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr); // ⇒ 8

    const char carray[] = "12345";
    printf("%zu\n", sizeof carray); // ⇒ ?
    printf("%zu\n", sizeof(const char[6])); // ⇒ ?

    for (size_t i = 0; i < sizeof carray; ++i)
        printf("%d␣", (int) carray[i]); // ⇒ ?
}
```
How long is a C string?

```c
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr); // ⇒ 8
    printf("%zu\n", sizeof *cptr); // ⇒ ?

    const char carray[] = "12345";
    printf("%zu\n", sizeof carray); // ⇒ ?
    printf("%zu\n", sizeof const char[6]); // ⇒ ?

    for (size_t i = 0; i < sizeof carray; ++i)
        printf("%d␣", (int) carray[i]); // ⇒ ?
}
```
How long is a C string?

```c
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);  // ⇒ 8
    printf("%zu\n", sizeof *cptr);  // ⇒ 1
    const char carray[] = "12345";
    printf("%zu\n", sizeof carray);  // ⇒ ?
    printf("%zu\n", sizeof const char[6]);  // ⇒ ?
    for (size_t i = 0; i < sizeof carray; ++i)
        printf("%d \n", (int) carray[i]);  // ⇒ ?
}
```
How long is a C string?

```c
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);  // ⇒ 8
    printf("%zu\n", sizeof *cptr); // ⇒ 1
    printf("%zu\n", sizeof(const char*)); // ⇒ 8
    printf("%zu\n", sizeof(const char)); // ⇒ 1
    for (size_t i = 0; i < sizeof carray; ++i)
        printf("%d␣", (int) carray[i]);
```

How long is a C string?

```c
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);        // ⇒ 8
    printf("%zu\n", sizeof *cptr);      // ⇒ 1
    printf("%zu\n", sizeof(const char*)); // ⇒ 8
    printf("%zu\n", sizeof(const char));  // ⇒ 1

    const char carray[] = "12345";
    printf("%zu\n", sizeof carray);      // ⇒ ?
}
```
How long is a C string?

```c
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);  // ⇒ 8
    printf("%zu\n", sizeof *cptr); // ⇒ 1
    printf("%zu\n", sizeof(const char*)); // ⇒ 8
    printf("%zu\n", sizeof(const char)); // ⇒ 1

    const char carray[] = "12345";
    printf("%zu\n", sizeof carray);  // ⇒ 6

    for (size_t i = 0; i < sizeof carray; ++i)
        printf("%d␣", (int) carray[i]);
    // ⇒ 12345
}
```
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);  // ⇒ 8
    printf("%zu\n", sizeof *cptr);  // ⇒ 1
    printf("%zu\n", sizeof(const char*)); // ⇒ 8
    printf("%zu\n", sizeof(const char)); // ⇒ 1

    const char carray[] = "12345";
    printf("%zu\n", sizeof carray); // ⇒ 6
    printf("%zu\n", sizeof(const char[6])); // ⇒ 6

}
How long is a C string?

```c
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr); // ⇒ 8
    printf("%zu\n", sizeof *cptr); // ⇒ 1
    printf("%zu\n", sizeof(const char*)); // ⇒ 8
    printf("%zu\n", sizeof(const char)); // ⇒ 1

    const char carray[] = "12345";
    printf("%zu\n", sizeof carray); // ⇒ 6
    printf("%zu\n", sizeof(const char[6])); // ⇒ 6

    for (size_t i = 0; i < sizeof carray; ++i)
        printf("%d␣", (int) carray[i]); // ⇒ ?
}
```
How long is a C string?

```c
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr); // ⇒ 8
    printf("%zu\n", sizeof *cptr); // ⇒ 1
    printf("%zu\n", sizeof(const char*)); // ⇒ 8
    printf("%zu\n", sizeof(const char)); // ⇒ 1

    const char carray[] = "12345";
    printf("%zu\n", sizeof carray); // ⇒ 6
    printf("%zu\n", sizeof(const char[6])); // ⇒ 6

    for (size_t i = 0; i < sizeof carray; ++i)
        printf("%d␣", (int) carray[i]);
    // ⇒ 49 50 51 52 53 0
}
```
A string algorithm

```c
size_t count_chars(const char* s) {
    size_t result = 0;
    while (*s++) ++result;
    return result;
}
```
A string algorithm

```c
size_t count_chars(const char* s)
{
    size_t result = 0;
    while (*s++) ++result;
    return result;
}
```

```c
size_t count_chars(const char* s)
{
    size_t i = 0;
    while (s[i] != '\0') ++i;
    return i;
}
```
A string algorithm

```c
size_t count_chars(const char* s)
{
    size_t result = 0;
    while (*s++) ++result;
    return result;
}
```

```c
size_t count_chars(const char* s)
{
    const char* t = s;
    while (*t) ++t;
    return t - s;
}
```
Counting characters

```c
int main()
{
    const char carray[] = "12345",
    *cptr    = "12345";

    printf("%zu\n", count_chars(carray));  // ⇒ ?
    printf("%zu\n", count_chars(cptr));   // ⇒ ?
    printf("%zu\n", count_chars(buf));   // ⇒ ?
    printf("%s\n", buf);                // ⇒ ?
}
```
Counting characters

```c
int main()
{
    const char carray[] = "12345",
    *cptr    = "12345";

    printf("%zu\n", count_chars(carray));  // ⇒ 5
    printf("%zu\n", count_chars(cptr));   // ⇒ 5
}
```
Counting characters

```c
int main()
{
    const char carray[] = "12345",
                        *cptr   = "12345";

    printf("%zu\n", count_chars(carray)); // ⇒ 5
    printf("%zu\n", count_chars(cptr));   // ⇒ 5

    char buf[800] = {'a'};
    printf("%zu\n", sizeof buf);          // ⇒ ?
    printf("%zu\n", count_chars(buf));    // ⇒ ?
}
```
Counting characters

```c
int main()
{
    const char carray[] = "12345",
                        *cptr = "12345";

    printf("%zu\n", count_chars(carray)); // ⇒ 5
    printf("%zu\n", count_chars(cptr));   // ⇒ 5

    char buf[800] = {'a'};
    printf("%zu\n", sizeof buf);         // ⇒ 800
    printf("%zu\n", count_chars(buf));   // ⇒ 1
}
```
Counting characters

```c
int main()
{
    const char carray[] = "12345",
    *cptr = "12345";

    printf("%zu\n", count_chars(carray)); // ⇒ 5
    printf("%zu\n", count_chars(cptr)); // ⇒ 5

    char buf[800] = {'a'};
    printf("%zu\n", sizeof buf); // ⇒ 800
    printf("%zu\n", count_chars(buf)); // ⇒ 1

    printf("%zu\n", count_chars(buf)); // ⇒ ?
    printf("%s\n", buf); // ⇒ ?
}
```
Counting characters

```c
int main()
{
    const char carray[] = "12345",
        *cptr = "12345";

    printf("%zu\n", count_chars(carray)); // ⇒ 5
    printf("%zu\n", count_chars(cptr));   // ⇒ 5

    char buf[800] = {'a'};
    printf("%zu\n", sizeof buf);        // ⇒ 800
    printf("%zu\n", count_chars(buf));  // ⇒ 1

    printf("%zu\n", count_chars(buf));  // ⇒ 3
    printf("%s\n", buf);               // ⇒ abb
}
```
– Next: More objects than you can name –