Pointers

EECS 211
Winter 2019
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?
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.
Review: variables, objects, values

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

- Variables name objects, which contain values

a  b

5  10
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
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.

<table>
<thead>
<tr>
<th>Array of chars:</th>
<th>(hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>100 101 102 103 104 105 106 107 108 109 110 111 ...</td>
</tr>
<tr>
<td>... 48 65 6C 6C 6F 20 77 6F 72 6C 64 00 ...</td>
<td></td>
</tr>
</tbody>
</table>
Memory is a huge array, and addresses are indices into it.

**Array of chars:**

<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:**

<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>

*(hexadecimal)*

*(little endian)*

Mixed double and 4 chars:

```
... 100 108 109 110 111 ...
... 6548 6C6C 206F 6F77 6C72 0064 ...
```
Memory is a huge array, and addresses are indices into it.

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

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

<table>
<thead>
<tr>
<th>Array of ints:</th>
<th>(big endian)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>100 104 108 ...</td>
</tr>
<tr>
<td>...</td>
<td>48656C6C 6F20776F 726C6400 ...</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>... 48 65 6C 6C 6F 20 77 6F 72 6C 64 00 ...</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>... 6548 6C6C 206F 6F77 6C72 0064 ...</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>... 48656C6C 6F20776F 726C6400 ...</td>
</tr>
</tbody>
</table>

**Mixed!** **double** and 4 **chars**:

<table>
<thead>
<tr>
<th>... 100</th>
<th>108 109 110 111 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>... 1.56C6C6F20776Fp+135</td>
<td>72 6C 64 00 ...</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);
}
```
$ build/addresses
a: 5
b: 7
c: 9
&a: 0x7ffee536816c
&b: 0x7ffee5368168
&c: 0x7ffee5368164
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 be \texttt{sizeof(int)} on my system.
Pointers

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

```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*
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;
}
```
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;
}
```
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 7 104
```
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 7
```

```
a @100  b @104  ip @108
```
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 7 0
a @100 b @104 ip @108
```
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 ;
```
What’s with the syntax?

```c
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?

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

What does this mean?

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

What does this mean?

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

So you gotta write:

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

What does this mean?

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

So you gotta write:

\[
\begin{align*}
\text{int}\ast \ p; \\
\text{int}\ast \ q; \quad \text{or} \quad \text{int} \ \ast p, \ \ast q;
\end{align*}
\]
Beware!

What does this mean?

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

So you gotta write:

\[ \text{int* p; } \]
\[ \text{int* q; } \text{ or } \text{ int *p, *q; } \ (\text{but please not } \text{int* p,* q;}) \]
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
}
```

x @100  y @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
}
```

x @100  y @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
}
```

x @100    y @104    ip @108

5          6          0
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
}
```

```
x @100 y @104 ip @108
9 6
```
FAQ

Can a struct contain a struct?  Yes.

Can a struct contain an array? Yes, but declaring it looks weird.

Can a struct contain a pointer? Yes.

Can you have an array of structs? Yes.

Can you have an array of arrays? Can you not have an array?

Can you have an array of pointers? Can you not have a pointer to an array?

Can you have a pointer to a struct? Yes.

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

Can you have a pointer to a pointer? Yes.

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

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

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.
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?
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?  

Yes.

Yes, but declaring it looks weird.

Can you not have a pointer to an array?  

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?
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?
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.
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 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];
    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;
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;
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.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;
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;
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!
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

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

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

```plaintext
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;
}

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

(x @100
5

ip @200
100)

(y @104
7

jp @208
104)

(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;
}

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

(x @100 5  (caller’s stack frame)
 y @104 7
 ip @200 100
 jp @208 104
 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;
}
```

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

7

ip @200

100

104

jp @208

ty @104

5

(temp @216

5

(caller’s stack frame)

(swamp’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);
```

(x @100  y @104)

(5  7)  (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: ⟨ptr⟩→⟨field⟩ means (∗⟨ptr⟩).⟨field⟩
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]);
put_int(*a);
```

The arrays decay to pointers when used in contexts that require a pointer type, such as when passing an array to a function that expects a pointer. This demonstrates how arrays can be treated as pointers in C programming.
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]); // ⇒ 0x7ffee5c6e2f0
put_ptr(a); // ⇒ 0x7ffee5c6e2f0
put_int(a[0]); // ⇒ 2
put_int(*a); // ⇒ 2
```
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);          // ⇒ 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);
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]);
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));
```
Arrays *decay* to pointers

```c
typemin
typemin
```

```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

⟨aexpr⟩[⟨iexpr⟩] means *(⟨aexpr⟩ + ⟨iexpr⟩)
Array indexing is pointer arithmetic

\[ \langle aexpr \rangle [\langle iexpr \rangle] \quad \text{means} \quad \star (\langle aexpr \rangle + \langle iexpr \rangle) \]

\&\langle aexpr \rangle [\langle iexpr \rangle] \quad \text{means} \quad \langle aexpr \rangle + \langle iexpr \rangle \]
Strings are arrays of chars

```c
#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

#include <stdio.h>

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

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

Strings are arrays of chars

#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

```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

#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

#include <stdio.h>

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

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

22
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

#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";
}
```
How long is a C string?

```c
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);
    // ⇒ ?
}
```
How long is a C string?

```c
int main()
{
    const char* cpotr = "12345";
    printf("%zu\n", sizeof cpotr);
    // ⇒ 8
}
```
How long is a C string?

```c
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr); // ⇒ 8
    printf("%zu\n", sizeof *cptr); // ⇒ ?
    for (size_t i = 0; i < sizeof cptr; ++i)
        printf("%d \n", (int) cptr[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
}
```
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
}
```
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
#include <stdio.h>

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

    return 0;
}
```
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

}
```
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\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

    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 \n", (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

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

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;
}

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));   // ⇒ ?
}
```
int main()
{
    const char carray[] = "12345",
                         *cptr = "12345";

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

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
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 –