Pointers

CS 211

Winter 2020

Initial code setup

The code in this course is available online. To download a copy of this lecture into your Unix shell account:

```
% cd cs211
% curl $URL211/lec/05pointer.tgz | tar zxvk
⋮
% cd 05pointer
```

Road map

- What's a pointer?
- What can it do?
- What's the point?

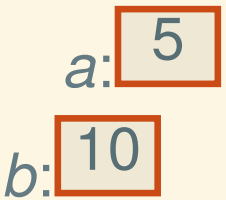
What is a pointer?

Review: variables, objects, values

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

Review: variables, objects, values

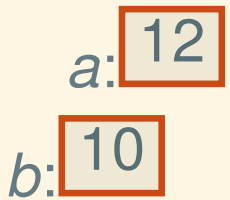
```
int main()
{
    int a = 5, b = 10;
    a = 12;
}
```



- Variables name objects, which contain values

Review: variables, objects, values

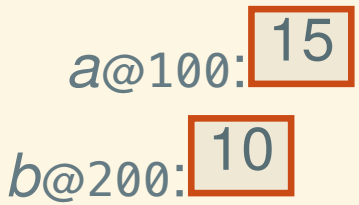
```
int main()
{
    int a = 5, b = 10;
    a = 12;
}
```



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

Review: variables, objects, values

```
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)

Memory is a huge array, and addresses are indices into it.

Array of `chars`: (hexadecimal)

...	100	101	102	103	104	105	106	107	108	109	110	111	...
···	48	65	6C	6C	6F	20	77	6F	72	6C	64	00	···

Array of `shorts`: (little endian)

...	100	102	104	106	108	110	...
···	6548	6C6C	206F	6F77	6C72	0064	···

Memory is a huge array, and addresses are indices into it.

Array of `chars`: (hexadecimal)

...	100	101	102	103	104	105	106	107	108	109	110	111	...
···	48	65	6C	6C	6F	20	77	6F	72	6C	64	00	···

Array of `shorts`: (little endian)

...	100	102	104	106	108	110	...
···	6548	6C6C	206F	6F77	6C72	0064	···

Array of `ints`: (big endian)

...	100	104	108	...
···	48656C6C	6F20776F	726C6400	···

Memory is a huge array, and addresses are indices into it.

Array of `chars`: (hexadecimal)

...	100	101	102	103	104	105	106	107	108	109	110	111	...
···	48	65	6C	6C	6F	20	77	6F	72	6C	64	00	···

Array of `shorts`: (little endian)

...	100	102	104	106	108	110	...
···	6548	6C6C	206F	6F77	6C72	0064	···

Array of `ints`: (big endian)

...	100	104	108	...
···	48656C6C	6F20776F	726C6400	···

Mixed! `double` and 4 `chars`:

...	100	108	109	110	111	...
···	1.56C6C6F20776Fp+135	72	6C	64	00	···

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*`):

```
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
```

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*∗

```
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*∗

```
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*∗

```
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*∗

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

a@100: 5 *b*@104: 7 *ip*@108: 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*∗

```
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*∗

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

Pointers

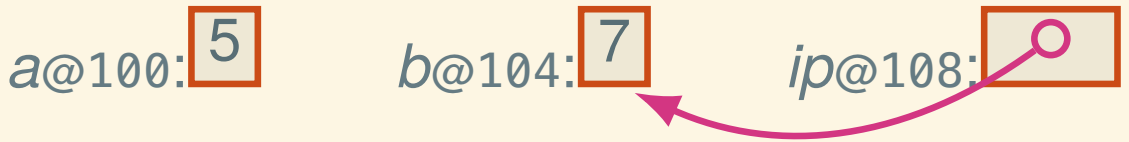
- We can store the address of one object in another object
- A object containing an address is called a *pointer*
- A pointer to an object of any type *T* has type *T*∗

```
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*∗

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



What's with the syntax?

```
int* p;
```

What's with the syntax?

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

What's with the syntax?

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

What's with the syntax?

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

    *
 p  ;
```

What's with the syntax?

```
int* p;
int *p;
int * p;   // don't
int*p;     // don't
    int

    *
 p  ;
```

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?

```
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?

```
int* p, q;
```

Beware!

What does this mean?

```
int* p, q;    ≡    int *p, q;
```

Beware!

What does this mean?

```
int* p, q;    ≡    int *p, q;    ≡    int *p; int q;
```

Beware!

What does this mean?

```
int* p, q;   ≡   int *p, q;   ≡   int *p; int q;
```

So you gotta write:

```
int* p;
int* q;
```

Beware!

What does this mean?

```
int* p, q;   ≡   int *p, q;   ≡   int *p; int q;
```

So you gotta write:

```
int* p;
int* q;   or int *p, *q;
```

Beware!

What does this mean?

```
int* p, q;   ≡   int *p, q;   ≡   int *p; int q;
```

So you gotta write:

```
int* p;
int* q;   or int *p, *q;   (but please not int* p,* q;)
```

What can it do?

What can you do with a pointer?

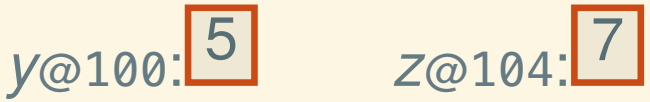
You can dereference (or “follow”) it, using the $*$ operator:

```
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:

```
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:

```
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?

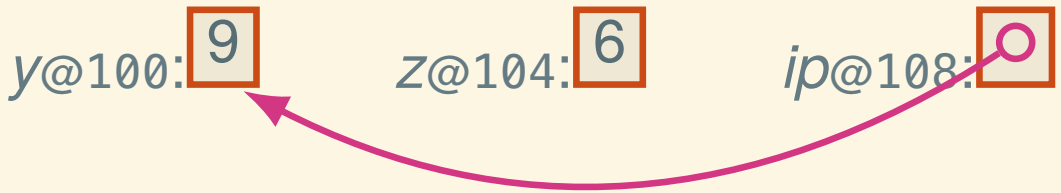
You can dereference (or “follow”) it, using the ∗ operator:

```
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:

```
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
}
```



FAQ

FAQ

Can a struct contain a struct?

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?

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.

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.

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;
```

Everything is compositional

```
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;
}
```

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;
    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
}
```

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;
```

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;
```

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;
```

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);
```

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);
```

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;
```

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;
```

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_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

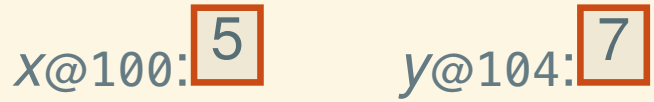
```
void swap(int* ip, int* jp)
{
    int temp = *ip;
    *ip = *jp;
    *jp = temp;
}
```

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

Let's talk about objects

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

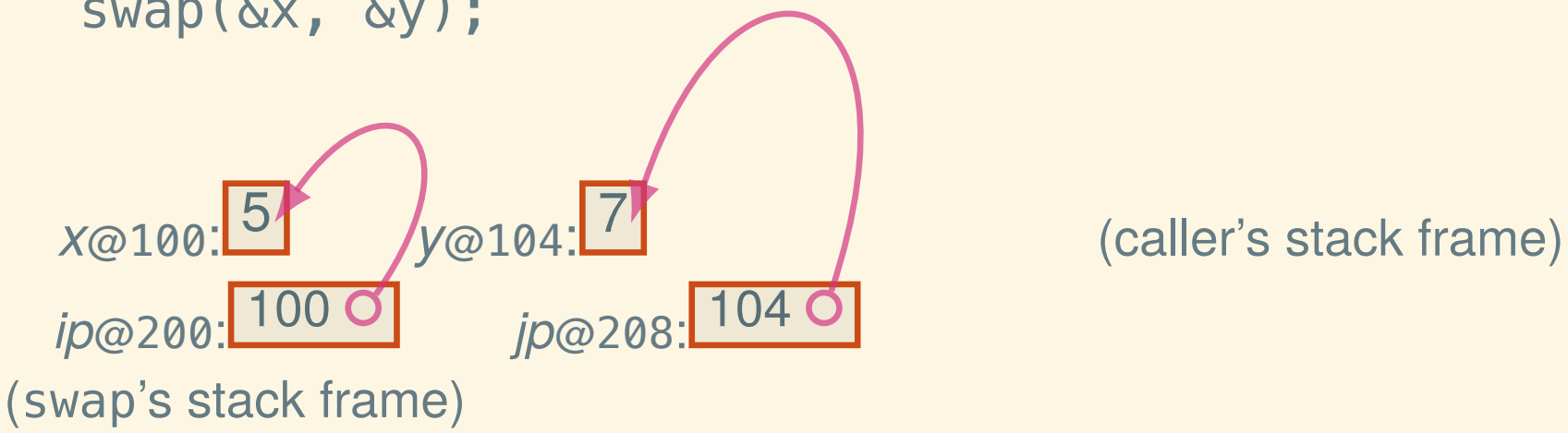
int x = 5, y = 7;
swap(&x, &y);
```



Let's talk about objects

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

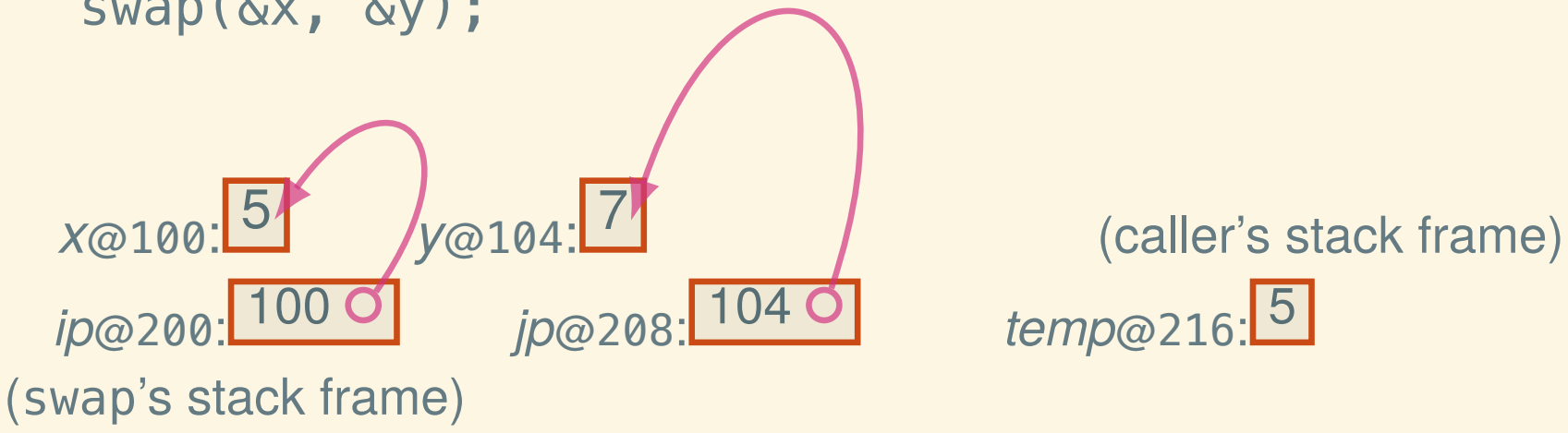
int x = 5, y = 7;
swap(&x, &y);
```



Let's talk about objects

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

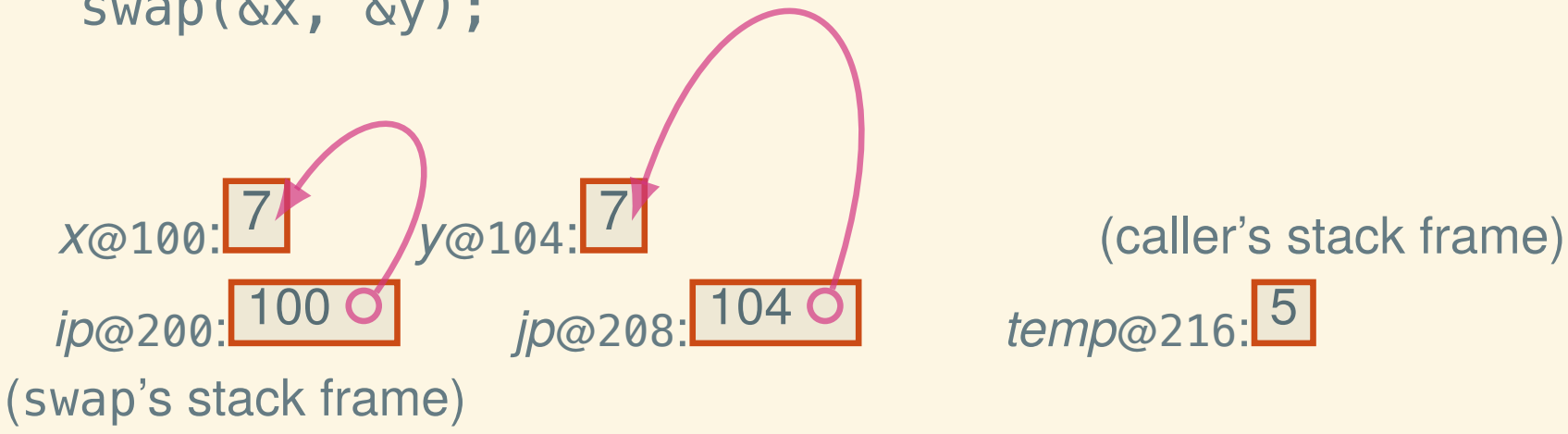
int x = 5, y = 7;
swap(&x, &y);
```



Let's talk about objects

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

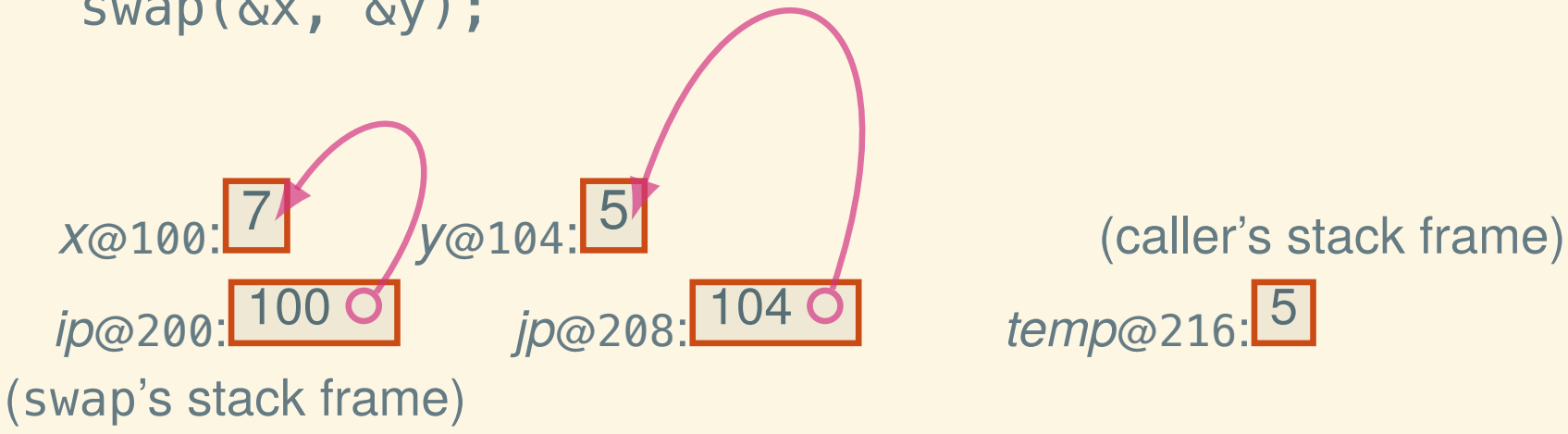
int x = 5, y = 7;
swap(&x, &y);
```



Let's talk about objects

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

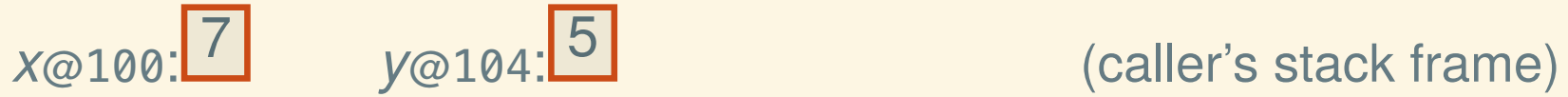
int x = 5, y = 7;
swap(&x, &y);
```



Let's talk about objects

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

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



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).data[(*r).count] = 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: $\langle ptr \rangle$`->`$\langle field \rangle$ *means* `(*`$\langle ptr \rangle$`).`$\langle field \rangle$

Arrays vs. Pointers

Arrays *decay* to pointers

```
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

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

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

Arrays *decay* to pointers

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

$\langle aexpr \rangle$ `[` $\langle iexpr \rangle$ `]` *means* `*(` $\langle aexpr \rangle$ `+` $\langle iexpr \rangle$ `)`

Array indexing is pointer arithmetic

$\langle aexpr \rangle$ [$\langle iexpr \rangle$] *means* $*($ $\langle aexpr \rangle$ + $\langle iexpr \rangle$)

&$\langle aexpr \rangle$ [$\langle iexpr \rangle$] *means* $\langle aexpr \rangle$ + $\langle iexpr \rangle$

Arrays vs. Strings

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

```c
#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

```c
#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

```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?

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









}
```

How long is a C string?

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









}
```

How long is a C string?

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









}
```

How long is a C string?

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









}
```

How long is a C string?

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









}
```

How long is a C string?

```
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?

```
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?

```
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









}
```

How long is a C string?

```
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?

```
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?

```
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

```
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

```
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

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

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









}
```

Counting characters

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

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









}
```

Counting characters

```
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

```
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

```
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

    buf[1] = buf[2] = buf[4] = buf[5] = 'b';
    printf("%zu\n", count_chars(buf));     // ⇒ ?
    printf("%s\n", buf);                   // ⇒ ?
}
```

Counting characters

```
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

    buf[1] = buf[2] = buf[4] = buf[5] = 'b';
    printf("%zu\n", count_chars(buf));     // ⇒ 3
    printf("%s\n", buf);                   // ⇒ abb
}
```

– Next: More objects than you can name –