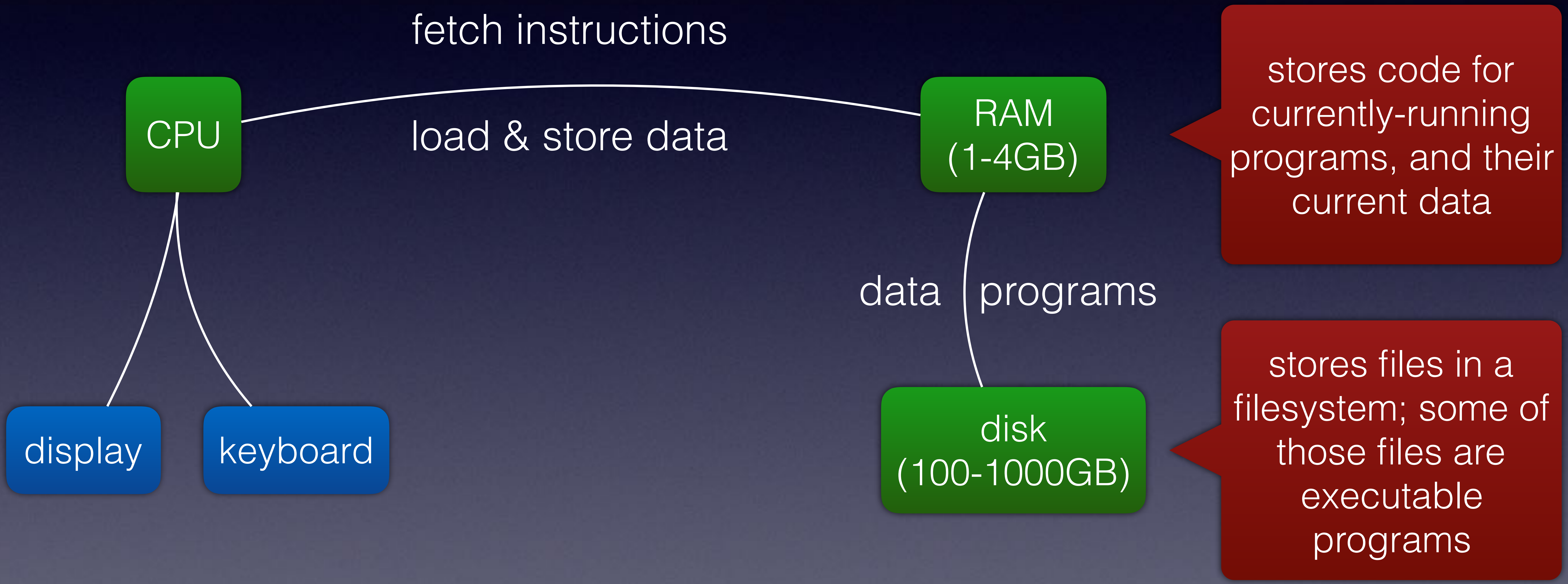
Memory, pointers, and C

David Kotz
Dartmouth College – Computer Science 50
April 2017

Computer architecture



Memory (RAM)

Every *process* (running program) has its own memory;
every byte in memory has a unique numeric *address.*

today, addresses are 64-bit numbers,
so they can refer to 18 x 10^{18} bytes;
that's 18 exabytes!
example: 0x00007FFD7865430C

Memory (RAM)

Every *process* (running program) has its own memory;
every byte in memory has a unique numeric *address.*

today, addresses are 64-bit numbers,
so they can refer to 18 x 10^{18} bytes;
that's 18 exabytes!
example: 0x00007FFD7865430C

0x means hexadecimal

Characters, and pointers

```
int main()
{
  char c = 'x';                   // a character
  char *p = &c;                   // a pointer to a character
  char **pp = &p;                 // a pointer to a pointer to a character

  printf("c  = '%c'\n", c);
  printf("p  = %12p, *p = '%c'\n", p, *p);
  printf("pp = %12p, *pp = %12p, **pp = '%c'\n", pp, *pp, **pp);

  return 0;
}
```

see `pointer0.c`

pointer0.c – output

```
c  = 'x'
p  = 0x7fffcc62d597, *p = 'x'
pp = 0x7fffcc62d588, *pp = 0x7fffcc62d597, **pp = 'x'
```

*notice that *pp == p*
*and that **pp == *p == c*

because that's how they were initialized

Strings in memory

C has no “string” type; a string is an array of characters ending in a null character (‘\0’); we often refer to a string with a pointer to its first character.

see pointer1.c

pointer1.c

```
char *CS = "Computer Science";

int main()
{
  printf(" CS = %12p, *CS = '%c', CS as string = '%s'\n", CS, *CS, CS);

  for (char *p = CS; *p != '\0'; p++) {
    printf(" p = %12p, *p = '%c'\n", p, *p);
  }

  return 0;
}
```

pointer1.c – output

```
CS = 0x0000400630, *CS = 'C', CS as string = 'Computer Science'
p = 0x0000400630, *p = 'C'
p = 0x0000400631, *p = 'o'
p = 0x0000400632, *p = 'm'
p = 0x0000400633, *p = 'p'
p = 0x0000400634, *p = 'u'
p = 0x0000400635, *p = 't'
p = 0x0000400636, *p = 'e'
p = 0x0000400637, *p = 'r'
p = 0x0000400638, *p = ' '
p = 0x0000400639, *p = 'S'
p = 0x000040063a, *p = 'c'
p = 0x000040063b, *p = 'i'
p = 0x000040063c, *p = 'e'
p = 0x000040063d, *p = 'n'
p = 0x000040063e, *p = 'c'
p = 0x000040063f, *p = 'e'
```

pointer1.c – output

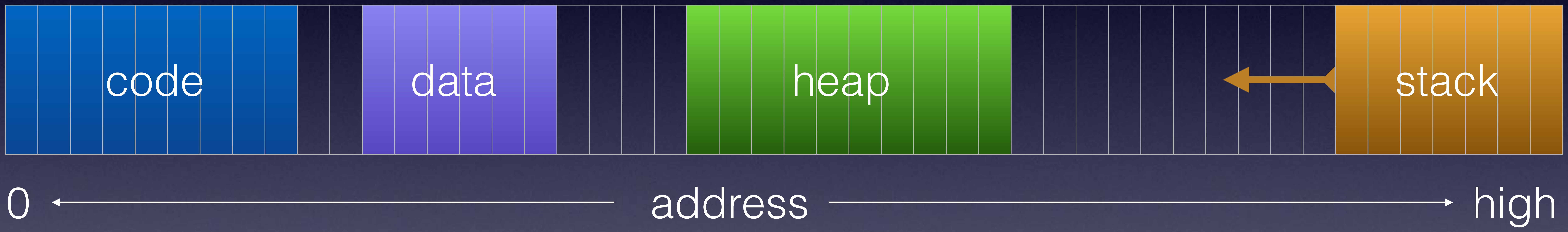
```
CS = 0x0000400630, *CS = 'C', CS as string = 'Computer Science'
p = 0x0000400630, *p = 'C'
p = 0x0000400631, *p = 'o'
p = 0x0000400632, *p = 'm'
p = 0x0000400633, *p = 'p'
p = 0x0000400634, *p = 'u'
p = 0x0000400635, *p = 't'
p = 0x0000400636, *p = 'e'
p = 0x0000400637, *p = 'r'
p = 0x0000400638, *p = ' '
p = 0x0000400639, *p = 'S'
p = 0x000040063a, *p = 'c'
p = 0x000040063b, *p = 'i'
p = 0x000040063c, *p = 'e'
p = 0x000040063d, *p = 'n'
p = 0x000040063e, *p = 'c'
p = 0x000040063f, *p = 'e'
```

notice:

- both **p** and **CS** are pointers
- **p** initially has the same value as **CS**
- (i.e., points to the same address)
- incrementing **p** steps to the next char
- since sizeof(char)=1, address increments by 1
- **p** is an address, ***p** is a character
- printf can print a pointer with **%p**, or interpret that pointer as address of a string with **%s**

Code, data, heap, and stack

Process memory includes compiled **code** (machine instructions), **data** (global variables), **heap** (dynamically allocated), and **stack** (local variables)



Not all addresses will be used; and different compilers and operating systems may lay out the four segments differently

Code memory

All your C code is *compiled* into machine instructions, *linked* with libraries, and laid out within the **code** segment

See example: pointer2.c

Again, the code segment is not necessarily in low memory.

Data memory

All your global variables are laid out in the **data** segment

See example: pointer2.c

Again, the data segment is not necessarily in low memory.

pointer2.c (part 1)

```
const int fifteen = 15;
int main()
{
  // local variables are on the stack
  int x = 2, y = 5;

  // global variables; note they are in low memory addresses
  printf("globals\n");
  printf(" fifteen @ %12p has value %d\n", &fifteen, fifteen);

  // main() is a function, and its code is at an address too!
  printf("main @ %12p\n", main);

  // local variables are on the stack
  printf(" x @ %12p has value %d\n", &x, x);
  printf(" y @ %12p has value %d\n", &y, y);
...
```

Stack memory

Functions' local variables live in the **stack** segment, aka, "on the stack" along with a record of the function-call sequence

The stack starts in high(er) memory addresses, and grows "down" toward lower addresses as function calls are nested

See pointer2.c

pointer2.c – output (part 1)

```
globals – data segment
 fifteen @ 0x0000400750 has value 15
main @ 0x00004005f6 – code segment
 x @ 0x7ffdbf396d3c has value 2 – stack segment
 y @ 0x7ffdbf396d38 has value 5 – stack segment
```

notice that all variables, and functions, have an address;
the name of a function is actually a pointer to that function.
stack variables are in high memory.

pointer2.c (part 2)

```
main...
  // pass x by reference, y by value
  change(&x,y);

  // see whether those changed
  printf("main @ %12p\n", main);
  printf(" x @ %12p has value %d\n", &x, x);
  printf(" y @ %12p has value %d\n", &y, y);
...
}

void change(int *a, int b)
{
  // as above, change() is a function,
  // and its parameters and local variables are on the stack
  printf("change @ %12p\n", change);
  printf(" a @ %12p has value %d at %12p\n", &a, *a, a);
  printf(" b @ %12p has value %d\n", &b, b);
  // attempt to change the values
  *a = 99;
  b = 99;
}
```

Stack memory

Functions' local variables live in the **stack** segment, aka, "on the stack" along with a record of the function-call sequence

The stack starts in high(er) memory addresses, and grows "down" toward lower addresses as function calls are nested

See pointer2.c

Stack memory

Functions' local variables live in the **stack** segment, aka, "on the stack" along with a record of the function-call sequence

Stack memory

Functions' local variables live in the **stack** segment, aka, "on the stack" along with a record of the function-call sequence

pointer2.c – output (part 2)

```
main @ 0x0000400566 – code segment
 x @ 0x7ffdbf396d3c has value 2
 y @ 0x7ffdbf396d38 has value 5
change @ 0x0000400645 – code segment
 a @ 0x7ffdbf396d18 has value 2 at 0x7ffdbf396d3c
 b @ 0x7ffdbf396d14 has value 5
main @ 0x0000400566
 x @ 0x7ffdbf396d3c has value 99
 y @ 0x7ffdbf396d38 has value 5
```

*notice that the addresses of **a** and **b** are not the same as **x** and **y**.*
*notice that **a** receives the value of **&x**, and thus points to the same address.*
*when **change()** assigns to ***x**, the value of **x** changes – not so for **y**.*

Heap

The **heap** is for dynamically-allocated memory – when you don't know in advance how much space you'll need, or so you can build complex data structures.

C has no language feature like Java's `new` and `delete`.
Instead, a library provides `malloc()` and `free()` functions;
that library manages used and free space within the heap.

pointer3.c

```
int main()
{
  char *hello = "hello world!";
  char buf[15];

  strcpy(buf, "something"); // initialize buf

  // local variables are on the stack
  printf(" hello @ %12p has value '%s', which resides at %12p\n", &hello, hello, hello);
  printf(" buf @ %12p has value '%s', which resides at %12p\n", &buf, buf, buf);

  // malloc allocates space on the heap
  hello = (char *)malloc(10);
  strcpy(hello, "new stuff");
  printf(" now hello @ %12p has value '%s', which resides at %12p\n", &hello, hello, hello);
  // free lets the heap re-use that space
  free(hello);
  printf(" note hello @ %12p still points to %12p\n", &hello, hello);
...
```

Heap

The **heap** is for dynamically-allocated memory – when you don't know in advance how much space you'll need, or so you can build complex data structures.

The heap manager allocates 10 bytes and returns the address, which we save in the pointer variable `hello`.

Heap

The **heap** is for dynamically-allocated memory – when you don't know in advance how much space you'll need, or so you can build complex data structures.

The heap manager allocates 10 bytes and returns the address, which we save in the pointer variable `hello`.

pointer3.c – output

```
hello @ 0x7ffc79c14658 has value 'hello world!', which resides at 0x0000400720
```

notice it is a different address! because it was initialized to point to a constant string.

```
buf @ 0x7ffc79c14640 has value 'something', which resides at 0x7ffc79c14640
```

notice it is the same address! because buf is a character array on the stack.

```
now hello @ 0x7ffc79c14658 has value 'new stuff', which resides at 0x0002278420
```

notice it now points to a new address – inside the heap – provided by malloc()

```
note hello @ 0x7ffc79c14658 still points to 0x0002278420
```

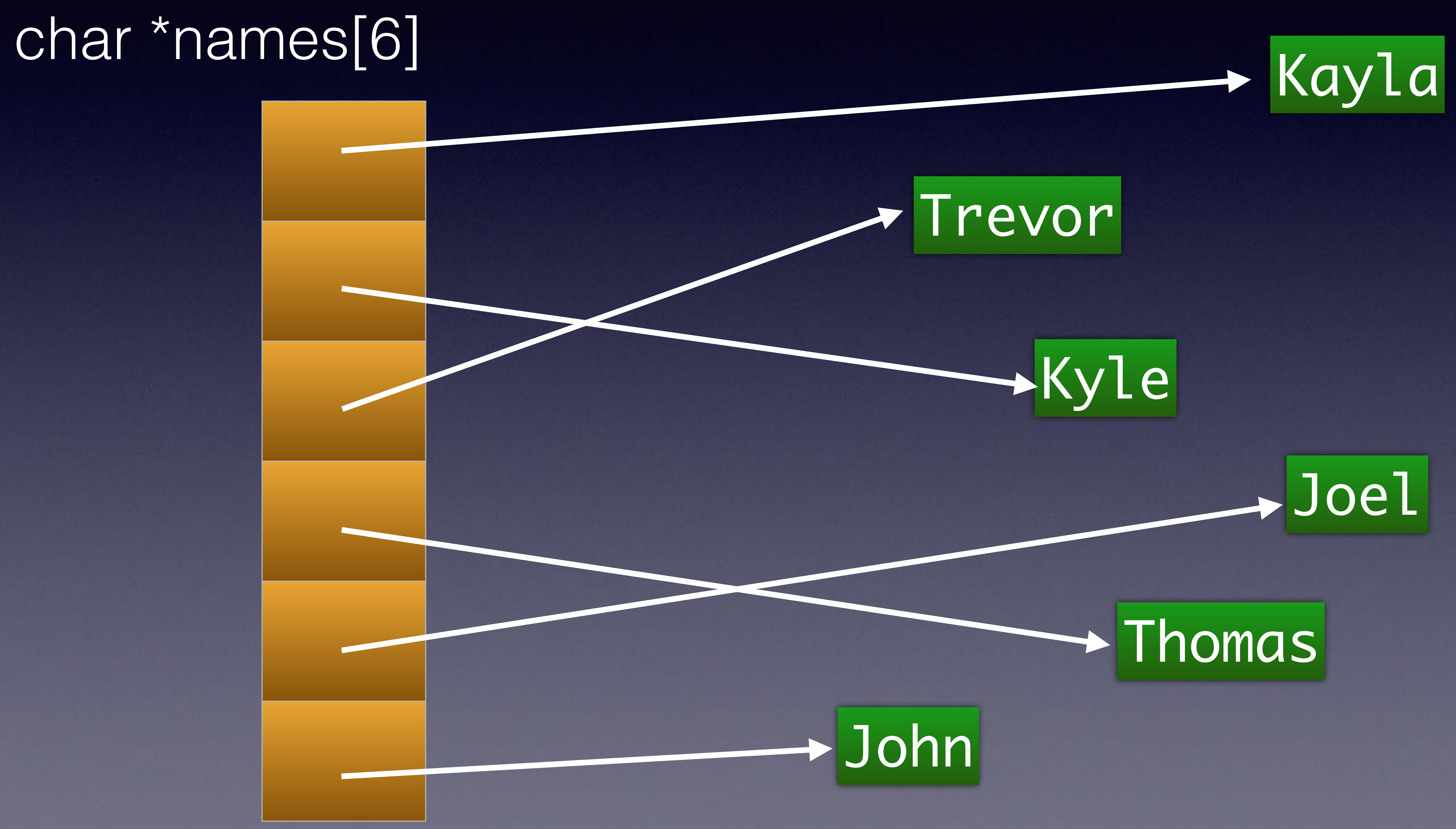
calling free(hello) did not change the pointer, but we should never use that pointer value!

Heap

The **heap** is for dynamically-allocated memory – when you don't know in advance how much space you'll need, or so you can build complex data structures.

After `free(hello)` the heap manager thinks the space is now unallocated and can be used to support future `malloc()` calls. If we keep and re-use the `hello` pointer, bad stuff happens!

Array of strings



see names3.c