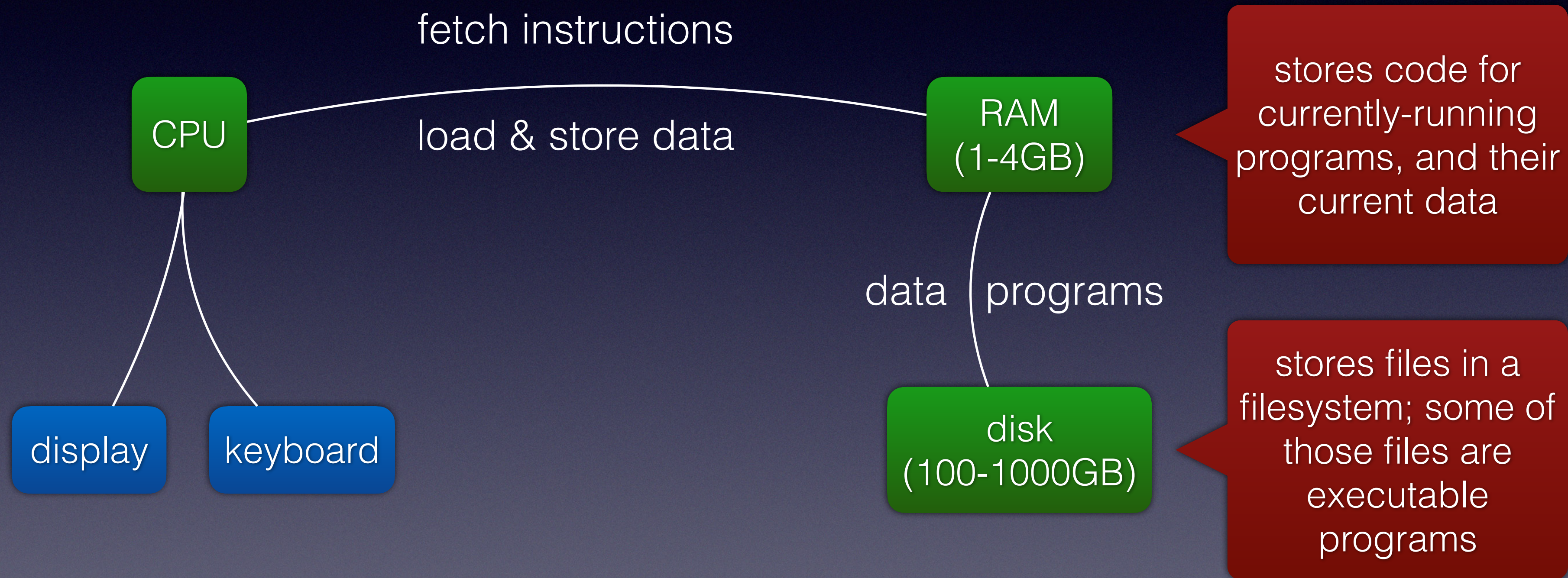# Memory, pointers, and C

David Kotz & Charles Palmer
Dartmouth College – Computer Science 50
Fall 2024

Computer architecture

CPU

display   keyboard

fetch instructions

load & store data

RAM
(1-4GB)

data   programs

disk
(100-1000GB)

stores code for currently-running programs, and their current data

stores files in a filesystem; some of those files are executable programs

Memory (RAM)

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

0 ←——————————————— address ——————————————→ high

today, addresses are 64-bit numbers,
so they can refer to 18×10^{18} bytes;
that's 18 exabytes!
example: `0x7ffc56f45610`

0x means hexadecimal

Pointers?

- A pointer is simply a symbolic representation of an address.

- When the & operator is used to determine the address of a variable **fred**, then the expression **&fred** is a "pointer to **fred**".

- C provides pointer variables that can hold an address just like a **int** variable can hold an integer.

```
int fred=42;
int* fptr;
…
fptr = &fred; /* assigns fred's address to fptr */
```

- Now, fptr is said to *point to fred*.

- The difference between fptr and &fred if that the former is a variable and the latter is a constant.

- A single pointer variable can be set and reset to any address

characters and pointers : pointer0.c

```c
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;
}
```
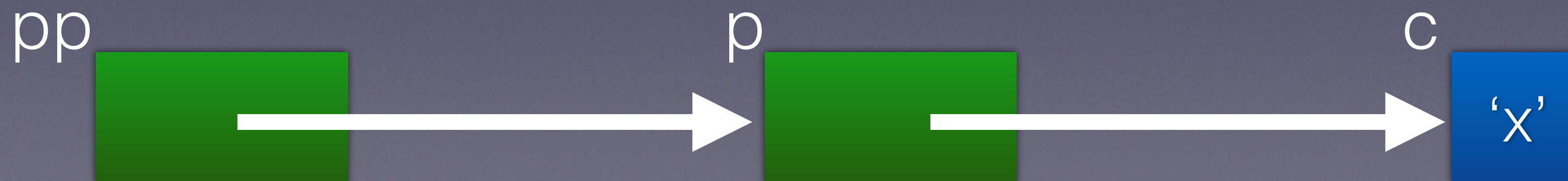
see pointer0.c

pointer0.c – output

```
c  = 'x'
p  = 0x7fff1e99e0e7, *p = 'x'
pp = 0x7fff1e99e0e8, *pp = 0x7fff1e99e0e7, **pp = 'x'
```

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

because that's how they were initialized

pp   p   c

'x'

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.

```
Computer Scienceø
```

0 ←——————————— address ——————————→ high

CS

see pointer1.c

pointer1.c

```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 = 0x55b4dc014768, *CS = 'C', CS as string = 'Computer Science'
p = 0x55b4dc014768, *p = 'C'
p = 0x55b4dc014769, *p = 'o'
p = 0x55b4dc01476a, *p = 'm'
p = 0x55b4dc01476b, *p = 'p'
p = 0x55b4dc01476c, *p = 'u'
p = 0x55b4dc01476d, *p = 't'
p = 0x55b4dc01476e, *p = 'e'
p = 0x55b4dc01476f, *p = 'r'
p = 0x55b4dc014770, *p = ' '
p = 0x55b4dc014771, *p = 'S'
p = 0x55b4dc014772, *p = 'c'
p = 0x55b4dc014773, *p = 'i'
p = 0x55b4dc014774, *p = 'e'
p = 0x55b4dc014775, *p = 'n'
p = 0x55b4dc014776, *p = 'c'
p = 0x55b4dc014777, *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**

These addresses are in the data segment

Code, data, heap, and stack

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

| code | data | heap | ← | stack |

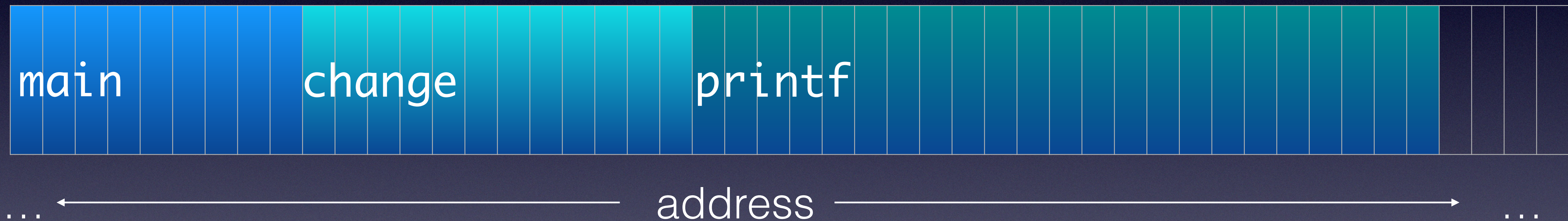0 ←——————————————————— address ———————————————→ high
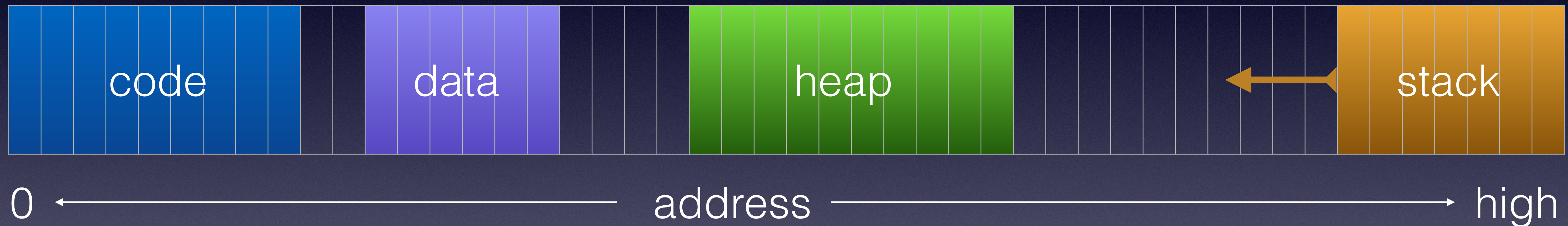
Not all addresses will be used; and different compilers and operating systems may lay out the four segments differently

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

fifteen…

… ⟵———————— address ———————— …

See example: `pointer2.c`
Again, the data segment is not necessarily in low memory.

pointer2.c (part 1)

```c
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

... ← address → high

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
 fifteen @ 0x561416c31918 has value 15 – data segment
main @ 0x561416c316fa – code segment
 x @ 0x7ffd44047790 has value 2 – stack segment
 y @ 0x7ffd44047794 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);
...
}
```
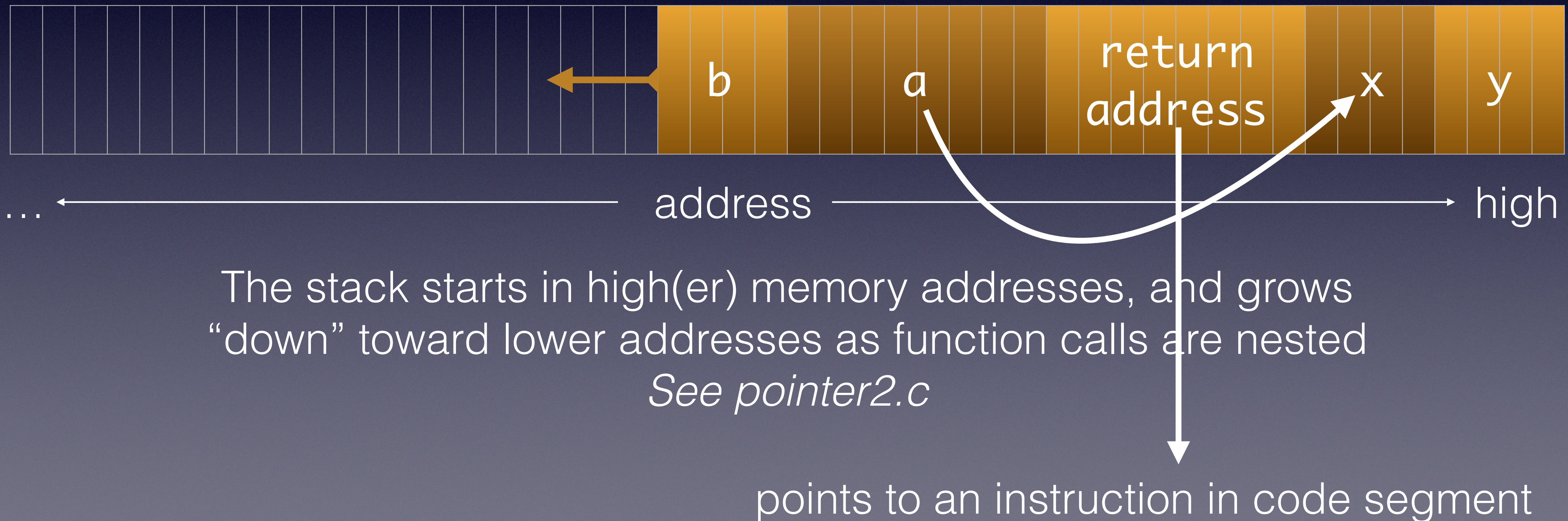
```
void change(int *a, int b)
{
  // as above, change() is a function,
  // and its parameters and local variables are on the stack
  printf("change @ %12p\n", change);
  printf(" a @ %12p has value %d at %12p\n", &a, *a, a);
  printf(" b @ %12p has value %d\n", &b, b);
  // attempt to change the values
  *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

b   a   return address   x   y

...   address   high

The stack starts in high(er) memory addresses, and grows
"down" toward lower addresses as function calls are nested
See pointer2.c

points to an instruction in code segment

pointer2.c – output (part 2)

```
main @ 0x561416c316fa – code segment
 x @ 0x7ffd44047790 has value 2 – stack segment
 y @ 0x7ffd44047794 has value 5 – stack segment
change @ 0x561416c31812 – code segment
 a @ 0x7ffd44047778 has value 2 at 0x7ffd44047790 – stack segment
 b @ 0x7ffd44047774 has value 5 – stack segment
main @ 0x561416c316fa
 x @ 0x7ffd44047790 has value 99
 y @ 0x7ffd44047794 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.

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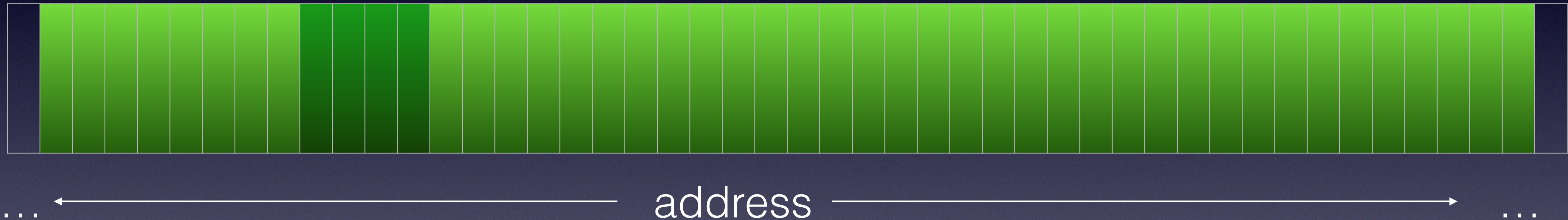
C has no language feature like Java's `new` and `delete`.
Instead, the C Library provides `malloc()` and `free()` functions.
That library manages used and free space within the heap.

pointer3.c

```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);
...
```
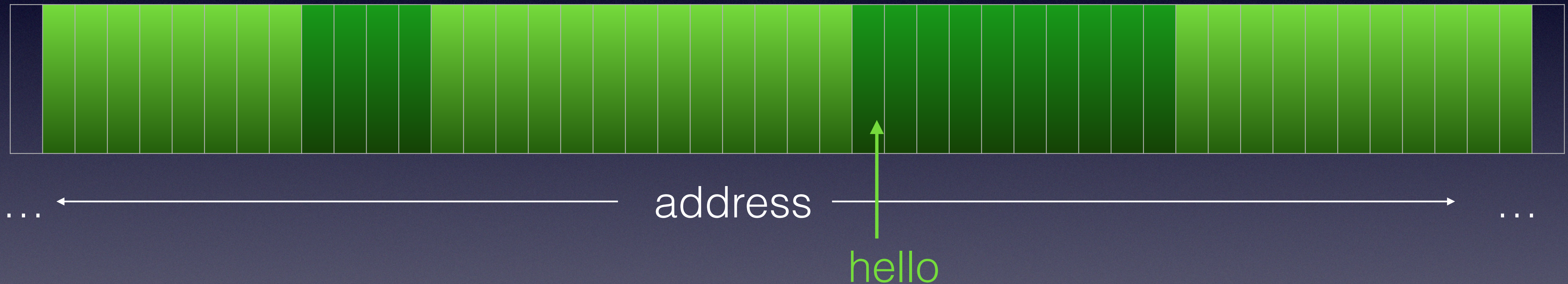
Heap

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.

address

...   ...

hello

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

pointer3.c – output

```
hello @ 0x7ffd8c452530 has value 'hello world!', which resides at 0x55a0a40ff8c8
```
notice the different address! because `hello` *was initialized to point to a constant in the data segment.*

```
buf @ 0x7ffd8c452539 has value 'something', which resides at 0x7ffd8c452539
```
notice it is the same address! because `buf` *is a character array on the stack.*

```
now hello @ 0x7ffd8c452530 has value 'new stuff', which resides at 0x55a0a5389670
```
notice `hello` *now points to a new address – inside the heap – provided by* `malloc()`

```
note hello @ 0x7ffd8c452530 still points to 0x55a0a5389670
```
calling `free(hello)` *did not change the pointer, but we should never use that pointer again!*

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.

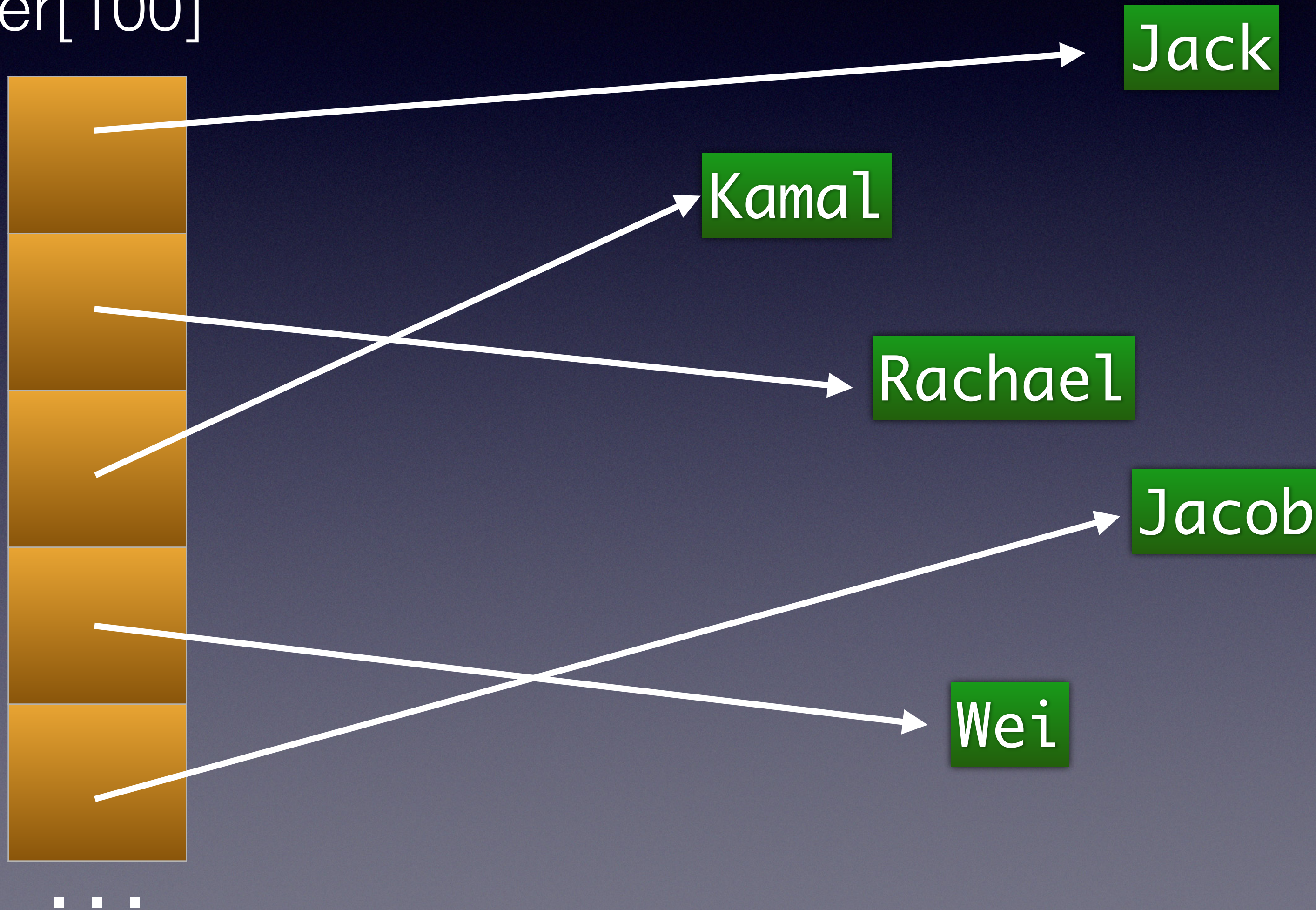... ←————————————— address ————— ...
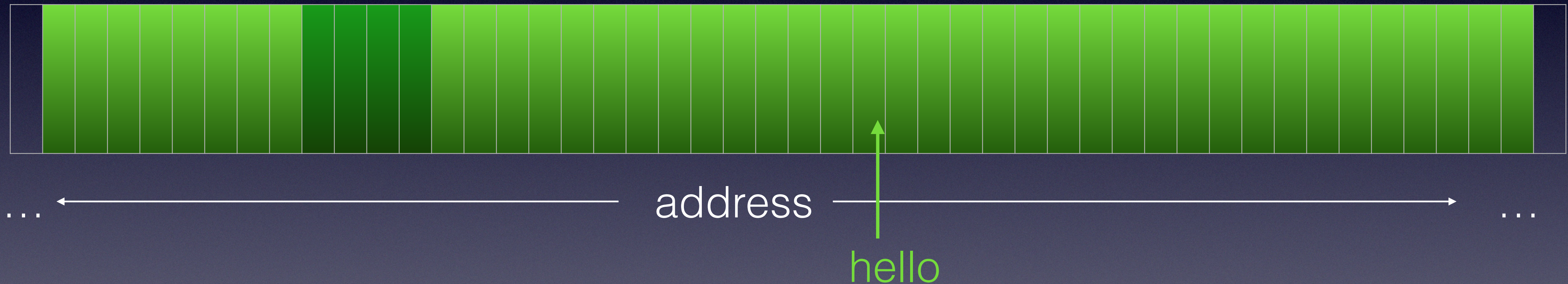
hello

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

char *sorter[100]

Jack

Kamal

Rachael

Jacob

Wei

see sorter2.c

. . .