Pointers

Pointers

- Pointers provide an alternative way to pass data between functions.
  - Recall that up to this point, we have passed all data **by value**, with one exception.
  - When we pass data by value, we only pass a copy of that data.

- If we use pointers instead, we have the power to pass the actual variable itself.
  - That means that a change that is made in one function can impact what happens in a different function.
  - Previously, this wasn't possible!

Pointers

- Before we dive into what pointers are and how to work with them, it’s worth going back to basics and have a look at our computer’s memory.

Pointers

- Every file on your computer lives on your disk drive, be it a hard disk drive (HDD) or a solid-state drive (SSD).

- Disk drives are just storage space; we can’t directly work there. Manipulation and use of data can only take place in RAM, so we have to move data there.

- Memory is basically a huge array of 8-bit wide bytes.
  - 512 MB, 1GB, 2GB, 4GB…

Pointers

Data Type	Size (in bytes)
`int`	4

Pointers

Data Type	Size (in bytes)
`int`	4
`char`	1

Pointers

Data Type	Size (in bytes)
`int`	4
`char`	1
`float`	4

Pointers

Data Type	Size (in bytes)
`int`	4
`char`	1
`float`	4
`double`	8

Pointers

Data Type	Size (in bytes)
`int`	4
`char`	1
`float`	4
`double`	8
`long long`	8

Pointers

Data Type	Size (in bytes)
`int`	4
`char`	1
`float`	4
`double`	8
`long long`	8
`string`	???

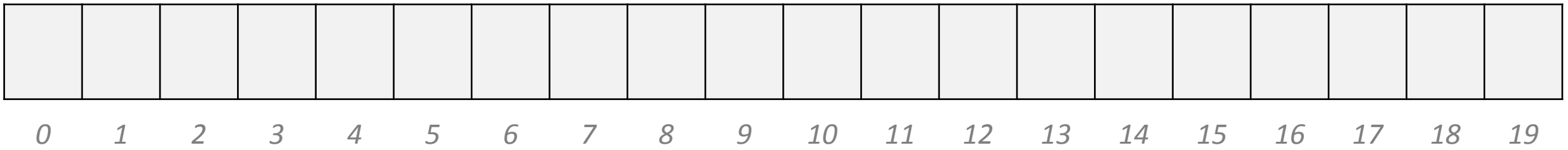
Pointers

- Back to this idea of memory as a big array of byte-sized cells.

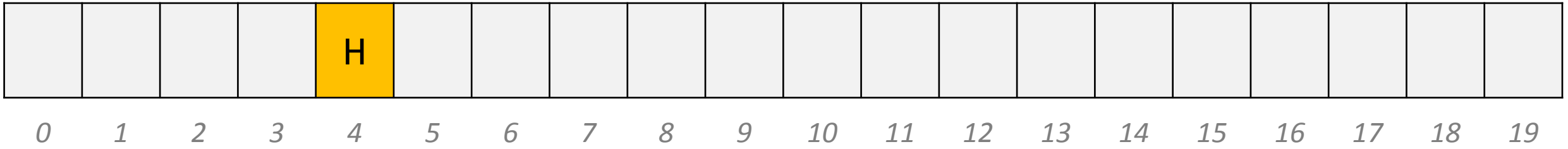
- Recall from our discussion of arrays that they not only are useful for storage of information but also for so-called **random access**.
  - We can access individual elements of the array by indicating which index location we want.

- Similarly, each location in memory has an **address**.

Pointers

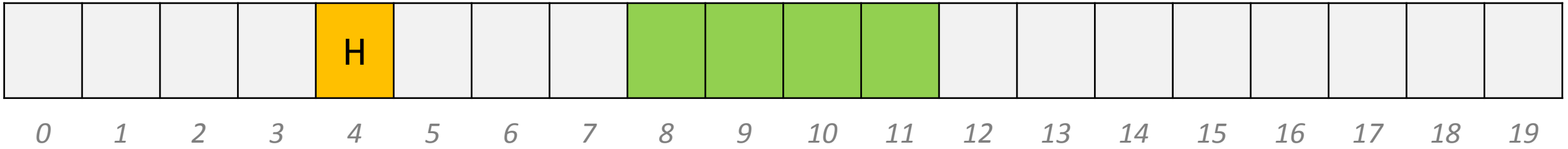


Pointers



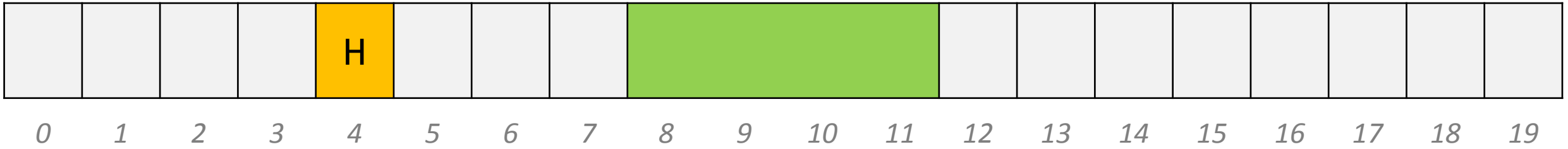

```
char c = ‘H’;
```

Pointers



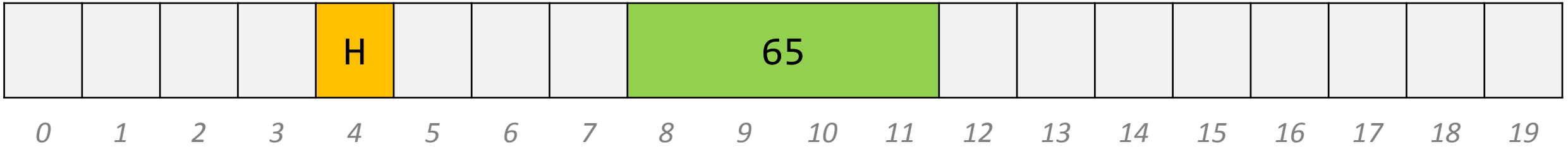

```
char c = ‘H’;
int speedlimit = 65;
```

Pointers



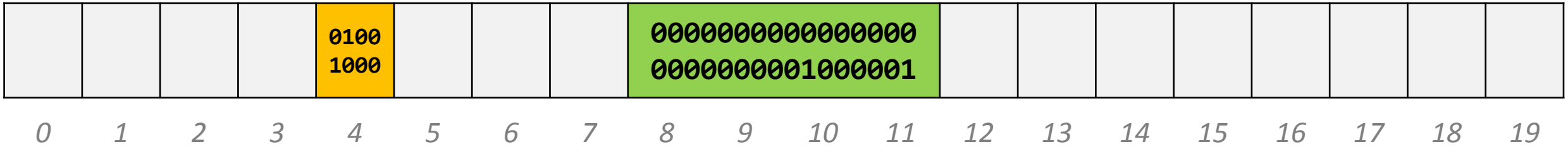

```
char c = ‘H’;
int speedlimit = 65;
```

Pointers



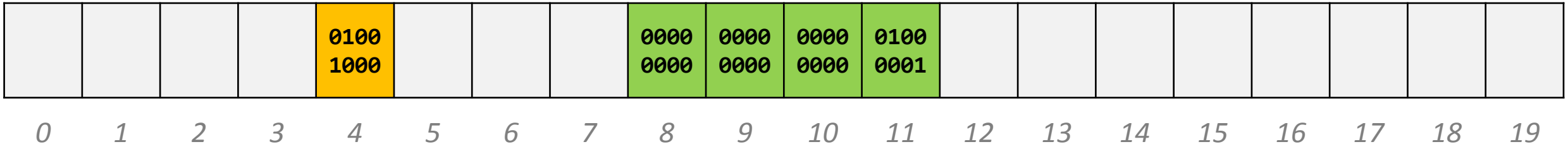

```
char c = ‘H’;
int speedlimit = 65;
```

Pointers



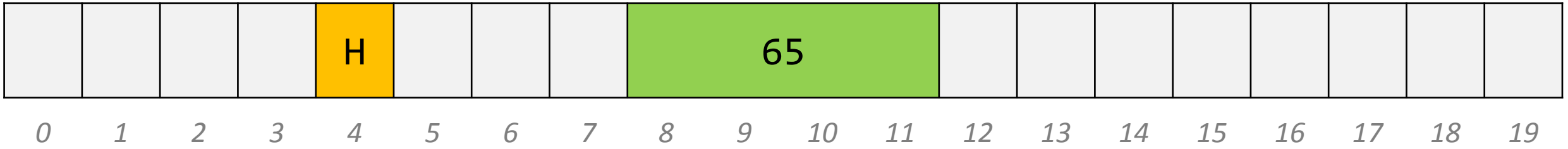

```
char c = ‘H’;
int speedlimit = 65;
```

Pointers



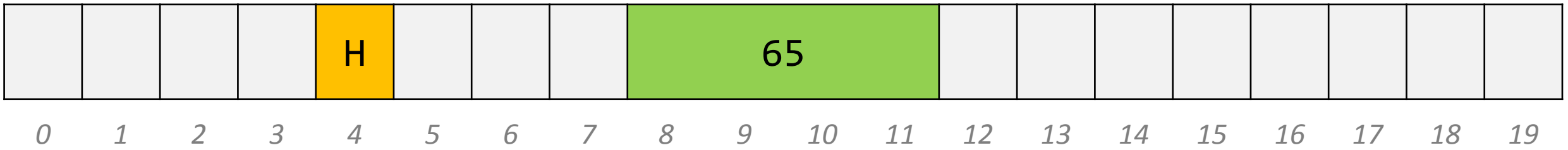

```
char c = ‘H’;
int speedlimit = 65;
```

Pointers



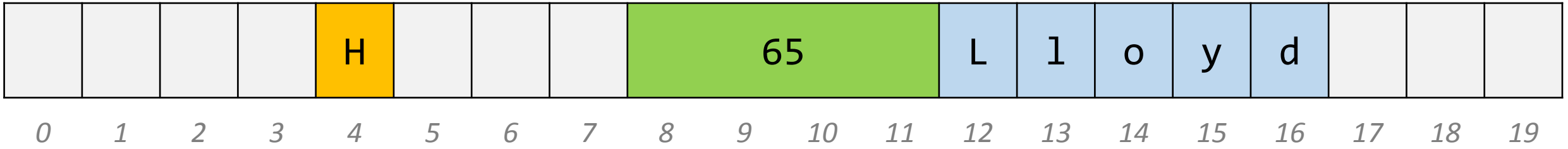

```
char c = ‘H’;
int speedlimit = 65;
```

Pointers



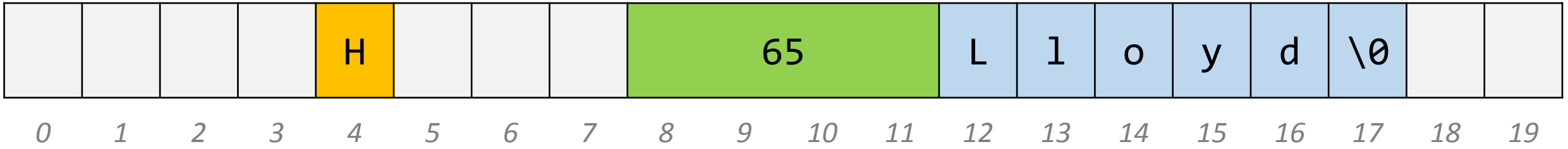

```
char c = ‘H’;
int speedlimit = 65;
string surname = “Lloyd”;
```

Pointers



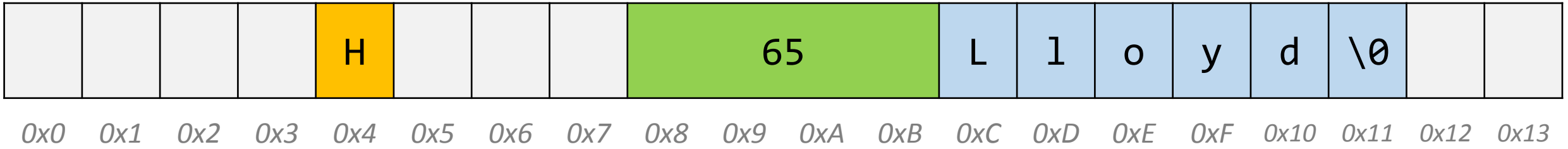

```
char c = ‘H’;
int speedlimit = 65;
string surname = “Lloyd”;
```

Pointers




```
char c = ‘H’;
int speedlimit = 65;
string surname = “Lloyd”;
```

Pointers




```
char c = ‘H’;
int speedlimit = 65;
string surname = “Lloyd”;
```

Pointers

- There’s only one critical thing to remember as we start working with pointers:

POINTERS ARE JUST ADDRESSES

Pointers

- As we start to work with pointers, just keep this image in mind:

Pointers

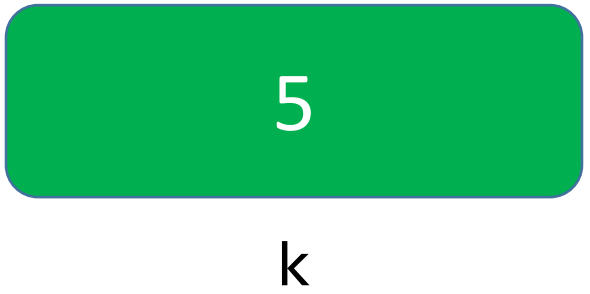
- As we start to work with pointers, just keep this image in mind:

k

```
int k;
```

Pointers

- As we start to work with pointers, just keep this image in mind:




```
int k;
k = 5;
```

Pointers

- As we start to work with pointers, just keep this image in mind:


```
int k;
k = 5;
int* pk;
```

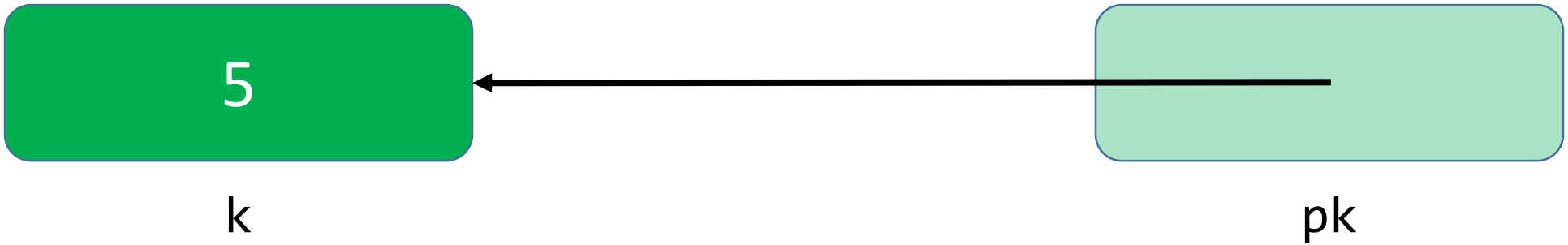
Pointers

- As we start to work with pointers, just keep this image in mind:


```
int k;
k = 5;
int* pk;
pk = &k;
```

Pointers

- As we start to work with pointers, just keep this image in mind:




```
int k;
k = 5;
int* pk;
pk = &k;
```

Pointers

- A **pointer**, then, is a data item whose
  - *value* is a memory address
  - *type* describes the data located at that memory address

- As such, pointers allow data structures and/or variables to be shared among functions.

- Pointers make a computer environment more like the real world.

Pointers

- The simplest pointer available to us in C is the NULL pointer.
  - As you might expect, this pointer points to nothing (a fact which can actually come in handy!)

- When you create a pointer and you don't set its value immediately, you should **always** set the value of the pointer to NULL.

- You can check whether a pointer is NULL using the equality operator (==).

Pointers

- Another easy way to create a pointer is to simply **extract** the address of an already existing variable. We can do this with the address extraction operator (`&`).

- If `x` is an `int`-type variable, then `&x` is a pointer-to-`int` whose value is the address of `x`.
- If `arr` is an array of `doubles`, then `&arr[i]` is a pointer-to-`double` who value is the address of the `i`th element of `arr`.
  - An array's name, then, is actually just a pointer to its first element – you've been working with pointers all along!

Pointers

- The main purpose of a pointer is to allow us to modify or inspect the location to which it points.
  - We do this by **dereferencing** the pointer.

- If we have a pointer-to-`char` called `pc`, then `*pc` is the data that lives at the memory address stored inside the variable `pc`.

Pointers

- Used in this context, * is known as the **dereference operator**.
- It “goes to the reference” and accesses the data at that memory location, allowing you to manipulate it at will.
- This is just like visiting your neighbor. Having their address isn’t enough. You need to go to the address and only then can you interact with them.

Pointers

- Can you guess what might happen if we try to dereference a pointer whose value is NULL?

Pointers

- Can you guess what might happen if we try to dereference a pointer whose value is NULL?

Segmentation fault

Pointers

- Can you guess what might happen if we try to dereference a pointer whose value is NULL?

Segmentation fault

- Surprisingly, this is actually good behavior! It defends against accidental dangerous manipulation of unknown pointers.
  - That's why we recommend you set your pointers to NULL immediately if you aren't setting them to a known, desired value.

Pointers

```
int* p;
```

- The value of p is an address.
- We can dereference p with the * operator.
- If we do, what we'll find at that location is an `int`.

Pointers

- One more annoying thing with those *s. They're an important part of both the type name **and** the variable name.
  - Best illustrated with an example.

```
int* px, py, pz;
```

Pointers

- One more annoying thing with those *s. They're an important part of both the type name **and** the variable name.
  - Best illustrated with an example.

~~`int* px, py, pz;`~~

```
int* pa, *pb, *pc;
```

Pointers

Data Type	Size (in bytes)
`int`	4
`char`	1
`float`	4
`double`	8
`long long`	8
`string`	???

Pointers

Data Type	Size (in bytes)
`int`	4
`char`	1
`float`	4
`double`	8
`long long`	8
`char*`	???

Pointers

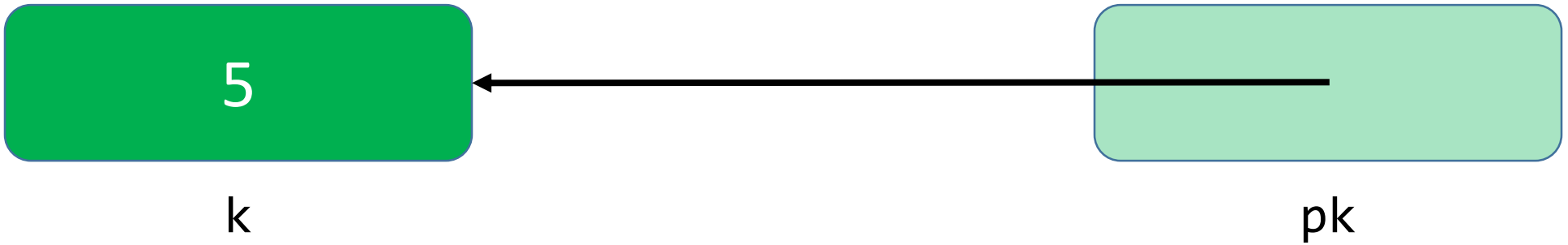
Data Type	Size (in bytes)
`int`	4
`char`	1
`float`	4
`double`	8
`long long`	8
`char*`	4 or 8

Pointers

Data Type	Size (in bytes)
`int`	4
`char`	1
`float`	4
`double`	8
`long long`	8
`char*, int*, float*, double*, _____*`	4 or 8

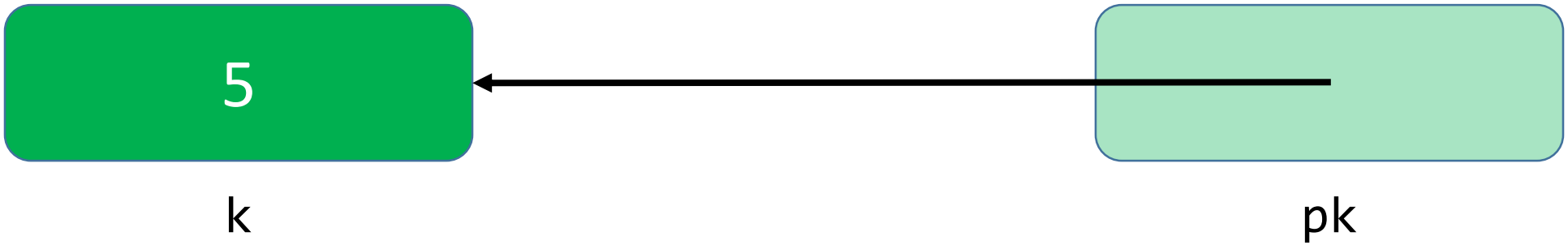
Pointers

- So what happens?



Pointers

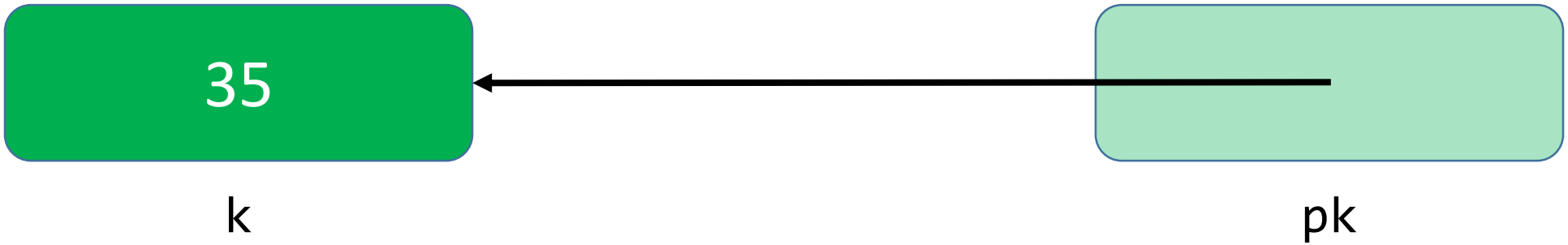
- So what happens?




```
*pk = 35;
```

Pointers

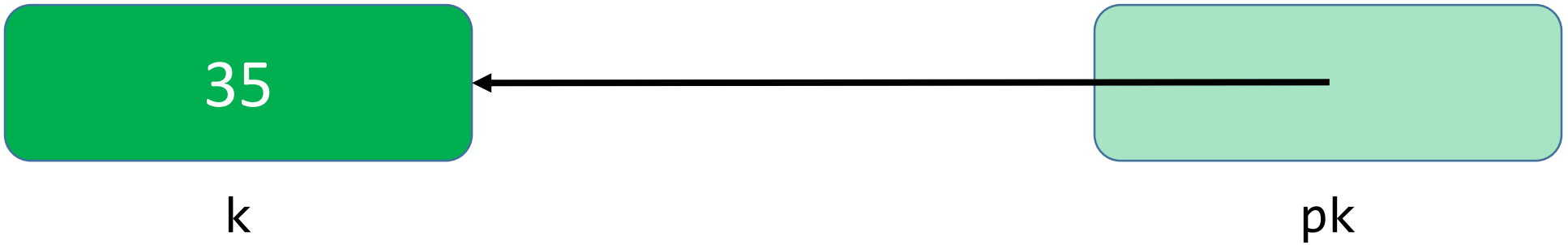
- So what happens?




```
*pk = 35;
```

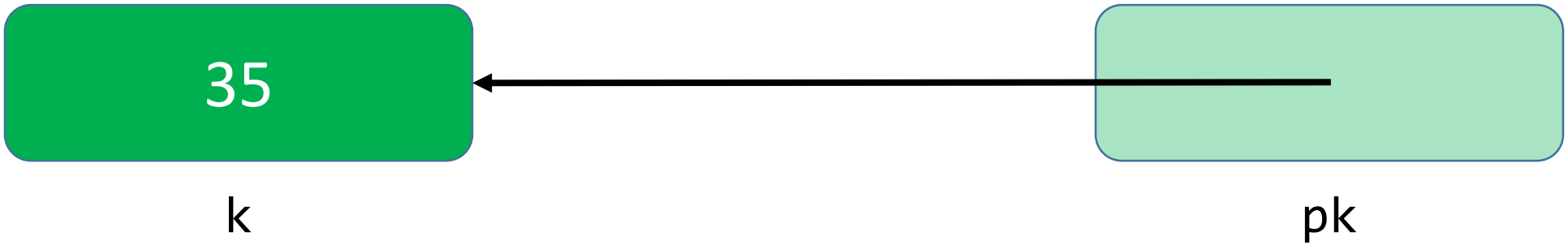
Pointers

- So what happens?



Pointers

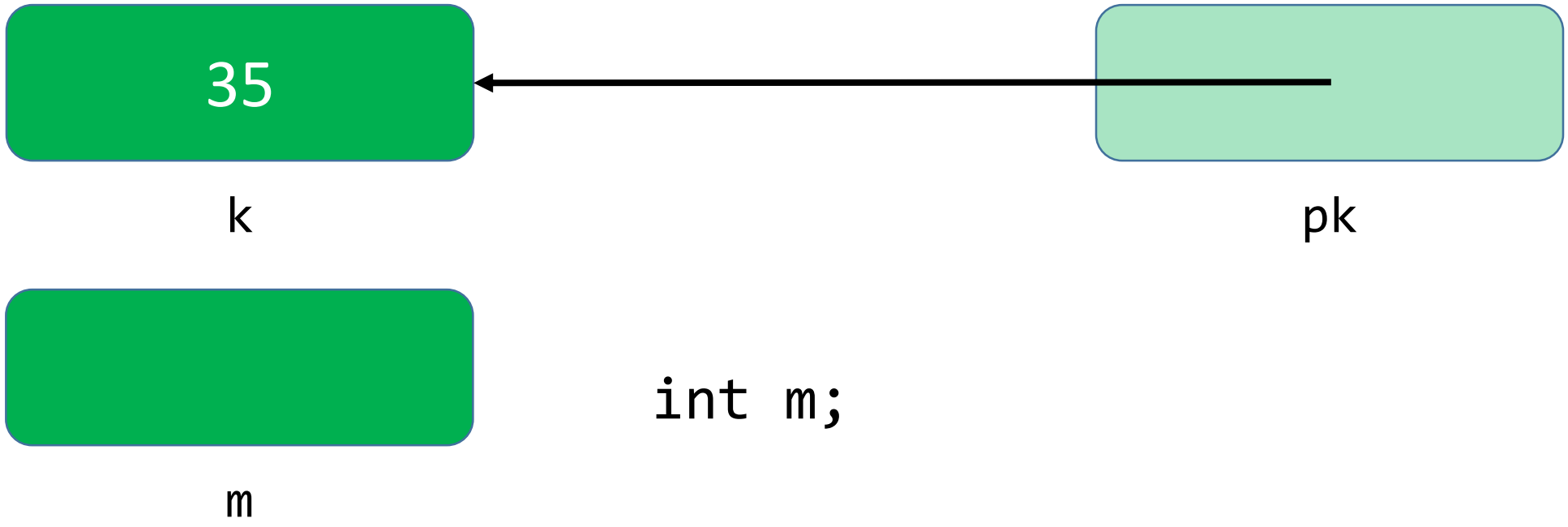
- So what happens?




```
int m;
```

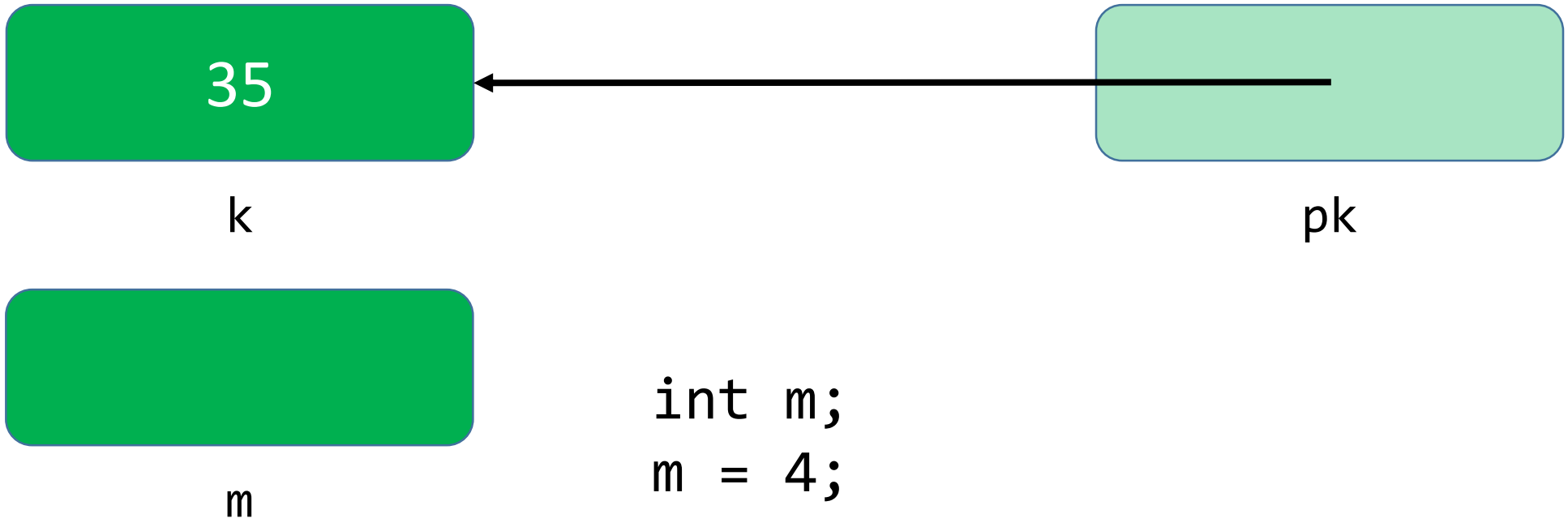
Pointers

- So what happens?



Pointers

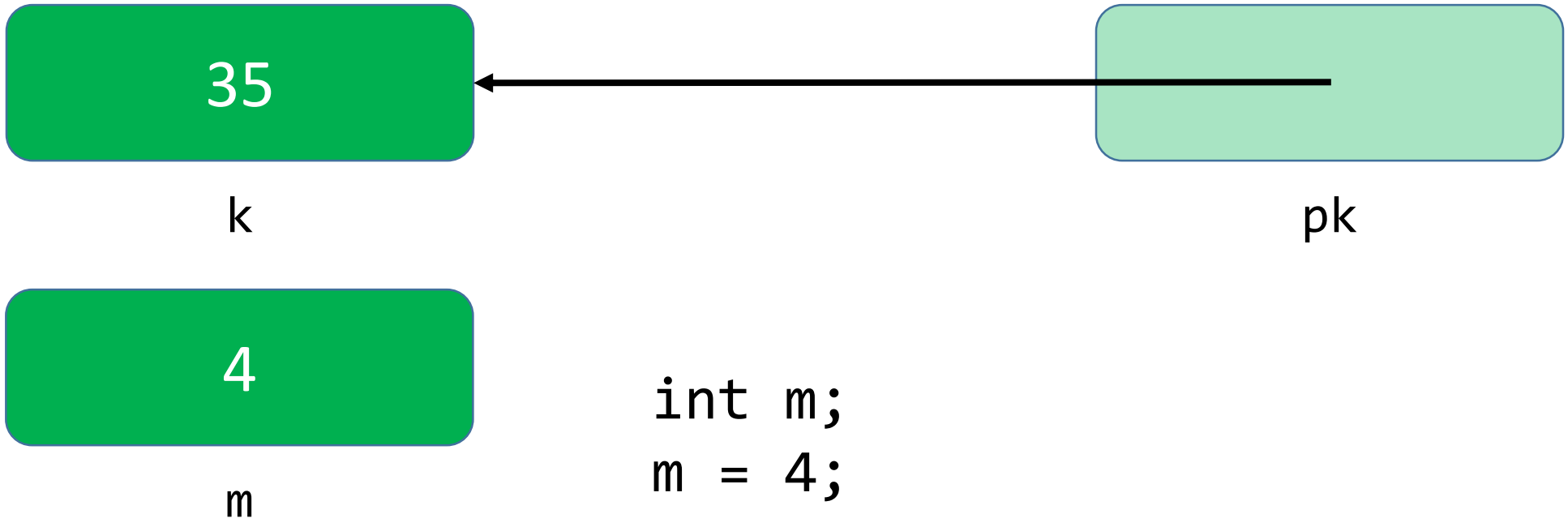
- So what happens?




```
int m;
m = 4;
```

Pointers

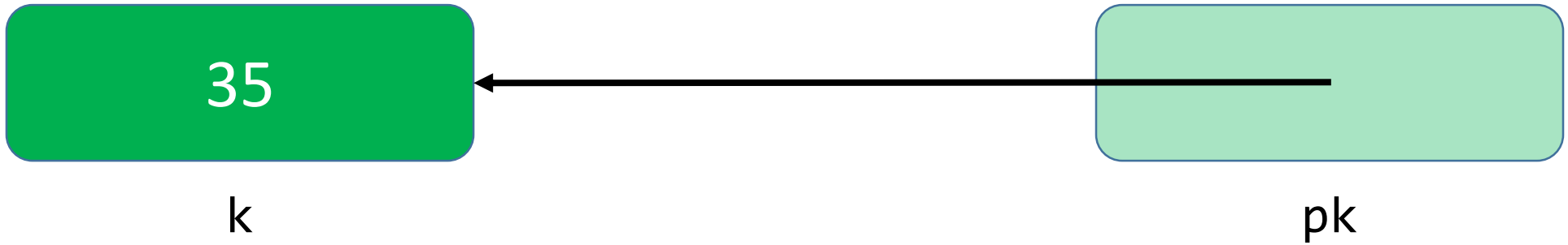
- So what happens?




```
int m;
m = 4;
```

Pointers

- So what happens?




```
int m;
m = 4;
pk = &m;
```

Pointers

- So what happens?