Pointers and Arrays

Based on materials by Dianna Xu

Today’s Goals

- Pointers
  - Declaration
  - Assignment
  - Indirection/de-referencing
- Arrays

Common C/C++ Data Types

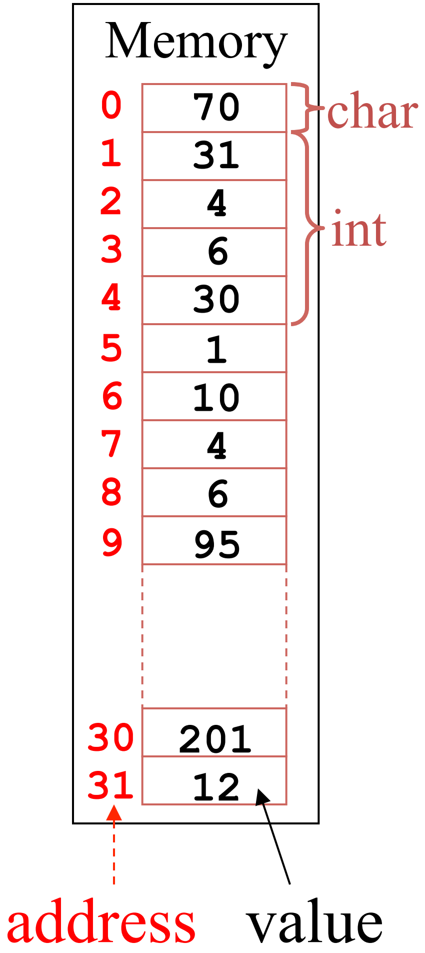
Type	Size		Largest value	Smallest value
	[bit]	[byte]		
`int`	32	4	2×10^9	-2×10^9
`float`	32	4	10^{38}	-10^{38}
`double`	64	8	10^{308}	-10^{308}
`char`	8	1	127	-128

`Double` stands for “double-precision floating point”.

- Based on 32-bit architecture
- Shaded values are approximate.
- Precision of **`float`** is 6 digits, **`double`** is 9-15 digits.

Variable and Address

- Variable = Storage in computer memory
  - Contains some value
  - Must reside at a specific location called *address*
  - Basic unit – byte
  - Imagine memory as a one-dimensional array with addresses as byte indices
  - A variable consists of one or more bytes, depending on its type (size)



Memory

0	70
1	31
2	4
3	6
4	30
5	1
6	10
7	4
8	6
9	95
30	201
31	12

Pointer – Reference

- A pointer (pointer variable) is a variable that stores an address (like Java reference)
  - value – address of some memory
  - type – size of that memory
- Recall in Java, when one declares variables of a *class* type, these are automatically references.
- In C/C++, pointers have special syntax and much greater flexibility.

Memory and Address

- A machine with 16 Megabytes of memory has ? bytes

$$16 \times 2^{20} = 2^4 \times 2^{20} = 16{,}777{,}216$$

- Since each byte has a unique address, there are at least that many addresses
- A pointer stores a memory address, thus the size of a pointer is machine dependent
- With most data models it is the largest integer on the machine, size of **`unsigned long`**
- Defined in **`inttypes.h`**
  - **`uintptr_t`** and **`uintmax_t`**

Address Operations in C/C++

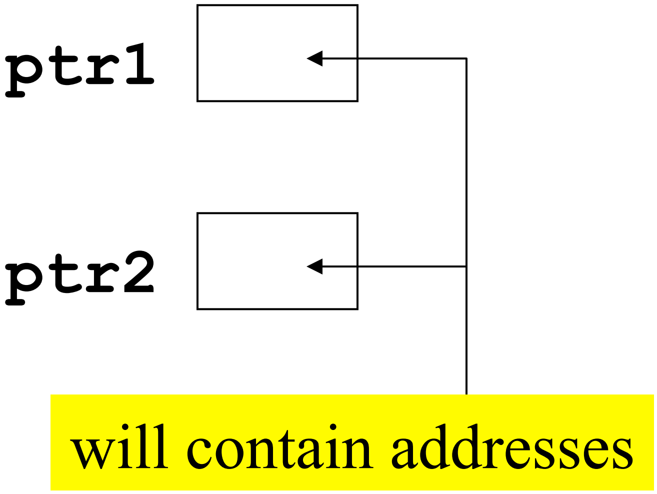
- Declaration of pointer variables
  - – The *pointer declarator* ‘`*`’
- Use of pointers
  - – The *address of* operator ‘`&`’
  - – The *indirection* operator ‘`*`’ – also known as de-referencing a pointer

Pointer Declaration

- Syntax
  - *destinationType* * *varName*;
- Must be declared with its associated type.
- Examples
  - `int *ptr1;`

    A pointer to an **`int`** variable
  - `char *ptr2;`

    A pointer to a **`char`** variable



Pointers are NOT integers

- Although memory addresses are essentially very large integers, pointers and integers are not interchangeable.
- Pointers are not of the same type
- A pointer’s type depends on what it points to
  - `int *p1; // sizeof(int)`
  - `char *p2; // sizeof(char)`
- C/C++ allows free conversion btw different pointer types via casting (dangerous)

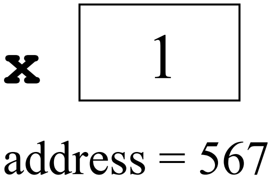
Address of Operator

- Syntax
  - `&` *expression*

    The expression must have an address. E.g., a constant such as “`1`” does not have an address.
- Example
  - `int x = 1;`

    `f(&x);`

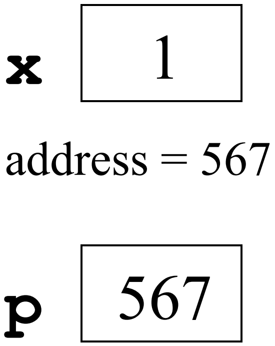
    The address of **`x`** (i.e. where **`x`** is stored in memory), say, the memory location 567, (not 1) is passed to **`f`**.



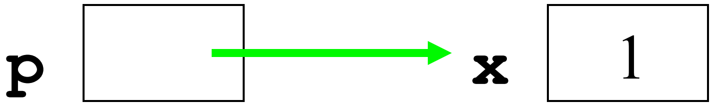
Pointer Assignment

- A pointer **`p`** *points* to **`x`** if **`x`**' s address is stored in **`p`**
- Example

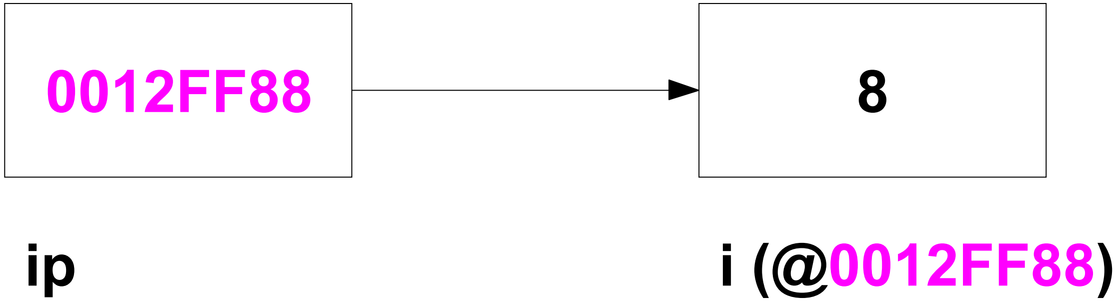
  -
    ```
    int x = 1;
    int *p;
    p = &x;
    ```



Interpreted as:



Pointer Diagram



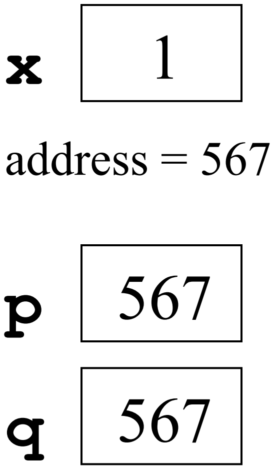
```
int i = 8;
int *ip;

ip = &i;
```

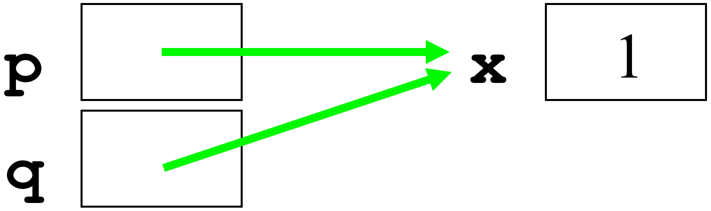
Pointer Assignment

- A pointer **p** *points* to **x** if **x**’ s address is stored in **p**
- Example

  -
    ```
    int x = 1;
    int *p, *q;
    p = &x;
    q = p;
    ```

    Interpreted as:

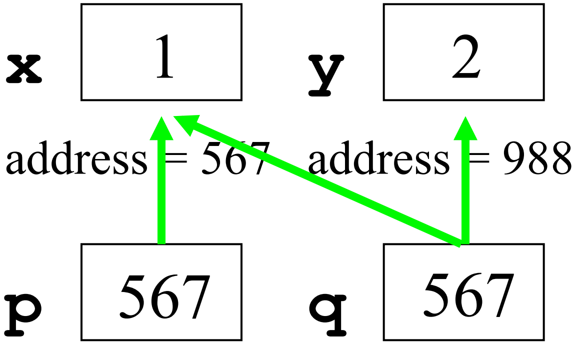




Pointer Assignment

- Example

  -
    ```
    int x=1, y=2, *p, *q;
    p = &x; q = &y;
    q = p;
    ```

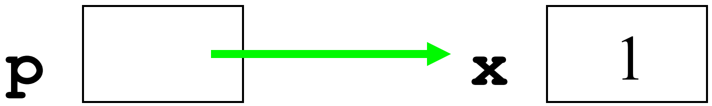


Indirection Operator

Note: '*' in a declaration and '*' in an expression are different.
`int *p; int * p; int* p;`

- Syntax
  - `*` *pointerVar*
  - Allows access to value of memory being pointed to
  - Also called *dereferencing*
- Example

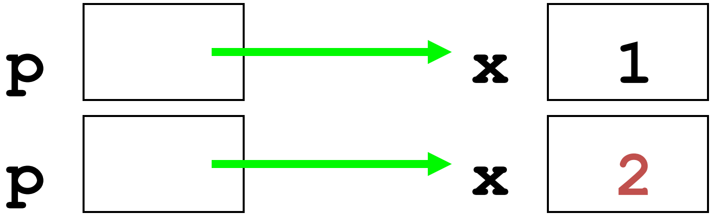
  -
    ```
    int x = 1, *p;
    p = &x;
    printf("%d\n", *p);
    ```
    `*p` refers to **`x`**; thus prints 1



Assignment Using Indirection Operator

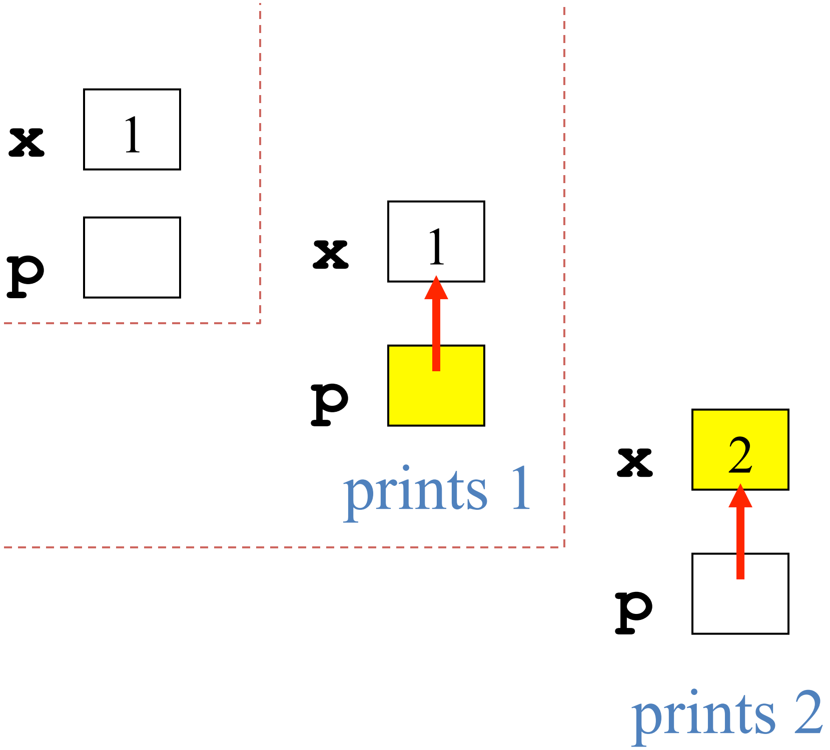
- Allows access to a variable indirectly through a pointer pointed to it.
- Pointers and integers are not interchangeable
- Example

  -
    ```
    int x = 1, *p;
    p = &x;
    *p = 2;
    printf("%d\n", x);
    ```
  - `*p` is equivalent to `x`



Schematically

```
int x = 1;
int *p;
p = &x;
printf("%d", *p);
*p = 2;
printf("%d", x);
```



The **NULL** Pointer

- C++ guarantees that zero is never a valid address for data
- A pointer that contains the address zero known as the **NULL** pointer
- It is often used as a signal for abnormal or terminal event
- It is also used as an initialization value for pointers

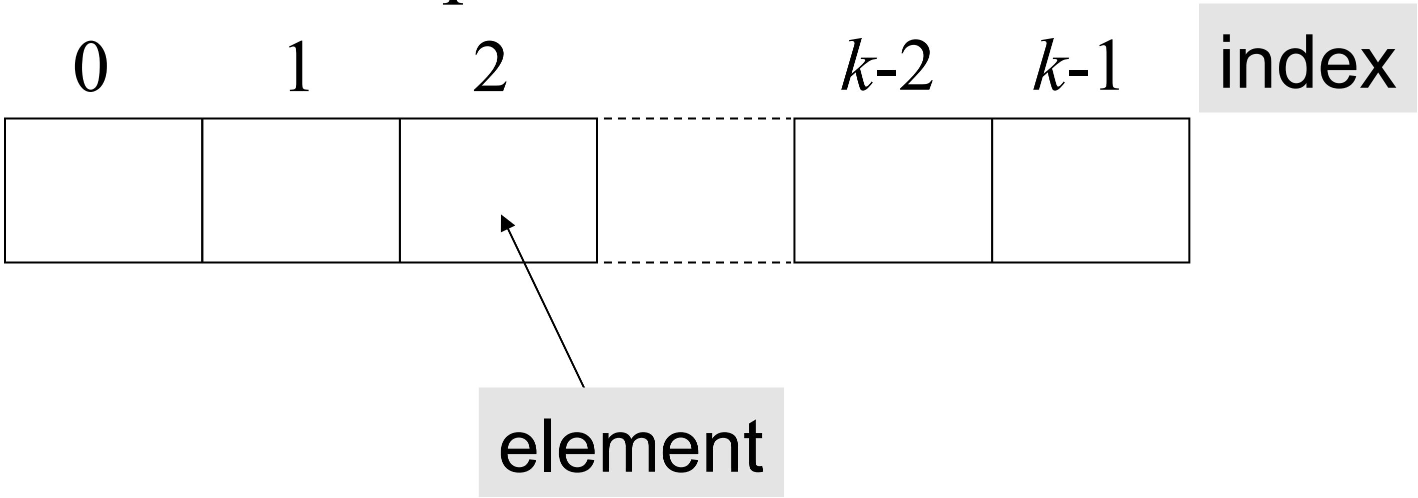
Arrays

- Declaration – `int a[5];`
- Assignment – `a[0] = 1;`
- Reference – `y = a[0];`

a	?	?	?	?	?

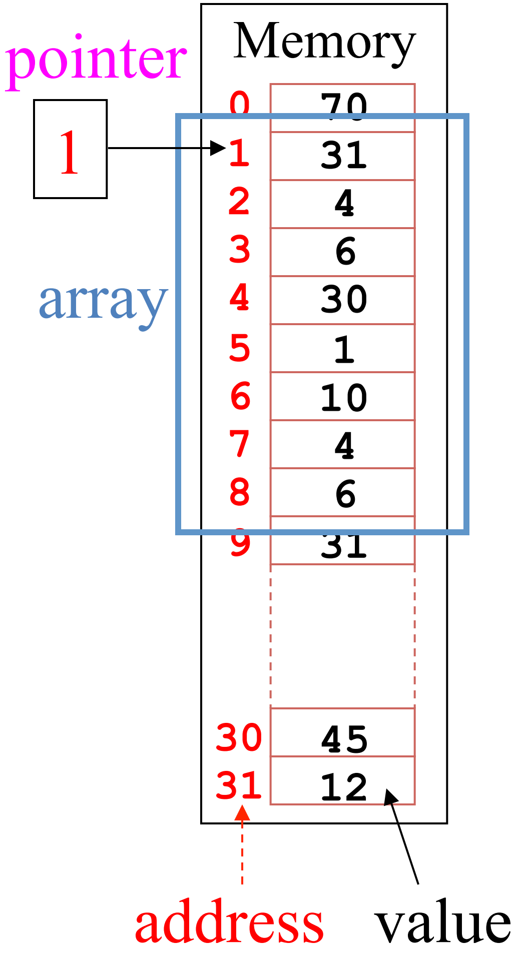
	0				4
a	1	?	?	?	?

- Schematic representation



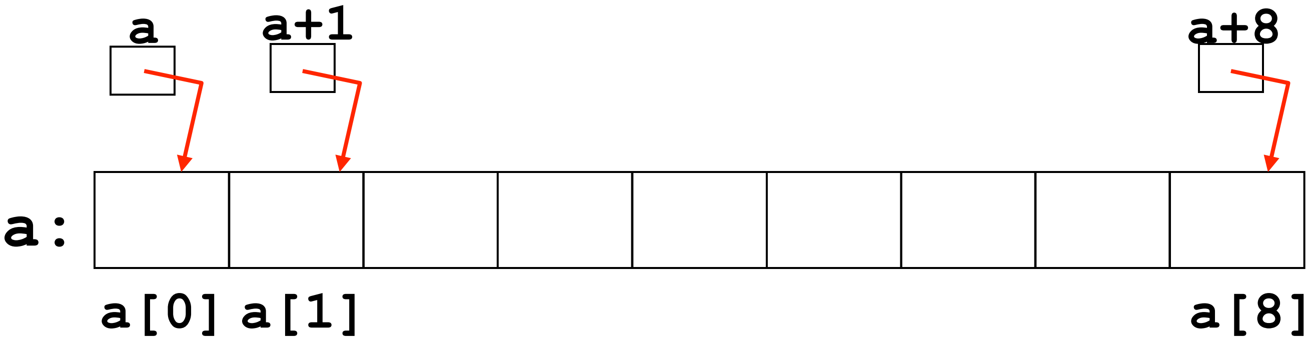
Pointers and Arrays

- Arrays are contiguous allocations of memory of the size: **`sizeof(elementType) * numberOfElements`**
- Given the address of the first byte, using the type (size) of the elements one can calculate addresses to access other elements



Name of an Array

- The variable name of an array is also a pointer to its first element.



- `a == &a[0]`
- `a[0] == *a`

Pointer Arithmetic

- One can add/subtract an integer to/from a pointer
- The pointer advances/retreats by that number of *elements (of the type being pointed to)*
  - `a+i == &a[i]`
  - `a[i] == *(a+i)`
- Subtracting two pointers yields the number of *elements* between them

Multi-Dimensional Array

```
int a[2][3];
```

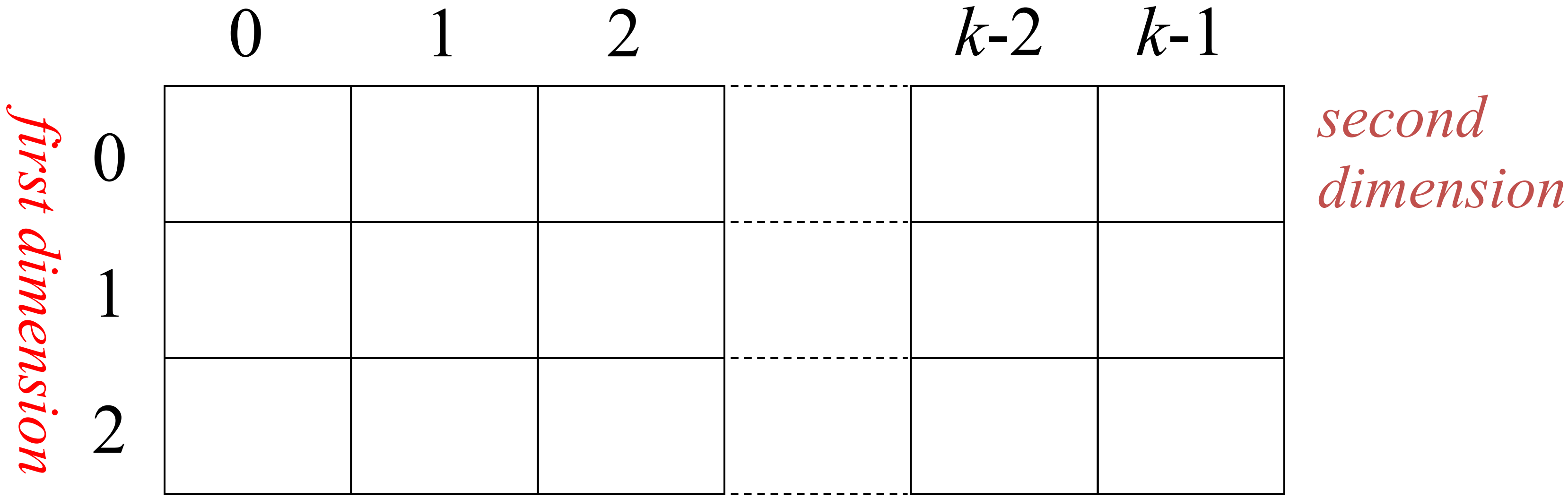
a

?	?	?
?	?	?

```
a[0][1] = 5;
y = a[0][1];
```

a

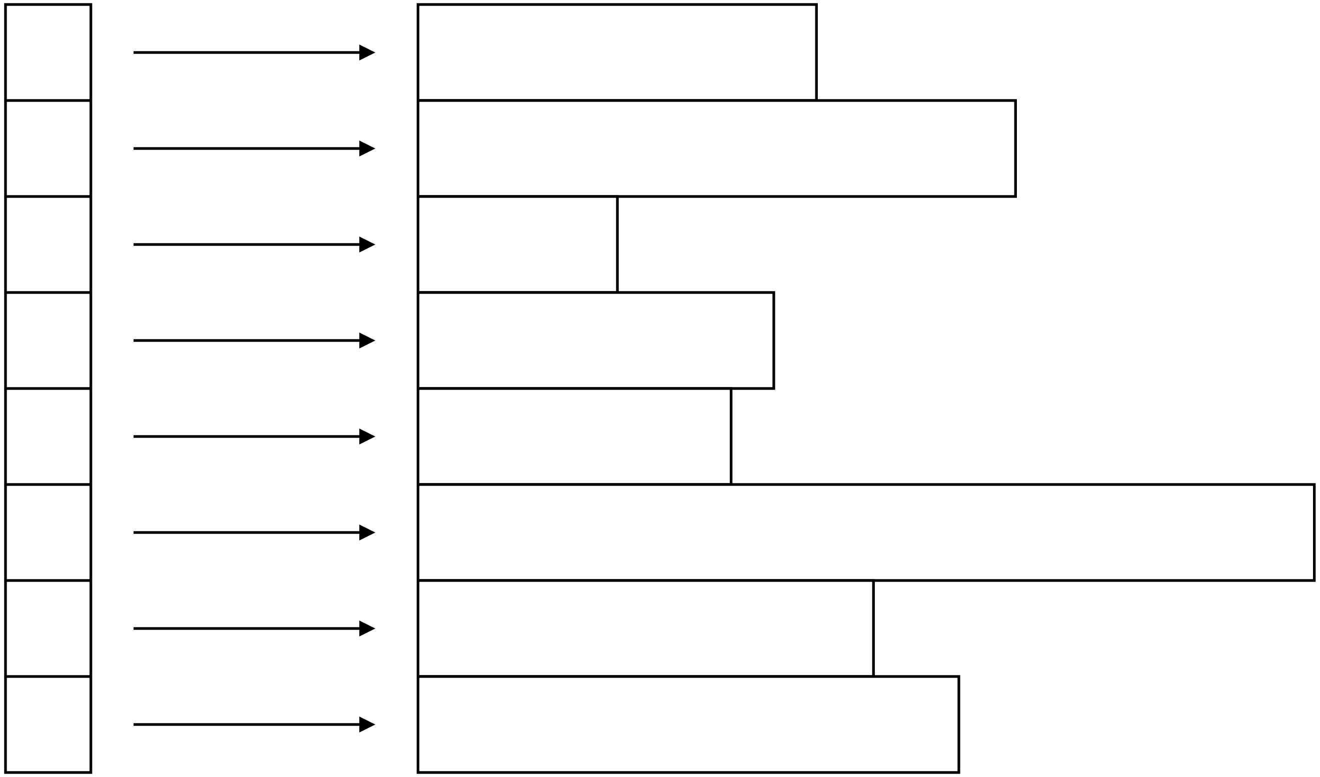
	0	1	2
0	?	5	?
1	?	?	?



Pointer Arrays: Pointer to Pointers

- Pointers can be stored in arrays
- Two-dimensional arrays are just arrays of pointers to arrays.
  - `int a[10][20]; int *b[10];`
  - Declaration for b allows 10 **`int`** pointers, with no space allocated.
  - Each of them can point to an array of 20 integers
  - `int c[20]; b[0] = c;`
  - What is the type of **`b`**?

Ragged Arrays



Summary

- Pointer and integers are not exchangeable
- Levels of addressing (i.e. layers of pointers) can be arbitrarily deep
- Remember the **`&`** that you MUST put in front of **`scanf`** variables?
- Failing to pass a pointer where one is expected or vise versa always leads to segmentation faults.
- Understand the relationship between arrays and pointers
- Understand the relationship between two-dimensional arrays and pointer arrays
- Pointer arithmetic is powerful yet dangerous!