CS 31: Introduction to Computer Systems

04: Introduction to C

01-30-2025

Announcements

- Register your clicker! https://forms.gle/YBFvNWPTXgiySMHx5
- Reading quizzes count from this week!
- HW 1 is out! – New Due Date is Monday Feb 3rd
  - Please let me know if you don't have a homework group!
- Please give me your accommodation forms this week
- Sophomore planning information session Feb 5th, Wed, 12-1pm, in Sci 204.

Reading Quiz

- Note the red border!
- 1 minute per question
- No talking, no laptops, phones during the quiz

Check your frequency:

- Iclicker2: frequency AA
- Iclicker+: green light next to selection

For new devices this should be okay,
For used you may need to reset frequency

Reset:

1. hold down power button until blue light flashes (2secs)
2. Press the frequency code: AA vote status light will indicate success

The First “Computers”: Women

FUN FACT

ENIAC was developed 10 mi from here, at UPenn

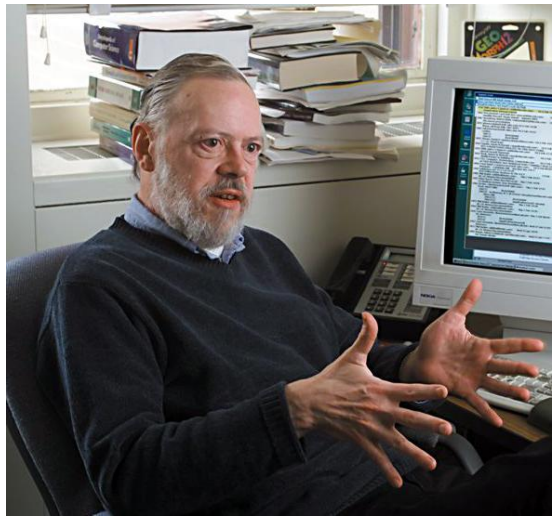
Smithsonian
MAGAZINE

SCIENCE

The Gendered History of Human Computers

It’s ironic that women today must fight for equality in Silicon Valley. After all, their math skills helped launch the digital age

How did we get to C?



Dennis Ritchie
worked at Bell Laboratories, Inc.

Machine / Hardware

C → Unix

C was created for systems programming back in 1972.

C was created to write Unix.

first transistor, solar cell, compilers,
C, C++, Unix, deep learning, + more!

Why C in this course?

Did you ever see the wizard of Oz?

What was going on behind the curtains?

More than what you would think!

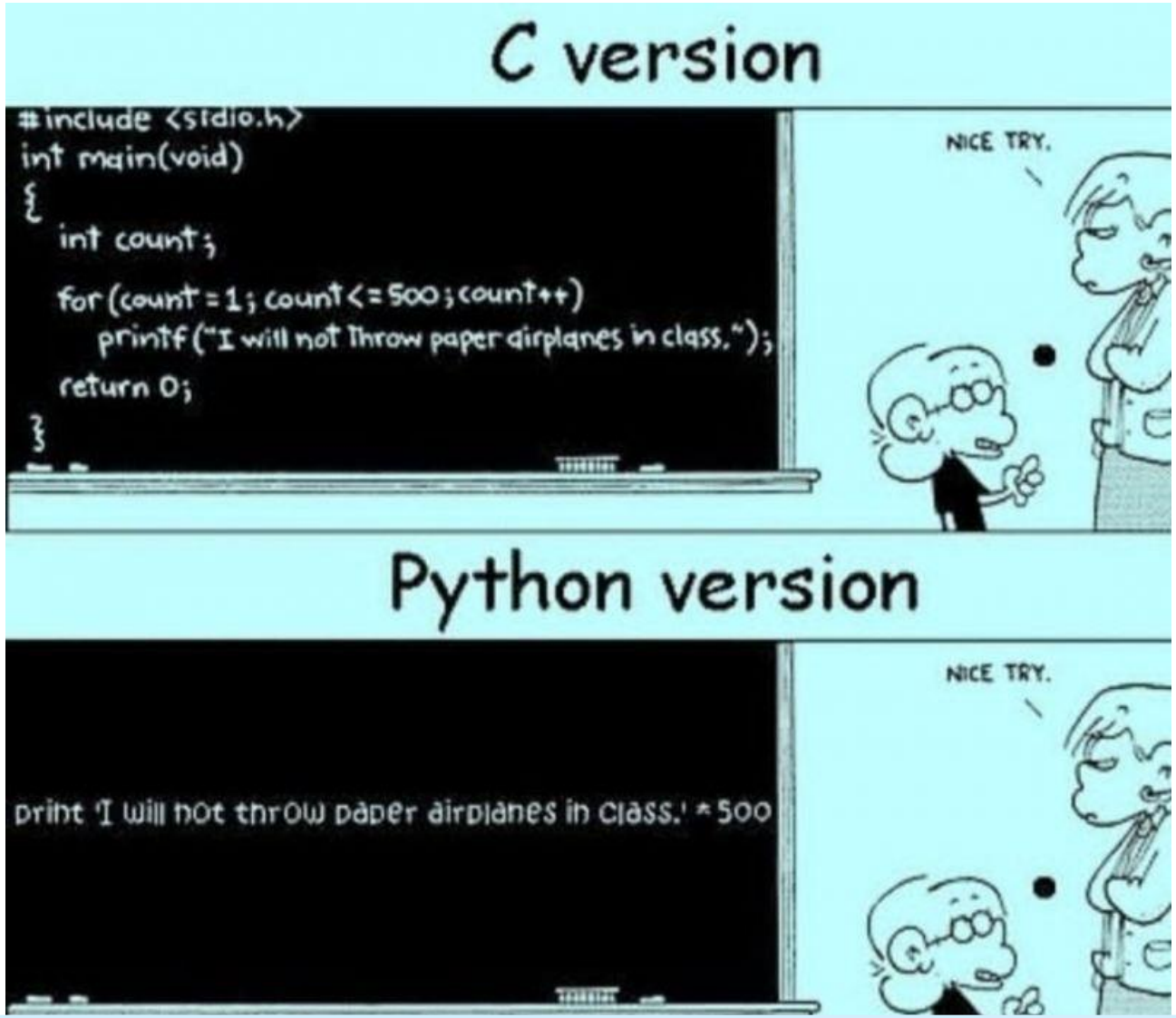
The mystery revealed!

Python versus C: Paradigms

Python and C follow different programming paradigms.

- C:
  - is procedure-oriented.
  - breaks down to functions.
- Python:
  - follows an object-oriented paradigm.
  - allows Python to break down objects and methods.

Python versus C: Paradigms



https://devrant.com/rants/1755638/c-vs-python

So, the point(er) is….?

- Programming languages are tools
  - Python is one language and it does its job well
  - C is another language and it does its job well

- Pick the right tool for the job
  - C is a good language to explore how the system works under-the-hood.
  - C is the Language of Systems Programmers: Fast running OS code that exposes the details of the hardware is really important!

- It’s the right tool for the job we need to accomplish in this course!

So, the point(er) is….?

- Programming languages are tools. C is a good high-level language to explore how the system works under-the-hood, revealing the **relationship** between code and computer execution
  - C is the **language of systems programmers**: Fast running OS code that exposes the details of the hardware is really important!
    - Greater control over how a program uses and accesses **memory**
    - Can more easily **parallelize** programs in C
- It’s the right tool for the job we need to accomplish in this course!

Python versus C: Paradigms

Python and C follow different programming paradigms.

- C:
  - is procedure-oriented.
  - breaks down to functions.
- Python:
  - follows an object-oriented paradigm.
  - allows Python to break down objects and methods.

Hello World

Python

```python
# hello world
import math

def main():
    print "hello world"

main()
```

C

```c
// hello world
#include <stdio.h>

int main( ) {
  printf("hello world\n");
  return 0;
}
```

Hello World

Python	C

`# hello world`
`import math`
 	
`def main():`
`  print "hello world"`
 	
`main()`
	`// hello world`
	`#include <stdio.h>`
	
	`int main( ) {`
	`  printf("hello world\n");`
	`  return 0;`
	`}`

#: single line comment	**//**: single line comment

Hello World

Python	C

`# hello world`
`import math`
 	
`def main():`
`  print "hello world"`
 	
`main()`
	`// hello world`
	`#include <stdio.h>`
	
	`int main( ) {`
	`  printf("hello world\n");`
	`  return 0;`
	`}`

#: single line comment	**//**: single line comment
import libname: include Python libraries	**#include<libname>**: include C libraries

Hello World

Python	C

`# hello world`
`import math`
 	
`def main():`
`  print "hello world"`
 	
`main()`
	`// hello world`
	`#include <stdio.h>`
	
	`int main( ) {`
	`  printf("hello world\n");`
	`  return 0;`
	`}`

#: single line comment	**//**: single line comment
import libname: include Python libraries	**#include<libname>**: include C libraries
Blocks: **indentation**	Blocks: **{ }** (indentation for readability)

"White Space"

- Python cares about how your program is formatted. Spacing has meaning.

- C compiler does NOT care. Spacing is ignored.
  - This includes spaces, tabs, new lines, etc.
  - Good practice (for your own sanity):
    - Put each statement on a separate line.
    - Keep indentation consistent within blocks.

Are these the same program?

```
#include <stdio.h>

int main(void) {
    int number = 7;
    if (number > 10) {
        do_this();
    } else {
        do_that();
    }
}
```

```
#include <stdio.h>
int main(void) { int number =7; if
(number > 10) { do_this();
   } else
{
do_that();}}
```

A. Yes, but one is harder to read

B. No

C. I can't tell…

Are these the same program?

```
#include <stdio.h>

int main(void) {
    int number = 7;
    if (number > 10) {
        do_this();
    } else {
        do_that();
    }
}
```

```
#include <stdio.h>
int main(void) { int number =7; if
(number > 10) { do_this();
   } else
{
do_that();}}
```

Yes – but one is harder to debug than the other

Hello World

Python	C

`# hello world`
`import math`
 	
`def main():`
`  print "hello world"`
 	
`main()`
	`// hello world`
	`#include <stdio.h>`
	
	`int main(void) {`
	`  printf("hello world\n");`
	`  return 0;`
	`}`

#: single line comment	**//**: single line comment
import libname: include Python libraries	**#include<libname>**: include C libraries
Blocks: **indentation**	Blocks: **{ }** (indentation for readability)
print: statement to printout string	**printf**: function to print out format string
statement: each on separate line	**statement**: each ends with **;**
def main(): : the main function definition	**int main(void)** : the main function definition (int specifies the **return type** of main)

Types

- Everything is stored as bits.
- Type tells us how to interpret those bits.
- “What type of data is it?”
  - integer, floating point, text, etc.

Type Matters!

- No self-identifying data
  - Looking at a sequence of bits doesn't tell you what they mean
  - Could be signed, unsigned integer
  - Could be floating-point number
  - Could be part of a string
- The machine interprets what those bits mean!

Types in C

- All variables have an explicit type!

- You (programmer) must declare variable types.
  - Where: at the beginning of a block, before use.
  - How: <variable type> <variable name>;

- Examples:

```
int humidity;          float temperature;
humidity = 20;         temperature = 32.5;
```

We have to explicitly declare variable types ahead of time? Lame! Python figured out variable types for us, why doesn't C?

A. C is old.
B. Explicit type declaration is more efficient.
C. Explicit type declaration is less error prone.
D. Dynamic typing (what Python does) is imperfect.
E. Some other reason (explain)

We have to explicitly declare variable types ahead of time? Lame! Python figured out variable types for us, why doesn't C?

A. C is old.
B. Explicit type declaration is more efficient.
C. Explicit type declaration is less error prone.
D. Dynamic typing (what Python does) is imperfect.
E. Some other reason (explain)

Numerical Type Comparison

Integers (int)

- Example:

```
int age;
age = 20;
```

- Only represents integers
- Small range, high precision
- Faster arithmetic

Floating Point (float, double)

- Example:

```
float temperature;
temperature = 32.5;
```

- Represents fractional values
- Large range, less precision
- Slower arithmetic

I need a variable to store a number, which type should I use?

Use the one that fits your specific need best…

An Example with Local Variables

```
/* a multiline comment:
   anything between slashdot and dotslash */

#include <stdio.h> // C's standard I/O library (for printf)

int main() {
   // first: declare main's local variables
   int x, y;
   float z;

   // followed by: main function statements
   x = 6;
   y = (x + 3)/2;
   z = x;
   z = (z + 3)/2;

   printf(...)  // Print x, y, z
}
```

An Example with Local Variables

```
/* a multiline comment:
   anything between slashdot and dotslash */

#include <stdio.h> // C's standard I/O library (for printf)

int main() {
   // first: declare main's local variables
   int x, y;
   float z;

   // followed by: main function statements
   x = 6;
   y = (x + 3)/2;
   z = x;
   z = (z + 3)/2;

   printf(...)  // What is the output here?
}
```

Clicker choices

	X	Y	Z
A	4	4	4
B	6	4	4
C	6	4.5	4
D	6	4	4.5
E	6	4.5	4.5

An Example with Local Variables

```
/* a multiline comment:
   anything between slashdot and dotslash */

#include <stdio.h> // C's standard I/O library (for printf)

int main() {
   // first: declare main's local variables
   int x, y;
   float z;

   // followed by: main function statements
   x = 6;
   y = (x + 3)/2; //x and y are both ints
   z = x; //z is a float, value of x is converted to a float
   z = (z + 3)/2;

   printf(...) // What is the output here?
}
```

Clicker choices

	X	Y	Z
A	4	4	4
B	6	4	4
C	6	4.5	4
D	**6**	**4**	**4.5**
E	6	4.5	4.5

Operators: need to think about type

Arithmetic: +, -, *, /, % (numeric type operands)

/: operation and result type depends on operand types:

- Two int operands: int division truncates: 3/2 is 1
- 1 or 2 float or double: float or double division: 3.0/2 is 1.5

%: mod operator: (only int or unsigned types)

- Gives you the (integer) remainder of division: 13 % 2 is 1, 27 % 3 is 0
- Shorthand operators :
  - **var op =** expr; (var = var op expr):
  - x += 4 is equivalent to x = x + 4
  - var**++**; var**--**; (var = var+1; var = var-1):

  x++ is same as x = x + 1 x-- is same as x = x -1;

Boolean values in C?

- There is no “boolean” type in C!
- Instead, **integer expressions** used in conditional statements are interpreted as true or false
- **Zero (0) is false, any non-zero value is true**
  - **Use this to always check return value of the function**
- Questions?
- “Which non-zero value does it use?
- E.g., int x = 10>5. what is x? //arbitrary non-zero value

The value of x is compiler specific don’t rely on the output to be a certain value

Operators: consider the type

- **Relational** (operands any type, result int “boolean”):
  - <, <=, >, >=, ==, !=
  - 6 != (4+2) is 0 (false)
  - 6 > 3 *some non-zero value (we don’t care which one) (true)*

- **Logical** (operands int “boolean”, result int “boolean”):
  - ! (not): !6 is 0 (false)
  - && (and): 8 && 0 is 0 (false)
  - || (or): 8 || 0 is non-zero (true)

Boolean values in C

- **Zero (0) is false, any non-zero value is true**
- **Logical** (operands int "boolean"->result int "boolean"):
  - ! (not): inverts truth value
  - && (and): true if both operands are true
  - || (or): true if either operand is true

Do the following statements evaluate to True or False?

#1: (!10) || (5 > 2)

#2: (-1) && ((!5) > -1)

Clicker choices

	#1	#2
A	True	True
B	True	False
C	False	True
D	False	False

Boolean values in C

- **Zero (0) is false, any non-zero value is true**
- **Logical** (operands int "boolean"->result int "boolean"):
  - ! (not): inverts truth value
  - && (and): true if both operands are true
  - || (or): true if either operand is true

Do the following statements evaluate to True or False?

#1: (!10) || (5 > 2)

#2: (-1) && ((!5) > -1)

Clicker choices

	#1	#2
A	True	True
B	True	False
C	False	True
D	False	False

Conditional Statements

Basic if statement:	With optional else:

if(<boolean expr>) {
  if-true-body
}
	if(<boolean expr>) {
	  if-true-body
	} else {
	  else body(expr-false)
	}

Very similar to Python, just remember { } are blocks:

w/o curly braces, only the next line will be executed!

Always use curly braces.

Conditional Statements

Chaining if-else if	With optional else:

if(<boolean expr1>) {
  if-expr1-true-body
} else if (<bool expr2>){
  else-if-expr2-true-body
  (expr1 false)
}
...
} else if (<bool exprN>){
  else-if-exprN-true-body
}
	if(<boolean expr1>) {
	  if-expr1-true-body
	} else if (<bool expr2>){
	  else-if-expr2-true-body
	}
	...
	} else if (<bool exprN>){
	  else-if-exprN-true-body
	} else {
	  else body
	  (all exprX's false)
	}

Very similar to Python, just remember { } are blocks

While Loops

Basically identical to Python while loops:

```
while(<boolean expr>) {
    while-expr-true-body
}
```

```
x = 20;
while (x < 100) {
  y = y + x;
  x += 4;  //  x = x + 4;
}
<next stmt after loop>;
```

```
x = 20;
while(1) {
  y = y + x;
  x += 4;
  if(x >= 100) {
     break;  // break out of loop
  }
}
<next stmt after loop>;
```

Which one of these results in an infinite loop?

A) while (x <100)
B) while(1)
C) Both A and B
D) Neither A or B

While Loops

Basically identical to Python while loops:

```
while(<boolean expr>) {
    while-expr-true-body
}
```

```
x = 20;
while (x < 100) {
  y = y + x;
  x += 4;  //  x = x + 4;
}
<next stmt after loop>;
```

```
x = 20;
while(1) {
  y = y + x;
  x += 4;
  if(x >= 100) {
     break;  // break out of loop
  }
}
<next stmt after loop>;
```

Which one of these results in an infinite loop?

A) while (x <100)
B) while(1)
C) Both A and B
D) Neither A or B

For loops: different than Python's

```
for (<init>; <cond>; <step>) {
    for-loop-body-statements
}
<next stmt after loop>;
```

1. Evaluate <init> one time, when first eval **for** statement
2. Evaluate <cond>, if it is false, drop out of the loop (<next stmt after>)
3. Evaluate the statements in the for loop body
4. Evaluate <step>
5. Goto step (2)

```
for(i=1; i <= 10; i++) {   // example for loop
    printf("%d\n", i*i);
}
```

What does this for loop print?

printf function

Python: `print "%d %s\t %f" % (6, "hello", 3.4)`

C: `printf("%d %s\t %f\n", 6, "hello", 3.4);`

`printf(<format string>, <values list>);`

%d	int placeholder (-13)
%f or %g	float or double (higher-precision than float) placeholder (9.6)
%c	char placeholder ('a')
%s	string placeholder ("hello there")
\t \n	tab character, new line character

Formatting Differences:

C: need to explicitly print end-of-line character (\n)

C: **string and char are different types**

'a': in Python is a string, in C is a **char**

"a": in Python is a string, in C is a **string**

Data Collections in C

- Many complex data types out there (CS 35)

- C has a few simple ones built-in:
  - Arrays
  - Structures (`struct`)
  - Strings (arrays of characters)

- Often combined in practice, e.g.:
  - An array of structs
  - A struct containing strings

Arrays

- C's support for collections of values
  - Array buckets store a single type of value
  - Specify max capacity (num buckets) when you declare an array variable (single memory chunk)

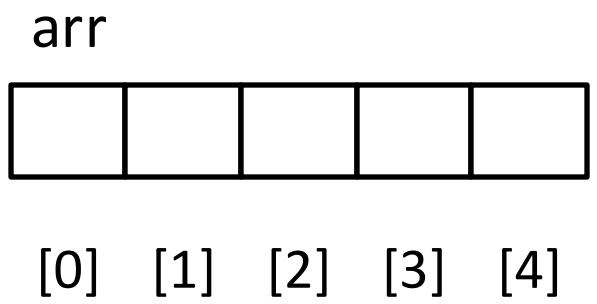
```
<type> <var_name>[<num buckets>];
int arr[5];  // an array of 5 integers
float rates[40]; // an array of 40 floats
```

Arrays

- C's support for collections of values
- Often accessed via a loop:

```
int arr[5];  // an array of 5 integers
float rates[40]; // an array of 40 floats
for (i=0; i < 5; i++) {
    arr[i] = i;
    rates[i] = arr[i]*2;
}
```

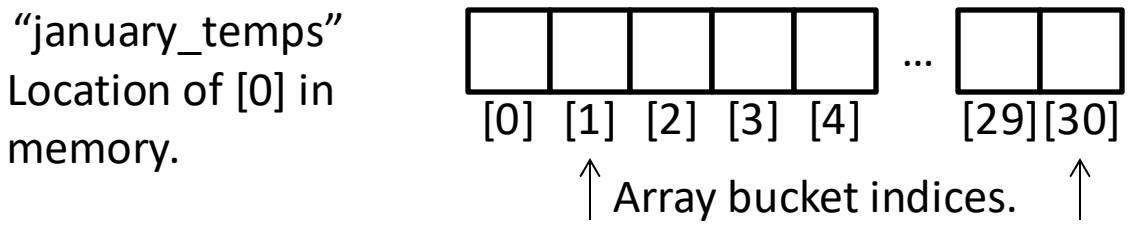

Get/Set value using brackets [] to index into array.



What does this for loop print?

Array Characteristics

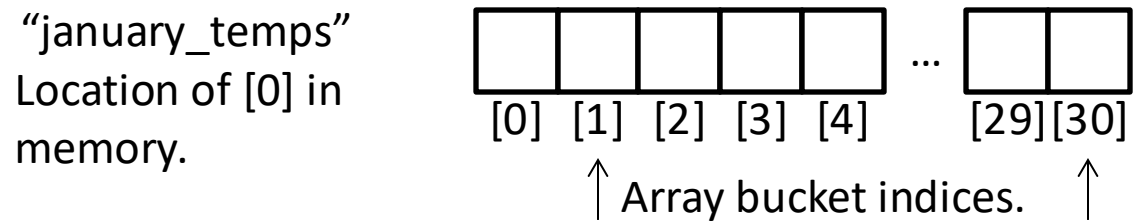
```
int january_temps[31];  // Daily high temps
```



- Indices start at 0! Why?
- Array variable name means, to the compiler, the beginning of the memory chunk. (The memory address)
  - january_temps" (without brackets!) Location of [0] in memory.
  - Keep this in mind, we'll return to it soon (functions).

Array Characteristics

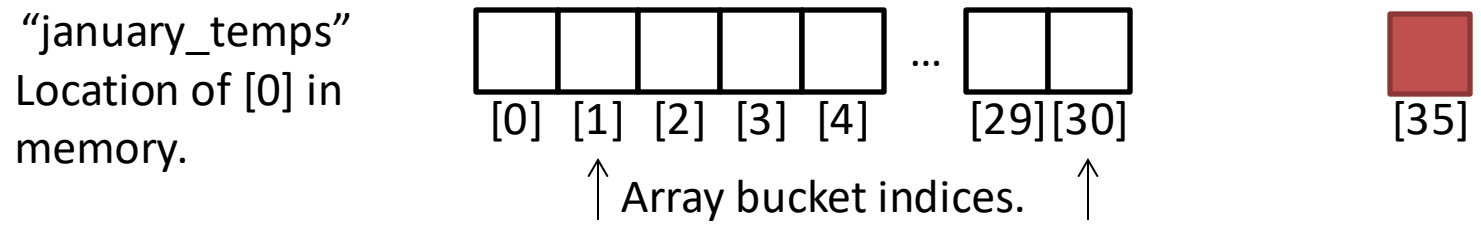
```
int january_temps[31];  // Daily high temps
```



- Indices start at 0! Why?
- The index refers to an offset from the start of the array
  - e.g., `january_temps`[3] means "three integers forward from the starting address of january_temps"

Array Characteristics

```
int january_temps[31];  // Daily high temps
```

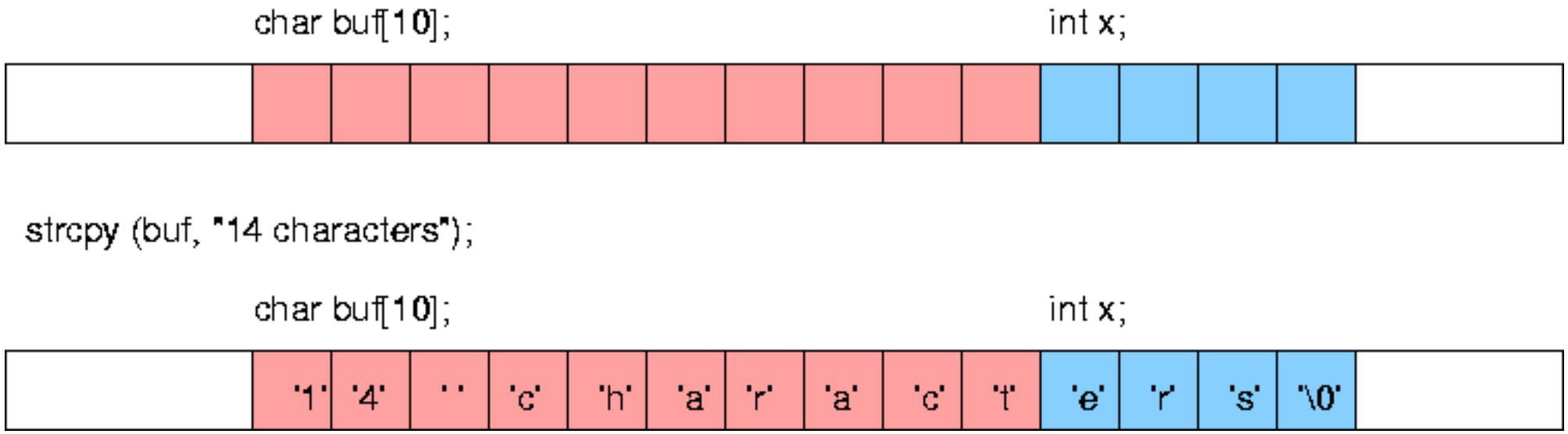


- Asking for january_temps[35]?

C does NOT do bounds checking.

- Python: error
- C: "Sure! I don't care .." <ominous silence while bad things happen>

Array Characteristics



C does NOT do bounds checking.

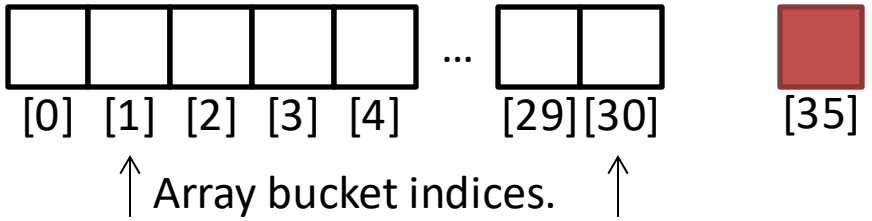
- Python: error
- C: “Sure! I don’t care ..” <ominous silence while bad things happen>

https://www.cs.fsu.edu/~baker/opsys/notes/bufferoverflow.html

Given what we know about arrays, how can we add a temperature reading second element in the array?

```
int january_temps[31];  // Daily high temps
```

"january_temps"
Location of [0] in memory.

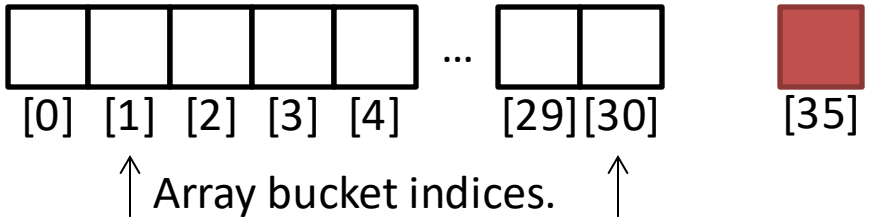


1. scanf("%d", january_temps);
2. scanf("%d", &january_temps[1]);
3. None of the above

Given what we know about arrays, how can we add a temperature reading second element in the array?

```
int january_temps[31];  // Daily high temps
```

"january_temps"
Location of [0] in memory.



1. scanf("%d", january_temps);
2. scanf("%d", &january_temps[1]);
3. None of the above

Functions and Stack Diagrams

Functions: Specifying Types

Need to specify the return type of the function, and the type of each parameter:

```
<return type> <func name> ( <param list> ) {
    // declare local variables first
    // then function statements
    return <expression>;
}

// my_function takes 2 int values and returns an int
int my_function(int x, int y) {
  int result;
  result = x;
  if(y > x) {
    result = y+5;
  }
  return result*2;
}
```

Compiler will yell at you if you try to pass the wrong type!

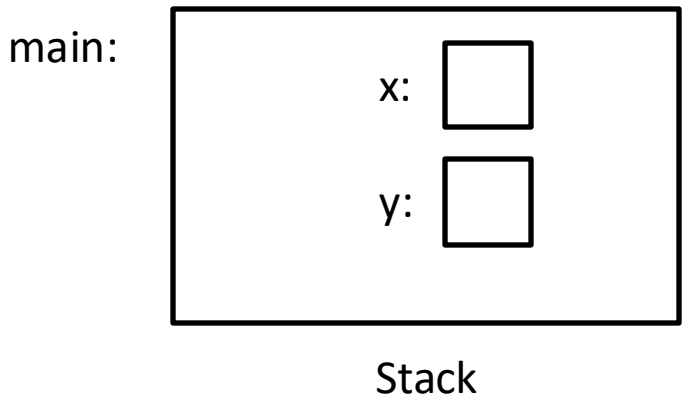
Function Arguments

Arguments are **passed by value**

- The function gets a separate copy of the passed variable

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```



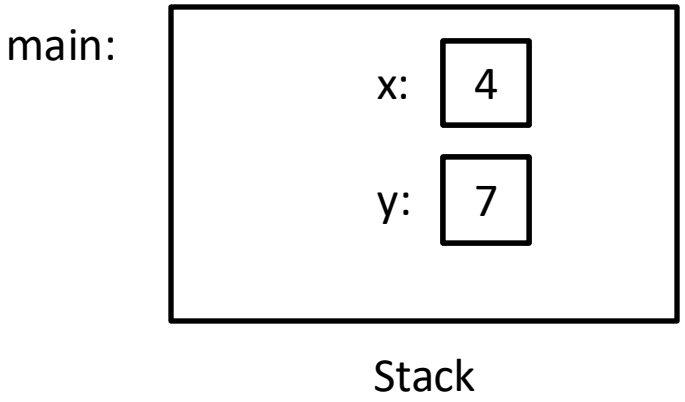
Function Arguments

Arguments are **passed by value**

- The function gets a separate copy of the passed variable

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```



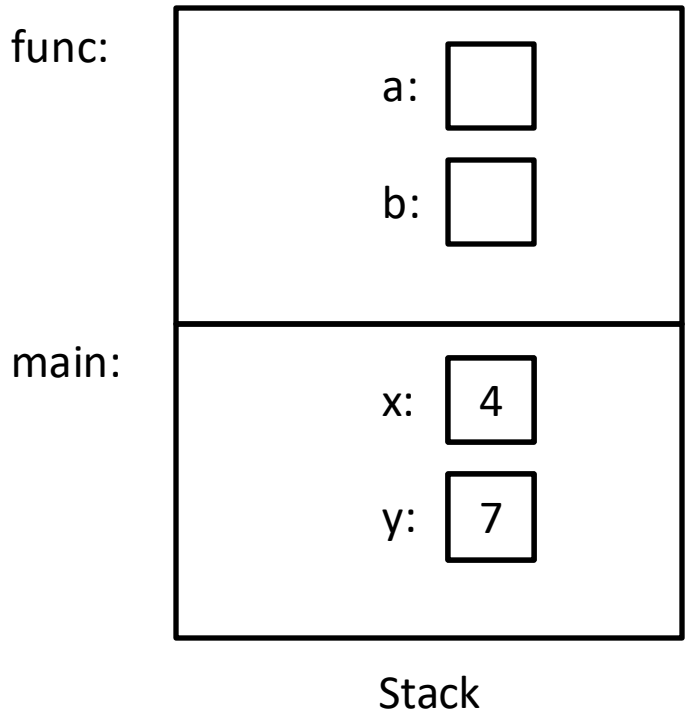
Function Arguments

Arguments are **passed by value**

- The function gets a separate copy of the passed variable

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```



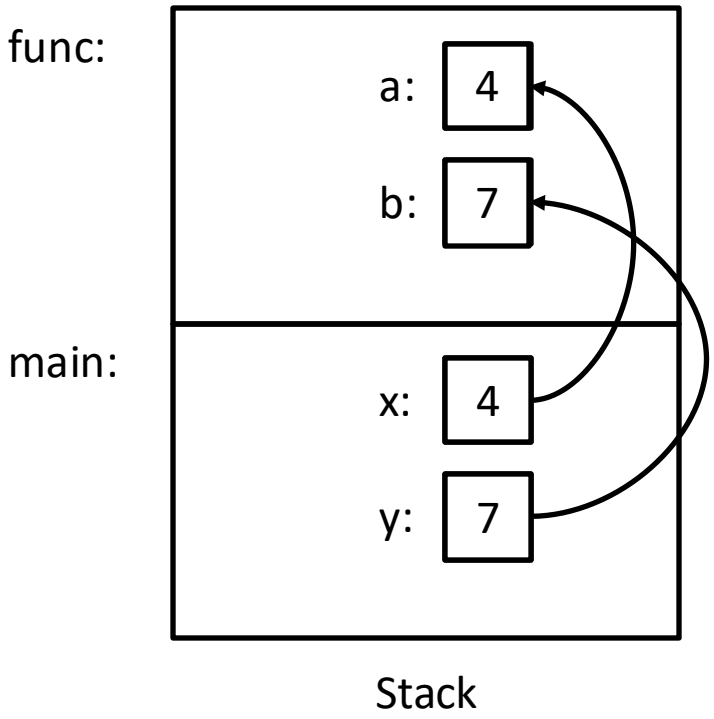
Function Arguments

Arguments are **passed by value**

- The function gets a separate copy of the passed variable

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```



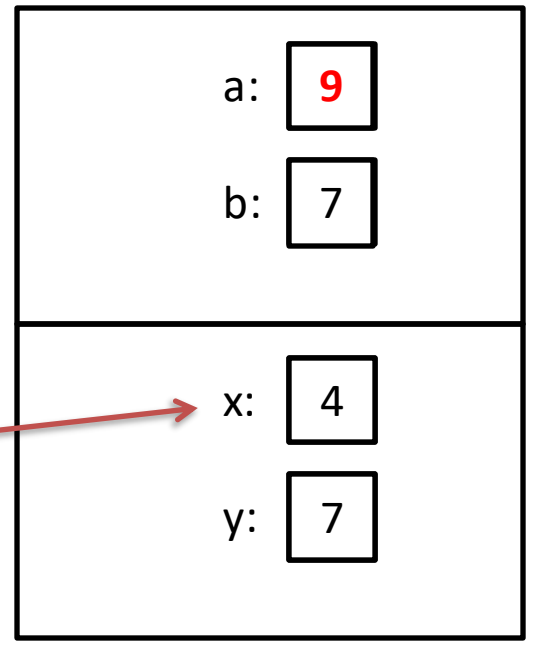
Function Arguments

Arguments are **passed by value**

- The function gets a separate copy of the passed variable

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```



Note: This doesn't change!

No impact on values in main!

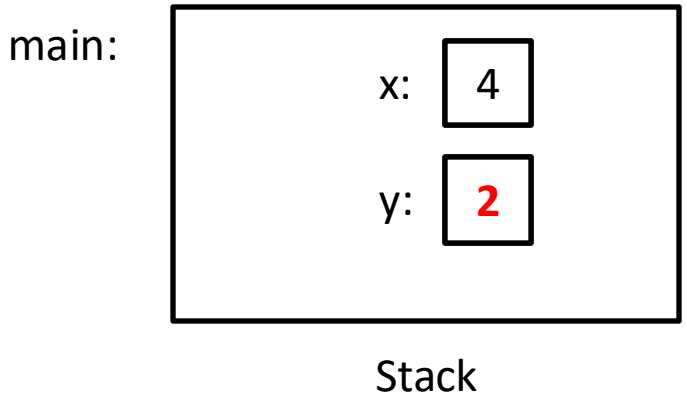
Function Arguments

Arguments are **passed by value**

- The function gets a separate copy of the passed variable

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```



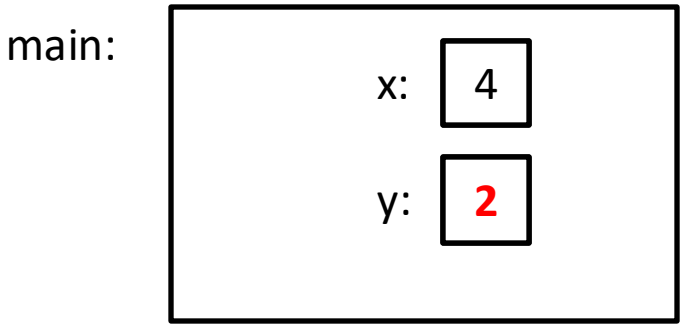
Function Arguments

Arguments are **passed by value**

- The function gets a separate copy of the passed variable

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```



Output: 4, 2

What will this print?

```
int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;//DRAW STACK DIAGRAM AT THIS POINT
    return y;
}

int main() {
    int x;
    int values[2];

    x = 0;
    values[0] = 5;
    values[1] = 10;

    x = func(x, x, values);

    printf("%d, %d, %d", x, values[0], values[1]);
}
```

A. 0, 5, 8
B. 0, 5, 10
C. 1, 0, 8
D. 1, 5, 8
E. 1, 5, 10

Hint: What does the name of an array mean to the compiler?

What will this print?

```
int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;
    return y;
}

int main() {
    int x;
    int values[2];

    x = 0;
    values[0] = 5;
    values[1] = 10;

    x = func(x, x, values);

    printf("%d, %d, %d", x, values[0], values[1]);
}
```

A. 0, 5, 8
B. 0, 5, 10
C. 1, 0, 8
D. 1, 5, 8
E. 1, 5, 10

Hint: Still accessing the same memory location of array in func

What will this print?

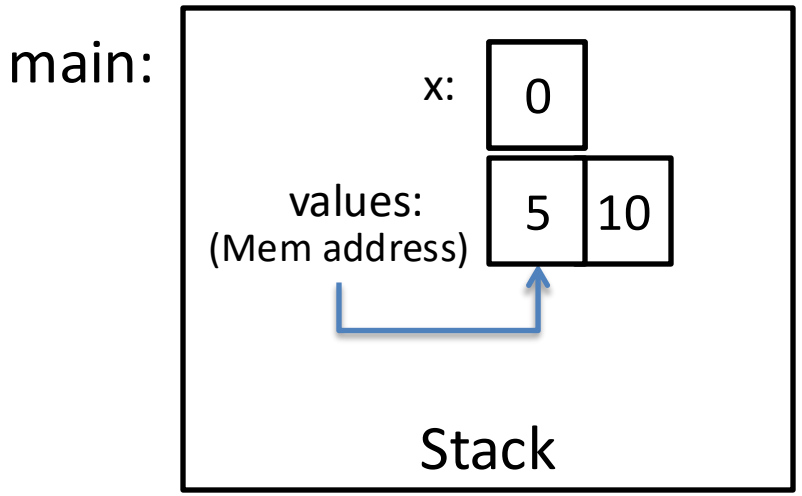
```
int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;
    return y;
}

int main() {
    int x;
    int values[2];

    x = 0;
    values[0] = 5;
    values[1] = 10;

    x = func(x, x, values);

    printf("%d, %d, %d", x, values[0], values[1]);
}
```



What will this print?

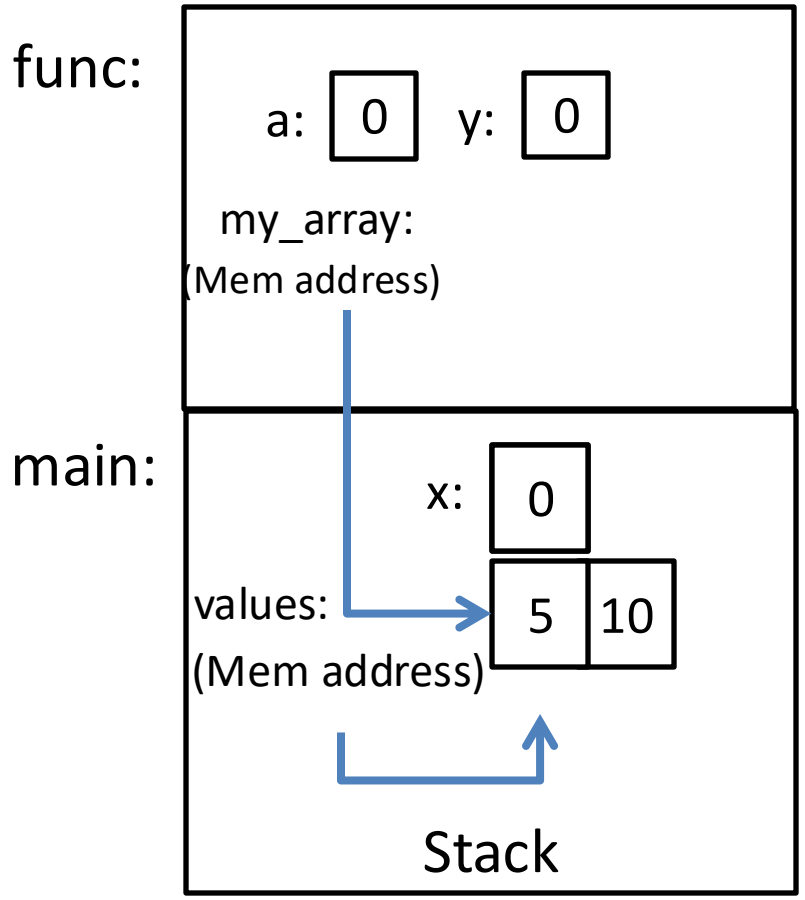
```
int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;
    return y;
}

int main() {
    int x;
    int values[2];

    x = 0;
    values[0] = 5;
    values[1] = 10;

    x = func(x, x, values);

    printf("%d, %d, %d", x, values[0], values[1]);
}
```



What will this print?

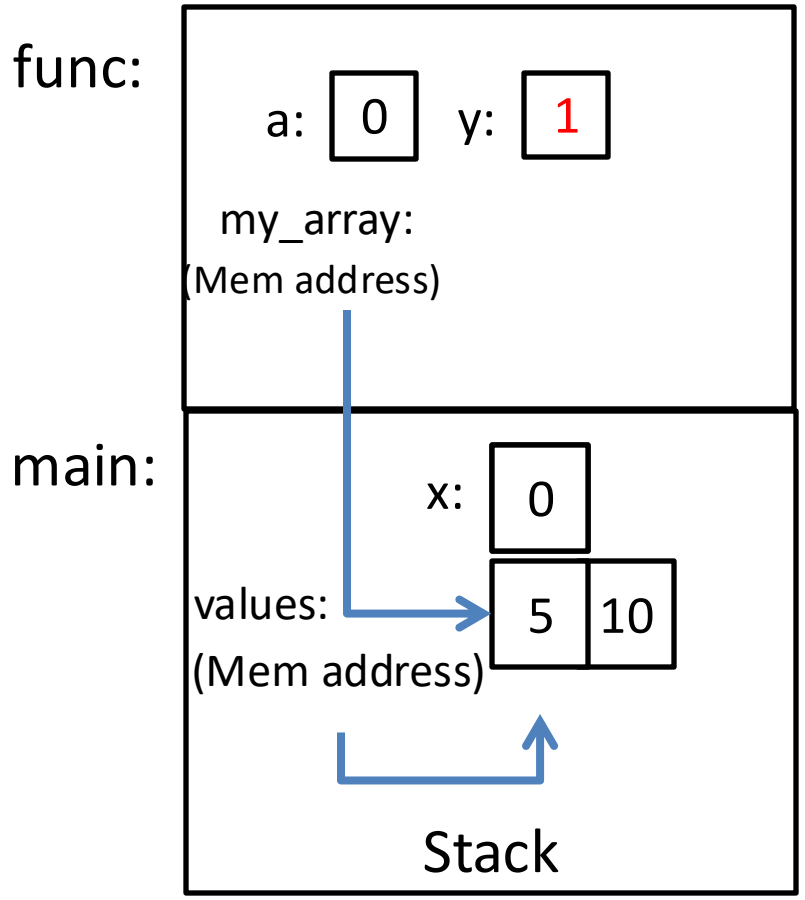
```
int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;
    return y;
}

int main() {
    int x;
    int values[2];

    x = 0;
    values[0] = 5;
    values[1] = 10;

    x = func(x, x, values);

    printf("%d, %d, %d", x, values[0], values[1]);
}
```



What will this print?

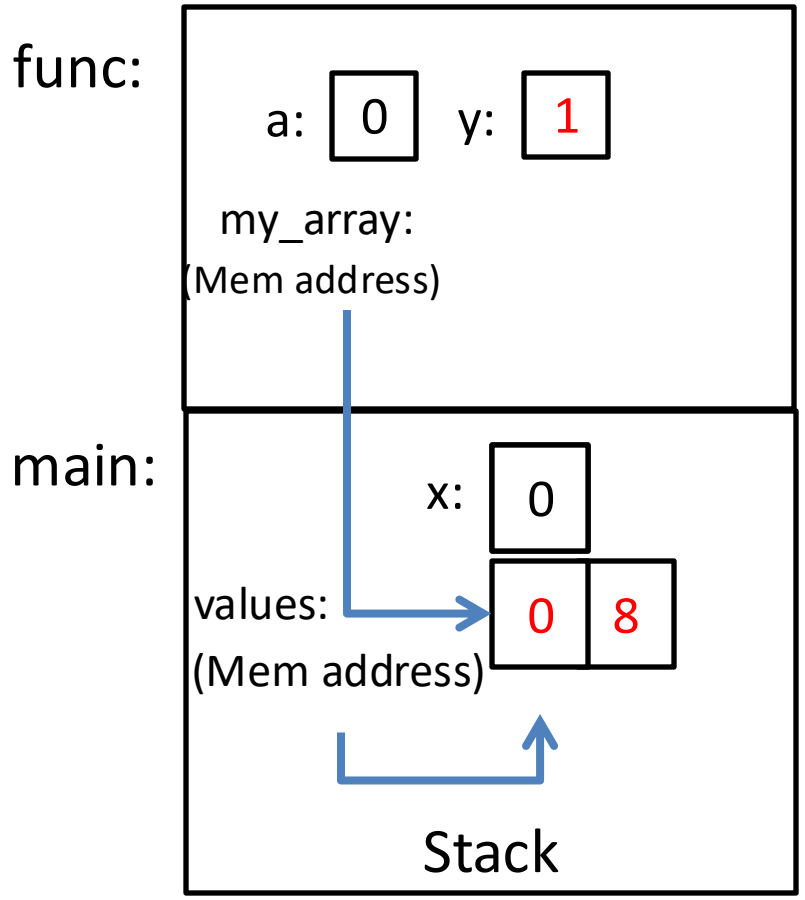
```
int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;
    return y;
}

int main() {
    int x;
    int values[2];

    x = 0;
    values[0] = 5;
    values[1] = 10;

    x = func(x, x, values);

    printf("%d, %d, %d", x, values[0], values[1]);
}
```



What will this print?

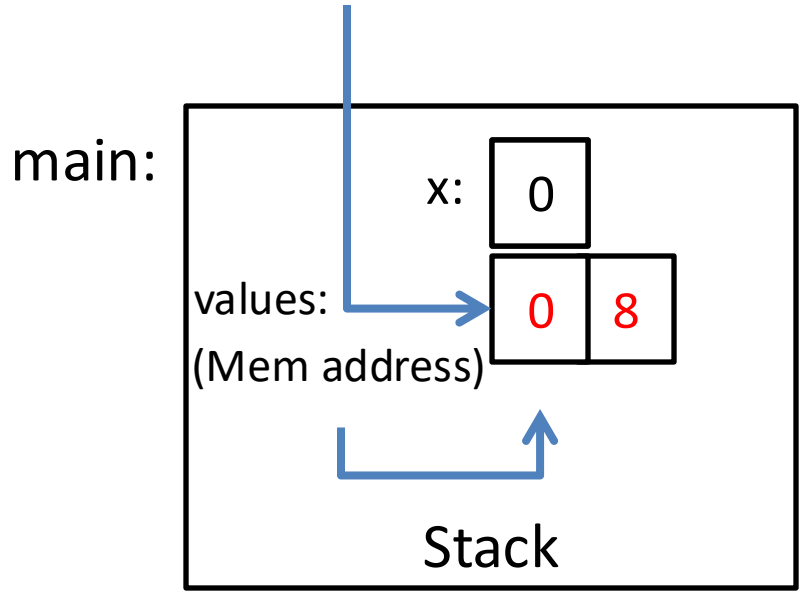
```
int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;
    return y;
}

int main() {
    int x;
    int values[2];

    x = 0;
    values[0] = 5;
    values[1] = 10;

    x = func(x, x, values);

    printf("%d, %d, %d", x, values[0], values[1]);
}
```



Fear not!

- Don’t worry, I don’t expect you to have mastered C.
- It’s a skill you’ll pick up as you go.
- We’ll revisit these topics when necessary.

- When in doubt: solve the problem in English, whiteboard pictures, whatever else!
  - Translate to C later.
  - Eventually, you’ll start to think in C.