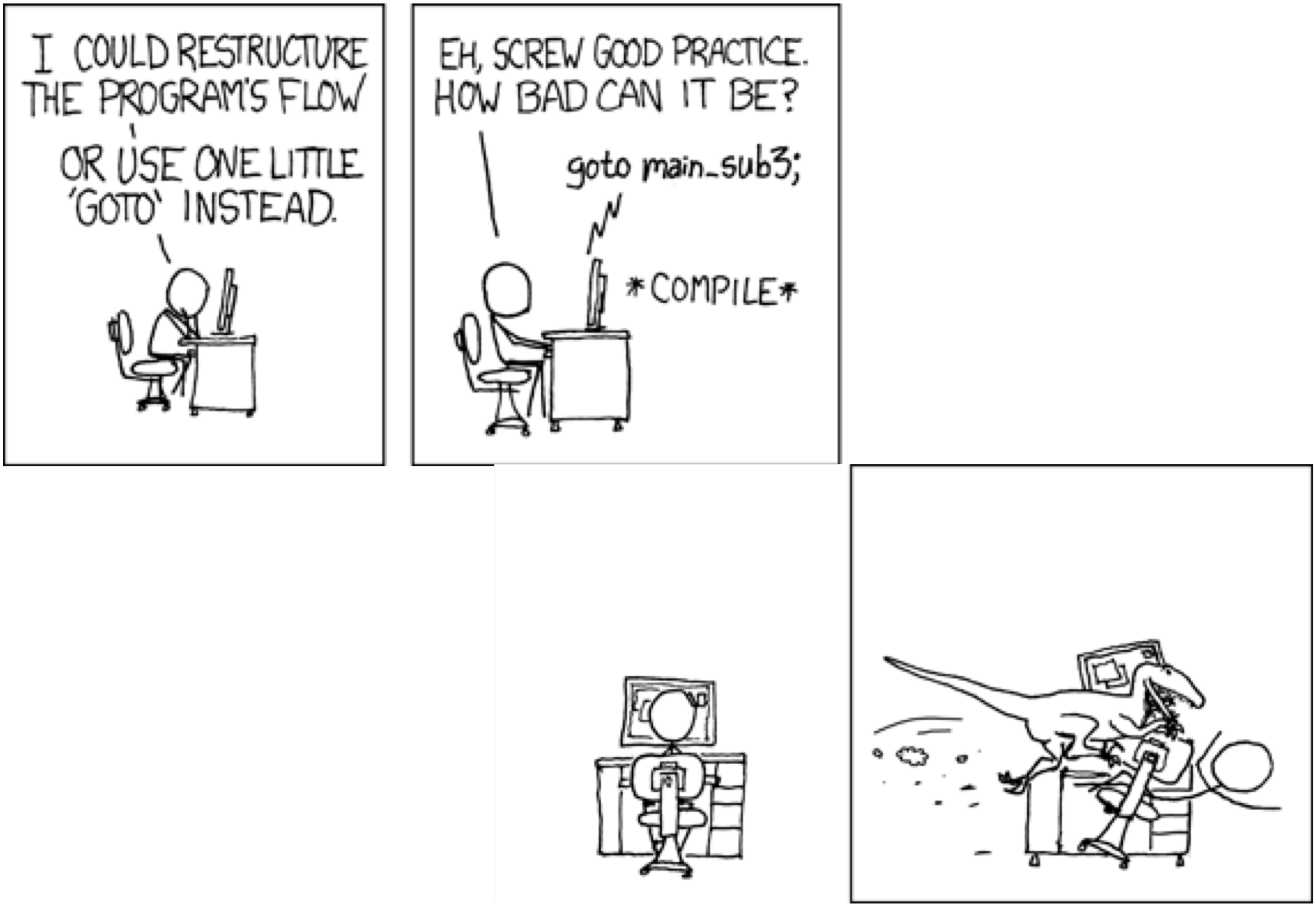
CS 31: Introduction to Computer Systems

04: C Programming

January 30

SWARTHMORE COLLEGE

Introduction to C programming



Announcements

- Midterm Exam in Lecture Hall – SCI 199
- Please let me know your accommodations TODAY!
- Homework 1 is out
  - Due in class next Thursday (Feb 6th)
  - Can work in groups of up to 3
  - All HWs worth 3%

Reading Quiz

Today

- Basics of C programming
  - Comments, variables, print statements, loops, conditionals, etc.
  - Ask questions if you have them!

- Comparison of C vs. Python
  - Data organization and strings
  - Functions

How a Computer Runs a Program

Binary Program	
Software Libraries	How instructions & data are encoded
Operating System	OS Abstractions, Resource management
Computer Hardware	How underlying HW organized & works

What we know so far:

- Binary Arithmetic, and overflow rules
  - Two's complement Subtraction can be reframed as addition
- Know encodings and sizes for different C types
  - char, unsigned char, int, unsigned int, …
- Know how to perform binary operations (Add, Sub)
  - Don't yet know how the Add HW circuit works (next week)

A Main Course Goal:

Understand how program written in a high-level language is run on the underlying System (OS/HW)

Understand the details

all the way down

C Program

```
main() {
    int x;
    x = 6 + 7;
    printf("x %d", x);
}
```

gcc

0101011010101010101010
1010101010101010100…

Binary Program

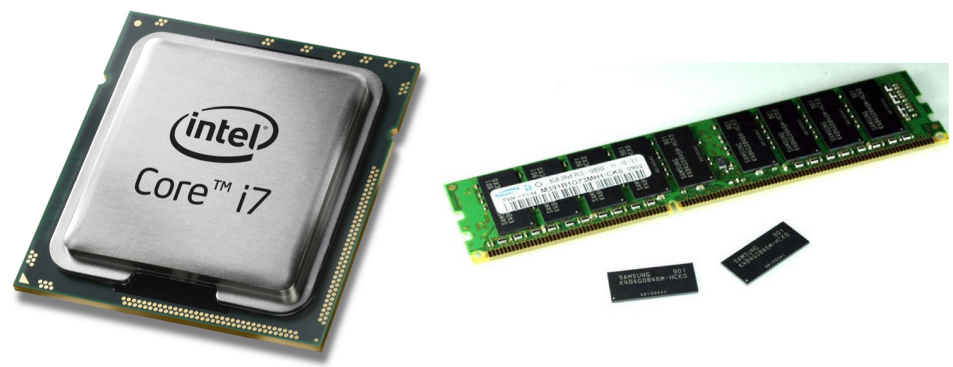
Software Libraries

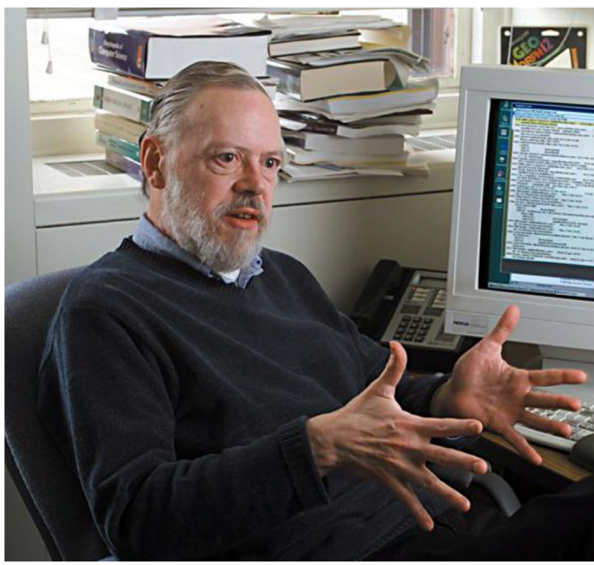
Operating System

Computer Hardware

What is C?

Machine / Hardware





Dennis Ritchie
worked at Bell Labs

C → Unix

C was created for **systems programming** back in 1972.

C was created to **write Unix.**

Evolution of C – K&R, ANSI C

- **K&R** (Brian Kernighan & Dennis Richie)
  - Bell Labs
  - "The C Programming Language", 1st Edition (1978)
- **ANSI C** (American National Standards Institute)
  - Defines both language and standard C library
  - "The C Programming Language", 2nd Edition (1989)
- **ISO C90** (International Standards Organization) ≅ ANSI C in 1990
- **ISO C99** (International Standards Organization)
  New data types, moving C to 64 bit processors
  Support for text strings with characters not in the English language
- **C18**: Current International Standard

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!

So, why C in this course?

- Closed Curtains
  - A safe place for programmers!
  - Python hides certain aspects of reality
  - This is good!
- Behind The Curtains Programming
  - Most Operating Systems (OS)
  - Access to memory and memory management
  - Dangerous, but necessary
  - Important to understand how the real system works
  - Makes you a **better** programmer!

Python versus C: Paradigms

Python and C follow different programming paradigms.

- C:
  - is more procedure-oriented.
  - breaks down to functions.
- Python
  - follows an object-oriented paradigm.
  - allows Python to break down Objects.

Python versus C: Paradigms

- Python is an “interpreted” language: interpreter does lexing, parsing, compiling and interpreting!
- Python does not allow you to access memory directly.
- Python provides automatic memory management. It is a “garbage collected” language.
- Python types “expand to available memory”.
- Python provides “exceptions”: if your program has an error at run-time it will throw an exception.

- C is compiled language: compiles to machine code that is “executed” by the underlying processor.
- C allows you to access memory directly, interpret that memory in any way you wish, and shoot yourself in the foot.
- C provides manual memory management.
- Types in C are dependent on the machine you are running on.
- No built-in error handling, if you are lucky (and smart) your program will check “error conditions” and fail gracefully.

Intro to C Programming

- is closer to the machine: see relationship between C code and computer execution
- can write faster code!
- want abstraction? out of luck – DIY implementation of dictionary
- It gives you access to aspects of the machine that are not accessible in 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 is really really important!

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

GNU Compiler Collection

```
$gcc –g –o prog prog.c
```

GNU Compiler Collection

```
$gcc –g –o prog prog.c
```

compile programs using the GNU C Compiler

GNU Compiler Collection

```
$gcc –g –o prog prog.c
```

compile programs using the GNU C Compiler

Command line options

-g: generates debugging information

-o: build to output file prog

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;`
	`}`

Hello World

Python	C

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

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

Hello World

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

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

```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;
}
```

Python	C
#: 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() {
   int number = 7;
   if (number > 10) {
      do_this();
   } else {
      do_that();
   }
}
```

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

Yes – but one is harder to debug than the other

Hello World

Python

```
# hello world
import math

def main():
   print “hello world”

main()
```

C

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

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

Python	C
#: 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( )** : 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.

Representation 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?

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 (maybe?)
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 humidity;

  humidity = 20;

- Only represents integers
- Small range, high precision
- Faster arithmetic
- (Maybe) less space required

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?

<u>Use the one that fits your specific need best…</u>

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

```c
/* 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
}
```

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;
   z = x;
   z = (z + 3)/2;

   printf(…)  // Print x, y, z
}
```

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?

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

Operators: need to think about 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?

F T

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

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

T T

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, no curly braces, only the next line will be executed! Always use curly braces.

Conditional Statements

<table>
<tr><th><u>Chaining if-else if</u></th><th><u>With optional else:</u></th></tr>
<tr><td><pre>
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
}
</pre></td><td><pre>
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)
}
</pre></td></tr>
</table>

<u>Very similar to Python, just remember { } are blocks</u>

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

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);
}
```

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

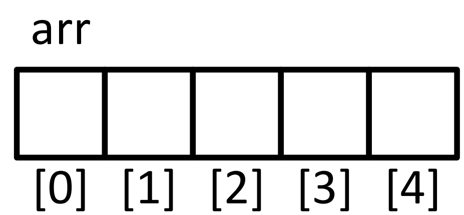
- C's support for <u>collections of values</u>
  - Array buckets store a single type of value
  - <u>Specify max capacity</u> (num buckets) when you declare an array variable (single memory chunk)

```
<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:

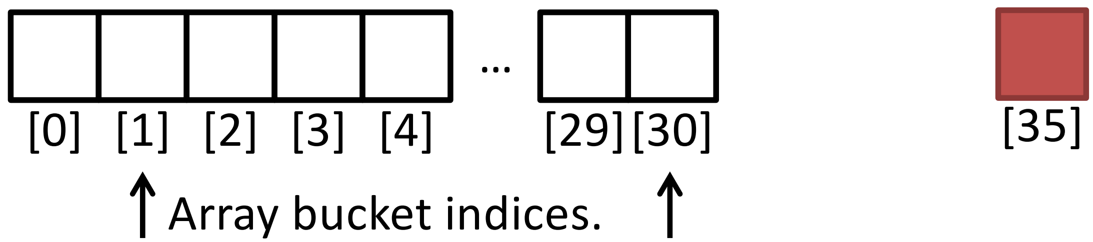
```
int arr[5];  // an array of 5 integers
float rates[40]; // an array of 40 floats
for(i=0; i < 5; i++) {
   arr[i] = i;
   rates[i] = (arr[i]*1.387)/4;
}
```



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

Array Characteristics

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

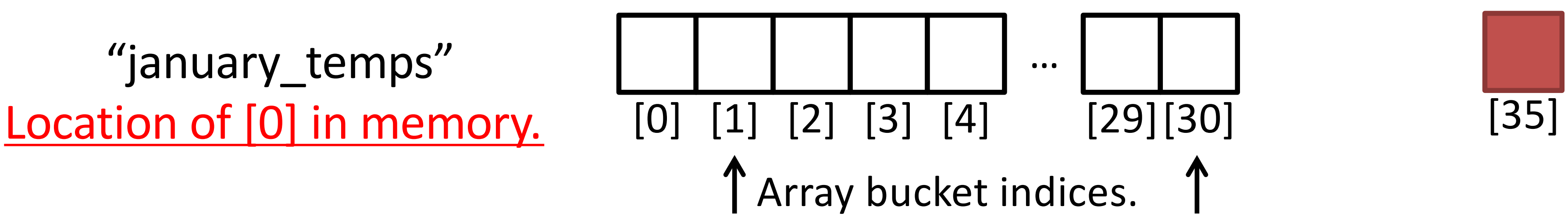


- Indices start at 0! Why? Computing the offset from the start of the array

january_temps" (without brackets!) <u>Location of [0] in memory.</u>

Array Characteristics

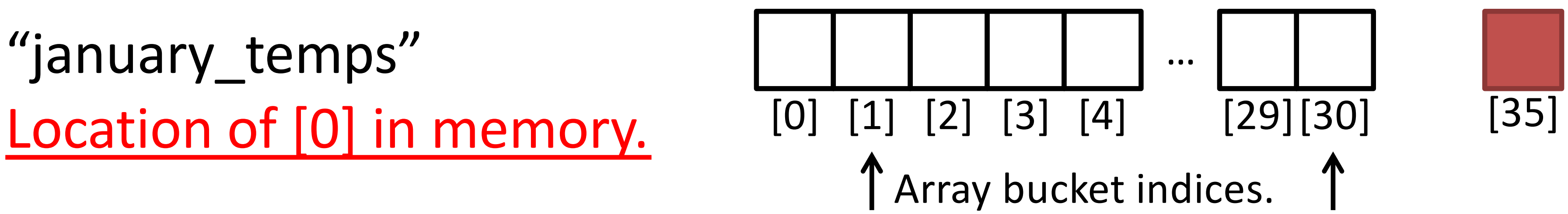
```
int january_temps[31];  // Daily high temps
```



- Indices start at 0! Why?
- Array variable name means, to the compiler, the beginning of the memory chunk. (address)
- Keep this in mind, we'll return to it soon (functions).
- Index number is an offset from beginning.

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
```



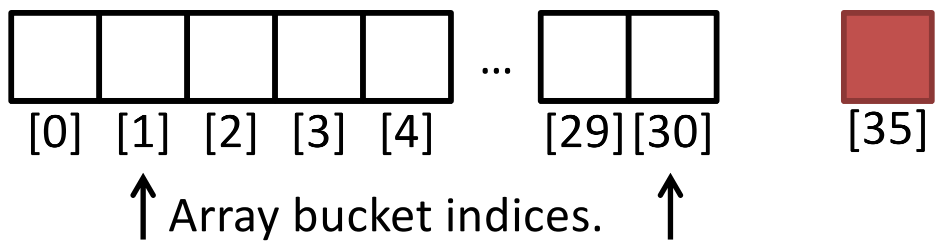
1. scanf(“%d”, january_temps);
2. scanf(“%d”, &january_temps[1]);
3. None of the above

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.



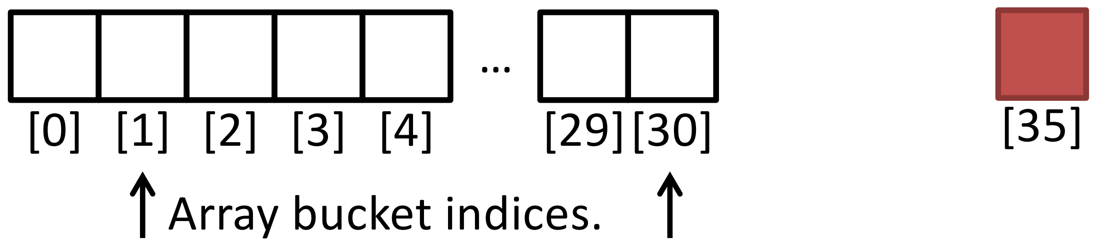
1. scanf(“%d”, january_temps);
2. scanf(“%d”, &january_temps[1]);
3. None of the above

Array Characteristics

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

“january_temps”

Location of [0] in memory.



- Array variable name means, to the compiler, the beginning of the memory chunk. (address)
- Index number is an offset from beginning.

C does NOT do bounds checking. Asking for january_temps[35]?

- Python: error
- C: “Sure! I don’t care ”

Characters and Strings

A character (type `char`) is numerical value that holds one letter.

char my_letter = ‘w’;   // Note: single quotes

What is the numerical value?

- printf(“%d   %c”, my_letter, my_letter);
- Would print: 119   w

Why is ‘w’ equal to 119?

- ASCII Standard says so.
- American Standard Code for Information Interchange

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Characters and Strings

$ man ascii

119 = w

Characters and Strings

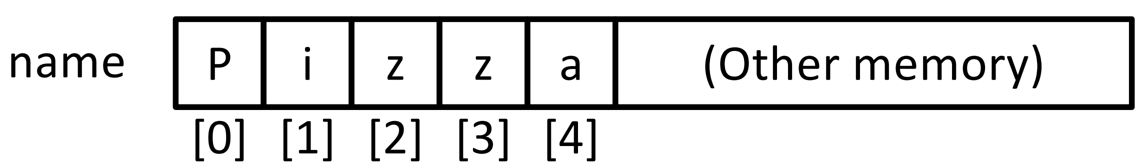
- A character (type `char`) is numerical value that holds one letter.
- A string is a memory block containing characters, one after another…

Hmm, suppose we used printf and %s to print name.

- Examples:

char food[6] = “Pizza”;

How does it know where the string ends and other memory begins?



How can we tell where a string ends?

A. Mark the end of the string with a special character.

B. Associate a length value with the string, and use that to store its current length.

C. A string is always the full length of the array it's contained within (e.g., `char name[20]` must be of length 20).

D. All of these could work (which is best?).

E. Some other mechanism (such as?).

How can we tell where a string ends?

A. Mark the end of the string with a special character. (what we do in C)

B. Associate a length value with the string, and use that to store its current length.

C. A string is always the full length of the array it's contained within (e.g., `char name[20]` must be of length 20) –
   - inconvenient and inflexible

D. <u>All of these could work (technically true)</u>

E. Some other mechanism (such as?).

Characters and Strings

0 = Null

Special stuff over here in the lower values.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Characters and Strings

- A character (type `char`) is numerical value that holds one letter.
- A string is a memory block containing characters, one after another, with a null terminator (numerical 0) at the end.
- Examples:

  char name[20] = “Pizza”;

P	i	z	z	a	\0			…		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]		[18]	[19]

Strings in C

- C String library functions: `#include <string.h>`
  - Common functions (strlen, strcpy, etc.) make strings easier
  - Less friendly than Python strings

- More on strings later, in labs.

- For now, remember about strings:
  - Allocate enough space for null terminator!
  - If you're modifying a character array (string), don't forget to set the null terminator!
  - If you see crazy, unpredictable behavior with strings, check these two things!

Functions: Specifying Types

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!

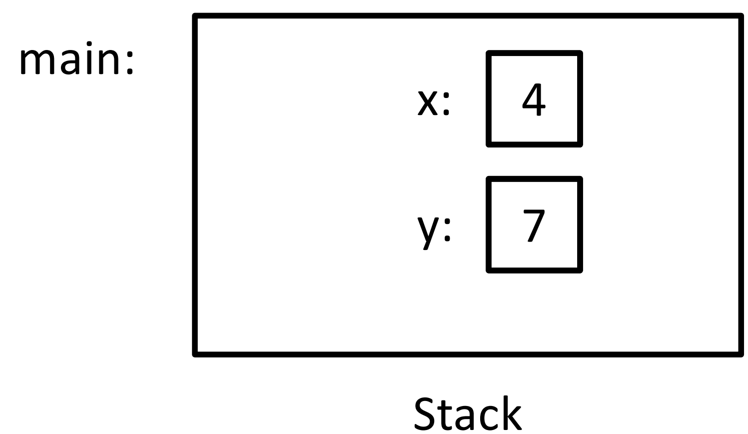
Function Arguments

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);
}
```



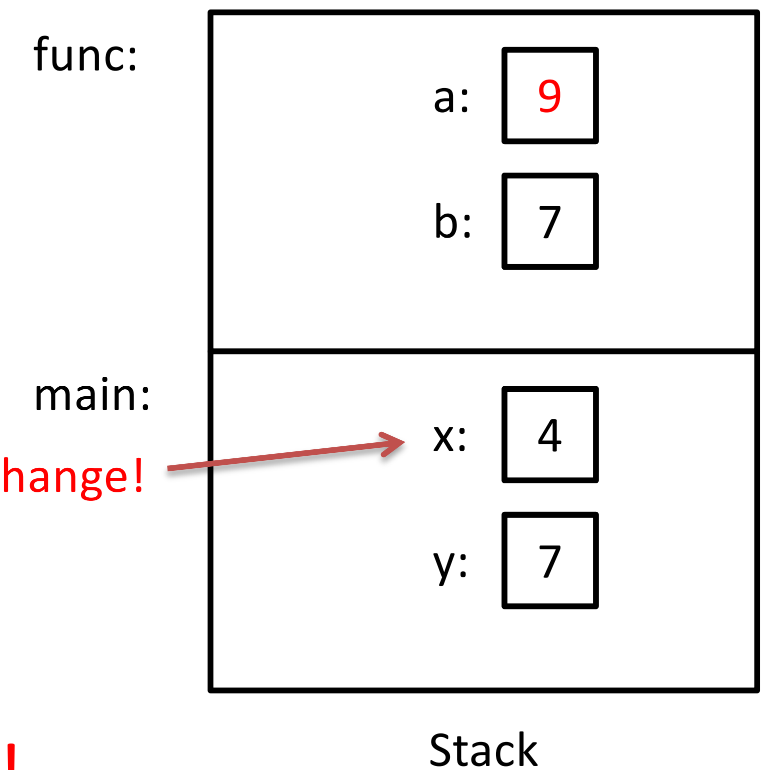
Function Arguments

- 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);
}
```

Note: This doesn't change!



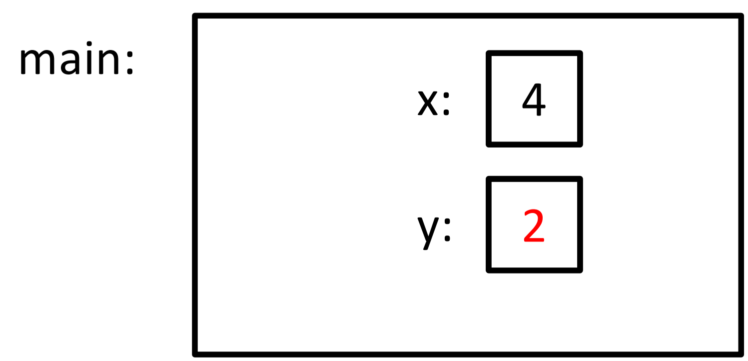
no impact on values in main!

Function Arguments

- Arguments are **passed by value**
  - The function gets a separate copy of the passed variable

```c
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);
}
```



Stack

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;
    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. <u>1, 0, 8</u>
D. 1, 5, 8
E. 1, 5, 10

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

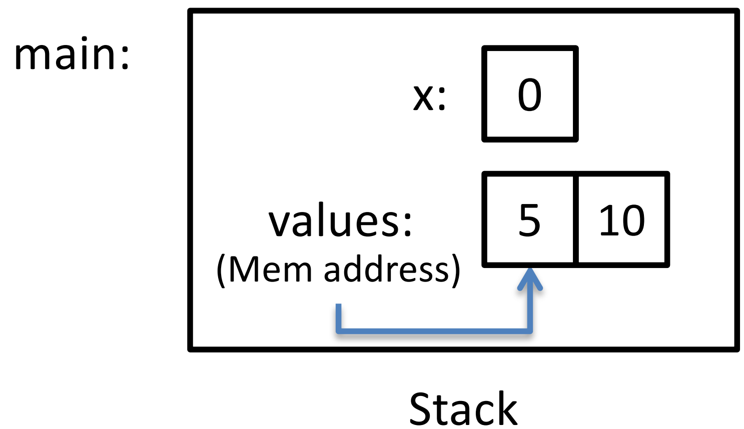
What will this print?

```
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]);
}
```



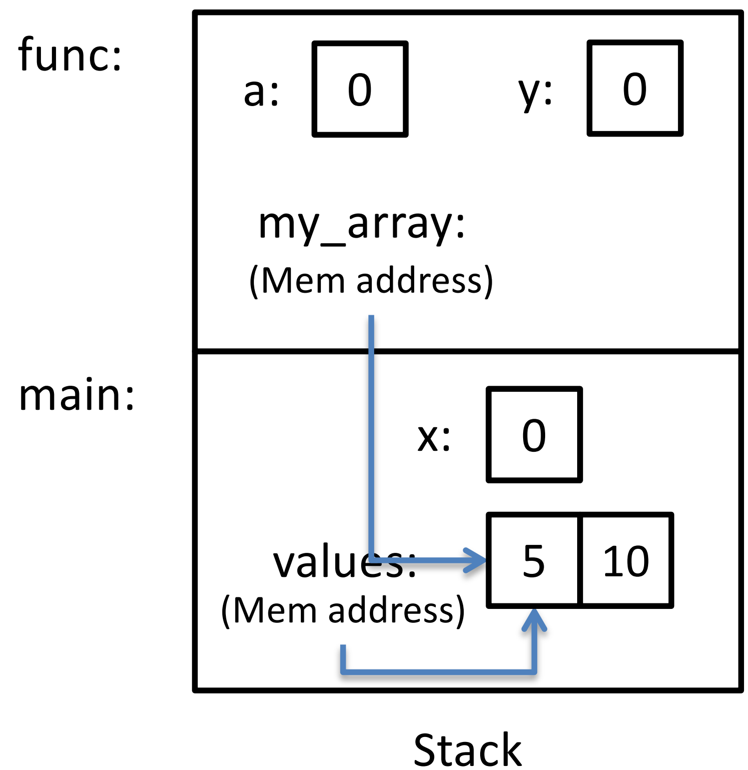
What will this print?

```
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]);
}
```



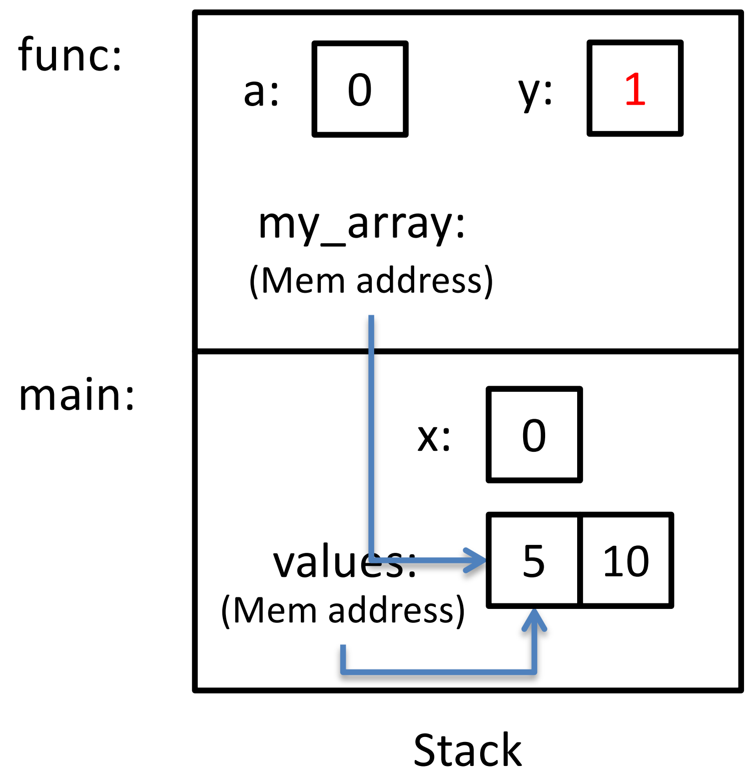
What will this print?

```
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]);
}
```



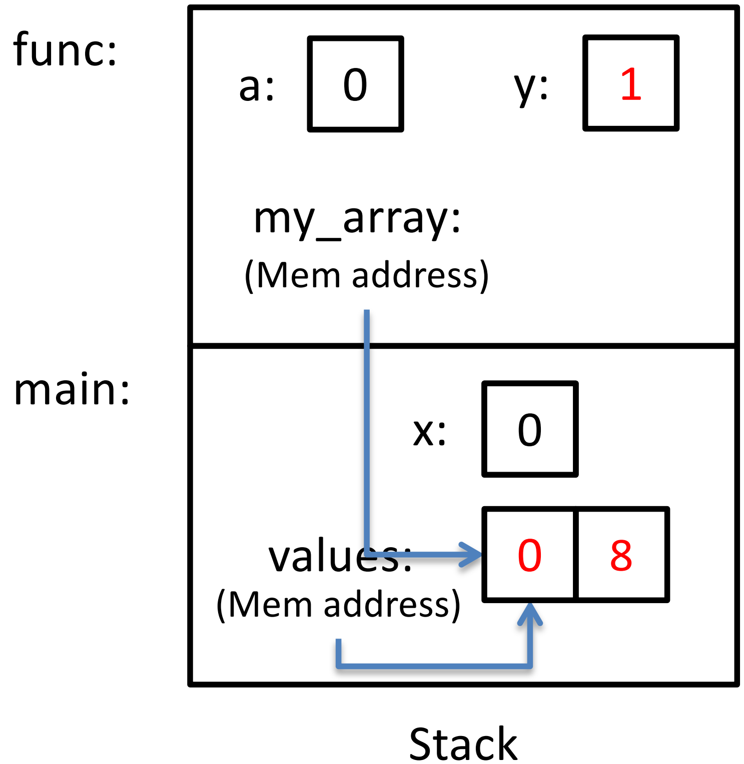
What will this print?

```
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]);
}
```



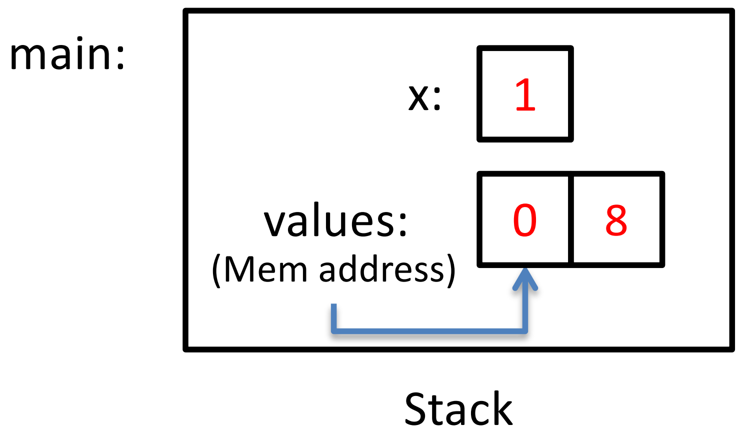
What will this print?

```
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]);
}
```



structs

- Treat a collection of values as a single type:
  - C is not an object oriented language, no classes
  - A `struct` is like just the data part of a class

- Rules:
  1. Define a new struct type outside of any function
  2. Declare variables of the new struct type
  3. Use dot notation to access the different field values of the struct variable

Struct Example

Suppose we want to represent a *student* type.

```
struct student {
    char name[20];
    int grad_year;
    float gpa;
};
// Variable bob is of type struct student
struct student  bob;
// Set name (string) with strcpy()
strcpy(bob.name, “Robert Paulson”);
bob.grad_year = 2019;
bob.gpa = 3.1;

printf(“Name: %s, year: %d, GPA: %f”, bob.name,
bob.grad_year, bob.gpa);
```

Arrays of Structs

```
struct student {
    char name[20];
    int grad_year;
    float gpa;
};
//create an array of struct students!
struct student classroom[50];

strcpy(classroom[0].name, “Alice”);
classroom[0].grad_year = 2014
classroom[0].gpa = 4.0;

// With a loop, create an army of Alice clones!
int i;
for (i = 0; i < 50; i++) {
    strcpy(classroom[i].name, “Alice”);
    classroom[i].grad_year = 2014;
    classroom[i].gpa = 4.0;
}
```

Arrays of Structs

```
struct student classroom[50];

strcpy(classroom[0].name, “Alice”);
classroom[0].grad_year = 2019;
classroom[0].gpa = 4.0;

strcpy(classroom[1].name, “Bob”);
classroom[1].grad_year = 2020;
classroom[1].gpa = 3.1

strcpy(classroom[2].name, “Cat”);
classroom[2].grad_year = 2021;
classroom[2].gpa = 3.4
```

Struct: Layout in Memory

`classroom:`

'A'	'l'	'i'	'c'	'e'	'\0'	…	'B'	'o'	'b'	'\0'	…	'C'	'a'	't'	'\0'	…
2019							2020					2021				
4.0							3.1					3.4				

[0] [1] [2]

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.

Up next…

- Digital circuits