C Programming

9/8/2016

Announcements

- Written homework #1 is due by 4pm tomorrow.
- Figure out your partner for future labs by Tuesday. There will be a Piazza post with more information.

From Tuesday: 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).

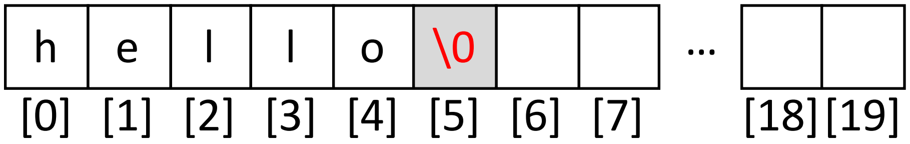
D. All of these could work (which is best?).

E. Some other mechanism (such as?).

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 message[20] = “hello”;
```



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!

Python vs. C: 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
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)

“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.

These are 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();}
```

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.

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;
humidity = 20;
```

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

What values will we see for x, y, and z?

```
/* a multiline comment:
   anything between slash-star and star-slash */

#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 & result type depends on operand types:
  - 2 int ops: 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;

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

- 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**

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:

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

With optional else:

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

Chaining if-else if

```
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
}
```

With optional else:

```
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

For loops

```
for (<init>; <cond>; <update>) {
   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 <update>
5. Goto step (2)

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

While Loops

- Basically identical to Python while loops:

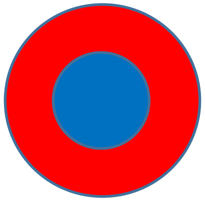
```
while (<boolean expr>) {
    while-expr-true-body
}
```

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

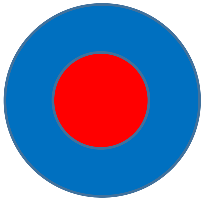
```
x = 20;
while(1) {  // while true
  y = y + x;
  x += 4;
  if(x >= 100) {
     break;  // break out of loop
  }
}
<next stmt after loop>;
```

What can/can’t while loops and for loops do in C?

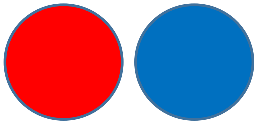
a) Anything you can compute with a C **while** loop (and more) can be computed with a C **for** loop.



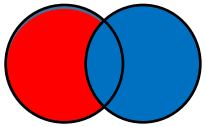
b) Anything you can compute with a C **for** loop (and more) can be computed with a C **while** loop.



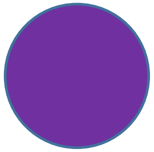
c) C’s **while** loops and **for** loops can perform completely disjoint sets of computations.



d) C’s **while** loops and **for** loops can perform partially overlapping sets of computations.



e) C’s **while** loops and **for** loops can perform exactly the same computations.



while loop → for loop

```
while(condition){
  body;
}
```

```
for(;condition;){
  body;
}
```

for loop → while loop

```
for(init; cond;
    update){
  body;
}
```

```
init;
while(cond){
  body;
  update;
}
```

Data Collections in C

- Many complex data types out there (CS 35)

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

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

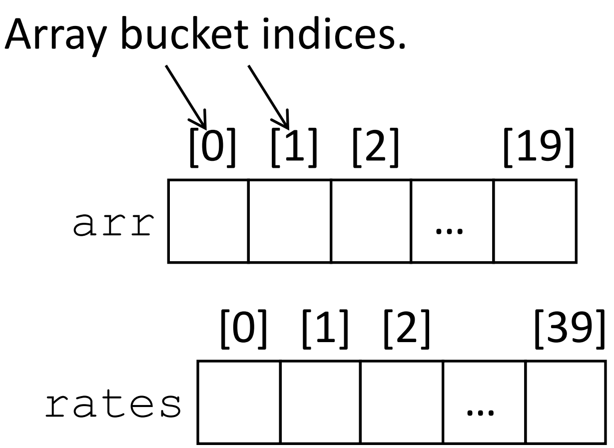
Arrays

- C’s support for lists of values
  - Array buckets store a single type of value
  - Need to specify the full capacity (num buckets) when you declare an array variable

```
<type> <var_name>[<num buckets>];
int arr[20];  // declare array of 20 ints
float rates[40]; // an array of 40 floats
```

Buckets often accessed w/loop:

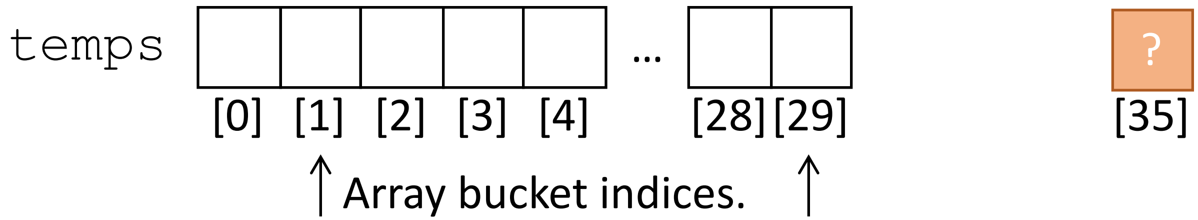
```
for(i=0; i < 20; i++) {
   arr[i] = i;
   rates[i] = (arr[i]*1.387)/4;
}
```



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

Array Characteristics

```
int temps[30];
```



- Name of array variable alone (`temps`) evaluates to the the beginning of the memory chunk (base address)

  (remember this for later)

  "`temps`": location of bucket [0] in memory
- Indices start at 0! Why?
  - Index number is an offset from beginning of array
  - `temps[0]`: int at offset 0 from base addr of array
- C does <u>NOT</u> do bounds checking.
  - Ask for `temps[35]`?
    - Python: error
    - C: okey dokey

Demo!

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

Need to Think about Type!

- Suppose we want to represent a *student* type.

```
struct student {
        char name[20];
        int grad_year;
        float gpa;
};

struct student jo;
```

jo:

field names	stored values (memory space)
name:	| | | | | | …
grad_year:	
gpa:	

What type is: `jo`

`jo.gpa`

`jo.name[3]`

`jo.name`

`jo:`	struct student
`jo.gpa:`	float
`jo.name[3]:`	char
`jo.name:`	array of char

Struct Example

```
// 1. define a new struct type (outside a function):
struct student {     // struct <name> {
         char name[20];  // <type> <field name>;
         int grad_year;
         float gpa;
};
```

field names: stored values (memory space)

jo:

name:	‘J’	‘o’	‘ ’	‘S’	‘c’	‘h’	…
grad _year:	2017						
gpa:	3.1						

```
// 2. Declare var of new type:
int main() {
  struct student jo; // jo’s type is “struct student”

  // 3. Use dot notation to access field values:
  strcpy(jo.name, “Jo Schmoe”);  // Set name with strcpy()
  jo.grad_year = 2016;
  jo.gpa = 3.1;
  printf(“Name: %s, year: %d, GPA: %f”,
         jo.name, jo.grad_year, jo.gpa);
}
```

Functions: example from lab 2

```
void open_file_and_check(char *filename);

int main (int argc, char *argv[]) {
  //...
}

void open_file_and_check(char *filename){
  int ret;
  ret = open_file(filename
  if(ret == -1) {
    printf("bad error: can't open %s\n",
           filename);
    exit(1);
  }
}
```

Declaration

Definition

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;
}
```

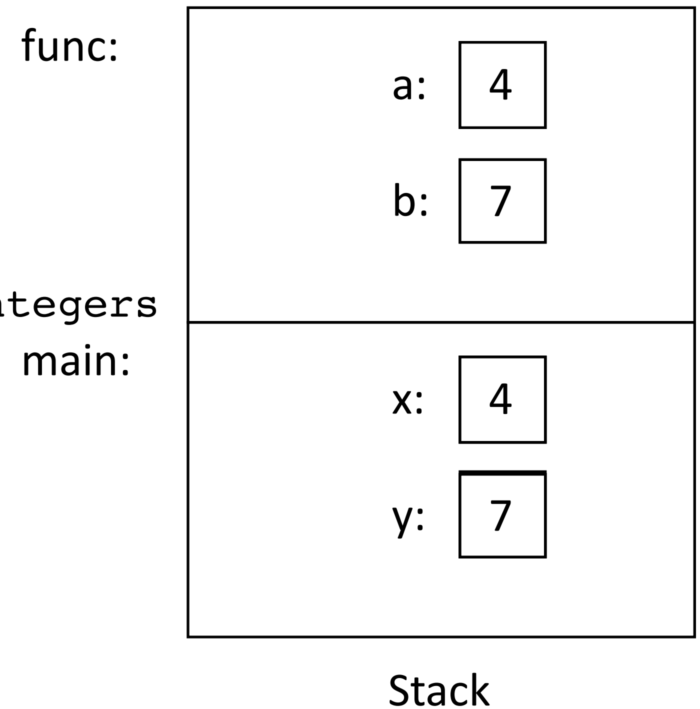
Compiler will yell at you if you try to pass the wrong type!

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



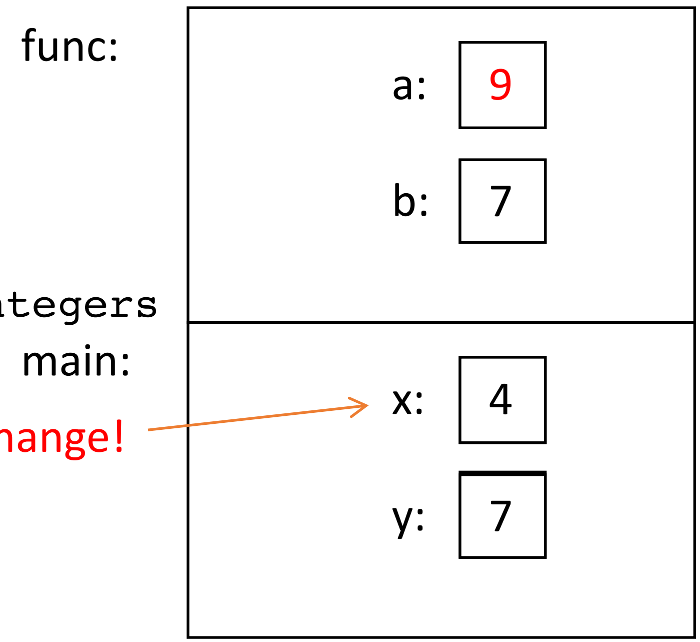
Function Arguments

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

Note: This doesn't change!



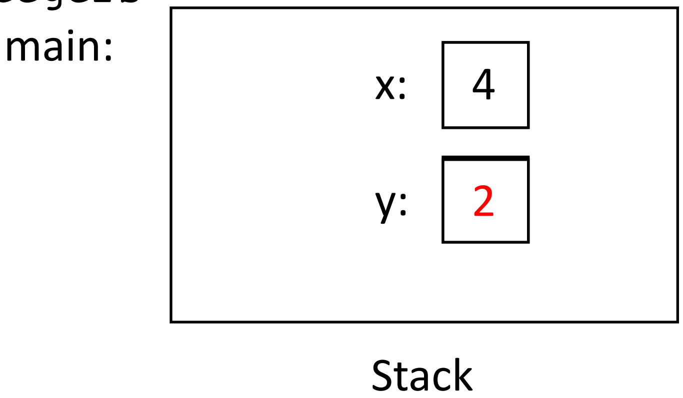
Stack

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



Output: 4, 2

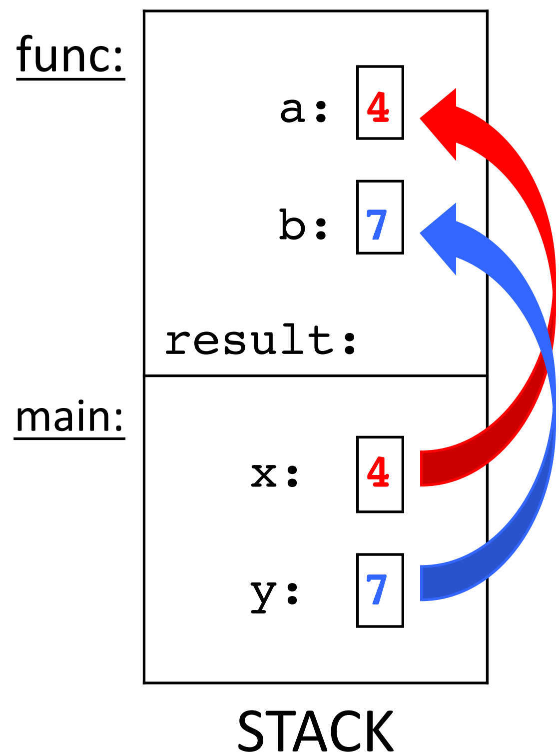
The Stack and Pass by Value

```
// function prototype:
// declares function
int func(int a, int b);

int main() {
   int x, y;
   x = 4;
   y = 7;
   y = func(x, y);
   printf("%d %d\n", x, y);
}

// function definition:
// implementation of func
int func(int a, int b) {
  a = a + 5;
  return a - b;
}
```

- Local variables and parameters:
  - storage locations allocated on the stack
  - store values of defined type **on the stack**



Arguments are Passed By Value: parameter gets a copy of its argument's value

Modifying a parameter CANNOT change its argument's value

What will this print?

```
int func(int a, int y, int arr[]) {
          y = 1;
          arr[a] = 0;
          arr[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?

Arrays and Functions

- Array Parameters:
    - Specify the type, but not the exact size of the array
        - makes the function more generic--works for any size array
    - Need to pass the size of the array (not nec. capacity)
        - the number of buckets in use to the function
    - Function Call takes the name of the array (base address)

```
void printArray(int a[], int n) {
    int I;
    for(i=0; I < n; i++) {
        printf("a[%d] = %d\n", i, a[i]);
    }
}
int main() {
   int array[20], list[40];
   ...
   printArray(array, 20);
   printArray(list, 40);
```

Passing Arrays is Pass by Value

- A parameter always gets the value of its argument
  - The value of an array argument is its base address

    Array name == memory location (the address of) its 0th bucket
  - Array parameter REFERS TO same array storage as its argument

    ➔ changing a bucket value in a function changes the corresponding bucket value in its argument

```
void test(int a[], int n) {
    a[3] = 8;
    n = 3;
}
int main() {
   int i, array[5],;
   for(i=0; i<5; i++) {
      array[i] = i;
   }
   test(array, 5);
   printf("%d", array[3]);
}
```

Pass by Value: Array Arguments

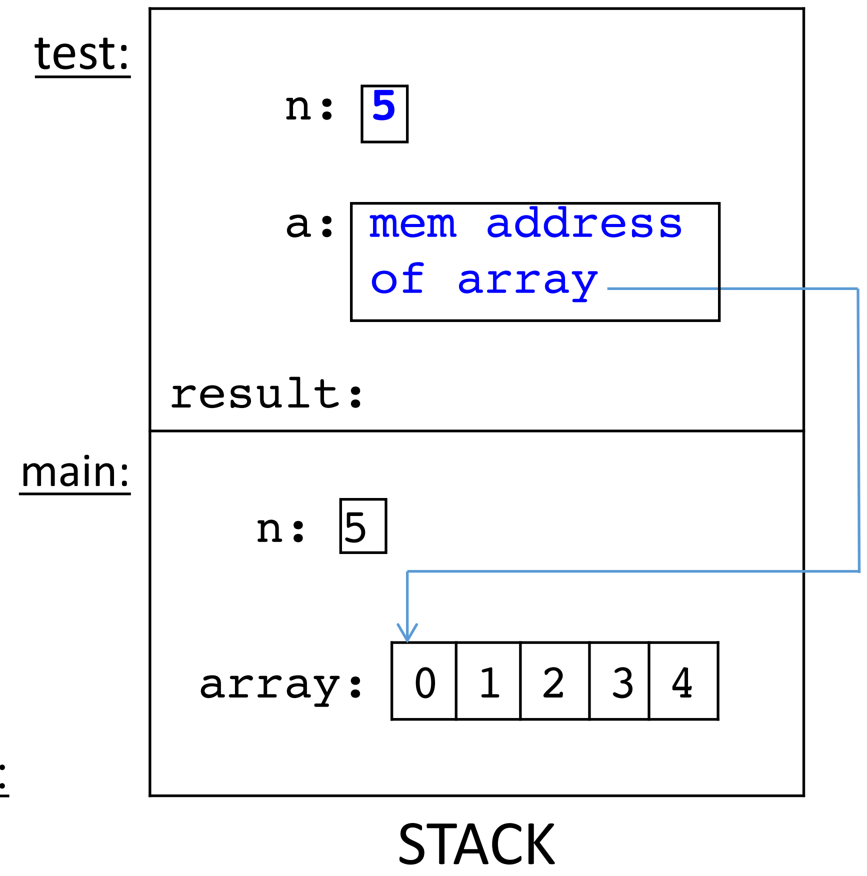
```
void test(int a[], int n) {
     a[3] = 8;
     n = 3;
}

int main() {
   int array[5], n = 5;
   for(i=0; i<n; i++) {
       array[i] = i;
   }
   test(array, n);
   printf("%d", array[3]);
}
```

The values of the arguments are passed:

n: 5

array: memory location (the address) of the start of array

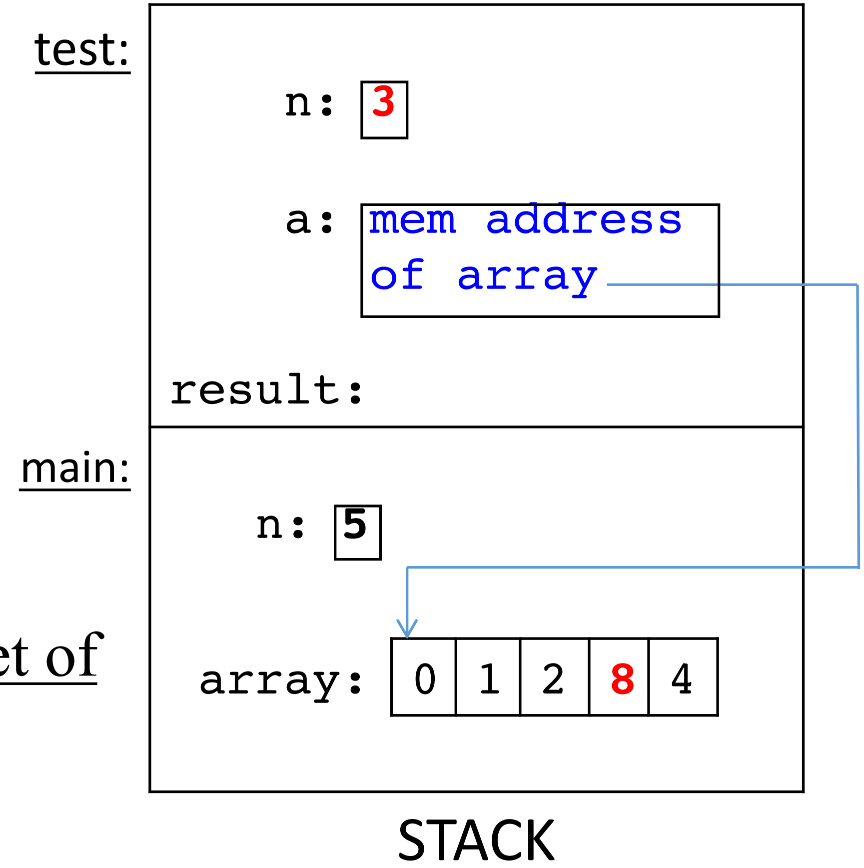


Pass by Value: Array Arguments

```
void test(int a[], int n){
     a[3] = 8;
     n = 3;
}
int main() {
   int array[5], n = 5;
   for(i=0; i<n; i++) {
       array[i] = i;
   }
   test(array, n);
}
```

<u>Changing value stored in bucket of an array parameter (a[3] = 8), changes value stored in corr. argument (array[3]):</u>

a and **array** refer to the same memory location



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.