C Basics for CS31 Students

Hello World

```
# hello world
import math

def main():
   print “hello world”

main()
```

```
// 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)

Need to Declare Variables in C

Variables must be declared before used,
and their type is fixed for duration of program.

- Where? At beginning of a block, before C stmts
- How? <variable type> <variable name>

```
int x;        // declare an int variable named x

float y, z;   // y and z are floats

char ch;      // ch stores a single char value (ascii value)

// then can use variables in C expressions:

x = 6 + 10;
y = 13.2;
z = (y*7)/3;
ch = ‘a’;   // char literal is between single quotes
            // it is stored as ascii value of ‘a’
            // A CHAR IS NOT A STRING IN C
```

A program with local variables

```
/* a multiline comment:
   anything between slashdot and dotslash
 */
#include <stdio.h> // C’s standard I/O library (for printf)

int main() {
   int x, y;   // first: declare main’s local variables
   float z;
   char ch;
              // followed by: main function statements
   x = 6;
   y = (x + 3)/2;
   z = x;
   z = (z + 3)/2;
   ch = ‘a’;
   printf(“%d %d %f %c\n”, x, y, z, ch+1);
}

// The program’s output:  6  4  4.5  b
// Do you understand Why?
```

printf function

- Similar to Python's formatted print statement:

```
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**

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

Boolean values in C

- There is no boolean type in C, instead **int expressions** used in conditional statements are interpreted as true or false according to this rule:

0: is false   non-zero value: is true

ex:

```
int x, y;
x = 4;
y = -10;
```

if (x < y)	4 < -10	is false
if ((x+3) > y)	(4+7) > -10	is true
if (y)	-10	is true
if (0)	0	is false

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)

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 “boolean”):
    - <, <=, >, >=, ==, !=

    - 6 != (4+2)   is 0 (false)
    - 6 > 3   some non-zero value (we don’t care wch one) (true)

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

    - Evaluate: (8 > 13) || !(4 < 7)
    - Evaluate: (((20 % 3) < (7-5)) && !(2/3))

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>;
```

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

For loops: different than Python’s

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

1. Evaluate <init> one time, when first eval **for** statement
2. Evaluate <cond>, if it is false, drop out of the loop (<next stmt>)
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);
}
// print out the odd values between 1 and 100?
```

Functions: need to specify types

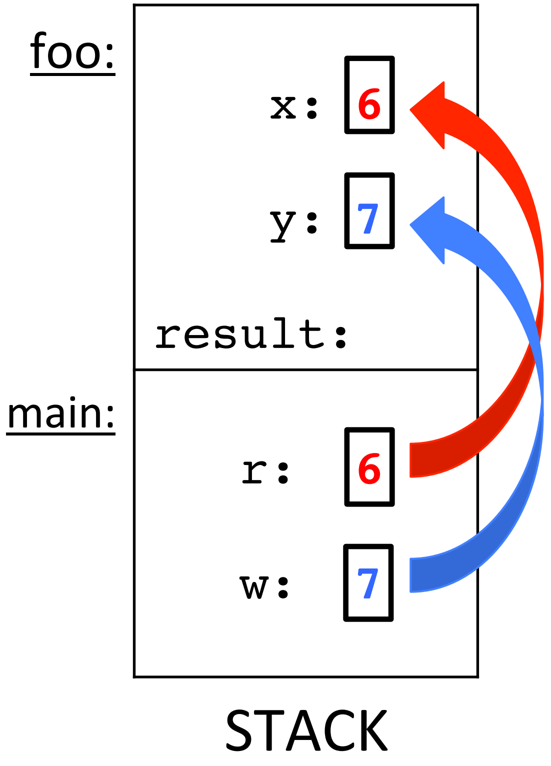
- Need to specify the return type of the function, and the type of each parameter:

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

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

The Stack and Pass by Value

```
// function prototype:
int foo(int x, int y);

int main() {
   int r, w;
   r = 6;  w = 7;
   r = foo(r, w);
}

// function definition:
int foo(int x, int y) {
  int result;
  result = x;
  if(y > x) {
    result = y + 5;
  }
  return result;
}

//TRY: write power function
```

- Local variables and parameters:
  - storage locations allocated on the stack
  - store values of defined type **<u>on the stack</u>**
- Value of arguments are copied to parameter storage



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

Modifying a parameter CANNOT change its argument's value

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];  // an array of 20 ints
float rates[40]; // an array of 40 floats
```

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

Arrays and Functions

- Array Parameters and Arguments:
  - Specify the type, but not the exact size of the array (this makes the function more generic--works for any size array)
  - Need to pass the size of the array (the number of buckets in use) or the capacity to the function too
  - Function Call takes the name of the array

```
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

- 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
  - The parameter REFERS TO the 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]);
```

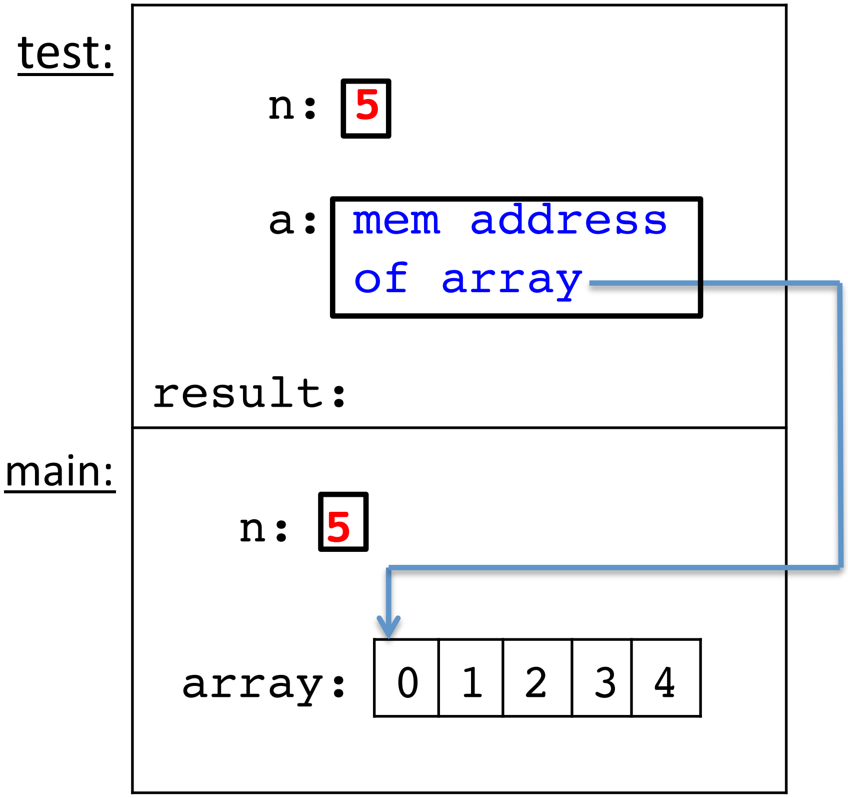
Pass by Value: Array Arguments

```
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



test:

n: 5

a: mem address of array

result:

main:

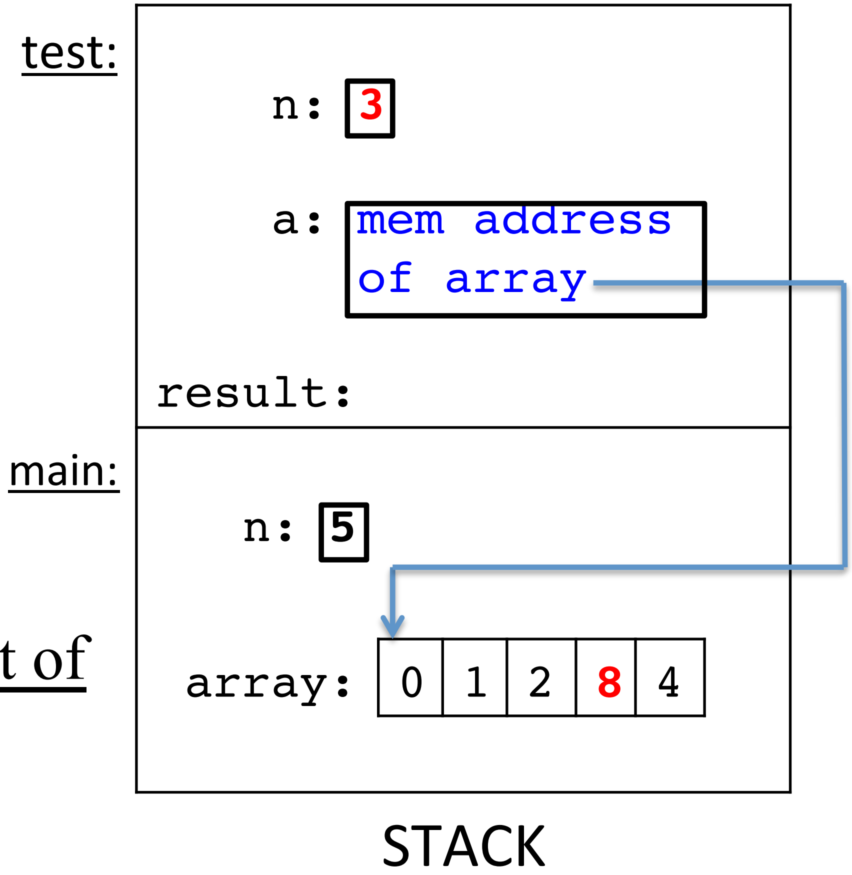
n: 5

array: 0 1 2 3 4

STACK

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);
}
```



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

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

Try Out:

1. Implement a Function to swap two array bucket values
   1. Function Interface:

      inputs values? → Swap Func → return value?

      how many? type? type?
   2. Write up function prototype
   3. Implement function body
   4. Add function call to main
2. Step through call drawing the stack

Strings in C

- An array of character values

  `char str[20];  // array of 20 chars`
  `               // need to specify capacity`

- Special end of string char: ‘\0’
- C string library: #include<string.h>
  - Functions to manipulate strings
  - User MUST ALLOCATE SPACE for string (array of char)
  - Library functions use ‘\0’ to find end of string
    - Don’t need to pass in size of string to function like you need to pass in size of array to function

C string examples

```
char str1[20], str2[40];
int val;

// remember to null terminate strings:
str1[0]=‘T’; str1[1]=‘i’; str1[2]=‘a’; str1[3]=‘\0’;

// some string library funcs do null termination
// for you:
strcpy(str2, “hello”);

// prints: hello Tia 3
printf(“%s %s %d\n”, str2, str1, strlen(str1));

// compare’s ascii values of coorsp chars in str1
// and str2, returns: 0 if equal, neg if str1< str2,
// pos if str1 > str2
val = strcmp(str1, str2);
If(val) { printf(“str1 and str2 are not equal\n”);
```

C string pitfalls

```
char str1[20], str2[5];
int val;

// (1) forgetting to null terminate strings:
str1[0]=‘T’;
str1[1]=‘i’;
str1[2]=‘a’;
val = strlen(str1);  // Likely something larger than 3
printf(“%s”, str1);  // Likely print out Tia followed
                     // by some garbage chars

// (2) forgetting to allocate enough space for the
//     terminating null char at end of string:
strcpy(str2, “hello”);
```

strcpy will write ‘\0’ one char beyond the end of str2 array

This kind of error can result in odd program crashes or weird, hard to explain program behavior

structs

- Way to treat a collection of values as a single whole/type:
  - C is not an objet oriented language, so 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

```
#include <stdio.h>
#define MAXNAME  64   // a constant definition

struct studentT {         // define a new struct type
    char name[MAXNAME];   // type1 field1name;
    int   age;            // type2 field2name; …
    float gpa;
};                        // don’t forget the ;

int main() {
   struct studentT jo, flo;  // declare variables

   jo.age = 18;              // use dot notation to
   jo.gpa = 3.5;             // access fields
   strcpy(jo.name, “Jo”);
   flo = jo;                // structs are lvalues
   strcpy(flo.name, “Flo”);
}
```

Let’s trace through this code

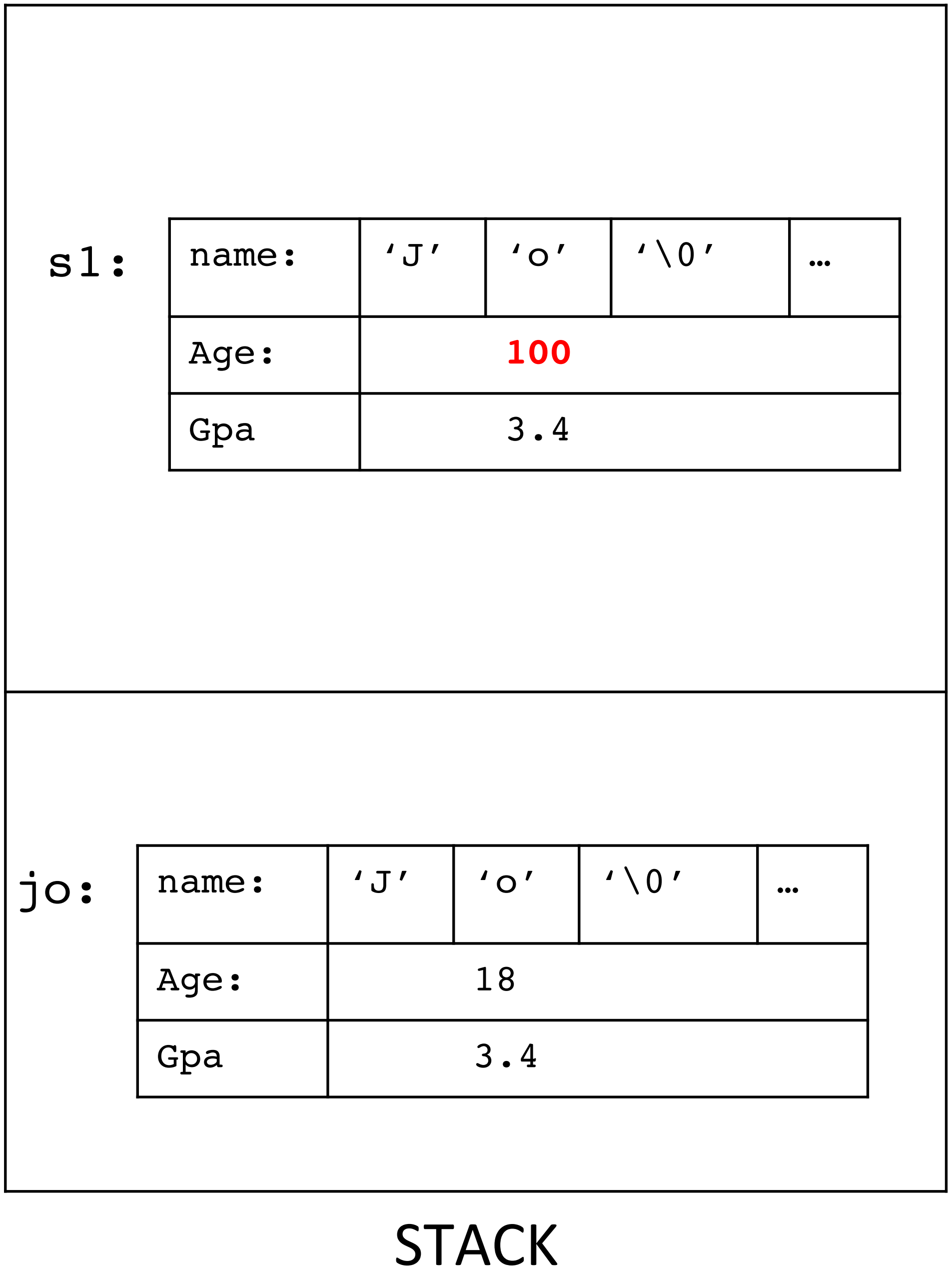
```
void crazy(struct studentT s1)
{
     s1.age = 100;
}

int main() {
    struct studentT jo, flo;
    …
    crazy(jo);
}
```

Pass by value:

param gets copy of argument's value

changing parm DOESN'T modify argument's value

crazy:



s1:

name:	'J'	'o'	'\0'	…
Age:	100			
Gpa	3.4			

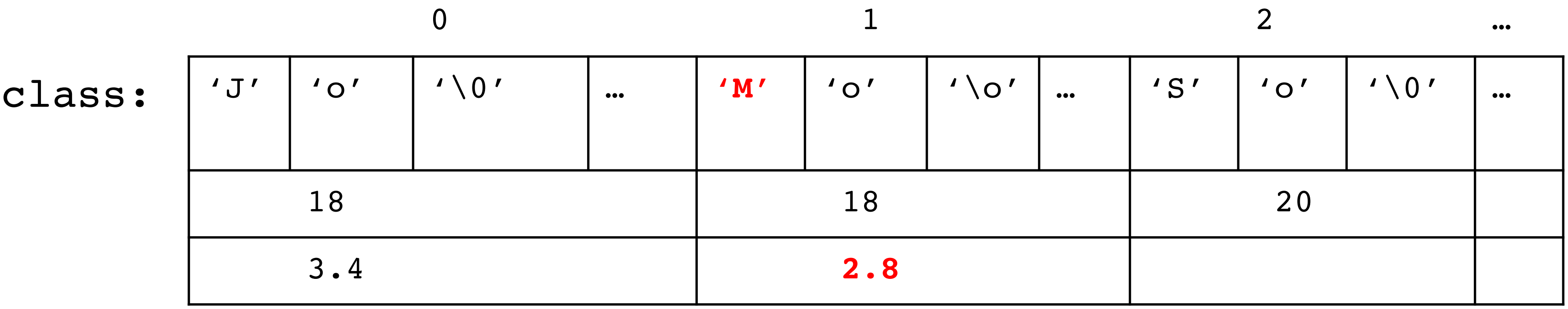
main:

jo:

name:	'J'	'o'	'\0'	…
Age:	18			
Gpa	3.4			

Arrays of structs…think about type!

```
int main() {
   struct studentT class[50];

   strcpy(class[0].name,”Jo”);
   class[0].age = 18;
   class[0].gpa = 3.4;
   class[1] = class[0];  // structs are lvalues
   class[1].name[0] = ‘M’;
   class[1].gpa = 2.8;
   strcpy(class[2].name,”So”);
   class[2].age = 20;
}
```



	0				1				2			…
class:	‘J’	‘o’	‘\0’	…	‘M’	‘o’	‘\o’	…	‘S’	‘o’	‘\0’	…
	18				18				20			
	3.4				2.8							

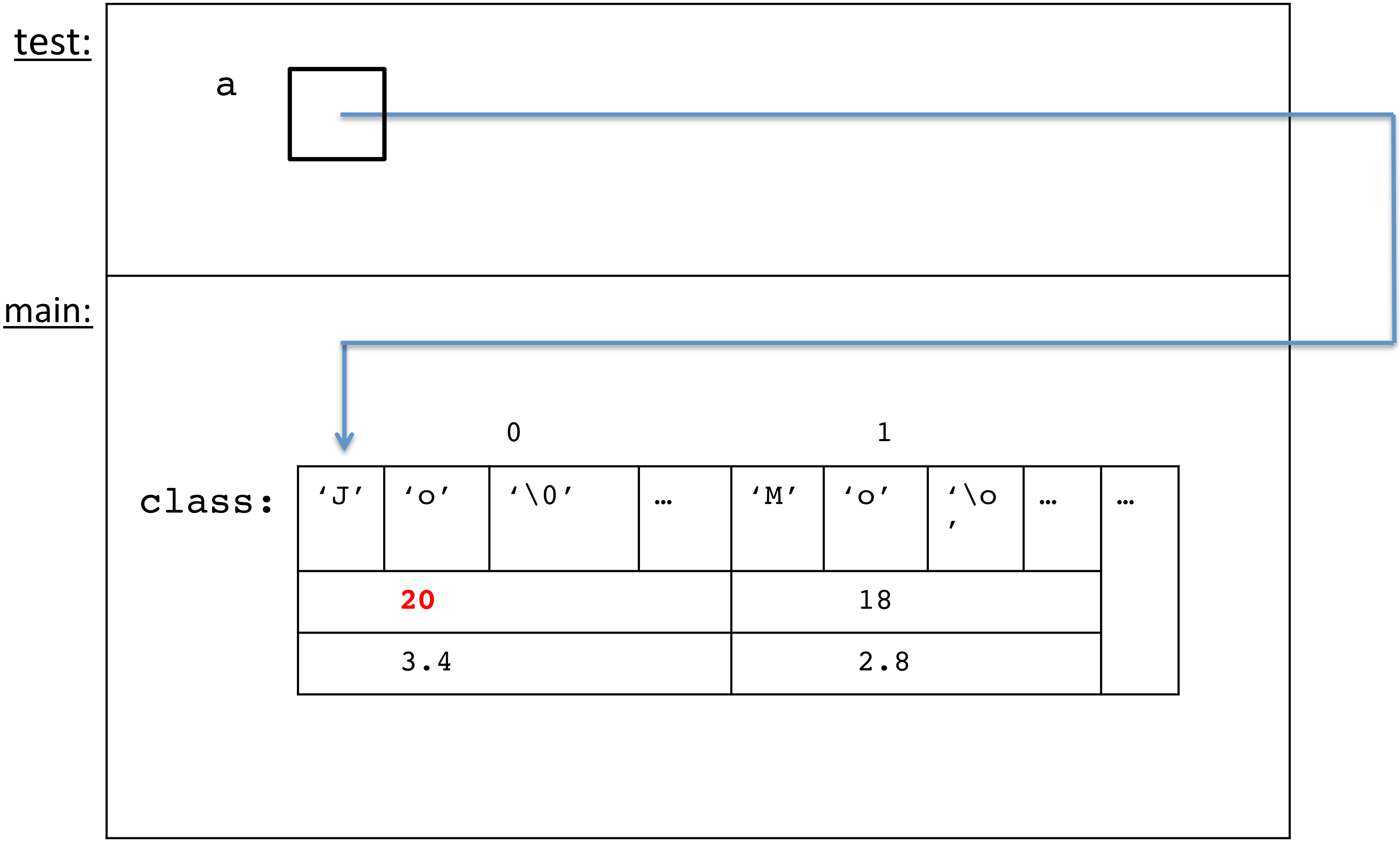
Arrays of structs parameters:

```
void test(struct studentT a[], int n) {
   a[0].age = 20;
}

int main() {

   struct studentT class[50];
   …
   test(class, 3);
}
```

Changing value stored in bucket of an array parameter (a[0].age = 20), changes the corresponding bucket value in argument (class[0].age):

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

Arrays of structs parameters:



test:

a

main:

class:

0				1				
‘J’	‘o’	‘\0’	…	‘M’	‘o’	‘\o ’	…	…
20				18				
3.4				2.8				

STACK

scanf

- For reading in values of different types
- Uses format string like printf
- The arguments are the memory locations into which the values will be stored (the address of program variables or base addr of arrays):

```
int x;
float y;
char s[100];

scanf(“%s%d%f”, s, &x, &y);

// s is the base address of the string array
// &x is the address of the variable x in memory
// &y is the address of the variable y in memory
```