

Review Qs: Refer to handout

1. What value is at address  $0x16fdceff0$ ?
2. What value is at address  $0x16fdceff8$ ? (note:  $\text{sizeof}(\text{double}) = 8$ )
3. What is surprising about `res1` and `res2`?

1.
  - 8-byte double representing 1.3
  - 1.3
  - the first byte of the 8 byte double for 1.3
  - the locally-declared array `vec1`
2. double for 4.2; the first byte of the double 4.2; the second element of `vec1`
3. `res1` and `res2` print as the same address  
`res1[0]` and `res2[0]` are both 9333  
 (expect `res1[0]` to be  $1.3 + 1.5 = 1.8$ )

AH

```

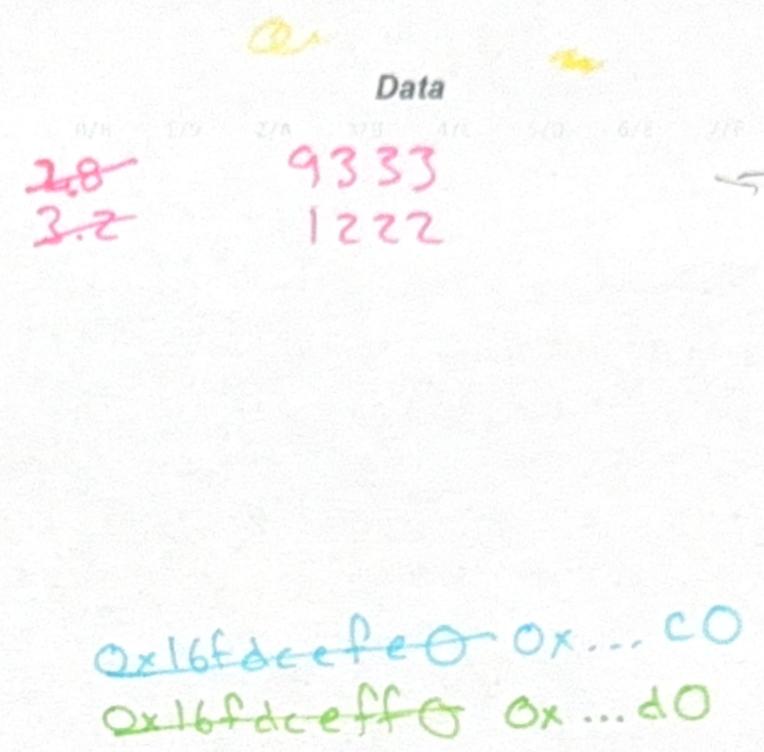
1 #include <stdio.h>
2
3 // vector_sum: takes two same-length vectors (double[])
4 // adds them together component-wise in a new array
5 // vector_sum({ 1.2, 3.4 }, {-1.0, 3.6 }) => { 0.2, 7.0 }
6 // Assume the vectors have the same length
7
8 // Q: What happens if double[] is used as a return type?
9 // double[] vector_sum(double vec1[], double vec2[]);
10
11 // Q: What about using double* as return type?
12 // double* vector_sum(double vec1[], double vec2[])
13 // double res[... ] don't know length!
14 // Pass in length as an argument. Maybe now we've got it!
15 double* vector_sum(double* v1, double* v2, int len) {
16     double res[len];
17     printf("v1%p : %p\tv2%p : %p\tres: %p\n",
18           &v1, v1, &v2, v2, res);
19     for(int i = 0; i < len; i += 1) { res[i] = v1[i] + v2[i]; }
20     return res;
21 }
22 int main() {
23     double vec1[] = { 1.3, 4.2 }, vec2[] = { 1.5, -1 };
24     double* res1 = vector_sum(vec1, vec2, 2);
25
26     double vec3[] = { 333, 222 }, vec4[] = { 9000, 1000 };
27     double* res2 = vector_sum(vec3, vec4, 2);
28
29     printf("res1[0]: %f\t res2[0]: %f\n", res1[0], res2[0]);
30
31     printf("vec1: %p\n", vec1);
32     printf("vec2: %p\n", vec2);
33     printf("vec3: %p\n", vec3);
34     printf("vec4: %p\n", vec4);
35     printf("res1: %p\n", res1);
36     printf("res2: %p\n", res2);
37 }

```

Syntax error

X the warning should be an error

Variable/Role	Address
res	0x...00
	0x...08
	0x...10
	0x...18
	0x...20
	0x...28
	0x...30
	0x...38
	0x...40
v2	0x...48
v1	0x...50
	0x...58
	0x...60
	0x...68
	0x...70
	0x...78
	0x...80
	0x...88
	0x...90
	0x...98
	0x...A0
	0x...A8
res1	0x...B0
res2	0x...B8
vec4	0x...C0
	0x...C8
vec3	0x...D0
	0x...D8
vec2	0x...E0
	0x...E8
vec1	0x...F0
	0x...F8

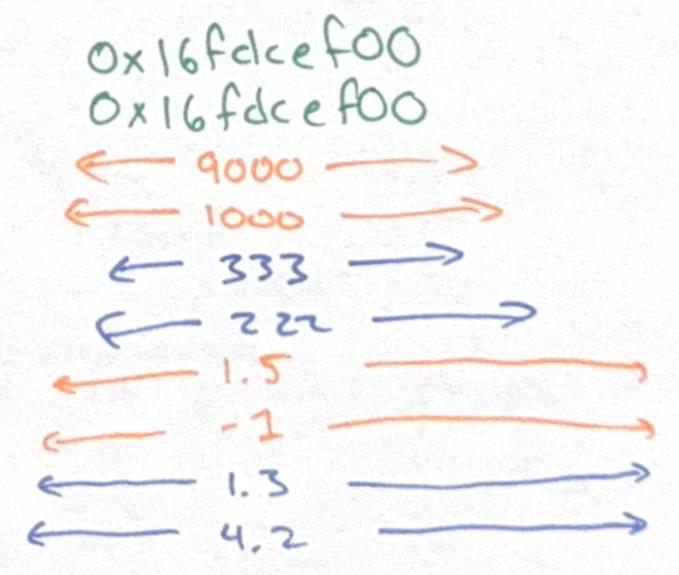


```

$ gcc vector_sum.c -o vector_sum
vector_sum.c:22:10: warning: address of stack memory associated with
local variable 'res' returned [-Wreturn-stack-address]
20 |     return res;
$ ./vector_sum
v1@0x16fdcef50 : 0x16fdceff0 v2@0x16fdcef48 : 0x16fdcefe0 res: 0x16fdcef00
v1@0x16fdcef50 : 0x16fdcefd0 v2@0x16fdcef48 : 0x16fdcefc0 res: 0x16fdcef00
res1[0]: 9333.000000 res2[0]: 9333.000000
vec1: 0x16fdceff0
vec2: 0x16fdcefe0
vec3: 0x16fdcefd0
vec4: 0x16fdcefc0
res1: 0x16fdcef00
res2: 0x16fdcef00

```

res1	0x...B0
res2	0x...B8
vec4	0x...C0
	0x...C8
vec3	0x...D0
	0x...D8
vec2	0x...E0
	0x...E8
vec1	0x...F0
	0x...F8



sizeof(x)  $\Rightarrow$  compile-time operation, not a function

(A2)

x could be

- a type: gives the # of bytes to store 1 of that type

int32\_t = 4  
double = 8  
char = 1

double\* = 8  
char\* = 8  
int32\_t\* = 8

- an expression (for example, a variable)

gives # of bytes to store 1 of the type of that expression/variable

char c = 'a';  
sizeof(c) = 1

int32\_t i = 22;  
sizeof(i) = 4

- an array variable

gives total # of bytes for the array declaration

char c[9];  
sizeof(c) = 9

double ns[5];  
sizeof(ns) = 40

char a[] = "abc";  
sizeof(a) = 4

v -> (double\* v1; ...)

v1[3]  $\Rightarrow$  look up 8 bytes at v1 + 24

(char\* s)

s[3]

look up 1 byte at s + 3

(int\* ns)

ns[3]

look up 4 bytes at ns + 12

sizeof(double)  
(char)  
(int)

3 \* sizeof(double)  
(char)  
(int)