CSE 29
Lecture 7 Summary

January 27, 2026

📣 Logistical Things

- Assignment 2 is due this Thursday (1/29), which includes:
  - PSet 2 on Prairie Learn
  - PA 2
  - Design Questions 2
  - We have gone over all the things you need to complete PA2
- Assignment 1 resubmission is also due next Thursday (1/29)
  - More details on resubmission policy: https://ucsd-cse29.github.io/wi26/#assignments

🧠 Review Questions Handout

```c
#include <stdio.h>

// vector_sum: takes two same-length vectors (double[])
// adds them together component-wise in a new array
// vector_sum({ 1.2, 3.4 }, {-1.0, 3.6 }) => { 0.2, 7.0 }
// Assume the vectors have the same length

// Q: What happens if double[] is used as a return type?
// double[] vector_sum(double vec1[], double vec2[]);

// Q: What about using double* as return type?
// double* vector_sum(double vec1[], double vec2[])

// Pass in length as an argument. Maybe now we've got it!
double* vector_sum(double* v1, double* v2, int len) {
  double res[len];
  printf("v1@%p : %p\tv2@%p : %p\tres: %p\n",
           &v1,  v1,   &v2,  v2,     res);
  for(int i = 0; i < len; i += 1) { res[i] = v1[i] + v2[i]; }
  return res;
}
int main() {
  double vec1[] = { 1.3, 4.2 }, vec2[] = { 1.5, -1 };
  double* res1 = vector_sum(vec1, vec2, 2);

  double vec3[] = { 333, 222 }, vec4[] = { 9000, 1000 };
  double* res2 = vector_sum(vec3, vec4, 2);

  printf("res1[0]: %f\t res2[0]: %f\n", res1[0], res2[0]);

  printf("vec1: %p\n", vec1);
  printf("vec2: %p\n", vec2);
  printf("vec3: %p\n", vec3);
  printf("vec4: %p\n", vec4);
  printf("res1: %p\n", res1);
  printf("res2: %p\n", res1);
}
```

```
$ 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]
   22 |   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
```

🧠 Review Questions

Answers in speaker notes!

Refer to handout

Q1: What value is at address `0x16fdceff0`?

Q2: What value is at address `0x16fdceff8`?

Q3: What is surprising about `res1` and `res2`?

Answer 1:

- 8-byte double representing 1.3
- The first element of `vec1`
- The first byte of the 8 byte double for 1.3
- The locally-declared array `vec1`

Because we have not talked about how doubles are represented, these are all valid answers

Answer 2:

- Double for 4.2
- The first byte of the double 4.2
- The second element of `vec1`

Answer 3:

- `res1` and `res2` print as the same address
- `res1[0]` and `res2[0]` are both 9333 (however, we are expecting `res1[0]` to be 1.3 + 1.5 = 2.8. Interesting 🤔)

Think: Why should we not do what's on lines 8 & 11?

```
 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
```

We would get a syntax error! We can't have an array as a return type in C!

```
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
```

Inside the function, there is no way of knowing what the length of the returning array should be! We would need an extra parameter to tells us the length.

Addresses and The Stack

Looking at the output

```
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", res1);
37 }
```

v2 is at memory address 0x…48, and it holds something at the memory address 0x…c0

v1 is at memory address 0x…50, and it holds something at the memory address 0x…f0

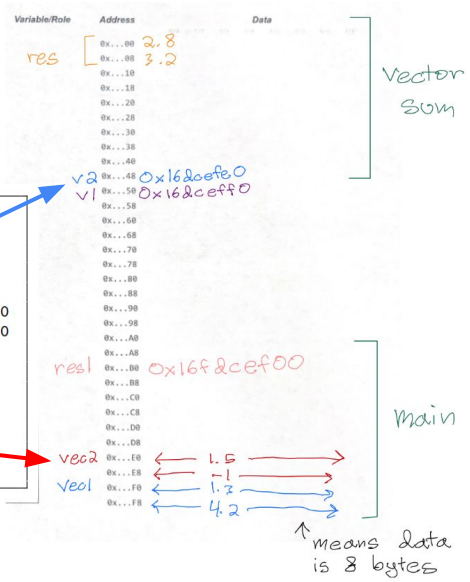
```
$ gcc vector_sum.c -o vector_sum
vector_sum.c:22:10: warning: address of stack memory associated with
    local variable 'res' returned [-Wreturn-stack-address]
   22 |   return res;
$ ./vector_sum
v1@0x16fdcef50 : 0x16fdceff0  v2@0x16fdcef48 : 0x16fdcefe0 res: 0x16fdcef00
v1@0x16fdcef50 : 0x16fdcefd0  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
```

`res` in both calls has the same address

1st call to `vector_sum` on line 24

- *Note: There is nothing in output that says `res1` is at that address, but we know it's in main and it fits there*

```
$ 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]
   22 |   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
```

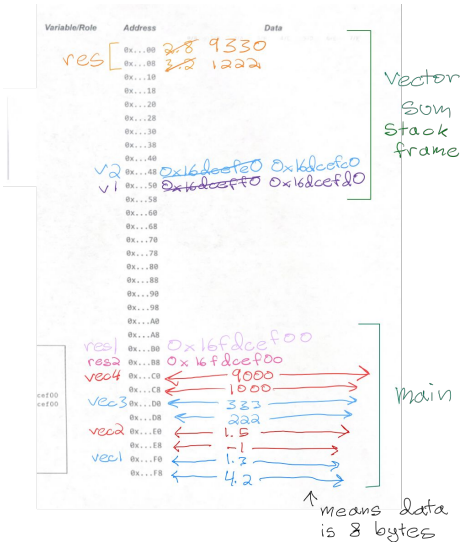


The Stack

- When a program is running, each function that is called gets a segment of the memory to store values. This is called the function's **stack frame**.
- **The stack grows** to fit more values from the function if needed
- However, once a function is done executing, the segment in memory no longer belongs to the function and is able to reused for other functions
  - Because of this, local variables in functions get overwritten once the function is done
- This is why we should **NEVER** return a pointer to a local variable on a function's stack frame because it may get overwritten
- The 2nd call to `vector_sum` shows this happening (see in next slide)

2nd call to `vector_sum` on line 27

- What is returned is the **address from the stack**
- Never return a pointer to a local variable on a function's stack frame because it may get overwritten
- Because there are no other function calls, the stack layout for `vector_sum` remained the same, but this isn't always the case
- This is why `res1[0]` and `res2[0]` both had the value `9333`



Questions

- Why is res1[0] and res2[0] the same?
  - After the first call, res1[0] is 1.3 + 1.5 = 2.8
  - After the second call, res1[0] is 9333 and res2[0] is 9333
  - res1 was an address to the stack of the vector_sum function
  - When we did the second call, it wrote over the values at the address that res1 was pointing to
  - After functions are done executing, the stack frame is then reused for other function calls
- Does C delete the data after we're done?
  - No, there's no meaningful reason to "delete" the data. There is no need to go in and zero out memory. Once memory is written, is stays that value until something else overwrites it

Side Note: Pros and drawbacks of C

```
$ 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]
   22 |   return res;
```

- This should be an error, not a warning
- C gives you *complete* control over memory, but this comes with mistakes being made because it will not check if memory is safe to deallocate

sizeof()

`sizeof(x)` => compile-time operation, **not** a function

`sizeof` has different behavior depending on what type you give it

x could be:

All pointers in a 64-bit machine is 8 bytes!

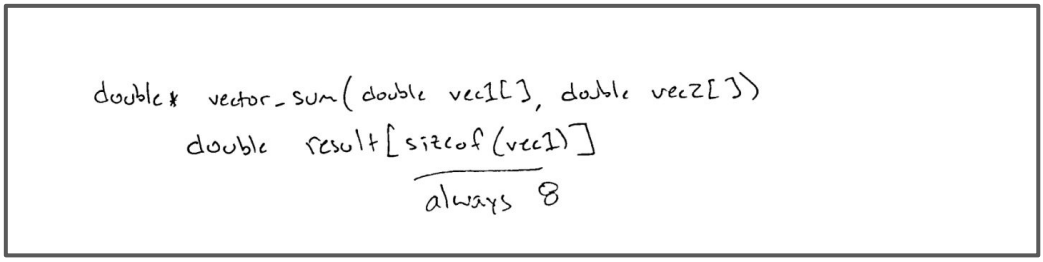
- A type: gives the number of bytes to store 1 of that type
  - sizeof(int32_t) = 4
  - sizeof(double) = 8
  - sizeof(char) = 1
  - sizeof(double*) = 8
  - sizeof(char*) = 8
  - sizeof(int32_t*) = 8
- An expression (for example, a variable): gives # of bytes to store 1 of the type of that variable
  - char c = ‘a’; sizeof(c) = 1
  - int32_t i = 22; sizeof(i) = 4
- An array variable (**not** a pointer): gives total number of bytes for array declaration
  - char c[9]; sizeof(c) = 9
  - double ns[5]; sizeof(ns) = 40
  - char a[] = “abc”; sizeof(a) = 4

For the systems you will be using, pointers are 8 bytes, 64 bits. On 32 bit systems, pointers are 4 bytes.

GCC calculates the number, and replaces sizeof with a number in your program. One sizeof call will always return the same number, therefore you cannot use it for variable length arrays

To get the length of an `vec1` in `vector_sum`, could we just do sizeof(`vec1`)?

No, when we try to do `sizeof(vec1)`, what is returned is 8 bytes. This is because `vec1` is a pointer and all pointers in a 64-bit machine is 8 bytes.



Array Indexing with a Pointer Variable

Array indexing with pointers

```
v_s(double *v1; …)
      v1[3] => 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)

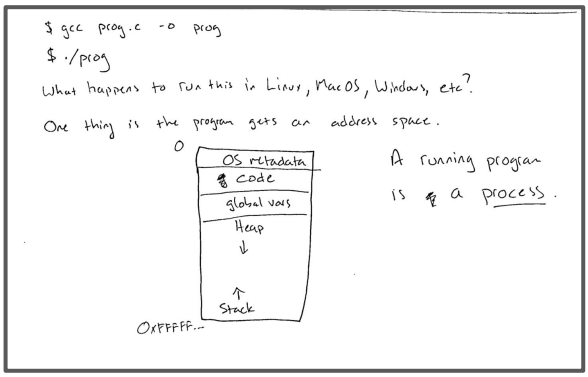
sizeof(char)

sizeof(int)

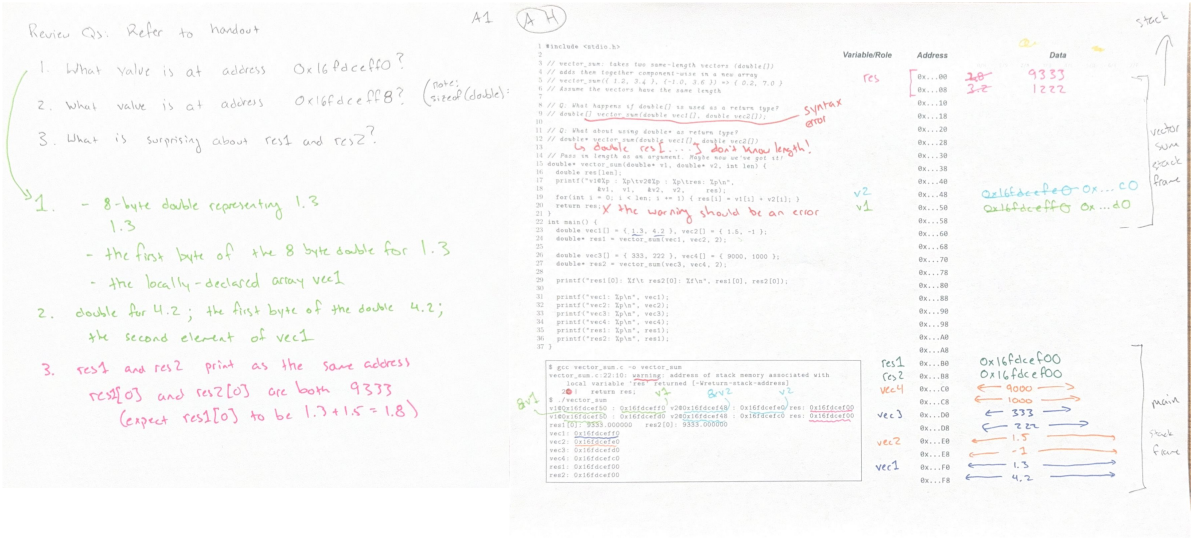
Look up sizeof(<type>) bytes at variable + (index*sizeof(type))

Side Note: What happens to run an executable?

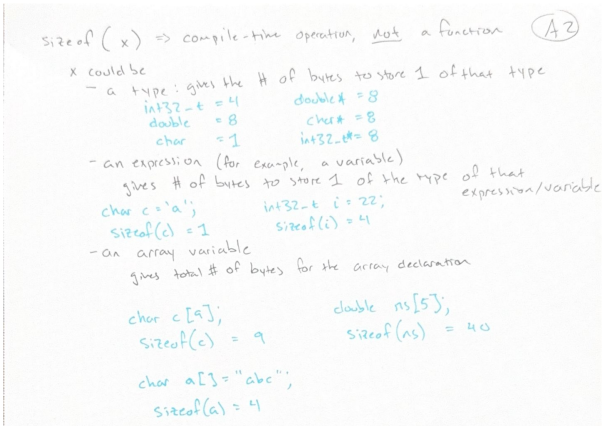
- When I do `./my_program` in my terminal, what happens?
  - The program gets an address space (a chunk of memory) that has a specific layout
  - The top has metadata for the operating system to use
  - Then there's space for the program's code
  - Then space for global variables
  - And then the stack is at the highest address and grows up
  - The heap grows down
- Once a program starts running, we call it a **process**

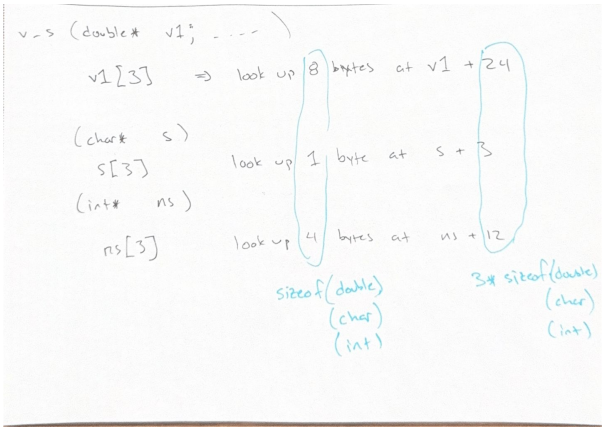


Joe's Notes (11am)



Joe's Notes (11am)





Joe's Notes (12:30pm)

Review Qs: Refer to handout

Q1: What value is stored at address 0x16fdceff0?

Q2: What value is stored at address 0x16fdceff8?

Q3: What is surprising about res1 and res2?

1: - double 1.3 (8 byte)
- the first element of vec1
- the entire array vec1
- the first byte of the 8 byte double 1.3

2: - double 4.2 (8 byte)
- the second element of vec1
- the first byte of the 8 byte double 4.2

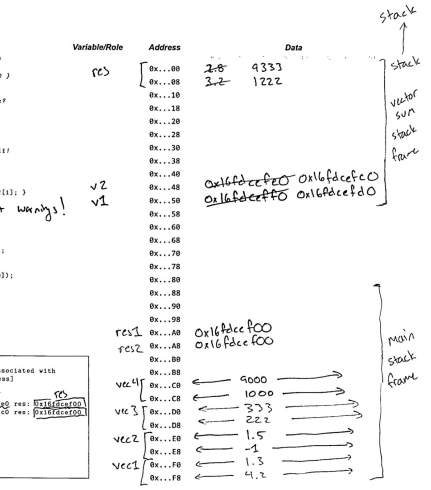
3: res1 and res2 ~~are~~ hold same address

res1[0] = 9333?!

should be 2.8

```
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                 syntax error
11 // Q: What about using double* as return type?
12 // double* vector_sum(double vec1[], double vec2[])
13                 double result[...]
14 // Pass in length as an argument. Might 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; X don't do this return-result warnings!
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 }
```

```
$ 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]
  22 |   return res;
$ ./vector_sum
v1@0x16fdceff0 : 0x16fdceff0 v2@0x16fdceff8 : 0x16fdcefa0 res: 0x16fdcefa0
v1@0x16fdcefb0 : 0x16fdcefd0 v2@0x16fdcefd8 : 0x16fdcefc0 res: 0x16fdcefc0
res1[0]: 9333.000000   res2[0]: 9333.000000
vec1: 0x16fdceff0
vec2: 0x16fdceff8
vec3: 0x16fdcefd0
vec4: 0x16fdcefc0
res1: 0x16fdcefa0
res2: 0x16fdcef00
```



Joe's Notes (12:30pm)

```
double* vector_sum(double vec1[], double vec2[])
    double result[sizeof(vec1)]
                  always 8
```

sizeof(x)

x could be:

- a type: the # of bytes for 1 value of that type

  sizeof(int32_t) = 4
  sizeof(char) = 1
  sizeof(double) = 8
  sizeof(double*) = 8
  sizeof(char*) = 8

  any pointer type is 8 bytes

- an expression (a variable)
  the # of bytes for 1 value of the variable's type.

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

even int x = 41;       sizeof(x) = 4
```

```
void f(double* ns) {
    sizeof(ns) = 8
}
```

- an array variable: # of bytes for entire array

```
char s[] = "abc";      uint16_t n[4];

int vals[] = {5, -3, 9};
```

sizeof(s) = 4 sizeof(n) = 8 sizeof(vals) = 12

```
double* ns
    ns[3]      look up 8 bytes at ns + 24

char* s
    s[3]       look up 1 byte  at s + 3

int32_t* vals
    vals[3]    look up 4 bytes at vals + 12
```

sizeof(double)
sizeof(char)
sizeof(int32_t)

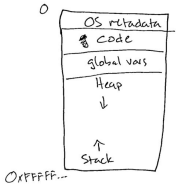
3 * sizeof(double)
3 * sizeof(char)
3 * sizeof(int32_t)

8:vals[3]

```
$ gcc prog.c -o prog
$ ./prog
```

What happens to run this in Linux, MacOS, Windows, etc?

One thing is the program gets an address space.



A running program is a process.