```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[]);
      syntax error
// Q: What about using double* as return type?
// double* vector_sum(double vec1[], double vec2[])
          double result[...]
// 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;
}
          X don't do this return-trust warnings!
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", res2);
}
```

```
$ 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
```

| Variable/Role | Address | Data |
|---|---|---|
| res | 0x...00 | ~~2.8~~ 9333 |
| | 0x...08 | ~~3.2~~ 1222 |
| | 0x...10 | |
| | 0x...18 | |
| | 0x...20 | |
| | 0x...28 | |
| | 0x...30 | |
| | 0x...38 | |
| | 0x...40 | |
| v2 | 0x...48 | ~~0x16fdcefe0~~ 0x16fdcefc0 |
| v1 | 0x...50 | ~~0x16fdceff0~~ 0x16fdcefd0 |
| | 0x...58 | |
| | 0x...60 | |
| | 0x...68 | |
| | 0x...70 | |
| | 0x...78 | |
| | 0x...80 | |
| | 0x...88 | |
| | 0x...90 | |
| | 0x...98 | |
| res1 | 0x...A0 | 0x16fdcef00 |
| res2 | 0x...A8 | 0x16fdcef00 |
| | 0x...B0 | |
| | 0x...B8 | |
| vec4 | 0x...C0 | 9000 |
| | 0x...C8 | 1000 |
| vec3 | 0x...D0 | 333 |
| | 0x...D8 | 222 |
| vec2 | 0x...E0 | 1.5 |
| | 0x...E8 | -1 |
| vec1 | 0x...F0 | 1.3 |
| | 0x...F8 | 4.2 |

stack

stack

vector sum stack frame

main stack frame

&v1 v1 &v2 v2 res

```
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

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[] = {7, -3, 9};
```

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

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 hold same address

res1[0] = 9333?!

should be 2.8

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)

&vals[3]

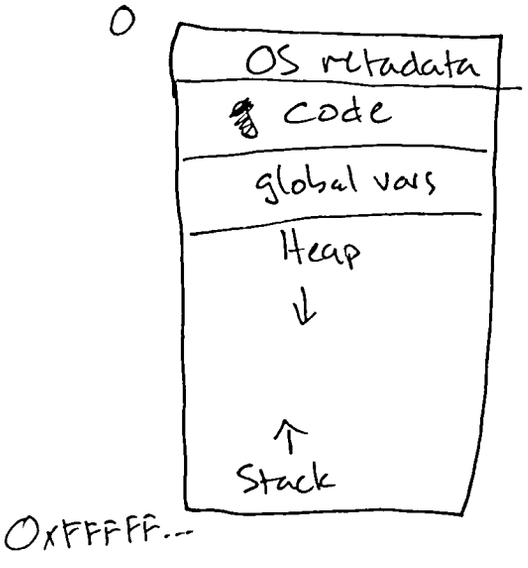
---

$ 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.