CSE 333 Section AB

Pointers, debugging & valgrind (w/ Yifan & Travis)

Logistics

Due Friday:

Exercise 3 @ 11 am

Due Monday:

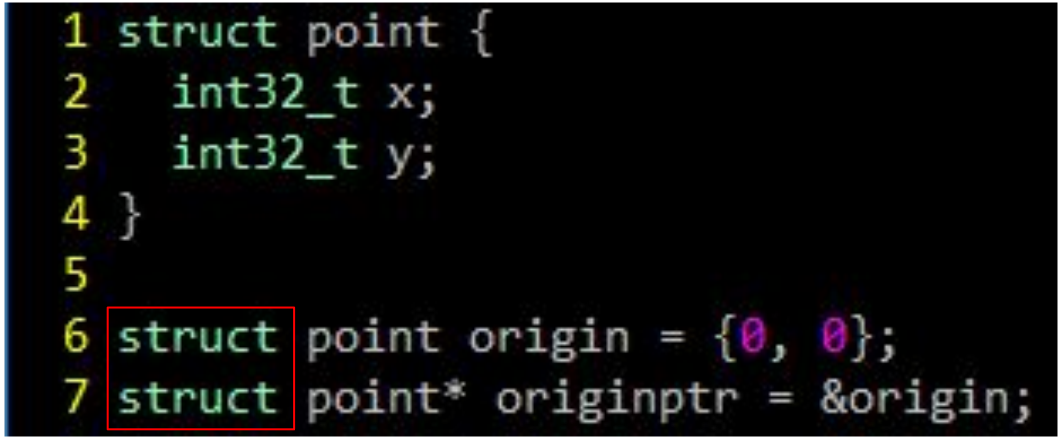
Exercise 4 @ 11 am

Due in 1 Week: (10/10)

HW1 @9:00 pm

Defining Structs

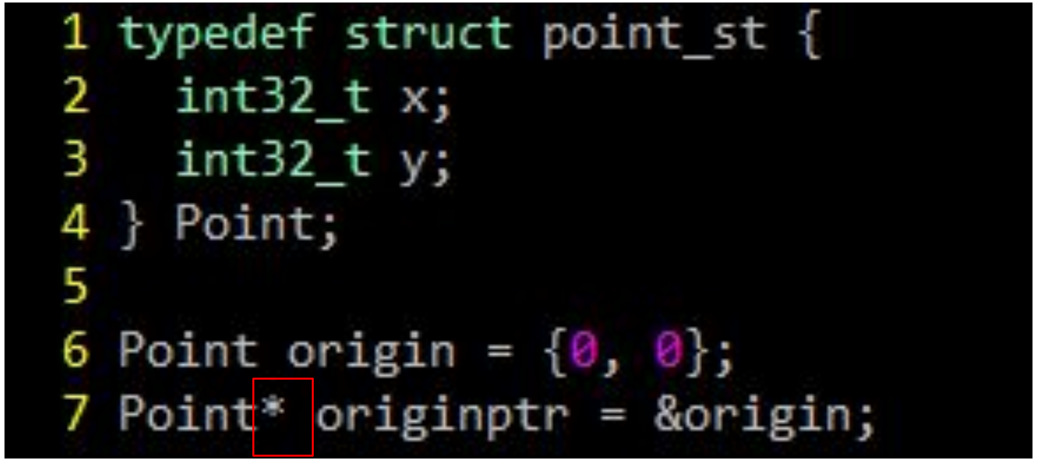
How can we get rid of this?



```
struct point {
  int32_t x;
  int32_t y;
}

struct point origin = {0, 0};
struct point* originptr = &origin;
```

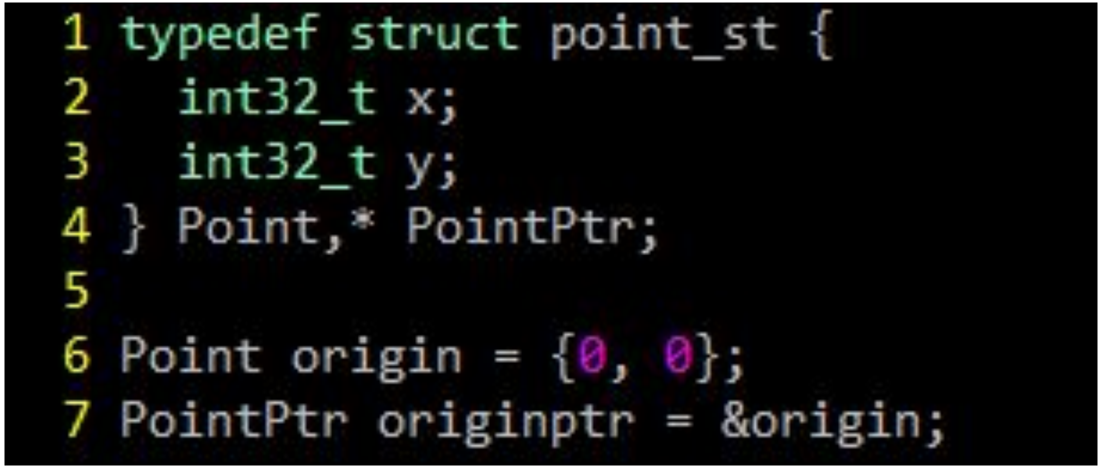
Defining Structs

How can we get rid of this?

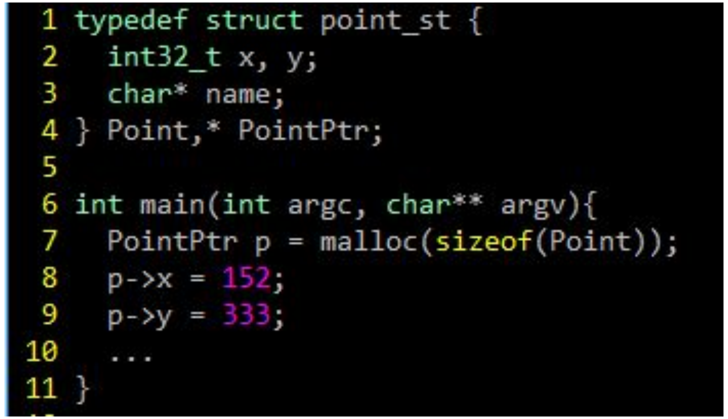


```
typedef struct point_st {
  int32_t x;
  int32_t y;
} Point;

Point origin = {0, 0};
Point* originptr = &origin;
```

Defining Structs

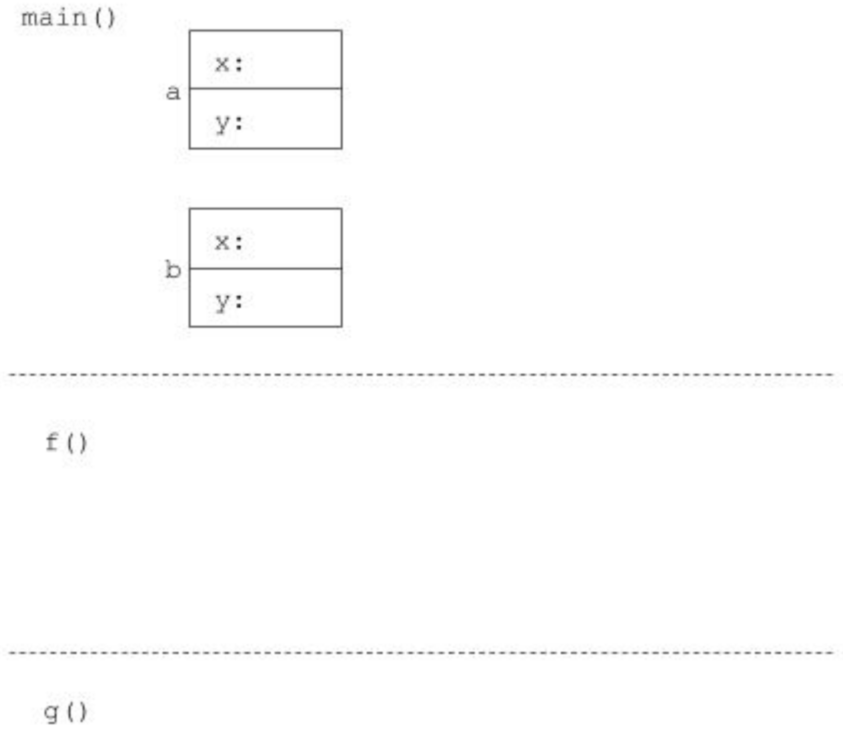

```c
typedef struct point_st {
  int32_t x;
  int32_t y;
} Point,* PointPtr;

Point origin = {0, 0};
PointPtr originptr = &origin;
```

Struct Memory diagrams



```c
typedef struct point_st {
  int32_t x, y;
  char* name;
} Point,* PointPtr;

int main(int argc, char** argv){
  PointPtr p = malloc(sizeof(Point));
  p->x = 152;
  p->y = 333;
  ...
}
```

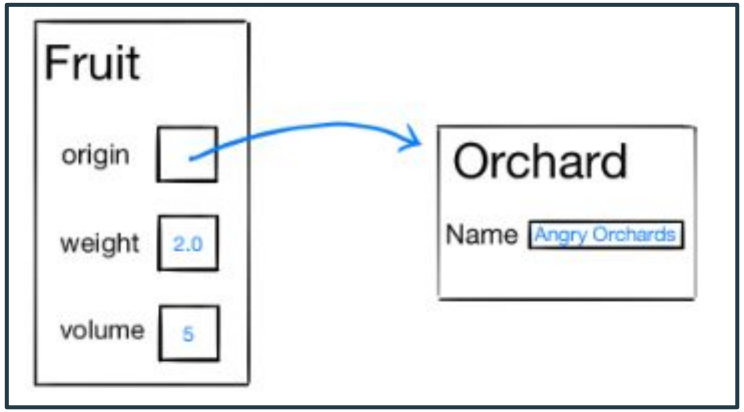
```
typedef struct coordinate {
  double x, y;
} Coordinate;

void f(Coordinate *one, Coordinate *two) {
  two->x *= 2;
  two = one;
  g(&two, *one);
}
void g(Coordinate **p1, Coordinate c) {
  Coordinate **p2 = p1;
  Coordinate c3 = {c.x * 2, c.y * 2};
  **p1 = c3;
  *p2 = NULL;
  //// HERE ////
}

int main(void) {
  Coordinate a = {1, 2};
  Coordinate b = {10, 20};
  f(&a, &b);
  return 0;
}
```

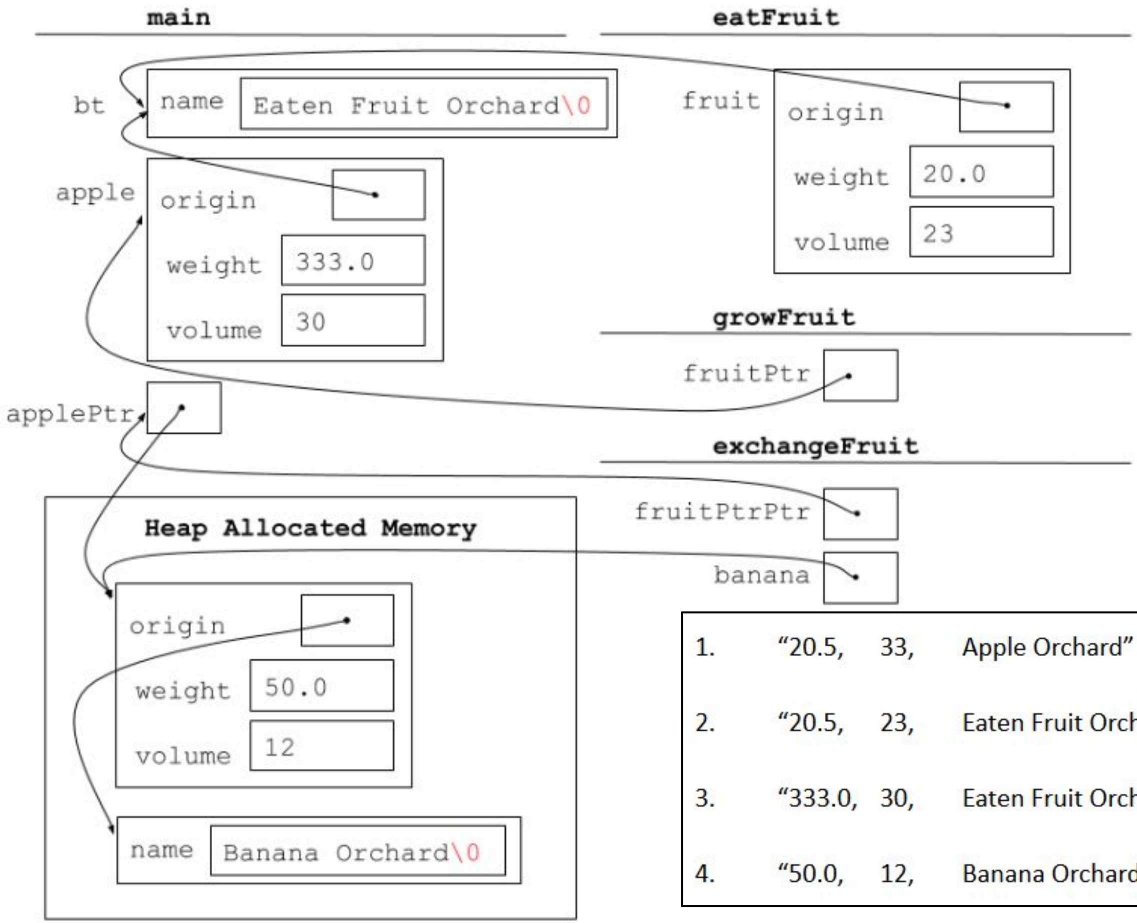


Fruits & Orchards

```
typedef struct fruit_st {
  OrchardPtr origin;
  double weight;
  int volume;
} Fruit, *FruitPtr;
```

```
typedef struct orchard_st {
  char name[20] ;
} Orchard, *OrchardPtr;
```





1.	“20.5,	33,	Apple Orchard”	Initial values that were assigned
2.	“20.5,	23,	Eaten Fruit Orchard”	Struct is passed by value
3.	“333.0,	30,	Eaten Fruit Orchard”	Struct passed by reference
4.	“50.0,	12,	Banana Orchard”	Struct is completely reassigned

Valgrind

- Our program compiles and runs fine, what’s the problem?
  - “./leaky 1 10”
- “valgrind --leak-check=full ./leaky 1 10”
- Travis’ alias:
  - `val() { valgrind --leak-check=full "$@"; }`

```c
int* rangeArray(int n, int m) {
  int length = m - n + 1;

  // Heap allocate the array needed to return
  int *array = (int*) malloc(sizeof(int) * length);

  // Initialize the elements
  for (int i = 0; i <= length; i++) {
    array[i] = i + n;
  }

  return array;
}

// Accepts two integers as arguments
int main(int argc, char *argv[]) {
  if (argc != 3) return EXIT_FAILURE;

  int n = atoi(argv[1]), m = atoi(argv[2]);
  int* nums = rangeArray(n, m);

  // Print the resulting array
  for (int i = 0; i <= (m - n + 1); i++) {
    printf("%d", nums[i]);
  }

  // Append newline char to our output
  puts("");

  return EXIT_SUCCESS;
}
```

```c
int* rangeArray(int n, int m) {
  int length = m - n + 1;

  // Heap allocate the array needed to return
  int *array = (int*) malloc(sizeof(int) * length);

  // Initialize the elements
  for (int i = 0; i <= length; i++) {
   array[i] = i + n;
  }

  return array;
}
// Accepts two integers as arguments
int main(int argc, char *argv[]) {
  if (argc != 3) return EXIT_FAILURE;

  int n = atoi(argv[1]), m = atoi(argv[2]);
  int* nums = rangeArray(n, m);
  // We're allocating space for 10 ints, but we access 11
  // ints with i <= instead of i <
  for (int i = 0; i < (m - n + 1); i++) {
   printf("%d", nums[i]);
  }
  // We need to free the array of integers malloced in RangeArray.
  free(nums);
  // Append newline char to our output
  puts("");

  return EXIT_SUCCESS;
}
```