# CSE 333 – Section 2: Structs, Debugging, Memory Management, and Valgrind

To define a struct, we use the **`struct`** statement. A struct typically has a name (a tag), and one or more members. The **`struct`** statement defines a new type:

```
struct fruit_st {
  OrchardPtr origin;
  double weight;
  int volume;
};
```

The C Programming language provides the keyword **`typedef`**, which defines an alternate name for a type:

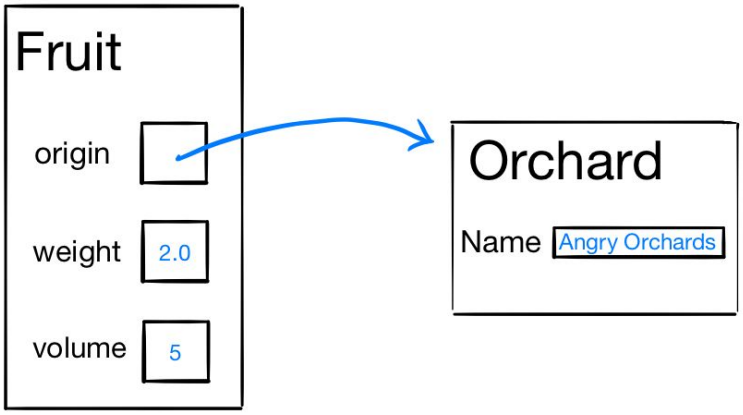
```
typedef struct fruit_st {
  OrchardPtr origin;
  double weight;
  int volume;
} Fruit, *FruitPtr;
```

The above defines the name `Fruit` to represent the type **`struct`** `fruit_st` as well as the name `FruitPtr` to represent a **`struct`** `fruit*` (a pointer to a **`struct`** `fruit_st`).

Now let's define the Orchard type used in Fruit:

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

*Assume we've initialized a Fruit and corresponding Orchard with 'random' values.*
Then we can draw a memory diagram for the above structs like so:



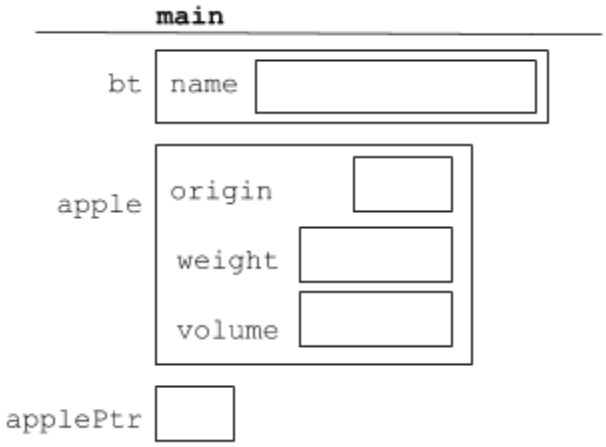

A **`struct`** is passed and returned by value. That means that **if we pass a struct as an argument, the callee function gets a local copy of the entire struct.** We will explore this in more detail in question 1.

## 1. Structs and Pointers

What does the following program output?
Use the definitions of `Fruit` and `Orchard` from the first page of the section handout.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int eatFruit(Fruit fruit) {
  fruit.weight -= 0.5;
  fruit.volume -= 10;
  strcpy(fruit.origin->name, "Eaten Fruit Orchard");
  return fruit.volume;
}

void growFruit(FruitPtr fruitPtr) {
  fruitPtr->weight = 333.0;
  fruitPtr->volume += 7;
}

void exchangeFruit(FruitPtr* fruitPtrPtr) {
  FruitPtr banana = (FruitPtr)malloc(sizeof(Fruit));
  banana->weight = 50.0;
  banana->volume = 12;
  banana->origin = (OrchardPtr)malloc(sizeof(Orchard));
  strcpy(banana->origin->name, "Banana Orchard");
  *fruitPtrPtr = banana;
}

int main(int argc, char* argv[]) {
  Orchard bt;
  strcpy(bt.name, "Apple Orchard");

  Fruit apple;
  FruitPtr applePtr = &apple;
  apple.origin = &bt;
  apple.weight = 10.5;
  apple.volume = 33;
  applePtr->weight = 20.5;
  applePtr->volume = apple.volume;

  printf("1. %.1f, %d, %s \n", applePtr->weight, applePtr->volume, applePtr->origin->name);
  apple.volume = eatFruit(apple);
  printf("2. %.1f, %d, %s \n", applePtr->weight, applePtr->volume, applePtr->origin->name);
  growFruit(applePtr);
  printf("3. %.1f, %d, %s \n", applePtr->weight, applePtr->volume, applePtr->origin->name);
  exchangeFruit(&applePtr);
  printf("4. %.1f, %d, %s \n", applePtr->weight, applePtr->volume, applePtr->origin->name);

  free(applePtr->origin);
  free(applePtr);

  return 0;
}
```

(a) Draw a memory diagram for the program. We've put some boxes for the variables in `main()` to help get you started.




(b) What does this program output?

1. _____, _____, _____

2. _____, _____, _____

3. _____, _____, _____

4. _____, _____, _____

## 2. Reverse a Linked List [Extra Practice]

A node in a linked list is defined as follows:

```
struct Node {
  int value;
  struct Node* next;
};
```

Complete the function `reverse` to reverse the linked list and return the head of the resulting list.

Do not create new list nodes and do not modify the contents of a list node.
Assume `next == NULL` implies the end of the list.

```
struct Node* reverse(struct Node* head) {

}
```

### 3. Sorted Array To Binary Search Tree [Extra Practice]

A node in a tree is defined as follows:

```
struct TreeNode {
  int value;
  struct TreeNode* left;
  struct TreeNode* right;
};
```

Complete the implementation of the `sortedArrayToBST` function to convert a sorted integer array into a binary search tree. The client to this method will invoke it as follows:

```
struct TreeNode* root = sortedArrayToBST(sortedArray, 0, n - 1);
```

where `sortedArray` is a sorted array of integers and `n` is the length of `sortedArray`.

```
struct TreeNode* sortedArrayToBST(int[] arr, int low, int high) {




}
```

## 4. Leaky Code and Valgrind

Consider the following leaky program:

```c
#include <stdio.h>
#include <stdlib.h>

// Returns an array containing [n, n+1, ... , m-1, m]. If n>m, then the
// array returned is []. If an error occurs, NULL is returned.
 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]);  // Parse cmd-line args
  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;
}
```

Here is the valgrind output from running the command:

```
valgrind --leak-check=full ./leaky 1 10
```

```
==17501== Memcheck, a memory error detector
==17501== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==17501== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==17501== Command: ./leaky 1 10
==17501==
==17501== Invalid write of size 4
==17501==    at 0x40062B: RangeArray (leaky.c:15)
==17501==    by 0x400698: main (leaky.c:24)
==17501==  Address 0x5203068 is 0 bytes after a block of size 40 alloc'd
==17501==    at 0x4C29BC3: malloc (vg_replace_malloc.c:299)
==17501==    by 0x400601: RangeArray (leaky.c:11)
==17501==    by 0x400698: main (leaky.c:24)
==17501==
==17501== Invalid read of size 4
==17501==    at 0x4006BA: main (leaky.c:28)
==17501==  Address 0x5203068 is 0 bytes after a block of size 40 alloc'd
==17501==    at 0x4C29BC3: malloc (vg_replace_malloc.c:299)
==17501==    by 0x400601: RangeArray (leaky.c:11)
==17501==    by 0x400698: main (leaky.c:24)
==17501==
1234567891011
==17501==
==17501== HEAP SUMMARY:
==17501==     in use at exit: 40 bytes in 1 blocks
==17501==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==17501==
==17501== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==17501==    at 0x4C29BC3: malloc (vg_replace_malloc.c:299)
==17501==    by 0x400601: RangeArray (leaky.c:11)
==17501==    by 0x400698: main (leaky.c:24)
==17501==
==17501== LEAK SUMMARY:
==17501==    definitely lost: 40 bytes in 1 blocks
==17501==    indirectly lost: 0 bytes in 0 blocks
==17501==      possibly lost: 0 bytes in 0 blocks
==17501==    still reachable: 0 bytes in 0 blocks
==17501==         suppressed: 0 bytes in 0 blocks
==17501==
==17501== For counts of detected and suppressed errors, rerun with: -v
==17501== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

Given the leaky code and the valgrind output, correct the program so that valgrind outputs no warnings/errors when run with the program with the same parameters.