CIT 5950 – Section 2: Structs, Debugging, Memory Management, and Valgrind

SOLUTIONS

1. Debugging with gdb

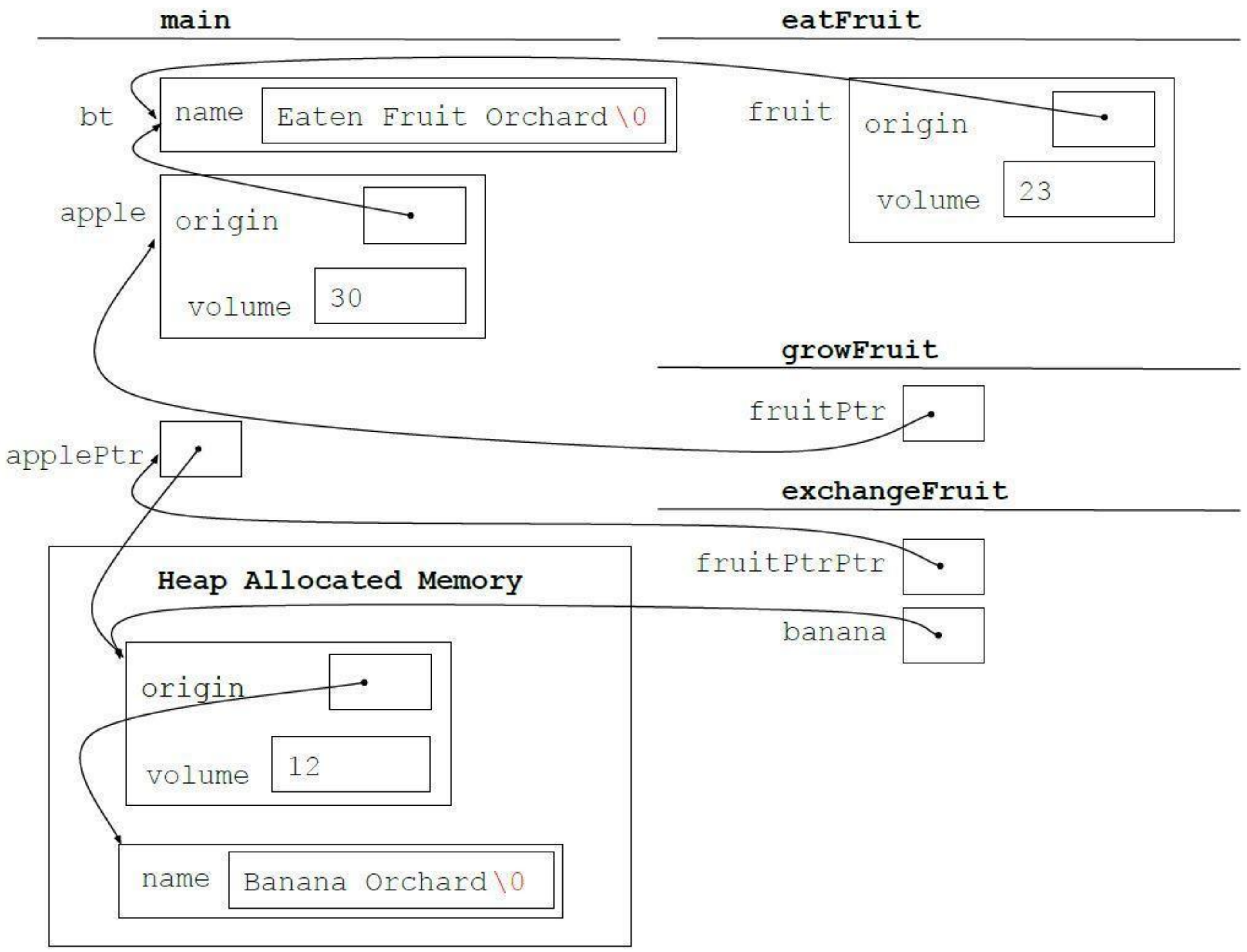
```
#define MAX_STR 100   /* length of longest input string */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Return a new string with the contents of s backwards */
char * reverse(char * s) {
  char * result = NULL;            /* the reversed string */
  int L, R;
  char ch;
  /* copy original string then reverse and return the copy */
  int strsize = strlen(s)+1;
  result = (char *)malloc(strsize);
  strncpy(result, s, strsize);
  L = 0;
  R = strlen(result) - 1;
  while (L < R) {
    ch = result[L];
    result[L] = result[R];
    result[R] = ch;
    L++; R--;
  }
  return result;
}

/* Ask the user for a string, then print it forwards and backwards.     */
int main() {
  char line[MAX_STR];    /* original input line */
  char * rev_line;       /* backwards copy from reverse function */

  printf("Please enter a string: ");
  fgets(line, MAX_STR, stdin);
  line[strlen(line)-1] = '\0';
  rev_line = reverse(line);
  printf("The original string was:   >%s<\n", line);
  printf("Backwards, that string is: >%s<\n", rev_line);
  printf("Thank you for trying our program.\n");
  free(rev_line);
  return EXIT_SUCCESS;
}
```

2. Leaky Code and Valgrind

```
#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
  // By using <=, we are writing to length + 1 ints instead of length ints
  // Change <= to < to fix this off-by-one error
  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
  // 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;
}
```

3. Structs and Pointers

Final memory diagram (Note: `eatFruit`, `growFruit`, and `exchangeFruit` contexts would be cleaned up and reused during program execution).



Output:

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

4. Reverse a Linked List [Extra Practice]

```
struct Node* reverse(struct Node* head) {
  struct Node *prev = NULL, *next = NULL;
  struct Node *current = head;

  while (current != NULL) {
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
  }

  return prev;
}
```

5. Sorted Array To Binary Search Tree [Extra Practice].

```
struct TreeNode *sortedArrayToBST(int[] arr, int low, int high) {
  if (low > high) {
    return NULL;
  }

  // Make the middle element the root of this subtree.
  int mid = (low + high) / 2;
  struct TreeNode *root = (struct TreeNode*)malloc(sizeof(TreeNode));
  root->value = arr[mid];

  // Construct the left subtree and assign it to be the left child.
  root->left = sortedArrayToBST(arr, low, mid - 1);

  // Construct the right subtree and assign it to be the right child.
  root->right = sortedArrayToBST(arr, mid + 1, high);

  return root;
}
```