

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

In this class, it is very helpful to be comfortable with gdb as a tool to debug C/C++ code. Note that gdb allows you to see the source code while you run it, and that it has many useful commands to analyze your program.

**For GDB to work with your C program, compile it using the “-g” flag**

### Starting GDB

To start up gdb, run the following command. Note that the `-tui` flag is optional. It is used to enable a text UI.

```
bash$ gdb -tui <program file name>
```

Here is a list of some essential gdb commands, if you want to know more, ask a TA or investigate the resources at the top of the page.

### [IN GDB] Controlling Program Execution

- `run <command_line_args>` Run the program with provided `command_line_args`
- `next` Go to next instruction, but don't dive into functions
- `step` Go to next instruction, and dive into functions
- `finish` Continue until current function returns
- `quit` close gdb

### [IN GDB] Examining the Current Program

- `list` Shows the current or given source context
- `backtrace` Shows the call stack
- `up` Moves up a stack frame
- `down` Moves down a stack frame
- `print <expression>` Prints content of variable/memory location/register

### [IN GDB] Setting Breakpoints and Continuing

- `break <where>` Set a new breakpoint
- `info breakpoints` Prints information about the set breakpoints
- `continue` Continue normal execution

## 1. Debugging with gdb

Provided below is a segment of `reverse.c`. The program intends to take a string, and then reverse the ordering of the characters in the string. For example, if "Hello" is provided, then "olleH" should be returned.

```
gcc -Wall -std=c11 -g -o reverse reverse.c
```

Identify and fix the errors that are in `reverse.c`. Use `gdb` to analyze the program for errors.

```
#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 */
    strcpy(result, s);

    L = 0;
    R = strlen(result);
    while (L < R) {
        ch = result[L];
        result[L] = result[R];
        result[R] = ch;
        L++; R--;
    }

    return result;
}
```

## 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 */
    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;
}
```

### 3. Structs and Pointers

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;
    int volume;
};
```

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

```
typedef struct fruit_st {
    OrchardPtr origin;
    int volume;
} Fruit;
```

The above defines the name `Fruit` to represent the type `struct fruit_st`.

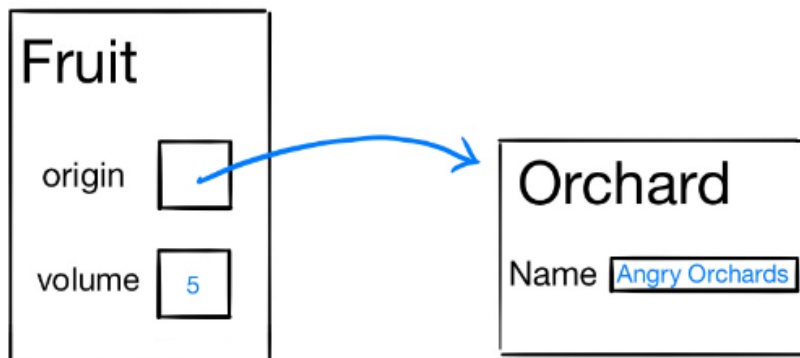
Now let's define the Orchard type used in Fruit:

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

The above defines the name `Orchard` to represent the type `struct orchard_st` as well as the name `OrchardPtr` to represent a `Orchard*` (a pointer to a `struct orchard_st`)

*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 3.

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

void growFruit(Fruit* fruitPtr) {
    fruitPtr->volume += 7;
}

void exchangeFruit(Fruit** fruitPtrPtr) {
    Fruit *banana = (Fruit*)malloc(sizeof(Fruit));
    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");

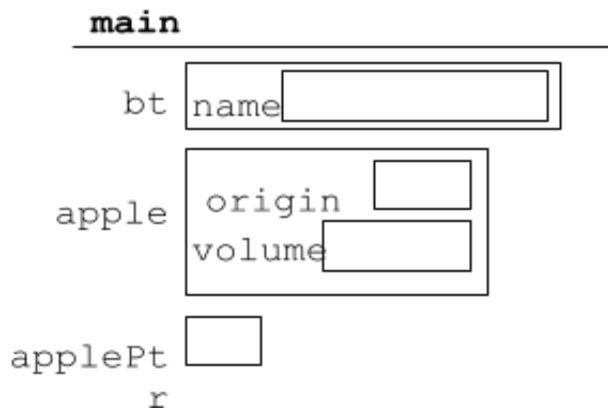
    Fruit apple;
    Fruit* applePtr = &apple;
    apple.origin = &bt;
    apple.volume = 33;
    applePtr->volume = apple.volume;

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

#### 4. 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) {
```

```
}
```

## 5. 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 balanced 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) {
```

```
}
```