CSE 333 – SECTION 2

gdb, valgrind, pointers & structs

Questions, Comments, Concerns

- Do you have any?
- Exercises going ok?
- Lectures make sense?
- Homework 1 – START EARLY!

Upcoming Due Dates:

- Due Jan 20th, EX3 due @ 10 am
- Due Jan 28th, HW1 due @ 11 pm

Structs (Recap from 351)

- A struct is a C data type that contains a set of fields
  - Useful for defining new structured types of data.
  - Act similarly to primitive variables.
- Generic Declaration:

```
struct tagname {
    type1 name1;
    …
    typeN nameN;
}
```

Example:

```
struct Point {
    int x;
    int y;
};
```

*defines a new data type called “struct Point”

- Declaring and initializing a struct:

```
// Remember to use “struct Point” to refer to the struct.
// Initializes a struct Point variable called origin with x = 0.0 & y = 0
struct Point origin = {0.0, 0};
```

Using Structs

- Use "." to refer to a field in a struct
- Use "->" to refer to a field from a struct pointer
  - Dereferences the pointer, then accesses the field.

```
struct Point {
    int x;
    int y;
};

struct Point p1 = {5, 10};
struct Point *p2 = &p1;  // Notice that this is a pointer to p1

p1.x = 15;  // p1 now = {15, 10}
p2->y = 0;  // since p2 points to p1, p1 now = {15,0}
```

Typedef

- Allows you to define an alternate name for existing data types.
- Generic format:

```
typedef type name;
```

Examples:

```
typedef int int_alias;
```

```
typedef struct Point point;
point origin = {0, 0}
```

- Joint struct definition and typedef:

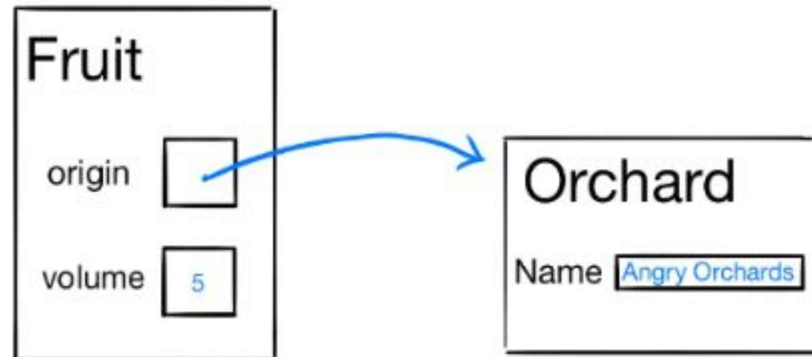
```
typedef struct {
    int x;
    int y;
} point;
// Just refer to it as “point”
point origin = {0, 0};
```

Exercise 3: Memory diagrams

Fruits & Orchards

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

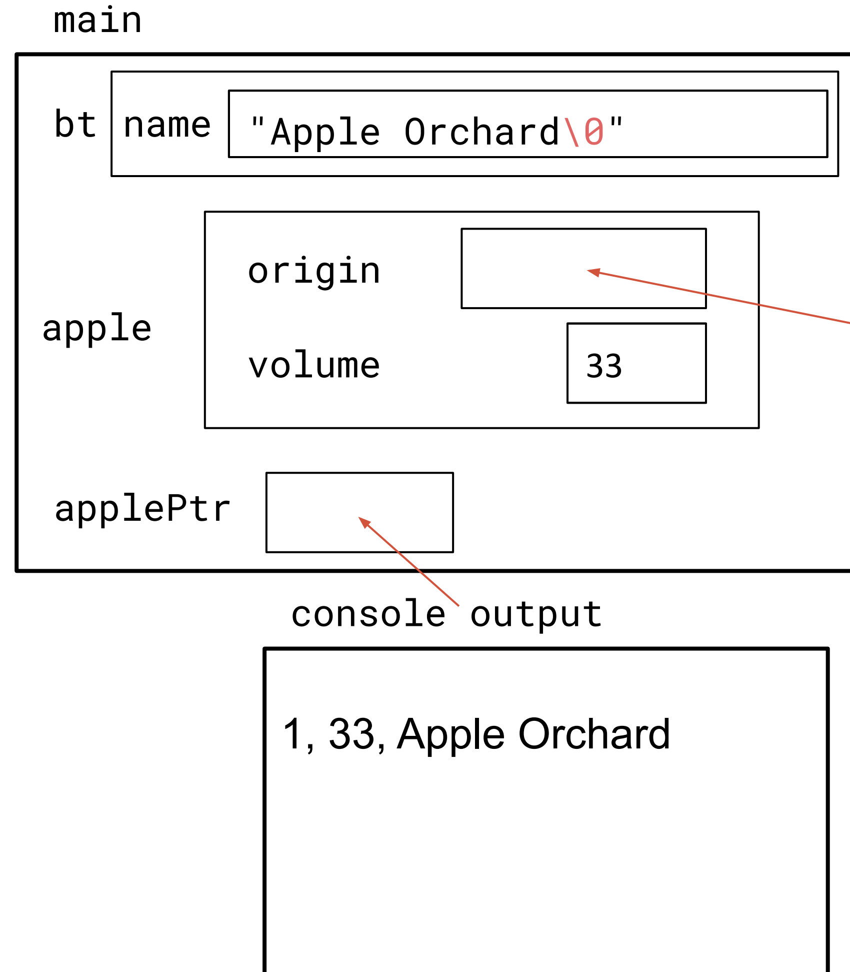
```
typedef struct orchard_st {
  char name[20] ;
} Orchard, *OrchardPtr;
```



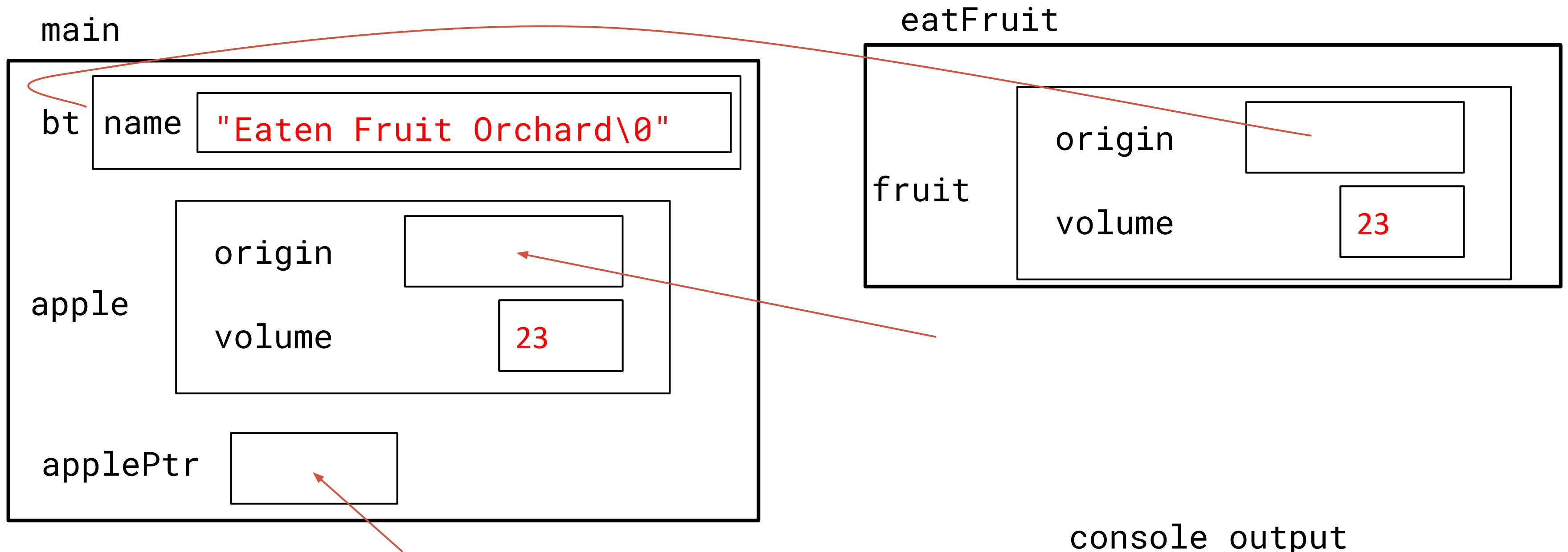
```c
int main(int argc, char* argv[]) {
  Orchard bt;
  strcpy(bt.name, "Apple Orchard");

  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);
  …
```



```c
…
apple.volume = eatFruit(apple);
printf("2, %d, %s \n", applePtr->volume,
          applePtr->origin->name);
```

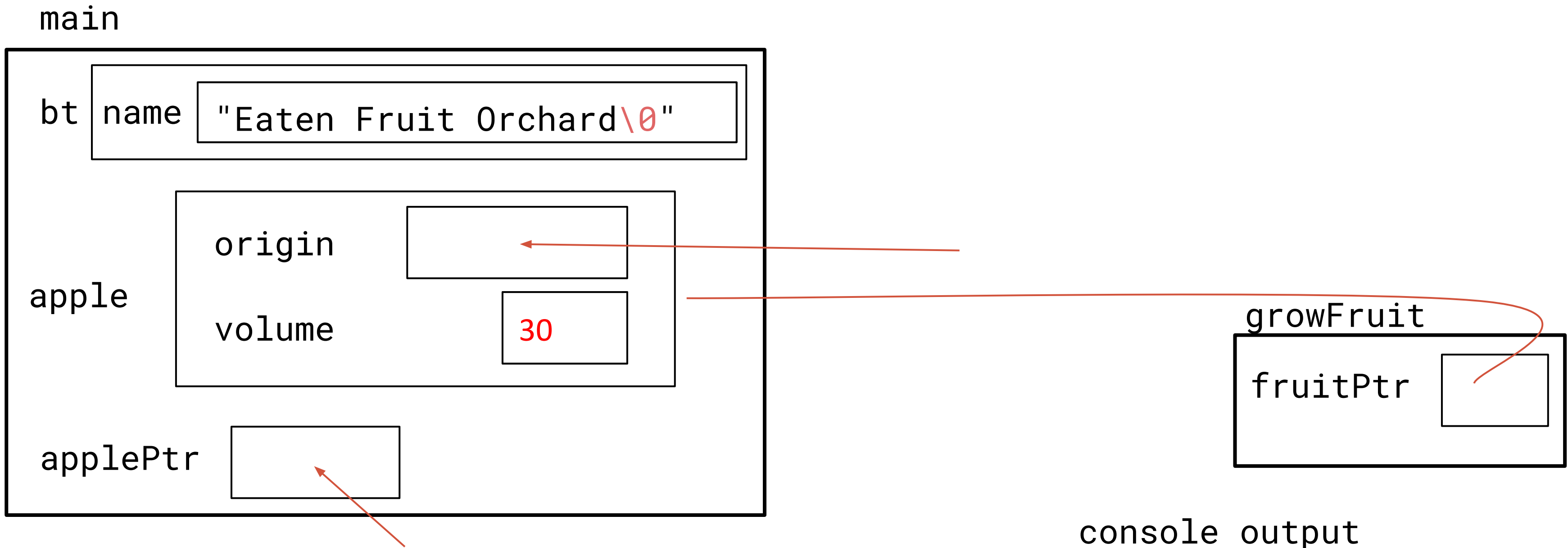



```c
int eatFruit(Fruit fruit) {
  fruit.volume -= 10;
  strcpy(fruit.origin->name,
      "Eaten Fruit Orchard");
  return fruit.volume;
}
```

console output

```
1, 33, Apple Orchard
2, 23, Eaten Fruit Orchard
```

```
…
growFruit(applePtr);
printf("3, %d, %s \n", applePtr->volume,
    applePtr->origin->name);
```



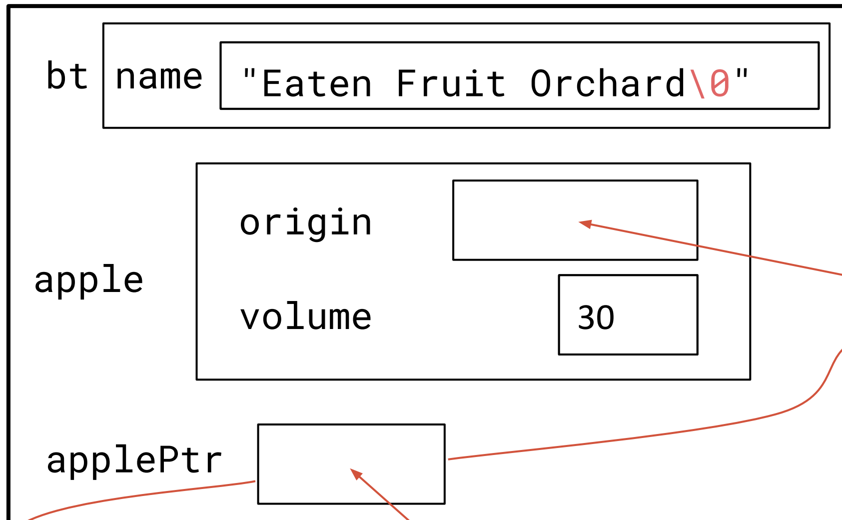

```
void growFruit(Fruit* fruitPtr) {
  fruitPtr->volume += 7;
}
```

console output

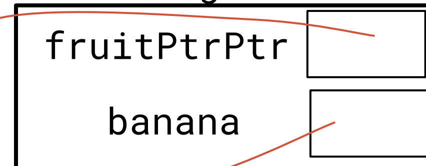
1, 33, Apple Orchard
2, 23, Eaten Fruit Orchard
3, 30, Eaten Fruit Orchard

```
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;
}
```

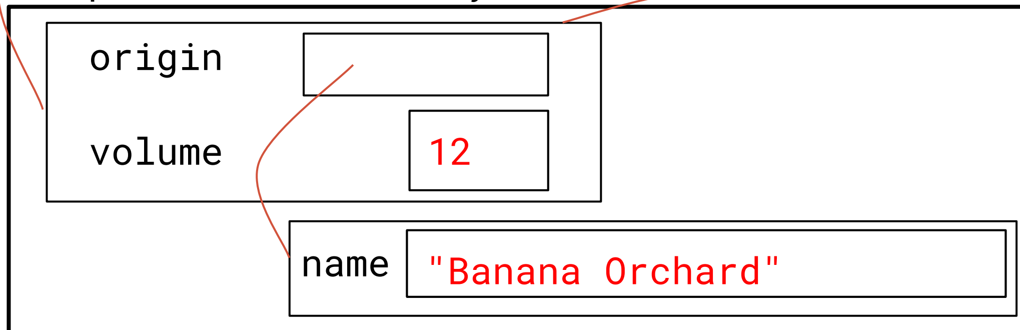
main



exchangeFruit



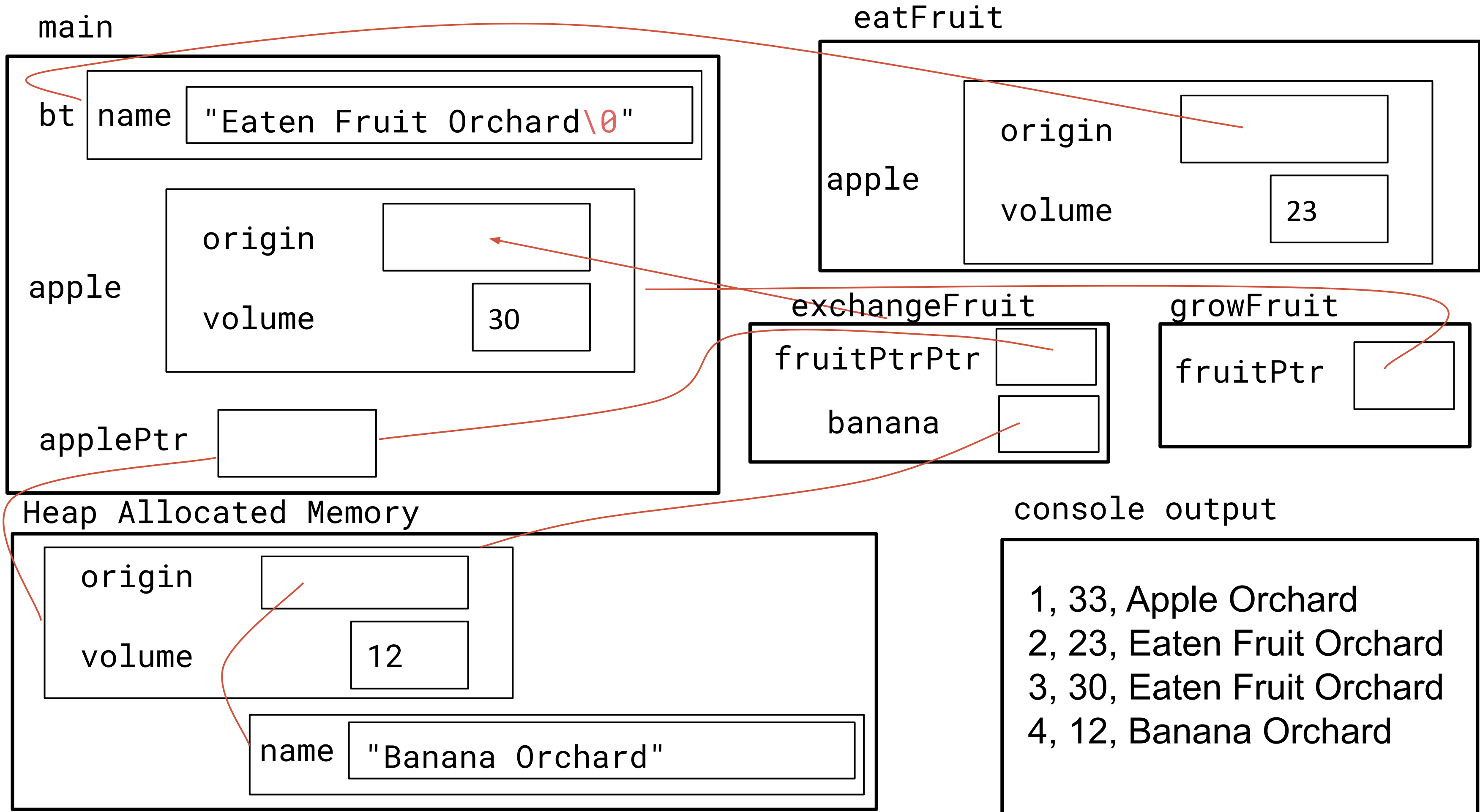
Heap Allocated Memory



console output

1, 33, Apple Orchard
2, 23, Eaten Fruit Orchard
3, 30, Eaten Fruit Orchard
4, 12, Banana Orchard

```
exchangeFruit(&applePtr);
printf("4, %d, %s \n", applePtr->volume,
    applePtr->origin->name);
```


main
bt
name
"Eaten Fruit Orchard\0"
apple
origin
volume
30
applePtr
eatFruit
apple
origin
volume
23
exchangeFruit
fruitPtrPtr
banana
growFruit
fruitPtr
Heap Allocated Memory
origin
volume
12
name
"Banana Orchard"
console output
1, 33, Apple Orchard
2, 23, Eaten Fruit Orchard
3, 30, Eaten Fruit Orchard
4, 12, Banana Orchard

Motivation & Tools

- The projects are big, lots of potential for bugs
- **Debugging is a skill that you will need throughout your career**

- gdb (GNU Debugger) is a debugging tool
    - Handles more than just assembly.
    - Lots of helpful features to help with debugging
    - Very useful in tracking undefined behavior

- Valgrind is a memory debugging tool
    - Checks for various memory errors
    - If you are running into odd behavior, running valgrind may point out the cause.

Exercise 1: Debugging with gdb

Segmentation fault

- Causes of segmentation fault
  - Dereferencing uninitialized pointer
  - Null pointer
  - A previously freed pointer
  - Accessing end of an array
  - …
- gdb (GNU Debugger) is very helpful for identifying the source of a segmentation fault
  - Backtrace

Other Esssential gdb Commands

- run <command_line_args>
- backtrace
- frame, up, down
- print <expression>
- quit
- breakpoints
    - (see next slide)

gdb Breakpoints

- Usage:
  - break <function_name>
  - break <filename:line#>
    - Example: break CSE333.c:20
      // ^ sets breakpoint for when Verify333 fails
- Can advance with:
  - continue – resume execution
  - next – execute next line of code, treat functions as one statement
  - step – execute next line of code, stepping into called functions
  - finish – run until current function returns
- More info linked from the course website!

reverse.c

Man pages

- If you are unsure of what a C library function does, use man to find more information.
  - Example: man strcpy

- Note: man also supports various unix commands, but doesn’t hold info for C++

Memory Errors

- Use of uninitialized memory
- Reading/writing memory after it has been freed – Dangling pointers
- Reading/writing to the end of malloc'd blocks
- Reading/writing to inappropriate areas on the stack
- Memory leaks where pointers to malloc'd blocks are lost

Valgrind is your friend!!

Exercise 2: Leaky code and Valgrind Demo

Leaky.c: Prints an array with a given range of values

Given: 2 integers, m & n.
Output: [n, n+1, n+2, …. , m-1, m]

Example:
n = 2
m = 5
Output = [2, 3, 4, 5]

<Demo>

Section exercise

- Handouts.
- Work with a partner, if you wish.
- Look at the expandable vector code in imsobuggy.c.
- First, try to find all the bugs by inspection.
- Then try to use Valgrind on the same code.

Code is located at
https://courses.cs.washington.edu/courses/cse333/21wi/sections/sec02-code/