C Programming: Bugs and debugging

Yipeng Huang

Rutgers University

February 13, 2023

Table of contents

- Announcements
- Strategies for correct software & debugging
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Programming assignment 2: Queues, trees, and graphs
  - `bstLevelOrder.c`: Level order traversal of a binary search tree
  - Binary search tree: `BSTNode`, `insert()`, `delete()`
  - Linked list implementation of a queue: `QueueNode`, `Queue`, `enqueue()`, `dequeue()`
  - Using `graphutils.h`

Canvas timed quiz 3 and programming assignment 2

Quiz 3

1. Due Friday 2/13.
2. 45 minutes.
3. Two tries.
4. Experimenting and identifying memory bugs.
5. Reviews recent concepts that would be fair game for exams.

Programming assignment 2

1. Due Friday 2/24.
2. More data structures: queues, BSTs, graphs; solidify managing memory.

Reading assignment: CS:APP Chapters 2.1, 2.2, 2.3

All about integers

1. We will launch in to our chapter on representing data in computers
2. First: all about integers, signs, capacities, operations.

Table of contents

- Announcements
- Strategies for correct software & debugging
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Programming assignment 2: Queues, trees, and graphs
  - `bstLevelOrder.c`: Level order traversal of a binary search tree
  - Binary search tree: `BSTNode`, `insert()`, `delete()`
  - Linked list implementation of a queue: `QueueNode`, `Queue`, `enqueue()`, `dequeue()`
  - Using `graphutils.h`

Challenges in CS programming assignments, strategies to get unstuck, resources

In CS 111, 112, 211, what are reasons programming assignments are challenging?

- Not sure where to start.
- It isn't working.
- The CS 211 teachers say that knowing Java helps programming in C, but C is nothing like Java.

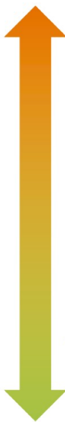
What are strategies to get unstuck?

Lessons and ways in which programming in class is not like the real world.

- Coding deliberately is important. Have a plan. Understand the existing code. Test assumptions. Don't code by trial and error.
- Less code is better, and more likely to be correct.
- Reading code is as important and takes more time than writing code.
- In the real world, people work in teams. Here, assignments are individual work. If the class can collectively crack a difficult assignment via Piazza, that is a more realistic model of real-world engineering.

Approaches to Software Reliability

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - "lint" tools, static analysis
  - Fuzzers, random testing
- Mathematical
  - Sound type systems
  - Formal verification



Less "formal": Lightweight, inexpensive techniques (that may miss problems)

This isn't an either/or tradeoff… a spectrum of methods is needed!

Even the most "formal" argument can still have holes:

- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth: *"Beware of bugs in the above code; I have only proved it correct, not tried it."*

More "formal": eliminate *with certainty* as many problems as possible.

From: https://www.seas.upenn.edu/~cis500/current/lectures/lec01.pdf

Strategies for debugging

Reduce to minimum example

- Check your assumptions.
- Use minimum example as basis for searching for help.

Debugging techniques

- Use assertions.
- Use debugging tools: Valgrind, Address Sanitizer, GDB.
- Use debugging printf statements.

Table of contents

Announcements

Strategies for correct software & debugging

Bugs and debugging related pointers, malloc, free

- Failure to free
- Use after free
- Pointer aliasing
- Pointer typing

Bugs and debugging related C memory model

- Non existent memory
- Returning null pointer

Programming assignment 2: Queues, trees, and graphs

- `bstLevelOrder.c`: Level order traversal of a binary search tree
- Binary search tree: `BSTNode`, `insert()`, `delete()`
- Linked list implementation of a queue: `QueueNode`, `Queue`, `enqueue()`, `dequeue()`
- Using `graphutils.h`

Failure to free

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main () {
5
6      int* pointer0 = malloc(sizeof(int));
7      *pointer0 = 100;
8      printf("*pointer0 = %d\n", *pointer0);
9
10 }
```

Note: calloc() functions like malloc(), but calloc() initializes memory to zero while malloc() offers no such guarantee.

Memory leaks

Have you ever had to restart software or hardware to recover it?

Debug by compilation in GCC, running with Valgrind, Address Sanitizer

Use after free

```
1     int* pointer0 = malloc(sizeof(int));
2
3     printf("pointer0 = %p\n", pointer0);
4     *pointer0 = 100;
5     printf("*pointer0 = %d\n", *pointer0);
6
7     free(pointer0);
8     pointer0 = NULL;
9
10    printf("pointer0 = %p\n", pointer0);
11    *pointer0 = 10;
12    printf("*pointer0 = %d\n", *pointer0);
```

Dangling pointers

- One defensive programming style is to set any freed pointer to NULL.
- Debug by running with Valgrind, Address Sanitizer.

Pointer aliasing

```
1     int* pointer0 = malloc(sizeof(int));
2     int* pointer1 = pointer0;
3
4     *pointer0 = 100;
5     printf("*pointer1 = %d\n", *pointer1);
6
7     *pointer0 = 10;
8     printf("*pointer1 = %d\n", *pointer1);
9
10    free(pointer0);
11    pointer0 = NULL;
12
13    *pointer1 = 1;
14    printf("*pointer1 = %d\n", *pointer1);
```

Debug by running with Valgrind, Address Sanitizer

Pointer aliasing and overhead of garbage collection

- Java garbage collection tracks dangling pointers but costs performance.
- C requires programmer to manage pointers but is more difficult.

Pointer typing

```
1     unsigned char n = 2;
2     unsigned char m = 3;
3
4     unsigned char ** p;
5     p = calloc( n, sizeof(unsigned char) );
6
7     for (int i = 0; i < n; i++)
8         p[i] = calloc( m, sizeof(unsigned char) );
9
10    for (int i = 0; i < n; i++)
11        for (int j = 0; j < m; j++) {
12            p[i][j] = 10*i+j;
13            printf("p[%d][%d] = %d\n", i, j, p[i][j]);
14        }
```

Defend using explicit pointer casting.

Table of contents

- Announcements
- Strategies for correct software & debugging
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Programming assignment 2: Queues, trees, and graphs
  - `bstLevelOrder.c`: Level order traversal of a binary search tree
  - Binary search tree: `BSTNode`, `insert()`, `delete()`
  - Linked list implementation of a queue: `QueueNode`, `Queue`, `enqueue()`, `dequeue()`
  - Using `graphutils.h`

Non existent memory

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

int main () {

  int **x = malloc(sizeof(int*));
  **x = 8;
  printf("x = %p\n", x);
  printf("*x = %p\n", *x);
  printf("**x = %d\n", **x);
  fflush(stdout);

}
```

Debug by running with Valgrind, Address Sanitizer

Returning null pointer

```
1
2 int* returnsNull () {
3   int val = 100;
4   return &val;
5 }
6
7 int main () {
8
9   int* pointer = returnsNull();
10  printf("pointer = %p\n", pointer);
11  printf("*pointer = %d\n", *pointer);
12
13 }
```

Prevent using -Werror compilation flag.

Table of contents

- Announcements
- Strategies for correct software & debugging
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Programming assignment 2: Queues, trees, and graphs
  - `bstLevelOrder.c`: Level order traversal of a binary search tree
  - Binary search tree: `BSTNode`, `insert()`, `delete()`
  - Linked list implementation of a queue: `QueueNode`, `Queue`, `enqueue()`, `dequeue()`
  - Using `graphutils.h`

Programming assignment 2: Queues, trees, and graphs

Programming Assignment 2 parts

1. bstLevelOrder: needs a queue (available in pa2/queue, will discuss today)
2. edgelist: will discuss today
3. isTree: needs either DFS (stack) or BFS (queue)
4. solveMaze: needs either DFS (stack) or BFS (queue)
5. mst: a greedy algorithm
6. findCycle: needs either DFS (stack) or BFS (queue)
7. matChainMul: another dynamic programming problem and prelude to integer operations

Binary search tree

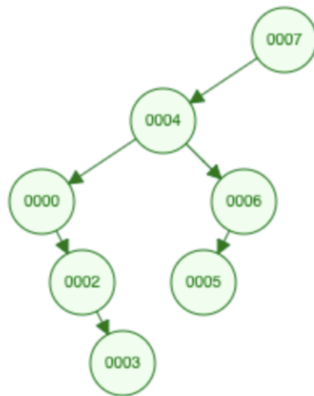


Figure: BST with input sequence 7, 4, 7, 0, 6, 5, 2, 3. Duplicates ignored.

Binary search tree level order traversal

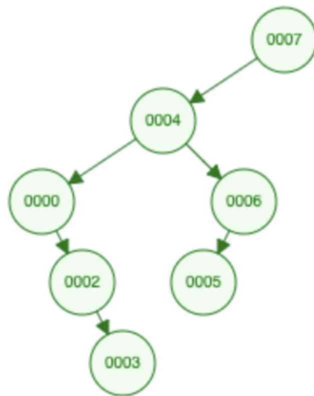


Figure: Level order, left-to-right traversal would return 7, 4, 0, 6, 2, 5, 3.

Binary search tree traversal orders

Breadth-first

- For example: level-order.
- Needs a queue (first in first out).
- Today in class we will build a BST and a Queue.

Depth-first

- For example: in-order traversal, reverse-order traversal.
- Needs a stack (first in last out).

typedef

Why types are important

- Natural language has nouns, verbs, adjectives, adverbs.
- Type safety.
- Interpretation vs. compilation.

struct

arrays vs structs

- Arrays group data of the same type. The `[]` operator accesses array elements.
- Structs group data of different type. The `.` operator accesses struct elements.

These are equivalent; the latter is shorthand:

```
BSTNode* root;
```

- `(*root).key = key;`
- `root->key = key;`

When structs are passed to functions, they are passed BY VALUE.

BSTNode

```
typedef struct BSTNode BSTNode;
struct BSTNode {
    int key;
    BSTNode* l_child; // nodes with smaller key will be in left s
    BSTNode* r_child; // nodes with larger key will be in right s
};
```

QueueNode, Queue

```
// queue needed for level order traversal
typedef struct QueueNode QueueNode;
struct QueueNode {
    BSTNode* data;
    QueueNode* next; // pointer to next node in linked list
};
typedef struct Queue {
    QueueNode* front; // front (head) of the queue
    QueueNode* back; // back (tail) of the queue
} Queue;
```

Let's implement `enqueue()`

`https://visualgo.net/en/queue`

- First, consider if queue is empty.
- Then, consider if queue is not empty. Only need to touch back (tail) of the queue.

Let's implement `deque()`

`https://visualgo.net/en/queue`

- First, consider if queue will become empty.
- Then, consider if queue will not not empty. Only need to touch front (head) of the queue.

Subtle point: why are the function signatures (return, parameters) of `enqueue()` and `dequeue()` the way they are?

Using `graphutils.h`

- The adjacency list representation
- The edgelist representation
- The query