C Programming: Memory, bugs, debugging

Yipeng Huang

Rutgers University

February 10, 2021

Table of contents

Quizzes and programming assignments

Short quiz 2

Has been out, due Friday February 11, 11:59 pm.

Programming assignment 1

Now due Friday February 11, 11:59 pm. Deadline firm.

Programming assignment 2

- Now out, due Thursday February 24, 11:59 pm.
- More review of CS 111 and 112 concepts in the C language.
- Stacks. Queues. BSTs. Graph algorithms.

Reading and class session plan

Reading: CS:APP Chapter 2

- Chapter 2: Representing and manipulating information
- First focus on Chapters 2.1-2.3, integers

Class session plan

1. Today: Memory, memory errors, bugs, debugging.
2. Next Tuesday: Discussion of PA2. Implementing a stack data structure.
3. Next Thursday: Representing integers in computer architecture.

What computer architecture concepts have we explored so far?

Recall: computer architecture is the science and engineering of the computer software-hardware interface.

What computer architecture concepts have we explored so far?

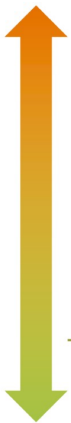
Recall: computer architecture is the science and engineering of the computer software-hardware interface.

- Everything (characters, strings, files, memory addresses) is a binary number.
- A program data is stored somewhere in memory.

Table of contents

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

Memory: stack and heap

Memory: stack and heap

- Everything you malloc() or calloc() lives in a region of memory called the heap.
- Everything else lives on the stack, and are subject to scoping rules (like Java).

Table of contents

Failure to free

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

int main () {

    int* pointer0 = malloc(sizeof(int));
    *pointer0 = 100;
    printf("*pointer0 = %d\n", *pointer0);

}
```

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

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

Why matMul() is written that way

The matMul function signature in the provided example code.

```
void matMul (
    unsigned int l,
    unsigned int m,
    unsigned int n,
    int** matrix_a,
    int** matrix_b,
    int** matMulProduct
);
```

A more "natural" function signature with return. How to implement?

```
int** matMul (
    unsigned int l,
    unsigned int m,
    unsigned int n,
    int** matrix_a,
    int** matrix_b
);
```

Why matMul() is written that way

The matMul function signature in the provided example code. Caller of matMul allocates memory.

```
1 void matMul (
2     unsigned int l,
3     unsigned int m,
4     unsigned int n,
5     int** matrix_a,
6     int** matrix_b,
7     int** matMulProduct
8 );
```

Suppose we want matMul() to be in charge of allocating memory. How to implement?

```
1 void matMul (
2     unsigned int l,
3     unsigned int m,
4     unsigned int n,
5     int** matrix_a,
6     int** matrix_b,
7     int*** matMulProduct
8 );
```