

C Programming: Bugs and debugging

Yipeng Huang

Rutgers University

February 9, 2021

Table of contents

Announcements

- Programming assignments

- Looking ahead

Strategies for correct software & debugging

- Strategies for correct software

- Strategies for 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

Understanding pass-by-value and pass-by-reference

Programming assignments

Programming assignment 1

- ▶ Due in two days: 11:59pm Thursday, February 11.
- ▶ Find class's frequently asked questions on Piazza.

Programming assignment 2

- ▶ Released on Thursday, February 11.
- ▶ Due after two weeks: February 25.
- ▶ Same techniques in programming C.
- ▶ Review of graph algorithms.

Looking ahead

Lecture plan

1. Today, Tuesday, 2/9: Common mistakes in programming, debugging techniques.
2. Thursday, 2/11: Data representation of integers.
3. Tuesday, 2/16: Data representation of floating point numbers.

Reading assignment

- ▶ Computer Systems: A Programmer's Perspective Chapter 2.

Table of contents

Announcements

- Programming assignments

- Looking ahead

Strategies for correct software & debugging

- Strategies for correct software

- Strategies for 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

Understanding pass-by-value and pass-by-reference

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.

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 statements.

Table of contents

Announcements

- Programming assignments

- Looking ahead

Strategies for correct software & debugging

- Strategies for correct software

- Strategies for 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

Understanding pass-by-value and pass-by-reference

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 }
```

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     }
```

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

Table of contents

Announcements

- Programming assignments

- Looking ahead

Strategies for correct software & debugging

- Strategies for correct software

- Strategies for 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

Understanding pass-by-value and pass-by-reference

Non existent memory

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main () {
5
6     int **x = malloc(sizeof(int*));
7     **x = 8;
8     printf("x = %p\n", x);
9     printf("*x = %p\n", *x);
10    printf("**x = %d\n", **x);
11    fflush(stdout);
12
13 }
```

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

- Programming assignments

- Looking ahead

Strategies for correct software & debugging

- Strategies for correct software

- Strategies for 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

Understanding pass-by-value and pass-by-reference

Understanding pass-by-value and pass-by-reference

In this section, we study the `push()` function for a stack.

The `push()` function needs to make changes to the top of the stack, and return pointers to stack elements such that the elements can later be freed from memory.

We consider four function signatures for `push()` that are incorrect.

1. `void push (char value, struct stack s);`
2. `void push (char value, struct stack* s);`
3. `struct stack push (char value, struct stack s);`
4. `struct stack push (char value, struct stack* s);`

And we consider two function signatures for `push()` that are correct.

5. `void push (char value, struct stack** s);`
6. `struct stack* push (char value, struct stack* s);`

Understanding pass-by-value and pass-by-reference

```
1 void push ( char value, struct stack
    s ) { // bug in signature
2
3     struct stack *bracket = malloc(
        sizeof(struct stack));
4     bracket->data = value;
5     bracket->next = &s;
6
7     s = *bracket;
8
9     return;
10 }
```

```
1 int main () {
2     struct stack s;
3     push( 'S', s );
4     printf ("s.data = %c\n", s.data)
        ;
5 }
```

Version 1. An incorrect function signature for push().

This version of push() completely passes-by-value and has no effect on struct stack s in main(), so s.data is uninitialized.

Understanding pass-by-value and pass-by-reference

```
1 void push ( char value, struct stack
    * s ) { // bug in signature
2
3     struct stack *bracket = malloc(
        sizeof(struct stack));
4     bracket->data = value;
5     bracket->next = s;
6
7     s = bracket;
8
9     return;
10 }
```

```
1 int main () {
2     struct stack s;
3     push( 'S', &s );
4     push( 'C', &s );
5     // printf ("s = %p\n", s);
6     struct stack* pointer = &s;
7     printf ("pop: %c\n", pop(&
        pointer));
8     printf ("pop: %c\n", pop(&
        pointer));
9 }
```

Version 2. An incorrect function signature for push().

This version of push() also has no effect on struct stack s in main().

Understanding pass-by-value and pass-by-reference

```
1 struct stack push ( char value,  
    struct stack s ) { // bug in  
    signature  
2  
3    struct stack *bracket = malloc(  
        sizeof(struct stack));  
4    bracket->data = value;  
5    bracket->next = &s;  
6  
7    s = *bracket;  
8  
9    return s;  
10 }
```

Version 3. An incorrect function signature for push () .

Here, we try returning an updated stack data structure via the return type of push (). Lines 3, 7, and 9 will lead to a memory leak (pointer is lost). Line 5 assigns the next pointer to an address &s which will be out of scope in main () .

Understanding pass-by-value and pass-by-reference

```
1 struct stack push ( char value,  
    struct stack* s ) { // bug in  
    signature  
2  
3    struct stack *bracket = malloc(  
        sizeof(struct stack));  
4    bracket->data = value;  
5    bracket->next = s;  
6  
7    s = bracket;  
8  
9    return *s;  
10 }
```

```
1 int main () {  
2     struct stack s;  
3     s = push( 'S', &s );  
4     printf ("s.data = %c\n", s.data)  
        ;  
5     s = push( 'C', &s );  
6     printf ("s.data = %c\n", s.data)  
        ;  
7 }
```

Version 4. An incorrect function signature for push().

Here, we again try returning an updated stack data structure via the return type of push(). Lines 3, 7, and 9 will still lead to a memory leak (pointer is lost).

Understanding pass-by-value and pass-by-reference

```
1 void push ( char value, struct stack
    ** s ) {
2
3     struct stack *bracket = malloc(
        sizeof(struct stack));
4     bracket->data = value;
5     bracket->next = *s;
6
7     *s = bracket;
8
9     return;
10 }
```

```
1 int main () {
2     struct stack* s;
3     push( 'S', &s );
4     push( 'C', &s );
5     printf ("pop: %c\n", pop(&s));
6     printf ("pop: %c\n", pop(&s));
7 }
```

Version 5. A correct function signature for push().

struct stack* s in main() updates by passing the struct stack * parameter via pass-by-reference, leading to the push() signature that you see here. This matches the signature that you see for the pop() function.

Understanding pass-by-value and pass-by-reference

```
1 struct stack* push ( char value,  
    struct stack* s ) {  
2  
3     struct stack *bracket = malloc(  
        sizeof(struct stack));  
4     bracket->data = value;  
5     bracket->next = s;  
6  
7     s = bracket;  
8  
9     return s;  
10 }
```

```
1 int main () {  
2     struct stack* s;  
3     s = push( 'S', s );  
4     s = push( 'C', s );  
5     printf ("pop: %c\n", pop(&s));  
6     printf ("pop: %c\n", pop(&s));  
7 }
```

Version 6. A correct function signature for push () .

struct stack* s updates via the return type of push () in main (), lines 3 and 4. Side note, this is similar to the function signature BSTNode* insert (BSTNode* root, int key) shown in class on 2/4. Side note, pop () needs to return the character data, so pop () cannot have a similar function signature.