C Programming: Structs, functions, memory debugging

Yipeng Huang

Rutgers University

February 8, 2024

Table of contents

Canvas timed quiz 3 and programming assignment 1

Programming assignment 1

1. Due Friday 2/9.
2. Arrays, pointers, recursion, beginning data structures.

Table of contents

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.

Table of contents

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

```c
void push ( char value, struct stack
      s ) { // bug in signature

    struct stack *bracket = malloc(
        sizeof(struct stack));
    bracket->data = value;
    bracket->next = &s;

    s = *bracket;

    return;
}
```

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

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

```c
void push ( char value, struct stack
     * s ) { // bug in signature

    struct stack *bracket = malloc(
        sizeof(struct stack));
    bracket->data = value;
    bracket->next = s;

    s = bracket;

    return;
}
```

```c
int main () {
    struct stack s;
    push( 'S', &s );
    push( 'C', &s );
    // printf ("s = %p\n", s);
    struct stack* pointer = &s;
    printf ("pop: %c\n", pop(&
        pointer));
    printf ("pop: %c\n", pop(&
        pointer));
}
```

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

```c
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 }
```

```c
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

```c
void push ( char value, struct stack
     ** s ) {

    struct stack *bracket = malloc(
        sizeof(struct stack));
    bracket->data = value;
    bracket->next = *s;

    *s = bracket;

    return;
}
```

```c
int main () {
    struct stack* s;
    push( 'S', &s );
    push( 'C', &s );
    printf ("pop: %c\n", pop(&s));
    printf ("pop: %c\n", pop(&s));
}
```

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.

Table of contents

`matMul.c`: Function for matrix-matrix multiplication

What to pay attention to

- How `matMulProduct` result is given back to caller of function.
- How and where memory is allocated and freed.

Why matMul() is written that way

The matMul function signature in the provided example code.

```c
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?

```c
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.

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

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

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

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

```
    int* pointer0 = malloc(sizeof(int));

    printf("pointer0 = %p\n", pointer0);
    *pointer0 = 100;
    printf("*pointer0 = %d\n", *pointer0);

    free(pointer0);
    pointer0 = NULL;

    printf("pointer0 = %p\n", pointer0);
    *pointer0 = 10;
    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

```
int* pointer0 = malloc(sizeof(int));
int* pointer1 = pointer0;

*pointer0 = 100;
printf("*pointer1 = %d\n", *pointer1);

*pointer0 = 10;
printf("*pointer1 = %d\n", *pointer1);

free(pointer0);
pointer0 = NULL;

*pointer1 = 1;
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

```
unsigned char n = 2;
unsigned char m = 3;

unsigned char ** p;
p = calloc( n, sizeof(unsigned char) );

for (int i = 0; i < n; i++)
    p[i] = calloc( m, sizeof(unsigned char) );

for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++) {
        p[i][j] = 10*i+j;
        printf("p[%d][%d] = %d\n", i, j, p[i][j]);
    }
```

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

```c

int* returnsNull () {
  int val = 100;
  return &val;
}

int main () {

  int* pointer = returnsNull();
  printf("pointer = %p\n", pointer);
  printf("*pointer = %d\n", *pointer);

}
```

Prevent using -Werror compilation flag.