

C Programming: 2D arrays, function returns

Yipeng Huang

Rutgers University

September 23, 2025

Table of contents

Announcements

Programming assignment 1

`pointers.c`: A lab exercise for pointers, arrays, and memory

Lesson 6: 2D arrays

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

Stack data structure: `struct`, `push()`, `pop()`

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

Programming assignment 1

Programming assignment 1

1. quickselect: arrays and array indexing
2. editDistance: 2D arrays
3. balanced: a basic stack data structure

Table of contents

Announcements

Programming assignment 1

`pointers.c`: A lab exercise for pointers, arrays, and memory

Lesson 6: 2D arrays

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

Stack data structure: `struct`, `push()`, `pop()`

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

Lesson 6: 2D arrays

Table of contents

Announcements

Programming assignment 1

`pointers.c`: A lab exercise for pointers, arrays, and memory

Lesson 6: 2D arrays

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

Stack data structure: `struct`, `push()`, `pop()`

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

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.

```
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 );
```

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

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

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 );
```

Table of contents

Announcements

Programming assignment 1

`pointers.c`: A lab exercise for pointers, arrays, and memory

Lesson 6: 2D arrays

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

Stack data structure: `struct`, `push()`, `pop()`

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

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

Announcements

Programming assignment 1

`pointers.c`: A lab exercise for pointers, arrays, and memory

Lesson 6: 2D arrays

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

Stack data structure: `struct`, `push()`, `pop()`

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 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.

