C Programming: Functions, pass-by-reference, returns

Yipeng Huang

Rutgers University

February 6, 2024
Table of contents

Announcements
Canvas timed quiz 3 and programming assignment 1

pointers.c: A lab exercise for pointers, arrays, and memory
Lesson 9: Passing an array leads to passing-by-reference
Lesson 10: How the stack works; recursion example

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

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

matMul.c: Function for matrix-matrix multiplication
Programming assignment 1

1. Due Friday 2/9.
2. Arrays, pointers, recursion, beginning data structures.
Announcements
Canvas timed quiz 3 and programming assignment 1

pointers.c: A lab exercise for pointers, arrays, and memory
Lesson 9: Passing an array leads to passing-by-reference
Lesson 10: How the stack works; recursion example

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

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

matMul.c: Function for matrix-matrix multiplication
Lesson 9: Passing an array leads to passing-by-reference
Lesson 10: How the stack works; recursion example

<table>
<thead>
<tr>
<th>Low addresses</th>
<th>Global / static data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Heap grows downward</td>
</tr>
<tr>
<td></td>
<td>Dynamic memory allocation</td>
</tr>
<tr>
<td>High addresses</td>
<td>Stack grows upward</td>
</tr>
<tr>
<td></td>
<td>Local variables, parameters</td>
</tr>
</tbody>
</table>

**Table**: Memory structure
Announcements
Canvas timed quiz 3 and programming assignment 1

pointers.c: A lab exercise for pointers, arrays, and memory
Lesson 9: Passing an array leads to passing-by-reference
Lesson 10: How the stack works; recursion example

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

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

matMul.c: Function for matrix-matrix multiplication
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:

```c
BSTNode* root;
▶ (*root).key = key;
▶ root->key = key;
```

When structs are passed to functions, they are passed BY VALUE.
Table of contents

Announcements
Canvas timed quiz 3 and programming assignment 1

pointers.c: A lab exercise for pointers, arrays, and memory
Lesson 9: Passing an array leads to passing-by-reference
Lesson 10: How the stack works; recursion example

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

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

matMul.c: Function for matrix-matrix multiplication
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.
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;
}

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));
    return;
}

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

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
struct stack 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 *s;
}
```

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


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

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

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

    return s;
}
```

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

**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

Announcements
Canvas timed quiz 3 and programming assignment 1

pointers.c: A lab exercise for pointers, arrays, and memory
Lesson 9: Passing an array leads to passing-by-reference
Lesson 10: How the stack works; recursion example

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

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

matMul.c: Function for matrix-matrix multiplication
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.

```c
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?

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