

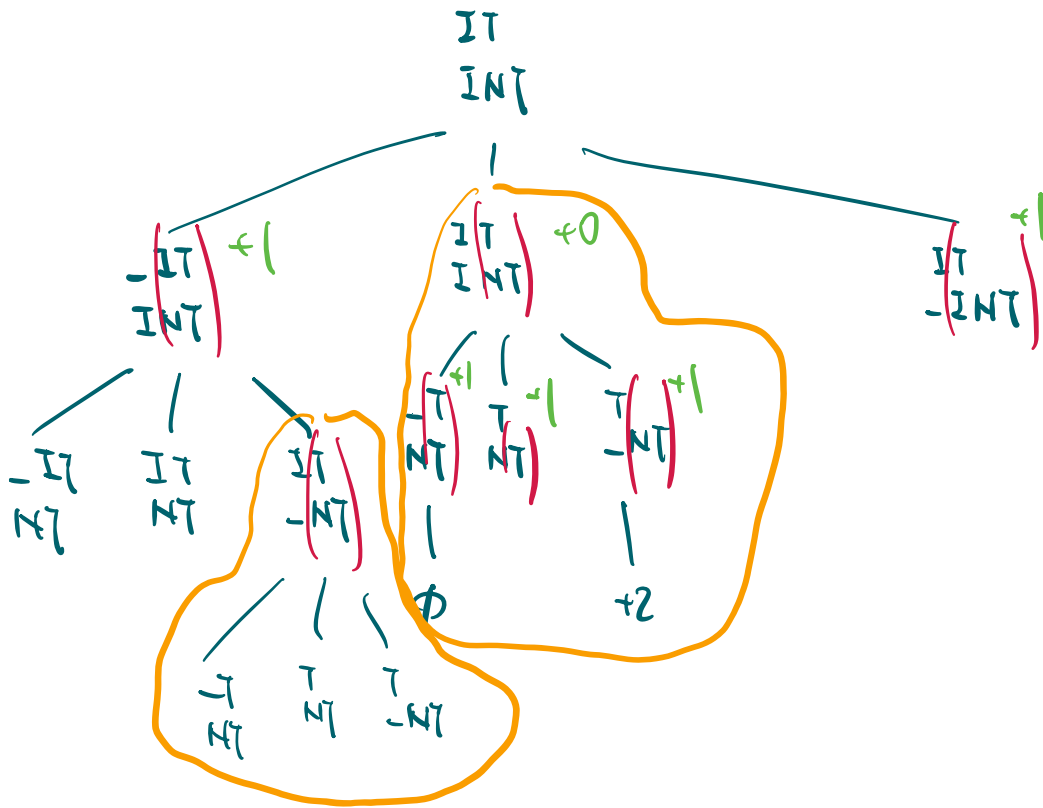
# C Programming: Functions and Recursion

Yipeng Huang

Rutgers University

September 22, 2026

CAT  
TAC



# C Programming: Functions and Recursion

## Announcements

Canvas timed quiz 2 and programming assignment 1

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

Lesson 6: Arrays are just places in memory

Lesson 6: 2D arrays

Lesson 7: Passing-by-value

Lesson 8: Passing-by-reference

Lesson 9: Passing an array leads to passing-by-reference

Lesson 10: How the stack works; recursion example

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

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

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

# C Programming: Functions and Recursion

## Announcements

Canvas timed quiz 2 and programming assignment 1

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

Lesson 6: Arrays are just places in memory

Lesson 6: 2D arrays

Lesson 7: Passing-by-value

Lesson 8: Passing-by-reference

Lesson 9: Passing an array leads to passing-by-reference

Lesson 10: How the stack works; recursion example

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

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

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

# Canvas timed quiz 2 and programming assignment 1

## Quiz 2

1. Spanning Friday 9/18 - Friday 9/25.
2. 60 minutes.
3. Two tries.
4. Pointer syntax, arrays, data types.

## Programming assignment 1

1. Due Tuesday 9/29.
2. Arrays, pointers, recursion, and algorithms.

# C Programming: Functions and Recursion

## Announcements

Canvas timed quiz 2 and programming assignment 1

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

Lesson 6: Arrays are just places in memory

Lesson 6: 2D arrays

Lesson 7: Passing-by-value

Lesson 8: Passing-by-reference

Lesson 9: Passing an array leads to passing-by-reference

Lesson 10: How the stack works; recursion example

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

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

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

# C Programming: Functions and Recursion

## Announcements

Canvas timed quiz 2 and programming assignment 1

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

Lesson 6: Arrays are just places in memory

Lesson 6: 2D arrays

Lesson 7: Passing-by-value

Lesson 8: Passing-by-reference

Lesson 9: Passing an array leads to passing-by-reference

Lesson 10: How the stack works; recursion example

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

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

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

## Lesson 6: Arrays are just places in memory

- ▶ Three types of array in C: Fixed length, variable length, heap-allocated.
- ▶ name of array points to first element
- ▶ stack and heap
- ▶ `malloc()` and `free()`
- ▶ using pointers instead of arrays
- ▶ pointer arithmetic
- ▶ `char* argv[]` and `char** argv` are the same thing

## Dynamic Memory Allocation: `malloc` vs. `calloc`

### ▶ **`malloc(size_t size)`**

- ▶ Allocates a specified number of bytes and returns a pointer to the start.
- ▶ Memory is left uninitialized (contains leftover "garbage" data).
- ▶ Slightly faster execution.

### ▶ **`calloc(size_t num, size_t size)`**

- ▶ Contiguous allocation: takes a count and an element size as separate arguments.
- ▶ Automatically zeroes out all allocated bytes.
- ▶ Slower, but safer when zero-initialization is strictly required.

### ▶ **`free(void *ptr)`**

- ▶ The mandatory counterpart to both functions; releases heap memory back to the system.

# Connecting C to Java: `new` and Garbage Collection

- ▶ **Java's `new` keyword is `malloc` under the hood.**
  - ▶ When calling `new`, the JVM uses `malloc` to allocate heap memory sized for the object's class.
  - ▶ It then immediately invokes the constructor to initialize the fields.
- ▶ **Garbage Collection vs. Manual `free()`**
  - ▶ Java's JVM automatically tracks object references and deallocates orphaned objects.
  - ▶ C requires manual deallocation via `free()`.
  - ▶ **Trade-off:** Garbage collection prevents manual memory leaks but introduces runtime overhead, making Java fundamentally slower than equivalent C programs.

# C Programming: Functions and Recursion

## Announcements

Canvas timed quiz 2 and programming assignment 1

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

Lesson 6: Arrays are just places in memory

**Lesson 6: 2D arrays**

Lesson 7: Passing-by-value

Lesson 8: Passing-by-reference

Lesson 9: Passing an array leads to passing-by-reference

Lesson 10: How the stack works; recursion example

`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 memory layout and pointer types

- ▶ A statically declared 2D array (e.g., `int arr[2][3]`) is a two-element array of three-integer arrays.
- ▶ The compiler tracks distinct data types for pointers, even if they point to the exact same hex address:
  - ▶ `&arr`: Pointer to the entire 2D array structure. `sizeof` yields pointer size (e.g., 8 bytes).
  - ▶ `arr`: Pointer to the array of rows. `sizeof` yields total grid size (e.g., 24 bytes).
  - ▶ `*arr`: Pointer to the first row. `sizeof` yields single row size (e.g., 12 bytes).
  - ▶ `**arr`: Dereferences down to the individual `int` element at `[0][0]`.
- ▶ Pointer arithmetic scales increments by the size of the pointed-to type.

# C Programming: Functions and Recursion

## Announcements

Canvas timed quiz 2 and programming assignment 1

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

Lesson 6: Arrays are just places in memory

Lesson 6: 2D arrays

Lesson 7: Passing-by-value

Lesson 8: Passing-by-reference

Lesson 9: Passing an array leads to passing-by-reference

Lesson 10: How the stack works; recursion example

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

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

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

## Lesson 7: Passing-by-value

Using stack and heap picture, understand how pass by value and pass by reference are different.

- ▶ C functions are entirely pass-by-value
- ▶ `swap_pass_by_values()` doesn't actually succeed in swapping two variables.

# C Programming: Functions and Recursion

## Announcements

Canvas timed quiz 2 and programming assignment 1

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

Lesson 6: Arrays are just places in memory

Lesson 6: 2D arrays

Lesson 7: Passing-by-value

**Lesson 8: Passing-by-reference**

Lesson 9: Passing an array leads to passing-by-reference

Lesson 10: How the stack works; recursion example

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

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

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

## Lesson 8: Passing-by-reference

Using stack and heap picture, understand how pass by value and pass by reference are different.

- ▶ You can create the illusion of pass-by-reference by passing pointers
- ▶ `swap_pass_by_references()` does succeed in swapping two variables.

# C Programming: Functions and Recursion

## Announcements

Canvas timed quiz 2 and programming assignment 1

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

Lesson 6: Arrays are just places in memory

Lesson 6: 2D arrays

Lesson 7: Passing-by-value

Lesson 8: Passing-by-reference

Lesson 9: Passing an array leads to passing-by-reference

Lesson 10: How the stack works; recursion example

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

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

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

## Lesson 9: Passing an array leads to passing-by-reference

# C Programming: Functions and Recursion

## Announcements

Canvas timed quiz 2 and programming assignment 1

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

Lesson 6: Arrays are just places in memory

Lesson 6: 2D arrays

Lesson 7: Passing-by-value

Lesson 8: Passing-by-reference

Lesson 9: Passing an array leads to passing-by-reference

Lesson 10: How the stack works; recursion example

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

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

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

## Lesson 10: How the stack works; recursion example

|                |                     |                             |
|----------------|---------------------|-----------------------------|
| Low addresses  |                     | Global / static data        |
|                | Heap grows downward | Dynamic memory allocation   |
| High addresses | Stack grows upward  | Local variables, parameters |

Table: Memory structure

# Stack vs. Heap Memory

- ▶ **The Stack:**

- ▶ Holds fixed-size variables whose dimensions are known at compile time.
- ▶ Memory is managed automatically as functions push and pop.

- ▶ **The Heap:**

- ▶ A dynamic memory region for runtime-sized allocations (e.g., when array sizes depend on user input).
- ▶ Requires explicit, manual memory management by the programmer.

# C Programming: Functions and Recursion

## Announcements

Canvas timed quiz 2 and programming assignment 1

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

Lesson 6: Arrays are just places in memory

Lesson 6: 2D arrays

Lesson 7: Passing-by-value

Lesson 8: Passing-by-reference

Lesson 9: Passing an array leads to passing-by-reference

Lesson 10: How the stack works; recursion example

`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. 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 );
```

---

# 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 );
```

---

# C Programming: Functions and Recursion

## Announcements

Canvas timed quiz 2 and programming assignment 1

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

Lesson 6: Arrays are just places in memory

Lesson 6: 2D arrays

Lesson 7: Passing-by-value

Lesson 8: Passing-by-reference

Lesson 9: Passing an array leads to passing-by-reference

Lesson 10: How the stack works; recursion example

`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.

# C Programming: Functions and Recursion

## Announcements

Canvas timed quiz 2 and programming assignment 1

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

Lesson 6: Arrays are just places in memory

Lesson 6: 2D arrays

Lesson 7: Passing-by-value

Lesson 8: Passing-by-reference

Lesson 9: Passing an array leads to passing-by-reference

Lesson 10: How the stack works; recursion example

`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 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, `pop()` needs to return the character data, so `pop()` cannot have a similar function signature.