C Programming: Memory programming patterns and antipatterns

Yipeng Huang

Rutgers University

September 29, 2026

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

C Programming: Memory programming patterns and antipatterns

Announcements

- Canvas timed quiz 3 and programming assignment 2

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

- 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

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

Canvas timed quiz 3 and programming assignment 2

Quiz 3

1. Spanning Friday 9/25 - Friday 10/2.
2. 60 minutes.
3. Two tries.
4. Data structures using pointers in C.

Programming assignment 2

1. Due Tuesday 10/13.
2. Data structures mastery.

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

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.

Stack Overflow & Debugging

- **The Missing Base Case:** Recursion without a proper base case causes the program to push new function frames indefinitely.
- **Stack Overflow:** Occurs when infinite recursive calls completely exhaust available stack memory.
- **Diagnostics:**
  - **Without Address Sanitizer:** Results in a generic segmentation fault (segfault) or "core dump" (attempting to access invalid memory).
  - **With Address Sanitizer:** The compiler explicitly diagnoses and reports a "stack overflow," drastically simplifying debugging.

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

`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.

Function Contracts and `assert`

- **Function Contracts:**
  - **Preconditions:** What must be true before the function executes (e.g., Matrix A columns must equal Matrix B rows).
  - **Postconditions:** What the function guarantees upon successful completion.
- **Enforcing Contracts in C:**
  - The `assert()` statement evaluates an expression at runtime.
  - If the expression is false (precondition violated), the program halts immediately with a clear error message.
  - Used in `matMul` to mathematically guarantee valid matrix multiplication dimensions.

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.

```
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?

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

Code Style: Memory Symmetry

- **The Problem:** When a function computes a dynamically sized result, who is responsible for allocating and freeing the memory?
- **Anti-Pattern (Delegated Allocation):**
  - Having the called function allocate memory forces the caller to remember to free it.
  - This separation of concerns increases the risk of memory leaks.
- **Best Practice (Symmetric Memory):**
  - The calling function (e.g., `main`) should both **allocate** and **free** the memory in the same scope.
  - This symmetry keeps lifecycle management visible and makes code easier to maintain.

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

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: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

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.

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

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

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

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.

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

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.

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

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        }
```

Defend using explicit pointer casting.

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

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

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 3 and programming assignment 2
- `pointers.c`: A lab exercise for pointers, arrays, and memory
  - 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
- 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

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.