Understanding Memory in C

Stack, Heap, and Pointers: Essentials for Parallel Programming

Pascal Zittlau
pascal.zittlau@ovgu.de

November 7, 2025

Faculty of Computer Science
Otto von Guericke University Magdeburg

Outline

Introduction

The Stack

The Heap

Pointers

Case Study: Debugging `simple.c`

Relevance to Parallel Programming

Conclusion & Best Practices

Why Memory Matters

- Foundation for writing **correct**, **efficient**, and **robust** C programs.

Why Memory Matters

- Foundation for writing **correct**, **efficient**, and **robust** C programs.
- C gives you direct memory control – a double-edged sword!
  - Great power for optimization and low-level tasks
  - Risk of bugs if not handled carefully (crashes, security vulnerabilities, …)

Why Memory Matters

- Foundation for writing **correct**, **efficient**, and **robust** C programs.
- C gives you direct memory control – a double-edged sword!
  - Great power for optimization and low-level tasks
  - Risk of bugs if not handled carefully (crashes, security vulnerabilities, …)
- Crucial for **parallel programming**:
  - How is data shared (or not shared) between threads/processes?
  - Who is responsible for allocating and freeing memory?
  - Avoiding race conditions and deadlocks related to memory access

Why Memory Matters

- Foundation for writing **correct**, **efficient**, and **robust** C programs.
- C gives you direct memory control – a double-edged sword!
    - Great power for optimization and low-level tasks
    - Risk of bugs if not handled carefully (crashes, security vulnerabilities, ...)
- Crucial for **parallel programming**:
    - How is data shared (or not shared) between threads/processes?
    - Who is responsible for allocating and freeing memory?
    - Avoiding race conditions and deadlocks related to memory access
- Today: Understand the "what, where, and how" of memory!

A Program's Memory Layout (Simplified)

Key Areas:

- **Code (Text):** Instructions (read-only).
- **Static/Global Data:** Variables lasting program's lifetime.
- **Stack:** Temporary, for function calls (automatic).
- **Heap:** Flexible, for dynamic data (manual).

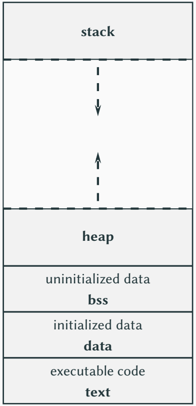


Figure 1: Conceptual Program Memory

Outline

Introduction

The Stack

The Heap

Pointers

Case Study: Debugging `simple.c`

Relevance to Parallel Programming

Conclusion & Best Practices

What is the Stack?

- **Analogy:** A stack of cafeteria trays or tidy in-boxes on a desk
- **LIFO Principle:** Last In, First Out.
  - New task (function call) → New "tray" (stack frame) on top
  - Task finished (function returns) → Its "tray" is removed
- **Contents of a Stack Frame:**
  - Function parameters
  - Local variables declared inside the function
  - Return address
- Memory is managed **automatically** by the compiler

Stack Characteristics

Advantages:

- Fast Allocation/Deallocation
- Automatic Management

Limitations/Properties:

- Scope-Bound Lifetime
- Fixed Size (Compile Time)
- Limited Total Size (Stack Overflow)

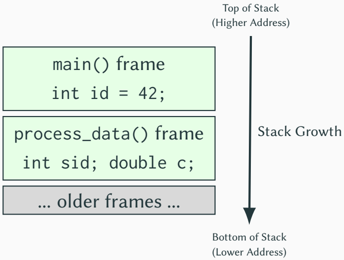


Figure 2: Stack Frames during Function Calls

Stack in Action: C Example

```c
#include <stdio.h>

void process_reading(int sensor_id, double value) {
    double calibrated_value = value * 0.98 + 2.5; // On process_reading's stack
    char label[20];                               // On process_reading's stack
    sprintf(label, "Sensor %d", sensor_id);
    printf("%s: Calibrated = %.2f\n", label, calibrated_value);
} // calibrated_value and label are gone after this function returns

int main() {
    int current_sensor = 7;           // On main's stack
    double raw_reading = 10.5;        // On main's stack

    process_reading(current_sensor, raw_reading);
    // current_sensor and raw_reading still exist here
    return 0;
} // current_sensor and raw_reading are gone after main returns
```

Listing 1: Local variables on the stack

Outline

- Introduction
- The Stack
- The Heap
- Pointers
- Case Study: Debugging `simple.c`
- Relevance to Parallel Programming
- Conclusion & Best Practices

What is the Heap?

- **Analogy:** A large public warehouse or a rental storage facility
- For **dynamic memory allocation**:
  - When you don't know the size of data at compile time
  - When you need data to exist longer than the function that created it
  - For large data structures that might not fit on the stack
- **Manual Management:** You, the programmer, are responsible for:
  - **Requesting** memory (e.g., using `malloc`)
  - **Releasing** memory when done (using `free`)
- The heap is a shared resource for the entire program

Heap Operations: The `malloc` Family

Key functions from `<stdlib.h>`:

- `void* malloc(size_t size)`
  - Allocates `size` bytes of **uninitialized** memory
  - Returns a `void*` pointer to the first byte, or `NULL` if allocation fails
  - **Always cast** the `void*` to your desired pointer type and **check for `NULL`!**
- `void* calloc(size_t num, size_t size)`
  - Like `malloc` but also **initializes** allocated memory to all bits zero
- `void* realloc(void* ptr, size_t new_size)`
  - Changes the size of the memory block pointed to by `ptr` to `new_size`
  - May move the memory block to a new location
  - Content up to `min(old_size, new_size)` is preserved
- `void free(void* ptr)`
  - Deallocates the memory block pointed to by `ptr` (must've been returned by `malloc`, ...)

Heap in Action: malloc and free

```c
#include <stdio.h>
#include <stdlib.h> // malloc, free

int main() {
    int num_elements = 5;
    int *data_array; // pointer will hold address of heap memory

    // 1. Allocate memory on the heap and check if it succeeded
    data_array = (int*)malloc(num_elements * sizeof(int));
    if (data_array == NULL) {
        fprintf(stderr, "Memory allocation failed!\n");
        return 1; // Indicate an error
    }

    // 2. Use allocated memory
    for (int i = 0; i < num_elements; i++) {
        data_array[i] = i * 10;
        printf("data_array[%d] = %d\n", i, data_array[i]);
    }

    // 3. When done, release the memory
    free(data_array);
    data_array = NULL; // Good practice: prevent dangling pointer use
    return 0;
}
```

Heap Management: Your Responsibility!

Common Heap Errors (many of which we'll see in `simple.c`!):

- **Memory Leaks:** Allocating with `malloc` but forgetting to `free`
- **Dangling Pointers:** Using a pointer after the memory it pointed to has been `freed`
- **Double Free:** Calling `free()` twice on the same memory pointer
- **Out-of-Bounds Access:** Writing/reading past the allocated block

The Cardinal Rule

For every `malloc`/`calloc`/`realloc`, there must be a corresponding `free`

Outline

Introduction

The Stack

The Heap

Pointers

Case Study: Debugging `simple.c`

Relevance to Parallel Programming

Conclusion & Best Practices

What are Pointers?

- **Analogy:** A piece of paper with a specific house address written on it
    - The paper itself (pointer variable) is small
    - The address tells you where the actual house (the data) is located
- A **pointer's** value is the **memory address** of another variable(block of memory)
- Essential for using the Heap, passing large data, dynamic data structures, etc.

Pointer Operations: & and *

Declaration: `data_type *pointer_name;`

```c
int *p_num;     // can point to an integer
char *p_char;   // can point to a character
```

Pointer Operations: & and *

Declaration: `data_type *pointer_name;`

```
int *p_num;     // can point to an integer
char *p_char;   // can point to a character
```

Address-of Operator (&): Gets the memory address of a variable.

```
int var = 10;
p_num = &var; // stores address of var
```

Pointer Operations: & and *

Declaration: `data_type *pointer_name;`

```c
int *p_num;     // can point to an integer
char *p_char;   // can point to a character
```

Address-of Operator (&): Gets the memory address of a variable.

```c
int var = 10;
p_num = &var; // stores address of var
```

Dereference Operator (*): Accesses the value at the address stored in the pointer.

```c
int var = 10;
p_num = &var; // stores address of var

printf("Value: %d\n", *p_num); // Prints 10
*p_num = 20; // var is now 20
```

Pointer Operations: & and *

Declaration: `data_type *pointer_name;`

```
int *p_num;     // can point to an integer
char *p_char;   // can point to a character
```

Address-of Operator (&): Gets the memory address of a variable.

```
int var = 10;
p_num = &var; // stores address of var
```

Dereference Operator (*): Accesses the value at the address stored in the pointer.

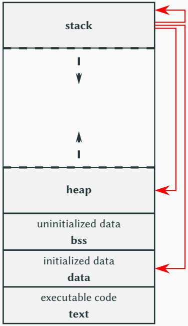
```
int var = 10;
p_num = &var; // stores address of var

printf("Value: %d\n", *p_num); // Prints 10
*p_num = 20; // var is now 20
```

NULL Pointers: Points to no valid memory. Initialize to `NULL`, set to `NULL` after `free`.

Pointers and Memory Regions

- A pointer variable itself (e.g., `int *p;`) lives on the **stack** (static data if global/static)
- The data it **points to** can be on the **stack**, **heap**, or in **static/global data**



Outline

Introduction

The Stack

The Heap

Pointers

Case Study: Debugging `simple.c`

Relevance to Parallel Programming

Conclusion & Best Practices

Revisiting simple.c

```c
// int* mistake1(void);
// int* mistake2(void);
// int* mistake3(void);
// int* mistake4(void); // May NOT allocate memory directly

int main(void) {
    /* Do NOT modify the following line. */
    int* p[4] = { &mistake1()[1], &mistake2()[1], mistake3(), mistake4()};

    printf("1: %d\n", *p[0]);
    printf("2: %d\n", *p[1]);
    printf("3: %d\n", *p[2]);
    printf("4: %d\n", *p[3]);

    /* Add the correct calls to free() here. */
    // free(p[1]); /* What was the correct pointer here? FIXME */
    return 0;
}
```

Listing 2: Original (buggy) `main()` and function signatures from `simple.c`

`mistake1()`: Pointer to Stack Memory

Original Code:

```c
int* mistake1(void) {
    int buf[] = { 1, 1, 2, 3, 4, 5 };
    return buf; // or &buf[0]
}
```

`mistake1()`: Pointer to Stack Memory

Original Code:

```c
int* mistake1(void) {
    int buf[] = { 1, 1, 2, 3, 4, 5 };
    return buf; // or &buf[0]
}
```

Problem:

- buf is a local array, stored on `mistake1`'s **stack frame**
- Stack frame is destroyed when `mistake1` returns
- The returned pointer becomes a **dangling pointer** – it points to invalid memory
- Accessing `*p[0]` in `main` is undefined behavior

`mistake1()`: Pointer to Stack Memory

Solution: Allocate buffer on the **heap** using `malloc`

```c
int* mistake1(void) {
    int* buf = malloc(sizeof(int) * 6);
    if (buf == NULL) return NULL; // Check allocation

    // Initialize buf (e.g., buf[0]=1, buf[1]=1, ...)
    int init_vals[] = {1,1,2,3,4,5};
    for(int i=0; i<6; ++i) buf[i] = init_vals[i];
    return buf;
}
```

Listing 3: Corrected `mistake1()`

`mistake2()`: Allocation Size & Indexing

Original Code:

```c
int* mistake2(void) {
    int* buf = malloc(sizeof(char) * 4);  // Problem 1
    buf[2] = 2;                           // Problem 2 (related to main)
    return buf;
}
```

Problems:

- **Size Mismatch**: Allocates 4 `* sizeof(char)` bytes, but buf is an `int*`
- **Out of Bounts**: Writing `buf[2]` (an `int`) writes out of bounds of the char-sized allocation
- **Indexing vs. `main`**: `main` uses `p[1] = &mistake2()[1]`, expecting the relevant data at index 1 of an `int` array. Original stores at index 2

`mistake2()`: Allocation Size & Indexing

Solution: Allocate correct size for `int`s, store at the index `main` expects

```c
int* mistake2(void) {
    int* buf = malloc(sizeof(int) * 4); // Correct size
    if (buf == NULL) return NULL;
    buf[1] = 2; // Store at index 1 for main's p[1]
    return buf;
}
```

Listing 4: Corrected `mistake2()`

`mistake3()`: Out-of-Bounds & Use-After-Free

Original Code:

```c
int* mistake3(void) {
    int* buf = malloc(sizeof(char) * 4);  // Problem 1: Size
    buf[4] = 3;                           // Problem 2: Out-of-bounds
    free(buf);                            // Problem 3
    return buf;                           // Returning dangling pointer
}
```

Problems:

- **Size Mismatch**: Same as `mistake2`
- **Out-of-Bounds Write**: Same as `mistake2`
- **Use-After-Free**: `free(buf)` deallocates the memory. Returning buf afterwards means returning a **dangling pointer**

`mistake3()`: Out-of-Bounds & Use-After-Free

Solution: Allocate correctly, write in-bounds, return *before* `free` (or let `main` free it)

```c
int* mistake3(void) {
    int* buf = malloc(sizeof(int) * 4); // Correct size
    if (buf == NULL) return NULL;
    buf[0] = 3; // Store at index 0 for main's p[2]
    return buf; // Main will free this
}
```

Listing 5: Corrected `mistake3()`

`mistake4()`: Invalid Pointer Conversion

Original Code: (*Constraint: May NOT allocate memory directly*)

```c
int* mistake4(void) {
    // This function may NOT allocate memory directly
    int* buf = (int*)&mistake2; // PROBLEM!
    buf[0] = 4;
    return buf;
}
```

Problem:

- `(int*)&mistake2` casts the **address of the function `mistake2`** to an `int*`
- `buf[0] = 4` then attempts to **write into the code segment** of the program
- This is undefined behavior, almost guaranteed to cause a **segmentation fault** (crash)

`mistake4()`: Invalid Pointer Conversion

Solution: Fulfill constraint by *calling* a function that allocates, or use static memory

```c
int* mistake4(void) {
    // This function may NOT allocate memory directly
    // So, let another function (like mistake2) do it.
    int* buf = mistake2(); // Call a function that does malloc
    if (buf == NULL) return NULL;
    buf[0] = 4;

    return buf;

    // Alt:
    // static int static_buf = 4;
    // return &static_buf;
}
```

Listing 6: Corrected `mistake4()`

`main()`: Pointer Arithmetic

The setup in `main`:

```
int* p[4] = { &mistake1()[1], &mistake2()[1], mistake3(), mistake4()};
```

Breaking down `p[0] = &mistake1()[1]`:

- `mistake1()` (corrected) returns `base_addr1` (start of heap block).
- `mistake1()[1]` is equivalent to `*(base_addr1 + 1)`.
- `&mistake1()[1]` is `&(*(base_addr1 + 1))`, which simplifies to `base_addr1 + 1`.
- So, `p[0]` points to the **second `int`** of the allocated block.
- Similarly, `p[1]` (from `&mistake2()[1]`) points to the **second `int`** of its block.
- `p[2]` (from `mistake3()`) points to the **start** of its block.
- `p[3]` (from `mistake4()`, which called `mistake2()`) points to the **start** of its block.

`main()`: Correct `free()`

```c
// For p[0], which points to buf[1] from mistake1's allocation
free(p[0] - 1); // Pointer arithmetic to get start of block

// For p[1], which points to buf[1] from mistake2's allocation
free(p[1] - 1); // Pointer arithmetic

// For p[2], which points to start of mistake3's allocation
free(p[2]);

// For p[3], which points to start of allocation from mistake4 (via mistake2)
free(p[3]);
```

Listing 7: Corrected `free()` calls in `main()`

`main()`: Correct `free()`

```
// For p[0], which points to buf[1] from mistake1's allocation
free(p[0] - 1); // Pointer arithmetic to get start of block

// For p[1], which points to buf[1] from mistake2's allocation
free(p[1] - 1); // Pointer arithmetic

// For p[2], which points to start of mistake3's allocation
free(p[2]);

// For p[3], which points to start of allocation from mistake4 (via mistake2)
free(p[3]);
```

Listing 8: Corrected `free()` calls in `main()`

Attention!
`free()` **MUST** be called with the original pointer returned by `malloc/calloc/realloc`.

Outline

Introduction

The Stack

The Heap

Pointers

Case Study: Debugging `simple.c`

Relevance to Parallel Programming

Conclusion & Best Practices

Memory in Parallel Contexts

How Stack, Heap, and Pointers relate to OpenMP, Pthreads, and MPI:

OpenMP / Pthreads (Shared Memory)

- Private Stacks
- Shared Heap (often needs **synchronization**)
- Pointers can point to shared heap or private stack

Memory in Parallel Contexts

How Stack, Heap, and Pointers relate to OpenMP, Pthreads, and MPI:

OpenMP / Pthreads (Shared Memory)

- Private Stacks
- Shared Heap (often needs **synchronization**)
- Pointers can point to shared heap or private stack

MPI (Distributed Memory)

- Separate address spaces
- Pointers are NOT shareable between processes
- Data Transfer: Data is explicitly **sent**
- Each process uses `malloc`/`free` for its own buffers

Outline

Introduction

The Stack

The Heap

Pointers

Case Study: Debugging `simple.c`

Relevance to Parallel Programming

Conclusion & Best Practices

Best Practices for Memory Management in C

- Initialize pointers
- Check `malloc` return
- `free` what you allocate
- Set pointers to `NULL` after `free`
- Beware of scope: don’t return pointers to local stack variables
- Understand ownership: Who `free`s what?
- Use tools: `valgrind` (on Linux) for memory leaks/errors
- Be disciplined!

Questions?

Thank You!

Any questions on stack, heap, pointers, `simple.c`, or parallel programming memory?