Phase 0: Foundations

Status: □ In Progress **Started:** 2024-11-11 **Target Duration:** 4 weeks (hardcore mode)

> ***Philosophy:*** *No checkbox learning. Every exercise must force you to think deeply about memory. If you can complete something without understanding what's happening at the memory level, it's not hard enough.*

Learning Objectives

By the end of this phase, you must be able to:

- [ ] Read and write C code involving complex pointer manipulations
- [ ] Debug segfaults and memory issues using proper tools
- [ ] Explain what happens at the memory/syscall level for malloc/free
- [ ] Read basic x86-64 assembly generated from C code
- [ ] Implement a simple memory allocator from scratch
- [ ] Use gdb, valgrind, and AddressSanitizer fluently

□ Tools Setup (Week 1, Day 1 - NON-NEGOTIABLE)

Required Tools

```
# Compiler with debug support
gcc --version  # or clang

# Debugger
gdb --version  # or lldb on Mac

# Memory debugging
valgrind --version  # Use Docker on Mac if needed

# Compilation flags you'll use CONSTANTLY
gcc -Wall -Wextra -g -O0 -fsanitize=address program.c
```

Setup Checklist

- [ ] Install gcc/clang with debug symbols
- [ ] Install gdb (Linux) or lldb (Mac)
- [ ] Install valgrind (or setup Docker container for Mac)
- [ ] Test AddressSanitizer: compile with -fsanitize=address
- [ ] Install strace (Linux) or dtraceunless (Mac)
- [ ] Setup godbolt.org account for assembly viewing

Tool Practice (Day 1)

- [ ] Create intentional segfault: `int *p = NULL; *p = 42;`
- [ ] Debug with gdb: set breakpoint, step, print variables
- [ ] Detect with valgrind: `valgrind ./program`
- [ ] Catch with ASan: compile with -fsanitize=address
- [ ] View syscalls: `strace ./program` (Linux) or `dtruss ./program` (Mac)

Resources

Primary (READ THESE)

- [ ] "The C Programming Language" (K&R) - Chapters 1-5, 6, 7
- [ ] "Computer Systems: A Programmer's Perspective" (CS:APP)
  - Chapter 3: Machine-Level Representation (x86-64 assembly)
  - Chapter 9: Virtual Memory (sections 9.1-9.9)
- [ ] "What Every Programmer Should Know About Memory" (Ulrich Drepper) - First 40 pages

Supplementary

- [ ] CS50 lectures on C and memory
- [ ] "Beej's Guide to C Programming"
- [ ] "Understanding glibc malloc" (online article)
- [ ] Source code: glibc malloc.c (simplified version)

Online Tools

- [ ] godbolt.org - Compiler Explorer (C to Assembly)
- [ ] cdecl.org - Decode complex C declarations
- [ ] onlinegdb.com - Quick debugging

Week-by-Week Breakdown

Week 1: Strings, Pointers, and Tools

Goal: Master pointer basics and tool usage. No more segfaults you can't debug.

Exercises

- [ ] **String functions from scratch** (NO <string.h>):
  - `my_strlen(const char *s)`
  - `my_strcpy(char *dst, const char *src)`
  - `my_strcat(char *dst, const char *src)`
  - `my_strcmp(const char *s1, const char *s2)`
- [ ] **Understand memory segments:**

```
char str1[] = "hello";      // Where is this?
char *str2 = "hello";       // Where is this?
char *str3 = malloc(6);     // Where is this?
// Write program that prints addresses and explains
```

- [ ] **Pointer arithmetic mastery:**
  - [ ] Array traversal using pointers (no [])
  - [ ] Reverse string in-place using two pointers
  - [ ] Implement `void *my_memcpy(void *dst, const void *src, size_t n)`
- [ ] **Tool practice:**
  - [ ] Deliberately create 5 different types of bugs
  - [ ] Debug each with gdb
  - [ ] Detect each with valgrind
  - [ ] Fix and verify

Reading

- [ ] K&R Chapters 1-5
- [ ] CS:APP 3.1-3.5 (Assembly basics)

Deliverable

- `week1/` folder with all exercises
- Each .c file must compile with `-Wall -Wextra -Werror`
- README explaining what you learned about memory layout

Week 2: Assembly and Memory Layout

Goal: Understand what the CPU actually executes. No more abstractions.

Exercises

- [ ] **C to Assembly exploration:**
  - [ ] Write simple functions, compile with `gcc -S`
  - [ ] Understand basic instructions: mov, add, call, ret
  - [ ] See how variables map to registers/stack
  - [ ] Use godbolt.org to see optimizations (-O0 vs -O2 vs -O3)
- [ ] **Stack deep dive:**
  - [ ] Write recursive function, view stack frames in gdb
  - [ ] Implement `void inspect_stack(void)` that prints stack addresses
  - [ ] Understand function prologue/epilogue
  - [ ] Deliberately overflow stack, see what happens
- [ ] **Heap exploration:**
  - [ ] malloc() 10 times, print addresses - see the pattern?
  - [ ] Use `strace` to see brk()/mmap() syscalls
  - [ ] Free in different orders, observe behavior
  - [ ] Double-free detection with ASan
- [ ] **Reading real code:**
  - [ ] Clone GNU coreutils
  - [ ] Read and annotate `cat.c` source
  - [ ] Read and annotate `ls.c` source (focus on memory management)

Reading

- [ ] CS:APP Chapter 3.6-3.10 (Assembly continued)
- [ ] CS:APP Chapter 9.1-9.5 (Virtual Memory basics)
- [ ] "Understanding glibc malloc" article

Deliverable

- `week2/` folder with:
  - Annotated assembly examples
  - Stack/heap exploration program
  - Notes on cat.c/ls.c code reading
  - Answers to malloc investigation (see questions.md)

Week 3: Data Structures and Memory Management

Goal: Build complex structures, manage memory correctly, no leaks.

Exercises

- [ ] **Dynamic array (vector):**

```
typedef struct {
    int *data;
    size_t size;
    size_t capacity;
} Vector;

// Implement:
Vector* vec_create(void);
void vec_push(Vector *v, int value);
int vec_get(Vector *v, size_t index);
void vec_destroy(Vector *v);
```

  - Must resize when full (capacity doubling)
  - Zero leaks with valgrind
  - Write tests for edge cases

- [ ] **Linked list (from scratch):**

```
typedef struct Node {
    int data;
    struct Node *next;
} Node;

// Implement:
Node* list_create(int data);
void list_append(Node **head, int data);
void list_delete(Node **head, int data);
void list_destroy(Node **head);
Node* list_reverse(Node *head);
```

  - Handle empty list, single node
  - Implement reverse IN-PLACE (no extra memory)
  - Zero leaks

- [ ] **Pointer to pointer mastery:**

  - Understand why `list_append(Node **head, ...)` needs **
  - Implement tree insertion using pointer-to-pointer
  - Draw memory diagrams

- [ ] **Memory leak debugging:**

    - [ ] Take a provided buggy program (create one with 10 leaks)
    - [ ] Find all leaks with valgrind
    - [ ] Fix systematically
    - [ ] Verify with valgrind --leak-check=full

Reading

- [ ] K&R Chapter 6 (Structures)
- [ ] CS:APP 9.9 (Dynamic Memory Allocation)

Deliverable

- `week3/` folder with:
    - Vector implementation + tests
    - Linked list implementation + tests
    - Leak debugging report
    - Memory diagrams for pointer-to-pointer

Week 4: Mini-Project - Memory Allocator

Goal: Implement your own malloc/free. Understand heap management from the inside.

Project: my_malloc / my_free

Specifications:

```
void* my_malloc(size_t size);
void my_free(void *ptr);
```

Requirements:

- [ ] Use `sbrk()` (simple) or `mmap()` (better) to get memory from OS
- [ ] Maintain a free list of available blocks
- [ ] Implement first-fit or best-fit allocation strategy
- [ ] Store metadata (block size, free/used flag) BEFORE each block
- [ ] Implement coalescing of adjacent free blocks
- [ ] Handle alignment (8-byte or 16-byte boundaries)
- [ ] Handle edge cases: size=0, free(NULL), double-free

Implementation phases:

1. **Day 1-2:** Basic version - just sbrk() and linear search
2. **Day 3-4:** Add free list and coalescing
3. **Day 5:** Add alignment and metadata
4. **Day 6-7:** Testing, debugging, benchmarking

Testing:

- [ ] Allocate → Free → Allocate (verify memory reuse)
- [ ] Allocate many small blocks, free in different orders
- [ ] Stress test: 10000 random malloc/free operations
- [ ] No leaks with valgrind
- [ ] Benchmark vs system malloc (don't worry if slower)

Resources:

- Tutorial: "A malloc Tutorial" (online)
- Read simplified dlmalloc source code
- CS:APP 9.9.12 (Explicit Free Lists)

Deliverable

- `week4/my_allocator/` folder with:
  - `my_malloc.c` / `my_malloc.h`
  - `test_allocator.c` with comprehensive tests
  - `benchmark.c` comparing to system malloc
  - `DESIGN.md` explaining your design choices
  - Valgrind report showing zero leaks

Key Concepts (Fill as you learn)

Pointers

What I've learned:

- [Date] ...
- [Date] ...

Stack vs Heap

What I've learned:

- [Date] ...

Memory Layout (Text, Data, BSS, Heap, Stack)

What I've learned:

- [Date] ...

Assembly Basics

What I've learned:

- [Date] ...

Memory Allocation Strategies

What I've learned:

- [Date] ...

Reality Checks

Week 1 Check (Date: _____)

- [ ] Can I debug a segfault in under 5 minutes?
- [ ] Do I understand why `char *s = "hello"; s[0] = 'H';` crashes?
- [ ] Can I explain pointer arithmetic to someone?

Week 2 Check (Date: _____)

- [ ] Can I read basic x86-64 assembly?
- [ ] Do I understand calling conventions?
- [ ] Can I explain what malloc() does at the syscall level?

Week 3 Check (Date: _____)

- [ ] Can I implement any data structure without memory leaks?
- [ ] Do I understand pointer-to-pointer usage?
- [ ] Can I draw memory layout diagrams?

Week 4 Check (Date: _____)

- [ ] Did I actually implement malloc, or just copy code?
- [ ] Can I explain heap fragmentation?
- [ ] Am I ready for Phase 1, or do I need more time?

Milestones

- [ ] Compiled first program with -Wall -Wextra -Werror with no warnings
- [ ] Debugged first segfault using gdb

- [ ] Found first memory leak with valgrind
- [ ] Implemented linked list with zero leaks
- [ ] Read first 100 lines of real C codebase
- [ ] Read first assembly output from compiled C
- [ ] Completed working memory allocator
- [ ] **Phase 0 complete - READY FOR ASSEMBLY & MEMORY SYSTEMS**

Success Criteria (ALL must be true)

Before moving to Phase 1, you MUST:

1. Implement working my_malloc/my_free with zero leaks
2. Be able to debug segfaults without help
3. Read basic x86-64 assembly from compiled C
4. Understand stack, heap, data, text segments
5. Have zero fear of pointers, including pointer-to-pointer

If any is false, extend Phase 0. No shortcuts.

Started: 2024-11-11 | Target Completion: 2024-12-09 | Actual: TBD