Memory Management in C

Computer Architecture Discussion

lujj2025@shanghaitech.edu.cn

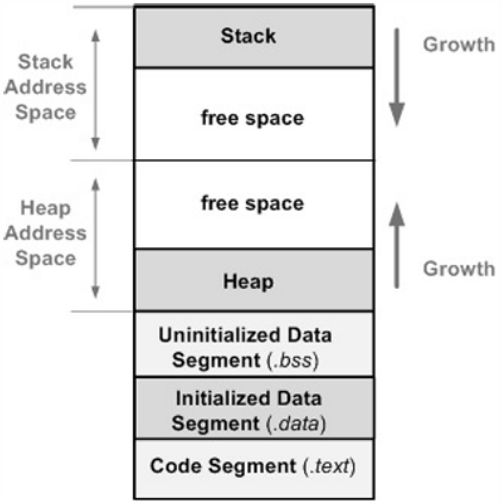
March 20, 2026

Roadmap

1. Memory Segments
   - layout
   - .bss vs .data Example
   - heap vs stack
2. Static Allocation
   - Global and Static Variables
   - Local Variables
3. Dynamic Memory Management
   - Dynamic Memory Allocation
   - Memory Release and Leaks
4. valgrind
5. Practice and Questions

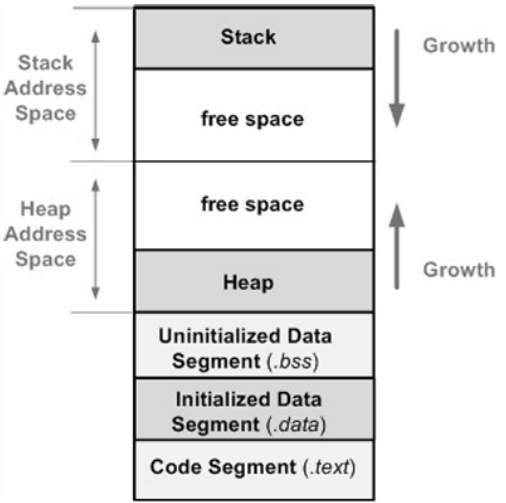
Recall: Memory Layout

- **Stack** growing downward
  - Function arguments
  - Return addresses
  - Local variables
- **Heap** growing upward
  - For dynamic demands (malloc)
- **.bss** for global/static initially being zero
- **.data** for global/static with initial values
- **.text** for machine code (instruction)



Why Bother?

- **Different lifespans of variables**
  - global/static: entire execution period
  - local / arguments / return address: within the function
  - dynamic: undeterminable lifetime
- **Smaller program size**
  - Many global/static variables start as zero
  - Wasteful if stored explicitly
- **Portable to non-von Neumann architectures**
  - Separate devices for code, variables, and constants



.bss vs .data Example

Program 1 (.bss example)

```
int array[30000];

int main() {
    return 0;
}
```

Program 2 (.data example)

```
int array[30000] = {1,2,3,4,5,6};

int main() {
    return 0;
}
```

- Observation: Program 2 executable is much larger than Program 1
- Reason: .bss segment Only reserves memory space; initial values are not stored in the program file $\rightarrow$ does not increase the executable size.
- When the program is loaded, the system initializes .bss to 0.
- .data segment Stores the variables' initial values $\rightarrow$ the initial values are written directly into the executable $\rightarrow$ increases the executable size.

Variable Location: objdump

Program 1:

```
objdump -t main | grep array
...
0000000000601060 g     O .bss
000000000001d4c0  array
```

Program 2:

```
objdump -t main | grep array
...
0000000000601040 g
O .data 000000000001d4c0  array
```

- Program 1: array in **.bss**
- Program 2: array in **.data**
- Confirms the executable size difference

Summary: .bss vs .data

- Uninitialized globals / static locals → **.bss segment**
- Initialized globals / static locals → **.data segment**
- Non-static locals → stack
- .bss does not increase executable size; memory initialized by OS
- .data increases executable size; memory initialized by program
- Linker allocates .bss memory right after .data

Example:

```
int a;
int b = 0;
```

Both variables are usually placed in **.bss**. Reason: initialization with `0` can be optimized by the compiler and stored in the **.bss** segment instead of **.data**.

Stack Memory Example

C Code: Stack Usage

```
#include <stdio.h>

void function_c(int param) {
    int local_var = param * 2;
    printf("C - local_var: %p\n", &local_var);
}

void function_b(int x) {
    int b_var = x + 10;
    printf("B - b_var: %p\n", &b_var);
    function_c(b_var);
}

int main() {
    int main_var = 100;
    printf("main - main_var: %p\n", &main_var);
    function_b(main_var);
    return 0;
}
```

Observation:

- Each function call creates a new stack frame
- Stack frames store local variables, function parameters and return address
- Stack frames typically grow downward in memory
- Memory is automatically released when function returns
- Example output addresses:
  - main: 0x7fff5fbff6ac
  - functionB: 0x7fff5fbff68c
  - functionC: 0x7fff5fbff66c

Heap Memory Example

C Code: Heap Usage (Simplified)

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int* ptr = malloc(sizeof(int));
// malloc
    *ptr = 100;

    int* arr = calloc(3, sizeof(int)); // calloc
    arr[0] = 200;

    ptr = realloc(ptr, 2*sizeof(int)); // realloc

    free(ptr);  // free memory
    free(arr);
    return 0;
}
```

Observation:

- Heap memory persists until explicitly freed
- malloc: allocate uninitialized memory
- calloc: allocate and initialize to 0
- realloc: resize memory, may return a new address
- free: release memory, prevent memory leaks

Stack vs Heap: Comparison

Feature	**Stack**	**Heap**
Memory Allocation	Automatic (by compiler)	Manual (by programmer)
Speed	Fast	Slower
Size / Capacity	Limited	Large
Memory Release	Automatic	Manual (free)
Typical Use	Local variables, function calls	Dynamic memory allocation

Out-of-range

- **Bad in global/static**
  - Secretly corrupt nearby variables
- **Worse in heap**
  - Secretly corrupt who-knows-where vars
  - May corrupt heap management
  - If allocation records are kept in heap
- **Worst in stack!**
  - Secretly corrupt nearby variables
  - arguments, callers' arguments
  - and even return addresses

```
yitro@RM-server ~/tmp/disc3 08:54
[558] $ cat global-overrun.c
#include <stdio.h>
int a[1] = {5}, b = 6;
int main() {
  a[1] = 0;
  printf("%d\n", b);
  return 0;
}
yitro@RM-server ~/tmp/disc3 08:54
[559] $ gcc global-overrun.c && ./a.out
0
```

```
yitro@RM-server ~/tmp/disc3 08:49
[553] $ cat stack-overrun.c
#include <stdio.h>
#include <stdlib.h>
void g() { puts("Surprise!"); exit(1); }
void f() { void(*a[1])(); a[3] = g; }
int main() { f(); return 0; }
yitro@RM-server ~/tmp/disc3 08:49
[554] $ gcc stack-overrun.c && ./a.out
Surprise!
```

Static Allocation

- Static allocation: memory is determined at compile time
- Memory is reserved in **data segment** or **BSS segment** for global/static variables
- Lifetime: entire duration of the program
- Advantages:
  - Fast access
  - No dynamic memory management needed
- Disadvantages:
  - Less flexible
  - Cannot resize at runtime

Global and Static Variables

Key Points:

- **Global variables:** declared outside any function, memory allocated at program start
- **Static variables:** declared with `static` keyword, retain value across function calls, in bss/data, not in stack

Example:

```
#include <stdio.h>

// Global variable
int globalVar = 10;

void function() {
    // Static variable
    static int staticVar = 20;
    printf("globalVar: %d, staticVar: %d\n", globalVar, staticVar);
}
int main() {
    function();  // prints: globalVar: 10, staticVar: 20
    function();  // prints: globalVar: 10, staticVar: 20 (retains value)
    return 0;
}
```

Local Variables

Key Points:

- Local variables are allocated on the stack when the function is called
- Memory is automatically released when the function returns
- Lifetime: only during the function execution
- Advantages:
  - Fast allocation and deallocation
  - No need for manual memory management
- Disadvantages:
  - Cannot retain value between function calls (unless declared static)
  - Limited by stack size

Example:

```
void function() {
    int localVar = 30; // Local variable
    printf("localVar: %d\n", localVar);
}
int main() {
    function();  // prints: localVar: 30
    function();  // prints: localVar: 30 (new allocation each call)
```

malloc

Purpose

- `malloc` allocates a block of memory on the **heap**
- Returns a pointer to the beginning of the allocated memory
- If allocation fails, it returns `NULL`

Function Prototype

```
void *malloc(size_t size);
```

Example

```
int *ptr = (int *)malloc(sizeof(int));
if (ptr != NULL) {
    *ptr = 10;
}
```

malloc Implementation

Algorithm

1. Align requested memory size
2. Search a suitable free block
3. If block is larger than needed, split it
4. Mark the block as allocated
5. If no free block exists, request memory from OS
6. Return pointer to the allocated block

Pseudo Code

```
function malloc(size):

    block = find_free_block(size)

    if block != NULL:
        mark block allocated
        return pointer(block)

    block = request_memory_from_OS(size)

    if block == NULL:
        return NULL

    return pointer(block)
```

calloc

Purpose

- `calloc` allocates memory for an array of elements
- The allocated memory is **initialized to zero**
- Returns a pointer to the allocated memory
- Returns `NULL` if allocation fails

Function Prototype

```
void *calloc(size_t num, size_t size);
```

- `num`: number of elements
- `size`: size of each element

Example

```
int *arr = (int *)calloc(5, sizeof(int));
if (arr != NULL) {
    arr[0] = 10;
}
```

calloc Implementation

Algorithm

1. Compute total size: num $\times$ size
2. Allocate memory using malloc
3. Initialize all bytes to zero
4. Return pointer to the memory block

Pseudo Code

```
function calloc(num, size):

    total_size = num * size

    ptr = malloc(total_size)

    if ptr == NULL:
        return NULL

    memset(ptr, 0, total_size)

    return ptr
```

realloc

Purpose

- `realloc` resizes a previously allocated memory block
- Can expand or shrink the memory
- Existing data is preserved (up to the minimum size)

Function Prototype

```
void *realloc(void *ptr, size_t new_size);
```

- `ptr`: pointer to existing memory block
- `new_size`: new size in bytes

Example

```
int *arr = (int *)malloc(2 * sizeof(int));

arr = (int *)realloc(arr, 4 * sizeof(int));
```

realloc Implementation

Algorithm

1. If pointer is NULL, call malloc
2. Check if current block fits new size
3. If not, allocate a new block
4. Copy old data to the new block
5. Free the old block
6. Return new pointer

Pseudo Code

```
function realloc(ptr, new_size):

    if ptr == NULL:
        return malloc(new_size)

    new_ptr = malloc(new_size)

    if new_ptr == NULL:
        return NULL

    copy_data(new_ptr, ptr)

    free(ptr)

    return new_ptr
```

Linked List Example

Create Node

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

struct Node* createNode(int value) {
    struct Node* newNode =
    (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
```

Insert Node into List

```
struct Node* insert(struct Node* head,
                    int value) {

    struct Node* newNode =
        createNode(value);

    newNode->next = head;
    return newNode;
    }
```

Linked List Example

Releasing Linked List Memory

- When nodes are no longer needed, they must be released using `free`
- Otherwise, the program may suffer from **memory leaks**

```
void freeList(struct Node* head) {

    struct Node* tmp;

    while (head) {
        tmp = head;
        head = head->next;
        free(tmp);
    }
}
```

Memory Fragmentation

- Frequent allocation and deallocation may create **memory fragmentation**
- Free memory becomes scattered into small blocks

Memory Release

Purpose

- Release dynamically allocated memory
- Return memory back to the system
- Allow the memory to be reused

Function Prototype

- `void free(void *ptr);`

Notes

- Memory must be allocated by `malloc`, `calloc`, or `realloc`
- `free(NULL)` does nothing
- Invalid pointer causes **undefined behavior**

How `free` Works:

- Check if `ptr` is NULL; if so, return immediately
- Find the metadata of the memory block (size, allocation info)
- Mark the block as free in the heap management structures
- Optionally coalesce adjacent free blocks to reduce fragmentation
- Return memory to the system if possible (via `sbrk`/`mmap`)
- The pointer itself is not modified

Example 1: Freeing Stack Memory

Problem

- Local variables are stored on the stack
- Stack memory is automatically managed
- Calling `free()` on stack memory is invalid

Possible Results

- Program crash
- Memory corruption
- Undefined behavior

```c
#include <stdio.h>
#include <stdlib.h>

int main() {

    int local_var = 10;

    free(&local_var);  // ERROR

    return 0;
}
```

Example 2: Freeing Static/Global Memory

Problem

- Global variables are stored in the data segment
- They are not dynamically allocated
- Using `free()` on them is invalid

Possible Results

- Program crash
- Data corruption
- Unpredictable program behavior

```
#include <stdio.h>
#include <stdlib.h>

int global_var = 20;

int main() {

    free(&global_var); // ERROR

    return 0;
}
```

Example 3: Double Free

Problem

- Memory is released more than once
- Pointer still holds the old address
- This leads to undefined behavior

Best Practice

- Set pointer to `NULL` after free

```
#include <stdio.h>
#include <stdlib.h>

int main() {

    int *p = malloc(10*sizeof(int))

    free(p);   // first free
    free(p);   // ERROR

    return 0;
}
```

Memory Leak

Definition

- Memory allocated dynamically but not released
- Consumes system memory over time
- May lead to crashes or degraded performance

Common Causes

- Forgetting to call `free()`
- Pointer overwrite without freeing, int *p = malloc(100); p = malloc(200);
- Memory allocation in recursion without proper release, void f(){int *p = malloc(100);f();}

How to Avoid: Always free memory when no longer needed

Example: Memory Leak

```
#include <stdio.h>
#include <stdlib.h>

void leaky_function() {
    int *p = malloc(10 * sizeof(int
    if (!p) return;

    for (int i = 0; i < 10; i++)
        p[i] = i * 10;

    // Memory not freed -> leak
    // free(p);
}
```

valgrind: a memcheck tool

Memcheck can detect:

- Illegal memory access
  - Overrunning / underrunning buffers
  - Use-after-free errors
- Use of uninitialized values
- Double free of heap blocks
- Fishy size parameters passed to allocation functions
- Memory leaks

Valgrind Example: Invalid Memory Access

Buggy Code:

```
int *f(void) {
    int a;
    return &a;
}

int main() {
    *f() = 1;
    return 0;
}
```

Valgrind Output:

```
==78779== Invalid write of size 4
==78779==    at 0x10918C: main (a.out)
==78779==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

Valgrind Example: Use of Uninitialized Value

Buggy Code:

```
int y = 0;

int main() {
    int x;
    if (x == 42)
        y += 8;
    return y;
}
```

Valgrind Output:

```
==11956== Conditional jump or move depends
==11956== on uninitialised value(s)
==11956==    at 0x109135: main (a.out)
```

Valgrind Example: Double Free

Buggy Code:

```
#include <stdlib.h>

int main() {
    void *buf = malloc(0);
    free(buf);
    free(buf);    // double free
}
```

Valgrind Output:

```
Invalid free()
Address 0x4a31040 is 0 bytes after a block of size 0 free'd
Block was alloc'd at malloc()
```

Valgrind Example: Fishy Allocation Size

Buggy Code:

```
#include <stdlib.h>

void main() {
    malloc(-1);   // invalid allocation size
}
```

Valgrind Output:

```
Argument 'size' of function malloc
has a fishy (possibly negative) value: -1
==28552==    at 0x4841878: malloc
==28552==    by 0x10915C: main (a.out)
```

Valgrind Example: Memory Leaks

Code:

```
#include <stdlib.h>
void f(void) {
    char *c = malloc(10);
    char **b = malloc(9);
    *b = c;   }
char *g(void) {
    char *d = malloc(11);
    return d + 1;   }
char **a;
int main() {
    f();
    a = malloc(8);
    *a = g();
    return 0;   }
```

Valgrind Output:

```
LEAK SUMMARY:
   definitely lost: 9 bytes in 1 blocks
   indirectly lost: 10 bytes in 1 blocks
   possibly lost: 11 bytes in 1 blocks
   still reachable: 8 bytes in 1 blocks
```

Practice 1

In C programming, which of the following statements about memory management is correct?

- A. Variables allocated on the stack stay throughout the program's execution.
- B. Memory allocated on the heap should be manually freed to prevent memory leaks.
- C. Heap memory is used for function call frames.
- D. The code segment contains uninitialized global variables.

Practice 1

In C programming, which of the following statements about memory management is correct?

- A. Variables allocated on the stack stay throughout the program's execution.
- B. Memory allocated on the heap should be manually freed to prevent memory leaks.
- C. Heap memory is used for function call frames.
- D. The code segment contains uninitialized global variables.

Solution: B

Practice 2

What will the following C code print?

```
#include <stdio.h>

int func(int* p) {
    p = p + 1;
}

void main() {
    int ar[2] = {4, 6};
    func(&ar[0]);
    printf("%d", *ar);
    return 0;
}
```

Options: A. 4 B. 5 C. 6 D. 7

Practice 2

What will the following C code print?

```
#include <stdio.h>

int func(int* p) {
    p = p + 1;
}

void main() {
    int ar[2] = {4, 6};
    func(&ar[0]);
    printf("%d", *ar);
    return 0;
}
```

Options: A. 4 B. 5 C. 6 D. 7

Solution: A

practice 3

Problem: Finish the `add_two` function, where we want to add 2 to an int variable passed by the user, and return its previous value.
Example: if x = 0, after calling `add_two`, the return value is 0 and x becomes 2. Constraints: Only one statement per line, comma (,) not allowed.

```
int add_two(         x) {

}
```

practice 3

Problem: Finish the `add_two` function, where we want to add 2 to an int variable passed by the user, and return its previous value.
Example: if x = 0, after calling `add_two`, the return value is 0 and x becomes 2. Constraints: Only one statement per line, comma (,) not allowed.

```
int add_two(         x) {

}
```

Solution (one possible answer):

```
int add_two(int *x) {
    *x += 2;
    return *x - 2;
}
```

practice 4

Problem:

The following macro calculates the product of a and b:

```
#define MUL(a, b) a * b
```

What is the value of:

```
16 / MUL(1 + 1, 3 - 1)
```

Is the result the same as:

```
16 / ((1 + 1) * (3 - 1))
```

If not, fix the macro to get the correct result.

practice 4

Problem:
The following macro calculates the product of a and b:

```
#define MUL(a, b) a * b
```

What is the value of:

```
16 / MUL(1 + 1, 3 - 1)
```

Is the result the same as:

```
16 / ((1 + 1) * (3 - 1))
```

If not, fix the macro to get the correct result.
Solution: - Original value: 18, not the same as 16 / ((1 + 1) * (3 - 1))
- Fixed macro:

```
#define MUL(a, b) ((a) * (b))
```

practice 5

Problem:
Write a macro that returns the minimum value between `a` and `b`. You should use the ternary conditional operator:

```
cond ? x : y
```

- Returns `x` when `cond` is true, `y` otherwise. - Define the macro as: ‘define MIN(a, b) ...‘

practice 5

Problem:
Write a macro that returns the minimum value between `a` and `b`. You should use the ternary conditional operator:

```
cond ? x : y
```

- Returns `x` when `cond` is true, `y` otherwise. - Define the macro as: ‘define MIN(a, b) ...‘
Solution:

```
#define MIN(a, b) ((a) < (b) ? (a) : (b))
// or
#define MIN(a, b) ((a) > (b) ? (b) : (a))
```

practice 6

Problem:
Write a C macro `PS` that prints the size of the required type.
Example:

`PS(long long)`

Should print:

```
Size of long long: 8
```

Hint: The format specifier for `size_t` is `%zu`.

practice 6

Problem:
Write a C macro `PS` that prints the size of the required type.
Example:

PS(**long long**)

Should print:

```
Size of long long: 8
```

Hint: The format specifier for `size_t` is `%zu`.
Solution:

```
#define PS(type) printf("Size of "#type": %zu", sizeof(type))
// or
#define PS(type) printf("Size of %s: %zu", #type, sizeof(type))
```