Memory Management

Learning Objectives

By the end of this lecture, you will be able to:

- **Explain** the memory hierarchy and program memory layout
- **Compare** stack vs heap allocation strategies
- **Identify** common memory management errors
- **Evaluate** algorithmic choices through memory optimization lens
- **Apply** memory management best practices in C/C++ and Java

Why Memory Management Matters

Real-world impact:

1. **Performance**: Efficient memory use = faster programs
2. **Stability**: Memory errors cause crashes and security vulnerabilities
3. **Scalability**: Poor memory management limits how big your system can grow
4. **Cost**: Memory is expensive at scale (servers, cloud computing)

In this course: Connects to computational theory (space complexity, Turing machines)

Part 1: Memory Hierarchy

Memory Hierarchy

Modern computers organize memory by speed and cost:

```
Registers (CPU)     ← Fastest, smallest, most expensive
    ↓
L1 Cache
    ↓
L2 Cache
    ↓
L3 Cache
    ↓
RAM (Main Memory)
    ↓
Disk Storage
    ↓
Network Storage     ← Slowest, largest, cheapest
```

Key Principle: Maximize use of faster memory, minimize slower memory access

Access Times Comparison

Level	Typical Size	Access Time	Relative Speed
Registers	Bytes	< 1 ns	1x
L1 Cache	KB	1-2 ns	~4x slower
L2 Cache	MB	3-10 ns	~10x slower
RAM	GB	50-100 ns	~100x slower
SSD	TB	50-150 µs	~100,000x slower
HDD	TB	1-10 ms	~10,000,000x slower

Takeaway: Memory location dramatically affects performance!

🎯 Active Learning: Quick Poll

Question: If we need to sum 1 million integers, which is faster?

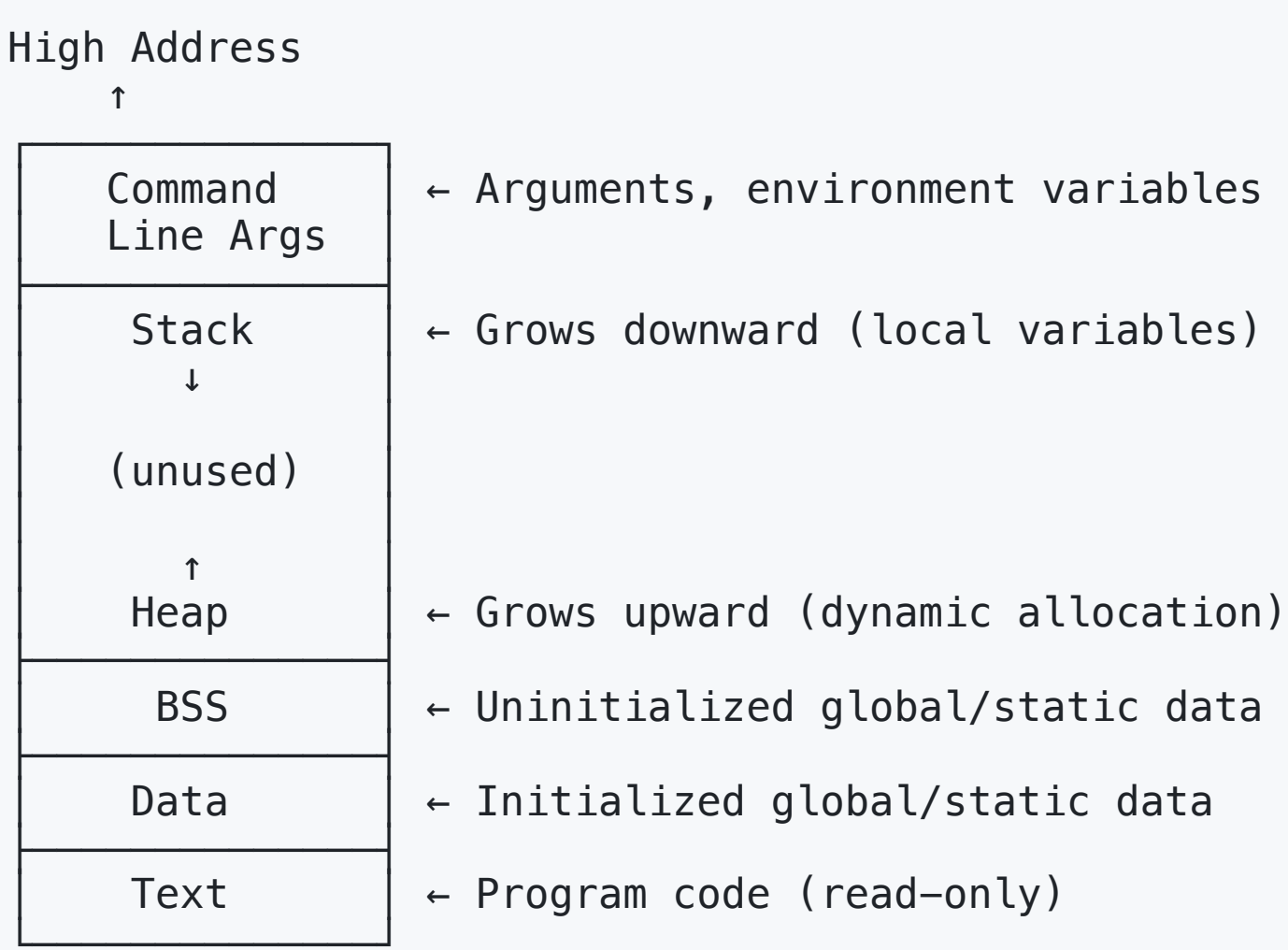
A) Array stored contiguously in memory (sequential access)

B) Linked list with nodes scattered in memory (random access)

Think about why one might be faster

Part 2: Program Memory Layout

Memory Layout of a Running Program



Memory Segments Explained

Segment	Purpose	Characteristics
Text	Executable code	Read-only, fixed size
Data	Initialized globals/statics	Fixed size, loaded at start
BSS	Uninitialized globals	Zeroed at program start
Heap	Dynamic allocation	Grows up, managed explicitly
Stack	Function calls, locals	Grows down, automatic management

Key Insight: Different memory regions serve different purposes with different management strategies

Stack Memory

Characteristics:

- Automatic allocation when entering function
- Automatic deallocation when exiting function
- Very fast (just pointer adjustment)
- Limited size (typically 1-8 MB)
- LIFO (Last In, First Out) structure

Example (C):

```c
void function() {
    int x = 10;           // Allocated on stack
    char buffer[100];     // Allocated on stack
    // Automatically freed when function returns
}
```

Heap Memory

Characteristics:

- Explicit allocation (malloc/new)
- Explicit deallocation (free/delete)
- Slower than stack
- Large size (limited by available RAM)
- Manual management or garbage collection
- Can become fragmented

Example (C):

```c
void function() {
    int* ptr = (int*)malloc(sizeof(int) * 100);  // Heap
    // Use the memory...
    free(ptr);  // MUST explicitly free!
}
```

Stack vs Heap Summary

Stack ✓

- **Fast** allocation/deallocation
- **Automatic** cleanup
- **No** fragmentation
- **Simple** management

Stack ✕

- **Limited** size
- **Fixed** lifetime (function scope)
- Can't return pointers to local variables

Heap ✓

- **Large** size available
- **Flexible** lifetime
- Can return allocated memory

Heap ✕

- **Slower** allocation
- **Manual** management needed
- **Fragmentation** possible
- Memory leak risks

🎯 Active Learning: Stack or Heap?

For each scenario, decide: **Stack** or **Heap**?

1. Small integer counter in a function
2. 10 MB image buffer
3. Function return value (int)
4. Data structure needed after function returns
5. Array of size known at compile time (100 elements)
6. Array of size determined at runtime

Part 3: Memory Management in C/C++

Manual Memory Management

In C/C++, **you** are responsible for memory:

Allocation:

```
int* ptr = (int*)malloc(sizeof(int) * 10);    // C style
int* ptr = new int[10];                        // C++ style
```

Deallocation:

```
free(ptr);          // C style
delete[] ptr;       // C++ style
```

Critical Rule: Every `malloc` needs a `free`, every `new` needs a `delete`

Common Memory Errors (1/2)

1. Memory Leak

```
void leak_example() {
    int* ptr = malloc(100 * sizeof(int));
    // Use ptr...
    // Forgot to free(ptr)!
}  // Memory is lost forever
```

2. Use After Free (Dangling Pointer)

```
int* ptr = malloc(sizeof(int));
*ptr = 42;
free(ptr);
*ptr = 10;   // ERROR: Undefined behavior!
```

Common Memory Errors (2/2)

3. Double Free

```
int* ptr = malloc(sizeof(int));
free(ptr);
free(ptr);   // ERROR: Crash or corruption!
```

4. Buffer Overflow

```
char* buf = malloc(10);
strcpy(buf, "This string is way too long");  // ERROR: Writes past end
```

5. Stack Overflow

```
void infinite_recursion() {
    infinite_recursion();  // ERROR: Stack exhausted
}
```

Tools to Find Memory Errors

Valgrind (Linux/Mac):

```
valgrind --leak-check=full ./myprogram
```

AddressSanitizer (GCC/Clang):

```
gcc -fsanitize=address myprogram.c
./myprogram
```

Benefits:

- Detects leaks, use-after-free, buffer overflows
- Provides detailed error reports with line numbers
- Essential for debugging memory issues

C++ Smart Pointers To Avoid Memory Errors

- `std::unique_ptr`
- Uses RAII pattern
  - When unique_ptr goes out of scope, memory is freed
- The code that has the unique_ptr is responsible for it
- Can't copy
- Can move (transfer ownership)

```cpp
#include <memory>

{
    // Creation
    std::unique_ptr<int> ptr1 = std::make_unique<int>(42);

    // Usage - like a raw pointer
    *ptr1 = 100;
    std::cout << *ptr1 << std::endl;  // 100

    // Cannot copy
    // std::unique_ptr<int> ptr2 = ptr1;  // ❌ Compile error!

    // Can move (transfer ownership)
    std::unique_ptr<int> ptr2 = std::move(ptr1);
    // Now ptr1 is nullptr, ptr2 owns the memory
}

// Now memory is deleted.
```

🎯 Active Learning: Spot the Errors

```
void process_data() {
    int* data = malloc(sizeof(int) * 100);

    for (int i = 0; i <= 100; i++) {  // Line 4
        data[i] = i * 2;
    }

    int* result = calculate(data);
    free(data);

    printf("Result: %d\n", result[0]);  // Line 11
    free(result);
}
```

Part 4: Memory Management in Java

Automatic Memory Management

Java uses **garbage collection** - no explicit `free`:

```
void method() {
    // Objects allocated on heap
    String s = new String("Hello");
    int[] arr = new int[1000];

    // No need to explicitly free
    // Garbage collector reclaims when unreachable
}
```

Tradeoff: Convenience vs control (and some runtime overhead)

Java Memory Regions

Heap (Generational):

1. **Young Generation**
    - Eden Space (new objects)
    - Survivor Spaces S0, S1 (survived minor GC)
2. **Old Generation** (long-lived objects)

Non-Heap:

3. **Metaspace** (class metadata)
4. **Stack** (method frames, local primitives)

Key Idea: Most objects die young, so optimize GC for that

Garbage Collection Process

How it works:

1. **Mark** - Starting from roots, mark all reachable objects
    - Roots: local variables, static variables, JNI references
2. **Sweep** - Reclaim memory from unreachable objects
3. **Compact** (optional) - Move objects together to reduce fragmentation

Types: Minor GC (young gen), Major/Full GC (old gen)

Java Memory Best Practices

Do:

- Reuse objects when possible (e.g., StringBuilder)
- Null out references when done (helps GC)
- Use primitives instead of wrappers when possible
- Close resources (try-with-resources)

Don't:

- Create unnecessary objects in loops
- Hold onto large objects longer than needed
- Ignore OutOfMemoryError
- Assume GC runs at specific times

🎯 Active Learning: Optimize This Code

```java
public String concatenate(List<String> strings) {
    String result = "";
    for (String s : strings) {
        result = result + s;  // What's the problem?
    }
    return result;
}
```

Question: If `strings` has 10,000 elements, how many String objects are created?

Discuss: How would you fix this?

Part 5: Memory Optimization

Optimization Strategy 1: Algorithm Choice

Different algorithms have different memory needs:

Algorithm	Time	Space	Notes
Bubble Sort	$O(n^2)$	$O(1)$	In-place
Merge Sort	$O(n \log n)$	$O(n)$	Needs temp array
Quick Sort	$O(n \log n)$	$O(\log n)$	Recursion stack
Hash Table	$O(1)$ avg	$O(n)$	Extra buckets

Tradeoff: Often must choose between time and space efficiency

Optimization Strategy 2: Memory Pooling

Problem: Many small allocations are slow

Solution: Pre-allocate a pool

```
// Inefficient - 1000 allocations
for (int i = 0; i < 1000; i++) {
    int* temp = malloc(sizeof(int));
    process(temp);
    free(temp);
}

// Efficient - 1 allocation
int* pool = malloc(1000 * sizeof(int));
for (int i = 0; i < 1000; i++) {
    process(&pool[i]);
}
free(pool);
```

Optimization Strategy 3: Cache-Friendly Code

Memory is loaded in cache lines (smallest unit of data xfer) (~64 bytes)

Cache-Unfriendly ✕

```
// Column-major access
for (int col = 0; col < N; col++) {
    for (int row = 0; row < N; row++) {
        sum += matrix[row][col];
    }
}
```

Jumps around memory

Cache-Friendly ✓

```
// Row-major access
for (int row = 0; row < N; row++) {
    for (int col = 0; col < N; col++) {
        sum += matrix[row][col];
    }
}
```

Sequential access

Impact: Can be 10x faster for large matrices!

Optimization Strategy 4: Data Structure Alignment

Problem: CPUs prefer aligned memory access

```
struct Bad {
    char c;       // 1 byte
    int i;        // 4 bytes - needs alignment
    char c2;      // 1 byte
    // Total: 12 bytes (with padding)
};

struct Good {
    int i;        // 4 bytes
    char c;       // 1 byte
    char c2;      // 1 byte
    // Total: 8 bytes (better packed)
};
```

Tip: Order struct fields largest to smallest

Measuring Memory Usage

C/C++:

```c
#include <sys/resource.h>
struct rusage usage;
getrusage(RUSAGE_SELF, &usage);
printf("Memory: %ld KB\n", usage.ru_maxrss);
```

Java:

```java
Runtime runtime = Runtime.getRuntime();
long memory = runtime.totalMemory() - runtime.freeMemory();
System.out.println("Memory: " + memory + " bytes");
```

Rule: Always measure before optimizing!

🎯 Active Learning: Optimization Challenge

You need to store 1 million user records with:

- User ID (int)
- Username (string, avg 20 chars)
- Email (string, avg 30 chars)

Calculate:

1. Minimum memory needed
2. Estimate actual memory with overhead
3. What's the best data structure?
4. How would you optimize for lookups?

Part 6: Connection to Theory

Memory and Computational Theory

1. Stack-Based Languages (CFLs)

- Context-free languages recognized by pushdown automata (PDA)
- PDA uses a stack - same as function call stack!
- This is why parsers for programming languages use stacks

2. Turing Completeness

- System is Turing complete if it has "unbounded" memory
- Real computers have finite memory, but we model them as Turing machines
- Any Turing-complete system can theoretically solve any computable problem

Space Complexity

Theory perspective:

- Algorithms analyzed by **space complexity** (memory usage)
- Space complexity classes:
  - **PSPACE**: Polynomial space
  - **LOGSPACE**: Logarithmic space
  - **NSPACE**: Nondeterministic space

Connection to practice:

- Big-O notation for space: $O(1)$, $O(n)$, $O(n^2)$, etc.
- Informs data structure and algorithm choices
- Example: In-place algorithms use $O(1)$ extra space

Summary and Best Practices

Key Takeaways

Core Concepts:

1. Memory hierarchy: registers → cache → RAM → disk
2. Program memory: heap vs stack
    i. Stack: Fast, automatic, limited
    ii. Heap: Flexible, manual, large
3. C/C++: Manual management
4. Java: Garbage collection

Key Takeaways

Optimization:

1. Choose algorithms considering space-time tradeoffs
2. Use memory pools, cache-friendly access patterns
3. Measure before optimizing
4. Connection to theory: space complexity, Turing completeness

Best Practices Checklist

- ✓ **Prefer stack over heap** when possible
- ✓ **Check allocation success** before using memory
- ✓ **Free memory** in reverse order of allocation
- ✓ **Initialize and null pointers** appropriately
- ✓ **Use smart pointers** if available
- ✓ **Use profiling tools** (Valgrind, AddressSanitizer)
- ✓ **Profile before optimizing** - measure actual usage
- ✓ **Consider tradeoffs** between time and space
- ✓ **Close resources** properly (RAII in C++)

Common Pitfalls to Avoid

❌ Memory leaks (forgetting to free)
❌ Use after free (dangling pointers)
❌ Double free (freeing twice)
❌ Buffer overflows (writing past bounds)
❌ Stack overflows (deep recursion)
❌ Creating unnecessary objects
❌ Premature optimization
❌ Ignoring memory profiler warnings