



Introduction, Big-O and Pointers

CEN207 Data Structures — Week 1

Asst. Prof. Dr. Uğur CORUH · Fall 2026–2027

Today's plan (3 hours)

Hour	Topic
1	Course plan · what a data structure is · linear vs non-linear
2	Big-O: search, growth, loops, space Anim 1–5
3	Pointers, stack/heap, TLV/PER Anim 6–13 · C workshop Anim 14

Learning outcomes: LO.1 (linear/non-linear structures) · LO.2 (time and space complexity) · LO.7 (choose the right structure)

This week's concepts — where

Concept	Where
Data structure, linear vs non-linear	Sections 1–2
Big-O, growth, best/worst/average, space	Section 3
Pointers, structs	Section 4
Stack, heap, TLV/PER, C workshop	Sections 5–7

How the code examples work

- Every idea has a complete **C** and **Java** program
- C: `gcc -std=c11 -Wall -Wextra -o /tmp/x file.c && /tmp/x`
- Java: `javac -d /tmp/j File.java && java -cp /tmp/j File`
- Sources: `code/week-01/c/` and `code/week-01/java/`
- Each program's expected output is in the week notes

Course logistics: how you are graded

- **One project**, built all semester: C first, then Java
- **Two checkpoints**: a midterm check in C, a final in Java
- **Two quizzes** — no separate homework
- Weekly notes like this one, English and Turkish
- Full grading breakdown: the course **syllabus**

Course logistics: teams and resources

- Team size: **up to 3** students per project team
- **No team changes after week 3**
- Project guide: `docs/project-guide/`
- Prerequisites: `docs/prerequisites/` (C toolchain, background)
- Older syllabus PDF conflicts with class? **Ask, don't guess**

The map of this week — and of the course

This week builds	Every later week adds
How to measure cost (Big-O)	One more way to arrange data
A picture of memory (pointers)	Judged with the same Big-O tool

1. What Is a Data Structure?

A question to start

Your contacts app finds "Alice" among ten thousand names almost instantly. Unsorted, the very same phone would scan every name.

A short history

- **1960s** — "data structure" enters common use in CS
- **1968** — Knuth's *TAOCP* Vol. 1 studies lists, stacks, trees
- **1974** — Liskov & Zilles: the **abstract data type** (ADT)
- ADT = *what* a structure does, not *how* it is built

Intuition: furniture for data

- A filing cabinet, a bookshelf, a stack of trays
- Each holds the same *kind* of content
- Each is good at **different things**
- No single "best" arrangement — only trade-offs

A data structure, precisely

A way of organizing data so that specific operations — access, search, insert, delete — can be done at a known, analyzable **cost**.

Why there isn't just one

Operation	Why it isn't free
Access	Some layouts compute a position; others walk from the start
Search	Unsorted: check one by one. Sorted: can be halved
Insert / Delete	Some shift everything after; others relink two pointers

Mini-quiz

Why is there no single "best" data structure that beats every other one at everything?

Answer

Every layout makes a **trade-off**: what makes access fast (e.g. a sorted array) tends to make insertion slow, and vice versa. No exceptions.

2. Linear vs Non-linear: the Map of the Course

A question to start

Your last five songs played form a single line.
A company's org chart branches. Is that just
about drawing, or a real structural fact?

Definitions

- **Linear:** every element has exactly one "next"
- You can always ask "what comes next?" — one answer
- **Non-linear:** an element can have several "next"s
- Traversing means *choosing* which branch to follow

The course, sorted: linear structures

Structure	When	Defining trade-off
Array	Week 2	$O(1)$ access; $O(n)$ insert/delete in the middle
Linked list	Week 2	$O(n)$ access; $O(1)$ insert/delete once there
Stack (LIFO)	Week 3	Touch one end only; every op $O(1)$
Queue (FIFO)	Week 3	Add one end, remove the other; every op $O(1)$

The course, sorted: non-linear and by-key

Structure	When	Defining trade-off
Binary tree, heap	Week 4	$O(\log n)$ when balanced, up to 2 children
Graph	Weeks 5–6	Any number of connections; networks, maps
Hash table (by key)	Week 7	$O(1)$ average, by key, not position
Files (on disk)	Weeks 13–15	Same trade-offs, paid in disk I/O

Common mistakes

- "Non-linear" does **not** mean "disorganized"
- "Sorted" is not the same as "linear" — different axes
- A hash table's *slots* are linear; access by key is new

Mini-quiz

Is a stack of plates linear or non-linear? Why?

Answer

Linear. Every plate (except the top) has exactly one plate above it, and (except the bottom) exactly one below it — one "next".

Mini-quiz

Is a family tree linear or non-linear?

Answer

Non-linear. A parent can have several children, so there is more than one "next" from a given person.

3. Performance Analysis: Counting Steps, Not Seconds

A question to start

You write a function that searches an array.
It works. Is it *fast*? A stopwatch tells you
about your laptop today — not about the algorithm.

A short history

- **1894** — Bachmann introduces capital-**O** notation
- **Early 1900s** — Landau popularizes it (a "Landau symbol")
- **1960s** — computer science adopts it for algorithm cost
- **1976** — Knuth's article standardizes $O/\Omega/\Theta$ for CS

Intuition: count the steps

Count how many times the algorithm does its basic unit of work — one comparison, one array access — as a function of input size n .

Linear search: try every box

Starts at the front, checks one element after another until it finds the target or runs out.
Needs **no particular order** in the data.

Linear search, step by step

target = 27

comparisons = 6



Result: `arr[5] = 27`, found after 6 comparisons. In the worst case (last element, or not present at all) linear search makes $n = 11$ comparisons: $O(n)$.

Edge case — target not in the array

target = 27

comparisons = 6



Result: `arr[5] = 27`, found after 6 comparisons. In the worst case (last element, or not present at all) linear search makes $n = 11$ comparisons: $O(n)$.

Code — linear_search (the loop)

```
for (int i = 0; i < n; i++) {  
    (*comparisons)++;  
    if (arr[i] == target)  
        return i;  
}
```

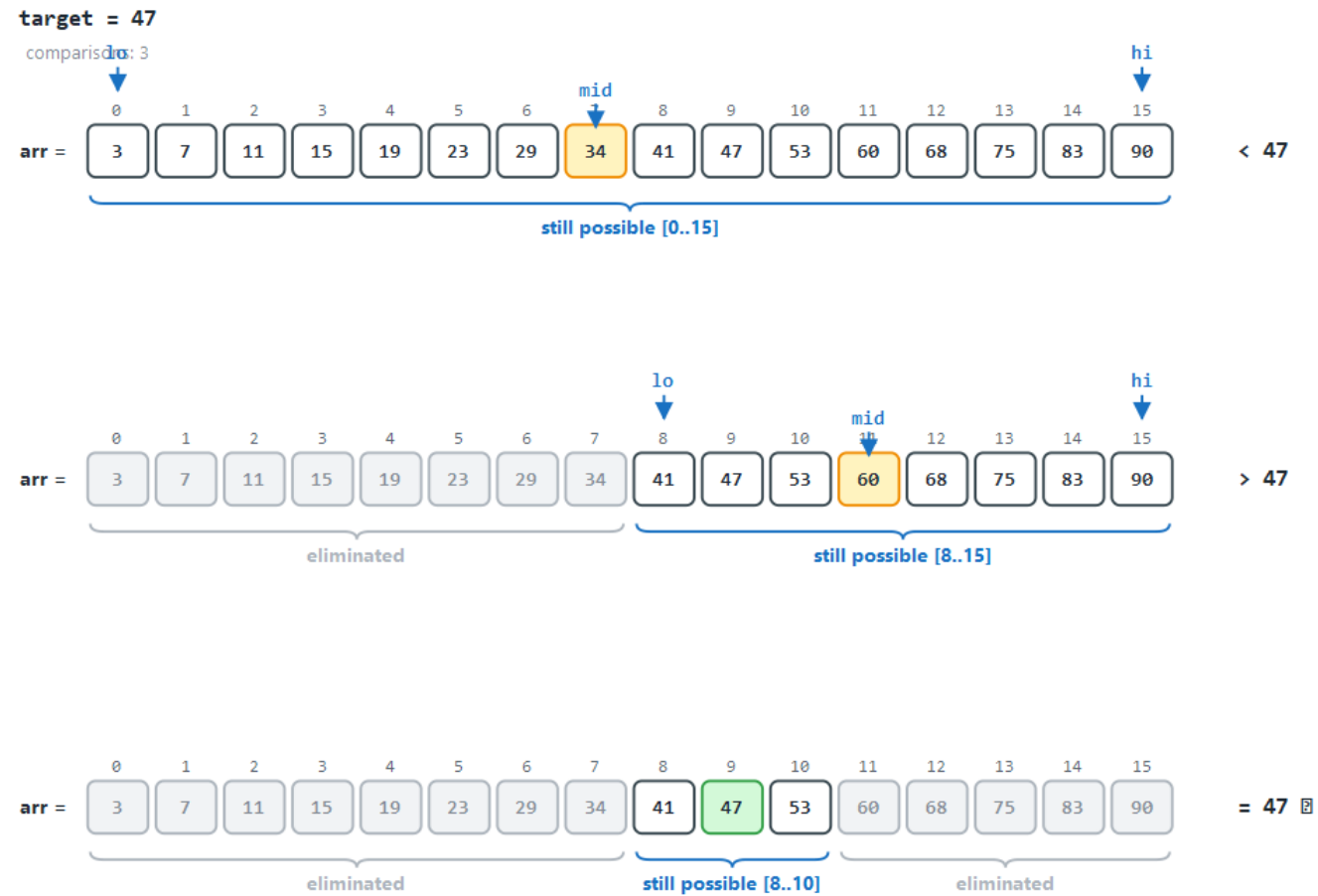
Linear search: the result

- Worst case (last element, or absent): **n** comparisons
- Best case (first element): **1** comparison
- Cost grows **directly with n** — this is **$O(n)$**

Binary search: eliminate half every time

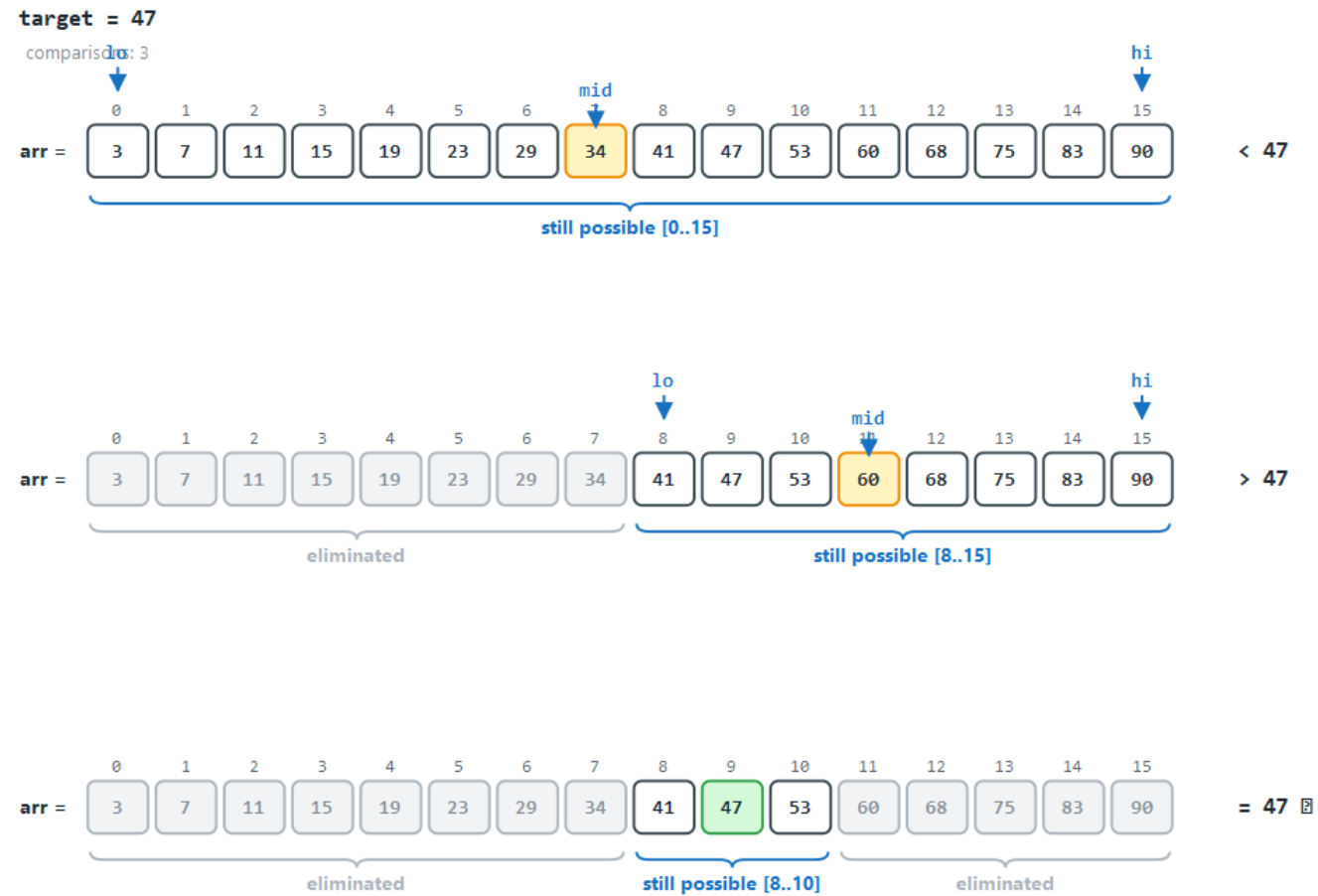
If the array is **sorted**: check the middle.
Too small? Discard the left half. Too big?
Discard the right half. Repeat.

Binary search, step by step



Result: arr[9] = 47, found in just 3 comparisons. Each step halves the search space: $O(\log n)$.

Edge case — not found: lo crosses hi



Result: arr[9] = 47, found in just 3 comparisons. Each step halves the search space: $O(\log n)$.

Code — binary_search (core step)

```
int mid = lo + (hi - lo) / 2;
(*comparisons)++;
if (arr[mid] == target)
    return mid;
if (arr[mid] < target)
    lo = mid + 1;
else
    hi = mid - 1;
```

Binary search: the result

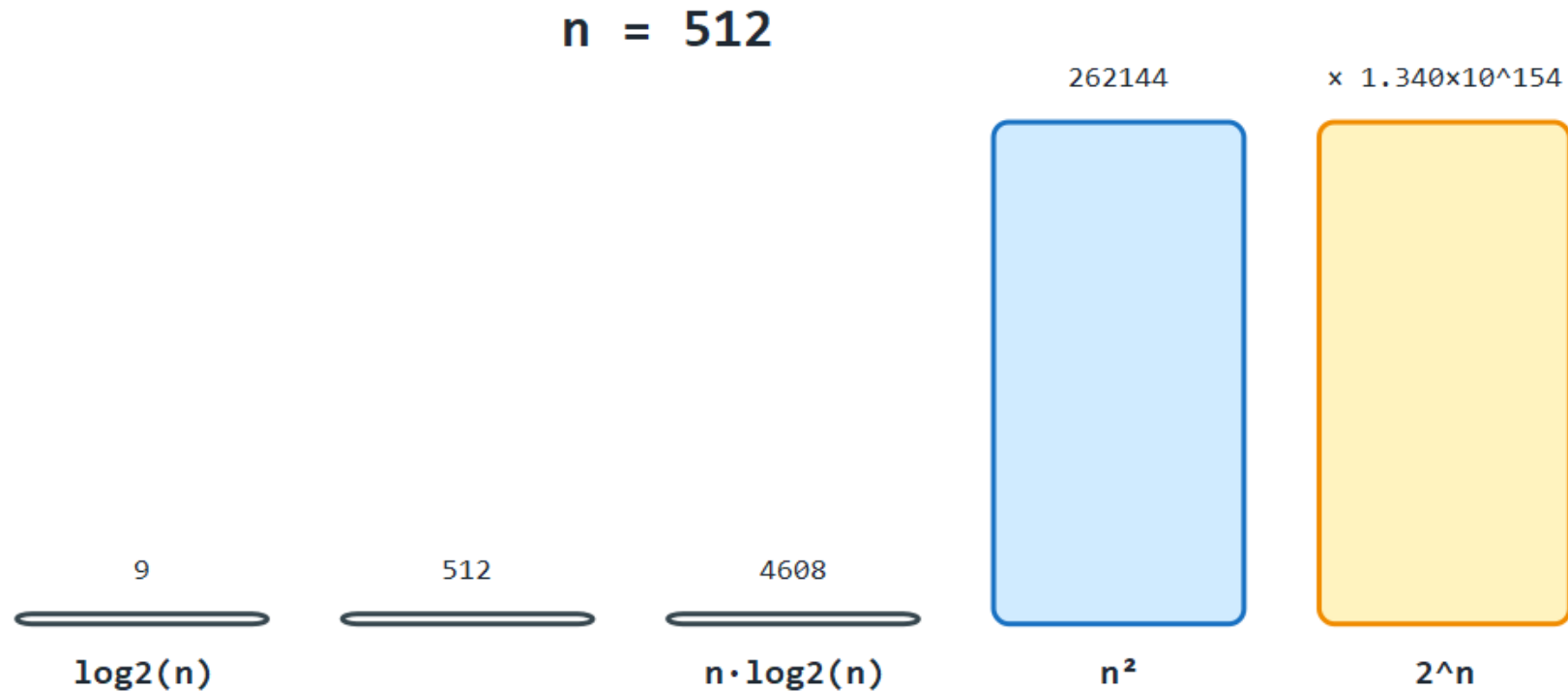
- The same 42 took 3 comparisons, not 9
- Every comparison throws away half of what is left
- Number of halvings before reaching 1: $\log_2 n$ — $O(\log n)$

The growth race: shape beats speed

Watch three functions race as n doubles:

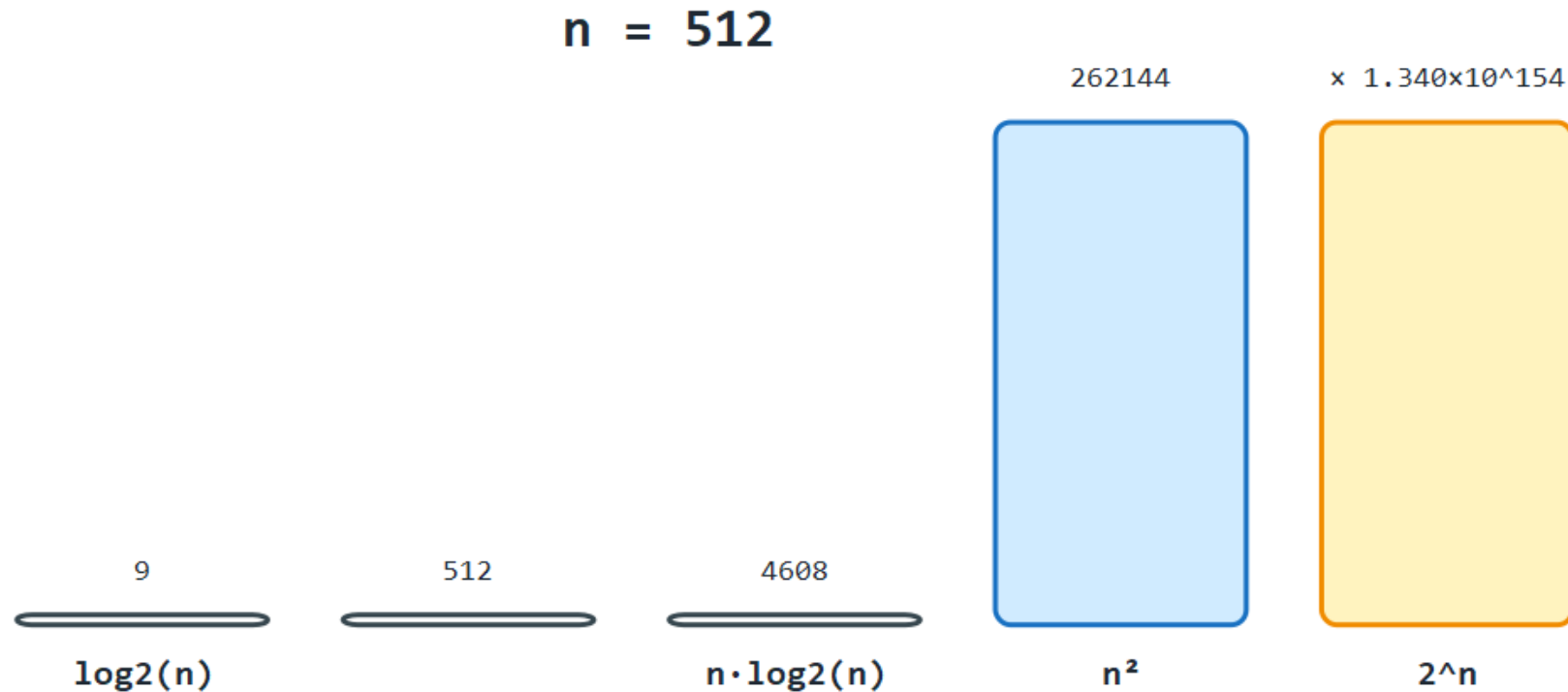
plain n , $n \cdot \log_2 n$ (best sorts), and n^2
(simple sorts, and nested loops in general).

The growth race, step by step



In order, $\log_2 n < n < n \cdot \log_2 n < n^2 < 2^n$ — each one eventually and permanently overtakes the previous. At the last n (512), $n^2 = 262144$ while $2^n = 1.340 \times 10^{154}$: exponential complexity grows brutally fast, even for modest inputs.

Edge case — 2^n overflows almost at once



In order, $\log_2 n < n < n \cdot \log_2 n < n^2 < 2^n$ — each one eventually and permanently overtakes the previous. At the last n (512), $n^2 = 262144$ while $2^n = 1.340 \times 10^{154}$: exponential complexity grows brutally fast, even for modest inputs.

Growth, at realistic sizes

n	$n \cdot \log_2 n$	n^2
100	664	10,000
10,000	132,877	100,000,000
100,000	1,660,964	10,000,000,000

Code — the growth table

```
for (int i = 0; i < count; i++) {  
    long n = ns[i];  
    double nlogn = (double) n * log2((double) n);  
    double nsq = (double) n * (double) n;  
}
```

Big-O, precisely

$f(n)$ is $O(g(n))$ if, from some point on, f never grows faster than a constant multiple of g . In practice: **drop constants, keep the dominant term.**

Reading off the dominant term

Steps counted	Dominant term	Big-O
$3n + 7$	n	$O(n)$
$n^2 + 2n + 1$	n^2	$O(n^2)$
$2 \cdot \log_2 n + 5$	$\log n$	$O(\log n)$

The order, fastest to slowest

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$$

You already met the first three today —

$O(n \log n)$ and $O(2^n)$ arrive later this semester.

Building $T(n)$ by hand: a nested loop

A loop inside another loop: for every one of the n outer iterations, the inner loop runs all n of its own — the classic n^2 source.

Counting a nested loop, step by step

n = 25 (square (j < n))

operations: 9

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

$$T(3) = 9$$

$$T(4) = 16$$

$$T(5) = 25$$

$$T(6) = 36$$

$$T(8) = 64$$

$$T(10) = 100$$

$$T(12) = 144$$

$$T(16) = 256$$

$$T(20) = 400$$

$$T(25) = 625$$

Dropping the constants and the lower-order terms leaves the dominant term of n^2 : complexity $O(n^2)$. This is exactly the rule from Section 3.7, arrived at by counting.

Edge case — a different loop shape

$n = 25$ (square ($j < n$))

operations: 9

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

$$T(3) = 9$$

$$T(4) = 16$$

$$T(5) = 25$$

$$T(6) = 36$$

$$T(8) = 64$$

$$T(10) = 100$$

$$T(12) = 144$$

$$T(16) = 256$$

$$T(20) = 400$$

$$T(25) = 625$$

Dropping the constants and the lower-order terms leaves the dominant term of n^2 : complexity $O(n^2)$. This is exactly the rule from Section 3.7, arrived at by counting.

Code — the nested loop

```
long count = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        count++;
        (*operations)++;
    }
}
```

Best, worst, and average case

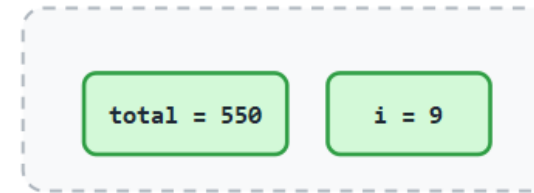
- **Best:** target is first checked — 1 comparison
- **Worst:** target is last, or absent — n comparisons
- **Average:** $(1 + 2 + \dots + n) / n = (n + 1) / 2$
- Unqualified "Big-O of an algorithm" usually means **worst**

Space complexity: the same idea, for memory

Big-O counts **time** by counting steps; space complexity counts **memory** — extra storage beyond the input itself.

Recursive vs iterative sum, step by step

sum_iterative: ONE frame

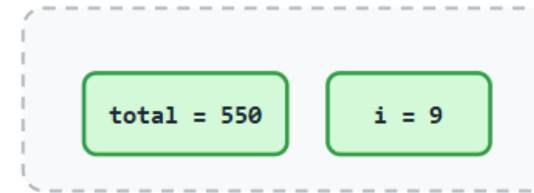


`main()`

Result: both functions return the same 550. But `sum_recursive` used $O(n)$ stack memory that grows with n ; `sum_iterative` always used exactly $O(1)$ — one frame, no matter what n is.

Edge case — deep recursion, 22 frames

sum_iterative: ONE frame



`main()`

Result: both functions return the same 550. But `sum_recursive` used $O(n)$ stack memory that grows with n ; `sum_iterative` always used exactly $O(1)$ — one frame, no matter what n is.

Code — recursive vs iterative sum

```
int sum_recursive(const int arr[], int n) {
    if (n == 0)
        return 0;
    return arr[n - 1] + sum_recursive(arr, n - 1);
}

int sum_iterative(const int arr[], int n) {
    int total = 0;
    for (int i = 0; i < n; i++)
        total += arr[i];
    return total;
}
```

Common mistakes

- Reading Big-O off the *code*, not the *steps*
- Forgetting binary search needs **sorted** input
- Confusing $O(1)$ with "instant" — it just means "not growing"
- Quoting only worst case when average matters more

Mini-quiz

An algorithm does exactly $5n + 20$ basic operations. What is its Big-O?

Answer

$O(n)$. Drop the constant factor (5) and the additive constant (20); only the fastest-growing term, n , survives.

Mini-quiz

Why can't binary search be used directly on a linked list the way it is on an array?

Answer

Binary search needs **$O(1)$ access** to the middle by index. A linked list must be walked node by node — already $O(n)$, erasing the gain.

Mini-quiz

Without qualification, which case does "Big-O of an algorithm" usually mean?

Answer

The **worst case**: the guaranteed upper bound, useful precisely because it holds no matter what input you are given.

4. Pointers and Objects

A question to start

Write `swap(a, b)` to exchange two variables in the caller. In many languages this is impossible — the function only sees *copies*.

A short history

- **1966/1969** — BCPL, then B: address-of, dereference
- **1972** — Dennis Ritchie's C makes pointers central
- **1995** — Java removes raw pointers, keeps **references**
- References: shared data, no arithmetic, no bad addresses

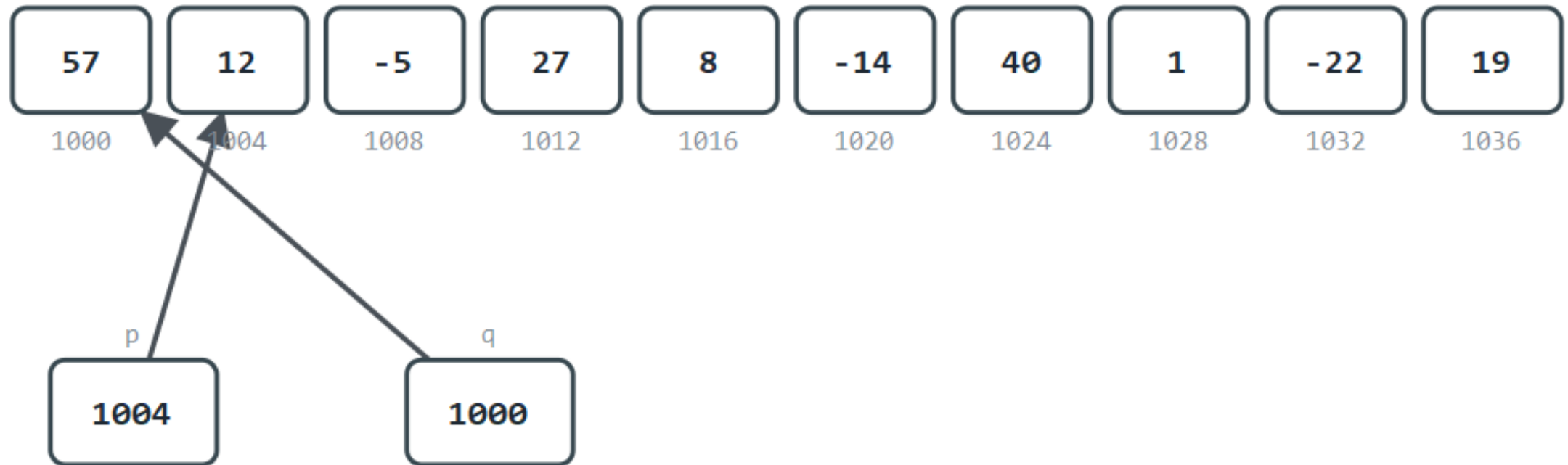
Intuition: memory as numbered mailboxes

- A variable is a mailbox: address + contents
- `&x` asks "what is x's mailbox number?"
- A **pointer** holds another mailbox's number
- Dereferencing (`*p`): go read that mailbox

The pointer operations

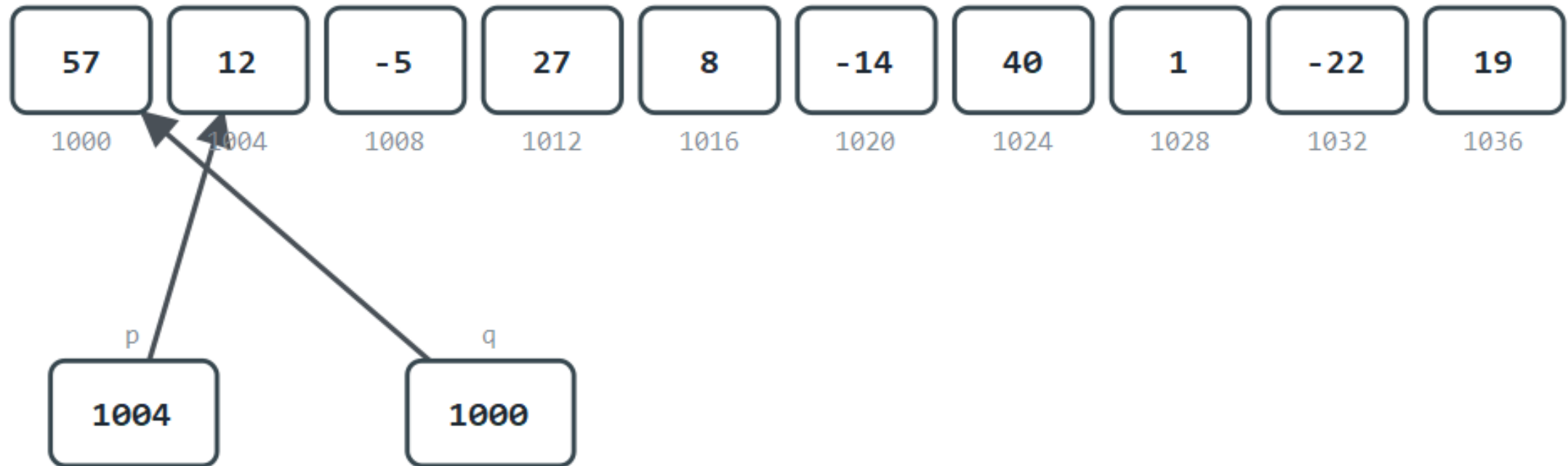
Operation	Syntax	Complexity
Address-of	<code>&x</code>	O(1)
Declare	<code>int *p;</code>	O(1)
Assign an address	<code>p = &x;</code>	O(1)
Dereference (read/write)	<code>*p / *p = v;</code>	O(1)

A variable, its address, a pointer



Done: final values 57, 12, -5, 27, 8, -14, 40, 1, -22, 19. Pointers: $p = 1004$, $q = 1000$.

Edge case — a NULL-pointer dereference



Done: final values 57, 12, -5, 27, 8, -14, 40, 1, -22, 19. Pointers: $p = 1004$, $q = 1000$.

Code — pointers: address, alias, NULL

```
int *p = NULL, *q = NULL;  
p = &v[i];  
*p = val;  
q = p;  
*q += k;  
if (r != NULL)  
    *r = val;
```

Java: reference vs value, side by side

```
int[] box = {3};  
int[] alias = box;  
alias[0] = 5;  
int y = x;  
y = 99;
```

Pointer arithmetic: $p + k$, not k bytes

$p + k$ never moves k bytes — it moves $k \times \text{sizeof}(*p)$ bytes. `int` is 4 bytes, so $p + 1$ skips 4 bytes, not 1.

Pointer arithmetic, step by step

`type = int, sizeof(int) = 4`

`p + 9 = 1000 + 9×4 = 1036, *(p+9) = 100`



Summary: `p + k` always moves $k \times \text{sizeof(int)} = k \times 4$ bytes forward, never k bytes. An out-of-range k is undefined behavior (UB). Java has no pointer arithmetic — only `a[k]`, and an out-of-range k throws a clear exception.

Edge case — out-of-range offsets

`type = int, sizeof(int) = 4`

`p + 9 = 1000 + 9×4 = 1036, *(p+9) = 100`



Summary: `p + k` always moves `k × sizeof(int) = k×4` bytes forward, never `k` bytes. An out-of-range `k` is undefined behavior (UB). Java has no pointer arithmetic — only `a[k]`, and an out-of-range `k` throws a clear exception.

Code — pointer arithmetic

```
int a[N];
int *p = a;
if (k < 0 || k >= N) {
    /* out of range: undefined behavior (UB) */
} else {
    void *addr = (void *) (p + k);
    int v = *(p + k);
}
```

Pointer to a struct, and ->

A `struct` groups fields into one value.

`(*p).field` is so common C gives it a shorthand: `p->field`.

Struct pointer, walking an array



Done: 10 records walked, 2 grades updated. Final records: 101:62, 102:78, 103:95, 104:91, 105:40, 106:83, 107:67, 108:40, 109:88, 110:59.

Edge case — one student only

101:62	102:78	103:95	104:91	105:40	106:83	107:67	108:40	109:88	110:59
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Done: 10 records walked, 2 grades updated. Final records: 101:62, 102:78, 103:95, 104:91, 105:40, 106:83, 107:67, 108:40, 109:88, 110:59.

Code — walking an array of structs

```
Student *p = students;
for (; p != students + N; p++) {
    int id = (*p).id;
    int grade = p->grade;
    p->grade = new_grade;
}
```

Common mistakes

- Using `int *p;` before it points anywhere (wild pointer)
- `int *p, x;` — only `p` is a pointer, not `x`
- Confusing `*` in a **declaration** vs an **expression**
- Java's exception is not the same as C's undefined behavior

Mini-quiz

Given `int *p = &x;`, what is the difference between `p` and `*p`?

Answer

`p` is the **address** stored in the pointer (x's address). `*p` is the **value** at that address (x's value).

Mini-quiz

Why does Java need no `->` operator at all?

Answer

Every object/array variable in Java is **already a reference** — `.` always means "follow it, then access the field".

5. Memory Layout: Stack, Heap, and Who Cleans Up

A question to start

A function declares a local array and returns.
What happens to it? Now compare: a function
`malloc` s a block and returns without freeing.

A short history

- **Stack** — one frame per call, back to Algol 60
- **malloc / free** — earliest Unix C libraries, early 1970s
- **Garbage collection** — McCarthy, Lisp, **1959**
- Java (1995) made GC mainstream for everyday programming

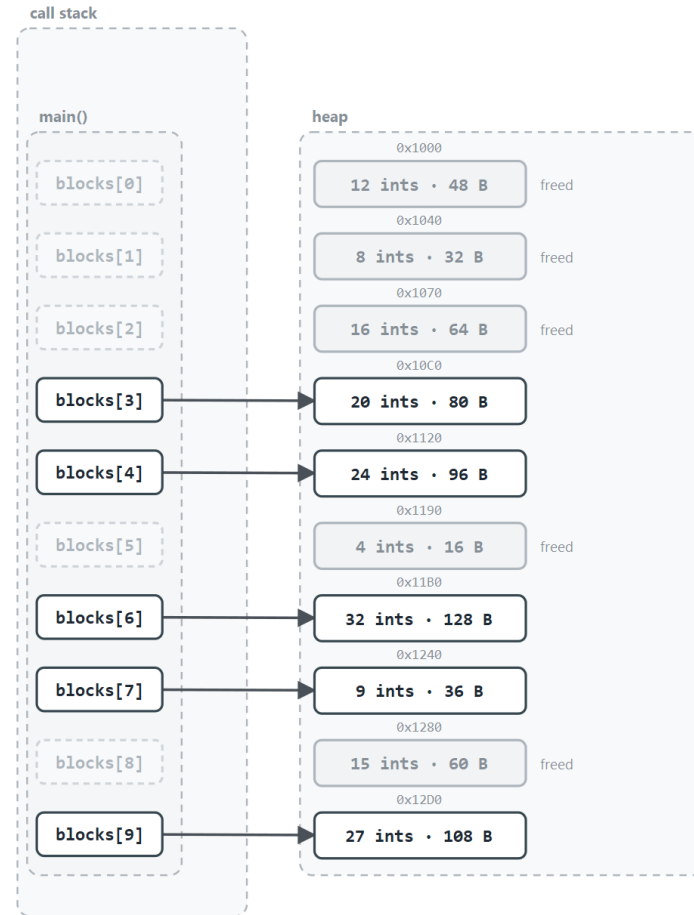
Intuition: a neat pile vs a warehouse

- **Stack:** every call pushes a frame; return pops it
- Always LIFO, always automatic, always fast
- **Heap:** you ask for a block; it stays until *given back*
- Nobody does that "giving back" for you in C

Stack vs heap, side by side

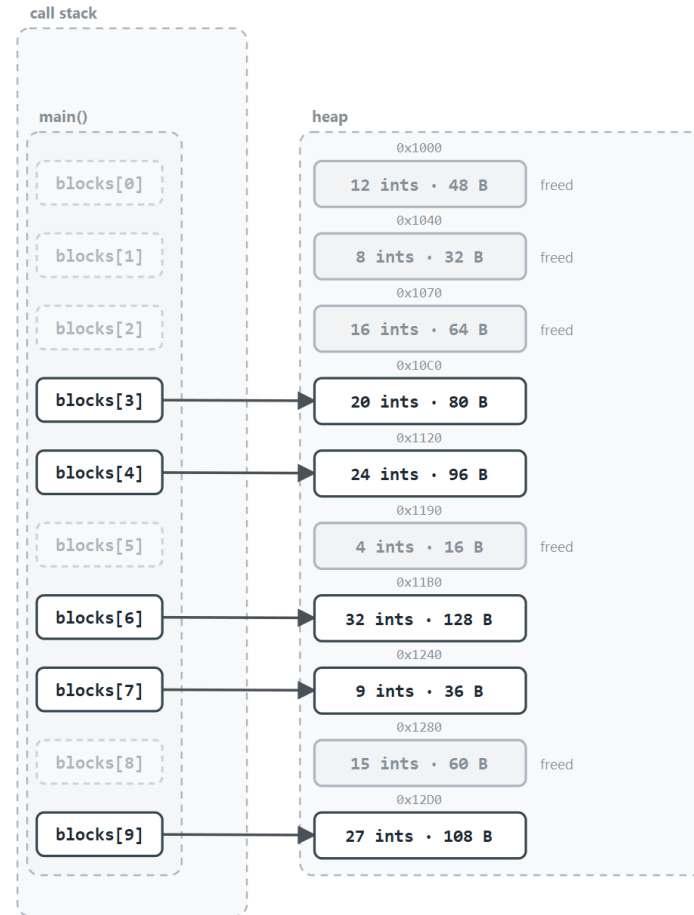
	Stack	Heap
Allocates	Compiler, automatically	You: <code>malloc</code> (C) / <code>new</code> (Java)
Freed	Automatically, on return	C: <code>free()</code> . Java: the GC
Speed	Extremely fast	Slower: finds a free block

How a stack frame connects to a heap block



Done: at most 2 frames at once (main plus one call), 10 blocks allocated. No leak: every block had a blocks[] owner.

Edge case — a dangling pointer, flagged



Done: at most 2 frames at once (main plus one call), 10 blocks allocated. No leak: every block had a blocks[] owner.

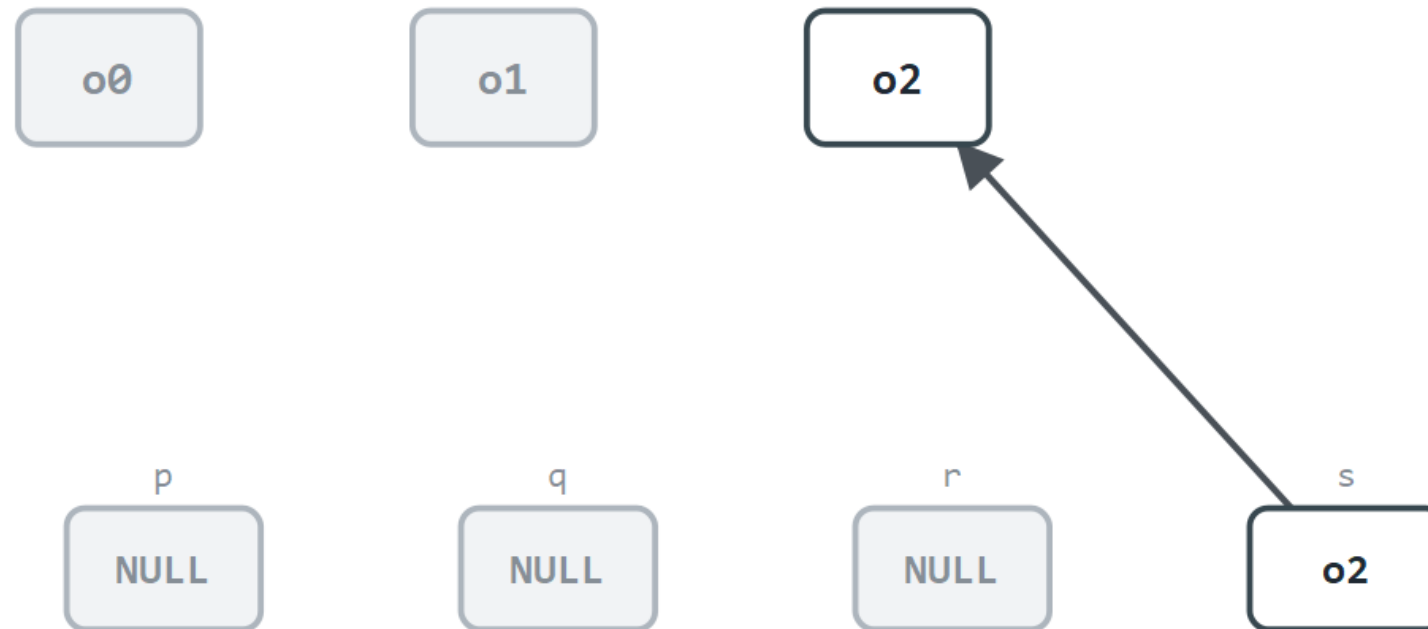
Code — alloc_block / free_block

```
static int *alloc_block(int size) {  
    int *p = malloc(size * sizeof(int));  
    return p;  
}  
  
static void free_block(int *p) {  
    free(p);  
}
```

Java: `new` and garbage collection

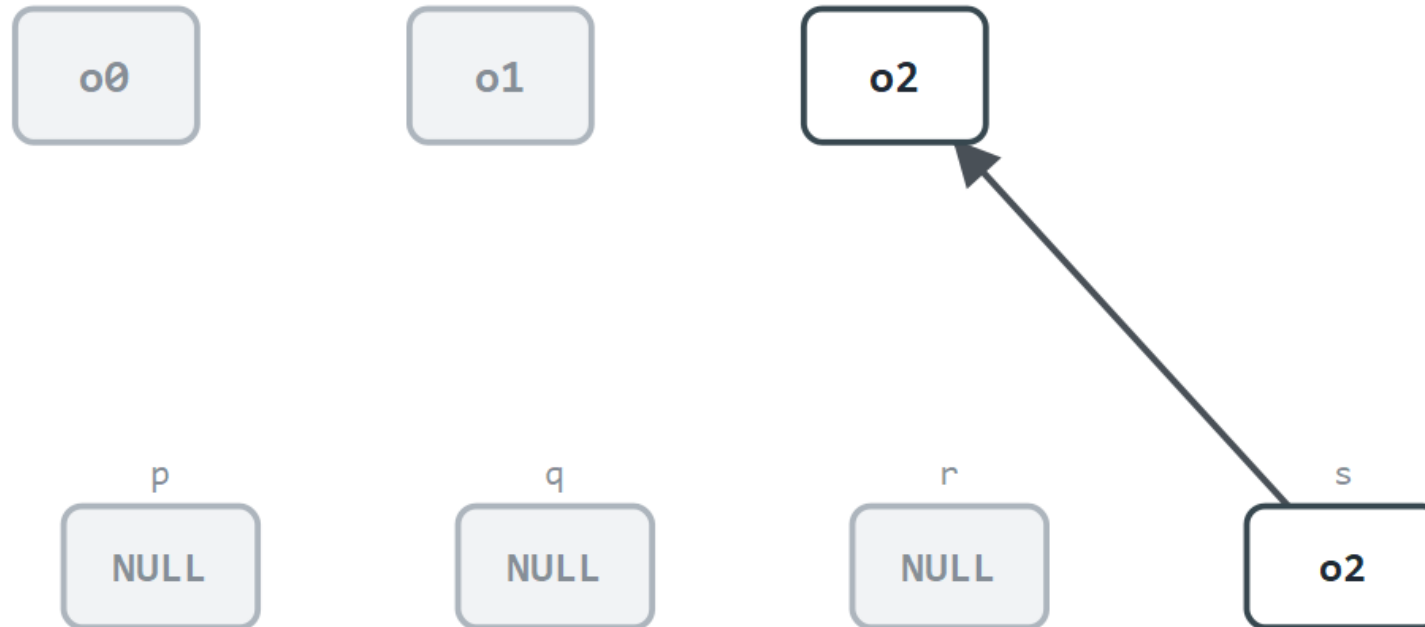
In C, forgetting `free` — or using a pointer after freeing it — is entirely your job to avoid. Java removes `free` from the language.

Java references, aliasing, GC



Done: 3 objects were created. Collected: `o0`, `o1`. Still reachable: `o2`.

Edge case — a cycle, still collected



Done: 3 objects were created. Collected: o0, o1. Still reachable: o2.

Code — Java references: alias and cycle

```
Node p = new Node();  
Node q = p;  
p.ref = q;  
q.ref = p;  
p = null;  
q = null;
```

Common mistakes

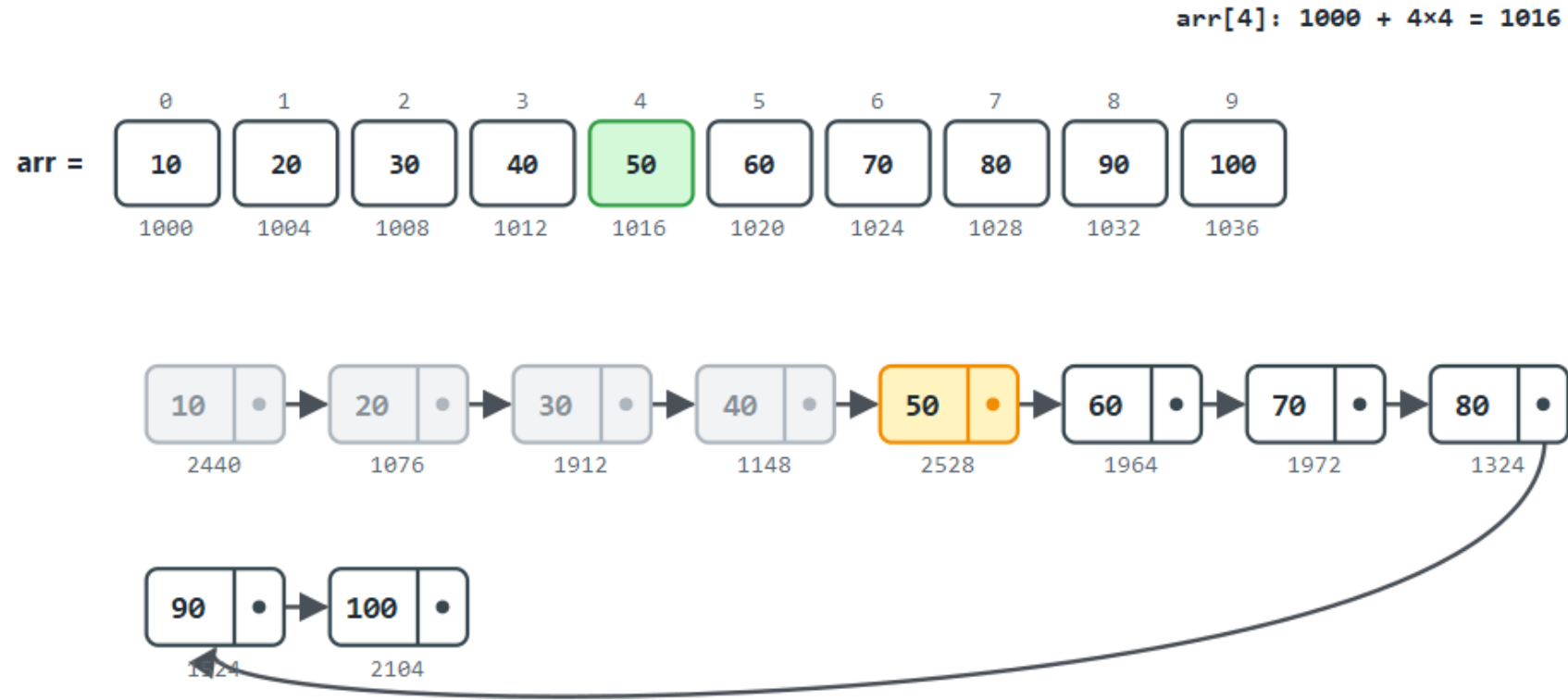
- Dangling pointer: use pointer after `free`, not `NULL`
- Double free: freeing the same pointer twice
- Forgetting `free` on an early-return path (a **leak**)
- Assuming Java's GC makes leaks impossible (it does not)

Preview: array layout vs linked layout

An array reserves one contiguous block:

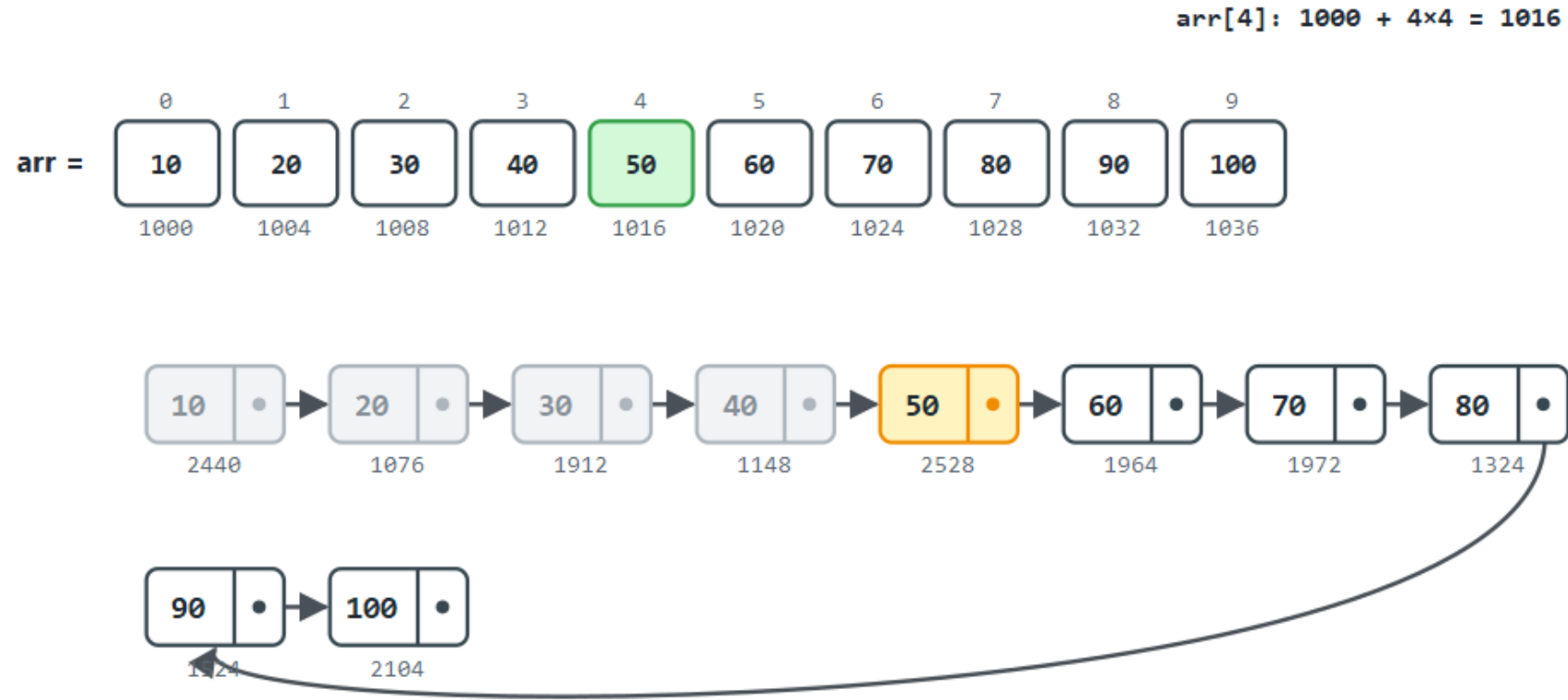
`arr[i]` is a single calculation, $O(1)$. A
linked list scatters nodes, joined by pointers.

Array vs linked layout, step by step



Result: for the same $k = 4$, the array reached it in 1 step (address 1016); the linked list started at the front and followed next 4 times: 4 hops, $O(n)$. Same data, OPPOSITE access cost.

Edge case — the last element: most hops



Result: for the same $k = 4$, the array reached it in 1 step (address 1016); the linked list started at the front and followed next 4 times: 4 hops, $O(n)$. Same data, OPPOSITE access cost.

Code — building a linked list

```
Node *head = NULL;
for (int i = N - 1; i >= 0; i--) {
    Node *node = malloc(sizeof(Node));
    node->data = arr[i];
    node->next = head;
    head = node;
}
```

Mini-quiz

A function declares `int arr[100];` locally and returns a pointer to it. What is wrong?

Answer

`arr` lives on the **stack**, in that frame.
The instant the function returns, the frame
is popped — the pointer is dangling already.

Mini-quiz

Why does `free(block); block = NULL;` — in that order — matter?

Answer

`free` needs the real address. `NULL` ing first makes `free(NULL)` do nothing — the block leaks, its address lost forever.

Mini-quiz

Why can a dangling pointer never occur in Java the way it does in C?

Answer

Java has no `free` : an object is only reclaimed once the GC proves nothing can reach it — never while still reachable.

6. ASN.1, BER TLV, and PER TLV

A question to start

Two programs, different languages, different machines, exchange a record as bytes. A raw `struct` layout is not portable at all.

A short history

- **1984** — ASN.1 born as ITU-T/CCITT X.409
- **1995** — X.680 series: the modern, current form
- **BER** (X.690) — self-describing tag + length + value
- **PER** (X.691, 1994) — packs bits when both sides know the schema

Intuition: an envelope, three parts

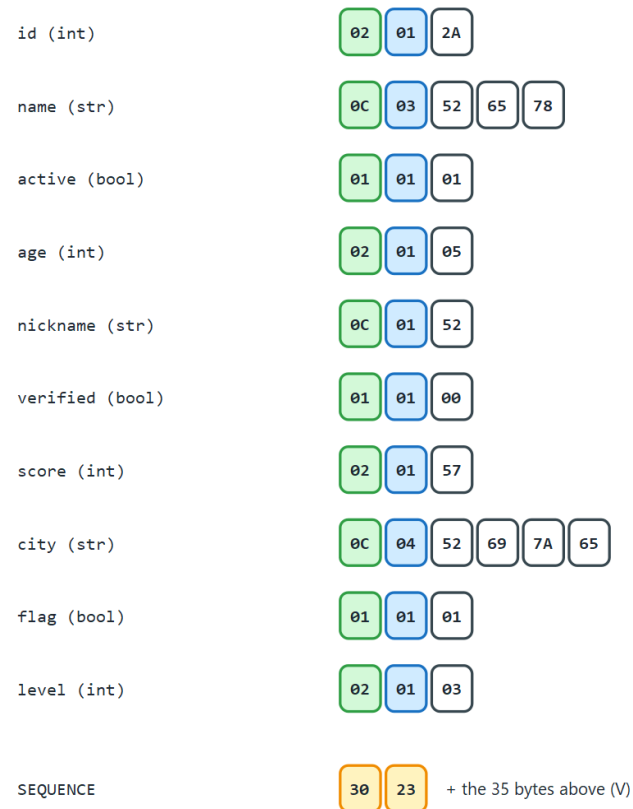
- **Tag**: what kind of value is this?
- **Length**: how many bytes does it take?
- **Value**: the content itself
- Universally called **TLV**

The TLV byte layout

Part	Size	What it encodes
Tag	1 byte	class · primitive/constructed · tag number
Length	1+ bytes	short form (0–127) or long form (≥ 128)
Value	Length bytes	the content, or nested TLVs

TLV encoding, field by field

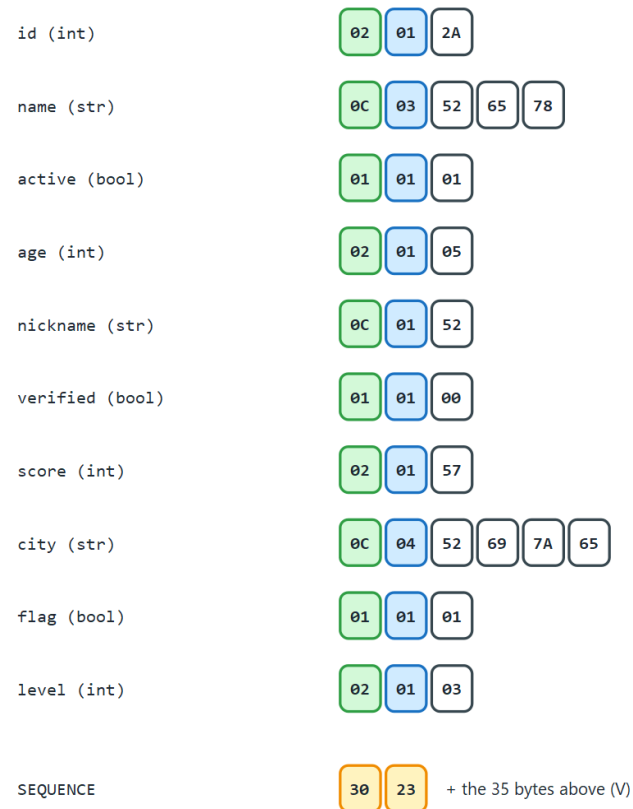
We want to send a 10-field record as a byte sequence.



Done: 37 bytes total, ready to send over the wire. 0 empty string field(s), 0 long-form length (the SEQUENCE itself).

Edge case — a length that needs the long form

We want to send a 10-field record as a byte sequence.



Done: 37 bytes total, ready to send over the wire. 0 empty string field(s), 0 long-form length (the SEQUENCE itself).

Code — encode_length: short vs long

```
if (len < 128) {  
    out[0] = (unsigned char) len;  
    return 1;  
}  
int n = 0;  
while (len > 0) {  
    tmp[n++] = (unsigned char) (len & 0xFF);  
    len >>= 8;  
}  
out[0] = (unsigned char) (0x80 | n);
```

BER vs PER

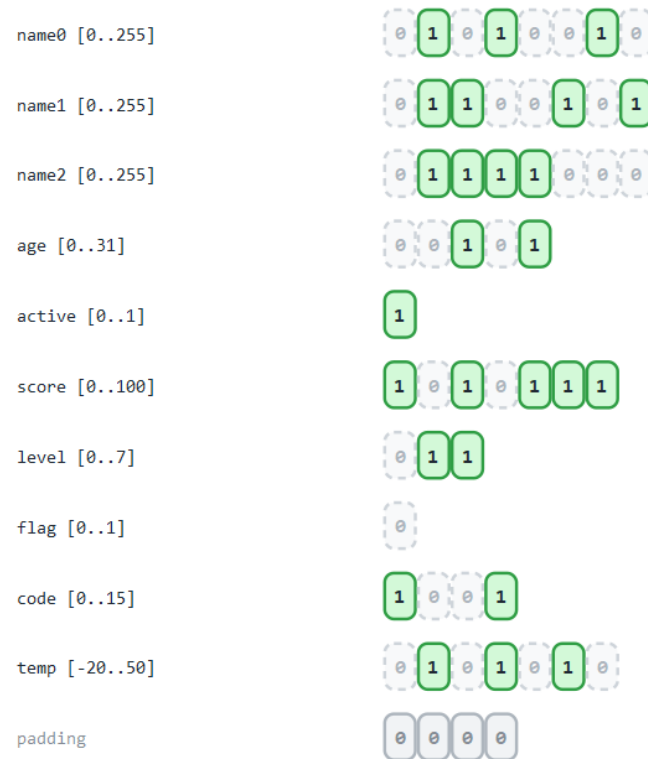
	BER	PER
Self-describing	Yes: tag + length every field	No: schema must be known
Size	Larger: 2+ bytes per field	Much smaller: bits, not bytes
Typical use	X.509, LDAP, SNMP	Bandwidth-critical (cellular)

PER: exactly as many bits as needed

If both sides know a field's range `[min, max]`,
no tag and no length are needed — only
`ceil(log2(max - min + 1))` bits.

PER encoding, bit by bit

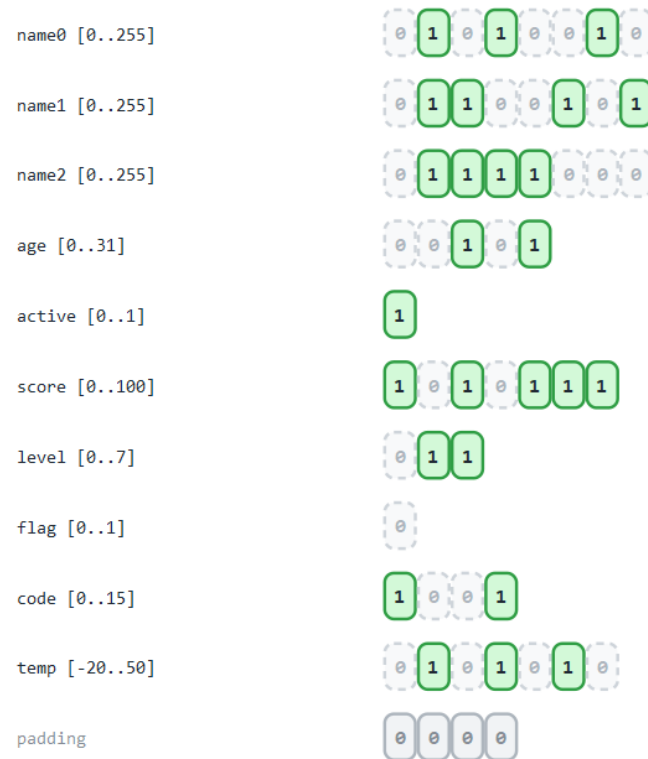
The SAME kind of 10-field record, this time packed bit by bit: no tags, no lengths.



Done: 52 significant bits, packed into 7 bytes. The same fields with a Tag and Length each (TLV-style) would need roughly 30 bytes (240 bits) — PER shrinks by giving up most of that self-description.

Edge case — a range of size 1: zero bits

The SAME kind of 10-field record, this time packed bit by bit: no tags, no lengths.



Done: 52 significant bits, packed into 7 bytes. The same fields with a Tag and Length each (TLV-style) would need roughly 30 bytes (240 bits) — PER shrinks by giving up most of that self-description.

Code — pack_bits: one bit at a time

```
int bit = (int) ((value >> i) & 1u);
int byte_index = *bitpos / 8;
int bit_index = 7 - (*bitpos % 8);
/* if (bit) buf[byte_index] |= the bit, MSB first */
(*bitpos)++;
```

Common mistakes

- Forgetting the length prefix, hoping a fixed read works
- Assuming length always fits one byte (only under 128)
- Confusing Length (just V) with the whole TLV's size
- Treating BER and PER bytes as interchangeable — they are not

Mini-quiz

What do the three letters in TLV stand for,
and in what order?

Answer

Tag, Length, Value — in exactly that order: what kind, how many bytes, then the bytes themselves.

Mini-quiz

Why is SEQUENCE's tag byte `0x30`, not `0x10` (tag number 16 in binary)?

Answer

The top 3 bits are not the tag number: 2
class bits (00) + 1 constructed bit (1 ,
since SEQUENCE holds more TLVs) = 0x30 .

7. Hands-on Lab: the C Toolchain

Why this matters

From here on, every week hands you C and Java programs to build yourself. The midterm project is graded, in part, on a clean build.

The toolchain, briefly

- **GCC** — Richard Stallman, GNU project, **1987**
- **GDB** — same GNU project, **1986**: step through, inspect
- **CMake** — generates build files for whatever tool you have
- Same `CMakeLists.txt` builds on Windows, WSL, and Linux

Code — your first compile

```
#include <stdio.h>

int main(void) {
    printf("Hello, Data Structures!\n");
    return 0;
}
```

```
gcc -std=c11 -Wall -Wextra -o /tmp/x hello_workshop.c && /tmp/x
```

Compile with warnings on, and read them

`-Wall -Wextra` catches unused variables,
suspicious comparisons, format mismatches.
Every program here must compile **cleanly**.

A small bug hunt

```
int average_buggy(const int arr[], int n) {  
    if (n == 0) return 0;  
    int sum = 0;  
    for (int i = 0; i < n; i++)  
        sum = sum + arr[i];  
    return sum / n;  
}
```

Finding it with gdb

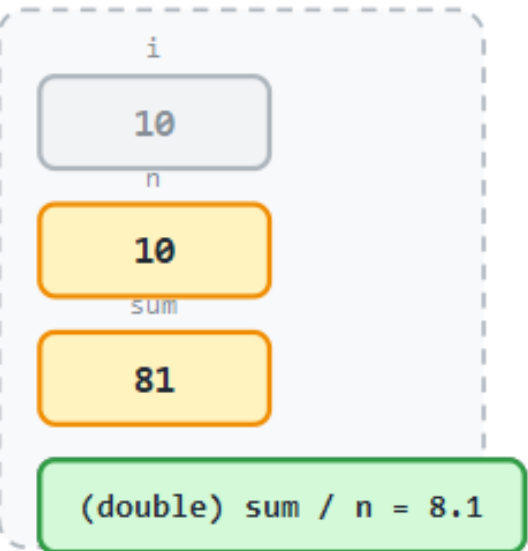
The method: set a breakpoint, run, inspect a variable, confirm or refute a hypothesis.

Watch it happen live, step by step.

Debugger stepping, live

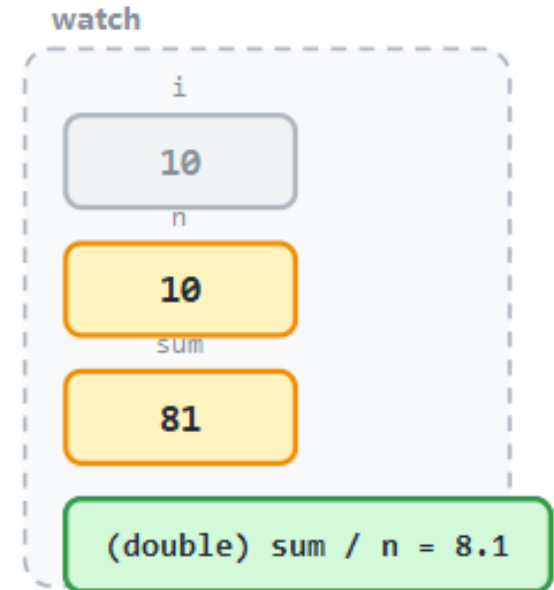


watch



continue — the program finishes. average_buggy -> 8, average_fixed -> 8.1. The method: set a breakpoint, run, print suspicious expressions, compare — not one line of code changed.

Edge case — the empty-array guard



continue — the program finishes. `average_buggy` -> 8, `average_fixed` -> 8.1. The method: set a breakpoint, run, print suspicious expressions, compare — not one line of code changed.

The gdb session

```
(gdb) break debug_average.c:11
(gdb) run
(gdb) print sum
(gdb) print sum / n
(gdb) print (double) sum / n
(gdb) continue
```

Code — CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20)
project(week1_cmake_demo C)
add_executable(week1_cmake_demo main.c)
```

```
cmake -S . -B build && cmake --build build
```

A note on Visual Studio

Build → Build Solution compiles; Debug →
Start Debugging (F5) runs under the debugger;
clicking the margin sets a breakpoint.

Common mistakes

- Forgetting `-std=c11` : different behavior per machine
- Committing a `build/` folder or `.exe` files to Git
- Running a stale executable after a failed compile
- Reaching only for `printf` , never a two-minute gdb session

Mini-quiz

What does `-Wall -Wextra` do, and why does this course require a clean build under it?

Answer

It enables a wide set of warnings. They usually point at real bugs — requiring a clean build fixes them immediately.

Mini-quiz

What is the difference between what

`cmake -S . -B build` and `cmake --build build` each do?

Answer

The first **configures**: generates build files, compiles nothing. The second **builds**: invokes that tool to compile and link.

Summary — foundations of the semester

Idea	Key fact
Data structure	Known, analyzable cost — always a trade-off
Linear vs non-linear	One "next" vs possibly several
Big-O	Upper bound on growth; read off the dominant term
Space complexity	Same idea, applied to extra memory

Summary — memory and encoding

Idea	Key fact
Pointer / reference	Stores an address; <code>&</code> , <code>*</code> , <code>-></code>
Stack / heap	Automatic, LIFO / requested, released explicitly
TLV / BER / PER	Self-describing bytes, or bits packed by schema
C toolchain	<code>gcc</code> , <code>gdb</code> , <code>CMake</code> — compile, run, debug

The big picture

Two tools, built today — **Big-O** to measure cost, and a picture of **memory** — judge every single structure for the rest of term.

Self-check round

Five short questions. Think before the answer appears. Full exercises and a ten-question quiz are in the week notes.

1. Why can two different, correct algorithms have very different Big-O?

Big-O measures the *number of steps*, not whether the answer is right

Linear search and binary search both correctly find a value, at $O(n)$ and $O(\log n)$ respectively.

2. Order these from fastest- to slowest-growing: $O(n \log n)$, $O(1)$, $O(n)$, $O(\log n)$, $O(n^2)$

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2)$$

3. Why does binary search require a sorted array?

Comparing to the middle only helps if one side is guaranteed smaller and the other larger

That guarantee is exactly what "sorted" means. Linear search relies on no ordering.

4. In a BER TLV encoding, what does the Length field actually count?

The bytes in the Value that immediately follow — not the whole TLV's size

The SEQUENCE example had Length = 8, but the whole encoded SEQUENCE was 10 bytes.

5. Why does `p->x` exist, and what is it shorthand for?

p->x is shorthand for **(*p).x** : dereference, then access the field

It exists because this exact combination is extremely common in C code.

Next week

Week 2 — Linked Lists

Build a chain of individually allocated nodes, connected by exactly the pointers you met today, living on today's heap.

References (1/2)

- Course syllabus, Week 1: `CEN207-2026-2027-Guz-Izlence.en.md`
- Knuth. *The Art of Computer Programming, Vol. 1*, 3rd ed.
- Liskov, Zilles. "Programming with Abstract Data Types," 1974
- Knuth. "Big Omicron and Big Omega and Big Theta," 1976
- Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms*, 3rd ed.

References (2/2)

- Sedgewick, Wayne. *Algorithms*, 4th ed.
- Kernighan, Ritchie. *The C Programming Language*, 2nd ed.
- Liang. *Introduction to Java Programming*, 10th ed.
- ITU-T X.680 / X.690 / X.691 — ASN.1, BER, PER
- GNU (GCC, GDB) and Kitware (CMake) documentation