Chapter 2: Arrays and Structures

Data Structures and Algorithms with C

The array ADT · Arrays and pointers · Dynamic arrays · Multidimensional arrays · addressing · Triangular and band matrices · struct · union · Strings · Matrix multiplication

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/dsa_c/slides/Chapter_02_Arrays_and_Structures.html

Learning Outcomes

Arrays, abstract and concrete

Define the array ADT and explain the contiguous memory layout of a C array.

Computing addresses

Find the address of an element of a 2-, 3- or n-dimensional array in row- or column-major order.

Saving memory

Store triangular and band matrices in a one-dimensional array with O(1) access.

Dynamic memory

Build growable arrays and 2D arrays with malloc/realloc/free; explain the amortized cost.

Record types

Use structs, unions and strings (char arrays) correctly and safely.

Matrix operations

Count the steps of matrix addition and multiplication and show the $O(n^3)$ result.

An Array: a Set of ⟨Index, Value⟩ Pairs

Abstractly, an array is a **mapping**: each defined index is associated with a value. "Consecutive memory cells" is the **most common implementation** of this ADT, not its definition.

Operation	Meaning
`Create(j, list)`	an empty array of j dimensions with index set `list`
`Retrieve(A, i)`	return A[i] if i is a valid index, else an error
`Store(A, i, x)`	if i is valid, return a new array where i maps to x

Why is it fast?

Given the base address α and element size s, A[i] is at **α + i·s**: one multiply and one add, **O(1)**.

The price

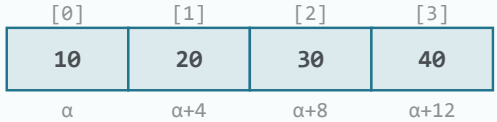
The size is fixed up front; inserting or deleting in the middle shifts the elements after it: **O(n)**.

Declaring Arrays in C, and Addresses

```
int list[5], *plist[5];
/* list : 5 ints          */
/* plist: 5 int pointers */

list[i]  /* address: α + i * sizeof(int) */
sizeof list   /* 5 * sizeof(int) bytes */
```

The compiler keeps the address of `list[0]` (α); the address of `list[i]` is computed **at run time**. Indices are **not checked**: writing `list[5]` is undefined behavior.



Addresses assuming sizeof(int) = 4

Indices start at 0: the valid indices of an n-element array are **0 … n−1**.

Array Name ≈ Address of the First Element

Expression	Equivalent
`list`	`&list[0]`
`list + i`	`&list[i]` (i·sizeof(int) bytes further)
`*(list + i)`	`list[i]`
`p = list; p[i]`	the same cell as `list[i]`

Pointer arithmetic counts **elements**, not bytes: `p + 1` points to the next `int`.

Difference 1: no assignment

`list = p;` does not compile; an array name is a constant address, not a variable.

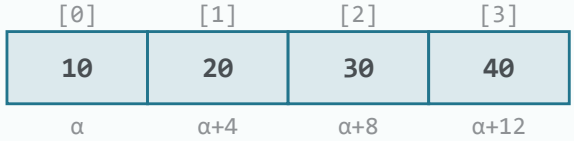
Difference 2: sizeof

For a local array, `sizeof list` gives the whole array; for a parameter, the pointer size.

Parameters

`void f(int a[])` ≡ `void f(int *a)`: the array is not copied, its address is passed; the size goes in a separate parameter.

Pointer Arithmetic: `*(p + i)`



`arr` is allocated on the stack as 4 consecutive `int`s; call the address of the first one **α**.

Step 1 / 12

```
1  void print_array(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
▶    int arr[] = {10, 20, 30, 40};
9    print_array(arr, 4);
10   return 0;
11 }
```

VARIABLES

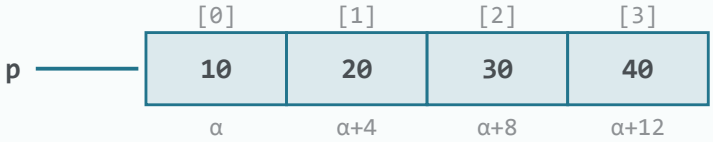
`arr = [10, 20, 30, 40]`

CALL STACK

`main()`

OUTPUT

Pointer Arithmetic: `*(p + i)`



The call does **not copy** the array: only the value `&arr[0]` (α) is passed to `p`.

Step 2 / 12

```
1  void print_array(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int arr[] = {10, 20, 30, 40};
▶    print_array(arr, 4);
10   return 0;
11 }
```

VARIABLES

`arr = [10, 20, 30, 40]`

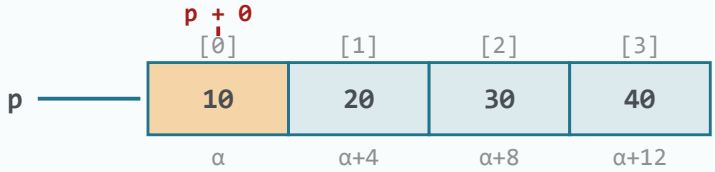
CALL STACK

`print_array(α, 4)`

`main()`

OUTPUT

Pointer Arithmetic: `*(p + i)`



0 < 4 → keep looping. `p + 0` = α + 0·4 = **α+0**.


```
1  void print_array(int *p, int n) {
2    int i;
▶   for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int arr[] = {10, 20, 30, 40};
9    print_array(arr, 4);
10   return 0;
11 }
```

VARIABLES

`p = α` `n = 4` `i = 0`

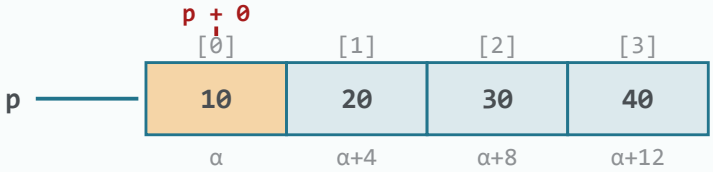
CALL STACK

`print_array(α, 4)`

`main()`

OUTPUT

Pointer Arithmetic: `*(p + i)`



`*(p + 0)` reads the value at that address: **10** (≡ `p[0]`).

Step 4 / 12

```
1  void print_array(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
▶      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int arr[] = {10, 20, 30, 40};
9    print_array(arr, 4);
10   return 0;
11 }
```

VARIABLES

`p = α` `n = 4` `i = 0`

CALL STACK

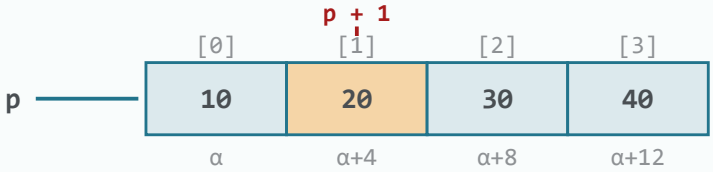
`print_array(α, 4)`

`main()`

OUTPUT

```
α 10
```

Pointer Arithmetic: `*(p + i)`



1 < 4 → keep looping. `p + 1` = α + 1·4 = **α+4**.

Step 5 / 12

```
1  void print_array(int *p, int n) {
2    int i;
▶    for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int arr[] = {10, 20, 30, 40};
9    print_array(arr, 4);
10   return 0;
11 }
```

VARIABLES

`p = α` `n = 4` `i = 1`

CALL STACK

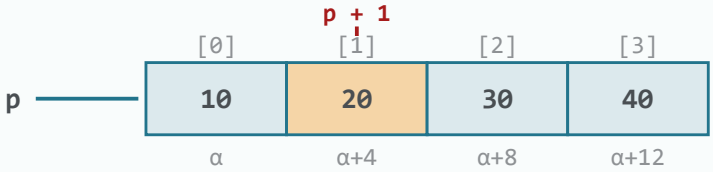
`print_array(α, 4)`

`main()`

OUTPUT

```
α 10
```

Pointer Arithmetic: `*(p + i)`



`*(p + 1)` reads the value at that address: **20** (≡ `p[1]`).

Step 6 / 12

```
1  void print_array(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
▶      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int arr[] = {10, 20, 30, 40};
9    print_array(arr, 4);
10   return 0;
11 }
```

VARIABLES

`p = α` `n = 4` `i = 1`

CALL STACK

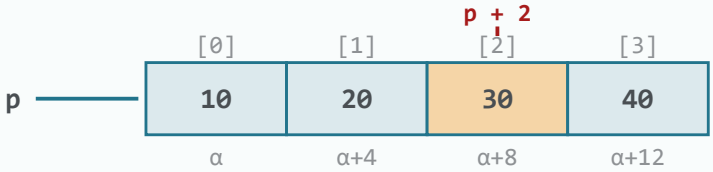
`print_array(α, 4)`

`main()`

OUTPUT

```
α 10
α+4 20
```

Pointer Arithmetic: `*(p + i)`



2 < 4 → keep looping. `p + 2` = α + 2·4 = **α+8**.

Step 7 / 12

```
1  void print_array(int *p, int n) {
2    int i;
▶   for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int arr[] = {10, 20, 30, 40};
9    print_array(arr, 4);
10   return 0;
11 }
```

VARIABLES

`p = α` `n = 4` `i = 2`

CALL STACK

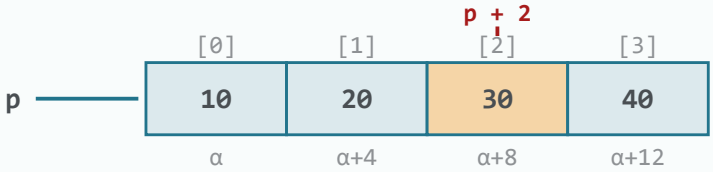
`print_array(α, 4)`

`main()`

OUTPUT

```
α 10
α+4 20
```

Pointer Arithmetic: `*(p + i)`



`*(p + 2)` reads the value at that address: **30** (≡ `p[2]`).

Step 8 / 12

```
1  void print_array(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
▶      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int arr[] = {10, 20, 30, 40};
9    print_array(arr, 4);
10   return 0;
11 }
```

VARIABLES

`p = α` `n = 4` `i = 2`

CALL STACK

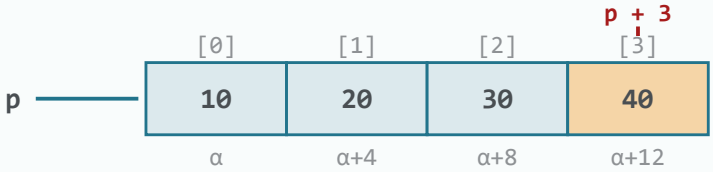
`print_array(α, 4)`

`main()`

OUTPUT

```
α 10
α+4 20
α+8 30
```

Pointer Arithmetic: `*(p + i)`



3 < 4 → keep looping. `p + 3` = α + 3·4 = **α+12**.

Step 9 / 12

```
1  void print_array(int *p, int n) {
2    int i;
▶   for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int arr[] = {10, 20, 30, 40};
9    print_array(arr, 4);
10   return 0;
11 }
```

VARIABLES

`p = α` `n = 4` `i = 3`

CALL STACK

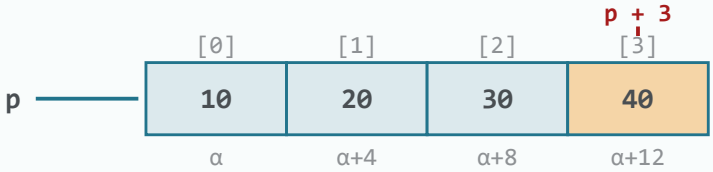
`print_array(α, 4)`

`main()`

OUTPUT

```
α 10
α+4 20
α+8 30
```

Pointer Arithmetic: `*(p + i)`



`*(p + 3)` reads the value at that address: **40** (≡ `p[3]`).

Step 10 / 12

```
1  void print_array(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
▶      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int arr[] = {10, 20, 30, 40};
9    print_array(arr, 4);
10   return 0;
11 }
```

VARIABLES

`p = α` `n = 4` `i = 3`

CALL STACK

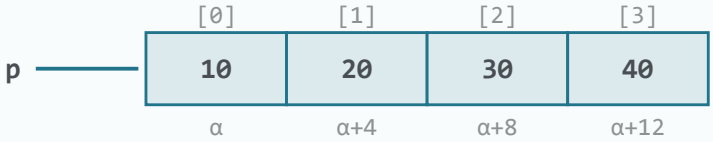
`print_array(α, 4)`

`main()`

OUTPUT

```
α 10
α+4 20
α+8 30
α+12 40
```

Pointer Arithmetic: `*(p + i)`



4 < 4 is false → the loop ends; `print_array` returns.

Step 11 / 12

```
1  void print_array(int *p, int n) {
2    int i;
▶    for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int arr[] = {10, 20, 30, 40};
9    print_array(arr, 4);
10   return 0;
11 }
```

VARIABLES

`p = α` `n = 4` `i = 4`

CALL STACK

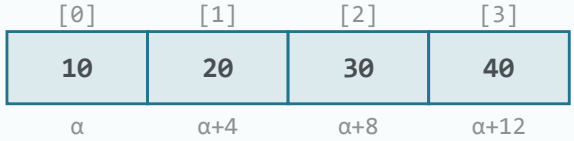
`print_array(α, 4)`

`main()`

OUTPUT

```
α 10
α+4 20
α+8 30
α+12 40
```

Pointer Arithmetic: `*(p + i)`



The address grows by 4 bytes per line. (Real `%p` output is a hexadecimal address; shown here as α.)

Step 12 / 12

```
1  void print_array(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int arr[] = {10, 20, 30, 40};
9    print_array(arr, 4);
▶    return 0;
11 }
```

VARIABLES

`arr = [10, 20, 30, 40]`

CALL STACK

`main()`

OUTPUT

```
α 10
α+4 20
α+8 30
α+12 40
```

An Array Whose Size Is Known Only at Run Time

```c
int n, i, *a;
scanf("%d", &n);
a = malloc(n * sizeof *a);    /* n ints */
if (a == NULL) {
  fprintf(stderr, "Out of memory\n");
  exit(EXIT_FAILURE);
}
for (i = 0; i < n; i++)
  a[i] = i * i;               /* like an array */
free(a);                      /* give it back */
a = NULL;           /* no dangling pointer */
```

Stack vs. heap memory

A local array disappears when the function returns; a `malloc` block lives until it is `free`d.

The `sizeof *a` idiom

Gives the right size even if the type changes; avoids mixing it up with `sizeof(int*)`.

`calloc(n, s)`

Allocates a zeroed block; `realloc` grows or shrinks a block.

A Growing Array: Double the Capacity



An empty dynamic array: `{NULL, 0, 0}`. We append 7, 3, 9, 4, 6 in order.

Step 1 / 19

```
▶ typedef struct { int *data; int size, cap; } DynArray;
2
3 int append(DynArray *d, int x) {
4   if (d->size == d->cap) {                /* full */
5     int newcap = d->cap ? 2 * d->cap : 1;
6     int *t = realloc(d->data, newcap * sizeof *t);
7     if (t == NULL) return 0;    /* old block intact */
8     d->data = t;
9     d->cap = newcap;
10  }
11  d->data[d->size++] = x;
12  return 1;
13 }
```

VARIABLES

`size = 0` `cap = 0` `copies = 0`

A Growing Array: Double the Capacity



0 == 0 → full, must grow.

Step 2 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
▶    if (d->size == d->cap) {              /* full */
5      int newcap = d->cap ? 2 * d->cap : 1;
6      int *t = realloc(d->data, newcap * sizeof *t);
7      if (t == NULL) return 0;    /* old block intact */
8      d->data = t;
9      d->cap = newcap;
10   }
11   d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

x = 7 | size = 0 | cap = 0 | copies = 0

A Growing Array: Double the Capacity



size = 0 cap = 1

+0 copies

newcap = 1. `realloc` returns a 1-cell block; the 0 old elements are copied (realloc from NULL = malloc). The result goes into `t` first, so the old block survives a failure.

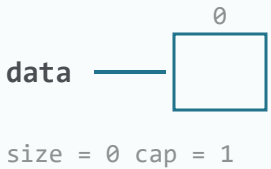
Step 3 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
4    if (d->size == d->cap) {                /* full */
▶      int newcap = d->cap ? 2 * d->cap : 1;
▶      int *t = realloc(d->data, newcap * sizeof *t);
▶      if (t == NULL) return 0;     /* old block intact */
8      d->data = t;
9      d->cap = newcap;
10   }
11   d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

x = 7 | size = 0 | cap = 0 | newcap = 1 | copies = 0

A Growing Array: Double the Capacity



`t ≠ NULL` : `d→data = t` , **cap = 1**.

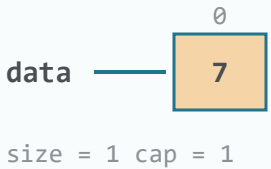
Step 4 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
4    if (d->size == d->cap) {                /* full */
5      int newcap = d->cap ? 2 * d->cap : 1;
6      int *t = realloc(d->data, newcap * sizeof *t);
7      if (t == NULL) return 0;     /* old block intact */
▶      d->data = t;
▶      d->cap = newcap;
10   }
11   d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 7` `size = 0` `cap = 1` `newcap = 1` `copies = 0`

A Growing Array: Double the Capacity



`data[0] = 7`, then **size++** → 1. (Postfix: use first, then increment.)

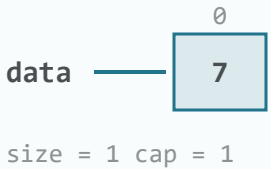
Step 5 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
4    if (d->size == d->cap) {                /* full */
5      int newcap = d->cap ? 2 * d->cap : 1;
6      int *t = realloc(d->data, newcap * sizeof *t);
7      if (t == NULL) return 0;     /* old block intact */
8      d->data = t;
9      d->cap = newcap;
10   }
▶    d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 7` `size = 1` `cap = 1` `copies = 0`

A Growing Array: Double the Capacity



1 == 1 → full, must grow.

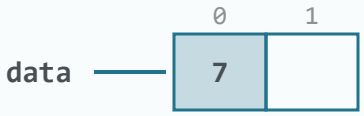
Step 6 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
▶    if (d->size == d->cap) {              /* full */
5      int newcap = d->cap ? 2 * d->cap : 1;
6      int *t = realloc(d->data, newcap * sizeof *t);
7      if (t == NULL) return 0;    /* old block intact */
8      d->data = t;
9      d->cap = newcap;
10   }
11   d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 3` `size = 1` `cap = 1` `copies = 0`

A Growing Array: Double the Capacity



size = 1 cap = 2

+1 copy

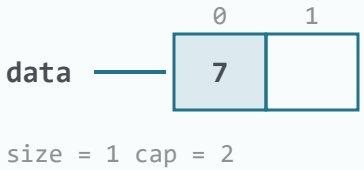
newcap = 2. `realloc` returns a 2-cell block; 1 old element is copied (1 copy so far). The result goes into `t` first, so the old block survives a failure.


```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
4    if (d->size == d->cap) {                /* full */
▶      int newcap = d->cap ? 2 * d->cap : 1;
▶      int *t = realloc(d->data, newcap * sizeof *t);
▶      if (t == NULL) return 0;     /* old block intact */
8      d->data = t;
9      d->cap = newcap;
10   }
11   d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 3` `size = 1` `cap = 1` `newcap = 2` `copies = 1`

A Growing Array: Double the Capacity



`t ≠ NULL` : `d→data = t` , **cap = 2**.

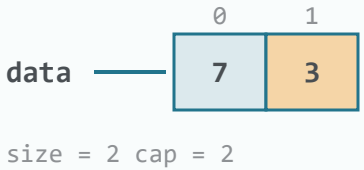
Step 8 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
4    if (d->size == d->cap) {                /* full */
5      int newcap = d->cap ? 2 * d->cap : 1;
6      int *t = realloc(d->data, newcap * sizeof *t);
7      if (t == NULL) return 0;     /* old block intact */
▶      d->data = t;
▶      d->cap = newcap;
10   }
11   d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 3` `size = 1` `cap = 2` `newcap = 2` `copies = 1`

A Growing Array: Double the Capacity



`data[1] = 3`, then **size++** → 2. (Postfix: use first, then increment.)

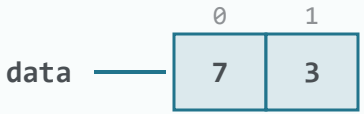
Step 9 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
4    if (d->size == d->cap) {                /* full */
5      int newcap = d->cap ? 2 * d->cap : 1;
6      int *t = realloc(d->data, newcap * sizeof *t);
7      if (t == NULL) return 0;     /* old block intact */
8      d->data = t;
9      d->cap = newcap;
10   }
▶    d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 3` `size = 2` `cap = 2` `copies = 1`

A Growing Array: Double the Capacity



2 == 2 → full, must grow.

Step 10 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
▶    if (d->size == d->cap) {              /* full */
5      int newcap = d->cap ? 2 * d->cap : 1;
6      int *t = realloc(d->data, newcap * sizeof *t);
7      if (t == NULL) return 0;    /* old block intact */
8      d->data = t;
9      d->cap = newcap;
10   }
11   d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 9` `size = 2` `cap = 2` `copies = 1`

A Growing Array: Double the Capacity



size = 2 cap = 4

+2 copies

newcap = 4. `realloc` returns a 4-cell block; 2 old elements are copied (3 copies so far). The result goes into `t` first, so the old block survives a failure.

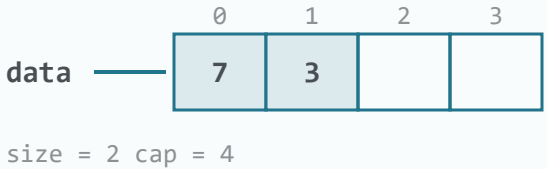
Step 11 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
4    if (d->size == d->cap) {                /* full */
▶      int newcap = d->cap ? 2 * d->cap : 1;
▶      int *t = realloc(d->data, newcap * sizeof *t);
▶      if (t == NULL) return 0;     /* old block intact */
8      d->data = t;
9      d->cap = newcap;
10   }
11   d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 9` `size = 2` `cap = 2` `newcap = 4` `copies = 3`

A Growing Array: Double the Capacity



`t ≠ NULL` : `d→data = t` , **cap = 4**.

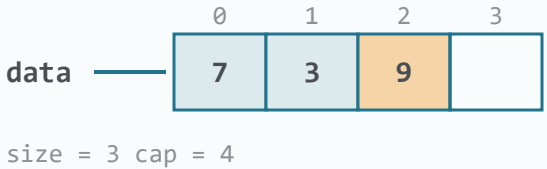
Step 12 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
4    if (d->size == d->cap) {                /* full */
5      int newcap = d->cap ? 2 * d->cap : 1;
6      int *t = realloc(d->data, newcap * sizeof *t);
7      if (t == NULL) return 0;     /* old block intact */
▶      d->data = t;
▶      d->cap = newcap;
10   }
11   d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 9` `size = 2` `cap = 4` `newcap = 4` `copies = 3`

A Growing Array: Double the Capacity



`data[2] = 9`, then **size++** → 3. (Postfix: use first, then increment.)

Step 13 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
4    if (d->size == d->cap) {                /* full */
5      int newcap = d->cap ? 2 * d->cap : 1;
6      int *t = realloc(d->data, newcap * sizeof *t);
7      if (t == NULL) return 0;     /* old block intact */
8      d->data = t;
9      d->cap = newcap;
10   }
▶    d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 9` `size = 3` `cap = 4` `copies = 3`

A Growing Array: Double the Capacity



size = 3 cap = 4

3 == 4 is false → there is room; store directly.

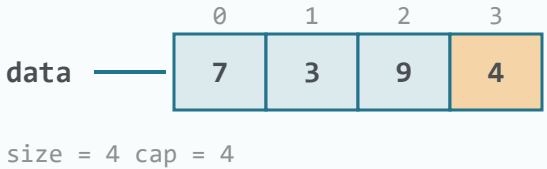
Step 14 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
▶    if (d->size == d->cap) {              /* full */
5      int newcap = d->cap ? 2 * d->cap : 1;
6      int *t = realloc(d->data, newcap * sizeof *t);
7      if (t == NULL) return 0;     /* old block intact */
8      d->data = t;
9      d->cap = newcap;
10   }
11   d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

x = 4 | size = 3 | cap = 4 | copies = 3

A Growing Array: Double the Capacity



`data[3] = 4`, then **size++** → 4. (Postfix: use first, then increment.)

Step 15 / 19

```
 1 typedef struct { int *data; int size, cap; } DynArray;
 2
 3 int append(DynArray *d, int x) {
 4   if (d->size == d->cap) {              /* full */
 5     int newcap = d->cap ? 2 * d->cap : 1;
 6     int *t = realloc(d->data, newcap * sizeof *t);
 7     if (t == NULL) return 0;    /* old block intact */
 8     d->data = t;
 9     d->cap = newcap;
10   }
▶    d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 4` `size = 4` `cap = 4` `copies = 3`

A Growing Array: Double the Capacity



4 == 4 → full, must grow.

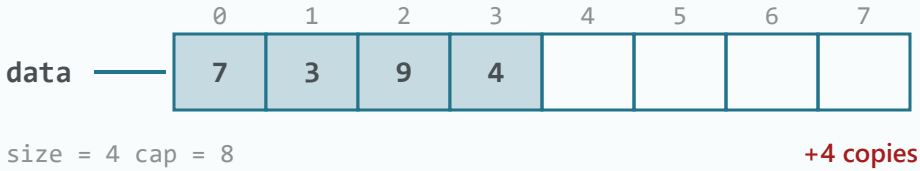
Step 16 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
▶    if (d->size == d->cap) {              /* full */
5      int newcap = d->cap ? 2 * d->cap : 1;
6      int *t = realloc(d->data, newcap * sizeof *t);
7      if (t == NULL) return 0;    /* old block intact */
8      d->data = t;
9      d->cap = newcap;
10   }
11   d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

x = 6 | size = 4 | cap = 4 | copies = 3

A Growing Array: Double the Capacity



newcap = 8. `realloc` returns an 8-cell block; 4 old elements are copied (7 copies so far). The result goes into `t` first, so the old block survives a failure.

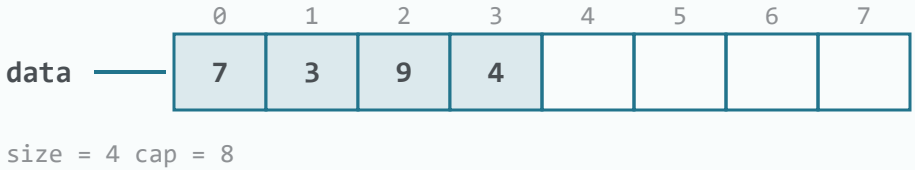
Step 17 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
4    if (d->size == d->cap) {                /* full */
▶      int newcap = d->cap ? 2 * d->cap : 1;
▶      int *t = realloc(d->data, newcap * sizeof *t);
▶      if (t == NULL) return 0;     /* old block intact */
8      d->data = t;
9      d->cap = newcap;
10   }
11   d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 6` `size = 4` `cap = 4` `newcap = 8` `copies = 7`

A Growing Array: Double the Capacity



`t ≠ NULL` : `d→data = t` , **cap = 8**.

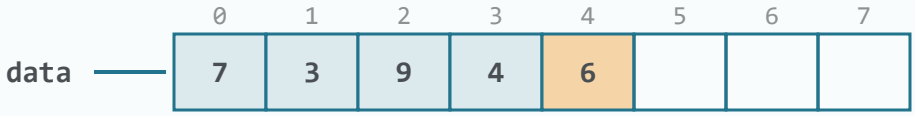
Step 18 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
4    if (d->size == d->cap) {                /* full */
5      int newcap = d->cap ? 2 * d->cap : 1;
6      int *t = realloc(d->data, newcap * sizeof *t);
7      if (t == NULL) return 0;     /* old block intact */
▶      d->data = t;
▶      d->cap = newcap;
10   }
11   d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 6` `size = 4` `cap = 8` `newcap = 8` `copies = 7`

A Growing Array: Double the Capacity



`data[4] = 6`, then **size++** → 5. Five appends cost only **7 copies** in total.

Step 19 / 19

```
1  typedef struct { int *data; int size, cap; } DynArray;
2
3  int append(DynArray *d, int x) {
4    if (d->size == d->cap) {                /* full */
5      int newcap = d->cap ? 2 * d->cap : 1;
6      int *t = realloc(d->data, newcap * sizeof *t);
7      if (t == NULL) return 0;     /* old block intact */
8      d->data = t;
9      d->cap = newcap;
10   }
▶    d->data[d->size++] = x;
12   return 1;
13 }
```

VARIABLES

`x = 6` `size = 5` `cap = 8` `copies = 7`

Doubling: Amortized O(1) per append

n appends	doubling: copies	copies / n	grow by 1: copies	copies / n
8	7	0.88	28	3.5
64	63	0.98	2016	31.5
1024	1023	1.00	523776	511.5

With doubling the copies are $1 + 2 + 4 + \ldots + 2^k < 2n \rightarrow$ **O(n)** in total (total cost < 3n including the n writes), so **O(1)** per append (amortized). Growing by 1 (or any constant c) each time costs $\mathbf{n(n-1)/2 = O(n^2)}$ copies.

A single append can still be O(n)

The append that triggers growth is expensive; amortized analysis guarantees the **average over a long sequence**.

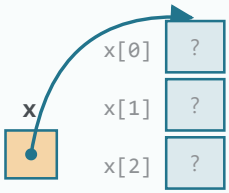
Memory overhead

Capacity is at most 2·size; the price is at most 50% empty cells.

The `realloc` trap

If `a = realloc(a, …)` fails, it returns NULL and the old block is lost (a leak). Assign to a temporary first.

A Dynamic 2D Array: An Array of Pointers + Rows



rows = 3
cols = 4

First allocate an **array of pointers** with 3 cells (`sizeof *x` = `sizeof(int *)`); its contents are still indeterminate.

Step 1 / 10

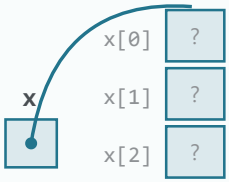
```
1  int **make2d(int rows, int cols) {
▶    int i, **x = malloc(rows * sizeof *x);
3    if (x == NULL) return NULL;
4    for (i = 0; i < rows; i++) {
5      x[i] = malloc(cols * sizeof **x);
6      if (x[i] == NULL) {                /* undo */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

VARIABLES

`rows = 3` `cols = 4`

`x = &block(3 int*)`

A Dynamic 2D Array: An Array of Pointers + Rows



rows = 3
cols = 4

`x ≠ NULL` : allocation succeeded, go on.

Step 2 / 10

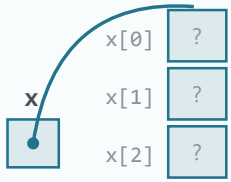
```
1  int **make2d(int rows, int cols) {
2    int i, **x = malloc(rows * sizeof *x);
▶    if (x == NULL) return NULL;
4    for (i = 0; i < rows; i++) {
5      x[i] = malloc(cols * sizeof **x);
6      if (x[i] == NULL) {                /* undo */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

VARIABLES

`rows = 3` `cols = 4`

`x = &block(3 int*)`

A Dynamic 2D Array: An Array of Pointers + Rows




```
rows = 3
cols = 4
```

i = 0 < 3 → allocate row 0.

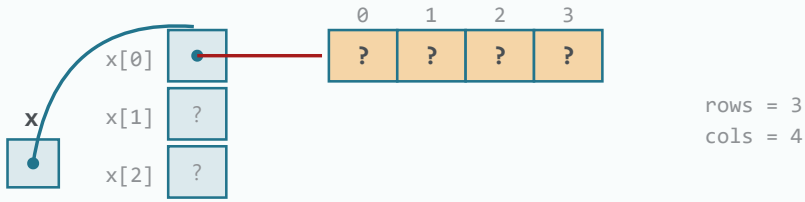
Step 3 / 10

```
1  int **make2d(int rows, int cols) {
2    int i, **x = malloc(rows * sizeof *x);
3    if (x == NULL) return NULL;
▶  4    for (i = 0; i < rows; i++) {
5      x[i] = malloc(cols * sizeof **x);
6      if (x[i] == NULL) {                /* undo */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

VARIABLES

`rows = 3` `cols = 4` `i = 0`

A Dynamic 2D Array: An Array of Pointers + Rows



`x[0]` points to a new block of 4 `int` s (`sizeof **x` = `sizeof(int)`). Not NULL.

Step 4 / 10

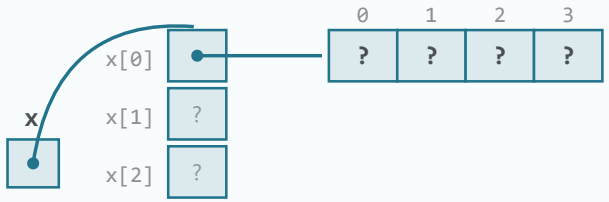
```
1  int **make2d(int rows, int cols) {
2    int i, **x = malloc(rows * sizeof *x);
3    if (x == NULL) return NULL;
4    for (i = 0; i < rows; i++) {
▶      x[i] = malloc(cols * sizeof **x);
▶      if (x[i] == NULL) {                /* undo */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

VARIABLES

`rows = 3` `cols = 4` `i = 0`

`x[0] = &row0(4 int)`

A Dynamic 2D Array: An Array of Pointers + Rows



i = 1 < 3 → allocate row 1.

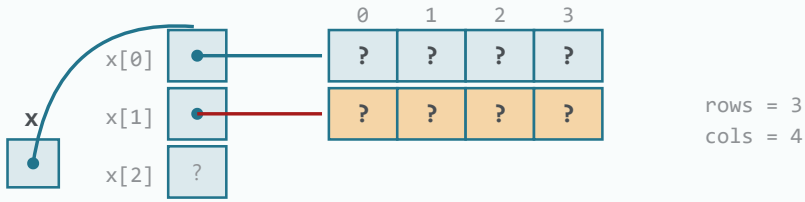
Step 5 / 10

```
1  int **make2d(int rows, int cols) {
2    int i, **x = malloc(rows * sizeof *x);
3    if (x == NULL) return NULL;
▶  for (i = 0; i < rows; i++) {
5      x[i] = malloc(cols * sizeof **x);
6      if (x[i] == NULL) {                /* undo */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

VARIABLES

`rows = 3` `cols = 4` `i = 1`

A Dynamic 2D Array: An Array of Pointers + Rows



`x[1]` points to a new block of 4 `int` s (`sizeof **x` = `sizeof(int)`). Not NULL.

Step 6 / 10

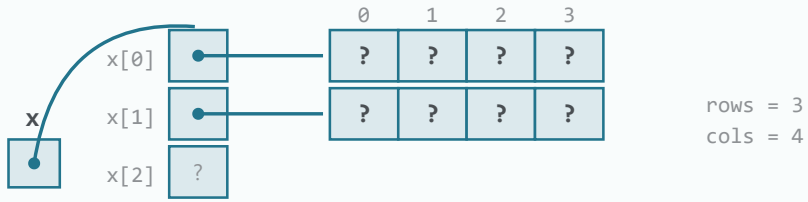
```
1  int **make2d(int rows, int cols) {
2    int i, **x = malloc(rows * sizeof *x);
3    if (x == NULL) return NULL;
4    for (i = 0; i < rows; i++) {
▶      x[i] = malloc(cols * sizeof **x);
▶      if (x[i] == NULL) {                /* undo */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

VARIABLES

`rows = 3` `cols = 4` `i = 1`

`x[1] = &row1(4 int)`

A Dynamic 2D Array: An Array of Pointers + Rows



i = 2 < 3 → allocate row 2.

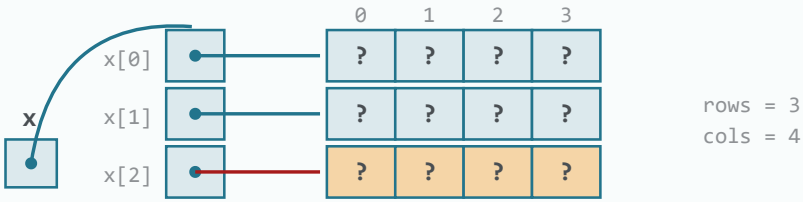
Step 7 / 10

```
1  int **make2d(int rows, int cols) {
2    int i, **x = malloc(rows * sizeof *x);
3    if (x == NULL) return NULL;
▶4   for (i = 0; i < rows; i++) {
5      x[i] = malloc(cols * sizeof **x);
6      if (x[i] == NULL) {                /* undo */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

VARIABLES

`rows = 3` `cols = 4` `i = 2`

A Dynamic 2D Array: An Array of Pointers + Rows



`x[2]` points to a new block of 4 `int` s (`sizeof **x` = `sizeof(int)`). Not NULL.

Step 8 / 10

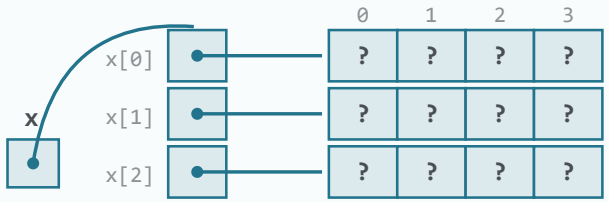
```
1  int **make2d(int rows, int cols) {
2    int i, **x = malloc(rows * sizeof *x);
3    if (x == NULL) return NULL;
4    for (i = 0; i < rows; i++) {
▶      x[i] = malloc(cols * sizeof **x);
▶      if (x[i] == NULL) {                /* undo */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

VARIABLES

`rows = 3` `cols = 4` `i = 2`

`x[2] = &row2(4 int)`

A Dynamic 2D Array: An Array of Pointers + Rows



3 < 3 is false → the loop ends.

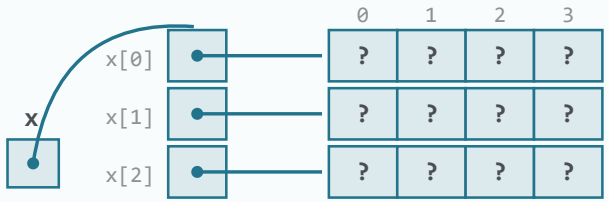
Step 9 / 10

```
1  int **make2d(int rows, int cols) {
2    int i, **x = malloc(rows * sizeof *x);
3    if (x == NULL) return NULL;
▶  for (i = 0; i < rows; i++) {
5      x[i] = malloc(cols * sizeof **x);
6      if (x[i] == NULL) {                /* undo */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

VARIABLES

`rows = 3` `cols = 4` `i = 3`

A Dynamic 2D Array: An Array of Pointers + Rows



Return `x` ; `x[i][j]` ≡ `*(*(x + i) + j)` . The rows need not be contiguous. To release it, `free` each `x[i]` first, then `x` .

Step 10 / 10

```
1  int **make2d(int rows, int cols) {
2    int i, **x = malloc(rows * sizeof *x);
3    if (x == NULL) return NULL;
4    for (i = 0; i < rows; i++) {
5      x[i] = malloc(cols * sizeof **x);
6      if (x[i] == NULL) {                  /* undo */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
▶    return x;
12 }
```

VARIABLES

`rows = 3` `cols = 4`

`x = &block(3 int*)`

Memory Is One-Dimensional, Arrays Are Not

An n-dimensional array declared as `A[upper0][upper1]…[uppern−1]` has $\prod$ **upper$_i$** elements. E.g. `A[10][10][10]` → 1000 elements.

Memory, however, is a one-dimensional array of bytes; the compiler must **lay the elements out in some order**:

- **Row major**: all of row 0 first, then row 1… (the default in C, C++, Python/NumPy)
- **Column major**: column 0 first… (Fortran, MATLAB)

The question is always the same: how many elements come **before A[i][j]**? Address = α + (number before it) · element size.

In C

`int x[3][4];` is really an array of 3 elements, each an array of 4; `x[1]` points to the start of row 1.

`int x[3][4]` Can Be Laid Out in Two Ways

	0	1	2	3
0	x00 0	x01 1	x02 2	x03 3
1	x10 4	x11 5	x12 6	x13 7
2	x20 8	x21 9	x22 10	x23 11

row major: row by row

0	1	2	3	4	5	6	7	8	9	10	11
x00	x01	x02	x03	x10	x11	x12	x13	x20	x21	x22	x23

α α + 11·s

Row major: before x[i][j] come i full rows (i·upper$_1$ elements) and j elements → **&x[i][j] = α + (i·upper$_1$ + j)·s**. C uses this.

Step 1 / 2

The formula `α + i·upper1 + j` counts elements (cells); multiply by s = sizeof(element) for a byte address.

`int x[3][4]` Can Be Laid Out in Two Ways

	0	1	2	3
0	x00 0	x01 3	x02 6	x03 9
1	x10 1	x11 4	x12 7	x13 10
2	x20 2	x21 5	x22 8	x23 11

column major: column by column

0	1	2	3	4	5	6	7	8	9	10	11
x00	x10	x20	x01	x11	x21	x02	x12	x22	x03	x13	x23

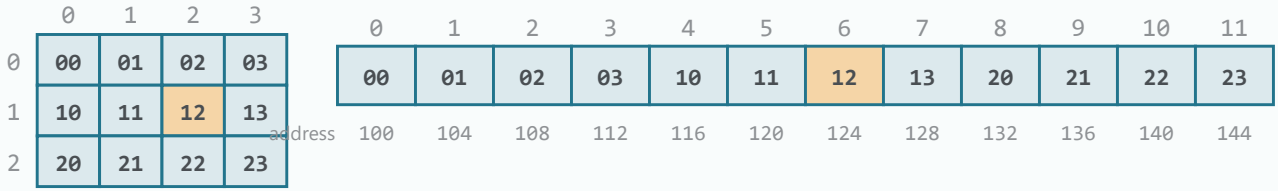
α α + 11·s

Column major: first j full columns ($j \cdot upper_0$ elements), then i elements → **$\&x[i][j] = \alpha + (j \cdot upper_0 + i) \cdot s$**. The same element lands elsewhere: x12 → 6 in row major, 7 in column major.

Step 2 / 2

The formula $\alpha + i \cdot upper_1 + j$ counts elements (cells); multiply by s = sizeof(element) for a byte address.

Computing a Row-Major Address



We want the address of `x[1][2]` : α = 100, row length **upper$_1$ = 4**, **s = 4** bytes.

Step 1 / 8

```
1  /* int x[3][4]: row-major byte address */
2  long address(long alpha, int i, int j,
3               int u1, int s) {
4    long k = (long)i * u1 + j;   /* elements before */
5    return alpha + k * s;
6  }
7
8  int main(void) {
▶    printf("%ld\n", address(100, 1, 2, 4, 4));
10   printf("%ld\n", address(100, 2, 3, 4, 4));
11   return 0;
12 }
```

VARIABLES

`alpha = 100` `i = 1` `j = 2` `u1 = 4` `s = 4`

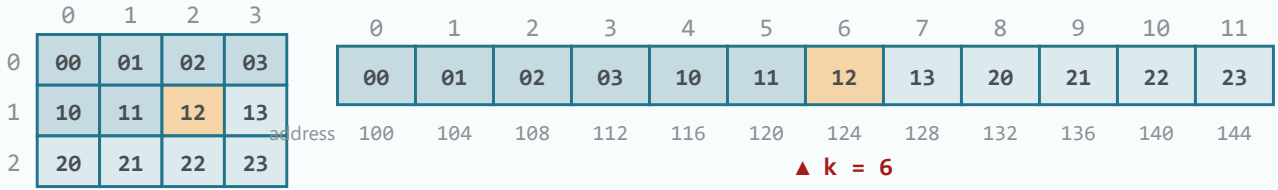
CALL STACK

`address(100, 1, 2, 4, 4)`

`main()`

OUTPUT

Computing a Row-Major Address



Before it: **1 full row** (1·4 = 4 elements) and **2 elements** in its row → **k = 6** (blue cells).

Step 2 / 8

```
1  /* int x[3][4]: row-major byte address */
2  long address(long alpha, int i, int j,
3               int u1, int s) {
▶    long k = (long)i * u1 + j;   /* elements before */
5    return alpha + k * s;
6  }
7
8  int main(void) {
9    printf("%ld\n", address(100, 1, 2, 4, 4));
10   printf("%ld\n", address(100, 2, 3, 4, 4));
11   return 0;
12 }
```

VARIABLES

alpha = 100 | i = 1 | j = 2 | u1 = 4 | s = 4 | k = 6

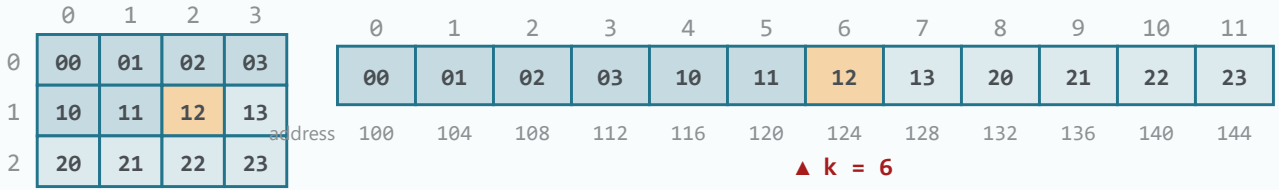
CALL STACK

address(100, 1, 2, 4, 4)

main()

OUTPUT

Computing a Row-Major Address



Return **100 + 6·4 = 124**.

Step 3 / 8

```
1  /* int x[3][4]: row-major byte address */
2  long address(long alpha, int i, int j,
3               int u1, int s) {
4    long k = (long)i * u1 + j;   /* elements before */
▶    return alpha + k * s;
6  }
7
8  int main(void) {
9    printf("%ld\n", address(100, 1, 2, 4, 4));
10   printf("%ld\n", address(100, 2, 3, 4, 4));
11   return 0;
12 }
```

VARIABLES

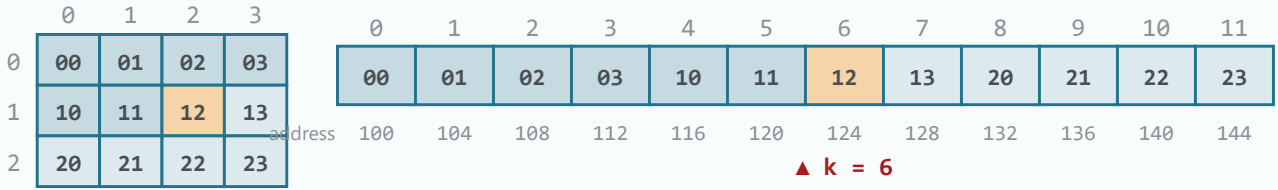
alpha = **100** · i = **1** · j = **2** · u1 = **4** · s = **4** · k = **6**

CALL STACK

- **address(100, 1, 2, 4, 4)**
- main()

OUTPUT

Computing a Row-Major Address



`printf` prints 124. In column-major order k = j·3 + i = 7 → 128.

Step 4 / 8

```
1  /* int x[3][4]: row-major byte address */
2  long address(long alpha, int i, int j,
3               int u1, int s) {
4    long k = (long)i * u1 + j;   /* elements before */
5    return alpha + k * s;
6  }
7
8  int main(void) {
▶    printf("%ld\n", address(100, 1, 2, 4, 4));
10   printf("%ld\n", address(100, 2, 3, 4, 4));
11   return 0;
12 }
```

VARIABLES

—

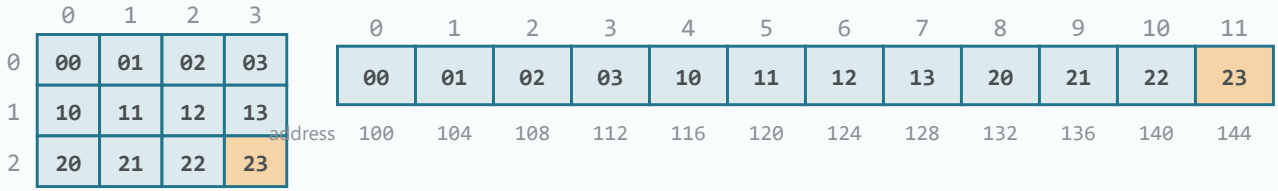
CALL STACK

`main()`

OUTPUT

```
124
```

Computing a Row-Major Address



We want the address of `x[2][3]` : α = 100, row length **upper$_1$ = 4**, **s = 4** bytes.

Step 5 / 8

```
1  /* int x[3][4]: row-major byte address */
2  long address(long alpha, int i, int j,
3               int u1, int s) {
4    long k = (long)i * u1 + j;   /* elements before */
5    return alpha + k * s;
6  }
7
8  int main(void) {
9    printf("%ld\n", address(100, 1, 2, 4, 4));
▶    printf("%ld\n", address(100, 2, 3, 4, 4));
11   return 0;
12 }
```

VARIABLES

`alpha = 100` `i = 2` `j = 3` `u1 = 4` `s = 4`

CALL STACK

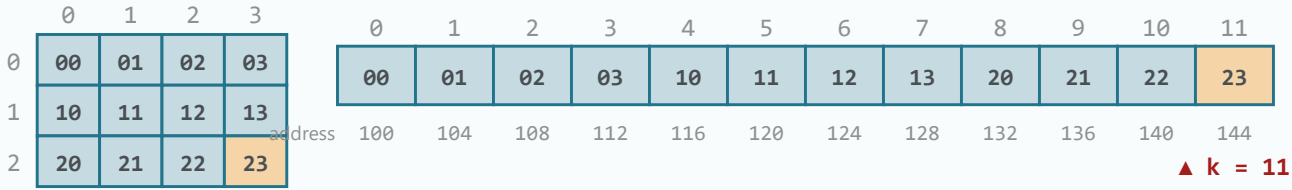
`address(100, 2, 3, 4, 4)`

`main()`

OUTPUT

```
124
```

Computing a Row-Major Address



Before it: **2 full rows** (2·4 = 8 elements) and **3 elements** in its row → **k = 11** (blue cells).

Step 6 / 8

```
1  /* int x[3][4]: row-major byte address */
2  long address(long alpha, int i, int j,
3               int u1, int s) {
▶    long k = (long)i * u1 + j;   /* elements before */
5    return alpha + k * s;
6  }
7
8  int main(void) {
9    printf("%ld\n", address(100, 1, 2, 4, 4));
10   printf("%ld\n", address(100, 2, 3, 4, 4));
11   return 0;
12 }
```

VARIABLES

alpha = 100 | i = 2 | j = 3 | u1 = 4 | s = 4 | k = 11

CALL STACK

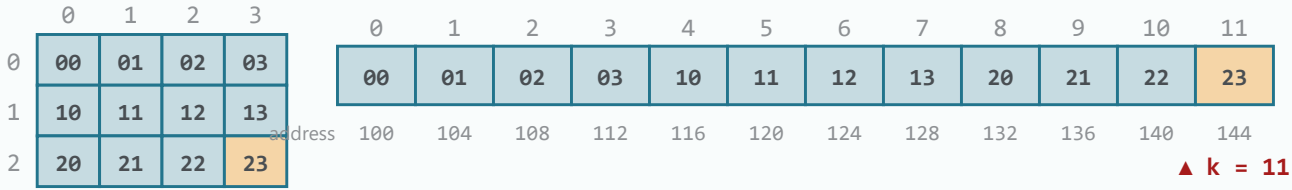
address(100, 2, 3, 4, 4)

main()

OUTPUT

```
124
```

Computing a Row-Major Address



Return **100 + 11·4 = 144**.

Step 7 / 8

```
1  /* int x[3][4]: row-major byte address */
2  long address(long alpha, int i, int j,
3               int u1, int s) {
4    long k = (long)i * u1 + j;    /* elements before */
▶    return alpha + k * s;
6  }
7
8  int main(void) {
9    printf("%ld\n", address(100, 1, 2, 4, 4));
10   printf("%ld\n", address(100, 2, 3, 4, 4));
11   return 0;
12 }
```

VARIABLES

alpha = **100** · i = **2** · j = **3** · u1 = **4** · s = **4** · k = **11**

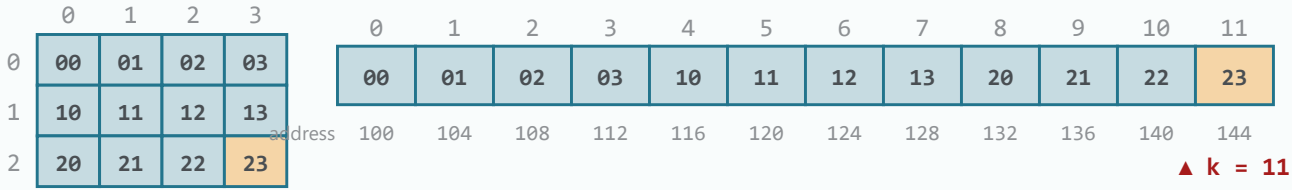
CALL STACK

- **address(100, 2, 3, 4, 4)**
- main()

OUTPUT

```
124
```

Computing a Row-Major Address



`printf` prints 144. The last element: 100 + 11·4 = 144.

Step 8 / 8

```
1  /* int x[3][4]: row-major byte address */
2  long address(long alpha, int i, int j,
3               int u1, int s) {
4    long k = (long)i * u1 + j;   /* elements before */
5    return alpha + k * s;
6  }
7
8  int main(void) {
9    printf("%ld\n", address(100, 1, 2, 4, 4));
▶    printf("%ld\n", address(100, 2, 3, 4, 4));
11   return 0;
12 }
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
124
144
```

`G[2][2][3]`: Plane · Row · Column

plane 0

`[0][0][0]` +0	`[0][0][1]` +1	`[0][0][2]` +2
`[0][1][0]` +3	`[0][1][1]` +4	`[0][1][2]` +5

plane 1

`[1][0][0]` +6	`[1][0][1]` +7	`[1][0][2]` +8
`[1][1][0]` +9	`[1][1][1]` +10	`[1][1][2]` +11

element	address (in elements)
`[0][0][0]`	α
`[i][0][0]`	$\alpha + i \cdot u_1 \cdot u_2$
`[i][j][0]`	$\alpha + i \cdot u_1 \cdot u_2 + j \cdot u_2$
`[i][j][k]`	$\alpha + i \cdot u_1 \cdot u_2 + j \cdot u_2 + k$

The last expression has **3 multiplications + 3 additions**. Multiplication costs more than addition, so the compiler factors it:

$$\alpha + (i \cdot u_1 + j) \cdot u_2 + k$$

→ **2 multiplications + 3 additions**. Example: [1][0][2] → (1·2 + 0)·3 + 2 = **8**.

The Address Formula for n Dimensions

For $A[u_0][u_1]\ldots[u_{n-1}]$:

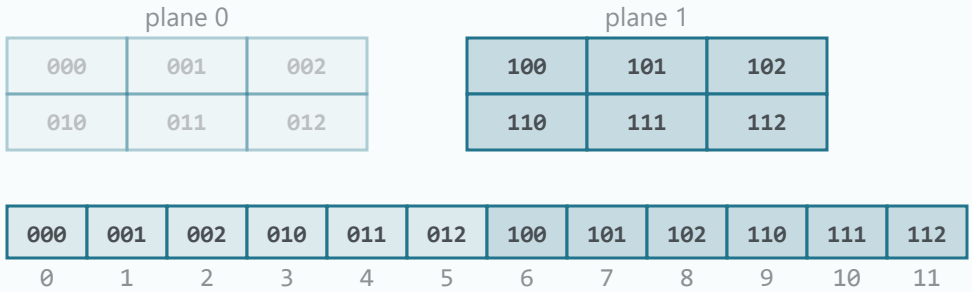
element	address
A[0][0]…[0]	α
$A[i_0][0]\ldots[0]$	$\alpha + i_0 \cdot u_1 u_2 \ldots u_{n-1}$
$A[i_0][i_1]\ldots[0]$	$\ldots + i_1 \cdot u_2 \ldots u_{n-1}$
$A[i_0]\ldots[i_{n-1}]$	$\ldots + i_{n-1}$

$$\&A[i_0]\ldots[i_{n-1}] = \alpha + \sum_{j=0}^{n-1} i_j \cdot a_j$$

$$a_j = \prod_{k=j+1}^{n-1} u_k \quad (0 \le j < n-1),\ a_{n-1} = 1$$

The coefficients a_j depend only on the dimensions, so the compiler computes them **once**. In Horner form **n−1 multiplications** suffice: $t = i_0$; $t = t \cdot u_j + i_j$.

Relative Position in n Dimensions: Horner's Method



We look for `G[1][0][2]`. **t = idx[0] = 1**: plane number 1.

Step 1 / 7

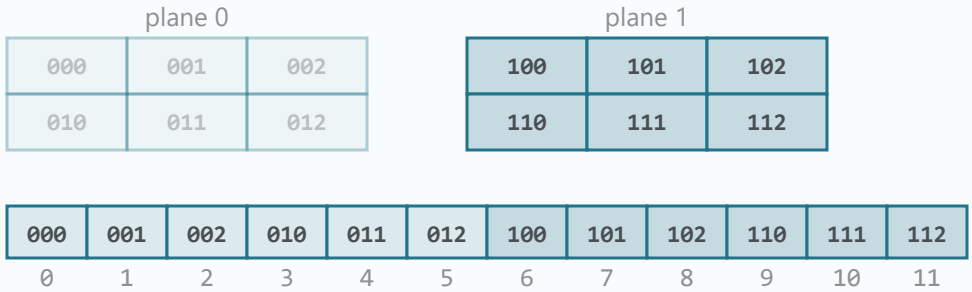
```
1  /* idx: indices, dim: sizes, n: dimensions */
2  int offset(const int idx[], const int dim[], int n) {
▶    int j, t = idx[0];
4    for (j = 1; j < n; j++)
5      t = t * dim[j] + idx[j];            /* Horner */
6    return t;                 /* address = alpha + t*s */
7  }
```

VARIABLES

`idx = [1, 0, 2]`

`dim = [2, 2, 3]` `n = 3` `t = 1`

Relative Position in n Dimensions: Horner's Method



j = 1 < 3 → one dimension further in.

Step 2 / 7

```
1 /* idx: indices, dim: sizes, n: dimensions */
2 int offset(const int idx[], const int dim[], int n) {
3   int j, t = idx[0];
▶   for (j = 1; j < n; j++)
5     t = t * dim[j] + idx[j];            /* Horner */
6   return t;                 /* address = alpha + t*s */
7 }
```

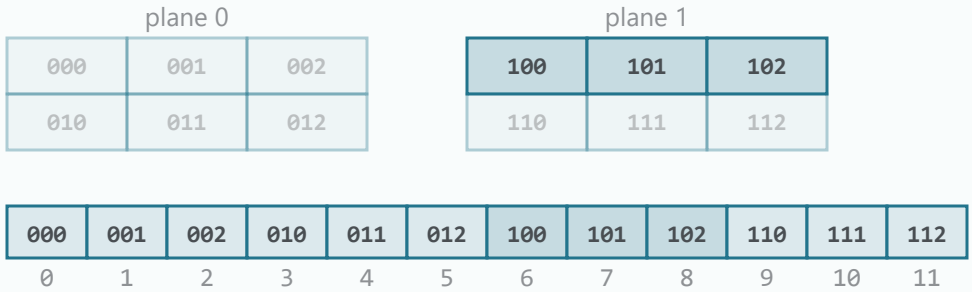
VARIABLES

idx = [1, 0, 2]

dim = [2, 2, 3] n = 3 t = 1

j = 1

Relative Position in n Dimensions: Horner's Method



t = 1·2 + 0 = 2: the "global row" number counted across planes (plane 1, row 0).

Step 3 / 7

```
1 /* idx: indices, dim: sizes, n: dimensions */
2 int offset(const int idx[], const int dim[], int n) {
3   int j, t = idx[0];
4   for (j = 1; j < n; j++)
▶     t = t * dim[j] + idx[j];            /* Horner */
6   return t;                 /* address = alpha + t*s */
7 }
```

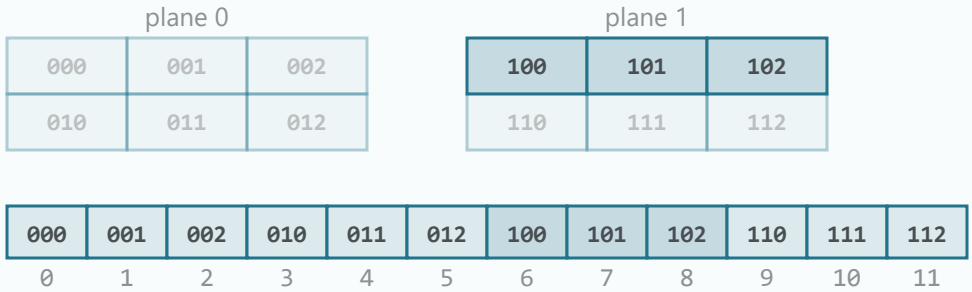
VARIABLES

idx = [1, 0, 2]

dim = [2, 2, 3] n = 3 t = 2

j = 1

Relative Position in n Dimensions: Horner's Method



j = 2 < 3 → one dimension further in.

Step 4 / 7

```
1  /* idx: indices, dim: sizes, n: dimensions */
2  int offset(const int idx[], const int dim[], int n) {
3    int j, t = idx[0];
▶    for (j = 1; j < n; j++)
5      t = t * dim[j] + idx[j];            /* Horner */
6    return t;                 /* address = alpha + t*s */
7  }
```

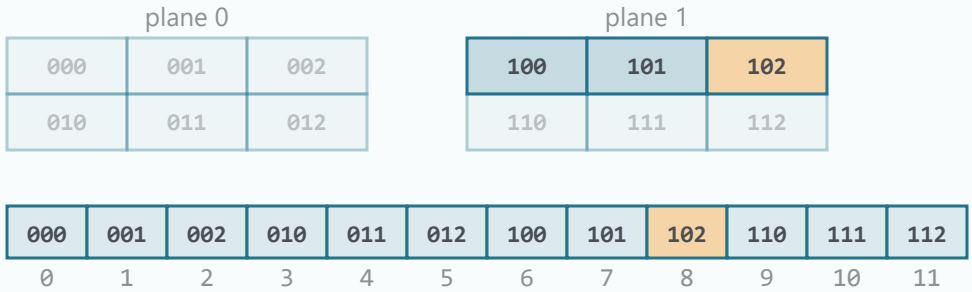
VARIABLES

`idx = [1, 0, 2]`

`dim = [2, 2, 3]` `n = 3` `t = 2`

`j = 2`

Relative Position in n Dimensions: Horner's Method



t = 2·3 + 2 = 8: 8 elements come before → cell 8.

Step 5 / 7

```
1  /* idx: indices, dim: sizes, n: dimensions */
2  int offset(const int idx[], const int dim[], int n) {
3    int j, t = idx[0];
4    for (j = 1; j < n; j++)
▶      t = t * dim[j] + idx[j];            /* Horner */
6    return t;                 /* address = alpha + t*s */
7  }
```

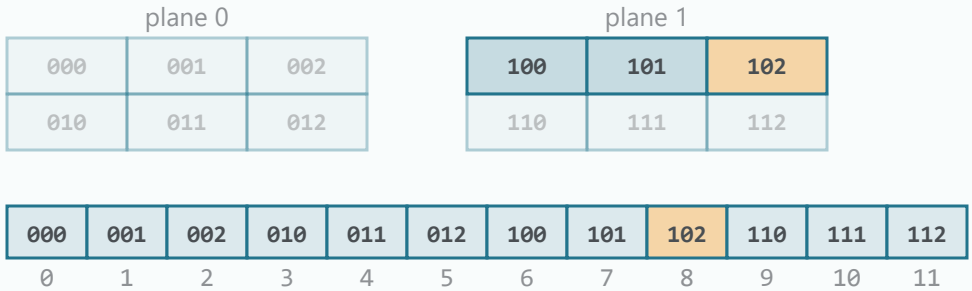
VARIABLES

idx = [1, 0, 2]

dim = [2, 2, 3] n = 3 t = 8

j = 2

Relative Position in n Dimensions: Horner's Method



3 < 3 is false → the loop ends.

Step 6 / 7

```
1 /* idx: indices, dim: sizes, n: dimensions */
2 int offset(const int idx[], const int dim[], int n) {
3   int j, t = idx[0];
▶   for (j = 1; j < n; j++)
5     t = t * dim[j] + idx[j];            /* Horner */
6   return t;                 /* address = alpha + t*s */
7 }
```

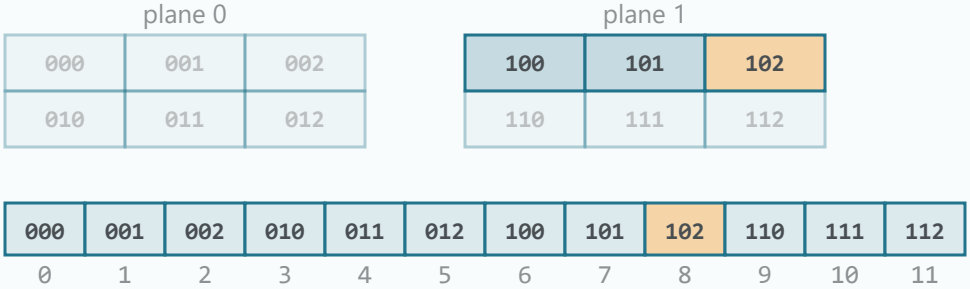
VARIABLES

idx = [1, 0, 2]

dim = [2, 2, 3] n = 3 t = 8

j = 3

Relative Position in n Dimensions: Horner's Method



Return **8**. **2 multiplications** in total (n−1). Check: 1·2·3 + 0·3 + 2 = 8. `dim[0]` was never used!

Step 7 / 7

```
1 /* idx: indices, dim: sizes, n: dimensions */
2 int offset(const int idx[], const int dim[], int n) {
3   int j, t = idx[0];
4   for (j = 1; j < n; j++)
5     t = t * dim[j] + idx[j];            /* Horner */
▶   return t;                 /* address = alpha + t*s */
7 }
```

VARIABLES

`idx = [1, 0, 2]`

`dim = [2, 2, 3]` `n = 3` `t = 8`

Keeping Half a Matrix in a One-Dimensional Array

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

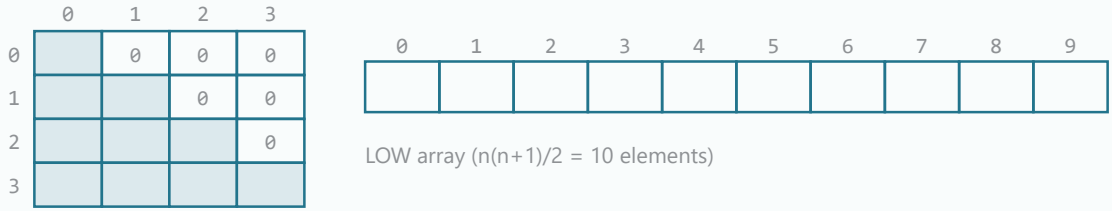
LOW array (n(n+1)/2 = 10 elements)

The diagonal splits a square matrix in two; sometimes **one half is enough** (distances between cities, atomic weights in the periodic table…). The data lives in a one-dimensional array **LOW**; the user gets an **interface** that behaves like a two-dimensional matrix.

Row i holds **i+1** elements → **n(n+1)/2** in total. For a_{ij}, count the elements of earlier rows plus its place in the row:

$$k = i(i+1)/2 + j \quad (j \le i)$$

Reading a Lower Triangle: Row start `k`



4·5/2 = **10 ≤ 100**: it fits in LOW. Input: `3 1 4 1 5 9 2 6 5 3` .

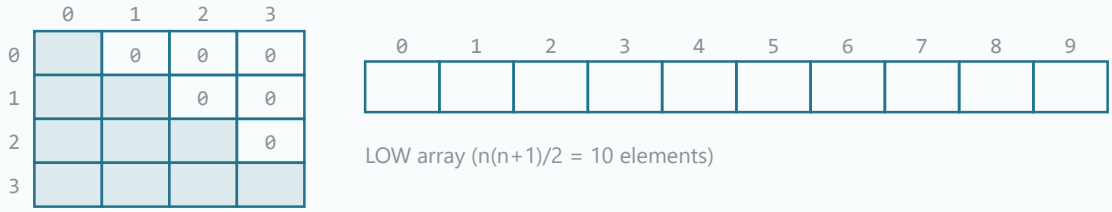
Step 1 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
▶    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
10       scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4`

Reading a Lower Triangle: Row start `k`



k = 1·0/2 = 0: 0 elements stored before row 0.

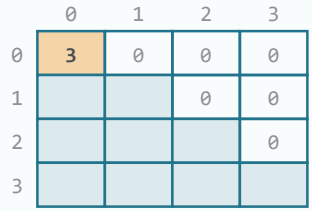
Step 2 / 16

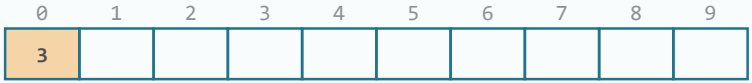
```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
▶      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
10       scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 0` `k = 0`

Reading a Lower Triangle: Row start `k`





LOW array (n(n+1)/2 = 10 elements)

`low[0+0] = low[0]` ← 3 (a_{00}).

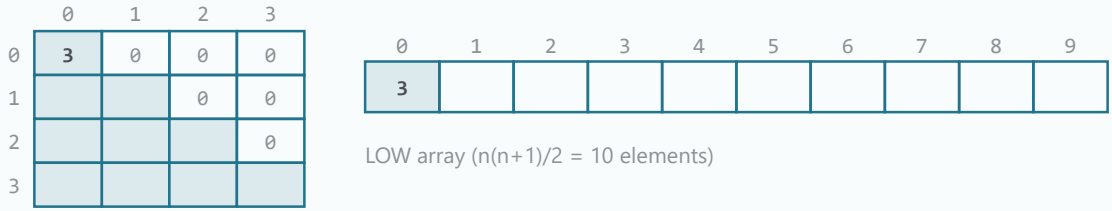
Step 3 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 0` `k = 0` `j = 0`

Reading a Lower Triangle: Row start `k`



k = 2·1/2 = 1: 1 element stored before row 1.

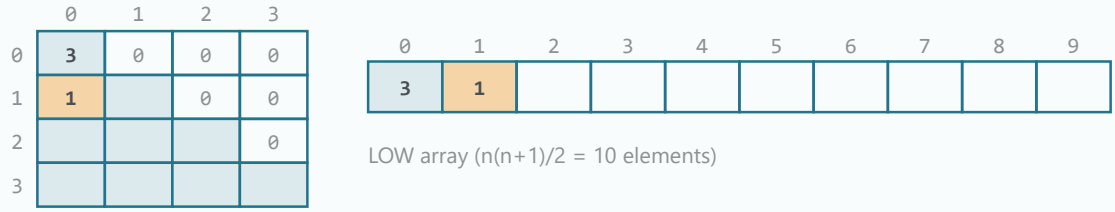
Step 4 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
▶      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
10       scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 1` `k = 1`

Reading a Lower Triangle: Row start `k`



`low[1+0] = low[1]` ← 1 (a_{10}).

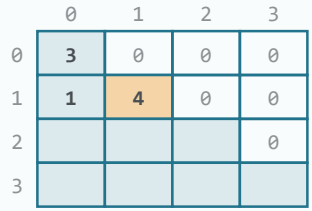
Step 5 / 16

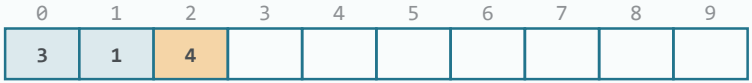
```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 1` `k = 1` `j = 0`

Reading a Lower Triangle: Row start `k`





LOW array (n(n+1)/2 = 10 elements)

`low[1+1] = low[2]` ← 4 (a_{11}).

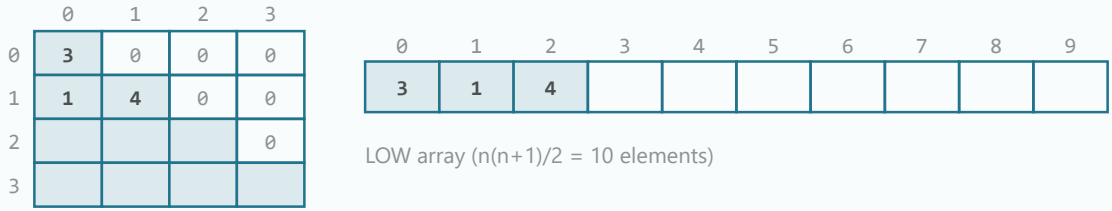
Step 6 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 1` `k = 1` `j = 1`

Reading a Lower Triangle: Row start `k`



k = 3·2/2 = 3: 3 elements stored before row 2.

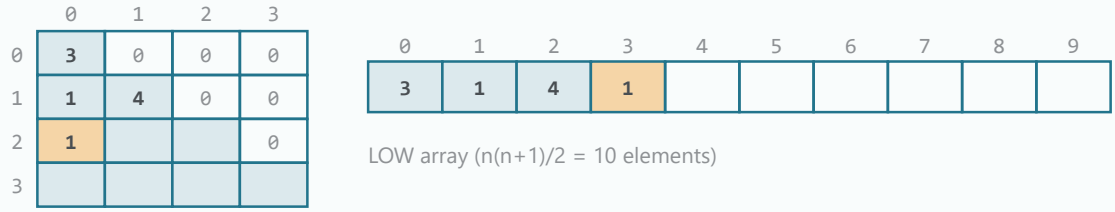
Step 7 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
▶      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
10       scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 2` `k = 3`

Reading a Lower Triangle: Row start `k`



`low[3+0] = low[3]` ← 1 (a_{20}).

Step 8 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;       /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 2` `k = 3` `j = 0`

Reading a Lower Triangle: Row start `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5		0
3				

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5					

LOW array (n(n+1)/2 = 10 elements)

`low[3+1] = low[4]` ← 5 (a_{21}).

Step 9 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 2` `k = 3` `j = 1`

Reading a Lower Triangle: Row start `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3				

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9				

LOW array (n(n+1)/2 = 10 elements)

`low[3+2] = low[5]` ← 9 (a_{22}).

Step 10 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 2` `k = 3` `j = 2`

Reading a Lower Triangle: Row start `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3				

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9				

LOW array (n(n+1)/2 = 10 elements)

k = 4·3/2 = 6: 6 elements stored before row 3.

Step 11 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
▶      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
10       scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 3` `k = 6`

Reading a Lower Triangle: Row start `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3	2			

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9	2			

LOW array (n(n+1)/2 = 10 elements)

`low[6+0] = low[6]` ← 2 (a_{30}).

Step 12 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 3` `k = 6` `j = 0`

Reading a Lower Triangle: Row start `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3	2	6		

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9	2	6		

LOW array (n(n+1)/2 = 10 elements)

`low[6+1] = low[7]` ← 6 (a_{31}).

Step 13 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 3` `k = 6` `j = 1`

Reading a Lower Triangle: Row start `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3	2	6	5	

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9	2	6	5	

LOW array (n(n+1)/2 = 10 elements)

`low[6+2] = low[8]` ← 5 (a_{32}).

Step 14 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 3` `k = 6` `j = 2`

Reading a Lower Triangle: Row start `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3	2	6	5	3

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9	2	6	5	3

LOW array (n(n+1)/2 = 10 elements)

`low[6+3] = low[9]` ← 3 (a_{33}).

Step 15 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &low[k + j]);
11   }
12 }
```

VARIABLES

`n = 4` `i = 3` `k = 6` `j = 3`

Reading a Lower Triangle: Row start `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3	2	6	5	3

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9	2	6	5	3

LOW array (n(n+1)/2 = 10 elements)

10 values read; the 6 zeros of the upper triangle were **never stored**.

Step 16 / 16

```
1  #define MAX_SIZE 100
2  void read_lower(int low[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Array too small!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* before row i */
9      for (j = 0; j <= i; j++)  /* j <= i */
10       scanf("%d", &low[k + j]);
11   }
▶ }
```

VARIABLES

`n = 4`

Access: An Index or an Error Code?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

LOW array (n(n+1)/2 = 10 elements)

We ask for a_{21}.

Step 1 / 13

```
1  int lower_index(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Invalid index\n");
4      return -2;
5    } else if (i < j) return -1;    /* upper: 0 */
6    else return (i + 1) * i / 2 + j;   /* valid */
7  }
8  /* main: */
▶  printf("%d\n", lower_index(2, 1, 4));
10 printf("%d\n", lower_index(1, 3, 4));
11 printf("%d\n", lower_index(4, 0, 4));
```

VARIABLES

i = 2 | j = 1 | n = 4

OUTPUT

Access: An Index or an Error Code?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

LOW array (n(n+1)/2 = 10 elements)

Bounds check: the indices are in 0...3.

Step 2 / 13

```
1  int lower_index(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Invalid index\n");
4      return -2;
5    } else if (i < j) return -1;    /* upper: 0 */
6    else return (i + 1) * i / 2 + j;   /* valid */
7  }
8  /* main: */
9  printf("%d\n", lower_index(2, 1, 4));
10 printf("%d\n", lower_index(1, 3, 4));
11 printf("%d\n", lower_index(4, 0, 4));
```

VARIABLES

i = 2 j = 1 n = 4

OUTPUT

Access: An Index or an Error Code?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

LOW array (n(n+1)/2 = 10 elements)

2 < 1? No → in the lower triangle.

Step 3 / 13

```
1  int lower_index(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Invalid index\n");
4      return -2;
▶    } else if (i < j) return -1;    /* upper: 0 */
6    else return (i + 1) * i / 2 + j;   /* valid */
7  }
8  /* main: */
9  printf("%d\n", lower_index(2, 1, 4));
10 printf("%d\n", lower_index(1, 3, 4));
11 printf("%d\n", lower_index(4, 0, 4));
```

VARIABLES

i = 2 | j = 1 | n = 4

OUTPUT

Access: An Index or an Error Code?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

LOW array (n(n+1)/2 = 10 elements)

k = 3·2/2 + 1 = 4 → LOW[4] = a_{21}.

Step 4 / 13

```
1  int lower_index(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Invalid index\n");
4      return -2;
5    } else if (i < j) return -1;    /* upper: 0 */
▶    else return (i + 1) * i / 2 + j;   /* valid */
7  }
8  /* main: */
9  printf("%d\n", lower_index(2, 1, 4));
10 printf("%d\n", lower_index(1, 3, 4));
11 printf("%d\n", lower_index(4, 0, 4));
```

VARIABLES

i = 2 | j = 1 | n = 4

OUTPUT

Access: An Index or an Error Code?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

LOW array (n(n+1)/2 = 10 elements)

`printf` prints 4.

Step 5 / 13

```
1  int lower_index(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Invalid index\n");
4      return -2;
5    } else if (i < j) return -1;    /* upper: 0 */
6    else return (i + 1) * i / 2 + j;   /* valid */
7  }
8  /* main: */
▶  printf("%d\n", lower_index(2, 1, 4));
10 printf("%d\n", lower_index(1, 3, 4));
11 printf("%d\n", lower_index(4, 0, 4));
```

VARIABLES

—

OUTPUT

```
4
```

Access: An Index or an Error Code?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

LOW array (n(n+1)/2 = 10 elements)

We ask for a_{13}.

Step 6 / 13

```
1  int lower_index(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Invalid index\n");
4      return -2;
5    } else if (i < j) return -1;    /* upper: 0 */
6    else return (i + 1) * i / 2 + j;   /* valid */
7  }
8  /* main: */
9  printf("%d\n", lower_index(2, 1, 4));
▶  printf("%d\n", lower_index(1, 3, 4));
11 printf("%d\n", lower_index(4, 0, 4));
```

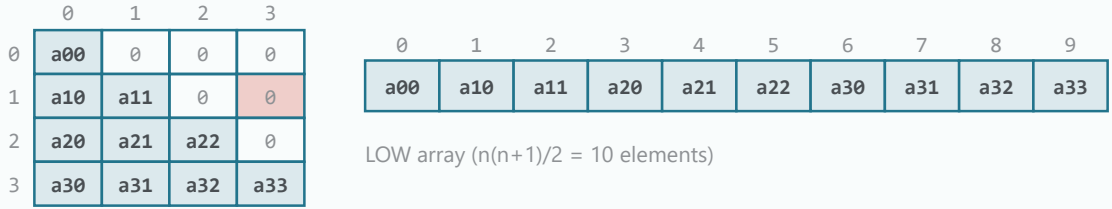
VARIABLES

i = 1 j = 3 n = 4

OUTPUT

```
4
```

Access: An Index or an Error Code?



Bounds check: the indices are in 0…3.

Step 7 / 13

```
1  int lower_index(int i, int j, int n) {
▶    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Invalid index\n");
4      return -2;
5    } else if (i < j) return -1;    /* upper: 0 */
6    else return (i + 1) * i / 2 + j;   /* valid */
7  }
8  /* main: */
9  printf("%d\n", lower_index(2, 1, 4));
10 printf("%d\n", lower_index(1, 3, 4));
11 printf("%d\n", lower_index(4, 0, 4));
```

VARIABLES

`i = 1` `j = 3` `n = 4`

OUTPUT

```
4
```

Access: An Index or an Error Code?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

LOW array (n(n+1)/2 = 10 elements)

1 < 3? Yes → upper triangle (the zero region): **−1**.

Step 8 / 13

```
1  int lower_index(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Invalid index\n");
4      return -2;
▶    } else if (i < j) return -1;    /* upper: 0 */
6    else return (i + 1) * i / 2 + j;   /* valid */
7  }
8  /* main: */
9  printf("%d\n", lower_index(2, 1, 4));
10 printf("%d\n", lower_index(1, 3, 4));
11 printf("%d\n", lower_index(4, 0, 4));
```

VARIABLES

i = 1 | j = 3 | n = 4

OUTPUT

```
4
```

Access: An Index or an Error Code?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

LOW array (n(n+1)/2 = 10 elements)

`printf` prints -1. The error codes {−1, −2} and the valid results {0, 1, 2, …} **never overlap**, so the caller can tell them apart.

Step 9 / 13

```
1  int lower_index(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Invalid index\n");
4      return -2;
5    } else if (i < j) return -1;    /* upper: 0 */
6    else return (i + 1) * i / 2 + j;   /* valid */
7  }
8  /* main: */
9  printf("%d\n", lower_index(2, 1, 4));
▶ printf("%d\n", lower_index(1, 3, 4));
11 printf("%d\n", lower_index(4, 0, 4));
```

VARIABLES

—

OUTPUT

```
4
-1
```

Access: An Index or an Error Code?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

LOW array (n(n+1)/2 = 10 elements)

We ask for a_{40}.

Step 10 / 13

```
1  int lower_index(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Invalid index\n");
4      return -2;
5    } else if (i < j) return -1;    /* upper: 0 */
6    else return (i + 1) * i / 2 + j;   /* valid */
7  }
8  /* main: */
9  printf("%d\n", lower_index(2, 1, 4));
10 printf("%d\n", lower_index(1, 3, 4));
▶  printf("%d\n", lower_index(4, 0, 4));
```

VARIABLES

i = 4 j = 0 n = 4

OUTPUT

```
4
-1
```

Access: An Index or an Error Code?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

LOW array (n(n+1)/2 = 10 elements)

Bounds check: **i = 4 ≥ n** → invalid!

Step 11 / 13

```
1  int lower_index(int i, int j, int n) {
▶    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Invalid index\n");
4      return -2;
5    } else if (i < j) return -1;    /* upper: 0 */
6    else return (i + 1) * i / 2 + j;   /* valid */
7  }
8  /* main: */
9  printf("%d\n", lower_index(2, 1, 4));
10 printf("%d\n", lower_index(1, 3, 4));
11 printf("%d\n", lower_index(4, 0, 4));
```

VARIABLES

`i = 4` `j = 0` `n = 4`

OUTPUT

```
4
-1
```

Access: An Index or an Error Code?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

LOW array (n(n+1)/2 = 10 elements)

Print an error message and return **−2**.

Step 12 / 13

```
1  int lower_index(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
▶      printf("Invalid index\n");
▶      return -2;
5    } else if (i < j) return -1;    /* upper: 0 */
6    else return (i + 1) * i / 2 + j;   /* valid */
7  }
8  /* main: */
9  printf("%d\n", lower_index(2, 1, 4));
10 printf("%d\n", lower_index(1, 3, 4));
11 printf("%d\n", lower_index(4, 0, 4));
```

VARIABLES

`i = 4` `j = 0` `n = 4`

OUTPUT

```
4
-1
Invalid index
```

Access: An Index or an Error Code?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

LOW array (n(n+1)/2 = 10 elements)

`printf` prints -2.

Step 13 / 13

```
1  int lower_index(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Invalid index\n");
4      return -2;
5    } else if (i < j) return -1;    /* upper: 0 */
6    else return (i + 1) * i / 2 + j;   /* valid */
7  }
8  /* main: */
9  printf("%d\n", lower_index(2, 1, 4));
10 printf("%d\n", lower_index(1, 3, 4));
▶  printf("%d\n", lower_index(4, 0, 4));
```

VARIABLES

—

OUTPUT

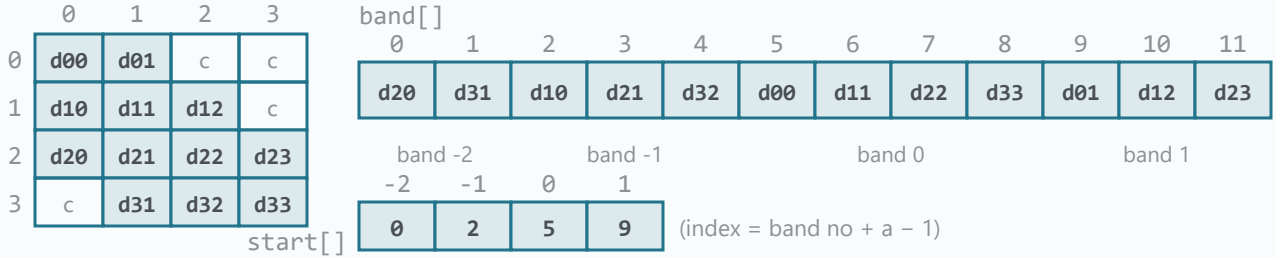
```
4
-1
Invalid index
-2
```

EXERCISE Solution: Four Kinds of Triangles and Symmetric Matrices

	stored region	elements in row i	k (row major)
	$j \le i$ (lower left)	$i + 1$	$\mathbf{i(i+1)/2 + j}$
	$j \ge i$ (upper right)	$n - i$	$\mathbf{i{\cdot}n - i(i-1)/2 + (j - i)}$
	$i + j \le n-1$ (upper left)	$n - i$	$\mathbf{i{\cdot}n - i(i-1)/2 + j}$
	$i + j \ge n-1$ (lower right)	$i + 1$	$\mathbf{i(i+1)/2 + j - (n-1-i)}$

A real symmetric matrix ($a_{ij} = a_{ji}$): instead of returning −1 for the upper triangle, swap the indices: if $i < j$, `k = j(j+1)/2 + i`. The formulas were verified by a script for $n = 1 \ldots 8$.

Band Matrices: Store Only the Diagonal band



Elements outside the band are unused or all equal to a **constant c**. n: side (4), a: number of lower bands *including the diagonal* (3), b: number of upper bands *including the diagonal* (2).

```
elements = n(a+b−1) − a(a−1)/2 − b(b−1)/2
                    = 12
```

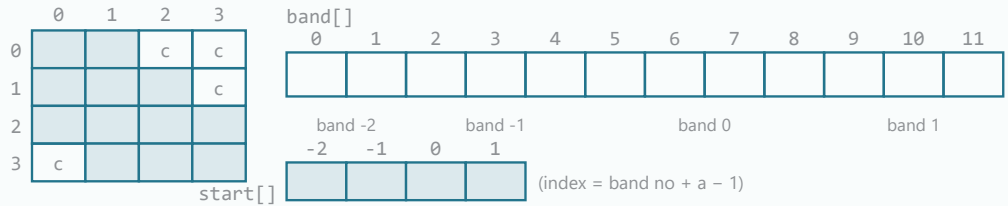
> **Instructor** *"Storing it is not the problem; can we access it quickly?"*

Band number j − i and the START Array

- **j − i** is constant along a band: for d20 and d31, 0−2 = 1−3 = **−2**. The main diagonal is 0; negatives are lower bands, positives upper.
- Valid bands lie in the closed range **[1 − a, b − 1]**; outside is the constant region.
- **START** holds the position in `band[]` of each band's first element: a + b − 1 entries. C has no negative indices, so we shift by **band number + a − 1**.
- Position within the band: **j** in a lower band, **i** in an upper band (each band starts in row 0 or column 0).

> **Instructor** *"I design the mechanism by analyzing as if every resource were scarce."*

Building a band Matrix (Lower Bands)



4·4 − 3 − 1 = **12 ≤ MAX_SIZE**. Input arrives band by band: 6 10 · 3 7 11 · 1 4 8 12 · 2 5 9.

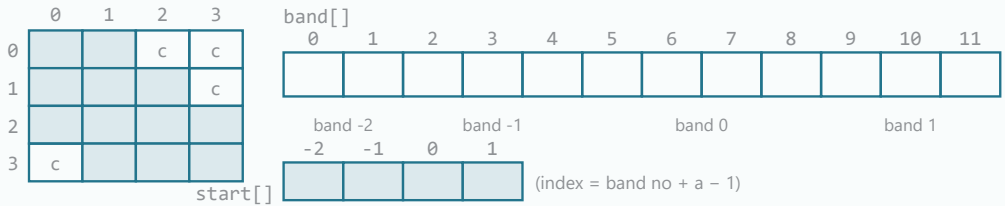
Step 1 / 12

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
▶    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `total = 0`

Building a band Matrix (Lower Bands)



Band i = -2 (lower): 2 elements.

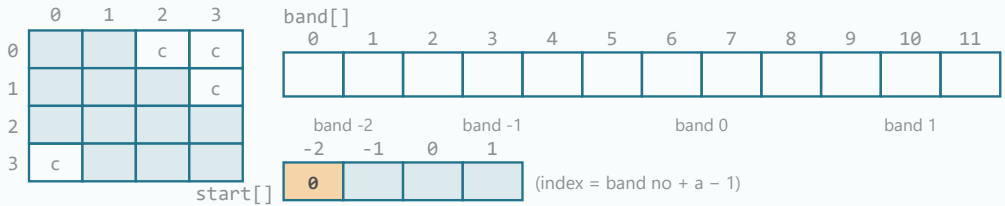
Step 2 / 12

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
▶    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = -2` `total = 0`

Building a band Matrix (Lower Bands)



`start[-2+2] = start[0] = 0` : the band begins at `band[0]`.

Step 3 / 12

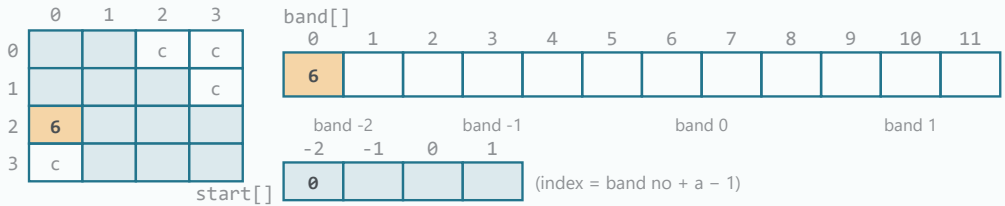
```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
▶      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = -2`

`total = 0`

Building a band Matrix (Lower Bands)



`band[0+0]` ← 6 (d_{20}).

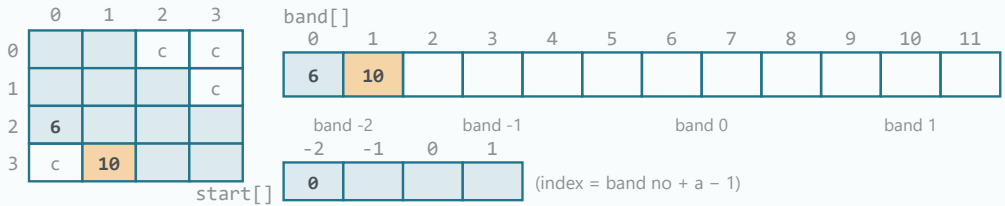
Step 4 / 12

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = -2` `k = 0` `total = 0`

Building a band Matrix (Lower Bands)



`band[0+1]` ← 10 (d_{31}).

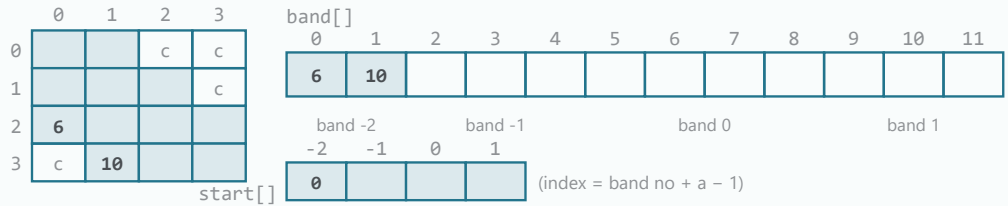
Step 5 / 12

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = -2` `k = 1` `total = 0`

Building a band Matrix (Lower Bands)



total += 2 → 2: where the next band begins.

Step 6 / 12

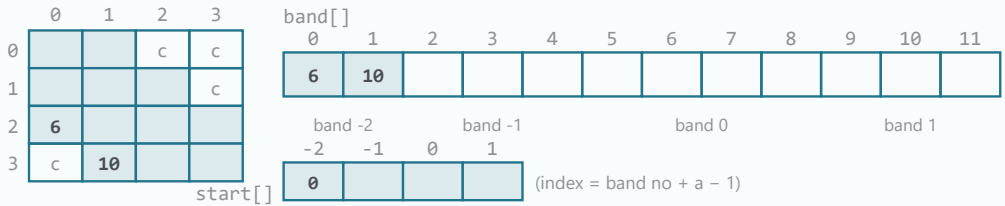
```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &band[start[i + a - 1] + k]);
▶      total += n - abs(i);
12   }
13 }
```

VARIABLES

n = 4 a = 3 b = 2 i = −2

total = 2

Building a band Matrix (Lower Bands)



Band i = -1 (lower): 3 elements.

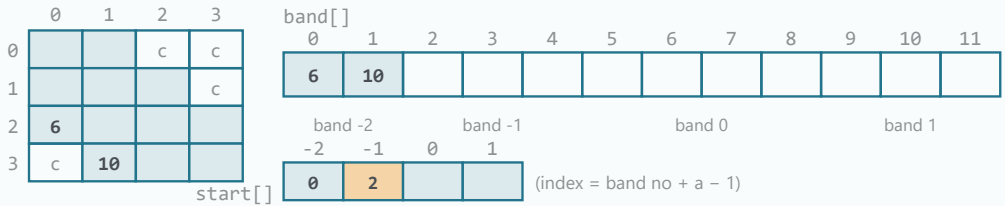
Step 7 / 12

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
▶    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

n = 4 a = 3 b = 2 i = -1 total = 2

Building a band Matrix (Lower Bands)



`start[-1+2] = start[1] = 2` : the band begins at `band[2]`.

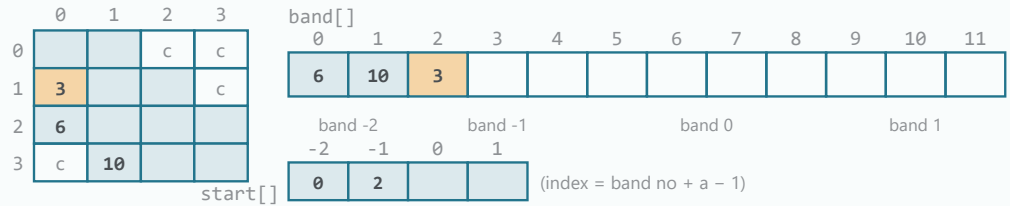
Step 8 / 12

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
▶      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = -1` `total = 2`

Building a band Matrix (Lower Bands)



`band[2+0]` ← 3 (d_{10}).

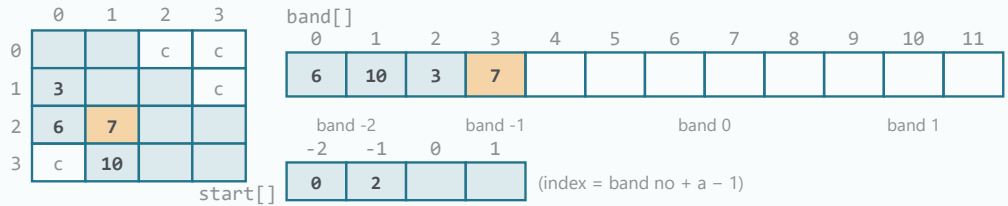
Step 9 / 12

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = -1` `k = 0` `total = 2`

Building a band Matrix (Lower Bands)



`band[2+1]` ← 7 (d_{21}).

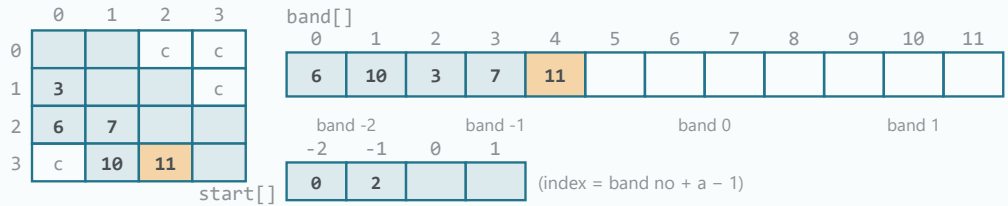
Step 10 / 12

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = -1` `k = 1` `total = 2`

Building a band Matrix (Lower Bands)



`band[2+2]` ← 11 (d_{32}).

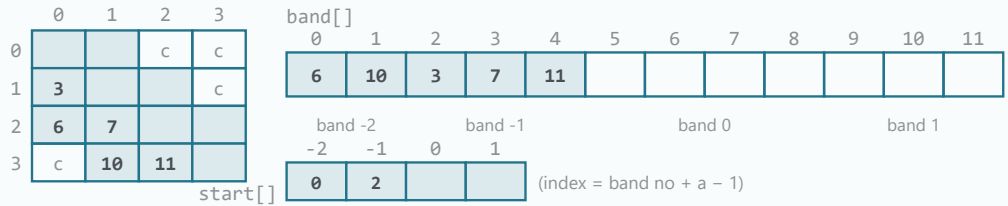
Step 11 / 12

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = -1` `k = 2` `total = 2`

Building a band Matrix (Lower Bands)



total += 3 → 5: where the next band begins.

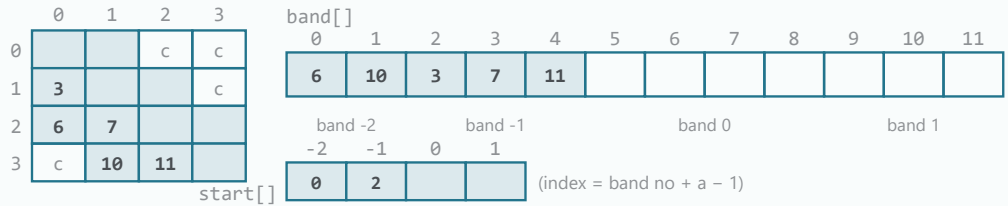
Step 12 / 12

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &band[start[i + a - 1] + k]);
▶      total += n - abs(i);
12   }
13 }
```

VARIABLES

n = 4 a = 3 b = 2 i = −1 total = 5

Building a band Matrix (Continued: Diagonal and Upper)



Band i = 0 (main diagonal): 4 elements.

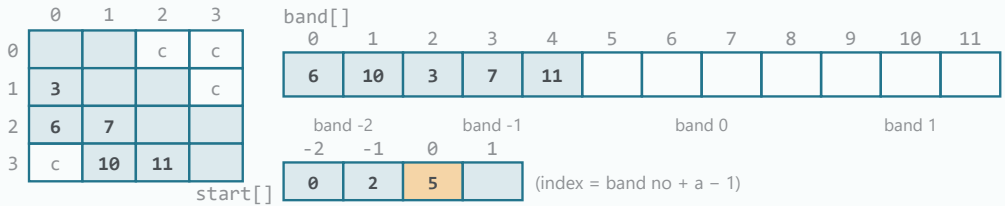
Step 1 / 13

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
▶    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

n = 4 a = 3 b = 2 i = 0 total = 5

Building a band Matrix (Continued: Diagonal and Upper)



`start[0+2] = start[2] = 5` : the band begins at `band[5]`.

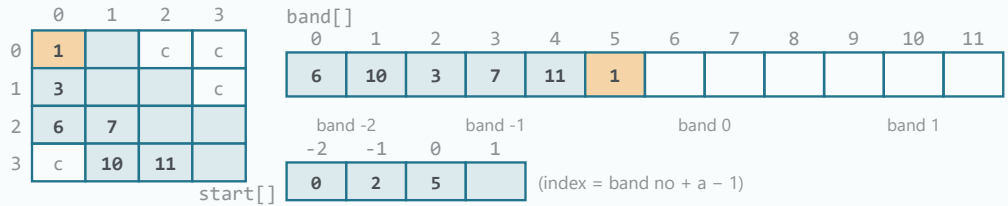
Step 2 / 13

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
▶      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = 0` `total = 5`

Building a band Matrix (Continued: Diagonal and Upper)



`band[5+0]` ← 1 (d_{00}).

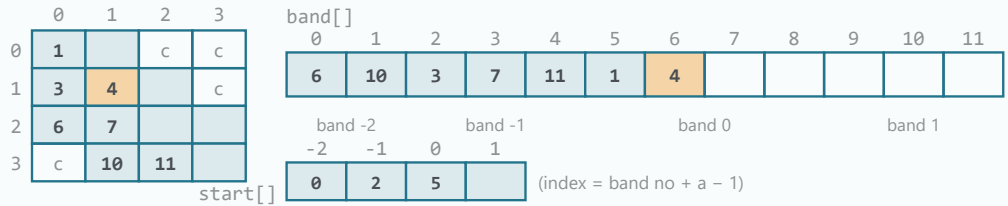
Step 3 / 13

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = 0` `k = 0` `total = 5`

Building a band Matrix (Continued: Diagonal and Upper)



`band[5+1]` ← 4 (d_{11}).

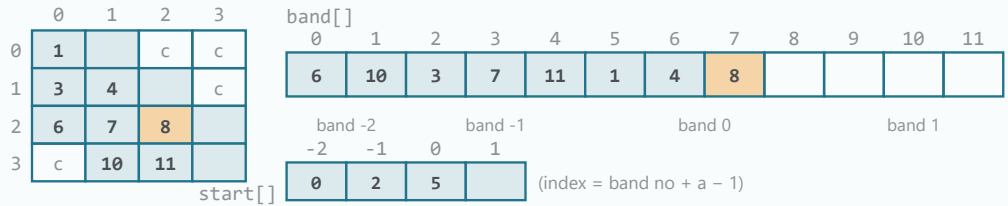
Step 4 / 13

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = 0` `k = 1` `total = 5`

Building a band Matrix (Continued: Diagonal and Upper)



`band[5+2]` ← 8 (d_{22}).

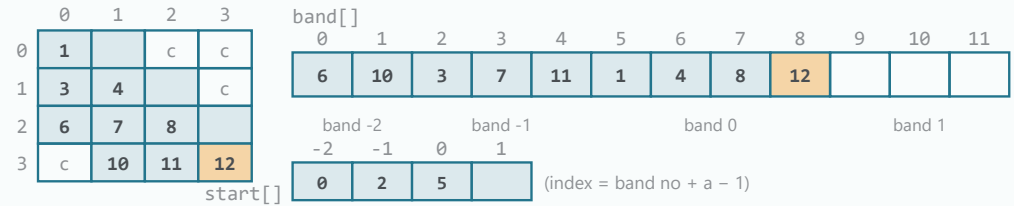
Step 5 / 13

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = 0` `k = 2` `total = 5`

Building a band Matrix (Continued: Diagonal and Upper)



`band[5+3]` ← 12 (d_{33}).

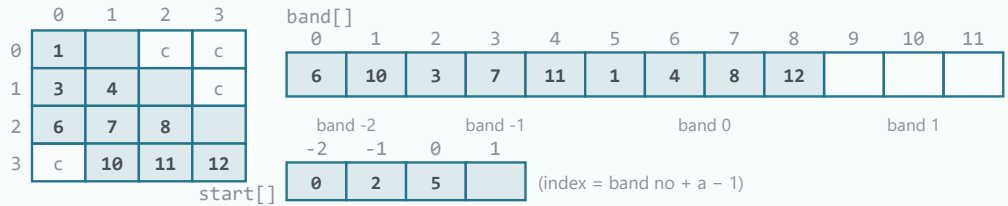
Step 6 / 13

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
       scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = 0` `k = 3` `total = 5`

Building a band Matrix (Continued: Diagonal and Upper)



total += 4 → 9: where the next band begins.

Step 7 / 13

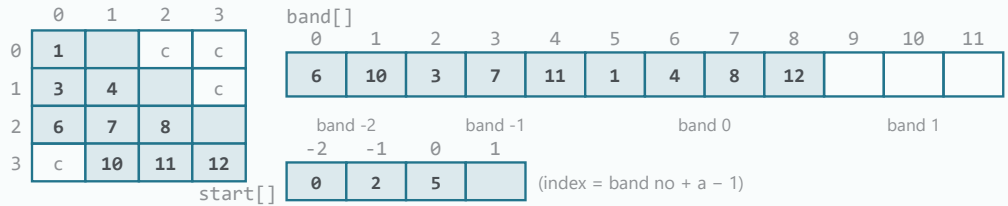
```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &band[start[i + a - 1] + k]);
▶      total += n - abs(i);
12   }
13 }
```

VARIABLES

n = 4 a = 3 b = 2 i = 0

total = 9

Building a band Matrix (Continued: Diagonal and Upper)



Band i = 1 (upper): 3 elements.

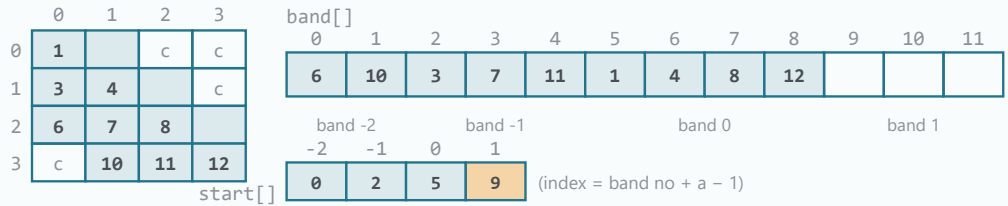
Step 8 / 13

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
▶    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

n = 4 a = 3 b = 2 i = 1 total = 9

Building a band Matrix (Continued: Diagonal and Upper)



`start[1+2] = start[3] = 9` : the band begins at `band[9]`.

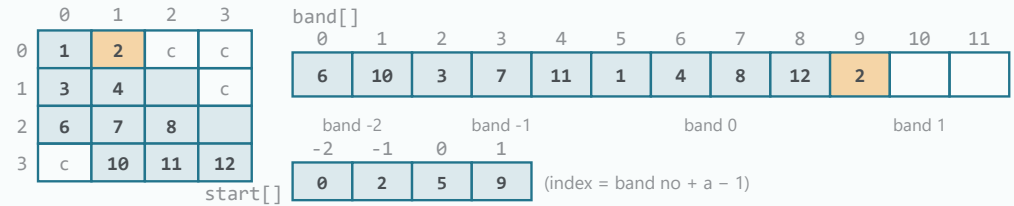
Step 9 / 13

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
▶      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = 1` `total = 9`

Building a band Matrix (Continued: Diagonal and Upper)



`band[9+0]` ← 2 (d_{01}).

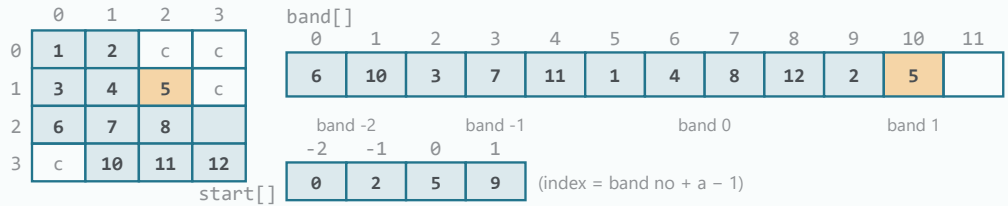
Step 10 / 13

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = 1` `k = 0` `total = 9`

Building a band Matrix (Continued: Diagonal and Upper)



`band[9+1]` ← 5 (d_{12}).

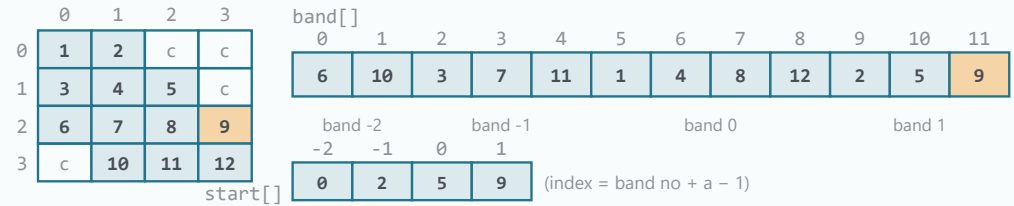
Step 11 / 13

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = 1` `k = 1` `total = 9`

Building a band Matrix (Continued: Diagonal and Upper)



`band[9+2]` ← 9 (d_{23}).

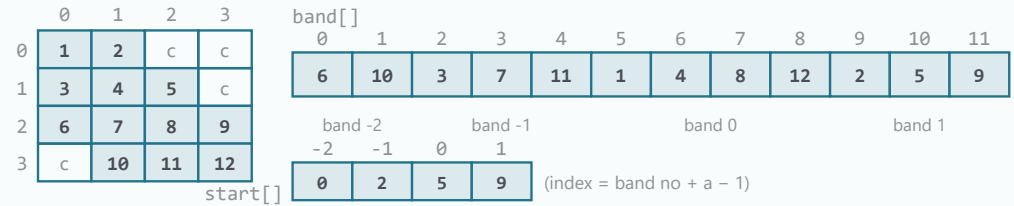
Step 12 / 13

```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &band[start[i + a - 1] + k]);
11     total += n - abs(i);
12   }
13 }
```

VARIABLES

`n = 4` `a = 3` `b = 2` `i = 1` `k = 2` `total = 9`

Building a band Matrix (Continued: Diagonal and Upper)



total += 3 → 12: where the next band begins.

Step 13 / 13

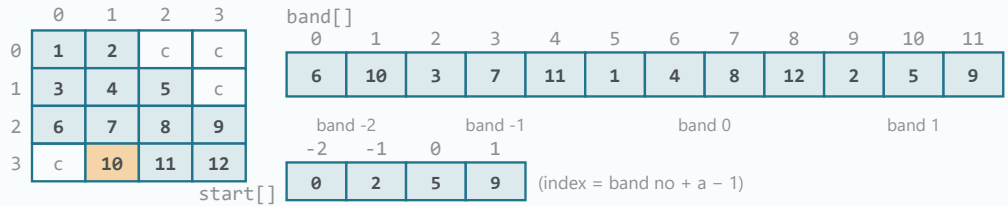
```
1  void build_band(int band[], int start[],
2                  int n, int a, int b) {
3    int i, k, total = 0;       /* earlier bands */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Out of memory!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* band by band */
8      start[i + a - 1] = total;       /* band start */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &band[start[i + a - 1] + k]);
▶      total += n - abs(i);
12   }
13 }
```

VARIABLES

n = 4 a = 3 b = 2 i = 1

total = 12

O(1) Access in a band Matrix



We ask for d_{31}; the indices are valid. Band number **j − i = -2**.

Step 1 / 12

```
1  int band_index(int i, int j, int n, int a,
2                 int b, int start[]) {
▶    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Index out of range\n"); return -3;
5    } else if (j - i >= 1) {          /* upper triangle */
6      if (j - i < b)                  /* valid band */
7        return start[j - i + a - 1] + i;
8      else return -1;                 /* upper constant */
9    } else if (i - j < a)             /* diagonal/lower */
10     return start[j - i + a - 1] + j;
11   else
12     return -2;                      /* lower constant */
13 }
```

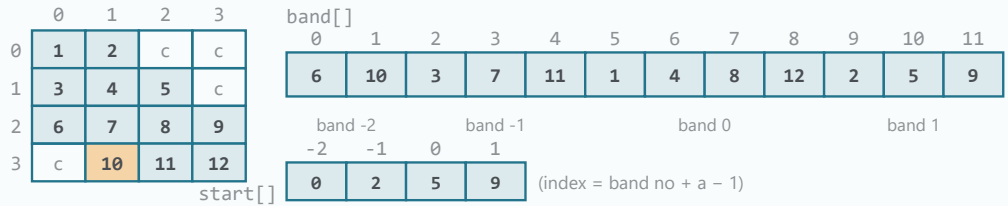
VARIABLES

`i = 3` `j = 1` `j−i = −2`

CALL STACK

`band_index(3, 1, 4, 3, 2, start)`

O(1) Access in a band Matrix



i − j = 2 < a = 3? Yes → a lower band.

Step 2 / 12

```
1  int band_index(int i, int j, int n, int a,
2                 int b, int start[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Index out of range\n"); return -3;
5    } else if (j - i >= 1) {          /* upper triangle */
6      if (j - i < b)                  /* valid band */
7        return start[j - i + a - 1] + i;
8      else return -1;                 /* upper constant */
▶    } else if (i - j < a)             /* diagonal/lower */
10     return start[j - i + a - 1] + j;
11   else
12     return -2;                      /* lower constant */
13 }
```

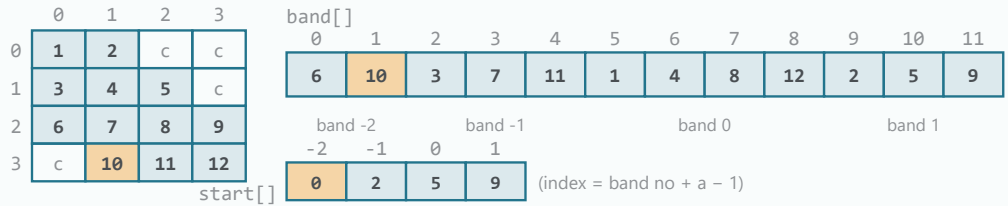
VARIABLES

`i = 3` `j = 1` `j−i = −2`

CALL STACK

`band_index(3, 1, 4, 3, 2, start)`

O(1) Access in a band Matrix



start[-2+2] + j = 0 + 1 = **1** → band[1] = 10.

Step 3 / 12

```
1  int band_index(int i, int j, int n, int a,
2                 int b, int start[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Index out of range\n"); return -3;
5    } else if (j - i >= 1) {          /* upper triangle */
6      if (j - i < b)                  /* valid band */
7        return start[j - i + a - 1] + i;
8      else return -1;                 /* upper constant */
9    } else if (i - j < a)             /* diagonal/lower */
▶      return start[j - i + a - 1] + j;
11   else
12     return -2;                      /* lower constant */
13 }
```

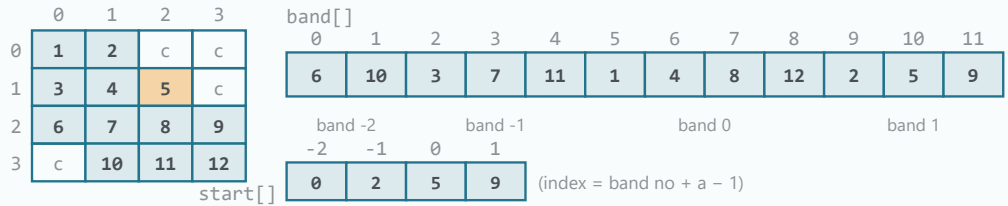
VARIABLES

`i = 3` `j = 1` `j-i = -2` `ret = 1`

CALL STACK

`band_index(3, 1, 4, 3, 2, start)`

O(1) Access in a band Matrix



We ask for d_{12}; the indices are valid. Band number **j − i = 1**.

Step 4 / 12

```
1  int band_index(int i, int j, int n, int a,
2                 int b, int start[]) {
▶    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Index out of range\n"); return -3;
5    } else if (j - i >= 1) {          /* upper triangle */
6      if (j - i < b)                  /* valid band */
7        return start[j - i + a - 1] + i;
8      else return -1;                 /* upper constant */
9    } else if (i - j < a)             /* diagonal/lower */
10     return start[j - i + a - 1] + j;
11   else
12     return -2;                      /* lower constant */
13 }
```

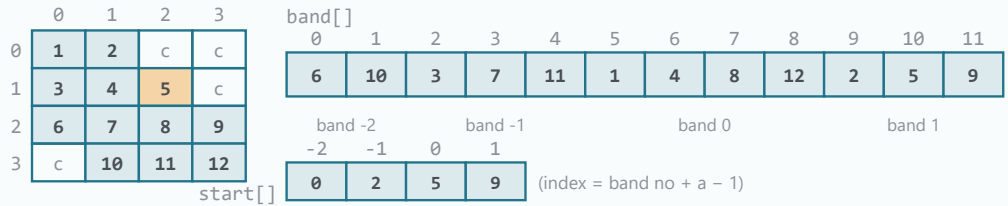
VARIABLES

`i = 1` `j = 2` `j-i = 1`

CALL STACK

`band_index(1, 2, 4, 3, 2, start)`

O(1) Access in a band Matrix



1 ≥ 1 → upper triangle.

Step 5 / 12

```
1  int band_index(int i, int j, int n, int a,
2                 int b, int start[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Index out of range\n"); return -3;
▶    } else if (j - i >= 1) {          /* upper triangle */
6      if (j - i < b)                  /* valid band */
7        return start[j - i + a - 1] + i;
8      else return -1;                 /* upper constant */
9    } else if (i - j < a)             /* diagonal/lower */
10     return start[j - i + a - 1] + j;
11   else
12     return -2;                      /* lower constant */
13 }
```

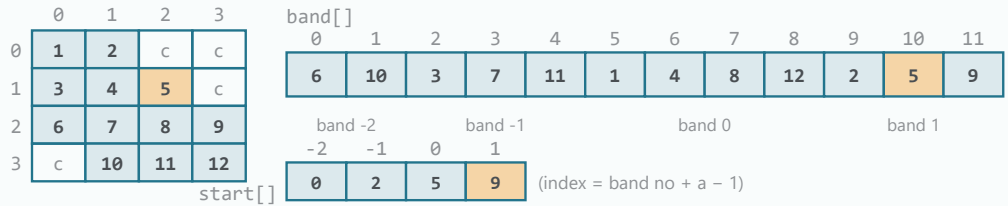
VARIABLES

`i = 1` `j = 2` `j−i = 1`

CALL STACK

`band_index(1, 2, 4, 3, 2, start)`

O(1) Access in a band Matrix



1 < b = 2 → a valid upper band: start[1+2] + i = 9 + 1 = **10**.

Step 6 / 12

```
1  int band_index(int i, int j, int n, int a,
2                 int b, int start[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Index out of range\n"); return -3;
5    } else if (j - i >= 1) {          /* upper triangle */
6      if (j - i < b)                  /* valid band */
▶        return start[j - i + a - 1] + i;
8      else return -1;                 /* upper constant */
9    } else if (i - j < a)             /* diagonal/lower */
10     return start[j - i + a - 1] + j;
11   else
12     return -2;                      /* lower constant */
13 }
```

VARIABLES

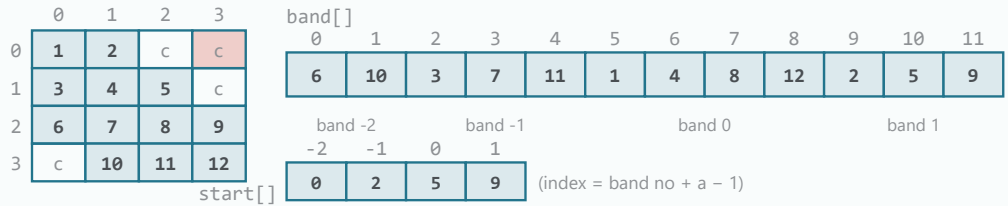
`i = 1` `j = 2` `j-i = 1`

`ret = 10`

CALL STACK

`band_index(1, 2, 4, 3, 2, start)`

O(1) Access in a band Matrix



We ask for d_{03}; the indices are valid. Band number **j − i = 3**.

Step 7 / 12

```
1  int band_index(int i, int j, int n, int a,
2                 int b, int start[]) {
▶    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Index out of range\n"); return -3;
5    } else if (j - i >= 1) {          /* upper triangle */
6      if (j - i < b)                  /* valid band */
7        return start[j - i + a - 1] + i;
8      else return -1;                 /* upper constant */
9    } else if (i - j < a)             /* diagonal/lower */
10     return start[j - i + a - 1] + j;
11   else
12     return -2;                      /* lower constant */
13 }
```

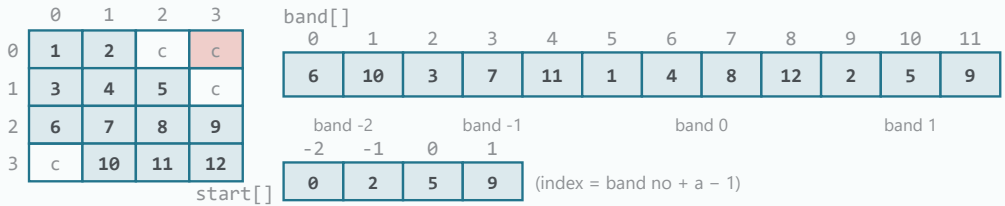
VARIABLES

`i = 0` `j = 3` `j−i = 3`

CALL STACK

`band_index(0, 3, 4, 3, 2, start)`

O(1) Access in a band Matrix



3 ≥ 1 → upper triangle.

Step 8 / 12

```
1  int band_index(int i, int j, int n, int a,
2                 int b, int start[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Index out of range\n"); return -3;
▶    } else if (j - i >= 1) {          /* upper triangle */
6      if (j - i < b)                  /* valid band */
7        return start[j - i + a - 1] + i;
8      else return -1;                 /* upper constant */
9    } else if (i - j < a)             /* diagonal/lower */
10     return start[j - i + a - 1] + j;
11   else
12     return -2;                      /* lower constant */
13 }
```

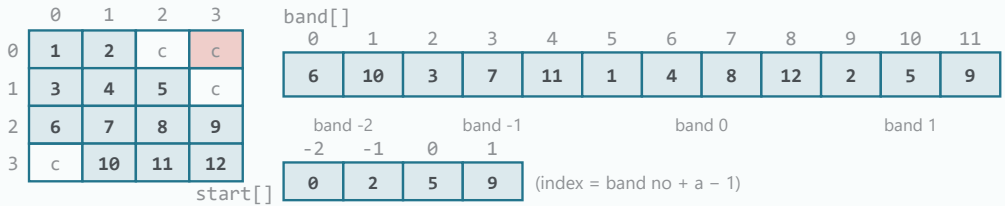
VARIABLES

`i = 0` `j = 3` `j−i = 3`

CALL STACK

`band_index(0, 3, 4, 3, 2, start)`

O(1) Access in a band Matrix



3 ≥ b = 2 → outside the band, upper constant region: **−1** (value c).

Step 9 / 12

```
1  int band_index(int i, int j, int n, int a,
2                 int b, int start[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Index out of range\n"); return -3;
5    } else if (j - i >= 1) {          /* upper triangle */
6      if (j - i < b)                  /* valid band */
7        return start[j - i + a - 1] + i;
▶      else return -1;                 /* upper constant */
9    } else if (i - j < a)             /* diagonal/lower */
10     return start[j - i + a - 1] + j;
11   else
12     return -2;                      /* lower constant */
13 }
```

VARIABLES

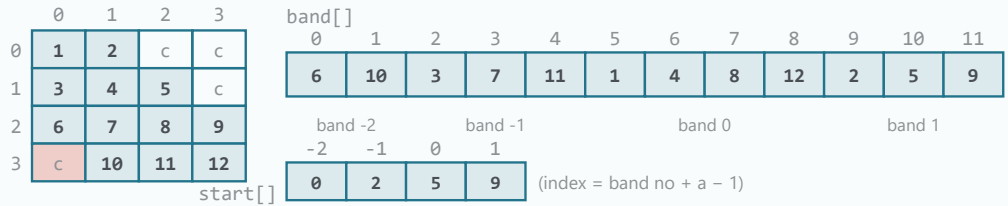
`i = 0` `j = 3` `j−i = 3`

`ret = −1`

CALL STACK

`band_index(0, 3, 4, 3, 2, start)`

O(1) Access in a band Matrix



We ask for d_{30}; the indices are valid. Band number **j − i = -3**.

Step 10 / 12

```
1  int band_index(int i, int j, int n, int a,
2                 int b, int start[]) {
▶   if (i < 0 || j < 0 || i >= n || j >= n) {
4     printf("Index out of range\n"); return -3;
5   } else if (j - i >= 1) {          /* upper triangle */
6     if (j - i < b)                  /* valid band */
7       return start[j - i + a - 1] + i;
8     else return -1;                 /* upper constant */
9   } else if (i - j < a)             /* diagonal/lower */
10    return start[j - i + a - 1] + j;
11  else
12    return -2;                      /* lower constant */
13 }
```

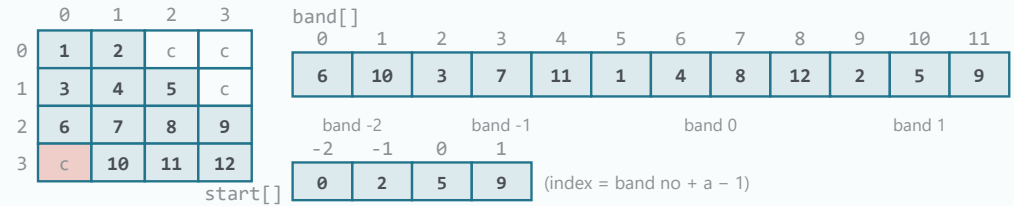
VARIABLES

`i = 3` `j = 0` `j-i = -3`

CALL STACK

`band_index(3, 0, 4, 3, 2, start)`

O(1) Access in a band Matrix



i − j = 3 < a = 3? No → outside the band.

Step 11 / 12

```
1  int band_index(int i, int j, int n, int a,
2                 int b, int start[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Index out of range\n"); return -3;
5    } else if (j - i >= 1) {          /* upper triangle */
6      if (j - i < b)                  /* valid band */
7        return start[j - i + a - 1] + i;
8      else return -1;                 /* upper constant */
▶    } else if (i - j < a)             /* diagonal/lower */
10     return start[j - i + a - 1] + j;
11   else
12     return -2;                      /* lower constant */
13 }
```

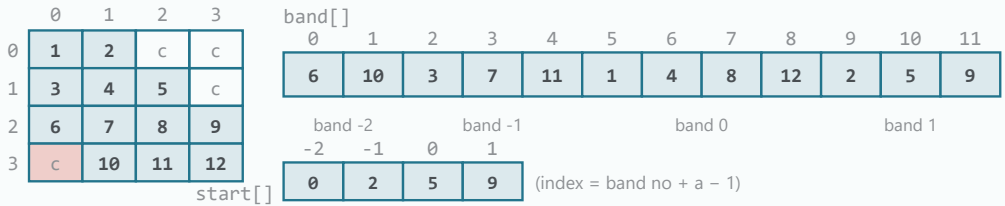
VARIABLES

`i = 3` `j = 0` `j−i = −3`

CALL STACK

`band_index(3, 0, 4, 3, 2, start)`

O(1) Access in a band Matrix



Lower constant region: **−2**. A constant number of operations in every case → **O(1)**.

Step 12 / 12

```
1  int band_index(int i, int j, int n, int a,
2                 int b, int start[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Index out of range\n"); return -3;
5    } else if (j - i >= 1) {          /* upper triangle */
6      if (j - i < b)                  /* valid band */
7        return start[j - i + a - 1] + i;
8      else return -1;                 /* upper constant */
9    } else if (i - j < a)             /* diagonal/lower */
10     return start[j - i + a - 1] + j;
11   else
▶      return -2;                      /* lower constant */
13 }
```

VARIABLES

`i = 3` `j = 0` `j−i = −3` `ret = −2`

CALL STACK

`band_index(3, 0, 4, 3, 2, start)`

A START Array or a Formula? (HOMEWORK 1)

`band_index` has no loop that depends on the input → **O(1)**. Counting band starts with a loop would raise the complexity; **START** avoids that but costs memory and makes the program "less dynamic".

> **Instructor** *"Extra memory is fine if it improves performance, within reason."*

HOMEWORK 1: Compute the band start without the START array. (A sample solution is on the right.)

```c
/* t: band number (1-a..b-1); same as start[t+a-1] */
int band_start(int t, int n, int a) {
  if (t <= 0)                       /* lower bands */
    return (t + a - 1) * n
           - (a * (a - 1) + t * (1 - t)) / 2;
  return (a - 1) * n - a * (a - 1) / 2  /* start of t=0 */
         + t * n - t * (t - 1) / 2;
}
```

n = 4, a = 3: t = −2, −1, 0, 1 → **0, 2, 5, 9** (same as START). A script tried every a, b for n ≤ 11. Access: `band_start(j-i, n, a) + (j > i ? i : j)`.

Is `if (expr1 && expr2)` Safe?

A common belief: "We can't know whether the compiler scans right to left or left to right, so the second of these two equivalent forms is safer."

```
/* 1 */ if (i < n && a[i] != 0) ...

/* 2 */ if (i < n)
          if (a[i] != 0) ...
```

In fact: the C standard guarantees **left-to-right** evaluation and **short-circuiting** for `&&` and `||` (C99 §6.5.13–14): if the left side is false, the right side is never evaluated. Both forms are **equally safe**; form 1 is a common and correct idiom.

The real trap

`&` (bitwise AND) does not short-circuit, and the order of evaluation of operands such as in `f(x) + g(x)` is **unspecified**.

struct: Fields of Different Types in One Record

```c
typedef struct {
  char name[10];
  int age;
  float salary;
} human;

human h1, h2;
strcpy(h1.name, "Ali");
h1.age = 21;
h1.salary = 3500.0f;
h2 = h1;          /* copied field by field */
/* if (h1 == h2) -> DOES NOT COMPILE */
```

Dot and arrow

`h1.age` is a field of a variable; `p->age` ≡ `(*p).age` (p: `human *`).

Assignment yes, == no

Structures can be assigned as a whole but not compared; write a function that compares field by field.

typedef

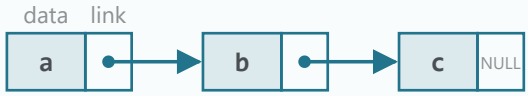
`typedef struct {…} human;` saves us from writing `struct` every time.

Structures Within Structures · Self-Referential Structures

```c
typedef struct {
  int day, month, year;
} date;

typedef struct {
  char name[10];
  date dob;              /* nested struct */
} person;
/* p.dob.year = 2004; */

typedef struct list {
  char data;
  struct list *link;     /* self-reference */
} list;
```



Self-referential structures are the basis of linked lists.

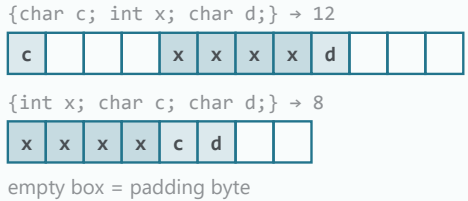
A structure **cannot contain** a field of its own type (infinite size), but it can contain a **pointer** to its own type.

Union: Fields Share the Same Memory

```
typedef struct {
  enum { FEMALE, MALE } sex;    /* tag */
  union {
    int children;               /* if FEMALE */
    int beard;                  /* if MALE */
  } u;
} sex_type;

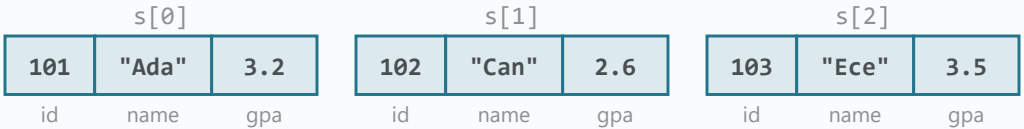
sex_type s;
s.sex = MALE;
s.u.beard = 0;    /* read only the field the tag allows */
```

A union is as large as its largest field (+ alignment). Tracking which field is valid with a **tag** field is the programmer's job.



Field order changes the memory layout (compiler-dependent; typical 32/64-bit values)

Walking Through an Array of Structures



An array of structures sits in memory record after record; each record is `sizeof(Student)` bytes.

Step 1 / 9

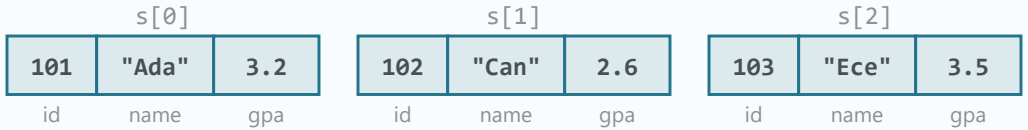
```
1  typedef struct {
2    int id;
3    char name[16];
4    double gpa;
5  } Student;
6
7  double class_avg(const Student s[], int n) {
▶    double sum = 0.0;
9    int i;
10   for (i = 0; i < n; i++)
11     sum += s[i].gpa;              /* field via . */
12   return n > 0 ? sum / n : 0.0;   /* empty class */
13 }
```

VARIABLES

`n = 3` `sum = 0.0`

`const Student s[]` : the array is not copied; `const` documents that the function will not modify the records.

Walking Through an Array of Structures



0 < 3 → record 0.

Step 2 / 9

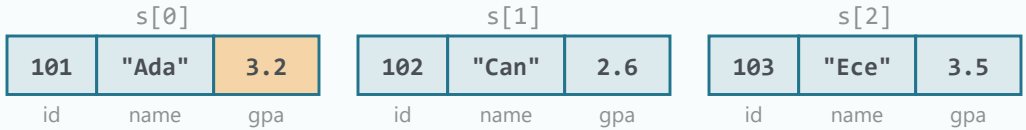
```
 1 typedef struct {
 2   int id;
 3   char name[16];
 4   double gpa;
 5 } Student;
 6
 7 double class_avg(const Student s[], int n) {
 8   double sum = 0.0;
 9   int i;
   ▶ for (i = 0; i < n; i++)
11     sum += s[i].gpa;               /* field via . */
12   return n > 0 ? sum / n : 0.0;    /* empty class */
13 }
```

VARIABLES

`n = 3` `sum = 0.0` `i = 0`

`const Student s[]` : the array is not copied; `const` documents that the function will not modify the records.

Walking Through an Array of Structures



Add `s[0].gpa` = 3.2 → sum = **3.2**.

Step 3 / 9

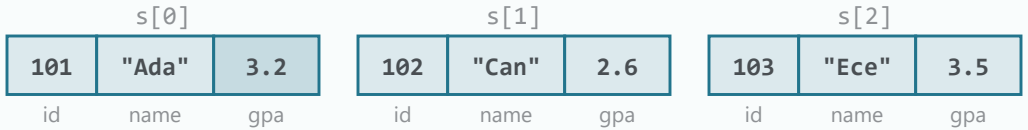
```
1  typedef struct {
2    int id;
3    char name[16];
4    double gpa;
5  } Student;
6
7  double class_avg(const Student s[], int n) {
8    double sum = 0.0;
9    int i;
10   for (i = 0; i < n; i++)
▶      sum += s[i].gpa;              /* field via . */
12   return n > 0 ? sum / n : 0.0;    /* empty class */
13 }
```

VARIABLES

`n = 3` `sum = 3.2` `i = 0`

`const Student s[]` : the array is not copied; `const` documents that the function will not modify the records.

Walking Through an Array of Structures



1 < 3 → record 1.

Step 4 / 9

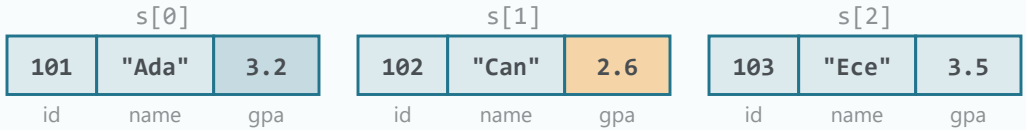
```
 1 typedef struct {
 2   int id;
 3   char name[16];
 4   double gpa;
 5 } Student;
 6
 7 double class_avg(const Student s[], int n) {
 8   double sum = 0.0;
 9   int i;
   ▶ for (i = 0; i < n; i++)
11     sum += s[i].gpa;              /* field via . */
12   return n > 0 ? sum / n : 0.0;    /* empty class */
13 }
```

VARIABLES

`n = 3` `sum = 3.2` `i = 1`

`const Student s[]` : the array is not copied; `const` documents that the function will not modify the records.

Walking Through an Array of Structures



Add `s[1].gpa` = 2.6 → sum = **5.8**.

Step 5 / 9

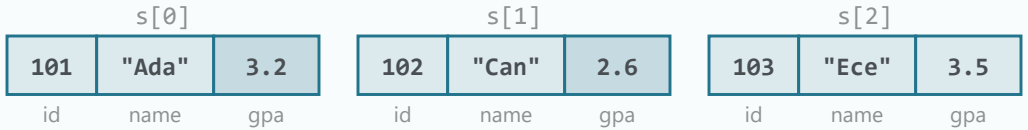
```
1  typedef struct {
2    int id;
3    char name[16];
4    double gpa;
5  } Student;
6
7  double class_avg(const Student s[], int n) {
8    double sum = 0.0;
9    int i;
10   for (i = 0; i < n; i++)
▶      sum += s[i].gpa;              /* field via . */
12   return n > 0 ? sum / n : 0.0;    /* empty class */
13 }
```

VARIABLES

`n = 3` `sum = 5.8` `i = 1`

`const Student s[]` : the array is not copied; `const` documents that the function will not modify the records.

Walking Through an Array of Structures



2 < 3 → record 2.

Step 6 / 9

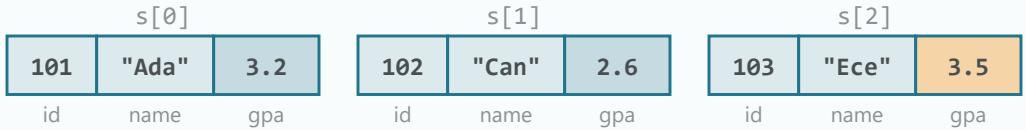
```
1  typedef struct {
2    int id;
3    char name[16];
4    double gpa;
5  } Student;
6
7  double class_avg(const Student s[], int n) {
8    double sum = 0.0;
9    int i;
▶    for (i = 0; i < n; i++)
11     sum += s[i].gpa;              /* field via . */
12   return n > 0 ? sum / n : 0.0;    /* empty class */
13 }
```

VARIABLES

`n = 3` `sum = 5.8` `i = 2`

`const Student s[]` : the array is not copied; `const` documents that the function will not modify the records.

Walking Through an Array of Structures



Add `s[2].gpa` = 3.5 → sum = **9.3**.

Step 7 / 9

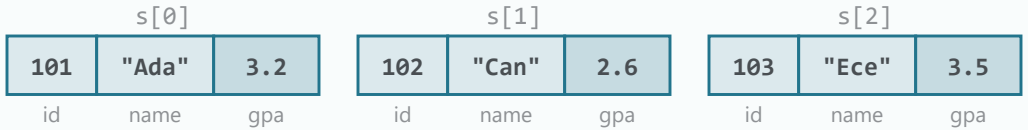
```
1  typedef struct {
2    int id;
3    char name[16];
4    double gpa;
5  } Student;
6
7  double class_avg(const Student s[], int n) {
8    double sum = 0.0;
9    int i;
10   for (i = 0; i < n; i++)
▶      sum += s[i].gpa;              /* field via . */
12   return n > 0 ? sum / n : 0.0;    /* empty class */
13 }
```

VARIABLES

`n = 3` `sum = 9.3` `i = 2`

`const Student s[]` : the array is not copied; `const` documents that the function will not modify the records.

Walking Through an Array of Structures



3 < 3 is false → the loop ends.

Step 8 / 9

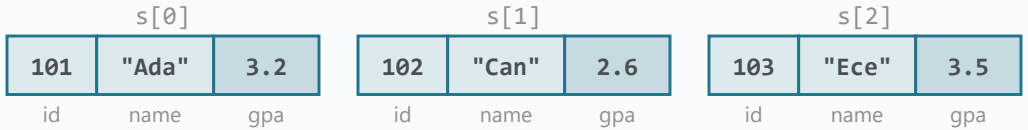
```
 1 typedef struct {
 2   int id;
 3   char name[16];
 4   double gpa;
 5 } Student;
 6
 7 double class_avg(const Student s[], int n) {
 8   double sum = 0.0;
 9   int i;
   ▶ for (i = 0; i < n; i++)
11     sum += s[i].gpa;              /* field via . */
12   return n > 0 ? sum / n : 0.0;    /* empty class */
13 }
```

VARIABLES

`n = 3` `sum = 9.3` `i = 3`

`const Student s[]` : the array is not copied; `const` documents that the function will not modify the records.

Walking Through an Array of Structures



n > 0 → return **9.3 / 3 = 3.10**. For n = 0 it returns 0.0 instead of dividing by zero.

Step 9 / 9

```
 1 typedef struct {
 2   int id;
 3   char name[16];
 4   double gpa;
 5 } Student;
 6
 7 double class_avg(const Student s[], int n) {
 8   double sum = 0.0;
 9   int i;
10   for (i = 0; i < n; i++)
11     sum += s[i].gpa;              /* field via . */
▶    return n > 0 ? sum / n : 0.0;    /* empty class */
13 }
```

VARIABLES

`n = 3` `sum = 9.3`

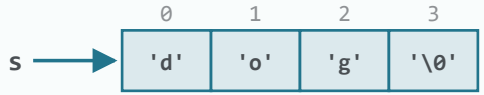
`const Student s[]` : the array is not copied; `const` documents that the function will not modify the records.

Strings in C: a Char Array Ending in `'\0'`

```c
char s[] = "dog";       /* 4 bytes: d o g \0 */
char t[10];
char *p = "dog";        /* read-only literal */

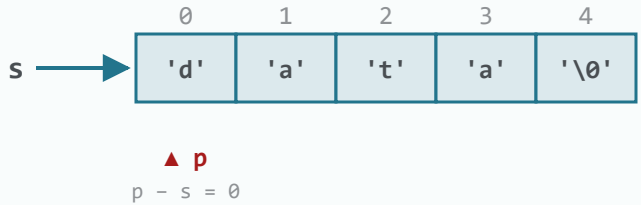
strlen(s)   /* 3 : up to \0 */
sizeof s    /* 4 : size of the array */

strcpy(t, s);                  /* is t big enough? */
snprintf(t, sizeof t, "%s!", s);   /* safe */
if (strcmp(s, "dog") == 0) ...     /* not ==! */
```



function	task
`strlen`	length (excluding \0)
`strcpy` / `strncpy`	copy
`strcat`	append
`strcmp`	compare lexicographically
`strchr` / `strstr`	find a char / substring

How Does strlen Work?



`p` points to the same place as `s` (the first character).

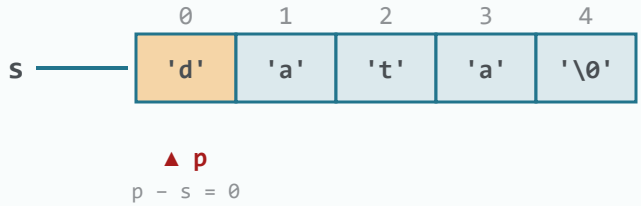
Step 1 / 11

```
1 size_t length(const char *s) {
▶   const char *p = s;
3   while (*p != '\0')      /* up to the terminator */
4     p++;
5   return (size_t)(p - s);  /* pointer difference */
6 }
7
8 /* length("data") */
```

VARIABLES

`s = &"data"[0]` `p = s`

How Does strlen Work?



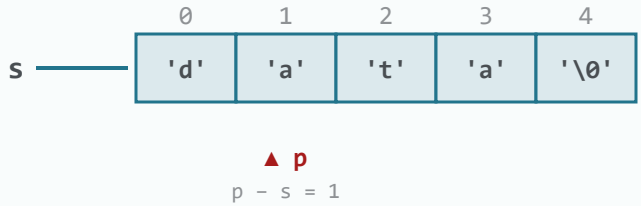
`*p = 'd'` ≠ '\0' → continue.


```
1 size_t length(const char *s) {
2   const char *p = s;
▶   while (*p != '\0')      /* up to the terminator */
4     p++;
5   return (size_t)(p - s);  /* pointer difference */
6 }
7
8 /* length("data") */
```

VARIABLES

`s = &"data"[0]` `p = s+0` `*p = 'd'`

How Does strlen Work?



p++: one character (1 byte) forward.

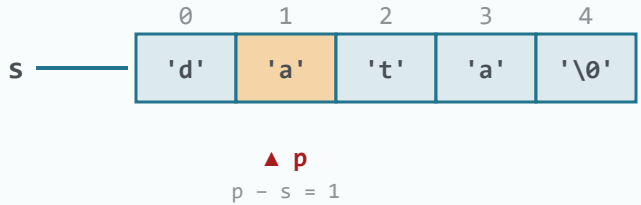
Step 3 / 11

```
1 size_t length(const char *s) {
2   const char *p = s;
3   while (*p != '\0')      /* up to the terminator */
▶     p++;
5   return (size_t)(p - s);  /* pointer difference */
6 }
7
8 /* length("data") */
```

VARIABLES

`s = &"data"[0]` `p = s+1`

How Does strlen Work?



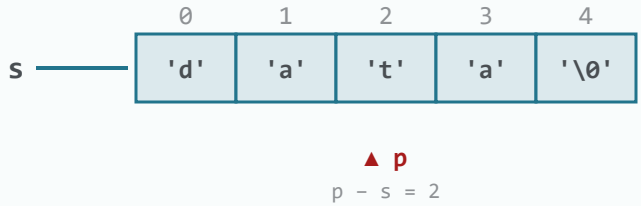
`*p = 'a'` ≠ '\0' → continue.


```
1 size_t length(const char *s) {
2   const char *p = s;
▶   while (*p != '\0')      /* up to the terminator */
4     p++;
5   return (size_t)(p - s);  /* pointer difference */
6 }
7
8 /* length("data") */
```

VARIABLES

`s = &"data"[0]` `p = s+1` `*p = 'a'`

How Does strlen Work?



p++: one character (1 byte) forward.

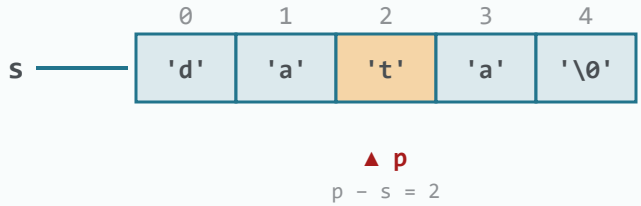
Step 5 / 11

```
1 size_t length(const char *s) {
2   const char *p = s;
3   while (*p != '\0')      /* up to the terminator */
▶     p++;
5   return (size_t)(p - s);  /* pointer difference */
6 }
7
8 /* length("data") */
```

VARIABLES

`s = &"data"[0]` `p = s+2`

How Does strlen Work?



`*p = 't'` ≠ '\0' → continue.

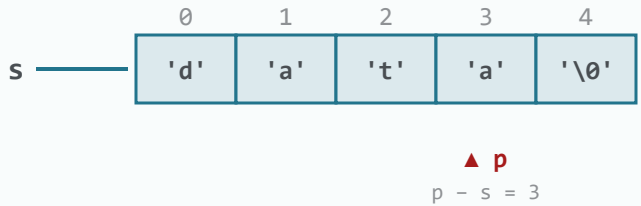
Step 6 / 11

```
1 size_t length(const char *s) {
2   const char *p = s;
▶   while (*p != '\0')      /* up to the terminator */
4     p++;
5   return (size_t)(p - s);  /* pointer difference */
6 }
7
8 /* length("data") */
```

VARIABLES

`s = &"data"[0]` `p = s+2` `*p = 't'`

How Does strlen Work?



p++: one character (1 byte) forward.

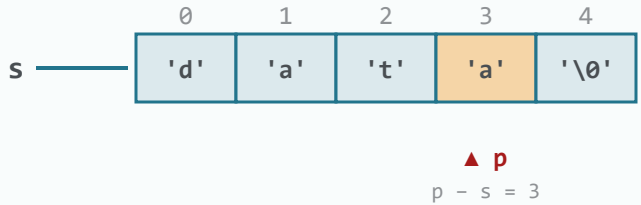
Step 7 / 11

```
1 size_t length(const char *s) {
2   const char *p = s;
3   while (*p != '\0')      /* up to the terminator */
▶     p++;
5   return (size_t)(p - s);  /* pointer difference */
6 }
7
8 /* length("data") */
```

VARIABLES

`s = &"data"[0]` `p = s+3`

How Does strlen Work?



`*p = 'a'` ≠ '\0' → continue.

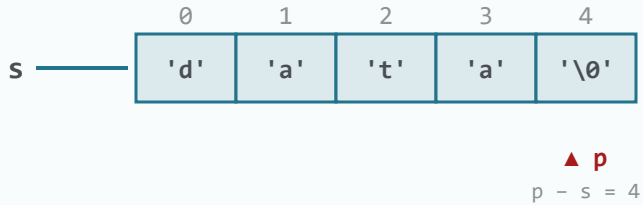
Step 8 / 11

```
1 size_t length(const char *s) {
2   const char *p = s;
▶   while (*p != '\0')      /* up to the terminator */
4     p++;
5   return (size_t)(p - s);  /* pointer difference */
6 }
7
8 /* length("data") */
```

VARIABLES

`s = &"data"[0]` `p = s+3` `*p = 'a'`

How Does strlen Work?



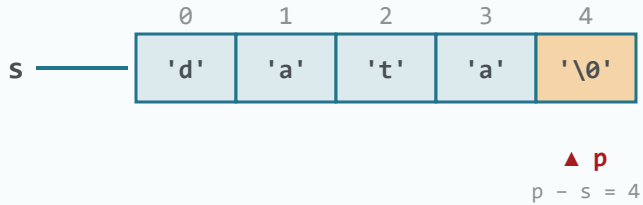
p++: one character (1 byte) forward.


```
1 size_t length(const char *s) {
2   const char *p = s;
3   while (*p != '\0')      /* up to the terminator */
▶     p++;
5   return (size_t)(p - s);  /* pointer difference */
6 }
7
8 /* length("data") */
```

VARIABLES

`s = &"data"[0]` `p = s+4`

How Does strlen Work?



`*p = '\0'` → the loop ends.

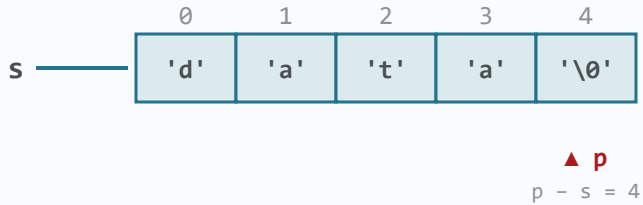
Step 10 / 11

```
1 size_t length(const char *s) {
2   const char *p = s;
▶   while (*p != '\0')      /* up to the terminator */
4     p++;
5   return (size_t)(p - s);  /* pointer difference */
6 }
7
8 /* length("data") */
```

VARIABLES

`s = &"data"[0]` `p = s+4` `*p = '\0'`

How Does strlen Work?



Return **p − s = 4**. Finding the length scans the whole string: **O(n)** — calling `strlen` in a loop condition makes it O(n²)!

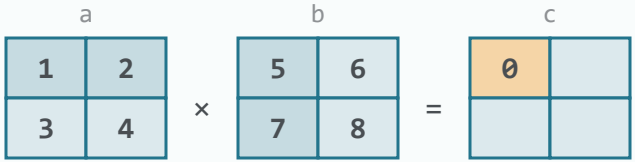
Step 11 / 11

```
1 size_t length(const char *s) {
2   const char *p = s;
3   while (*p != '\0')      /* up to the terminator */
4     p++;
▶   return (size_t)(p - s);  /* pointer difference */
6 }
7
8 /* length("data") */
```

VARIABLES

`s = &"data"[0]` `p = s+4`

Matrix Multiplication: Three Nested Loops



c[0][0] uses **row 0** of a and **column 0** of b; start at 0.

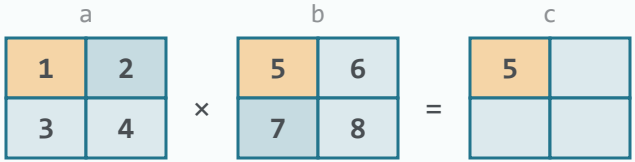
Step 1 / 13

```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
▶        c[i][j] = 0;
7        for (k = 0; k < N; k++)
8          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

VARIABLES

`i = 0` `j = 0` `c[0][0] = 0`

Matrix Multiplication: Three Nested Loops



k = 0: += a[0][0]·b[0][0] = 1·5 → **5**.

Step 2 / 13

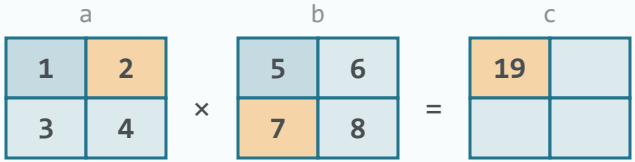
```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

VARIABLES

`i = 0` `j = 0` `k = 0`

`c[0][0] = 5`

Matrix Multiplication: Three Nested Loops



k = 1: += a[0][1]·b[1][0] = 2·7 → **19**.

Step 3 / 13

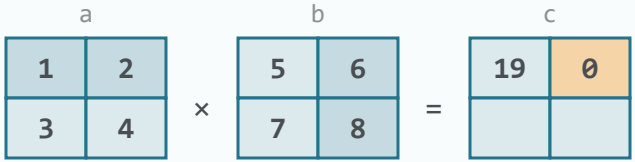
```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

VARIABLES

`i = 0` `j = 0` `k = 1`

`c[0][0] = 19`

Matrix Multiplication: Three Nested Loops



c[0][1] uses **row 0** of a and **column 1** of b; start at 0.

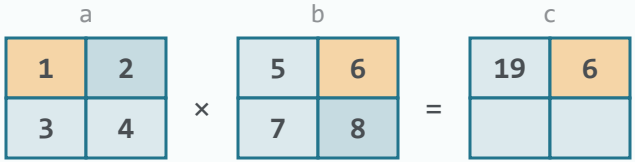
Step 4 / 13

```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
▶        c[i][j] = 0;
7        for (k = 0; k < N; k++)
8          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

VARIABLES

`i = 0` `j = 1` `c[0][1] = 0`

Matrix Multiplication: Three Nested Loops



k = 0: += a[0][0]·b[0][1] = 1·6 → **6**.

Step 5 / 13

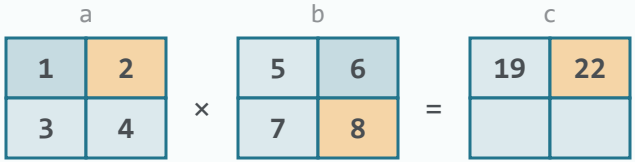
```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

VARIABLES

`i = 0` `j = 1` `k = 0`

`c[0][1] = 6`

Matrix Multiplication: Three Nested Loops



k = 1: += a[0][1]·b[1][1] = 2·8 → **22**.

Step 6 / 13

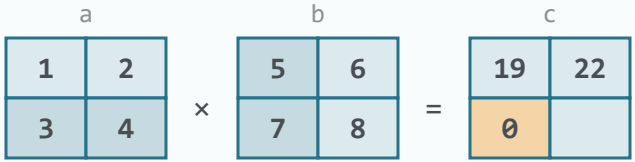
```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

VARIABLES

`i = 0` `j = 1` `k = 1`

`c[0][1] = 22`

Matrix Multiplication: Three Nested Loops



c[1][0] uses **row 1** of a and **column 0** of b; start at 0.

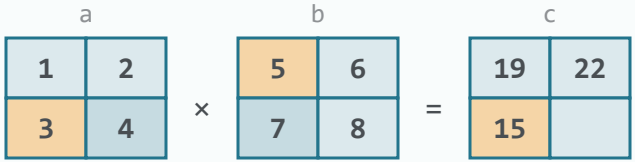
Step 7 / 13

```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
▶        c[i][j] = 0;
7        for (k = 0; k < N; k++)
8          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

VARIABLES

`i = 1` `j = 0` `c[1][0] = 0`

Matrix Multiplication: Three Nested Loops



k = 0: += a[1][0]·b[0][0] = 3·5 → **15**.

Step 8 / 13

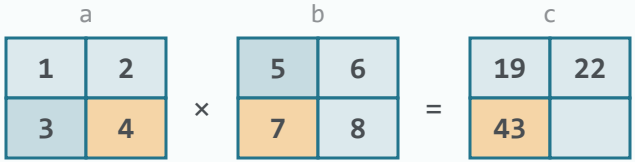
```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

VARIABLES

`i = 1` `j = 0` `k = 0`

`c[1][0] = 15`

Matrix Multiplication: Three Nested Loops



k = 1: += a[1][1]·b[1][0] = 4·7 → **43**.

Step 9 / 13

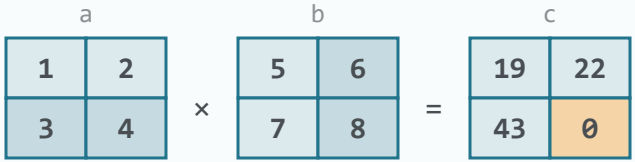
```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

VARIABLES

`i = 1` `j = 0` `k = 1`

`c[1][0] = 43`

Matrix Multiplication: Three Nested Loops



c[1][1] uses **row 1** of a and **column 1** of b; start at 0.

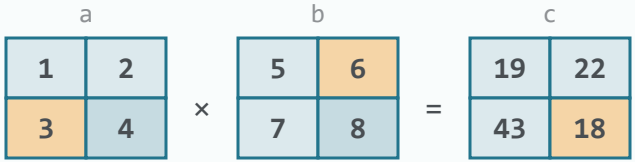
Step 10 / 13

```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
▶        c[i][j] = 0;
7        for (k = 0; k < N; k++)
8          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

VARIABLES

`i = 1` `j = 1` `c[1][1] = 0`

Matrix Multiplication: Three Nested Loops



k = 0: += a[1][0]·b[0][1] = 3·6 → **18**.

Step 11 / 13

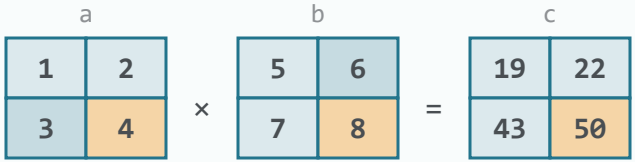
```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

VARIABLES

`i = 1` `j = 1` `k = 0`

`c[1][1] = 18`

Matrix Multiplication: Three Nested Loops



k = 1: += a[1][1]·b[1][1] = 4·8 → **50**.

Step 12 / 13

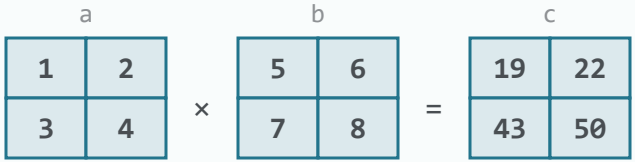
```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

VARIABLES

`i = 1` `j = 1` `k = 1`

`c[1][1] = 50`

Matrix Multiplication: Three Nested Loops



Done: [[19, 22], [43, 50]]. The innermost statement ran **$N^3 = 8$** times → **$O(n^3)$**.

Step 13 / 13

```
1 #define N 2
2 void mult(int a[][N], int b[][N], int c[][N]) {
3   int i, j, k;
4   for (i = 0; i < N; i++)
5     for (j = 0; j < N; j++) {
6       c[i][j] = 0;
7       for (k = 0; k < N; k++)
8         c[i][j] += a[i][k] * b[k][j];
9     }
▶ }
```

VARIABLES

—

Homework: Matrix Multiplication and Transpose

1. Build the step count table for `mult` (n×n); how many multiplications are performed?
2. What is the complexity of multiplying an m×n matrix by an n×p matrix? How many cells does the result have?
3. Write a function that transposes an n×n matrix **without an extra array** (in place).

▶ **Solutions**

Lab: Dynamic Array

Appending 7, 3, 9, 4, 6

Input (comma-separated numbers) 7, 3, 9, 4, 6 Apply

CODE

```
 1 typedef struct {
 2   int *data;  int size, cap;
 3 } DynArray;
 4
 5 int append(DynArray *d, int x) {
 6   if (d->size == d->cap) {       /* full */
 7     int newcap = d->cap ? 2 * d->cap : 1;
 8     int *t = realloc(d->data,
 9                      newcap * sizeof *t);
10     if (t == NULL) return 0;
11     d->data = t;
12     d->cap = newcap;
13   }
14   d->data[d->size++] = x;
```

data

x – size 0 cap 0 copies 0

OUTPUT (none yet)

An empty dynamic array: `{data = NULL, size = 0, cap = 0}`. When it is full, the capacity doubles; on every resize the old items are moved to the new block.

Complexity Summary

Operation	Time	Extra space
Access A[i]; n-dimensional address (n−1 multiplications)	O(1)	O(1)
Append to a dynamic array (doubling)	amortized O(1), worst O(n)	≤ 2·size
Insert / delete in the middle	O(n)	O(1)
Lower triangular / band matrix access	O(1)	n(n+1)/2 / band elements
build_band, read_lower	O(stored elements)	START: a+b−1
strlen / length	O(n)	O(1)
Matrix addition (m×n)	O(mn)	O(1)
Matrix multiplication (n×n)	$O(n^3)$	O(1)

WRAP-UP

Common Mistakes

1. Writing past the end of an array (`a[n]`); C does not check.
2. `a = realloc(a, …)`: losing the old block on failure.
3. Mixing up `sizeof(int *)` and `sizeof(int)`; using `sizeof a` on a parameter.
4. Freeing only the pointer array of a 2D dynamic array, not its rows.
5. Mixing up the row- and column-major formulas; forgetting the element size (s).
6. Using `==` on strings, or leaving no room for `'\0'`.

Exercises

1. `double d[5][8]` row-major, α = 2000: `&d[3][5]`? Column-major?
2. `int A[4][5][6]`: relative position of A[2][3][4] by Horner's method?
3. 5×5 lower triangular matrix: where in LOW is a_{43}?
4. Band matrix n = 6, a = 2, b = 3: how many elements? START array?
5. A dynamic array starts at capacity 1 and doubles. Total copies after 100 appends?
6. How can reordering the fields shrink `struct { char a; double b; char c; }`?

▶ **Answers**

WRAP-UP

Chapter Summary

Array = mapping

ADT: ⟨index, value⟩; the C implementation is contiguous memory with address $\alpha + i \cdot s$.

Addressing

Row major: $\alpha + (i \cdot u_1 + j) \cdot s$; in n dimensions, $n-1$ multiplications with Horner.

Saving space

Triangle: $n(n+1)/2$ elements, $k = i(i+1)/2 + j$. Band: bands numbered $j - i + \text{START}$.

Dynamic memory

malloc/realloc/free; growth by doubling is amortized O(1); a 2D array = an array of pointers.

Records

A struct groups fields, a union shares them; a string is a char array ending in '\0'.

Matrices

Addition $O(mn)$, multiplication $O(n^3)$.

References and Next Steps

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Chapter 2 (arrays, structures, multidimensional arrays, strings) and 1.4.
- B. Kernighan, D. Ritchie, *The C Programming Language*, 2nd ed., Chapters 5–6 (pointers, arrays, structures).
- T. Cormen et al., *Introduction to Algorithms*, 17.4 (dynamic tables and amortized analysis).

← Chapter 1: Introduction and Algorithm Analysis

Course page

Chapter 3: Recursion, Binary Search and Backtracking

The End

Chapter 2: Arrays and Structures

Next chapter: Chapter 3: Recursion, Binary Search and Backtracking · Course page