

GATE CS: Arrays and Pointers in C — Bilingual Worked Lessons

C में ऐरे और पॉइंटर · द्विभाषी हल किए हुए पाठ

Six complete lessons, each with two worked examples and four practice exercises with full solutions. English and Hindi are kept together by lesson, with each language readable on its own.

Coverage and convention

Mapped to the GATE 2027 CS syllabus; checked 2 October 2026. This workbook covers introductory fixed-size arrays, scans, pointer targets, array parameters, pointer movement and two-dimensional arrays. It does not complete C, Programming and Data Structures, or the GATE CS syllabus. Recursion and later data structures belong to separate study.

C11 is the explicit teaching convention, not a claimed GATE-mandated language version. Follow the assumptions and safety boundaries stated in each lesson. Deliberately unsafe fragments are for classification only, not execution.

यह कार्यपुस्तिका ऐरे और पॉइंटर की सीमित शुरुआती पढ़ाई है; यह पूरा C, प्रोग्रामिंग और डेटा स्ट्रक्चर या GATE CS पाठ्यक्रम नहीं है। C11 यहाँ शिक्षण की स्पष्ट परंपरा है, GATE की बताई हुई अनिवार्य भाषा-शैली नहीं। हर पाठ की धारणाओं और सुरक्षा सीमाओं का पालन करें। जानबूझकर असुरक्षित अंश केवल वर्गीकरण के लिए हैं, चलाने के लिए नहीं।

How to use this workbook

Start with the lesson's prerequisites. Predict the state and output before reading each trace. Try the four unscored exercises, then use the linked solutions and quick reference to check your reasoning. The contents and PDF bookmarks jump to either language and to practice or solutions. The separate integrated review is not included here.

Syllabus context: [Official GATE 2027 CS syllabus, Section 4](#). Each lesson retains its original C semantic source note. Existing 2026 resources retain their separate identity. This edition republishes the accepted original lessons without adding teaching or assessments.

Contents

Choose a language, or jump straight to practice, full solutions or the quick reference.

01 Array elements and valid indices

[English 3](#) · [Practice 6](#) · [Solutions 8](#) · [Quick reference 9](#)

[हिन्दी 11](#) · [अभ्यास 14](#) · [हल 16](#) · [त्वरित संदर्भ 17](#)

02 Array scans and what the accumulator means

[English 19](#) · [Practice 22](#) · [Solutions 25](#) · [Quick reference 26](#)

[हिन्दी 28](#) · [अभ्यास 31](#) · [हल 33](#) · [त्वरित संदर्भ 35](#)

03 Pointer targets and shared objects

[English 36](#) · [Practice 40](#) · [Solutions 42](#) · [Quick reference 45](#)

[हिन्दी 47](#) · [अभ्यास 51](#) · [हल 53](#) · [त्वरित संदर्भ 56](#)

04 Array parameters and caller-visible changes

[English 58](#) · [Practice 62](#) · [Solutions 64](#) · [Quick reference 66](#)

[हिन्दी 68](#) · [अभ्यास 72](#) · [हल 74](#) · [त्वरित संदर्भ 76](#)

05 Pointer movement and array boundaries

[English 78](#) · [Practice 83](#) · [Solutions 85](#) · [Quick reference 87](#)

[हिन्दी 89](#) · [अभ्यास 94](#) · [हल 96](#) · [त्वरित संदर्भ 98](#)

06 Rows and columns in two-dimensional arrays

[English 100](#) · [Practice 105](#) · [Solutions 108](#) · [Quick reference 110](#)

[हिन्दी 112](#) · [अभ्यास 117](#) · [हल 120](#) · [त्वरित संदर्भ 122](#)

Lesson 01 · English

Array elements and valid indices

Which values change

You know how to record the current value of an integer variable. Now suppose four related values belong together. We can store them in an array and still trace every assignment one step at a time.

In this lesson you will identify an array's valid indices, read its initial values, and explain which element an assignment changes. You will also distinguish a saved scalar copy from an array element, and explain the zeros supplied by a partial initializer.

Prerequisites: the foundation lessons on variables, current-state tracing, simple arithmetic, conditions and copied values. No pointer, loop, string or memory-address knowledge is required here.

We continue to use C11 as our teaching convention. Our arrays contain ordinary signed `int` elements and have explicit positive constant sizes. Every value read in an executable example has been initialized, and every computed integer fits within -32767 through 32767 . We do not assume that an `int` occupies four bytes. Every complete program starts afresh and can be compiled separately. One deliberately unsafe fragment is clearly marked for classification only.

One array name and several element positions

Read `int scores[4] = {6, 2, 9, 4};` in two parts. `scores[4]` in this declaration creates an array with four integer elements. The initializer in braces supplies their starting values in order.

Index	0	1	2	3
Initial value	6	2	9	4

An index identifies a position. The first position has index 0, so `scores[0]` supplies 6 when it is read, and `scores[1]` supplies 2. The fourth and last element is `scores[3]`. The declared count is 4; it is not a valid index for this array.

Keep a position and its stored value separate. The value 9 is stored at index 2. That does not give this array an element at index 9. An element may contain a negative number or a number larger than the array's length; the stored number does not change the valid positions.

The same square brackets have different jobs in a declaration and in a later expression. The declaration `int scores[4]` states the element count. A later read of `scores[4]` would try to access a fifth element, which does not exist in this array.

For an array with N elements and an integer index i , an element access needs both $i \geq 0$ and $i < N$. Here N means the element count in our explanation; our actual declarations use positive constants such as 4 and 5. The last valid index is $N - 1$. Neither a negative index nor an index equal to the count is valid for these accesses.

Apply the familiar assignment rule to one element

In `scores[1] = scores[0] + scores[2];`, the right side reads two current element values. Their sum is stored in the element at index 1. This statement does not assign to every element.

Use a record with an entry for each position:

1. Work out the index of every accessed element and check its bounds
2. Read the current values used by the right-hand calculation
3. Calculate the result
4. Write it to the selected destination element and carry all other entries forward

Our statements keep index changes separate from element updates. We do not combine an index increment and another use of that index inside one expression. This carries forward the foundation lesson's sequencing discipline.

Worked example 1 A saved value and two element updates

Predict the complete output before reading the trace. Does `saved` change when `scores[1]` changes?

```
#include <stdio.h>

int main(void)
{
    int scores[4] = {6, 2, 9, 4};
    int saved = scores[1];
    scores[1] = scores[0] + scores[2];
    scores[0] = saved - 1;
    printf("%d %d %d %d | %d\n",
           scores[0], scores[1], scores[2], scores[3], saved);
    return 0;
}
```

- After initialization: `scores = {6, 2, 9, 4}`
- `saved = scores[1]` reads the current value 2. Now `saved = 2`; the array is unchanged
- `scores[1] = scores[0] + scores[2]` computes `6 + 9 = 15`. Now `scores = {6, 15, 9, 4}` and `saved = 2`
- `scores[0] = saved - 1` computes `2 - 1 = 1`. Now `scores = {1, 15, 9, 4}` and `saved = 2`
- The printed argument order is the four elements, then `saved`. The output is `1 15 9 4 | 2`, followed by a newline. The `|` is just a printed separator

All accessed indices are 0 through 3. Each assignment replaces one element's stored value; the other positions keep their values.

saved holds a copied integer. It is not another name for element 1 and does not keep following it. If you predicted 14 15 9 4 | 15, you treated that saved copy as a continuing connection. If you left the first element at 6, you skipped the second element assignment. If you put 15 into every position, you treated one selected element as the whole array.

Partial initialization supplies the remaining zeros

A positive fixed-size integer array can have fewer initializer values than elements. In `int bins[5] = {4, -1};`, the two supplied numbers initialize positions 0 and 1. The remaining positions 2, 3 and 4 start at zero.

Index	0	1	2	3	4
Initial value	4	-1	0	0	0

The size is still five. An omitted initializer does not remove an element, repeat the last written value, or leave that remaining element waiting for input. `int ready[3] = {0};` therefore starts all three elements at zero: the first is explicitly initialized to zero and the other two receive zero implicitly.

Do not transfer that rule to `int bins[5];` inside `main`. That declaration has no initializer. Its automatic elements do not receive guaranteed zeros. Assign known values before reading them; do not run an uninitialized read to decide what C “usually” provides. Static-storage arrays have different initialization rules and are outside this lesson.

These initializer lists set starting values once when the declarations are reached. A later assignment does not restart the initialization or refill the other elements with zeros.

Worked example 2 Use an implicitly initialized element

This example combines supplied values and implicit zeros. Work from the state left by each statement.

```
#include <stdio.h>

int main(void)
{
    int bins[5] = {4, -1};
    bins[3] = bins[0] + bins[2];
    bins[2] = bins[1] - bins[4];
    printf("%d %d %d %d %d\n",
           bins[0], bins[1], bins[2], bins[3], bins[4]);
    return 0;
}
```

- The declaration gives `bins = {4, -1, 0, 0, 0}`
- For `bins[3] = bins[0] + bins[2]`, the operands are 4 and 0. Store 4 at index 3: `{4, -1, 0, 4, 0}`
- For `bins[2] = bins[1] - bins[4]`, the operands are -1 and 0. Store -1 at index 2: `{4, -1, -1, 4, 0}`

- Output: 4 -1 -1 4 0, followed by a newline

The read of `bins[2]` in the first assignment is valid: its implicit initial value is zero. The later assignment changes that same element to `-1`. The implicit zero is a starting value, not a permanent constraint.

Five copies of 4, or three trailing copies of `-1`, would misread the initializer list. Claiming that `bins[4]` is uninitialized would miss the rule for the remaining elements of this initialized array. Reading `bins[5]` would still be invalid; partial initialization creates no extra element beyond the declared size.

An index can be calculated

The brackets can contain an integer expression. With a valid value of `k`, `scores[k]` selects the position given by the current `k`, while `scores[k - 1]` selects the previous position. Check each resulting index separately. A valid `k` does not automatically make `k - 1` or `k + 1` valid.

For the four-element `scores`, `k = 0` is valid for `scores[k]` but not for `scores[k - 1]`. At `k = 3`, `scores[k]` is valid but `scores[k + 1]` is not. Updating an index variable later does not undo a past array update. It changes which position a subsequent expression selects.

Practice before opening the solutions

These four original learning exercises are unscored. For defined programs, write intermediate array states as well as the output. Question 3 is a classification-and-repair task: do not execute its unsafe fragment.

Practice 1 A moving index

Find the output. Explain both the position changed by the assignment and the position selected after `k` changes.

```
#include <stdio.h>

int main(void)
{
    int rack[5] = {8, 3, 6, 1, 4};
    int k = 2;
    rack[k] = rack[k - 1] + rack[k + 1];
    k = k + 1;
    printf("%d %d %d\n", rack[2], rack[k], k);
    return 0;
}
```

Practice 2 Two different initializer lists

Give both final arrays and the printed triple. Does {2} fill an array with copies of 2?

```
#include <stdio.h>

int main(void)
{
    int first[4] = {2};
    int second[4] = {2, 2, 2, 2};
    first[3] = first[0] + second[1];
    second[0] = first[1];
    printf("%d %d %d\n", first[3], second[0], first[1]);
    return 0;
}
```

Practice 3 Classify and repair the last index

The intention is to add the first and last elements of a three-element array. The following fragment is deliberately unsafe. Read it only; do not compile or execute it as an output question.

```
int data[3] = {2, 5, 8};
int total = data[0] + data[3];
```

Is a particular numeric value of `total` prescribed by C11? Identify the invalid access, then change only its index to meet the stated intention. Give the result of the repaired calculation.

Practice 4 A guard before an access

Trace both `if` statements. Give the final array and explain why the first selected index does not cause an invalid access in this program.

```
#include <stdio.h>

int main(void)
{
    int cells[3] = {4, 6, 8};
    int index = -1;
    if (index >= 0 && index < 3) {
        cells[index] = 9;
    }
    index = 2;
    if (index >= 0 && index < 3) {
        cells[index] = 9;
    }
    printf("%d %d %d\n", cells[0], cells[1], cells[2]);
    return 0;
}
```

For a general integer index, would changing the upper test to `index <= 3` still protect a three-element array? Explain the boundary that decides the answer.

Full solutions and wrong-turn feedback

Practice 1 solution

Initially `k = 2` and `rack = {8, 3, 6, 1, 4}`. The assignment uses destination index 2 and source indices `2 - 1 = 1` and `2 + 1 = 3`. All three are within 0 through 4.

Read `rack[1] = 3` and `rack[3] = 1`; their sum is 4. Store it at index 2, leaving `rack = {8, 3, 4, 1, 4}`. Next `k = k + 1` makes `k = 3` without changing the array. The print reads `rack[2] = 4`, `rack[3] = 1` and `k = 3`. Output: 4 1 3, then a newline.

`4 4 3` assumes that `rack[k]` keeps naming the previously selected element after `k` changes. `6 1 3` ignores the array assignment. Using 8 and 6 as the sum's operands reads positions 0 and 2 instead of positions 1 and 3; calculate `k - 1` and `k + 1` again.

Practice 2 solution

The declarations give `first = {2, 0, 0, 0}` and `second = {2, 2, 2, 2}`. Only the first array has omitted initializer values.

`first[3] = first[0] + second[1]` computes `2 + 2 = 4`, so `first = {2, 0, 0, 4}`. Then `second[0] = first[1]` reads the zero at index 1 of `first` and stores it in `second`. The final arrays are `first = {2, 0, 0, 4}` and `second = {0, 2, 2, 2}`. Output: 4 0 0, then a newline.

`4 2 2` repeats the supplied 2 into omitted positions. `4 4 0` reads `first[3]` where the code says `first[1]`. The two arrays remain distinct even when some of their elements have equal values.

Practice 3 solution

The valid indices are 0, 1 and 2. `data[3]` tries to read outside this array. The fragment has undefined behavior; C11 does not prescribe a numeric value for `total`. An observed number, zero or a crash would not establish a valid language rule. The partial-initializer rule says nothing about storage outside an array.

Replace `data[3]` with `data[2]`. The repaired program is:

```
#include <stdio.h>

int main(void)
{
    int data[3] = {2, 5, 8};
    int total = data[0] + data[2];
    printf("%d\n", total);
    return 0;
}
```

It reads the first value 2 and last value 8, computes 10 and prints 10 followed by a newline. Using `data[1]` would produce 7 but add the first and middle elements, contrary to the task. Increasing the array size would change the given problem rather than repair only the index.

Practice 4 solution

Start with `cells = {4, 6, 8}` and `index = -1`. The first comparison `index >= 0` is false. `&&` skips the second comparison, the `if` condition is false, and the body containing `cells[index]` is not executed. No array access using `-1` occurs.

After `index = 2`, both comparisons in the second condition are true: 2 is nonnegative and below 3. The body writes 9 to `cells[2]`. The final array is `{4, 6, 9}`, and the output is `4 6 9`, followed by a newline.

The correct guard for these element accesses requires both boundaries. `index <= 3` would allow index 3, which is one beyond the last valid index 2. It happens not to change the two index cases shown, but it is not a valid guard for every integer index. Checking only `index < 3` would also admit negative indices. A guard must decide before the element access; a test afterward cannot repair an access that has already occurred.

Quick reference

- In `int a[4] = {6, 2, 9, 4};`, the array has four elements, at indices 0 through 3
- An index selects a position; it is not the value stored there
- For `N` elements, an integer element index must satisfy `0 <= i` and `i < N`
- Trace one selected destination and carry unchanged elements forward
- A scalar initialized from `a[i]` keeps its copied value after that element changes
- In a positive fixed-size integer array with an initializer list, omitted elements start at zero
- An automatic array with no initializer does not get guaranteed zeros
- Check every calculated index. A valid `k` does not guarantee a valid `k - 1` or `k + 1`
- Invalid accesses are behavior-classification questions, not numeric-output puzzles

Before moving on

Explain why the count 4 in a declaration does not make index 4 valid, and why `{2}` and `{2, 2, 2, 2}` give different four-element starting states. You are ready for the next lesson when you can follow each element's state without losing the current index. Next we will visit several elements with a loop and explain what a running total or count means.

Source note

The examples, traces, exercises and explanatory wording are original. Facts were checked against the WG14 N1570 C11 committee draft: array types §6.2.5p20; subscripting §6.5.2.1p2; bounds §6.5.6p8; initialization §§6.7.9p10, p17, p21; constant array-size constraint §6.7.6.2p1; simple assignment §6.5.16.1p2; short circuit §6.5.13p4. Reference: <https://open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>

C11 is this course's explicit teaching convention, not a claimed GATE-specified version. This lesson covers introductory arrays, not the complete C language or GATE CS syllabus.

Lesson 01 · हिन्दी

ऐरे के तत्व और सही इंडेक्स

कौन-से मान बदलते हैं

आप पूर्णांक चर का वर्तमान मान दर्ज करना जानते हैं। अब मान लें कि चार संबंधित मान एक साथ रखने हैं। उन्हें एक ऐरे में रखकर भी हम हर असाइनमेंट का एक-एक चरण में ट्रेस कर सकते हैं।

इस पाठ में आप ऐरे के सही इंडेक्स पहचानेंगे, उसके शुरुआती मान पढ़ेंगे और समझाएँगे कि असाइनमेंट किस तत्व को बदलता है। आप अलग पूर्णांक चर में बचाई गई कॉपी को ऐरे के तत्व से अलग रखेंगे और आंशिक इनिशियलाइज़र से मिलने वाले शून्यों का कारण भी समझेंगे।

पहले से आवश्यक बातें: चर, वर्तमान अवस्था का ट्रेस, सरल अंकगणित, शर्तें और मानों की कॉपी वाले बुनियादी पाठ। यहाँ पॉइंटर, लूप, स्ट्रिंग या मेमोरी एड्रेस की जानकारी आवश्यक नहीं है।

पढ़ाने के लिए हम C11 का उपयोग जारी रखते हैं। हमारे ऐरे में साधारण signed int तत्व हैं और उनका आकार स्पष्ट धनात्मक स्थिरांक है। चलाए जा सकने वाले हर उदाहरण में पढ़े गए मान पहले इनिशियलाइज़ किए गए हैं और गणना का हर पूर्णांक परिणाम -32767 से 32767 के भीतर है। हम यह नहीं मानते कि एक int चार बाइट का होता है। हर पूरा प्रोग्राम नई शुरुआत वाला अलग उदाहरण है और अलग से कंपाइल किया जा सकता है। एक जानबूझकर असुरक्षित अंश केवल वर्गीकरण के लिए स्पष्ट रूप से चिह्नित है।

एक ऐरे का नाम और कई तत्वों की जगहें

`int scores[4] = {6, 2, 9, 4};` को दो हिस्सों में पढ़ें। इस घोषणा में `scores[4]` चार पूर्णांक तत्वों वाला ऐरे बनाता है। ब्रेसिज़ में दिया इनिशियलाइज़र क्रम से उनके शुरुआती मान देता है।

इंडेक्स	0	1	2	3
शुरुआती मान	6	2	9	4

इंडेक्स किसी जगह को पहचानता है। पहली जगह का इंडेक्स 0 है, इसलिए पढ़ने पर `scores[0]` से 6 और `scores[1]` से 2 मिलता है। चौथा और अंतिम तत्व `scores[3]` है। घोषित तत्वों की संख्या 4 है; वह इस ऐरे का सही इंडेक्स नहीं है।

किसी जगह और उसमें रखे मान को अलग रखें। मान 9 इंडेक्स 2 पर है। इससे ऐरे में इंडेक्स 9 वाला तत्व नहीं बन जाता। किसी तत्व में ऋणात्मक संख्या या ऐरे की लंबाई से बड़ी संख्या हो सकती है; रखी हुई संख्या सही जगहों की सीमा नहीं बदलती।

घोषणा में और बाद के व्यंजक में इन्हीं वर्ग कोष्ठकों का काम अलग है। घोषणा `int scores[4]` तत्वों की संख्या बताती है। बाद में `scores[4]` पढ़ने का अर्थ पाँचवें तत्व को पढ़ने की कोशिश होगा, जबकि इस ऐरे में वह तत्व है ही नहीं।

N तत्वों वाले ऐरे में पूर्णांक इंडेक्स i से किसी तत्व तक पहुँचने के लिए दोनों शर्तें चाहिए: $i \geq 0$ और $i < N$ । यहाँ व्याख्या में N तत्वों की संख्या है; वास्तविक घोषणाओं में हम 4 और 5 जैसे धनात्मक स्थिरांक लिखते हैं। अंतिम सही इंडेक्स $N - 1$ है। इन ऐरे-अभिगमों के लिए ऋणात्मक इंडेक्स और तत्वों की संख्या के बराबर इंडेक्स, दोनों गलत हैं।

परिचित असाइनमेंट नियम एक तत्व पर लगाएँ

`scores[1] = scores[0] + scores[2];` की दाईं ओर दो तत्वों के वर्तमान मान पढ़े जाते हैं। उनका योग इंडेक्स 1 वाले तत्व में रखा जाता है। यह स्टेटमेंट हर तत्व में असाइनमेंट नहीं करता।

हर जगह के लिए एक प्रविष्टि वाला रिकॉर्ड बनाएँ:

1. जिन तत्वों तक पहुँचना है, उनके इंडेक्स निकालें और सीमा जाँचें
2. दाईं ओर की गणना में लगने वाले वर्तमान मान पढ़ें
3. परिणाम निकालें
4. चुने गए तत्व में परिणाम लिखें और बाकी प्रविष्टियाँ ज्यों की त्यों आगे रखें

हमारे स्टेटमेंट इंडेक्स बदलने और तत्व अपडेट करने को अलग रखते हैं। एक ही व्यंजक में इंडेक्स को बढ़ाने के साथ उसका दूसरा उपयोग नहीं जोड़ते। यह बुनियादी पाठ में सीखे सुरक्षित अनुक्रम का ही विस्तार है।

हल किया हुआ उदाहरण 1 सहेजा गया मान और दो तत्वों में बदलाव

ट्रेस पढ़ने से पहले पूरा आउटपुट बताएँ। `scores[1]` बदलने पर क्या `saved` भी बदलेगा?

```
#include <stdio.h>

int main(void)
{
    int scores[4] = {6, 2, 9, 4};
    int saved = scores[1];
    scores[1] = scores[0] + scores[2];
    scores[0] = saved - 1;
    printf("%d %d %d %d | %d\n",
           scores[0], scores[1], scores[2], scores[3], saved);
    return 0;
}
```

- इनिशियलाइज़ेशन के बाद: `scores = {6, 2, 9, 4}`
- `saved = scores[1]` वर्तमान मान 2 पढ़ता है। अब `saved = 2` है; ऐसे नहीं बदला
- `scores[1] = scores[0] + scores[2]` में $6 + 9 = 15$ मिलता है। अब `scores = {6, 15, 9, 4}` और `saved = 2` है
- `scores[0] = saved - 1` में $2 - 1 = 1$ मिलता है। अब `scores = {1, 15, 9, 4}` और `saved = 2` है
- प्रिंट होने वाले आर्गुमेंट का क्रम पहले चार तत्व, फिर `saved` है। आउटपुट `1 15 9 4 | 2` है, जिसके बाद नई पंक्ति आती है। | केवल प्रिंट किया गया विभाजक है

उपयोग हुए सभी इंडेक्स 0 से 3 के भीतर हैं। हर असाइनमेंट एक तत्व का रखा हुआ मान बदलता है; बाकी जगहों पर पुराने मान बने रहते हैं।

saved में पूर्णांक मान की कॉपी है। वह तत्व 1 का दूसरा नाम नहीं है और उसके साथ लगातार नहीं बदलता। यदि आपने 14 15 9 4 | 15 बताया, तो सहेजी गई कॉपी को स्थायी जुड़ाव मान लिया। यदि पहला तत्व 6 ही रखा, तो तत्व पर दूसरा असाइनमेंट छोड़ दिया। यदि हर जगह 15 रख दिया, तो चुने हुए एक तत्व को पूरा एरे समझ लिया।

आंशिक इनिशियलाइज़ेशन बाकी शून्य देता है

धनात्मक स्थिर आकार वाले पूर्णांक एरे में तत्वों से कम शुरुआती मान दिए जा सकते हैं। `int bins[5] = {4, -1};` में लिखी दो संख्याएँ जगह 0 और 1 को इनिशियलाइज़ करती हैं। बाकी जगह 2, 3 और 4 का शुरुआती मान शून्य होता है।

इंडेक्स	0	1	2	3	4
शुरुआती मान	4	-1	0	0	0

आकार अभी भी पाँच है। इनिशियलाइज़र में मान न लिखने से तत्व हटता नहीं, आखिरी लिखा मान दोहराया नहीं जाता और बचा तत्व इनपुट की प्रतीक्षा में नहीं रहता। इसलिए `int ready[3] = {0};` तीनों तत्वों को शून्य से शुरू करता है: पहले के लिए शून्य स्पष्ट लिखा है और बाकी दो को शून्य अपने-आप मिलता है।

यह नियम `main` के भीतर `int bins[5];` पर न लगाएँ। उस घोषणा में कोई इनिशियलाइज़र नहीं है। उसके `automatic` तत्वों को शून्य मिलने की गारंटी नहीं है। पढ़ने से पहले ज्ञात मान रखें; बिना इनिशियलाइज़ किए मान पढ़कर यह तय न करें कि C में “आमतौर पर” क्या मिलता है। `static storage` वाले एरे के इनिशियलाइज़ेशन नियम अलग हैं और इस पाठ के बाहर हैं।

घोषणाओं तक पहुँचने पर इनिशियलाइज़र सूची एक बार शुरुआती मान रखती है। बाद का असाइनमेंट इनिशियलाइज़ेशन फिर शुरू नहीं करता और बाकी तत्वों को दोबारा शून्य से नहीं भरता।

हल किया हुआ उदाहरण 2 अपने-आप मिले शुरुआती मान का उपयोग

इस उदाहरण में स्पष्ट दिए गए मान और अपने-आप मिले शून्य, दोनों हैं। हर स्टेटमेंट के बाद बची हुई अवस्था से अगला चरण करें।

```
#include <stdio.h>

int main(void)
{
    int bins[5] = {4, -1};
    bins[3] = bins[0] + bins[2];
    bins[2] = bins[1] - bins[4];
    printf("%d %d %d %d %d\n",
           bins[0], bins[1], bins[2], bins[3], bins[4]);
    return 0;
}
```

- घोषणा से `bins = {4, -1, 0, 0, 0}` मिलता है
- `bins[3] = bins[0] + bins[2]` में ऑपरेंड 4 और 0 हैं। इंडेक्स 3 पर 4 रखें: {4, -1, 0, 4, 0}
- `bins[2] = bins[1] - bins[4]` में ऑपरेंड -1 और 0 हैं। इंडेक्स 2 पर -1 रखें: {4, -1, -1, 4, 0}
- आउटपुट: 4 -1 -1 4 0, फिर नई पंक्ति

पहले असाइनमेंट में `bins[2]` को पढ़ना सही है: उसका अपने-आप मिला शुरुआती मान शून्य है। बाद का असाइनमेंट उसी तत्व को -1 कर देता है। अपने-आप मिला शून्य शुरुआती मान है, हमेशा बने रहने वाली पाबंदी नहीं।

4 की पाँच कॉपियाँ या अंत में -1 की तीन कॉपियाँ लिखना इनिशियलाइज़र सूची को गलत पढ़ना होगा। `bins[4]` को बिना इनिशियलाइज़र हुआ बताना भी गलत होगा, क्योंकि इस इनिशियलाइज़र किए गए ऐसे के बचे तत्वों पर शून्य वाला नियम लागू है। फिर भी `bins[5]` पढ़ना गलत है; आंशिक इनिशियलाइज़ेशन घोषित आकार के बाहर कोई अतिरिक्त तत्व नहीं बनाता।

इंडेक्स की गणना भी हो सकती है

कोष्ठकों के भीतर पूर्णांक व्यंजक हो सकता है। k का उपयुक्त मान होने पर `scores[k]` वर्तमान k की बताई जगह चुनता है, जबकि `scores[k - 1]` पिछली जगह चुनता है। हर निकले हुए इंडेक्स को अलग जाँचें। k सही होने से $k - 1$ या $k + 1$ अपने-आप सही नहीं हो जाता।

चार तत्वों वाले `scores` में $k = 0$ होने पर `scores[k]` सही है, पर `scores[k - 1]` नहीं। $k = 3$ होने पर `scores[k]` सही है, पर `scores[k + 1]` नहीं। बाद में इंडेक्स चर बदलने से पहले हुआ ऐसे अपडेट वापस नहीं होता। उससे यह बदलता है कि अगला व्यंजक कौन-सी जगह चुनेगा।

हल देखने से पहले अभ्यास करें

ये चार मौलिक प्रश्न सीखने के लिए हैं और इनके अंक नहीं हैं। निश्चित व्यवहार वाले प्रोग्राम में आउटपुट के साथ बीच की ऐसे अवस्थाएँ भी लिखें। प्रश्न 3 वर्गीकरण और सुधार का प्रश्न है: उसका असुरक्षित अंश न चलाएँ।

अभ्यास 1 बदलता हुआ इंडेक्स

आउटपुट बताएँ। समझाएँ कि असाइनमेंट किस जगह को बदलता है और k बदलने के बाद कौन-सी जगह चुनी जाती है।

```
#include <stdio.h>

int main(void)
{
    int rack[5] = {8, 3, 6, 1, 4};
    int k = 2;
    rack[k] = rack[k - 1] + rack[k + 1];
    k = k + 1;
    printf("%d %d %d\n", rack[2], rack[k], k);
    return 0;
}
```

अभ्यास 2 दो अलग इनिशियलाइज़र सूचियाँ

दोनों अंतिम ऐरे और प्रिंट होने वाली तिकड़ी बताएँ। क्या {2} ऐरे को 2 की कॉपियों से भर देता है?

```
#include <stdio.h>

int main(void)
{
    int first[4] = {2};
    int second[4] = {2, 2, 2, 2};
    first[3] = first[0] + second[1];
    second[0] = first[1];
    printf("%d %d %d\n", first[3], second[0], first[1]);
    return 0;
}
```

अभ्यास 3 अंतिम इंडेक्स का वर्गीकरण और सुधार

उद्देश्य तीन तत्वों वाले ऐरे के पहले और अंतिम तत्व को जोड़ना है। नीचे का अंश जानबूझकर असुरक्षित है। इसे केवल पढ़ें, आउटपुट वाले प्रश्न की तरह कंपाइल या निष्पादित न करें।

```
int data[3] = {2, 5, 8};
int total = data[0] + data[3];
```

क्या C11 total के लिए कोई निश्चित संख्यात्मक मान बताता है? गलत अभिगम पहचानें, फिर बताएं उद्देश्य के लिए केवल उसका इंडेक्स बदलें। सुधारी गई गणना का परिणाम दें।

अभ्यास 4 अभिगम से पहले गार्ड

दोनों if का ट्रेस करें। अंतिम ऐरे दें और समझाएँ कि पहले चुना गया इंडेक्स इस प्रोग्राम में गलत ऐरे-अभिगम क्यों नहीं कराता।

```
#include <stdio.h>

int main(void)
{
    int cells[3] = {4, 6, 8};
    int index = -1;
    if (index >= 0 && index < 3) {
        cells[index] = 9;
    }
    index = 2;
    if (index >= 0 && index < 3) {
        cells[index] = 9;
    }
    printf("%d %d %d\n", cells[0], cells[1], cells[2]);
    return 0;
}
```

सामान्य पूर्णांक इंडेक्स के लिए ऊपरी जाँच को `index <= 3` कर दें, तो क्या तीन तत्वों वाला ऐरे अब भी सुरक्षित रहेगा? उत्तर तय करने वाली सीमा समझाएँ।

पूरे हल और गलत उत्तरों का कारण

अभ्यास 1 का हल

शुरुआत में `k = 2` और `rack = {8, 3, 6, 1, 4}` है। असाइनमेंट का गंतव्य इंडेक्स 2 है और स्रोत इंडेक्स $2 - 1 = 1$ तथा $2 + 1 = 3$ हैं। तीनों 0 से 4 के भीतर हैं।

`rack[1] = 3` और `rack[3] = 1` पढ़ें, उनका योग 4 है। इसे इंडेक्स 2 पर रखें, जिससे `rack = {8, 3, 4, 1, 4}` मिलता है। अगला `k = k + 1` ऐरे बदले बिना `k = 3` कर देता है। प्रिंट में `rack[2] = 4`, `rack[3] = 1` और `k = 3` पढ़े जाते हैं। आउटपुट: 4 1 3, फिर नई पंक्ति।

4 4 3 मानता है कि `k` बदलने पर भी `rack[k]` पहले चुने हुए तत्व को ही बताता रहता है। 6 1 3 ऐरे पर हुए असाइनमेंट को छोड़ देता है। जोड़ में 8 और 6 लेने से जगह 1 और 3 के बजाय जगह 0 और 2 पढ़ी जाती हैं; `k - 1` और `k + 1` का इंडेक्स फिर निकालें।

अभ्यास 2 का हल

घोषणाओं से `first = {2, 0, 0, 0}` और `second = {2, 2, 2, 2}` मिलता है। केवल पहले ऐरे की सूची में कुछ शुरुआती मान नहीं लिखे गए हैं।

`first[3] = first[0] + second[1]` में $2 + 2 = 4$ मिलता है, इसलिए `first = {2, 0, 0, 4}` हो जाता है। फिर `second[0] = first[1]` में `first` के इंडेक्स 1 वाला शून्य पढ़कर `second` में रखा जाता है। अंतिम ऐरे `first = {2, 0, 0, 4}` और `second = {0, 2, 2, 2}` हैं। आउटपुट: 4 0 0, फिर नई पंक्ति।

4 2 2 लिखे हुए 2 को छोड़ी गई जगहों पर दोहराता है। 4 4 0 वहाँ `first[3]` पढ़ता है जहाँ कोड में `first[1]` है। कुछ तत्वों के मान बराबर होने पर भी दोनों ऐरे अलग रहते हैं।

अभ्यास 3 का हल

सही इंडेक्स 0, 1 और 2 हैं। `data[3]` इस ऐरे के बाहर पढ़ने की कोशिश करता है। अंश में अपरिभाषित व्यवहार है; C11 total के लिए कोई निश्चित संख्यात्मक मान नहीं देता। कोई संख्या, शून्य या क्रैश दिख जाने से भाषा का सही नियम सिद्ध नहीं होता। आंशिक इनिशियलाइज़र वाला नियम ऐरे के बाहर की जगह के बारे में कुछ नहीं बताता।

`data[3]` की जगह `data[2]` रखें। सुधरा हुआ प्रोग्राम है:

```
#include <stdio.h>

int main(void)
{
    int data[3] = {2, 5, 8};
    int total = data[0] + data[2];
    printf("%d\n", total);
    return 0;
}
```

यह पहला मान 2 और अंतिम मान 8 पढ़ता है, 10 निकालता है और 10 के बाद नई पंक्ति प्रिंट करता है। `data[1]` लेने पर 7 मिलेगा, लेकिन वह पहले और बीच के तत्व का जोड़ होगा, जो प्रश्न का उद्देश्य नहीं है। ऐरे का आकार बढ़ाना केवल इंडेक्स सुधारने के बजाय दिए हुए प्रश्न को बदल देगा।

अभ्यास 4 का हल

शुरुआत में `cells = {4, 6, 8}` और `index = -1` है। पहली तुलना `index >= 0` गलत है। `&&` दूसरी तुलना को छोड़ देता है, `if` की शर्त गलत होती है और `cells[index]` वाली `body` नहीं चलती। `-1` का उपयोग करके ऐरे तक कोई अभिगम नहीं होता।

`index = 2` के बाद दूसरी शर्त की दोनों तुलनाएँ सही हैं: 2 गैर-ऋणात्मक है और 3 से छोटा है। `body cells[2]` में 9 लिखती है। अंतिम ऐरे `{4, 6, 9}` है और आउटपुट 4 6 9 है, जिसके बाद नई पंक्ति आती है।

इन तत्वों तक पहुँचने के सही गार्ड में दोनों सीमाएँ चाहिए। `index <= 3` से इंडेक्स 3 भी स्वीकार होगा, जबकि वह अंतिम सही इंडेक्स 2 से एक आगे है। दिए हुए दोनों चरणों का परिणाम संयोग से नहीं बदलेगा, लेकिन हर पूर्णांक इंडेक्स के लिए यह सही गार्ड नहीं है। केवल `index < 3` जाँचने पर ऋणात्मक इंडेक्स भी स्वीकार होंगे। गार्ड को तत्व तक पहुँचने से पहले निर्णय लेना चाहिए; बाद की जाँच पहले हो चुके गलत अभिगम को ठीक नहीं कर सकती।

त्वरित संदर्भ

- `int a[4] = {6, 2, 9, 4};` में चार तत्व हैं, जिनके इंडेक्स 0 से 3 हैं
- इंडेक्स जगह चुनता है; वह वहाँ रखा हुआ मान नहीं है
- `N` तत्वों के लिए पूर्णांक तत्व-इंडेक्स में `0 <= i` और `i < N` दोनों सही होने चाहिए
- चुने हुए एक गंतव्य का ट्रेस करें और न बदले तत्व आगे वैसे ही रखें
- `a[i]` से इनिशियलाइज़र हुआ पूर्णांक चर, वह तत्व बदलने के बाद भी अपना कॉपी हुआ मान रखता है
- धनात्मक स्थिर आकार वाले पूर्णांक ऐरे की इनिशियलाइज़र सूची में बचे तत्व शून्य से शुरू होते हैं
- बिना इनिशियलाइज़र वाले `automatic` ऐरे को शून्य मिलने की गारंटी नहीं है
- हर गणना किया इंडेक्स जाँचें। सही `k` से सही `k - 1` या `k + 1` की गारंटी नहीं मिलती
- गलत अभिगम व्यवहार-वर्गीकरण के प्रश्न हैं, किसी संख्या वाले आउटपुट की पहली नहीं

आगे बढ़ने से पहले

समझाएँ कि घोषणा में संख्या 4 लिखने से इंडेक्स 4 सही क्यों नहीं होता और {2} तथा {2, 2, 2, 2} चार तत्वों को अलग शुरुआती अवस्थाएँ क्यों देते हैं। जब वर्तमान इंडेक्स न खोते हुए हर तत्व की अवस्था का ट्रेस कर सकें तो अगले पाठ के लिए तैयार हैं। आगे हम लूप से कई तत्वों तक जाएँगे और समझेंगे कि चलता हुआ योग या गिनती क्या बताता है।

स्रोत टिप्पणी

उदाहरण, ट्रेस, अभ्यास और समझाने वाले वाक्य मौलिक हैं। तथ्यों की जाँच WG14 N1570 C11 समिति-ड्राफ्ट से की गई: ऐरे प्रकार §6.2.5p20; सबस्क्रिप्ट §6.5.2.1p2; सीमाएँ §6.5.6p8; इनिशियलाइज़ेशन §§6.7.9p10, p17, p21; स्थिर ऐरे-आकार की शर्त §6.7.6.2p1; सरल असाइनमेंट §6.5.16.1p2; शॉर्ट सर्किट §6.5.13p4। संदर्भ: <https://open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>

C11 इस पाठ्यक्रम में पढ़ाने के लिए चुना गया संस्करण है; इसे GATE द्वारा निर्धारित संस्करण नहीं बताया जा रहा। यह पाठ ऐरे की शुरुआती समझ तक सीमित है, पूरी C भाषा या पूरा GATE CS पाठ्यक्रम नहीं है।

Lesson 02 · English

Array scans and what the accumulator means

Visiting elements without losing the state

An array trace records each element. A loop lets us visit several positions in order, but the old rules still apply: use the current state, check the selected index and record each update where it happens.

In this lesson you will explain which prefix of an array has been processed, distinguish an index from a count or total, and find the first matching position without overwriting it with a later match. You will also handle a chosen empty prefix safely.

Prerequisites: the accepted lessons on conditions, short circuit, finite loops, accumulators and current-state tracing, plus the previous lesson on array elements and valid indices. No pointers, early-exit statements or formal complexity notation are needed.

We use C11 as our teaching convention. Arrays have explicit positive constant sizes and initialized `int` elements. A separate `length` selects how many elements from the beginning we will process. It must be between zero and the actual element count, inclusive. All integer calculations shown fit within -32767 through 32767 . Each complete program is independent; do not join them together. Incorrect boundary conditions discussed in prose are for reasoning only, not execution.

Capacity, selected length and the next index

For `int readings[5] = {3, 8, 0, 6, 2};`, the array has five elements at indices 0 through 4. In this lesson, **capacity** means this actual element count. It is not a byte count.

A chosen `length` of 3 means process indices 0, 1 and 2. That beginning part is a prefix. The elements at indices 3 and 4 still exist, but they are outside the selected prefix. A `length` of 5 selects the whole array. A `length` of 0 selects no elements; it does not declare a zero-length C array or remove the existing storage.

For a forward scan, start `i` at 0 and use `i < length`. Just before a test, `i` is the next candidate index. When the test is true, the body may access that position. After the loop's `i++` update stage, the next test uses the new index. This update happens after the body; it is not an extra statement hidden inside the body.

Do not read `readings[length]` as the final selected element. For a nonempty prefix, its final element is at `length - 1`. When `length` is zero, there is no final selected element at all. Our scans handle that case by executing no bodies, without trying to calculate and access index -1 .

Worked example 1 Count selected values and total them

We want the count and sum of readings that are at least 5. The index, the count and the total have different jobs. Predict the output, then explain the meaning of each accumulator.

```
#include <stdio.h>

int main(void)
{
    int readings[5] = {3, 8, 0, 6, 2};
    int length = 5;
    int count = 0;
    int total = 0;
    for (int i = 0; i < length; i++) {
        if (readings[i] >= 5) {
            count++;
            total += readings[i];
        }
    }
    printf("%d %d\n", count, total);
    return 0;
}
```

Trace the state immediately after each body:

Visited index	Element value	At least 5	Count after body	Total after body
0	3	No	0	0
1	8	Yes	1	8
2	0	No	1	8
3	6	Yes	2	14
4	2	No	2	14

After the last body, the update makes the loop's i equal to 5. The test $5 < 5$ is false, so no access at index 5 occurs. The output is 2 14, followed by a newline. The array itself is unchanged: the body reads its elements and writes only the accumulators.

The loop has five bodies, five updates and six loop-condition tests, including the final failed test. Its inner `if` condition is evaluated five times. The two selected inner branches update `count` and `total` twice. A loop body and a selected inner branch are not the same event.

Say what an accumulator means at a checkpoint

Choose the checkpoint just before each loop test. At that point:

- i elements of the chosen prefix have already been visited, at indices 0 through $i - 1$

- `count` is the number of those visited elements whose value is at least 5
- `total` is the sum of those qualifying visited values

When `i` is zero, there are no visited elements. Count 0 and total 0 are therefore the correct starting state. The phrase “0 through `i - 1`” describes an empty set at this point; it is not an instruction to access index `-1`.

If the test succeeds, inspect exactly the next element. When it qualifies, add one to the count and add its value to the total. When it does not, leave both unchanged. The update advances to the next index. Either way, the checkpoint description is true again at the next test. This is the loop invariant: a stable explanation of what the current variables represent.

When the test finally fails, `i` equals `length`, so the description covers the entire selected prefix. The starting value, progress by one and finite valid length also explain why the loop stops. With `length = 0`, the first test fails: zero bodies, one loop test, count 0 and total 0. The real five-element array still exists.

5 19 would count and add every reading, ignoring the inner condition. 2 2 would confuse the sum with the number of qualifying elements. Resetting `total = 0` inside each body would discard earlier contributions. These are different mistakes, so the explanation must name what each variable is meant to retain.

Worked example 2 Keep the first matching position

A target can occur more than once. We will scan the whole chosen prefix while recording only its earliest matching index. The value `-1` is our **not-found marker** because it is outside the valid element indices. It is a signal, never an index to read.

```
#include <stdio.h>

int main(void)
{
    int values[6] = {7, 2, 5, 2, 9, 2};
    int length = 6;
    int target = 2;
    int first = -1;
    for (int i = 0; i < length; i++) {
        if (first == -1 && values[i] == target) {
            first = i;
        }
    }
    printf("%d\n", first);
    return 0;
}
```

- Before visiting index 0: `first = -1`; no visited match exists
- At index 0, `first == -1` is true, so compare value 7 with target 2. It does not match; `first` stays `-1`
- At index 1, no match has yet been recorded. Value 2 matches, so store the index: `first = 1`

- At indices 2, 3, 4 and 5, `first == -1` is false. `&&` skips the element comparison, the assignment is skipped, and `first` stays 1
- The update after index 5 prepares index 6; the loop test fails. Output: 1, followed by a newline

The variable stores a position, not the target value. The answer is 1, not 2. The later matching positions 3 and 5 do not replace the saved position. The `first == -1` guard protects the first result from being overwritten; it does not stop the loop. There are still six bodies and seven loop tests. The comparison `values[i] == target` is evaluated only at indices 0 and 1 in this run.

At the test checkpoint, `first` has one of two meanings: `-1` if no visited element matches, or the smallest matching index among the positions already visited. A newly encountered match is recorded only while the marker is still `-1`. Once a first match exists, preserving it is correct even if later positions also match.

If no chosen element matches, the marker remains `-1`. If the chosen length is zero, no positions are visited and it also remains `-1`. Neither case permits reading `values[first]`. Test whether a match was found before using the recorded position. In the programs here, `if (first != -1)` is sufficient because our algorithm produces only `-1` or a valid selected index; that is not a general validation rule for an arbitrary integer supplied by someone else.

Define the contract before the loop

Every scan here assumes initialized storage and `0 <= length <= capacity`. The loop condition enforces the selected upper boundary only when that contract is already true. Giving a five-element array `length = 8` does not grow it, and `i < length` cannot protect against that incorrect length.

The sum examples also assume that every intermediate addition fits in `int`. Small examples satisfy this explicitly; a loop pattern is not a promise that any length or values will be safe. We count source-level steps to understand the code, without claiming a particular machine running time or covering formal algorithm complexity.

Practice before the solutions

These four original learning exercises are unscored. Record the state that explains your answer. Do not execute a proposed bad boundary condition to discover whether it happens to print a number.

Practice 1 Accumulate the updated elements

Give the final array and output. Does total accumulate the original values or the values after their updates?

```
#include <stdio.h>

int main(void)
{
    int a[4] = {2, 5, 1, 4};
    int total = 0;
    for (int i = 0; i < 4; i++) {
        a[i] = a[i] + i;
        total += a[i];
    }
    printf("%d %d %d\n", total, a[1], a[3]);
    return 0;
}
```

Practice 2 An empty selected prefix

Count the bodies and loop tests, and give the output. Is changing `i < length` to `i <= length` a correct condition for every allowed length from 0 through 3?

```
#include <stdio.h>

int main(void)
{
    int values[3] = {6, 1, 9};
    int length = 0;
    int count = 0;
    int total = 0;
    for (int i = 0; i < length; i++) {
        count++;
        total += values[i];
    }
    printf("%d %d %d\n", count, total, values[0]);
    return 0;
}
```

Practice 3 A name does not enforce first-match behavior

The program is defined, but does it find the first occurrence? Give its actual output. Then change only the inner condition so it retains the first matching position.

```
#include <stdio.h>

int main(void)
{
    int a[5] = {4, 7, 4, 1, 4};
    int first = -1;
    for (int i = 0; i < 5; i++) {
        if (a[i] == 4) {
            first = i;
        }
    }
    printf("%d\n", first);
    return 0;
}
```

Practice 4 Search only the chosen prefix

The full array contains 6. Explain why the program does or does not report it. Give the output and state whether the guarded element read occurs.

```
#include <stdio.h>

int main(void)
{
    int a[5] = {4, 8, 6, 8, 1};
    int length = 2;
    int target = 6;
    int first = -1;
    for (int i = 0; i < length; i++) {
        if (first == -1 && a[i] == target) {
            first = i;
        }
    }
    int matched_value = 0;
    if (first != -1) {
        matched_value = a[first];
    }
    printf("%d %d\n", first, matched_value);
    return 0;
}
```

Full solutions and wrong-turn feedback

Practice 1 solution

At index 0, store $2 + 0 = 2$; add the updated 2, so `total = 2`. At index 1, store $5 + 1 = 6$; the total becomes 8. At index 2, store $1 + 2 = 3$; the total becomes 11. At index 3, store $4 + 3 = 7$; the total becomes 18.

The final array is {2, 6, 3, 7}. After the final update the loop tests $4 < 4$, which is false. There are four bodies and five loop tests. Output: 18 6 7, then a newline.

At each test, the visited prefix has been updated and `total` contains its updated values' sum. The unvisited suffix still has its original values. The statement order makes the difference: each new element value is stored before the addition reads that element.

A total of 12 adds only the original values. A total of 6 adds only the indices. Outputting 18 5 4 ignores the array's changes even though it correctly totals the updated values. A state record must account for both the stored array and the accumulator.

Practice 2 solution

Initialization sets $i = 0$. The first test is $0 < 0$, which is false. There are zero bodies, zero updates and one loop test. Both accumulators remain zero. The later `values[0]` in `printf` is a separate, valid access to the real array's first element, whose value is still 6. Output: 0 0 6, then a newline.

An empty selected prefix does not make the whole array disappear or require a zero-length declaration. Conversely, the existence of the value 6 does not make a loop with a false first test execute once.

The proposed $i \leq \text{length}$ is wrong. At length 0, it would process index 0 even though no elements were selected. At length 3, it would eventually attempt index 3, outside this three-element array. That full-length case has undefined behavior; no numeric output is prescribed. Classify the bad boundary without running it. The correct count-to-index relationship needs the strict upper test.

Practice 3 solution

The matching positions are 0, 2 and 4. The assignment executes at each of them: `first` becomes 0, then 2, then 4. The intervening elements do not match, so they leave it unchanged. The program prints 4 followed by a newline: it retained the last matching index, despite the name `first`.

Use `first == -1 && a[i] == 4` as the inner condition. A complete repaired program is:

```
#include <stdio.h>

int main(void)
{
    int a[5] = {4, 7, 4, 1, 4};
    int first = -1;
    for (int i = 0; i < 5; i++) {
        if (first == -1 && a[i] == 4) {
            first = i;
        }
    }
    printf("%d\n", first);
    return 0;
}
```

At index 0 the repaired condition records `first = 0`. Later bodies see that the marker is no longer `-1` and leave it unchanged. The repaired output is 0, then a newline. Both versions still have five bodies and six loop tests; only the inner condition differs.

An answer of 4 for the repaired version forgets the new guard. Changing the initializer to 0 would not be a general repair: it would claim a match at index 0 before any comparison, even for a target absent from the array. Renaming the variable does not change execution either.

Practice 4 solution

Only positions 0 and 1 belong to the selected prefix. At index 0, value 4 does not equal 6, so `first` stays `-1`. At index 1, value 8 does not equal 6 either. After the update to index 2, `2 < 2` is false and the scan ends. It never compares the value at index 2.

The test `first != -1` is false, so `a[first]` is not evaluated. `matched_value` keeps its explicitly initialized fallback 0. Output: `-1 0`, then a newline. The marker `-1` carries the not-found information; the fallback 0 is only what this program chose to print alongside it, not a universal not-found element value.

2 6 searches beyond the selected prefix. `-1 6` reads an unselected element despite the failed guard. A zero in `matched_value` alone would not distinguish a found element whose value is zero from no match; keep the position/marker test as part of the explanation.

Quick reference

- Capacity is the actual element count; `length` chooses a prefix from 0 through that capacity
- For a forward scan, `i = 0; i < length; i++` visits exactly the selected positions
- Just before a test, explain the already visited prefix and what each accumulator contains
- A count records how many elements meet a rule; a total adds their values; an index names a position
- Initialize accumulators before the loop so earlier contributions are retained
- Length 0 means zero bodies and one initial failed test, not a zero-sized C array

- A first-match marker of -1 means no match yet; record an index only while the marker remains -1
- The first-match guard does not stop the loop; it prevents later overwrites
- Check that a match exists before using its index; never read an array at the -1 marker
- The chosen length must fit the real array; a loop condition cannot repair a false capacity claim

Before moving on

Explain what each variable means before the next loop test, not just what it prints at the end. You should be able to distinguish an empty prefix, an absent target and a later duplicate match. The next lesson introduces a pointer's target and follows two routes to the same scalar object; it does not change the current-state method you have used here.

Source note

The explanations, programs, traces and exercises are original. C facts were checked against the WG14 N1570 C11 committee draft: indexed elements and bounds §6.5.2.1p2, 6.5.6p8; for order §6.8.5.3p1; loop stopping §6.8.5p4; selected branches §6.8.4.1p2; short circuit §6.5.13p4; assignment and updates §6.5.16.1p2, 6.5.16.2p3, 6.5.2.4p2. Reference: <https://open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>

C11 is this course's teaching convention. These scans introduce array reasoning and a simple search; they do not complete GATE searching, complexity or Programming and Data Structures.

Lesson 02 · हिन्दी

ऐरे में क्रम से जाँच और संचायक का अर्थ

तत्वों तक जाएँ और अवस्था पर नज़र रखें

ऐरे के ट्रेस में हर तत्व दर्ज होता है। लूप से हम कई जगहों तक क्रम से जा सकते हैं, लेकिन पुराने नियम वही हैं: वर्तमान अवस्था लें, चुना इंडेक्स जाँचें और हर अपडेट को उसी जगह दर्ज करें जहाँ वह होता है।

इस पाठ में आप बताएँगे कि ऐरे का कौन-सा शुरुआती हिस्सा संसाधित हो चुका है, इंडेक्स को गिनती और योग से अलग रखेंगे तथा बाद के मिलान से पहला परिणाम बदले बिना पहली मेल खाती जगह खोजेंगे। आप चुने हुए खाली शुरुआती हिस्से को भी सुरक्षित ढंग से संभालेंगे।

पहले से आवश्यक बातें: शर्तें, शॉर्ट सर्किट, सीमित लूप, संचायक और वर्तमान अवस्था के ट्रेस वाले स्वीकृत बुनियादी पाठ, साथ में ऐरे के तत्व और सही इंडेक्स वाला पिछला पाठ। पॉइंटर, बीच में लूप छोड़ने वाले स्टेटमेंट या औपचारिक जटिलता-संकेत आवश्यक नहीं हैं।

पढ़ाने के लिए हम C11 का उपयोग करते हैं। ऐरे के आकार स्पष्ट धनात्मक स्थिरांक हैं और उनके `int` तत्व इनिशियलाइज़ हैं। अलग `length` बताता है कि शुरुआत से कितने तत्व संसाधित करने हैं। यह शून्य से वास्तविक तत्वों की संख्या तक, दोनों सिरों सहित, होना चाहिए। दिखाई गई हर पूर्णांक गणना -32767 से 32767 के भीतर है। हर पूरा प्रोग्राम अलग उदाहरण है; उन्हें जोड़कर एक प्रोग्राम न बनाएँ। गद्य में चर्चा की गई गलत सीमा-शर्तें केवल तर्क के लिए हैं, चलाने के लिए नहीं।

वास्तविक क्षमता, चुनी लंबाई और अगला इंडेक्स

`int readings[5] = {3, 8, 0, 6, 2};` में पाँच तत्व हैं, जिनके इंडेक्स 0 से 4 हैं। इस पाठ में **क्षमता** का अर्थ वास्तविक तत्वों की यही संख्या है। यह बाइट की संख्या नहीं है।

`length` का चुना मान 3 हो, तो इंडेक्स 0, 1 और 2 संसाधित करें। इस शुरुआती हिस्से को `prefix` कहते हैं। इंडेक्स 3 और 4 के तत्व मौजूद रहते हैं, लेकिन चुने हिस्से के बाहर हैं। `length = 5` पूरा ऐरे चुनता है। `length = 0` कोई तत्व नहीं चुनता; उससे C में शून्य लंबाई का ऐरे घोषित नहीं होता और मौजूद स्टोरेज हटता भी नहीं।

आगे की ओर जाँच के लिए `i` को 0 से शुरू करें और `i < length` लें। जाँच से ठीक पहले `i` अगला संभावित इंडेक्स है। जाँच सही हो, तो `body` उस जगह तक पहुँच सकती है। लूप के `i++` अपडेट चरण के बाद अगली जाँच नया इंडेक्स लेती है। यह अपडेट `body` के बाद होता है; यह `body` के भीतर छिपा कोई अतिरिक्त स्टेटमेंट नहीं है।

`readings[length]` को अंतिम चुना तत्व न समझें। गैर-खाली शुरुआती हिस्से का अंतिम तत्व `length - 1` पर है। लंबाई शून्य हो, तो कोई अंतिम चुना तत्व है ही नहीं। हमारे लूप उस स्थिति में कोई `body` नहीं चलाते; वे इंडेक्स -1 निकालकर उस तक पहुँचने की कोशिश नहीं करते।

हल किया हुआ उदाहरण 1 चुने मानों की गिनती और योग

हमें कम-से-कम 5 वाले readings की गिनती और उनका योग चाहिए। इंडेक्स, गिनती और योग का काम अलग है। आउटपुट बताएँ, फिर हर संचायक का अर्थ समझाएँ।

```
#include <stdio.h>

int main(void)
{
    int readings[5] = {3, 8, 0, 6, 2};
    int length = 5;
    int count = 0;
    int total = 0;
    for (int i = 0; i < length; i++) {
        if (readings[i] >= 5) {
            count++;
            total += readings[i];
        }
    }
    printf("%d %d\n", count, total);
    return 0;
}
```

हर body के तुरंत बाद की अवस्था देखें:

देखा गया इंडेक्स	तत्व का मान	कम-से-कम 5	body के बाद गिनती	body के बाद योग
0	3	नहीं	0	0
1	8	हाँ	1	8
2	0	नहीं	1	8
3	6	हाँ	2	14
4	2	नहीं	2	14

आखिरी body के बाद अपडेट लूप के `i` को 5 करता है। जाँच `5 < 5` गलत है, इसलिए इंडेक्स 5 तक कोई अभिगम नहीं होता। आउटपुट 2 14 है, जिसके बाद नई पंक्ति आती है। ऐसे नहीं बदला: body उसके तत्व पढ़ती है और केवल संचायकों में लिखती है।

लूप की पाँच body, पाँच अपडेट और अंतिम गलत जाँच सहित छह लूप-शर्त जाँच हैं। भीतर की `if` शर्त पाँच बार जाँची जाती है। भीतर की दो चुनी शाखाएँ `count` और `total` को दो बार अपडेट करती हैं। लूप की body चलना और भीतर की चुनी शाखा चलना एक ही घटना नहीं है।

किसी चुने बिंदु पर संचायक का अर्थ बताएँ

हर लूप-जाँच से ठीक पहले वाला बिंदु चुनें। वहाँ:

- चुने शुरूआती हिस्से के `i` तत्व देखे जा चुके हैं, जिनके इंडेक्स 0 से `i - 1` हैं

- count उन देखे तत्वों की संख्या है जिनका मान कम-से-कम 5 है
- total उन्हीं योग्य देखे मानों का योग है

i शून्य हो, तो कोई तत्व नहीं देखा गया है। इसलिए शुरुआती गिनती 0 और योग 0 सही हैं। इस बिंदु पर “0 से $i - 1$ ” खाली समूह का वर्णन है; यह इंडेक्स -1 तक पहुँचने का निर्देश नहीं है।

जाँच सही हो, तो ठीक अगला तत्व देखें। वह योग्य हो, तो गिनती में एक और योग में उसका मान जोड़ें। योग्य न हो, तो दोनों को वैसे ही रहने दें। अपडेट अगला इंडेक्स तैयार करता है। दोनों स्थितियों में अगली जाँच पर चुने बिंदु का वर्णन फिर सही रहता है। यही लूप इनवेरिएंट है: वर्तमान चरों का अर्थ समझाने वाला कायम रहने वाला कथन।

अंत में जाँच गलत होने पर i , $length$ के बराबर है, इसलिए वर्णन पूरे चुने शुरुआती हिस्से पर लागू है। शुरुआती मान, हर बार एक की प्रगति और सीमित सही लंबाई यह भी समझाते हैं कि लूप क्यों रुकता है। $length = 0$ हो, तो पहली जाँच ही गलत है: शून्य $body$, एक लूप-जाँच, गिनती 0 और योग 0। वास्तविक पाँच तत्वों वाला ऐरे मौजूद रहता है।

5 19 हर reading को गिनता और जोड़ता है, यानी भीतर की शर्त छोड़ देता है। 2 2 योग को योग्य तत्वों की संख्या समझता है। हर $body$ के भीतर $total = 0$ करने से पहले जोड़े गए मान खो जाएँगे। ये अलग गलतियाँ हैं, इसलिए व्याख्या में बताएँ कि हर चर को क्या बचाकर रखना है।

हल किया हुआ उदाहरण 2 पहली मेल खाती जगह बचाएँ

लक्ष्य एक से अधिक जगह हो सकता है। हम पूरा चुना शुरुआती हिस्सा जाँचेंगे, लेकिन केवल सबसे पहली मेल खाती जगह का इंडेक्स रखेंगे। -1 हमारा **नहीं-मिला संकेत** है, क्योंकि वह सही तत्व-इंडेक्स के बाहर है। यह संकेत है, पढ़ने का इंडेक्स कभी नहीं।

```
#include <stdio.h>

int main(void)
{
    int values[6] = {7, 2, 5, 2, 9, 2};
    int length = 6;
    int target = 2;
    int first = -1;
    for (int i = 0; i < length; i++) {
        if (first == -1 && values[i] == target) {
            first = i;
        }
    }
    printf("%d\n", first);
    return 0;
}
```

- इंडेक्स 0 देखने से पहले: $first = -1$; पहले देखी किसी जगह पर मिलान नहीं है
- इंडेक्स 0 पर $first == -1$ सही है, इसलिए मान 7 की तुलना लक्ष्य 2 से करें। मिलान नहीं है; $first -1$ रहता है
- इंडेक्स 1 पर अभी तक कोई मिलान दर्ज नहीं हुआ है। मान 2 मिलता है, इसलिए इंडेक्स रखें: $first = 1$
- इंडेक्स 2, 3, 4 और 5 पर $first == -1$ गलत है। && तत्व की तुलना छोड़ता है, असाइनमेंट नहीं होता और $first 1$ रहता है
- इंडेक्स 5 के बाद अपडेट इंडेक्स 6 तैयार करता है; लूप-जाँच गलत होती है। आउटपुट: 1, फिर नई पंक्ति

चर में जगह रखी जाती है, लक्ष्य का मान नहीं। उत्तर 1 है, 2 नहीं। बाद के मेल खाते इंडेक्स 3 और 5 सहेजी गई जगह को नहीं बदलते। `first == -1` वाला गार्ड पहले परिणाम को बाद के असाइनमेंट से बचाता है; वह लूप नहीं रोकता। अभी भी छह `body` और सात लूप-जाँच हैं। इस निष्पादन में `values[i] == target` की तुलना केवल इंडेक्स 0 और 1 पर होती है।

जाँच वाले बिंदु पर `first` का दो में से एक अर्थ है: देखे गए किसी तत्व का मिलान न हो, तो `-1`; अन्यथा अभी तक देखी जगहों में सबसे छोटा मेल खाता इंडेक्स। नया मिलान केवल तब दर्ज होता है जब संकेत अभी `-1` है। एक बार पहला मिलान मिल जाए, तो उसे बचाए रखना सही है, भले ही बाद की जगहों पर भी मिलान हों।

किसी चुने तत्व का मिलान न हो, तो संकेत `-1` रहता है। चुनी लंबाई शून्य हो, तो कोई जगह नहीं देखी जाती और तब भी वह `-1` रहता है। दोनों ही स्थितियों में `values[first]` पढ़ना सही नहीं है। दर्ज जगह का उपयोग करने से पहले जाँचें कि मिलान मिला या नहीं। यहाँ के प्रोग्रामों में `if (first != -1)` पर्याप्त है, क्योंकि हमारा तरीका केवल `-1` या कोई सही चुना इंडेक्स देता है; यह किसी और से मिले मनमाने पूर्णांक की सामान्य वैधता-जाँच नहीं है।

लूप से पहले उसकी शर्तें तय करें

यहाँ हर जाँच में इनिशियलाइज़ किया हुआ स्टोरेज और `0 <= length <= capacity` माना गया है। लूप-शर्त चुनी हुई ऊपरी सीमा तभी लागू करती है जब ये शर्तें पहले से सही हों। पाँच तत्वों के ऐरे को `length = 8` देने से वह बड़ा नहीं हो जाता और `i < length` उस गलत लंबाई से सुरक्षा नहीं दे सकता।

योग वाले उदाहरणों में हर बीच का जोड़ `int` में समाने की शर्त भी है। छोटे उदाहरण स्पष्ट रूप से इसे पूरा करते हैं; लूप का ढाँचा हर लंबाई और हर मान के सुरक्षित होने का वादा नहीं है। हम कोड समझने के लिए स्रोत-स्तर के चरण गिनते हैं; किसी निश्चित मशीन-समय का दावा या औपचारिक एल्गोरिदम जटिलता का अध्ययन नहीं कर रहे।

हल देखने से पहले अभ्यास

ये चार मौलिक प्रश्न सीखने के लिए हैं और इनके अंक नहीं हैं। अपने उत्तर को समझाने वाली अवस्था दर्ज करें। कोई प्रस्तावित गलत सीमा-शर्त इसलिए न चलाएँ कि उससे संयोग से कौन-सी संख्या निकलती है।

अभ्यास 1 अपडेट हुए तत्वों को जोड़ें

अंतिम ऐरे और आउटपुट दें। `total` में मूल मान जुड़ते हैं या अपडेट होने के बाद वाले मान?

```
#include <stdio.h>

int main(void)
{
    int a[4] = {2, 5, 1, 4};
    int total = 0;
    for (int i = 0; i < 4; i++) {
        a[i] = a[i] + i;
        total += a[i];
    }
    printf("%d %d %d\n", total, a[1], a[3]);
    return 0;
}
```

अभ्यास 2 चुना शुरुआती हिस्सा खाली है

body और लूप-जाँच की संख्या बताएँ और आउटपुट दें। क्या $i < \text{length}$ की जगह $i \leq \text{length}$ रखना 0 से 3 तक हर अनुमत लंबाई के लिए सही है?

```
#include <stdio.h>

int main(void)
{
    int values[3] = {6, 1, 9};
    int length = 0;
    int count = 0;
    int total = 0;
    for (int i = 0; i < length; i++) {
        count++;
        total += values[i];
    }
    printf("%d %d %d\n", count, total, values[0]);
    return 0;
}
```

अभ्यास 3 नाम पहली जगह मिलने की गारंटी नहीं देता

प्रोग्राम का व्यवहार निश्चित है, लेकिन क्या यह पहली occurrence खोजता है? वास्तविक आउटपुट दें। फिर केवल भीतर की शर्त बदलें, ताकि पहली मेल खाती जगह बनी रहे।

```
#include <stdio.h>

int main(void)
{
    int a[5] = {4, 7, 4, 1, 4};
    int first = -1;
    for (int i = 0; i < 5; i++) {
        if (a[i] == 4) {
            first = i;
        }
    }
    printf("%d\n", first);
    return 0;
}
```

अभ्यास 4 केवल चुना शुरुआती हिस्सा खोजें

पूरे ऐरे में 6 मौजूद है। समझाएँ कि प्रोग्राम उसे बताता है या नहीं और क्यों। आउटपुट दें और बताएँ कि गार्ड के भीतर वाला तत्व-पठन होता है या नहीं।

```
#include <stdio.h>

int main(void)
{
    int a[5] = {4, 8, 6, 8, 1};
    int length = 2;
    int target = 6;
    int first = -1;
    for (int i = 0; i < length; i++) {
        if (first == -1 && a[i] == target) {
            first = i;
        }
    }
    int matched_value = 0;
    if (first != -1) {
        matched_value = a[first];
    }
    printf("%d %d\n", first, matched_value);
    return 0;
}
```

पूरे हल और गलत उत्तरों का कारण

अभ्यास 1 का हल

इंडेक्स 0 पर $2 + 0 = 2$ रखें; अपडेट हुआ 2 जोड़ने से $\text{total} = 2$ मिलता है। इंडेक्स 1 पर $5 + 1 = 6$ रखें; योग 8 हो जाता है। इंडेक्स 2 पर $1 + 2 = 3$ रखें; योग 11 होता है। इंडेक्स 3 पर $4 + 3 = 7$ रखें; योग 18 होता है।

अंतिम ऐरे $\{2, 6, 3, 7\}$ है। अंतिम अपडेट के बाद लूप $4 < 4$ जाँचता है, जो गलत है। चार body और पाँच लूप-जाँच हैं। आउटपुट: 18 6 7, फिर नई पंक्ति।

हर जाँच पर देखे जा चुके शुरुआती हिस्से को अपडेट किया जा चुका है और total में उसके अपडेट हुए मानों का योग है। अभी न देखे गए अंतिम हिस्से में मूल मान हैं। स्टेटमेंट का क्रम यह अंतर बनाता है: हर तत्व का नया मान पहले रखा जाता है, उसके बाद जोड़ वाला स्टेटमेंट वही तत्व पढ़ता है।

योग 12 केवल मूल मान जोड़ता है। योग 6 केवल इंडेक्स जोड़ता है। आउटपुट 18 5 4 अपडेट हुए मानों का योग सही निकालकर भी ऐरे के बदलाव भूलता है। अवस्था के रिकॉर्ड में रखे हुए ऐरे और संचायक, दोनों का हिसाब चाहिए।

अभ्यास 2 का हल

इनिशियलाइज़ेशन $i = 0$ करता है। पहली जाँच $0 < 0$ गलत है। शून्य body, शून्य अपडेट और एक लूप-जाँच हैं। दोनों संचायक शून्य रहते हैं। बाद में printf का $\text{values}[0]$ वास्तविक ऐरे के पहले तत्व तक अलग, सही अभिगम है, जिसका मान अभी भी 6 है। आउटपुट: 0 0 6, फिर नई पंक्ति।

खाली चुने शुरुआती हिस्से से पूरा ऐरे गायब नहीं होता और शून्य लंबाई की घोषणा भी आवश्यक नहीं है। दूसरी ओर, मान 6 मौजूद होने से गलत पहली जाँच वाला लूप एक बार चलने नहीं लगता।

प्रस्तावित `i <= length` गलत है। लंबाई 0 पर वह इंडेक्स 0 संसाधित करेगा, जबकि कोई तत्व चुना ही नहीं गया। लंबाई 3 पर वह अंततः इंडेक्स 3 तक पहुँचने की कोशिश करेगा, जो तीन तत्वों वाले ऐरे के बाहर है। उस पूरी लंबाई वाली स्थिति में अपरिभाषित व्यवहार है; कोई निश्चित संख्यात्मक आउटपुट नहीं दिया जा सकता। गलत सीमा को चलाए बिना उसका वर्गीकरण करें। तत्वों की संख्या और इंडेक्स के सही संबंध के लिए ऊपरी जाँच में `<` चाहिए।

अभ्यास 3 का हल

मेल खाते इंडेक्स 0, 2 और 4 हैं। हर जगह असाइनमेंट होता है: `first` पहले 0, फिर 2, फिर 4 बनता है। बीच के तत्व मेल नहीं खाते, इसलिए उसे नहीं बदलते। प्रोग्राम 4 और फिर नई पंक्ति प्रिंट करता है: `first` नाम होने पर भी उसने अंतिम मेल खाता इंडेक्स रखा।

भीतर की शर्त `first == -1 && a[i] == 4` लें। पूरा सुधरा हुआ प्रोग्राम है:

```
#include <stdio.h>

int main(void)
{
    int a[5] = {4, 7, 4, 1, 4};
    int first = -1;
    for (int i = 0; i < 5; i++) {
        if (first == -1 && a[i] == 4) {
            first = i;
        }
    }
    printf("%d\n", first);
    return 0;
}
```

सुधरी शर्त इंडेक्स 0 पर `first = 0` दर्ज करती है। बाद की `body` देखती हैं कि संकेत अब `-1` नहीं है और उसे वैसे ही रहने देती हैं। सुधरा आउटपुट 0 है, फिर नई पंक्ति। दोनों संस्करणों में पाँच `body` और छह लूप-जाँच हैं; केवल भीतर की शर्त अलग है।

सुधरे संस्करण का उत्तर 4 बताना नया गार्ड भूलना है। शुरुआती मान को 0 कर देना सामान्य सुधार नहीं है: इससे किसी तुलना के पहले ही इंडेक्स 0 पर मिलान का दावा होगा, चाहे लक्ष्य ऐरे में न हो। चर का नाम बदलने से भी निष्पादन नहीं बदलता।

अभ्यास 4 का हल

केवल जगह 0 और 1 चुने शुरुआती हिस्से में हैं। इंडेक्स 0 पर मान 4, लक्ष्य 6 के बराबर नहीं है, इसलिए `first -1` रहता है। इंडेक्स 1 पर मान 8 भी 6 के बराबर नहीं है। अपडेट से इंडेक्स 2 होने के बाद `2 < 2` गलत है और खोज समाप्त होती है। इंडेक्स 2 के मान से तुलना कभी नहीं होती।

जाँच `first != -1` गलत है, इसलिए `a[first]` का मूल्यांकन नहीं होता। `matched_value` का स्पष्ट शुरुआती वैकल्पिक मान 0 बना रहता है। आउटपुट: `-1 0`, फिर नई पंक्ति। `-1` वाला संकेत नहीं मिलने की जानकारी देता है; साथ में 0 प्रिंट करना इस प्रोग्राम की चुनी व्यवस्था है, वह नहीं-मिले तत्व का कोई सार्वभौमिक मान नहीं है।

2 6 चुने शुरुआती हिस्से के बाहर खोजता है। `-1 6` गलत गार्ड के बावजूद किसी न चुने तत्व को पढ़ता है। केवल `matched_value` में शून्य देखकर शून्य मान वाला मिला तत्व और कोई मिलान न होना अलग नहीं किए जा सकते; जगह/संकेत की जाँच को व्याख्या का हिस्सा रखें।

त्वरित संदर्भ

- क्षमता वास्तविक तत्वों की संख्या है; length शून्य से उस क्षमता तक का शुरुआती हिस्सा चुनता है
- आगे की जाँच में $i = 0$; $i < \text{length}$; $i++$ ठीक चुनी हुई जगहों तक जाता है
- जाँच से ठीक पहले देखे हुए शुरुआती हिस्से और हर संचायक का अर्थ बताएँ
- गिनती बताती है कि कितने तत्व नियम पूरा करते हैं; योग उनके मान जोड़ता है; इंडेक्स जगह का नाम देता है
- संचायक लूप से पहले इनिशियलाइज़ करें, ताकि पहले जोड़े गए मान बने रहें
- लंबाई 0 में शून्य body और एक शुरुआती गलत जाँच होती है, शून्य आकार का C ऐसे नहीं
- पहली जगह खोजते समय -1 का संकेत बताता है कि अभी मिलान नहीं मिला; संकेत -1 होने पर ही इंडेक्स दर्ज करें
- पहले मिलान का गार्ड लूप नहीं रोकता; वह बाद के असाइनमेंट से पहले परिणाम को बचाता है
- इंडेक्स के उपयोग से पहले मिलान मिलने की जाँच करें; -1 संकेत को ऐसे पढ़ने के लिए कभी न लें
- चुनी लंबाई वास्तविक ऐसे में समानी चाहिए; लूप-शर्त क्षमता के गलत दावे को ठीक नहीं कर सकती

आगे बढ़ने से पहले

केवल अंत में छपने वाला मान नहीं, बल्कि अगली लूप-जाँच से पहले हर चर का अर्थ समझाएँ। आप खाली शुरुआती हिस्से, अनुपस्थित लक्ष्य और बाद के दोहराए मिलान में अंतर कर सकें। अगला पाठ पॉइंटर का लक्ष्य बताएगा और एक ही पूर्णांक ऑब्जेक्ट तक दो रास्तों का ट्रेस करेगा; यहाँ इस्तेमाल हुआ वर्तमान अवस्था का तरीका नहीं बदलेगा।

स्रोत टिप्पणी

व्याख्याएँ, प्रोग्राम, ट्रेस और अभ्यास मौलिक हैं। C के तथ्यों की जाँच WG14 N1570 C11 समिति-ड्राफ्ट से की गई: इंडेक्स वाले तत्व और सीमाएँ §6.5.2.1p2, 6.5.6p8; for का क्रम §6.8.5.3p1; लूप का रुकना §6.8.5p4; चुनी शाखाएँ §6.8.4.1p2; शॉर्ट सर्किट §6.5.13p4; असाइनमेंट और अपडेट §6.5.16.1p2, 6.5.16.2p3, 6.5.2.4p2। संदर्भ: <https://open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>

C11 इस पाठ्यक्रम में पढ़ाने के लिए चुना गया संस्करण है। ये जाँचें ऐसे का तर्क और सरल खोज शुरू करती हैं; इनसे GATE की पूरी searching, complexity या Programming and Data Structures की तैयारी पूरी नहीं होती।

Lesson 03 · English

Pointer targets and shared objects

Follow the target before changing a value

A copied integer keeps its own value. A copied pointer value can leave two pointer variables identifying the same integer object. To explain a later write, you need two records: the numbers stored in the integer objects, and the current targets of the pointers.

This lesson introduces simple pointers to `int`, address-of `&`, and dereference `*`. You will distinguish a pointer variable from its stored pointer value and its target, trace shared targets and retargeting, and recognise null-pointer and lifetime boundaries.

Prerequisites: initialized integer variables, sequential assignments, copied values, `if`, and short circuit. Array knowledge from the previous lessons is useful context, but every target here is an ordinary scalar integer. We do not yet use pointer arithmetic, array-to-pointer conversion, pointer parameters, allocation or machine addresses.

C11 remains our explicit teaching convention, not a version specified by the GATE syllabus. All arithmetic in the defined complete programs stays within -32767 through 32767 . Each program starts afresh. The two unsafe practice fragments are for classification only; do not compile or run them to guess an output.

A pointer variable holds a pointer value

Consider the declarations `int score = 7;` and `int *p = &score;` inside `main`.

- `score` is an integer object whose current value is 7
- `p` is a separate pointer variable, with type “pointer to `int`”
- The value stored in `p` identifies `score`. We write its target symbolically as `p → score`

A pointer variable is itself an object, but it is not the integer object it points to. Pointer types are scalar types in C: `p` holds one pointer value. That does not make the pointer value an ordinary integer used in our arithmetic.

The `&` in `&score` obtains a pointer to `score`; it does not read the integer 7 and turn 7 into an address. No invented address such as 1000 is needed. Our arrow records identity only, not a byte count, physical location, relative placement or machine layout. The arrow is a drawing convention, not C syntax.

The `*` in the declaration `int *p = &score;` is part of declaring `p` as a pointer. In a later expression, `*p` follows the current pointer value and designates its target. With `p → score`, reading `*p` reads the current integer in `score`. An assignment such as `*p = 12;` writes 12 into `score`. It does not change the target of `p`.

By contrast, `p = &other;`, with another live integer `other`, replaces the pointer value in `p`. It does not write into either integer. Before the next dereference, update the arrow.

Item to record	In this starting example	Question it answers
Integer object and value	<code>score = 7</code>	What number is stored?
Pointer variable and value	<code>p</code> contains the pointer obtained from <code>&score</code>	What does this separate variable hold?
Current target	<code>p → score</code>	Which integer will <code>*p</code> select?

Use a separate declaration for each pointer while learning. A declaration alone is not a promise that a target exists. An automatic pointer declared without an initializer, such as `int *p;`, does not acquire a usable target or a guaranteed null value. Give it a known value before using it.

Copying a pointer value can preserve a shared target

After `int *q = p;`, both arrows end at `score`: `p → score ← q`. The pointer variables `p` and `q` remain separate. The initialization copies the pointer value, not the integer object, and it does not make `q` point to the variable `p`.

We call this a shared target, or aliasing: the expressions `*p` and `*q` currently access the same integer object. A write through either expression changes that one object's value, so a later read through the other expression sees the updated value.

Compare `int saved = *p;`. Here the initializer reads an integer and initializes a separate integer variable. `saved` gets a number, not a continuing connection to the target. This follows the copied-value model from the foundation lessons. What matters is which value was copied: an integer or a pointer.

For each statement:

1. Read the current arrows before resolving any `*p` or `*q`
2. Check that each accessed target is a live, suitable object and that every integer read has a known value
3. Compute from the current integer values
4. Update the one destination: an integer value for `*p = ...`, or an arrow for `p = ...`
5. Carry forward every other number and arrow

Worked example 1 Two routes to one integer

Predict the output. Will q still read 7 after the first store through p? Will saved follow the later updates?

```
#include <stdio.h>

int main(void)
{
    int score = 7;
    int *p = &score;
    int *q = p;
    int saved = *p;
    *p = *p + 5;
    *q = saved + *q;
    printf("%d %d %d\n", score, *q, saved);
    return 0;
}
```

After this step	score	saved	Target of p	Target of q
All declarations	7	7	score	score
*p = *p + 5	12	7	score	score
*q = saved + *q	19	7	score	score

- p is initialized from &score; q copies that pointer value. The diagram is $p \rightarrow \text{score} \leftarrow q$
- saved reads 7 through p, then retains that integer copy
- The first store reads the current 7, adds 5 and writes 12 into score. Neither arrow moves
- The second store reads saved = 7 and the current *q = 12, then writes 19 into the same score
- The final target diagram is still $p \rightarrow \text{score} \leftarrow q$, now with score = 19. Output: 19 19 7, followed by a newline

Predicting 12 14 7 gives q a private copy of the old integer, even though it copied a pointer value.

Predicting 24 24 24 makes saved follow target updates instead of keeping its copied integer. Both errors disappear when pointer values and saved integer values have different entries in the trace. The shared object is still just one integer; writing through two different pointer variables does not create two versions of it.

Worked example 2 Retarget one pointer and keep the other

Now there are two integer objects. Reassigning `p` must not silently move `q`.

```
#include <stdio.h>

int main(void)
{
    int left = 3;
    int right = 11;
    int *p = &left;
    int *q = p;
    p = &right;
    *p = *q + 2;
    *q = *p + 4;
    printf("%d %d %d %d\n", left, right, *p, *q);
    return 0;
}
```

After this step	left	right	Target of p	Target of q
All declarations	3	11	left	left
<code>p = &right</code>	3	11	right	left
<code>*p = *q + 2</code>	3	5	right	left
<code>*q = *p + 4</code>	9	5	right	left

Initially `p → left ← q`, with `right` separate. After retargeting, the diagram is `p → right` and `q → left`. No integer has changed yet.

The first store reads 3 through `q`, adds 2 and writes 5 through `p` into `right`. The next store reads the new 5 through `p`, adds 4 and writes 9 through `q` into `left`. Output: 9 5 5 9, followed by a newline.

If you moved both arrows when executing `p = &right`, you treated `q = p` as a permanent link between the pointer variables. It was a one-time value copy. If you changed `left` to 11 at retargeting, you confused the pointer assignment with `*p = right`. Those are different destinations. The current targets, rather than the pointer variable names, decide which later store reaches which integer.

Null is an explicit absence of a target

Include `<stddef.h>` when using `NULL` in our examples. `int *p = NULL;` initializes a pointer to a known null value. We draw `p → no target`. The pointer variable exists, but this value supplies no integer object that may be read or written through `*p`.

A null pointer is different from a pointer to an integer containing zero. After `int value = 0;` and `int *p = &value;`, the diagram is `p → value` with `value = 0`. Reading `*p` is valid and yields 0. With `p = NULL`, reading or assigning through `*p` is invalid, regardless of the number you wanted to read or store. Do not interpret `NULL` as a promise about an all-zero byte pattern or a physical address.

When `p` is known to be either null or a pointer to a suitable live integer, `if (p != NULL)` can guard a dereference in its body. With null, the body is skipped; with a valid target, the body may use that target. Skipping a write does not create a target or achieve a required update.

The same ordering matters in `p != NULL && *p > 0`: under that same known-value assumption, the left comparison prevents the right operand from being evaluated when `p` is null. Reversing those operands would attempt the read before checking for null. A null comparison is not a general test that an arbitrary pointer is safe. In particular, it does not repair an uninitialized pointer or a pointer whose target has stopped existing.

A target has to remain alive

A plain local `int` in these examples has automatic storage duration. If it belongs to a nested block, its lifetime ends when execution leaves that block. A pointer variable in an enclosing block can remain alive for longer than the integer it once identified. Keeping the pointer variable does not keep the target object alive.

In a target trace, mark the inner object as ended at the closing brace. Any pointer still holding a pointer to that object has an indeterminate value at that boundary under C11. It does not automatically become null. Do not continue drawing it as a usable arrow, dereference it, or try to validate its old value by comparing it with `NULL`.

A safe plan is to finish the target's work and copy any needed integer result into a longer-lived variable before leaving the block. If the longer-lived pointer is no longer needed, assign `NULL` to it while its target is still alive. Clearing one pointer does not clear other pointers to that object; each pointer has its own stored value. Another possible plan is to put the target in a block that lasts through all its required uses. The exercise below uses the first plan.

Lifetime and current number are separate facts. An integer copy can remain valid after the source object ends; a copied pointer does not preserve that object. We need no assumption about “old bytes still being there” to make this distinction.

Practice before opening the solutions

These four original transfer exercises are unscored. For the defined programs, give the output, intermediate integer states and target diagrams. For exercises 3 and 4, classify the unsafe fragment without compiling or executing it, then solve the specified repair.

Practice 1 Equal numbers and a new shared target

Initially, do p and q identify the same object? Show what changes at q = p, then give the output.

```
#include <stdio.h>

int main(void)
{
    int first = 6;
    int second = 6;
    int *p = &first;
    int *q = &second;
    *p = 9;
    q = p;
    *q = *q - 2;
    printf("%d %d %d %d\n", first, second, *p, *q);
    return 0;
}
```

Practice 2 An integer copy and two pointer changes

Trace each destination. Explain why q = p does not copy the number 10 and why the following retargeting of p does not move q.

```
#include <stdio.h>

int main(void)
{
    int red = 4;
    int blue = 10;
    int *p = &red;
    int *q = &blue;
    int saved = *q;
    *p = *q;
    q = p;
    p = &blue;
    *q = *q - 3;
    *p = saved + *q;
    printf("%d %d %d %d %d\n", red, blue, saved, *p, *q);
    return 0;
}
```

Practice 3 Supply the missing target

The intention is to increase total from 8 by 3, using a store through p. Assume <stddef.h> is included and this fragment occurs inside main. It is deliberately unsafe; read it only.

```
int total = 8;
int *p = NULL;
*p = total + 3;
```

Classify the store. Insert one pointer assignment between the declaration of `p` and the store so that the stated update is valid. Explain why merely guarding the original store with `if (p != NULL)` would not achieve the intended update.

Practice 4 Keep the result after the inner object ends

Assume the usual headers and an enclosing `main`. This fragment is deliberately unsafe; do not compile or run it.

```
int saved = 0;
int *p = NULL;
{
    int local = 9;
    p = &local;
    *p = *p + 2;
    saved = *p;
}
printf("%d %d\n", saved, *p);
```

Trace the valid work inside the block. At what boundary does `p` cease to hold a usable pointer to `local`? Does C11 prescribe 11 11 as the output? Repair the fragment while keeping `local` and its update inside the inner block: retain the final integer in `saved`, clear `p` before leaving that block, and print only `saved` afterward.

Full solutions and wrong-turn feedback

Practice 1 solution

After this step	first	second	Target of p	Target of q
All declarations	6	6	first	second
*p = 9	9	6	first	second
q = p	9	6	first	first
*q = *q - 2	7	6	first	first

The initial diagram is `p → first` and `q → second`. Equal stored numbers do not merge two objects. The first store changes only `first`. Then `q = p` changes only `q`'s target: the diagram becomes `p → first ← q`, and `second` still contains 6. The last store reads 9 through `q`, subtracts 2 and writes 7 into `first`. Output: 7 6 7 7, followed by a newline.

7 7 7 7 wrongly joins the two integer objects because their initial values match. 9 4 9 4 ignores the pointer assignment and continues using `second` as `q`'s target. A copied pointer value must be recorded at the moment it is copied.

Practice 2 solution

After this step	red	blue	saved	Target of p	Target of q
All declarations	4	10	10	red	blue
*p = *q	10	10	10	red	blue
q = p	10	10	10	red	red
p = &blue	10	10	10	blue	red
*q = *q - 3	7	10	10	blue	red
*p = saved + *q	7	17	10	blue	red

The first store copies the integer 10 from blue to red; the diagram stays $p \rightarrow \text{red}$ and $q \rightarrow \text{blue}$. Then $q = p$ copies a pointer value, leaving $p \rightarrow \text{red} \leftarrow q$. It does not read $*p$ or $*q$. Retargeting only p produces $p \rightarrow \text{blue}$ and $q \rightarrow \text{red}$.

The subtraction therefore writes $10 - 3 = 7$ into red. The last store reads the saved integer 10 and the current $\text{red} = 7$, then writes 17 into blue. Output: 7 17 10 17 7, followed by a newline.

Making saved become 17 treats an integer copy as an alias. Making the subtraction change blue makes q follow a later assignment to p . Treating $*p = *q$ as an arrow change confuses copying the targets' contents with copying a pointer value. The left side's $*$ is decisive.

Practice 3 solution

Initially $\text{total} = 8$ and $p \rightarrow \text{no target}$. The right-hand arithmetic $8 + 3$ is within range, but that does not supply a destination for the store. Dereferencing the null pointer for $*p = \dots$ has undefined behavior. C11 does not promise an update, a particular output or a crash.

Insert $p = \&\text{total};$ before the store. The complete repaired program is:

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

int main(void)
{
    int total = 8;
    int *p = NULL;
    p = &total;
    *p = total + 3;
    printf("%d %d\n", total, *p);
    return 0;
}
```

- After declarations: $\text{total} = 8, p \rightarrow \text{no target}$
- After $p = \&\text{total}$: $\text{total} = 8, p \rightarrow \text{total}$

- The store reads 8, computes 11 and writes it into `total`. Now `total = 11`, `p → total`
- Both printed expressions read the same live integer. Output: 11 11, followed by a newline

`p = NULL` again would still provide no target. `*p = 0` cannot initialize the pointer; it would try another invalid target write. Guarding the original store with `if (p != NULL)` safely skips it in this known-null case, but leaves `total = 8`, so it does not satisfy the requested update. Assigning `p = &total` first supplies the target rather than merely avoiding the operation.

Practice 4 solution

Follow the valid prefix without assigning a numeric output to the whole unsafe fragment:

Checkpoint	saved	Inner local	State of p
Before inner block	0	Not yet alive	Known null
After <code>local</code> initialization	0	9	Known null
After <code>p = &local</code>	0	9	<code>p → local</code>
After <code>*p = *p + 2</code>	0	11	<code>p → local</code>
After <code>saved = *p</code>	11	11	<code>p → local</code>
After leaving inner block	11	Lifetime ended	Indeterminate; no usable old target

The inner read and write are valid. The closing brace ends `local`'s lifetime, even though `saved` and the pointer variable `p` remain alive in the enclosing block. The later `*p` cannot read that ended object. The fragment has undefined behavior, so C11 does not prescribe 11 11, or even a guaranteed first printed 11. A correct trace of the valid prefix is not a guarantee about the output of an execution containing undefined behavior.

The requested repair is:

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

int main(void)
{
    int saved = 0;
    int *p = NULL;
    {
        int local = 9;
        p = &local;
        *p = *p + 2;
        saved = *p;
        p = NULL;
    }
    printf("%d\n", saved);
    return 0;
}
```

The repair follows the same valid states through `saved = 11`. Then, while `local` is still alive with value 11, `p = NULL` replaces the pointer value: `p → no target`. It does not change `local` or `saved`. After the inner block ends, `p` remains known null and `saved` remains 11. The print reads only the longer-lived integer copy. Output: 11, followed by a newline.

Adding a null test to the original print is not a repair: the original `p` is indeterminate after the lifetime boundary, not a known choice between null and a live target. Copying `p` into a second pointer before the block ends would leave that second pointer with the same lifetime problem. Keeping the integer result is different from keeping another pointer to the ended object.

Quick reference

- `int *p = &x`; creates a pointer variable and initializes it to identify the live integer `x`
- Record integer values and pointer targets separately; arrows are symbolic, not machine addresses
- `*p` selects the current target. Reading it gets the target's integer; assigning through it changes that integer
- `p = &y` retargets `p` without changing either integer
- `q = p` copies the current pointer value. It can create a shared target, but does not make `q` follow future changes to `p`
- `int saved = *p`; copies an integer; later target updates do not change `saved`
- A known null pointer has no dereferenceable target. A pointer to an integer containing zero is different
- A null guard only excludes null under a known-valid-value assumption; it does not establish general pointer validity
- An automatic target must outlive every access to it. Save needed integer results before its lifetime ends

- Invalid dereferences are classification questions; an observed output cannot make them valid

Before moving on

Explain the different destinations in $p = q$ and $*p = *q$. Draw the two arrows after copying p into q and then retargeting only p . Finally, explain why saving an integer can preserve a result across a block boundary while saving a pointer to that block's local integer cannot preserve the target. Next we will connect these target records to function calls while keeping the copied-argument model.

Source note

The examples, diagrams, traces, exercises and explanatory wording are original. Semantic facts were checked against the [WG14 N1570 C11 committee draft](#): pointer and scalar types §§6.2.5p20–21; address-of and dereference §§6.5.3.2p3–4; pointer declarations §6.7.6.1p1; scalar initialization §§6.7.9p10–11; assignment §6.5.16.1p2; null pointers §6.3.2.3p3 and `NULL` §7.19p3; pointer/null comparison §§6.5.9p2, p5–6; short circuit §6.5.13p4; automatic lifetime §§6.2.4p2, p5–6.

C11 is a teaching convention, not a claimed GATE requirement. This lesson covers a bounded introduction to scalar pointer targets, not all pointers, C or GATE CS.

Lesson 03 · हिन्दी

पॉइंटर का लक्ष्य और साझा ऑब्जेक्ट

मान बदलने से पहले लक्ष्य पहचानें

कॉपी किया गया पूर्णांक अपना अलग मान रखता है। पॉइंटर के मान की कॉपी होने पर दो पॉइंटर चर एक ही पूर्णांक ऑब्जेक्ट की ओर हो सकते हैं। बाद में होने वाले लेखन को समझने के लिए दो रिकॉर्ड चाहिए: पूर्णांक ऑब्जेक्ट में रखी संख्याएँ और पॉइंटर के वर्तमान लक्ष्य।

इस पाठ में हम `int` के सरल पॉइंटर, एड्रेस लेने वाला `&` और डीरेफ़रेंस करने वाला `*` सीखेंगे। आप पॉइंटर चर, उसमें रखे पॉइंटर मान और उसके लक्ष्य को अलग पहचानेंगे। साझा लक्ष्य और लक्ष्य बदलने का ट्रेस करेंगे तथा नल पॉइंटर और ऑब्जेक्ट के जीवनकाल की सीमाएँ समझेंगे।

पहले से आवश्यक बातें: इनिशियलाइज़ किए पूर्णांक चर, क्रम से होने वाले असाइनमेंट, मानों की कॉपी, `if` और शॉर्ट सर्किट। पिछले पाठों की ऐरे संबंधी जानकारी उपयोगी संदर्भ है, लेकिन यहाँ हर लक्ष्य एक साधारण स्केलर पूर्णांक है। अभी पॉइंटर अंकगणित, ऐरे से पॉइंटर में रूपांतरण, पॉइंटर पैरामीटर, मेमोरी आवंटन या मशीन एड्रेस का उपयोग नहीं होगा।

पढ़ाने के लिए हम C11 का उपयोग जारी रखते हैं; GATE पाठ्यक्रम ने इस संस्करण को निर्धारित नहीं किया है। निश्चित व्यवहार वाले पूरे प्रोग्रामों की सारी अंकगणित `-32767` से `32767` के भीतर रहती है। हर प्रोग्राम नई शुरुआत वाला उदाहरण है। अभ्यास के दो असुरक्षित अंश केवल वर्गीकरण के लिए हैं; आउटपुट का अनुमान लगाने के लिए उन्हें कंपाइल या निष्पादित न करें।

पॉइंटर चर में पॉइंटर मान होता है

`main` के भीतर घोषणाएँ `int score = 7;` और `int *p = &score;` देखें।

- `score` एक पूर्णांक ऑब्जेक्ट है, जिसका वर्तमान मान 7 है
- `p` एक अलग पॉइंटर चर है, जिसका प्रकार “`int` का पॉइंटर” है
- `p` में रखा मान `score` को पहचानता है। उसका लक्ष्य हम प्रतीक के रूप में `p → score` लिखते हैं

पॉइंटर चर स्वयं भी एक ऑब्जेक्ट है, लेकिन वह उस पूर्णांक ऑब्जेक्ट से अलग है जिसकी ओर वह जाता है। C में पॉइंटर प्रकार भी स्केलर प्रकार हैं: `p` में एक पॉइंटर मान होता है। इससे पॉइंटर मान हमारी अंकगणित में इस्तेमाल होने वाला साधारण पूर्णांक नहीं बन जाता।

`&score` में `&`, `score` की ओर पॉइंटर देता है; वह पूर्णांक 7 पढ़कर 7 को एड्रेस में नहीं बदलता। 1000 जैसा कोई काल्पनिक एड्रेस लेने की आवश्यकता नहीं है। हमारा तीर केवल ऑब्जेक्ट की पहचान बताता है, बाइट की संख्या, भौतिक स्थान, आपसी दूरी या मशीन में रखने का क्रम नहीं। तीर चित्र बनाने का तरीका है, C का सिंटैक्स नहीं।

घोषणा `int *p = &score;` में `*`, `p` को पॉइंटर घोषित करने का हिस्सा है। बाद के व्यंजक में `*p`, वर्तमान पॉइंटर मान के लक्ष्य को चुनता है। `p → score` होने पर `*p` पढ़ने से `score` का वर्तमान पूर्णांक मिलता है। `*p = 12;` जैसा असाइनमेंट `score` में 12 लिखता है। इससे `p` का लक्ष्य नहीं बदलता।

इसके विपरीत, किसी दूसरे जीवित पूर्णांक `other` के लिए `p = &other;`, `p` में रखा पॉइंटर मान बदलता है। वह किसी भी पूर्णांक में नहीं लिखता। अगली बार डीरेफ़रेंस करने से पहले तीर अपडेट करें।

दर्ज करने वाली बात	इस शुरुआती उदाहरण में	इससे किस प्रश्न का उत्तर मिलता है
पूर्णांक ऑब्जेक्ट और मान	<code>score = 7</code>	कौन-सी संख्या रखी है?
पॉइंटर चर और मान	<code>p</code> में <code>&score</code> से मिला पॉइंटर है	इस अलग चर में क्या है?
वर्तमान लक्ष्य	<code>p → score</code>	<code>*p</code> कौन-सा पूर्णांक चुनेगा?

सीखते समय हर पॉइंटर की अलग घोषणा लिखें। केवल घोषणा होने से लक्ष्य मौजूद होने की गारंटी नहीं मिलती। बिना इनिशियलाइज़र घोषित automatic पॉइंटर, जैसे `int *p;`, को उपयोग योग्य लक्ष्य या निश्चित नल मान नहीं मिलता। उपयोग करने से पहले उसे ज्ञात मान दें।

पॉइंटर मान की कॉपी से लक्ष्य साझा रह सकता है

`int *q = p;` के बाद दोनों तीर `score` पर पहुँचते हैं: `p → score ← q`। पॉइंटर चर `p` और `q` अलग ही रहते हैं। इस इनिशियलाइज़ेशन में पॉइंटर मान की कॉपी होती है, पूर्णांक ऑब्जेक्ट की नहीं। इससे `q`, चर `p` की ओर भी नहीं हो जाता।

इसे साझा लक्ष्य या एलियसिंग कहते हैं: अभी व्यंजक `*p` और `*q` एक ही पूर्णांक ऑब्जेक्ट तक पहुँचते हैं। किसी भी व्यंजक से लिखने पर उसी एक ऑब्जेक्ट का मान बदलता है। इसलिए दूसरे व्यंजक से बाद में पढ़ने पर अपडेट किया हुआ मान मिलता है।

इसकी तुलना `int saved = *p;` से करें। यहाँ इनिशियलाइज़र एक पूर्णांक पढ़कर अलग पूर्णांक चर को शुरुआती मान देता है। `saved` में संख्या आती है, लक्ष्य से लगातार जुड़ाव नहीं। यह बुनियादी पाठों के मान-कॉपी वाले मॉडल का ही पालन है। मुख्य बात यह है कि कॉपी किस मान की हुई: पूर्णांक की या पॉइंटर की।

हर स्टेटमेंट के लिए:

1. `*p` या `*q` का लक्ष्य चुनने से पहले वर्तमान तीर देखें
2. जाँचें कि जिस लक्ष्य तक पहुँचना है वह जीवित और उपयुक्त ऑब्जेक्ट है तथा पढ़े जाने वाले हर पूर्णांक का मान ज्ञात है
3. पूर्णांकों के वर्तमान मानों से गणना करें
4. केवल एक गंतव्य अपडेट करें: `*p = ...` में पूर्णांक मान, या `p = ...` में तीर
5. बाकी सभी संख्याएँ और तीर ज्यों के त्यों आगे रखें

हल किया हुआ उदाहरण 1 एक पूर्णांक तक पहुँचने के दो रास्ते

आउटपुट बताएँ। p से पहले लेखन के बाद भी क्या q से 7 पढ़ा जाएगा? क्या saved बाद के अपडेट के साथ बदलेगा?

```
#include <stdio.h>

int main(void)
{
    int score = 7;
    int *p = &score;
    int *q = p;
    int saved = *p;
    *p = *p + 5;
    *q = saved + *q;
    printf("%d %d %d\n", score, *q, saved);
    return 0;
}
```

इस चरण के बाद	score	saved	p का लक्ष्य	q का लक्ष्य
सभी घोषणाएँ	7	7	score	score
*p = *p + 5	12	7	score	score
*q = saved + *q	19	7	score	score

- p को &score से शुरुआती मान मिलता है; q उसी पॉइंटर मान की कॉपी करता है। चित्र है $p \rightarrow \text{score} \leftarrow q$
- saved, p के रास्ते 7 पढ़ता है और फिर उस पूर्णांक की कॉपी रखता है
- पहला लेखन वर्तमान 7 पढ़ता है, उसमें 5 जोड़ता है और score में 12 लिखता है। कोई तीर नहीं बदलता
- दूसरा लेखन saved = 7 और वर्तमान *q = 12 पढ़कर उसी score में 19 लिखता है
- अंतिम लक्ष्य-चित्र अब भी $p \rightarrow \text{score} \leftarrow q$ है और अब score = 19 है। आउटपुट: 19 19 7, फिर नई पंक्ति

12 14 7 बताने पर आपने q को पुराने पूर्णांक की अलग कॉपी दे दी, जबकि उसमें पॉइंटर मान की कॉपी हुई थी। 24 24 24 बताने पर saved अपनी कॉपी रखने के बजाय लक्ष्य के अपडेट के साथ बदल रहा है। ट्रेस में पॉइंटर मान और सहेजे गए पूर्णांक मान की अलग प्रविष्टियाँ रखने से दोनों गलतियाँ दूर होती हैं। साझा ऑब्जेक्ट अब भी एक ही पूर्णांक है; दो अलग पॉइंटर चरों से लिखने पर उसके दो अलग संस्करण नहीं बनते।

हल किया हुआ उदाहरण 2 एक पॉइंटर का लक्ष्य बदलें और दूसरे का बनाए रखें

अब दो पूर्णांक ऑब्जेक्ट हैं। p में दोबारा असाइनमेंट करने पर q का तीर अपने-आप नहीं बदलना चाहिए।

```
#include <stdio.h>

int main(void)
{
    int left = 3;
    int right = 11;
    int *p = &left;
    int *q = p;
    p = &right;
    *p = *q + 2;
    *q = *p + 4;
    printf("%d %d %d %d\n", left, right, *p, *q);
    return 0;
}
```

इस चरण के बाद	left	right	p का लक्ष्य	q का लक्ष्य
सभी घोषणाएँ	3	11	left	left
p = &right	3	11	right	left
*p = *q + 2	3	5	right	left
*q = *p + 4	9	5	right	left

शुरुआत में $p \rightarrow \text{left} \leftarrow q$ है और right अलग है। लक्ष्य बदलने के बाद चित्र $p \rightarrow \text{right}$ और $q \rightarrow \text{left}$ है। अभी कोई पूर्णांक नहीं बदला है।

पहला लेखन q के रास्ते 3 पढ़ता है, उसमें 2 जोड़ता है और p के रास्ते right में 5 लिखता है। अगला लेखन p के रास्ते नया 5 पढ़ता है, उसमें 4 जोड़ता है और q के रास्ते left में 9 लिखता है। आउटपुट: 9 5 5 9, फिर नई पंक्ति।

यदि $p = \&\text{right}$ पर आपने दोनों तीर बदल दिए, तो $q = p$ को पॉइंटर चरों के बीच स्थायी जुड़ाव मान लिया। वह एक बार हुई मान की कॉपी थी। यदि लक्ष्य बदलते समय left को 11 कर दिया, तो पॉइंटर असाइनमेंट को $*p = \text{right}$ समझ लिया। दोनों के गंतव्य अलग हैं। बाद का लेखन किस पूर्णांक तक पहुँचेगा, यह वर्तमान लक्ष्य तय करते हैं, पॉइंटर चर के नाम नहीं।

नल मान स्पष्ट रूप से बताता है कि लक्ष्य नहीं है

हमारे उदाहरणों में NULL इस्तेमाल करते समय `<stddef.h>` शामिल करें। `int *p = NULL;` पॉइंटर को ज्ञात नल मान से इनिशियलाइज़ करता है। हम चित्र में $p \rightarrow$ कोई लक्ष्य नहीं लिखते हैं। पॉइंटर चर मौजूद है, लेकिन उसके इस मान से कोई ऐसा पूर्णांक ऑब्जेक्ट नहीं मिलता जिसे $*p$ से पढ़ा या लिखा जा सके।

नल पॉइंटर और शून्य रखे हुए पूर्णांक का पॉइंटर अलग हैं। `int value = 0;` और `int *p = &value;` के बाद चित्र `p → value` है तथा `value = 0` है। `*p` पढ़ना सही है और उससे 0 मिलता है। `p = NULL` होने पर `*p` से पढ़ना या उसमें लिखना गलत है, चाहे आप कोई भी संख्या पढ़ना या रखना चाहते हों। `NULL` से यह निष्कर्ष न निकालें कि सभी बाइट शून्य होंगे या कोई खास भौतिक एड्रेस होगा।

जब यह ज्ञात हो कि `p` या तो नल है या किसी उपयुक्त जीवित पूर्णांक का पॉइंटर है, तब `if (p != NULL)` अपनी `body` में डीरेफ़रेंस से पहले गार्ड लगा सकता है। नल होने पर `body` छोड़ दी जाती है; सही लक्ष्य होने पर `body` उस लक्ष्य का उपयोग कर सकती है। लेखन छोड़ देने से लक्ष्य नहीं बनता और आवश्यक अपडेट भी नहीं होता।

`p != NULL && *p > 0` में भी यही क्रम महत्वपूर्ण है: उसी ज्ञात-मान वाली शर्त के अंतर्गत, बाईं तुलना नल होने पर दायें ऑपरेंड चलने नहीं देती। ऑपरेंड उलट देने पर नल की जाँच से पहले पढ़ने की कोशिश होगी। नल से तुलना किसी भी मनमाने पॉइंटर की सुरक्षा जाँचने का सामान्य तरीका नहीं है। विशेष रूप से, यह बिना इनिशियलाइज़ किए पॉइंटर या उस पॉइंटर को ठीक नहीं करती जिसका लक्ष्य समाप्त हो चुका हो।

लक्ष्य जीवित रहना चाहिए

इन उदाहरणों का साधारण स्थानीय `int`, `automatic storage duration` वाला है। यदि वह भीतर के ब्लॉक का है, तो उस ब्लॉक से निष्पादन बाहर निकलते ही उसका जीवनकाल समाप्त हो जाता है। बाहरी ब्लॉक का पॉइंटर चर उस पूर्णांक से अधिक समय तक जीवित रह सकता है जिसे वह पहले पहचानता था। पॉइंटर चर बचा रहने से लक्ष्य ऑब्जेक्ट जीवित नहीं रहता।

लक्ष्य के ट्रेस में बंद होने वाले ब्रेस पर भीतर के ऑब्जेक्ट को समाप्त चिह्नित करें। जिस पॉइंटर में अब भी उसी ऑब्जेक्ट की ओर पॉइंटर था, उसका मान C11 के अनुसार उस सीमा पर अनिर्धारित हो जाता है। वह अपने-आप नल नहीं बनता। उसे उपयोग योग्य तीर की तरह आगे न दिखाएँ, डीरेफ़रेंस न करें और पुराने मान की `NULL` से तुलना करके उसे सही साबित करने की कोशिश न करें।

एक सुरक्षित तरीका है कि ब्लॉक छोड़ने से पहले लक्ष्य का काम पूरा करें और आवश्यक पूर्णांक परिणाम को अधिक समय तक जीवित रहने वाले चर में कॉपी कर लें। यदि अधिक समय तक रहने वाले पॉइंटर की अब जरूरत नहीं है, तो उसके लक्ष्य के जीवित रहते ही उसमें `NULL` रख दें। एक पॉइंटर साफ करने से उस ऑब्जेक्ट के दूसरे पॉइंटर साफ नहीं होते; हर पॉइंटर का अपना मान होता है। एक और तरीका लक्ष्य को ऐसे ब्लॉक में रखना है जो उसके सभी जरूरी उपयोगों तक चले। नीचे का अभ्यास पहला तरीका अपनाता है।

जीवनकाल और वर्तमान संख्या दो अलग बातें हैं। स्रोत ऑब्जेक्ट समाप्त होने के बाद भी पूर्णांक की कॉपी सही रह सकती है; पॉइंटर की कॉपी उस ऑब्जेक्ट को बचाए नहीं रखती। यह अंतर समझने के लिए “पुराने बाइट अब भी वहीं होंगे” जैसी धारणा की जरूरत नहीं है।

हल देखने से पहले अभ्यास करें

नई स्थितियों के इन चार मौलिक अभ्यासों के अंक नहीं हैं। इनमें सीखी हुई समझ लागू करें। निश्चित व्यवहार वाले प्रोग्रामों के लिए आउटपुट, बीच की पूर्णांक अवस्थाएँ और लक्ष्य-चित्र दें। अभ्यास 3 और 4 के असुरक्षित अंशों को बिना कंपाइल या निष्पादित किए वर्गीकृत करें, फिर बताया गया सुधार करें।

अभ्यास 1 बराबर संख्याएँ और नया साझा लक्ष्य

क्या शुरुआत में p और q एक ही ऑब्जेक्ट को पहचानते हैं? दिखाएँ कि q = p पर क्या बदलता है, फिर आउटपुट दें।

```
#include <stdio.h>

int main(void)
{
    int first = 6;
    int second = 6;
    int *p = &first;
    int *q = &second;
    *p = 9;
    q = p;
    *q = *q - 2;
    printf("%d %d %d %d\n", first, second, *p, *q);
    return 0;
}
```

अभ्यास 2 पूर्णांक की कॉपी और पॉइंटर में दो बदलाव

हर गंतव्य का ट्रेस करें। समझाएँ कि q = p संख्या 10 की कॉपी क्यों नहीं करता और उसके बाद p का लक्ष्य बदलने से q का लक्ष्य क्यों नहीं बदलता।

```
#include <stdio.h>

int main(void)
{
    int red = 4;
    int blue = 10;
    int *p = &red;
    int *q = &blue;
    int saved = *q;
    *p = *q;
    q = p;
    p = &blue;
    *q = *q - 3;
    *p = saved + *q;
    printf("%d %d %d %d %d\n", red, blue, saved, *p, *q);
    return 0;
}
```

अभ्यास 3 आवश्यक लक्ष्य दें

उद्देश्य p के रास्ते लिखकर total के मान 8 में 3 जोड़ना है। मान लें कि `<stddef.h>` शामिल है और यह अंश main के भीतर है। यह जानबूझकर असुरक्षित है; इसे केवल पढ़ें।

```
int total = 8;
int *p = NULL;
*p = total + 3;
```

लेखन को वर्गीकृत करें। p की घोषणा और लेखन के बीच एक पॉइंटर असाइनमेंट डालें, जिससे बताया हुआ अपडेट सही हो जाए। समझाएँ कि मूल लेखन को केवल if (p != NULL) का गार्ड देने से उद्देश्य पूरा क्यों नहीं होगा।

अभ्यास 4 भीतर का ऑब्जेक्ट समाप्त होने के बाद परिणाम रखें

मान लें कि सामान्य हेडर और बाहर का main मौजूद हैं। यह अंश जानबूझकर असुरक्षित है; इसे कंपाइल या निष्पादित न करें।

```
int saved = 0;
int *p = NULL;
{
    int local = 9;
    p = &local;
    *p = *p + 2;
    saved = *p;
}
printf("%d %d\n", saved, *p);
```

ब्लॉक के भीतर सही काम का ट्रेस करें। किस सीमा पर p में local की ओर उपयोग योग्य पॉइंटर नहीं रहता? क्या C11 आउटपुट के रूप में 11 11 तय करता है? local और उसका अपडेट भीतर के ब्लॉक में रखते हुए अंश सुधारें: अंतिम पूर्णांक saved में रखें, उस ब्लॉक को छोड़ने से पहले p साफ करें और बाद में केवल saved प्रिंट करें।

पूरे हल और गलत उत्तरों का कारण

अभ्यास 1 का हल

इस चरण के बाद	first	second	p का लक्ष्य	q का लक्ष्य
सभी घोषणाएँ	6	6	first	second
*p = 9	9	6	first	second
q = p	9	6	first	first
*q = *q - 2	7	6	first	first

शुरुआती चित्र $p \rightarrow \text{first}$ और $q \rightarrow \text{second}$ है। रखी संख्याएँ बराबर होने से दो ऑब्जेक्ट एक नहीं हो जाते। पहला लेखन केवल `first` बदलता है। फिर $q = p$ केवल q का लक्ष्य बदलता है: चित्र $p \rightarrow \text{first} \leftarrow q$ हो जाता है और `second` में अब भी 6 है। अंतिम लेखन q के रास्ते 9 पढ़ता है, उसमें से 2 घटाता है और `first` में 7 लिखता है। आउटपुट: 7 6 7 7, फिर नई पंक्ति।

7 7 7 7 दोनों पूर्णांक ऑब्जेक्ट को केवल इसलिए गलत ढंग से जोड़ देता है क्योंकि उनके शुरुआती मान बराबर थे। 9 4 9 4 पॉइंटर असाइनमेंट को छोड़ देता है और q का लक्ष्य अब भी `second` मानता है। पॉइंटर का कॉपी हुआ मान कॉपी होते समय ही दर्ज करना चाहिए।

अभ्यास 2 का हल

इस चरण के बाद	red	blue	saved	p का लक्ष्य	q का लक्ष्य
सभी घोषणाएँ	4	10	10	red	blue
<code>*p = *q</code>	10	10	10	red	blue
<code>q = p</code>	10	10	10	red	red
<code>p = &blue</code>	10	10	10	blue	red
<code>*q = *q - 3</code>	7	10	10	blue	red
<code>*p = saved + *q</code>	7	17	10	blue	red

पहला लेखन पूर्णांक 10 की कॉपी `blue` से `red` में करता है; चित्र $p \rightarrow \text{red}$ और $q \rightarrow \text{blue}$ ही रहता है। फिर $q = p$ पॉइंटर मान की कॉपी करता है, जिससे $p \rightarrow \text{red} \leftarrow q$ मिलता है। वह `*p` या `*q` नहीं पढ़ता। केवल p का लक्ष्य बदलने पर $p \rightarrow \text{blue}$ और $q \rightarrow \text{red}$ मिलता है।

इसलिए घटाव $10 - 3 = 7$ को `red` में लिखता है। अंतिम लेखन सहेजा हुआ पूर्णांक 10 और वर्तमान `red = 7` पढ़ता है, फिर `blue` में 17 लिखता है। आउटपुट: 7 17 10 17 7, फिर नई पंक्ति।

`saved` को 17 कर देना पूर्णांक की कॉपी को एलियस मानना है। घटाव से `blue` बदलना, p के बाद वाले असाइनमेंट के साथ q को भी बदल देना है। `*p = *q` को तीर का बदलाव मानना, लक्ष्य में रखे मान की कॉपी और पॉइंटर मान की कॉपी को मिला देना है। बाई ओर का `*` ही गंतव्य तय करने में महत्वपूर्ण है।

अभ्यास 3 का हल

शुरुआत में `total = 8` और $p \rightarrow$ कोई लक्ष्य नहीं है। दाई ओर की अंकगणित $8 + 3$ सीमा में है, लेकिन उससे लेखन का गंतव्य नहीं मिलता। `*p = ...` के लिए नल पॉइंटर को डीरेफ़रेंस करने पर अपरिभाषित व्यवहार होता है। C11 किसी अपडेट, खास आउटपुट या क्रैश की गारंटी नहीं देता।

लेखन से पहले `p = &total`; डालें। पूरा सुधरा हुआ प्रोग्राम है:

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

int main(void)
{
    int total = 8;
    int *p = NULL;
    p = &total;
    *p = total + 3;
    printf("%d %d\n", total, *p);
    return 0;
}
```

- घोषणाओं के बाद: `total = 8`, `p` → कोई लक्ष्य नहीं
- `p = &total` के बाद: `total = 8`, `p` → `total`
- लेखन 8 पढ़ता है, 11 निकालता है और उसे `total` में रखता है। अब `total = 11`, `p` → `total` है
- प्रिंट होने वाले दोनों व्यंजक उसी जीवित पूर्णांक को पढ़ते हैं। आउटपुट: 11 11, फिर नई पंक्ति

फिर से `p = NULL` करने पर भी लक्ष्य नहीं मिलेगा। `*p = 0` पॉइंटर को इनिशियलाइज़ नहीं कर सकता; वह लक्ष्य में एक और गलत लेखन का प्रयास होगा। मूल लेखन पर `if (p != NULL)` का गार्ड लगाने से इस ज्ञात-नल स्थिति में लेखन सुरक्षित रूप से छोड़ दिया जाएगा, लेकिन `total = 8` ही रहेगा। इसलिए माँगा गया अपडेट पूरा नहीं होगा। पहले `p = &total` करना केवल ऑपरेशन छोड़ने के बजाय उसका लक्ष्य देता है।

अभ्यास 4 का हल

पूरे असुरक्षित अंश का संख्यात्मक आउटपुट तय किए बिना उसके सही शुरुआती भाग का ट्रेस करें:

जाँच-बिंदु	saved	भीतर का local	p की अवस्था
भीतर के ब्लॉक से पहले	0	अभी जीवित नहीं	ज्ञात नल
local के इनिशियलाइज़ेशन के बाद	0	9	ज्ञात नल
<code>p = &local</code> के बाद	0	9	<code>p</code> → local
<code>*p = *p + 2</code> के बाद	0	11	<code>p</code> → local
<code>saved = *p</code> के बाद	11	11	<code>p</code> → local
भीतर का ब्लॉक छोड़ने के बाद	11	जीवनकाल समाप्त	अनिर्धारित; पुराना लक्ष्य उपयोग योग्य नहीं

भीतर पढ़ना और लिखना सही है। बंद होने वाला ब्रेस `local` का जीवनकाल समाप्त करता है, भले ही `saved` और पॉइंटर चर `p` बाहरी ब्लॉक में जीवित रहें। बाद का `*p` उस समाप्त ऑब्जेक्ट को नहीं पढ़ सकता। अंश में अपरिभाषित व्यवहार है, इसलिए C11 11.11 तय नहीं करता और पहले प्रिंट होने वाले 11 की भी गारंटी नहीं देता। सही शुरुआती भाग का ठीक ट्रेस, अपरिभाषित व्यवहार वाले पूरे निष्पादन के आउटपुट की गारंटी नहीं है।

बताया गया सुधार है:

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

int main(void)
{
    int saved = 0;
    int *p = NULL;
    {
        int local = 9;
        p = &local;
        *p = *p + 2;
        saved = *p;
        p = NULL;
    }
    printf("%d\n", saved);
    return 0;
}
```

सुधार में `saved = 11` होने तक वही सही अवस्थाएँ आती हैं। फिर, जब `local` मान 11 के साथ अब भी जीवित है, `p = NULL` पॉइंटर मान बदलता है: `p → कोई लक्ष्य नहीं`। इससे `local` या `saved` नहीं बदलता। भीतर का ब्लॉक समाप्त होने पर `p` ज्ञात नल ही रहता है और `saved` में 11 बना रहता है। प्रिंट केवल अधिक समय तक जीवित रहने वाले पूर्णांक की कॉपी पढ़ता है। आउटपुट: 11, फिर नई पंक्ति।

मूल प्रिंट से पहले नल की जाँच जोड़ना सुधार नहीं है: जीवनकाल की सीमा के बाद मूल `p` अनिर्धारित है, न कि नल और जीवित लक्ष्य के बीच कोई ज्ञात विकल्प। ब्लॉक समाप्त होने से पहले `p` को दूसरे पॉइंटर में कॉपी करने पर उस दूसरे पॉइंटर में भी वही जीवनकाल की समस्या रहेगी। पूर्णांक परिणाम रखना और समाप्त ऑब्जेक्ट की ओर एक और पॉइंटर रखना अलग बातें हैं।

त्वरित संदर्भ

- `int *p = &x`; पॉइंटर चर बनाता है और उसे जीवित पूर्णांक `x` की ओर इनिशियलाइज़ करता है
- पूर्णांक मान और पॉइंटर लक्ष्य अलग दर्ज करें; तीर प्रतीक हैं, मशीन एड्रेस नहीं
- `*p` वर्तमान लक्ष्य चुनता है। पढ़ने पर लक्ष्य का पूर्णांक मिलता है; उसमें असाइनमेंट करने पर वही पूर्णांक बदलता है
- `p = &y` किसी भी पूर्णांक को बदले बिना `p` का लक्ष्य बदलता है
- `q = p` वर्तमान पॉइंटर मान की कॉपी करता है। इससे साझा लक्ष्य बन सकता है, लेकिन `p` के भविष्य के बदलावों के साथ `q` नहीं बदलता
- `int saved = *p`; पूर्णांक की कॉपी करता है; लक्ष्य में बाद के अपडेट से `saved` नहीं बदलता
- ज्ञात नल पॉइंटर का डीरेफ़रेंस योग्य लक्ष्य नहीं होता। शून्य रखे हुए पूर्णांक का पॉइंटर अलग है
- नल गार्ड ज्ञात और मान्य पॉइंटर मान की शर्त पर केवल नल को बाहर करता है; वह सामान्य पॉइंटर वैधता सिद्ध नहीं करता

- automatic लक्ष्य अपने हर अभिगम तक जीवित रहना चाहिए। उसके जीवनकाल के अंत से पहले आवश्यक पूर्णांक परिणाम सहेजें
- गलत डीरेफ़रेंस वर्गीकरण के प्रश्न हैं; दिख गया आउटपुट उन्हें सही नहीं बना सकता

आगे बढ़ने से पहले

$p = q$ और $*p = *q$ के अलग गंतव्य समझाएँ। p को q में कॉपी करने और फिर केवल p का लक्ष्य बदलने के बाद दोनों तीर बनाएँ। अंत में समझाएँ कि पूर्णांक सहेजने से ब्लॉक की सीमा के बाद भी परिणाम क्यों रह सकता है, जबकि उस ब्लॉक के स्थानीय पूर्णांक का पॉइंटर सहेजने से लक्ष्य क्यों नहीं बचता। आगे हम कॉपी किए आर्ग्युमेंट का मॉडल बनाए रखते हुए इन लक्ष्य-रिकॉर्ड को फ़ंक्शन कॉल से जोड़ेंगे।

स्रोत टिप्पणी

उदाहरण, चित्र, ट्रेस, अभ्यास और समझाने वाले वाक्य मौलिक हैं। अर्थ संबंधी तथ्यों की जाँच [WG14 N1570 C11 समिति-ड्राफ्ट](#) से की गई: पॉइंटर और स्केलर प्रकार §6.2.5p20–21; एड्रेस लेना और डीरेफ़रेंस §6.5.3.2p3–4; पॉइंटर घोषणाएँ §6.7.6.1p1; स्केलर इनिशियलाइज़ेशन §6.7.9p10–11; असाइनमेंट §6.5.16.1p2; नल पॉइंटर §6.3.2.3p3 और NULL §7.19p3; पॉइंटर और नल की तुलना §6.5.9p2, p5–6; शॉर्ट सर्किट §6.5.13p4; automatic जीवनकाल §6.2.4p2, p5–6।

C11 पढ़ाने के लिए चुना गया संस्करण है, GATE की बताई अनिवार्यता नहीं। यह पाठ स्केलर पॉइंटर के लक्ष्य का सीमित परिचय है, सभी पॉइंटर, पूरी C भाषा या पूरा GATE CS नहीं।

Lesson 04 · English

Array parameters and caller-visible changes

A copied argument can still identify shared storage

The foundation lesson showed that changing an integer parameter does not change the caller's integer variable. The pointer lesson then showed that copying a pointer value can leave two routes to one object. Put those ideas together: a helper can receive a copied pointer value and use it to change an element of the caller's array.

There is no change to C's copied-argument model. What changes is the kind of value being supplied and the object selected by a later write.

By the end of this lesson you will trace an array call with an explicit length, distinguish changed elements from changed helper parameters, and state the assumptions that make a helper's accesses valid. You will also distinguish a helper's return value from its caller-visible writes.

Prerequisites: the accepted scalar function/copy lesson, arrays and valid indices, prefix scans, and pointer target records. We do not yet use pointer arithmetic, `sizeof`, allocation or pointers to pointers.

C11 is our teaching convention. All real arrays shown have positive constant sizes and initialized `int` elements. Defined calls use live suitable storage and valid selected lengths. Every arithmetic result fits within -32767 through 32767 . Each complete program is independent. The explicitly unsafe practice fragment is for classification only and must not be compiled or executed.

Read the call and the parameter separately

In the caller, `int marks[4] = {3, 1, 4, 2};` creates a real four-element array. In the call `add_to_prefix(marks, used, 2)`, the expression `marks` supplies a pointer to its first element. The array object stays in the caller; the call does not make an element-by-element copy of it.

In the helper's parameter list, `int values[]` declares a parameter whose type is adjusted to a pointer to `int`. For the plain parameter forms used here, it can also be written `int *values`. This parameter form is different from the caller's actual array declaration. The helper receives a pointer value, not a new array whose size can be read from the empty brackets.

`values[i]` accesses the corresponding element through that pointer. For the calls here, the pointer identifies the first element of the caller's array, and the caller supplies the separate valid `length`. A pointer value does not carry an automatic count of available elements. We will study array-size expressions next; do not invent a length from the helper's parameter spelling.

The integer arguments still work as before. `length` receives the current value of the caller's `used`, and `delta` receives 2. They are separate helper parameters. The argument expressions here have no side effects, so their evaluation order cannot change the result.

A helper that returns no value

In `void add_to_prefix(...)`, `void` says that the call supplies no result value. We use it as a statement: `add_to_prefix(marks, used, 2);`. Its purpose is the writes made by its body, not a returned number to store in a variable.

Reaching the closing brace of this `void` helper returns control to the caller. The caller continues at the next statement. This is different from the integer helpers that explicitly return a number, and neither kind of return is itself a print operation.

Worked example 1 Change a prefix but not the caller's length

Which of the caller's five printed integers change?

```
#include <stdio.h>

void add_to_prefix(int values[], int length, int delta)
{
    for (int i = 0; i < length; i++) {
        values[i] += delta;
    }
    length = 0;
}

int main(void)
{
    int marks[4] = {3, 1, 4, 2};
    int used = 3;
    add_to_prefix(marks, used, 2);
    printf("%d %d %d %d | %d\n",
        marks[0], marks[1], marks[2], marks[3], used);
    return 0;
}
```

Before the call, draw the caller's real array once: `marks = {3, 1, 4, 2}` and caller `used = 3`. On entry to the helper, add separate parameter records: `values → marks[0]`, `length = 3`, `delta = 2`. Do not draw a second four-element array for `values`.

Helper checkpoint	Caller array marks	Helper length	Caller used
Before first loop test	{3, 1, 4, 2}	3	3
After index 0 body	{5, 1, 4, 2}	3	3
After index 1 body	{5, 3, 4, 2}	3	3
After index 2 body	{5, 3, 6, 2}	3	3
After failed test at index 3	{5, 3, 6, 2}	3	3
After <code>length = 0</code>	{5, 3, 6, 2}	0	3

The loop accesses indices 0, 1 and 2. Each store reaches the corresponding caller element. The fourth element remains 2 because it is outside the selected prefix. There are three bodies and four loop tests.

The final `length = 0` changes only the helper's integer parameter. It neither erases the array nor assigns to caller used. After the helper ends, the caller prints `5 3 6 2 | 3`, followed by a newline.

The element changes are already writes to the original storage while the helper executes. They do not wait for a special “copy back” operation at return. The helper's own parameter objects finish their lifetime when the call ends; the caller's array remains alive in `main` and retains its new values.

An unchanged array would incorrectly assume that the helper worked on a private array copy. A last element of 4 would process one element beyond the selected prefix. A caller used of 0 would copy a change back from a separate integer parameter. Each mistake confuses a different object or boundary.

Put the access contract beside the helper

For `add_to_prefix`, the calls in this lesson require:

- A pointer to the first element of a live initialized integer array
- A `length` from 0 through that array's actual element count
- Permission, as part of this helper's intended behavior, to change the selected elements
- Every evaluated addition to remain representable in `int`

With `length = 0`, the first loop test fails and no element is read or written. We still supply the real array; we are not introducing null-array conventions or zero-sized arrays. With `length 4` in the worked program, all four elements would be updated. `Length 5` would exceed its four-element capacity and is not an allowed call.

The helper cannot infer or check real capacity merely from the pointer parameter and the supplied `length`. Writing an incorrect `length` does not enlarge the caller's array. These are explicit preconditions, not checks C silently performs on every call.

Worked example 2 Retarget the local parameter

Now pass two scalar targets. Does assigning a new value to the helper's pointer parameter also retarget the caller's pointer variable?

```
#include <stdio.h>

void retarget_then_write(int *target, int *other)
{
    target = other;
    *target = 9;
}

int main(void)
{
    int left = 2;
    int right = 7;
    int *selected = &left;
    retarget_then_write(selected, &right);
    printf("%d %d %d\n", left, right, *selected);
    return 0;
}
```

- Caller before the call: left = 2, right = 7, and selected → left
- Helper entry: its separate target → left and other → right. The pointer argument values have been copied into those parameters
- target = other changes the helper's target to identify right. Caller selected still identifies left; both integers are unchanged
- *target = 9 now writes into right, making it 9
- After return, caller selected → left still reads 2. Output: 2 9 2, followed by a newline

There are three pointer variables in the trace: caller selected and helper parameters target and other. Sharing a target does not merge those variables. target = other changes one local pointer value. *target = 9 changes the integer selected by that local pointer value.

The wrong result 2 9 9 lets the helper's pointer reassignment move caller selected. The wrong result 9 7 9 ignores the retargeting before the store. The wrong result 2 7 2 misses the actual store to caller right. Drawing each pointer variable separately resolves all three errors.

We are not teaching how to reassign the caller's pointer variable from a helper here. That needs a different interface. Do not treat an ordinary pointer argument as an automatic reference to the caller's pointer variable.

State what the function returns and what it changes

A function that only sums elements may return an integer without writing the array. Another function may change elements and return nothing. A function can also both modify storage and return a value, but neither behavior follows solely from the name or the presence of brackets in a parameter list. Read the body and the stated contract.

For each call, keep four pieces of evidence: the argument values, the separate helper parameters, the caller objects reached by writes, and any returned value. Use distinct caller and helper state records, exactly as in the scalar function lesson, adding target arrows where needed.

Practice before the solutions

These four original exercises are unscored. Keep call order, target identity and selected length visible. Question 4 deliberately violates a bound and is for classification and repair only.

Practice 1 Read-only calculation followed by a mutation

Give the output and final array. Which helper changes the array, and does the saved return value before change afterward?

```
#include <stdio.h>

int sum_prefix(int values[], int length)
{
    int total = 0;
    for (int i = 0; i < length; i++) {
        total += values[i];
    }
    return total;
}

void clear_first(int values[], int length)
{
    if (length > 0) {
        values[0] = 0;
    }
}

int main(void)
{
    int a[3] = {3, 4, 6};
    int before = sum_prefix(a, 3);
    clear_first(a, 2);
    int after = sum_prefix(a, 2);
    printf("%d %d %d\n", before, after, a[2]);
    return 0;
}
```

Use valid selected lengths. “Read-only” here describes what `sum_prefix` actually does; it is not a new type qualifier.

Practice 2 A local length change affects the helper's own loop

This program is defined for the shown four-element array, but the intended helper contract is to fill exactly the requested prefix. Find the actual output, identify the contract error and repair it by deleting one assignment. The caller's requested must stay unchanged.

```
#include <stdio.h>

void fill_prefix(int values[], int length, int value)
{
    length = 2;
    for (int i = 0; i < length; i++) {
        values[i] = value;
    }
}

int main(void)
{
    int a[4] = {2, 4, 6, 8};
    int requested = 1;
    fill_prefix(a, requested, 9);
    printf("%d %d %d %d | %d\n",
        a[0], a[1], a[2], a[3], requested);
    return 0;
}
```

Practice 3 Swap target values and consider a shared target

Trace both calls. The second call supplies the same target twice. Explain whether it creates a problem in this particular helper, and give the output.

```
#include <stdio.h>

void swap_values(int *left, int *right)
{
    int saved = *left;
    *left = *right;
    *right = saved;
}

int main(void)
{
    int x = 4;
    int y = 9;
    swap_values(&x, &y);
    swap_values(&x, &x);
    printf("%d %d\n", x, y);
    return 0;
}
```

Both supplied targets are live initialized integers. Do not assume that all pointer-based helpers either require distinct targets or always support aliasing; inspect this one's statements.

Practice 4 The caller supplies too large a length

The following fragment is deliberately unsafe. Assume the shown call occurs inside `main` and the helper is defined beforehand. Do not compile or execute it.

```
void add_one(int values[], int length)
{
    for (int i = 0; i < length; i++) {
        values[i] += 1;
    }
}

int a[2] = {3, 7};
add_one(a, 3);
```

Identify the first invalid index that the loop would attempt. Does the helper's `values[]` parameter create enough elements for the supplied length? Repair only the call so the helper increments the whole real array, then give the repaired output for its two elements.

Full solutions and wrong-turn feedback

Practice 1 solution

The first `sum_prefix` call receives access to `a` and length 3. Its local totals are 3, 7 and 13 as indices 0, 1 and 2 are visited. It returns 13 without changing any element; caller `before = 13`.

`clear_first(a, 2)` receives a copied pointer value and local length 2. Its condition is true, so it writes 0 to `a[0]`. It does not clear both selected elements: its body contains only a store at index 0. The caller's array is now {0, 4, 6}.

The second `sum` uses length 2, so its fresh local total becomes 0 then 4. It returns 4 into `after`. The saved caller integer `before` stays 13. Output: 13 4 6, then a newline.

4 4 6 makes a saved return value follow later array changes. 13 10 6 sums the whole array despite the second length being 2. 13 0 6 imagines that `clear_first` fills the whole prefix, although the body writes only its first element. Function contracts and actual bodies must agree; names alone cannot supply missing operations.

Practice 2 solution

On entry, the helper's length is 1 and caller requested is 1. The local assignment changes helper length to 2. The loop therefore writes 9 at indices 0 and 1. Both indices exist in this four-element array, so this displayed run is defined. The final array is {9, 9, 6, 8} and caller requested remains 1. Output: 9 9 6 8 | 1, then a newline.

The helper has violated the intended selected-prefix contract even though this run stays inside allocated storage. Defined behavior and intended behavior are separate questions. The copied parameter rule does not prevent a helper from using its changed local length in later statements.

Delete `length = 2;`. The complete repaired program is:

```
#include <stdio.h>

void fill_prefix(int values[], int length, int value)
{
    for (int i = 0; i < length; i++) {
        values[i] = value;
    }
}

int main(void)
{
    int a[4] = {2, 4, 6, 8};
    int requested = 1;
    fill_prefix(a, requested, 9);
    printf("%d %d %d %d | %d\n",
           a[0], a[1], a[2], a[3], requested);
    return 0;
}
```

The repaired helper keeps `length` 1, writes only `a[0]`, then stops at the failed test with index 1. It prints `9 4 6 8 | 1`, followed by a newline. Setting caller `requested` to 2 would change what the caller asked for rather than repair the helper. Treating the original assignment as a change to caller `requested` would wrongly predict a final `| 2`.

Practice 3 solution

First call: helper `left` → `x`, `right` → `y`; saved copies `x = 4`. The store `*left = *right` makes `x = 9`, then `*right = saved` makes `y = 4`. No pointer parameter is retargeted. After return, caller values are `x = 9`, `y = 4`.

Second call: both helper parameters identify `x`. Its new local saved is 9. The first store reads and writes the same `x`, leaving 9. The second stores the saved 9 there, again leaving it unchanged. These are separate, sequenced statements, not conflicting unsequenced updates. This specific helper permits the two supplied targets to be the same object.

Output: `9 4`, followed by a newline. Predicting `4 9` for the final pair invents another exchange with `y`, which the second call never references. Predicting an automatic error merely because the pointers share a target overgeneralizes aliasing: safety depends on the operations and the helper's contract.

Practice 4 solution

The caller's array has exactly two elements, at indices 0 and 1. The helper receives access to that array and a separate integer `length` 3. At index 2, `values[2] += 1` would attempt an out-of-bounds element access. The parameter declaration creates no third element. The fragment has undefined behavior; do not assign it a C11-prescribed output or a guaranteed “partly updated” final array.

Use `add_one(a, 2);`. A complete safe program is:

```
#include <stdio.h>

void add_one(int values[], int length)
{
    for (int i = 0; i < length; i++) {
        values[i] += 1;
    }
}

int main(void)
{
    int a[2] = {3, 7};
    add_one(a, 2);
    printf("%d %d\n", a[0], a[1]);
    return 0;
}
```

The valid call changes 3 to 4 and 7 to 8, then the loop test fails at index 2 without accessing it. Output: 4 8, followed by a newline. Using length 1 would be safe but would not increment the whole real array. Keeping length 3 because it was passed “by value” confuses copying a count with validating it.

Quick reference

- In the calls here, an array argument supplies a pointer to its first element; the real array is not copied into the helper
- A plain `int values[]` parameter is adjusted to a pointer parameter; it is not a new local array
- Pass a valid selected length explicitly. A pointer value does not carry an automatic element count
- Parameters receive copied values, including when those values are pointers
- A write to `values[i]` or `*target` reaches the identified caller object when that object is shared and alive
- Reassigning a helper's pointer parameter does not reassign the caller's pointer variable
- Changing a helper's integer length changes its later local decisions, not the caller's length variable
- A void helper supplies no returned result value; its writes can still be visible to the caller
- Distinguish return values, parameter updates and writes to shared objects in every call trace
- State capacity, lifetime, allowed mutation and arithmetic assumptions; neither a parameter name nor a supplied length validates them

Before moving on

Explain how a copied pointer value can allow an element change while a copied length remains separate. Then compare a pointer reassignment with a store through that pointer. Next we will move a pointer within an array and distinguish an actual array's element count from the size of a pointer parameter, without assuming a machine's byte sizes.

Source note

The explanations, code, traces and exercises are original. The first two program designs develop this module's earlier original planning prototypes. C facts were checked against the WG14 N1570 C11 committee draft: array conversion §6.3.2.1p3; parameter adjustment §6.7.6.3p7; copied arguments §6.5.2.2p4; dereference §6.5.3.2p4; indexed access and bounds §§6.5.2.1p2, 6.5.6p8; assignment §6.5.16.1p2; void expressions §6.3.2.2p1 and function completion §6.9.1; automatic lifetime §6.2.4. Reference: <https://open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>

C11 is a teaching convention, not a claimed GATE language-version requirement. This is a bounded lesson on calls and shared array storage, not full C or GATE CS coverage.

Lesson 04 · हिन्दी

ऐरे पैरामीटर और कॉलर में दिखने वाले बदलाव

कॉपी हुआ आर्गुमेंट भी साझा स्टोरेज बता सकता है

बुनियादी पाठ में देखा कि पूर्णांक पैरामीटर बदलने से कॉलर का पूर्णांक चर नहीं बदलता। पॉइंटर वाले पाठ में फिर देखा कि पॉइंटर मान की कॉपी से एक ऑब्जेक्ट तक दो रास्ते बने रह सकते हैं। दोनों बातें जोड़ें: helper को पॉइंटर के मान की कॉपी मिल सकती है और वह उससे कॉलर के ऐरे का तत्व बदल सकता है।

C में आर्गुमेंट के मान की कॉपी बनने वाला तरीका नहीं बदला है। केवल दिए जाने वाले मान का प्रकार और बाद के लेखन से चुना जाने वाला ऑब्जेक्ट अलग है।

इस पाठ के अंत तक आप स्पष्ट लंबाई के साथ ऐरे कॉल का ट्रेस करेंगे, बदले तत्वों को बदले helper-पैरामीटरों से अलग रखेंगे और helper के अभिगम सही होने की शर्तें बताएँगे। आप फ़ंक्शन से लौटे मान को कॉलर में दिखने वाले लेखन से भी अलग पहचानेंगे।

पहले से आवश्यक बातें: स्वीकृत scalar फ़ंक्शन/कॉपी पाठ, ऐरे और सही इंडेक्स, शुरुआती हिस्से की जाँच तथा पॉइंटर के लक्ष्य का रिकॉर्ड। अभी पॉइंटर अंकगणित, sizeof, allocation या पॉइंटर के पॉइंटर का उपयोग नहीं है।

C11 पढ़ाने के लिए हमारा चुना संस्करण है। दिखाए गए सभी वास्तविक ऐरे धनात्मक स्थिर आकार के हैं और उनके int तत्व इनिशियलाइज़ हैं। निश्चित व्यवहार वाली कॉल में जीवित उपयुक्त स्टोरेज और सही चुनी लंबाई दी जाती है। हर अंकगणितीय परिणाम -32767 से 32767 के भीतर है। हर पूरा प्रोग्राम अलग उदाहरण है। स्पष्ट असुरक्षित अभ्यास-अंश केवल वर्गीकरण के लिए है; उसे कंपाइल या निष्पादित न करें।

कॉल और पैरामीटर को अलग पढ़ें

कॉलर में `int marks[4] = {3, 1, 4, 2};` वास्तविक चार तत्वों वाला ऐरे बनाता है। कॉल `add_to_prefix(marks, used, 2)` में व्यंजक `marks` उसके पहले तत्व का पॉइंटर देता है। ऐरे ऑब्जेक्ट कॉलर में ही रहता है; कॉल उसकी तत्व-दर-तत्व कॉपी नहीं बनाती।

helper की पैरामीटर सूची में `int values[]` ऐसा पैरामीटर घोषित करता है जिसका प्रकार int के पॉइंटर में समायोजित होता है। यहाँ इस्तेमाल हुए साधारण पैरामीटर रूप में इसे `int *values` भी लिख सकते हैं। यह पैरामीटर रूप कॉलर की वास्तविक ऐरे-घोषणा से अलग है। helper को पॉइंटर मान मिलता है, नया ऐरे नहीं जिसकी लंबाई खाली कोष्ठकों से पढ़ी जा सके।

`values[i]` उस पॉइंटर के रास्ते संबंधित तत्व तक पहुँचता है। यहाँ की कॉल में पॉइंटर कॉलर के ऐरे का पहला तत्व बताता है और कॉलर अलग से सही length देता है। पॉइंटर मान में उपलब्ध तत्वों की गिनती अपने-आप नहीं जुड़ी होती। अगले पाठ में ऐरे के आकार वाले व्यंजक पढ़ेंगे; helper के पैरामीटर की लिखावट देखकर लंबाई न गढ़ें।

पूर्णांक आर्गुमेंट पहले जैसे ही काम करते हैं। length को कॉलर के used का वर्तमान मान मिलता है और delta को 2। वे helper के अलग पैरामीटर हैं। यहाँ आर्गुमेंट व्यंजकों में कोई side effect नहीं है, इसलिए उनका मूल्यांकन-क्रम परिणाम नहीं बदल सकता।

ऐसा helper जो कोई मान वापस नहीं देता

`void add_to_prefix(...)` में `void` कहता है कि कॉल कोई परिणाम-मान नहीं देती। हम उसे स्टेटमेंट की तरह लेते हैं: `add_to_prefix(marks, used, 2);`। उसका उद्देश्य body के लेखन हैं, किसी चर में रखने के लिए लौटने वाली संख्या नहीं।

इस void helper के अंतिम ब्रेस तक पहुँचने पर नियंत्रण कॉलर में लौटता है। कॉलर अगला स्टेटमेंट चलाता है। यह उन पूर्णांक helper से अलग है जो स्पष्ट रूप से संख्या लौटाते हैं; दोनों तरह का लौटना अपने-आप प्रिंट करना नहीं है।

हल किया हुआ उदाहरण 1 शुरुआती हिस्सा बदलें, कॉलर की लंबाई नहीं

कॉलर के प्रिंट होने वाले पाँच पूर्णांकों में कौन-से बदलते हैं?

```
#include <stdio.h>

void add_to_prefix(int values[], int length, int delta)
{
    for (int i = 0; i < length; i++) {
        values[i] += delta;
    }
    length = 0;
}

int main(void)
{
    int marks[4] = {3, 1, 4, 2};
    int used = 3;
    add_to_prefix(marks, used, 2);
    printf("%d %d %d %d | %d\n",
           marks[0], marks[1], marks[2], marks[3], used);
    return 0;
}
```

कॉल से पहले कॉलर के वास्तविक ऐरे का एक ही चित्र बनाएँ: marks = {3, 1, 4, 2} और कॉलर का used = 3। helper में प्रवेश पर अलग पैरामीटर रिकॉर्ड जोड़ें: values → marks[0], length = 3, delta = 2। values के लिए दूसरा चार-तत्वों वाला ऐरे न बनाएँ।

helper का बिंदु	कॉलर का ऐरे marks	helper का length	कॉलर का used
पहली लूप-जाँच से पहले	{3, 1, 4, 2}	3	3
इंडेक्स 0 की body के बाद	{5, 1, 4, 2}	3	3
इंडेक्स 1 की body के बाद	{5, 3, 4, 2}	3	3
इंडेक्स 2 की body के बाद	{5, 3, 6, 2}	3	3
इंडेक्स 3 पर गलत जाँच के बाद	{5, 3, 6, 2}	3	3
length = 0 के बाद	{5, 3, 6, 2}	0	3

लूप इंडेक्स 0, 1 और 2 तक जाता है। हर लेखन कॉलर के संबंधित तत्व तक पहुँचता है। चौथा तत्व 2 रहता है, क्योंकि वह चुने शुरुआती हिस्से के बाहर है। तीन body और चार लूप-जाँच हैं।

अंतिम `length = 0` केवल `helper` का पूर्णांक पैरामीटर बदलता है। वह न ऐरे मिटाता है, न कॉलर के `used` में असाइन करता है। `helper` समाप्त होने पर कॉलर `5 3 6 2 | 3` और फिर नई पंक्ति प्रिंट करता है।

`helper` के चलते समय ही तत्वों के बदलाव मूल स्टोरेज में लिखे जा चुके होते हैं। वे लौटते समय किसी खास “copy back” क्रिया की प्रतीक्षा नहीं करते। कॉल समाप्त होने पर `helper` के अपने पैरामीटर ऑब्जेक्ट का जीवनकाल समाप्त होता है; कॉलर का ऐरे `main` में जीवित रहता है और नए मान रखता है।

ऐरे को अपरिवर्तित बताना गलत रूप से मानता है कि `helper` ने ऐरे की निजी कॉपी पर काम किया। अंतिम तत्व को 4 करना चुने शुरूआती हिस्से से एक अतिरिक्त तत्व संसाधित करता है। कॉलर के `used` को 0 बताना अलग पूर्णांक पैरामीटर का बदलाव वापस कॉपी करता है। हर गलती किसी अलग ऑब्जेक्ट या सीमा को मिलाती है।

helper के पास ही अभिगम की शर्तें लिखें

यहाँ `add_to_prefix` की कॉल के लिए चाहिए:

- जीवित और इनिशियलाइज़ किए गए पूर्णांक ऐरे के पहले तत्व का पॉइंटर
- शून्य से वास्तविक तत्वों की संख्या तक का `length`
- `helper` के तय उद्देश्य के अनुसार चुने तत्व बदलने की अनुमति
- हर वास्तव में होने वाले जोड़ का परिणाम `int` में समाना

`length = 0` पर पहली लूप-जाँच गलत होती है और कोई तत्व पढ़ा या लिखा नहीं जाता। तब भी वास्तविक ऐरे दिया जाता है; हम `null-array` की नई व्यवस्था या शून्य आकार का ऐरे नहीं ला रहे। हल किए प्रोग्राम में लंबाई 4 होने पर चारों तत्व बदलेंगे। लंबाई 5 उसकी चार-तत्व क्षमता के बाहर होगी और ऐसी कॉल अनुमत नहीं है।

केवल पॉइंटर पैरामीटर और दी हुई लंबाई से `helper` वास्तविक क्षमता अपने-आप नहीं जान या जाँच सकता। गलत लंबाई लिखने से कॉलर का ऐरे बड़ा नहीं होता। ये स्पष्ट पूर्वशर्तें हैं, C द्वारा हर कॉल पर चुपचाप की जाने वाली जाँचें नहीं।

हल किया हुआ उदाहरण 2 स्थानीय पैरामीटर का लक्ष्य बदलें

अब दो पूर्णांक लक्ष्य दें। helper के पॉइंटर पैरामीटर में नया मान रखने से क्या कॉलर के पॉइंटर चर का लक्ष्य भी बदल जाएगा?

```
#include <stdio.h>

void retarget_then_write(int *target, int *other)
{
    target = other;
    *target = 9;
}

int main(void)
{
    int left = 2;
    int right = 7;
    int *selected = &left;
    retarget_then_write(selected, &right);
    printf("%d %d %d\n", left, right, *selected);
    return 0;
}
```

- कॉल से पहले कॉलर: left = 2, right = 7 और selected → left
- helper में प्रवेश: उसके अलग target → left और other → right हैं। पॉइंटर आर्गुमेंट के मान इन पैरामीटरों में कॉपी हुए हैं
- target = other helper के target को right की ओर करता है। कॉलर का selected अभी भी left बताता है; दोनों पूर्णांक नहीं बदले
- *target = 9 अब right में लिखकर उसे 9 करता है
- लौटने पर कॉलर का selected → left अभी भी 2 पढ़ता है। आउटपुट: 2 9 2, फिर नई पंक्ति

ट्रेस में तीन पॉइंटर चर हैं: कॉलर का selected और helper के पैरामीटर target तथा other। लक्ष्य साझा होने से ये चर मिलकर एक नहीं हो जाते। target = other एक स्थानीय पॉइंटर मान बदलता है। *target = 9 उस स्थानीय पॉइंटर मान से चुना गया पूर्णांक बदलता है।

गलत परिणाम 2 9 9 helper का पॉइंटर असाइनमेंट कॉलर के selected तक पहुँचा देता है। गलत परिणाम 9 7 9 लेखन से पहले बदले लक्ष्य को भूलता है। गलत परिणाम 2 7 2 कॉलर के right में हुए वास्तविक लेखन को छोड़ता है। हर पॉइंटर चर का अलग चित्र बनाने से तीनों गलतियाँ दूर होती हैं।

यहाँ हम helper से कॉलर के पॉइंटर चर में नया मान रखना नहीं सिखा रहे। उसके लिए अलग इंटरफ़ेस चाहिए। साधारण पॉइंटर आर्गुमेंट को कॉलर के पॉइंटर चर का अपने-आप मिला संदर्भ न मानें।

बताएँ कि फ़ंक्शन क्या लौटाता और क्या बदलता है

केवल तत्व जोड़ने वाला फ़ंक्शन ऐरे में लिखे बिना पूर्णांक लौटा सकता है। दूसरा फ़ंक्शन तत्व बदल सकता है और कोई मान नहीं लौटा सकता। फ़ंक्शन स्टोरेज बदलने के साथ मान भी लौटा सकता है, लेकिन केवल नाम या पैरामीटर सूची में कोष्ठक देखकर इनमें से कोई व्यवहार तय नहीं होता। body और बताई शर्तें पढ़ें।

हर कॉल में चार बातें रखें: आर्गुमेंट के मान, helper के अलग पैरामीटर, लेखन से पहुँचे कॉलर ऑब्जेक्ट और कोई लौटा हुआ मान। scalar फ़ंक्शन पाठ की तरह कॉलर और helper के अलग अवस्था-रिकॉर्ड बनाएँ; जहाँ आवश्यक हो लक्ष्य के तीर जोड़ें।

हल से पहले अभ्यास

ये चार मौलिक अभ्यास हैं और इनके अंक नहीं हैं। कॉल का क्रम, लक्ष्य की पहचान और चुनी लंबाई सामने रखें। प्रश्न 4 जानबूझकर सीमा तोड़ता है और केवल वर्गीकरण तथा सुधार के लिए है।

अभ्यास 1 केवल पढ़कर गणना, फिर बदलाव

आउटपुट और अंतिम ऐरे दें। कौन-सा helper ऐरे बदलता है और क्या बाद में सहेजा गया लौटा मान before भी बदलता है?

```
#include <stdio.h>

int sum_prefix(int values[], int length)
{
    int total = 0;
    for (int i = 0; i < length; i++) {
        total += values[i];
    }
    return total;
}

void clear_first(int values[], int length)
{
    if (length > 0) {
        values[0] = 0;
    }
}

int main(void)
{
    int a[3] = {3, 4, 6};
    int before = sum_prefix(a, 3);
    clear_first(a, 2);
    int after = sum_prefix(a, 2);
    printf("%d %d %d\n", before, after, a[2]);
    return 0;
}
```

सही चुनी लंबाई लें। यहाँ “केवल पढ़ना” `sum_prefix` के वास्तविक काम का वर्णन है; यह नया type qualifier नहीं है।

अभ्यास 2 स्थानीय लंबाई बदलने से helper का अपना लूप बदलता है

दिखाए गए चार-तत्वों वाले ऐरे के लिए प्रोग्राम का व्यवहार निश्चित है, लेकिन helper का तय उद्देश्य ठीक माँगा हुआ शुरुआती हिस्सा भरना है। वास्तविक आउटपुट दें, उद्देश्य से जुड़ी गलती पहचानें और एक असाइनमेंट हटाकर सुधारें। कॉलर का requested नहीं बदलना चाहिए।

```
#include <stdio.h>

void fill_prefix(int values[], int length, int value)
{
    length = 2;
    for (int i = 0; i < length; i++) {
        values[i] = value;
    }
}

int main(void)
{
    int a[4] = {2, 4, 6, 8};
    int requested = 1;
    fill_prefix(a, requested, 9);
    printf("%d %d %d %d | %d\n",
           a[0], a[1], a[2], a[3], requested);
    return 0;
}
```

अभ्यास 3 लक्ष्य के मान बदलें और साझा लक्ष्य भी जाँचें

दोनों कॉल का ट्रेस करें। दूसरी कॉल एक ही लक्ष्य दो बार देती है। समझाएँ कि इस खास helper में इससे समस्या होती है या नहीं और आउटपुट दें।

```
#include <stdio.h>

void swap_values(int *left, int *right)
{
    int saved = *left;
    *left = *right;
    *right = saved;
}

int main(void)
{
    int x = 4;
    int y = 9;
    swap_values(&x, &y);
    swap_values(&x, &x);
    printf("%d %d\n", x, y);
    return 0;
}
```

दोनों दिए लक्ष्य जीवित इनिशियलाइज़ किए गए पूर्णांक हैं। यह न मानें कि पॉइंटर वाले सभी helper को अलग लक्ष्य चाहिए या वे हमेशा साझा लक्ष्य स्वीकार करते हैं; इस helper के स्टेटमेंट जाँचें।

अभ्यास 4 कॉलर ज़रूरत से बड़ी लंबाई देता है

नीचे का अंश जानबूझकर असुरक्षित है। मानें कि दिखाई कॉल main के भीतर है और helper पहले परिभाषित है। उसे कंपाइल या निष्पादित न करें।

```
void add_one(int values[], int length)
{
    for (int i = 0; i < length; i++) {
        values[i] += 1;
    }
}
```

```
int a[2] = {3, 7};
add_one(a, 3);
```

पहला गलत इंडेक्स पहचानें जहाँ लूप पहुँचने की कोशिश करेगा। क्या helper का values[] पैरामीटर दी गई लंबाई के लिए पर्याप्त तत्व बना देता है? केवल कॉल सुधारें, ताकि helper पूरे वास्तविक ऐरे के तत्व एक-एक बढ़ाए; फिर सुधरे प्रोग्राम के दोनों तत्वों का आउटपुट दें।

पूरे हल और गलत उत्तरों का कारण

अभ्यास 1 का हल

पहली sum_prefix कॉल को a तक पहुँच और लंबाई 3 मिलती है। इंडेक्स 0, 1 और 2 देखने पर उसके स्थानीय योग 3, 7 और 13 होते हैं। कोई तत्व बदले बिना वह 13 लौटाती है; कॉलर का before = 13 होता है।

clear_first(a, 2) को कॉपी हुआ पॉइंटर मान और स्थानीय लंबाई 2 मिलती है। उसकी शर्त सही है, इसलिए वह a[0] में 0 लिखती है। वह दोनों चुने तत्व साफ़ नहीं करती: body में केवल इंडेक्स 0 पर एक लेखन है। कॉलर का ऐरे अब {0, 4, 6} है।

दूसरी sum कॉल लंबाई 2 लेती है, इसलिए उसका नया स्थानीय योग पहले 0, फिर 4 होता है। वह 4 लौटाकर after में देती है। कॉलर का सहेजा पूर्णांक before 13 रहता है। आउटपुट: 13 4 6, फिर नई पंक्ति।

4 4 6 सहेजे लौटे मान को बाद के ऐरे बदलावों के साथ बदलता है। 13 10 6 दूसरी लंबाई 2 होने पर भी पूरा ऐरे जोड़ता है। 13 0 6 मानता है कि clear_first पूरा शुरुआती हिस्सा भरता है, जबकि body केवल पहला तत्व लिखती है। फ़ंक्शन की शर्तें और वास्तविक body मेल खानी चाहिए; केवल नाम से गायब काम नहीं जुड़ जाते।

अभ्यास 2 का हल

प्रवेश पर helper का length 1 है और कॉलर का requested भी 1 है। स्थानीय असाइनमेंट helper का length 2 करता है। इसलिए लूप इंडेक्स 0 और 1 पर 9 लिखता है। दोनों इंडेक्स इस चार-तत्वों वाले ऐरे में हैं, इसलिए दिखाए निष्पादन का व्यवहार निश्चित है। अंतिम ऐरे {9, 9, 6, 8} है और कॉलर का requested 1 रहता है। आउटपुट: 9 9 6 8 | 1, फिर नई पंक्ति।

स्टोरेज की सीमा के भीतर रहते हुए भी helper ने माँगे शुरुआती हिस्से वाली शर्त तोड़ी है। व्यवहार निश्चित होना और व्यवहार उद्देश्य के अनुसार होना अलग प्रश्न हैं। कॉपी हुए पैरामीटर का नियम helper को अपनी बदली स्थानीय लंबाई बाद के स्टेटमेंट में इस्तेमाल करने से नहीं रोकता।

length = 2; हटाएँ। पूरा सुधरा प्रोग्राम है:

```
#include <stdio.h>

void fill_prefix(int values[], int length, int value)
{
    for (int i = 0; i < length; i++) {
        values[i] = value;
    }
}

int main(void)
{
    int a[4] = {2, 4, 6, 8};
    int requested = 1;
    fill_prefix(a, requested, 9);
    printf("%d %d %d %d | %d\n",
           a[0], a[1], a[2], a[3], requested);
    return 0;
}
```

सुधरा helper लंबाई 1 रखता है, केवल a[0] लिखता है और इंडेक्स 1 पर गलत जाँच से रुकता है। वह 9 4 6 8 | 1 और फिर नई पंक्ति प्रिंट करता है। कॉलर के requested को 2 करना helper सुधारने के बजाय कॉलर की माँग बदल देगा। मूल असाइनमेंट को कॉलर के requested का बदलाव मानने पर अंतिम | 2 वाला गलत उत्तर मिलेगा।

अभ्यास 3 का हल

पहली कॉल: helper का left → x, right → y; saved में x = 4 की कॉपी है। लेखन *left = *right से x = 9 होता है, फिर *right = saved से y = 4। किसी पॉइंटर पैरामीटर का लक्ष्य नहीं बदलता। लौटने पर कॉलर में x = 9, y = 4 है।

दूसरी कॉल: helper के दोनों पैरामीटर x बताते हैं। नया स्थानीय saved 9 है। पहला लेखन उसी x को पढ़कर उसी में लिखता है, इसलिए 9 रहता है। दूसरा सहेजा हुआ 9 वहीं लिखता है, इसलिए फिर भी मान वही है। ये अलग, क्रमबद्ध स्टेटमेंट हैं, परस्पर टकराते बिना-क्रम वाले अपडेट नहीं। इस खास helper में दोनों दिए लक्ष्य एक ही ऑब्जेक्ट हो सकते हैं।

आउटपुट: 9 4, फिर नई पंक्ति। अंतिम जोड़ी 4 9 बताना y के साथ एक और अदला-बदली गढ़ता है, जबकि दूसरी कॉल उसका उपयोग करती ही नहीं। केवल साझा लक्ष्य होने से अपने-आप त्रुटि बताना aliasing के बारे में बहुत व्यापक निष्कर्ष है: सुरक्षा क्रियाओं और helper की शर्तों पर निर्भर करती है।

अभ्यास 4 का हल

कॉलर के ऐरे में ठीक दो तत्व हैं, जिनके इंडेक्स 0 और 1 हैं। helper को उस ऐरे तक पहुँच और अलग पूर्णांक लंबाई 3 मिलती है। इंडेक्स 2 पर values[2] += 1 ऐरे की सीमा के बाहर तत्व तक पहुँचने की कोशिश करेगा। पैरामीटर की घोषणा तीसरा तत्व नहीं बनाती। अंश में अपरिभाषित व्यवहार है; उसके लिए C11 का निर्धारित आउटपुट या गारंटी वाला “कुछ तत्व बदला हुआ” अंतिम ऐरे न बताएँ।

`add_one(a, 2);` लें। पूरा सुरक्षित प्रोग्राम है:

```
#include <stdio.h>

void add_one(int values[], int length)
{
    for (int i = 0; i < length; i++) {
        values[i] += 1;
    }
}

int main(void)
{
    int a[2] = {3, 7};
    add_one(a, 2);
    printf("%d %d\n", a[0], a[1]);
    return 0;
}
```

सही कॉल 3 को 4 और 7 को 8 करती है, फिर इंडेक्स 2 पर बिना तत्व तक पहुँचे लूप-जाँच गलत हो जाती है। आउटपुट: 4 8, फिर नई पंक्ति। लंबाई 1 सुरक्षित होगी, लेकिन पूरे वास्तविक ऐरे को एक-एक नहीं बढ़ाएगी। लंबाई 3 को केवल “by value” भेजे जाने के कारण सही मानना, गिनती की कॉपी और उसकी वैधता-जाँच को मिलाता है।

त्वरित संदर्भ

- यहाँ की कॉल में ऐरे आर्गुमेंट पहले तत्व का पॉइंटर देता है; वास्तविक ऐरे helper में कॉपी नहीं होता
- साधारण `int values[]` पैरामीटर पॉइंटर पैरामीटर में समायोजित होता है; वह नया स्थानीय ऐरे नहीं है
- सही चुनी लंबाई स्पष्ट दें। पॉइंटर मान में अपने-आप तत्वों की गिनती नहीं होती
- पैरामीटरों को मानों की कॉपी मिलती है, तब भी जब वे मान पॉइंटर हों
- `values[i]` या `*target` में लेखन पहचाने गए कॉलर ऑब्जेक्ट तक पहुँचता है, जब वह ऑब्जेक्ट साझा और जीवित है
- helper का पॉइंटर पैरामीटर दोबारा असाइन करने से कॉलर का पॉइंटर चर दोबारा असाइन नहीं होता
- helper की पूर्णांक लंबाई बदलने से उसके बाद के स्थानीय निर्णय बदलते हैं, कॉलर का लंबाई-चर नहीं
- void helper कोई परिणाम-मान नहीं लौटाता; फिर भी उसके लेखन कॉलर को दिख सकते हैं
- हर कॉल-ट्रेस में लौटे मान, पैरामीटर अपडेट और साझा ऑब्जेक्ट के लेखन अलग रखें
- क्षमता, जीवनकाल, अनुमत बदलाव और अंकगणित की शर्तें बताएँ; न पैरामीटर का नाम, न दी गई लंबाई उन्हें अपने-आप जाँचती है

आगे बढ़ने से पहले

समझाएँ कि कॉपी हुआ पॉइंटर मान तत्व बदलने दे सकता है, जबकि कॉपी हुई लंबाई अलग रहती है। फिर पॉइंटर दोबारा असाइन करने और उसके रास्ते लेखन की तुलना करें। अगले पाठ में ऐसे के भीतर पॉइंटर आगे-पीछे करेंगे और मशीन के बाइट आकार माने बिना वास्तविक ऐसे की तत्व-गिनती को पॉइंटर पैरामीटर के आकार से अलग रखेंगे।

स्रोत टिप्पणी

व्याख्याएँ, कोड, ट्रेस और अभ्यास मौलिक हैं। पहले दो प्रोग्राम इस मॉड्यूल की पहले की मौलिक योजना के प्रोटोटाइप को विकसित करते हैं। C के तथ्यों की जाँच WG14 N1570 C11 समिति-ड्राफ्ट से की गई: ऐसे रूपांतरण §6.3.2.1p3; पैरामीटर समायोजन §6.7.6.3p7; कॉपी हुए आर्गुमेंट §6.5.2.2p4; डिरेफरेंस §6.5.3.2p4; इंडेक्स वाले अभिगम और सीमाएँ §6.5.2.1p2, 6.5.6p8; असाइनमेंट §6.5.16.1p2; void व्यंजक §6.3.2.2p1 और फ़ंक्शन का पूरा होना §6.9.1; automatic जीवनकाल §6.2.4। संदर्भ: <https://open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>

C11 पढ़ाने के लिए चुना संस्करण है, GATE की बताई भाषा-संस्करण शर्त का दावा नहीं। यह कॉल और साझा ऐसे स्टोरेज का सीमित पाठ है, पूरी C या GATE CS की तैयारी नहीं।

Lesson 05 · English

Pointer movement and array boundaries

A position can be valid without holding an element

An index tells us which array element to select. A pointer can identify that same element, and moving the pointer changes the selected position without changing the stored integers. The important boundary is that a pointer may reach one position beyond the final element, although there is no element there to read or write.

In this lesson you will match indexed and pointer-based scans, distinguish movement from a store, classify boundary operations, and find an actual array's element count using `sizeof`. You will also explain why the same size calculation cannot recover a caller's array length inside an array-parameter helper.

Prerequisites: initialized arrays, valid indices, finite loops and accumulators, pointer targets and copied pointer values, and the previous lesson's array arguments, adjusted parameters and explicit length contracts. C11 is our teaching convention, not a version mandated by the GATE syllabus. Every array here has a positive constant size and initialized `int` elements. All arithmetic on `int` stays within -32767 through 32767 . Complete programs are independent. Classification-only text is never an instruction to compile or execute an invalid operation.

We reason with positions in one live array, not invented numeric addresses. No `int` byte size, pointer width, common size for all pointer types or physical placement of separate objects is assumed. Two-dimensional arrays, allocation and casts are outside this lesson.

Count steps in elements

Start with `int marks[4] = {5, 2, 8, 1};` and `int *p = marks;`. In this initializer, `marks` supplies a pointer to its first element. Record `p → marks[0]`.

- Evaluating `p + 1` produces a pointer to `marks[1]`. By itself, this expression does not replace the value in `p`
- The assignment `p = p + 1;` saves that new position in `p`. No element value changes
- The separate statement `p++;` advances the pointer by one element too. We keep movement separate from reads and stores
- With `p → marks[1]`, `p + 2` identifies `marks[3]`, and `p - 1` identifies `marks[0]`

The integer being added counts elements of the pointed-to type. Here the type is `int *`, so each step is one `int` element. Do not multiply the step by an assumed byte size. Starting at `marks[1]`, an offset of 2 reaches index 3, irrespective of how many bytes an `int` occupies on the implementation.

A useful symbolic record is (array identity, position). For example, (`marks`, 1) means the pointer to element 1 of this particular array. The record is our reasoning notation, not a C data structure or an integer representation of a pointer.

Let an actual array `a` have N elements. In this lesson, a known pointer into that live array is represented by position j , allowing the special end position N . For an integer offset k , forming $p + k$ requires the resulting position $j + k$ to stay from 0 through N , inclusive. Reading or writing through the result needs the stricter range 0 through $N - 1$. Check formation before asking whether an element access follows. Going beyond the permitted positions is already invalid even if you never dereference the result.

Brackets and dereference choose the same element

For a valid index i into an actual array `a`, `a[i]` and `*(a + i)` select the same element. The parentheses matter: first form the pointer to position i , then follow it. Either spelling can read that element or select it for an assignment.

For a pointer `p`, `p[i]` means `*(p + i)`. Its index is relative to the current position of `p`. If `p → marks[1]`, then `p[0]` selects `marks[1]`, and `p[2]` selects `marks[3]`. It does not select `marks[2]` merely because the subscript says 2. The resulting target still has to be a valid element of the same array. Equivalent spellings do not make an out-of-bounds access valid.

Keep these three effects separate when `p` points to a readable, writable initialized integer:

Expression or statement	What it does
<code>p = p + 1;</code>	Moves the pointer, subject to the formation bounds
<code>*p = *p + 1;</code>	Increases the selected integer; the pointer stays in place
<code>*(p + 1)</code>	Selects the next element for a read here, if that element exists; <code>p</code> stays in place

The value expression `*p + 1` adds 1 to the current integer. It is not the same expression as `*(p + 1)`. For `p → marks[1]`, they yield 3 and 8 respectively. Neither expression alone stores its result.

One past is a stopping position

For the four-element `marks`, `marks + 4` is a valid one-past pointer. It identifies the boundary after `marks[3]`; it does not identify a fifth integer and does not extend the array. We write it as `marks + 4 (end)`, rather than drawing an arrow to a fictitious element.

You may store this pointer, compare a traversal pointer with it for equality or inequality, and move back from it to an element. In particular, `(marks + 4) - 1` identifies `marks[3]`. You must not evaluate a dereference of the end pointer to read or write an integer. Forming `marks + 5` or `marks - 1` is outside this array's permitted movement range as well. A presumed neighbouring object would not fix that.

Our forward scan starts at position 0, checks `p != end` before reading, and advances by exactly one after each body. Those facts together establish that every body has an element. The condition alone is not a general pointer-validity test. If the starting pointer, end pointer or movement rule is wrong, an inequality test does not repair it. We use no ordering comparison between unrelated pointers.

Worked example 1 Two scans of the same four elements

Predict both totals and the final equality result. The indexed scan finishes before the pointer scan starts; neither scan changes marks.

```
#include <stdio.h>

int main(void)
{
    int marks[4] = {5, 2, 8, 1};
    int indexed_total = 0;
    for (int i = 0; i < 4; i++) {
        indexed_total += marks[i];
    }
    int pointer_total = 0;
    int *p = marks;
    int *end = marks + 4;
    while (p != end) {
        pointer_total += *p;
        p = p + 1;
    }
    printf("%d %d %d\n", indexed_total, pointer_total, p == end);
    return 0;
}
```

The table aligns equivalent checkpoints in the two separate scans. The position after the last body is allowed even though it cannot supply another integer.

Elements already processed	Next index in indexed scan	Next position in pointer scan	Total in each scan
0	0	marks + 0	0
1	1	marks + 1	5
2	2	marks + 2	7
3	3	marks + 3	15
4	4	marks + 4	16

Before each successful test, the next position is an element. The body reads 5, 2, 8 and 1 in order. In each scan, the total at its checkpoint is the sum of exactly the already processed prefix. The final update reaches index 4 or pointer position 4. The next test fails, so there is no fifth read. Each scan has four bodies and five loop-condition tests.

The output is 16 16 1, followed by a newline. Equality produces the `int` value 1 when true, so `%d` matches its type. Both totals are also `int`. The array remains {5, 2, 8, 1}.

If you omit the final value 1, you probably stopped at the last element rather than after it. If you try to add a fifth value, you treated the stopping position as a readable element. If moving `p` changed marks, you confused pointer assignment with a store through `*p`.

Sizes need their own type

`sizeof` asks for a size in C bytes, using the operand's type. It is an operator, not a function call. For our fixed arrays and ordinary integer or pointer operands, its operand is not evaluated. We do not use variable-length arrays, whose rules need separate treatment.

Introduce `size_t` before storing these results. It is an unsigned integer type provided by `<stddef.h>` and is the result type of `sizeof`. Use `%zu` to print a `size_t` value. Do not assume it is `int`, or use `%d` for it. Our counts are small nonnegative values; keep a `size_t` loop index and its `size_t` length together. Unsigned counts are not a place for a `-1` not-found marker. Before subtracting 1 from a count, establish that it is positive.

For an actual array declaration `int values[5] = {4, 7, 1, 6, 2};` in `main`, `sizeof values` measures the entire array, while `sizeof values[0]` measures one `int` element. The array does not convert to a pointer as the operand of this `sizeof`. If one element occupies `S` C bytes, the whole array occupies `5 * S`; the ratio is 5. We need no numeric value for `S`.

Thus `size_t count = sizeof values / sizeof values[0];` records 5. The ratio is an element count, not a byte count. It describes this actual array where that array type is available. It is not a general “length of whatever this name points at” operation.

Worked example 2 Count the actual array and pass a chosen length

This helper uses the previous lesson's array-parameter spelling, now with a `size_t` length. The parameter values is adjusted to `int *`. For this example's contract, the caller supplies the first element of a live array of at most five initialized integers, each from 0 through 20, and `length` is from zero through its actual element count. The helper reads that prefix, changes no elements and returns an `int` total from 0 through 100. Even for `length` zero, our contract supplies the real array's first element; it does not introduce null-pointer calls.

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

int sum_prefix(int values[], size_t length)
{
    int total = 0;
    for (size_t i = 0; i < length; i++) {
        total += values[i];
    }
    return total;
}

int main(void)
{
    int values[5] = {4, 7, 1, 6, 2};
    size_t count = sizeof values / sizeof values[0];
    size_t selected = 3;
    int prefix_total = sum_prefix(values, selected);
    int full_total = sum_prefix(values, count);
    printf("%zu %zu %d %d\n", count, selected, prefix_total, full_total);
    return 0;
}
```

Call	Supplied length	Visited indices	Running totals	Returned total
<code>sum_prefix(values, selected)</code>	3	0, 1, 2	4, 11, 12	12
<code>sum_prefix(values, count)</code>	5	0, 1, 2, 3, 4	4, 11, 12, 18, 20	20

The first call selects only three of the five existing elements. The second selects all five. Every body uses `i < length`, and the caller's supplied length also satisfies the capacity bound. The two totals start independently at 0. Nothing stores through values, so the caller's array stays unchanged. Output: 5 3 12 20, followed by a newline.

Now ask what `sizeof values` would mean in each scope:

- In `main`, `values` names an actual five-element array, so its size covers all five elements
- Inside `sum_prefix`, `values` is a pointer parameter, so its size would be the size of that `int *` parameter type. Writing brackets in the parameter declaration does not preserve the caller's array type

- Inside that helper, `sizeof values[0]` would still measure an `int`. Dividing a pointer size by an element size has no general connection to the supplied length or capacity

These are facts about the types, not a claim that the two byte sizes must differ. A numeric coincidence on one machine proves no length rule. A pointer parameter does not carry an automatic array-length field. Replacing `length` by that ratio would discard the explicit contract. The defined program uses no such replacement.

A distance between positions also has a type

Subtracting two suitable pointers in the same live array can report a signed distance in elements. One or both pointers may be at that array's one-past position. The result must fit in `ptrdiff_t`, a signed integer type declared in `<stddef.h>`. Use `%td` to print it. Our distances are only -5 through 5 and fit. Do not store or print an arbitrary pointer difference as though its type were necessarily `int` or `size_t`.

For positions `j` and `k`, the result of the first pointer minus the second is `j - k`. The order matters: end minus start is positive, start minus end is negative, and a position minus itself is zero. This counts element steps, not bytes. We do not subtract pointers from separate arrays, even if both arrays have equal values or were declared next to each other. This lesson does not use pointer subtraction to discover whether two unknown pointers belong to the same array.

Practice before opening the solutions

These four original transfer tasks are unscored. State the type and bounds facts as well as the numeric answers. Treat the two text-fenced sets as classification-only material; never compile or execute their invalid operations.

Practice 1 Classify the positions and measure the span

Assume a live initialized array `int a[4] = {9, 3, 7, 2};`. Classify each expression independently: valid element pointer, valid end pointer, valid integer read, or undefined operation. They are not consecutive statements.

```
a + 0
a + 4
*(a + 3)
*(a + 4)
a + 5
a - 1
```

Then give the output of the safe complete program in the solution. Before looking, reason about the last value, `end - first`, and `first - end`, with `first = a` and `end = a + 4`. Which type and print format are needed for the two distances?

Practice 2 Move one pointer and change two elements

Trace the current positions separately from the array values. Explain what `q[0]` selects after `q` moves.

```
#include <stdio.h>

int main(void)
{
    int a[5] = {2, 5, 8, 4, 1};
    int *p = a + 1;
    int *q = p;
    p = p + 2;
    *p = *q + 3;
    q = q + 1;
    q[0] = *p - 2;
    printf("%d %d %d %d %d | %d %d\n",
           a[0], a[1], a[2], a[3], a[4], *p, *q);
    return 0;
}
```

Practice 3 Repair a backward scan

The goal is to sum all three elements, visiting them from last to first. This unsafe fragment is for classification only.

```
int values[3] = {3, 4, 6};
int *p = values + 3;
int total = 0;
while (p != values) {
    total += *p;
    p = p - 1;
}
```

Identify the first invalid operation. Repair only the order of the two body statements, then give the visit order, running totals and final pointer position. Explain why the repaired body never forms a before-first pointer. Would the repaired loop read anything if it started with `p = values` instead?

Practice 4 Find the final value without inventing a length

A helper's declaration is `int final_value(int data[], size_t length)`. Its contract requires a pointer to the first element of a live initialized `int` array and a selected length of at least 1 and no greater than the actual count. It returns the last selected integer and changes no elements.

For `int data[5] = {2, 4, 6, 8, 9};` in `main`, complete the call using a count computed from the actual array. Explain why replacing the explicit length inside the helper by `sizeof data / sizeof data[0]` is unjustified. Does an observed pointer size on one machine prove the element count? Explain why the positive-length condition is needed for `data[length - 1]`.

Full solutions and wrong-turn feedback

Practice 1 solution

- `a + 0` is a valid pointer to `a[0]`
- `a + 4` is a valid one-past pointer; it cannot supply an integer read or write
- `*(a + 3)` is a valid read of the last element and yields 2
- `*(a + 4)` attempts an evaluated one-past dereference: undefined behavior, with no prescribed integer result
- `a + 5` is already invalid pointer formation: undefined behavior even without a later dereference
- `a - 1` is also invalid pointer formation: there is no permitted before-first position

The safe program below uses only the allowed endpoints and moves back before reading. `forward` and `backward` are `ptrdiff_t`, whereas the element read is `int`.

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

int main(void)
{
    int a[4] = {9, 3, 7, 2};
    int *first = a;
    int *end = a + 4;
    int *last = end - 1;
    ptrdiff_t forward = end - first;
    ptrdiff_t backward = first - end;
    printf("%d %td %td\n", *last, forward, backward);
    return 0;
}
```

Positions are 0 for `first`, 4 for `end` and 3 for `last`. The differences are $4 - 0 = 4$ and $0 - 4 = -4$. Output: 2 4 -4, followed by a newline. The array is unchanged.

A distance of 3 confuses the last index with the end position. Multiplying 4 by an assumed byte size answers a different question. Treating `a + 5` as harmless because it is not read misses that pointer formation has its own bounds. No test run can give a required answer for the invalid expressions.

Practice 2 solution

After this step	Array values	Position of p	Position of q
All declarations	{2, 5, 8, 4, 1}	1	1
<code>p = p + 2</code>	{2, 5, 8, 4, 1}	3	1
<code>*p = *q + 3</code>	{2, 5, 8, 8, 1}	3	1
<code>q = q + 1</code>	{2, 5, 8, 8, 1}	3	2

After this step	Array values	Position of p	Position of q
q[0] = *p - 2	{2, 5, 6, 8, 1}	3	2

The first movement changes only p; copied pointer values are independent. The first store reads 5 through q and writes 8 to a[3] through p. Then q moves to position 2. Its subscript 0 is relative to that position, so q[0] selects a[2], not a[0]. The last store reads the current 8 at a[3], subtracts 2 and stores 6 at a[2].

All pointer positions used for reads and writes are 1, 2 or 3, within the five-element array. Output: 2 5 6 8 1 | 8 6, followed by a newline. Predicting a change to a[0] treats q as though it still named the array's beginning. Moving q together with p would instead confuse a copied pointer value with a permanent link.

Practice 3 solution

Forming values + 3 is valid. The first loop test is true, but its original first body statement tries to read through that end pointer. That read has undefined behavior. C does not prescribe a total for the unsafe fragment.

Move backward before reading:

```
#include <stdio.h>

int main(void)
{
    int values[3] = {3, 4, 6};
    int *p = values + 3;
    int total = 0;
    while (p != values) {
        p = p - 1;
        total += *p;
    }
    printf("%d %d\n", total, p == values);
    return 0;
}
```

Body number	Position before body	Position read after movement	Value read	Total after body
1	3	2	6	6
2	2	1	4	10
3	1	0	3	13

At each successful test, the current position is from 1 through 3. Moving back produces a readable position from 0 through 2. After the third body, p equals values; the next test fails before another decrement. The loop therefore never forms position -1. It has three bodies and four tests. Output: 13 1, followed by a newline, with the array unchanged.

If the repaired loop starts at `values`, its first test fails: no body, no movement, no read, total 0. The real three-element array still exists. Merely moving the original starting pointer to the last element without reconsidering the condition would skip the first element. Repair the relationship among starting position, test, movement and read, rather than guessing from a plausible total.

Practice 4 solution

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

int final_value(int data[], size_t length)
{
    return data[length - 1];
}

int main(void)
{
    int data[5] = {2, 4, 6, 8, 9};
    size_t length = sizeof data / sizeof data[0];
    int answer = final_value(data, length);
    printf("%zu %d\n", length, answer);
    return 0;
}
```

In `main`, the ratio is $5 * S / S = 5$, where S denotes `sizeof data[0]`. The call passes the pointer to the first element and the copied count 5. Inside the helper, `length - 1` is 4, a valid index, and the initialized value there is 9. Output: 5 9, followed by a newline. Both the array and the caller's count remain unchanged.

Inside the helper, `data` has adjusted type `int *`. Its size measures that pointer type, while `sizeof data[0]` measures `int`. Their ratio does not discover the selected length or the array capacity. Equal pointer sizes for two particular calls would not make their arrays equally long; the same helper could be called with a suitable three-element array as well. We claim nothing about all pointer types having the same size.

The contract's `length >= 1` establishes a final selected element and makes subtracting 1 appropriate. A zero count violates this helper's contract; do not evaluate an element access to test what happens. For a positive count within capacity, the index is from 0 through capacity minus 1. A supplied length of 3 for this same array would select index 2 and return 6. The size ratio in the caller gives its capacity; a chosen shorter length still has to be passed deliberately.

Quick reference

- Record the array identity and current pointer position separately from element values
- Adding 1 to an `int *` advances one `int` element, without assuming a byte size
- Evaluating `p + k` alone leaves `p` unchanged; assignment or incrementing `p` can update its stored pointer value
- For valid targets, `a[i]` selects `*(a + i)`; `p[i]` is relative to `p`'s current position

- Formation allows positions 0 through the count; an element read or write requires 0 through count minus 1
- One past is a valid stopping position, never an extra readable or writable element
- Prove the start, test and step of a traversal together; the comparison alone proves no safety
- Same-array pointer differences count signed element steps; use `ptrdiff_t` and `%td`
- `sizeof` yields `size_t`; use `%zu` and check positivity before subtracting 1 from a count
- The array-size ratio gives a count only for an actual array; an adjusted pointer parameter still needs an explicit length

Before moving on

You should be able to justify every formed position before predicting an output, and say whether each statement moves a pointer or changes an integer. You should also know whether a name denotes an actual array or an adjusted pointer parameter at the exact `sizeof` expression. This lesson stays with one-dimensional fixed arrays; it does not provide a general test for an arbitrary pointer's validity.

Semantic fact checks: [WG14 N1570](#), clauses 6.3.2.1, 6.5.2.1, 6.5.3.4, 6.5.6, 6.7.6.3, 7.19 and 7.21.6.1.

Explanations, examples, traces and practice are original. This attribution identifies verification sources; it is not a claim of recovered historical source accounting or global rights clearance.

Lesson 05 · हिन्दी

पॉइंटर की चाल और ऐरे की सीमाएँ

किसी वैध स्थिति पर तत्व होना आवश्यक नहीं है

इंडेक्स बताता है कि ऐरे का कौन-सा तत्व चुनना है। पॉइंटर उसी तत्व को पहचान सकता है, और पॉइंटर को आगे बढ़ाने से चुनी हुई स्थिति बदलती है, रखे हुए पूर्णांक नहीं। एक महत्वपूर्ण सीमा समझें: पॉइंटर अंतिम तत्व से एक स्थान आगे तक पहुँच सकता है, लेकिन वहाँ पढ़ने या लिखने के लिए कोई तत्व नहीं होता।

इस पाठ में आप इंडेक्स वाले और पॉइंटर वाले स्कैन का मेल समझेंगे, पॉइंटर की चाल को किसी तत्व में लेखन से अलग पहचानेंगे, सीमा संबंधी ऑपरेशन का वर्गीकरण करेंगे और `sizeof` से वास्तविक ऐरे के तत्व गिनेंगे। आप यह भी समझाएँगे कि ऐरे-पैरामीटर वाले सहायक फ़ंक्शन के भीतर वही आकार-गणना कॉलर के ऐरे की लंबाई क्यों नहीं निकाल सकती।

पहले से आवश्यक बातें: इनिशियलाइज़ किए ऐरे, वैध इंडेक्स, सीमित लूप और संचयक, पॉइंटर के लक्ष्य और पॉइंटर मान की कॉपी, तथा पिछले पाठ के ऐरे आर्गुमेंट, समायोजित पैरामीटर और स्पष्ट लंबाई-अनुबंध। पढ़ाने के लिए हम C11 चुनते हैं; GATE पाठ्यक्रम ने यह संस्करण निर्धारित नहीं किया है। यहाँ हर ऐरे का आकार धनात्मक स्थिरांक है और उसके `int` तत्व इनिशियलाइज़ किए गए हैं। `int` की सारी अंकगणित -32767 से 32767 के भीतर रहती है। पूरे प्रोग्राम स्वतंत्र हैं। केवल वर्गीकरण के लिए दिए पाठ का अर्थ किसी अवैध ऑपरेशन को कंपाइल या निष्पादित करने का निर्देश नहीं है।

हम एक जीवित ऐरे की स्थितियों से तर्क करेंगे, काल्पनिक संख्यात्मक एड्रेस से नहीं। `int` का बाइट आकार, पॉइंटर की चौड़ाई, सभी पॉइंटर प्रकारों का समान आकार या अलग ऑब्जेक्टों का भौतिक क्रम नहीं माना गया है। द्वि-आयामी ऐरे, मेमोरी आवंटन और कास्ट इस पाठ के बाहर हैं।

कदमों को तत्वों में गिनें

`int marks[4] = {5, 2, 8, 1};` और `int *p = marks;` से शुरू करें। इस इनिशियलाइज़र में `marks` अपने पहले तत्व का पॉइंटर देता है। दर्ज करें: `p → marks[0]`।

- `p + 1` का मूल्यांकन `marks[1]` का पॉइंटर देता है। केवल यह व्यंजक `p` में रखा मान नहीं बदलता
- असाइनमेंट `p = p + 1;` नई स्थिति को `p` में रखता है। किसी तत्व का मान नहीं बदलता
- अलग स्टेटमेंट `p++;` भी पॉइंटर को एक तत्व आगे बढ़ाता है। हम चाल को पढ़ने और लिखने से अलग रखते हैं
- `p → marks[1]` होने पर `p + 2, marks[3]` को पहचानता है और `p - 1, marks[0]` को

जो पूर्णांक जोड़ा जाता है, वह उस प्रकार के तत्व गिनता है जिसकी ओर पॉइंटर जाता है। यहाँ प्रकार `int *` है, इसलिए हर कदम एक `int` तत्व का है। कदम को किसी मान लिए गए बाइट आकार से गुणा न करें। `marks[1]` से 2 का ऑफ़सेट इंडेक्स 3 पर पहुँचाता है, चाहे उस कार्यान्वयन में `int` कितने भी बाइट का हो।

एक उपयोगी प्रतीकात्मक रिकॉर्ड है (ऐरे की पहचान, स्थिति)। उदाहरण के लिए, `(marks, 1)` इसी विशेष ऐरे के तत्व 1 का पॉइंटर बताता है। यह हमारे तर्क की लिखावट है, कोई C डेटा संरचना या पॉइंटर का पूर्णांक रूप नहीं।

मान लें कि वास्तविक ऐरे a में N तत्व हैं। इस पाठ में उस जीवित ऐरे के ज्ञात पॉइंटर की स्थिति j से बताई जाती है, जिसमें विशेष अंतिम सीमा N भी शामिल है। पूर्णांक ऑफ़सेट k के लिए $p + k$ बनाना तभी वैध है जब परिणामी स्थिति $j + k$, 0 से N तक हो, दोनों सीमाएँ शामिल हैं। परिणाम के रास्ते पढ़ने या लिखने के लिए सीमा और कड़ी है: 0 से $N - 1$ । तत्व तक पहुँचने पर विचार करने से पहले पॉइंटर बनाना जाँचें। अनुमत स्थितियों से बाहर जाना अपने आप में अवैध है, भले ही आप बाद में परिणाम को डीरेफ़रेंस न करें।

कोष्ठक और डीरेफ़रेंस एक ही तत्व चुनते हैं

वास्तविक ऐरे a के वैध इंडेक्स i के लिए $a[i]$ और $*(a + i)$ एक ही तत्व चुनते हैं। गोल कोष्ठकों का महत्व है: पहले स्थिति i का पॉइंटर बनाएँ, फिर उसके लक्ष्य तक जाएँ। दोनों रूप उस तत्व को पढ़ सकते हैं या असाइनमेंट के लिए चुन सकते हैं।

पॉइंटर p के लिए $p[i]$ का अर्थ $*(p + i)$ है। इसका इंडेक्स p की वर्तमान स्थिति से सापेक्ष है। यदि $p \rightarrow \text{marks}[1]$, तो $p[0]$, $\text{marks}[1]$ को चुनता है और $p[2]$, $\text{marks}[3]$ को। केवल सबस्क्रिप्ट 2 होने से वह $\text{marks}[2]$ नहीं चुनता। परिणामी लक्ष्य अब भी उसी ऐरे का वैध तत्व होना चाहिए। समान अर्थ वाली लिखावट सीमा से बाहर की पहुँच को वैध नहीं बनाती।

जब p ऐसे इनिशियलाइज़ किए पूर्णांक की ओर हो जिसे पढ़ना और बदलना वैध है, तब ये तीन प्रभाव अलग रखें:

व्यंजक या स्टेटमेंट	उसका प्रभाव
$p = p + 1;$	पॉइंटर को आगे बढ़ाता है; नई स्थिति बनाने की सीमाएँ लागू हैं
$*p = *p + 1;$	चुने हुए पूर्णांक को बढ़ाता है; पॉइंटर वहीं रहता है
$*(p + 1)$	यहाँ अगले तत्व को पढ़ने के लिए चुनता है, यदि वह तत्व मौजूद हो; p वहीं रहता है

मान देने वाला व्यंजक $*p + 1$ वर्तमान पूर्णांक में 1 जोड़ता है। यह $*(p + 1)$ वाला व्यंजक नहीं है। $p \rightarrow \text{marks}[1]$ होने पर इनके मान क्रमशः 3 और 8 हैं। अकेला कोई भी व्यंजक अपना परिणाम सहेजता नहीं है।

अंतिम तत्व के एक आगे की स्थिति रुकने के लिए है

चार तत्वों वाले marks के लिए $\text{marks} + 4$ एक वैध one-past पॉइंटर है, यानी अंतिम तत्व के ठीक बाद वाली स्थिति। यह $\text{marks}[3]$ के बाद की सीमा पहचानता है; पाँचवाँ पूर्णांक नहीं पहचानता और ऐरे का आकार नहीं बढ़ाता। हम इसे $\text{marks} + 4$ (end) लिखेंगे, किसी काल्पनिक तत्व तक तीर नहीं बनाएँगे।

आप यह पॉइंटर सहेज सकते हैं, ट्रैवर्सल पॉइंटर से इसकी समानता या असमानता जाँच सकते हैं और इससे पीछे किसी तत्व पर लौट सकते हैं। विशेष रूप से, $(\text{marks} + 4) - 1$, $\text{marks}[3]$ को पहचानता है। इस अंतिम सीमा के पॉइंटर को डीरेफ़रेंस करके पूर्णांक पढ़ना या लिखना वैध नहीं है। $\text{marks} + 5$ या $\text{marks} - 1$ बनाना भी इस ऐरे की अनुमत चाल-सीमा से बाहर है। पास में कोई दूसरा ऑब्जेक्ट मान लेने से यह सही नहीं होगा।

हमारा आगे जाने वाला स्कैन स्थिति 0 से शुरू होता है, पढ़ने से पहले $p \neq \text{end}$ जाँचता है और हर बॉडी के बाद ठीक एक कदम आगे बढ़ता है। ये बातें मिलकर प्रमाणित करती हैं कि हर बॉडी के लिए तत्व मौजूद है। केवल शर्त किसी भी पॉइंटर की वैधता जाँचने का सामान्य तरीका नहीं है। यदि शुरुआती पॉइंटर, अंतिम सीमा का पॉइंटर या चाल का नियम गलत हो, तो असमानता की जाँच उसे ठीक नहीं करती। हम असंबंधित पॉइंटर्स के बीच क्रम-संबंधी तुलना नहीं करते।

हल किया हुआ उदाहरण 1 उन्हीं चार तत्वों के दो स्कैन

दोनों योग और अंतिम समानता-जाँच का परिणाम बताएँ। पॉइंटर वाला स्कैन शुरू होने से पहले इंडेक्स वाला स्कैन समाप्त हो जाता है; कोई भी स्कैन marks नहीं बदलता।

```
#include <stdio.h>

int main(void)
{
    int marks[4] = {5, 2, 8, 1};
    int indexed_total = 0;
    for (int i = 0; i < 4; i++) {
        indexed_total += marks[i];
    }
    int pointer_total = 0;
    int *p = marks;
    int *end = marks + 4;
    while (p != end) {
        pointer_total += *p;
        p = p + 1;
    }
    printf("%d %d %d\n", indexed_total, pointer_total, p == end);
    return 0;
}
```

तालिका दोनों अलग स्कैन के समान चरणों का मेल दिखाती है। अंतिम बॉडी के बाद की स्थिति अनुमत है, हालाँकि उससे एक और पूर्णांक नहीं मिल सकता।

पहले ही संसाधित तत्व	इंडेक्स स्कैन का अगला इंडेक्स	पॉइंटर स्कैन की अगली स्थिति	प्रत्येक स्कैन का योग
0	0	marks + 0	0
1	1	marks + 1	5
2	2	marks + 2	7
3	3	marks + 3	15
4	4	marks + 4	16

हर सफल जाँच से पहले अगली स्थिति एक तत्व होती है। बॉडी क्रम से 5, 2, 8 और 1 पढ़ती है। हर स्कैन के जाँच-बिंदु पर योग ठीक उस शुरुआती हिस्से का है जिसे अब तक संसाधित किया गया है। अंतिम अपडेट इंडेक्स 4 या पॉइंटर स्थिति 4 तक पहुँचता है। अगली जाँच असफल होती है, इसलिए पाँचवीं बार पढ़ना नहीं होता। हर स्कैन की चार बॉडी और लूप-शर्त की पाँच जाँचें हैं।

आउटपुट है 16 16 1, फिर नई पंक्ति। समानता सही होने पर int मान 1 देती है, इसलिए %d उसके प्रकार से मेल खाता है। दोनों योग भी int हैं। ऐसे {5, 2, 8, 1} ही रहता है।

यदि आपने अंतिम मान 1 छोड़ दिया, तो संभवतः उसके बाद की सीमा के बजाय अंतिम तत्व पर ही रुक गए। यदि आप पाँचवाँ मान जोड़ने की कोशिश करते हैं, तो रुकने की स्थिति को पढ़ने योग्य तत्व मान रहे हैं। यदि p को चलाने पर आपने marks बदल दिया, तो पॉइंटर असाइनमेंट को *p के रास्ते लेखन समझ लिया।

आकारों के लिए अपना प्रकार चाहिए

sizeof ऑपरेटर के प्रकार के आधार पर C बाइटों में आकार देता है। यह ऑपरेटर है, फ़ंक्शन कॉल नहीं। हमारे स्थिर आकार वाले ऐरे और साधारण पूर्णांक या पॉइंटर ऑपरेंड के लिए इसका ऑपरेंड मूल्यांकित नहीं होता। हम परिवर्तनीय लंबाई वाले ऐरे नहीं लेते, जिनके नियम अलग से समझने होंगे।

इन परिणामों को सहेजने से पहले size_t समझें। यह <stddef.h> में उपलब्ध unsigned पूर्णांक प्रकार है और sizeof के परिणाम का प्रकार है। size_t मान छापने के लिए %zu उपयोग करें। इसे int न मानें और इसके लिए %d न लें। हमारी गिनतियाँ छोटी और गैर-ऋणात्मक हैं; size_t लूप इंडेक्स को size_t लंबाई के साथ रखें। unsigned गिनती में -1 को नहीं-मिला संकेत की तरह न रखें। किसी गिनती से 1 घटाने से पहले उसका धनात्मक होना स्थापित करें।

main में वास्तविक ऐरे घोषणा `int values[5] = {4, 7, 1, 6, 2};` के लिए sizeof values पूरे ऐरे को मापता है, जबकि sizeof values[0] एक int तत्व को। इस sizeof का ऑपरेंड बनने पर ऐरे पॉइंटर में नहीं बदलता। यदि एक तत्व S C बाइट लेता है, तो पूरा ऐरे 5 * S लेता है; अनुपात 5 है। हमें S का संख्यात्मक मान नहीं चाहिए।

इसलिए `size_t count = sizeof values / sizeof values[0];` में 5 दर्ज होता है। अनुपात तत्वों की संख्या है, बाइटों की नहीं। यह उस स्थान पर वास्तविक ऐरे का वर्णन करता है जहाँ उसका ऐरे प्रकार उपलब्ध है। यह सामान्य रूप से “यह नाम जिसकी ओर जाता है उसकी लंबाई” निकालने का तरीका नहीं है।

हल किया हुआ उदाहरण 2 वास्तविक ऐरे की गिनती और चुनी हुई लंबाई

यह सहायक फ़ंक्शन पिछले पाठ का ऐरे-पैरामीटर रूप इस्तेमाल करता है, अब `size_t` लंबाई के साथ। पैरामीटर `values` का प्रकार समायोजित होकर `int *` होता है। इस उदाहरण का अनुबंध है: कॉलर ऐसे जीवित ऐरे का पहला तत्व देता है जिसमें अधिकतम पाँच इनिशियलाइज़ किए पूर्णांक हों, हर मान 0 से 20 तक हो और `length`, शून्य से ऐरे की वास्तविक तत्व-संख्या तक हो। सहायक फ़ंक्शन वह शुरुआती हिस्सा पढ़ता है, कोई तत्व नहीं बदलता और 0 से 100 तक का `int` योग लौटाता है। शून्य लंबाई पर भी हमारा अनुबंध वास्तविक ऐरे का पहला तत्व देता है; इसमें नल-पॉइंटर कॉल शामिल नहीं हैं।

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

int sum_prefix(int values[], size_t length)
{
    int total = 0;
    for (size_t i = 0; i < length; i++) {
        total += values[i];
    }
    return total;
}

int main(void)
{
    int values[5] = {4, 7, 1, 6, 2};
    size_t count = sizeof values / sizeof values[0];
    size_t selected = 3;
    int prefix_total = sum_prefix(values, selected);
    int full_total = sum_prefix(values, count);
    printf("%zu %zu %d %d\n", count, selected, prefix_total, full_total);
    return 0;
}
```

कॉल	दी गई लंबाई	देखे गए इंडेक्स	क्रमिक योग	लौटाया योग
<code>sum_prefix(values, selected)</code>	3	0, 1, 2	4, 11, 12	12
<code>sum_prefix(values, count)</code>	5	0, 1, 2, 3, 4	4, 11, 12, 18, 20	20

पहली कॉल पाँच मौजूद तत्वों में से केवल तीन चुनती है। दूसरी सभी पाँच चुनती है। हर बॉडी `i < length` का पालन करती है और कॉलर की दी हुई लंबाई भी क्षमता की सीमा में है। दोनों योग अलग-अलग 0 से शुरू होते हैं। `values` के रास्ते कहीं लेखन नहीं है, इसलिए कॉलर का ऐरे अपरिवर्तित रहता है। आउटपुट: 5 3 12 20, फिर नई पंक्ति।

अब पूछें कि हर स्कोप में `sizeof values` का क्या अर्थ होगा:

- `main` में `values` वास्तविक पाँच-तत्वों वाला ऐरे है, इसलिए उसका आकार सभी पाँच तत्वों को शामिल करता है
- `sum_prefix` के भीतर `values` पॉइंटर पैरामीटर है, इसलिए उसका आकार उस `int *` पैरामीटर प्रकार का होगा। पैरामीटर घोषणा में वर्ग कोष्ठक लिखने से कॉलर का ऐरे प्रकार सुरक्षित नहीं रहता

- उस सहायक फ़ंक्शन में `sizeof values[0]` अब भी `int` को मापेगा। पॉइंटर के आकार को तत्व के आकार से भाग देने का दी गई लंबाई या क्षमता से कोई सामान्य संबंध नहीं है

ये प्रकारों से जुड़े तथ्य हैं, यह दावा नहीं कि दोनों बाइट आकारों का अलग होना अनिवार्य है। किसी मशीन पर संख्यात्मक संयोग कोई लंबाई-नियम सिद्ध नहीं करता। पॉइंटर पैरामीटर अपने आप ऐरे-लंबाई का अलग फ़ील्ड नहीं रखता। `length` की जगह वह अनुपात रखने से स्पष्ट अनुबंध छूट जाएगा। निश्चित व्यवहार वाले इस प्रोग्राम में ऐसा प्रतिस्थापन नहीं है।

दो स्थितियों की दूरी का भी प्रकार होता है

उसी जीवित ऐरे के दो उपयुक्त पॉइंटर्स को घटाने से तत्वों में `signed` दूरी मिल सकती है। एक या दोनों पॉइंटर उस ऐरे की `one-past` स्थिति पर हो सकते हैं। परिणाम को `<stddef.h>` में घोषित `signed` पूर्णांक प्रकार `ptrdiff_t` में समाना चाहिए। उसे छापने के लिए `%td` लें। हमारी दूरियाँ केवल `-5` से `5` तक हैं और इस प्रकार में समायोजित हैं। किसी सामान्य पॉइंटर-अंतर को इस धारणा से न सहेजें या छापें कि उसका प्रकार आवश्यक रूप से `int` या `size_t` है।

स्थितियाँ `j` और `k` होने पर पहले पॉइंटर में से दूसरा घटाने का परिणाम `j - k` है। क्रम महत्वपूर्ण है: अंत में से आरंभ घटाना धनात्मक, आरंभ में से अंत घटाना ऋणात्मक और किसी स्थिति में से वही स्थिति घटाना शून्य देता है। यह तत्वों के कदम गिनता है, बाइट नहीं। हम अलग ऐरे के पॉइंटर नहीं घटाते, भले दोनों ऐरे के मान समान हों या उनकी घोषणाएँ साथ लिखी हों। यह पाठ पॉइंटर घटाव का उपयोग यह पता लगाने के लिए नहीं करता कि दो अज्ञात पॉइंटर एक ही ऐरे के हैं या नहीं।

हल देखने से पहले अभ्यास करें

ये चार मौलिक ट्रांसफ़र अभ्यास बिना अंक के हैं। संख्यात्मक उत्तर के साथ प्रकार और सीमाओं के तथ्य भी बताएँ। नीचे पाठ-ब्लॉक में दिए दोनों समूह केवल वर्गीकरण के लिए हैं; उनके अवैध ऑपरेशन कभी कंपाइल या निष्पादित न करें।

अभ्यास 1 स्थितियों का वर्गीकरण और पूरी दूरी

मान लें कि एक जीवित इनिशियलाइज़ किया ऐरे `int a[4] = {9, 3, 7, 2};` मौजूद है। हर व्यंजक का स्वतंत्र वर्गीकरण करें: वैध तत्व-पॉइंटर, वैध अंतिम-सीमा पॉइंटर, वैध पूर्णांक पठन या अपरिभाषित ऑपरेशन। ये क्रम से चलने वाले स्टेटमेंट नहीं हैं।

```
a + 0
a + 4
*(a + 3)
*(a + 4)
a + 5
a - 1
```

फिर हल में दिए सुरक्षित पूरे प्रोग्राम का आउटपुट बताएँ। देखने से पहले `first = a` और `end = a + 4` मानकर अंतिम तत्व के मान, `end - first` और `first - end` पर तर्क करें। दोनों दूरियों के लिए कौन-सा प्रकार और प्रिंट फ़ॉर्मेट चाहिए?

अभ्यास 2 एक पॉइंटर चलाएँ और दो तत्व बदलें

वर्तमान स्थितियों का ट्रेस ऐसे के मानों से अलग करें। समझाएँ कि `q` के चलने के बाद `q[0]` क्या चुनता है।

```
#include <stdio.h>

int main(void)
{
    int a[5] = {2, 5, 8, 4, 1};
    int *p = a + 1;
    int *q = p;
    p = p + 2;
    *p = *q + 3;
    q = q + 1;
    q[0] = *p - 2;
    printf("%d %d %d %d %d | %d %d\n",
           a[0], a[1], a[2], a[3], a[4], *p, *q);
    return 0;
}
```

अभ्यास 3 पीछे जाने वाला स्कैन ठीक करें

लक्ष्य तीनों तत्वों को अंतिम से पहले की ओर देखते हुए उनका योग निकालना है। यह असुरक्षित अंश केवल वर्गीकरण के लिए है।

```
int values[3] = {3, 4, 6};
int *p = values + 3;
int total = 0;
while (p != values) {
    total += *p;
    p = p - 1;
}
```

पहला अवैध ऑपरेशन पहचानें। बॉडी के केवल दो स्टेटमेंटों का क्रम बदलकर सुधारें, फिर देखने का क्रम, क्रमिक योग और अंतिम पॉइंटर स्थिति बताएँ। समझाएँ कि सुधरी हुई बॉडी पहले तत्व से पहले का पॉइंटर कभी क्यों नहीं बनाती। यदि शुरुआत `p = values` से होती, तो क्या सुधरा हुआ लूप कुछ पढ़ता?

अभ्यास 4 लंबाई गढ़े बिना अंतिम मान निकालें

एक सहायक फ़ंक्शन की घोषणा `int final_value(int data[], size_t length)` है। इसके अनुबंध में जीवित इनिशियलाइज़ किए `int` ऐसे के पहले तत्व का पॉइंटर और कम-से-कम 1 तथा वास्तविक गिनती से अधिक नहीं चुनी गई लंबाई चाहिए। यह अंतिम चुना हुआ पूर्णांक लौटाता है और कोई तत्व नहीं बदलता।

`main` में `int data[5] = {2, 4, 6, 8, 9};` के लिए वास्तविक ऐसे से गिनती निकालकर कॉल पूरी करें। समझाएँ कि सहायक फ़ंक्शन के भीतर स्पष्ट लंबाई की जगह `sizeof data / sizeof data[0]` रखना उचित क्यों नहीं है। क्या किसी मशीन पर देखा गया पॉइंटर आकार तत्वों की संख्या साबित करता है? समझाएँ कि `data[length - 1]` के लिए धनात्मक लंबाई की शर्त क्यों चाहिए।

पूरे हल और गलत रास्तों की प्रतिक्रिया

अभ्यास 1 का हल

- $a + 0$, $a[0]$ का वैध पॉइंटर है
- $a + 4$ वैध one-past पॉइंटर है; इससे पूर्णांक पढ़ना या लिखना संभव नहीं है
- $*(a + 3)$ अंतिम तत्व का वैध पठन है और 2 देता है
- $*(a + 4)$ one-past पॉइंटर का मूल्यांकित डीरेफ़रेंस करने का प्रयास है: अपरिभाषित व्यवहार, जिसका कोई निर्धारित पूर्णांक परिणाम नहीं है
- $a + 5$ में पॉइंटर बनाना ही अवैध है: बाद में डीरेफ़रेंस किए बिना भी अपरिभाषित व्यवहार
- $a - 1$ भी अवैध पॉइंटर निर्माण है: पहले तत्व से पहले कोई अनुमत स्थिति नहीं है

नीचे का सुरक्षित प्रोग्राम केवल अनुमत सिरों का उपयोग करता है और पढ़ने से पहले पीछे चलता है। forward और backward, ptrdiff_t हैं, जबकि तत्व का पढ़ा गया मान int है।

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

int main(void)
{
    int a[4] = {9, 3, 7, 2};
    int *first = a;
    int *end = a + 4;
    int *last = end - 1;
    ptrdiff_t forward = end - first;
    ptrdiff_t backward = first - end;
    printf("%d %td %td\n", *last, forward, backward);
    return 0;
}
```

स्थितियाँ हैं: first के लिए 0, end के लिए 4 और last के लिए 3। अंतर हैं $4 - 0 = 4$ और $0 - 4 = -4$ । आउटपुट: 2 4 -4, फिर नई पंक्ति। ऐसे अपरिवर्तित है।

दूरी 3 बताना अंतिम इंडेक्स को अंतिम सीमा समझ लेना है। 4 को किसी मान लिए बाइट आकार से गुणा करना किसी और प्रश्न का उत्तर देता है। $a + 5$ को इस कारण हानिरहित मानना कि उसे पढ़ा नहीं गया, पॉइंटर निर्माण की अपनी सीमाएँ भूलना है। कोई परीक्षण रन अवैध व्यंजकों का अनिवार्य उत्तर नहीं दे सकता।

अभ्यास 2 का हल

इस चरण के बाद	ऐरे के मान	p की स्थिति	q की स्थिति
सभी घोषणाएँ	{2, 5, 8, 4, 1}	1	1
$p = p + 2$	{2, 5, 8, 4, 1}	3	1

इस चरण के बाद	ऐरे के मान	p की स्थिति	q की स्थिति
*p = *q + 3	{2, 5, 8, 8, 1}	3	1
q = q + 1	{2, 5, 8, 8, 1}	3	2
q[0] = *p - 2	{2, 5, 6, 8, 1}	3	2

पहली चाल केवल p बदलती है; कॉपी किए पॉइंटर मान स्वतंत्र हैं। पहला लेखन q से 5 पढ़ता है और p के रास्ते a[3] में 8 लिखता है। फिर q स्थिति 2 पर जाता है। उसका सबस्क्रिप्ट 0 उसी स्थिति से सापेक्ष है, इसलिए q[0], a[2] चुनता है, a[0] नहीं। अंतिम लेखन a[3] का वर्तमान 8 पढ़ता है, 2 घटाता है और a[2] में 6 रखता है।

पढ़ने और लिखने के लिए इस्तेमाल सारी पॉइंटर स्थितियाँ 1, 2 या 3 हैं, जो पाँच-तत्वों वाले ऐरे के भीतर हैं। आउटपुट: 2 5 6 8 1 | 8 6, फिर नई पंक्ति। a[0] में बदलाव बताना q को अब भी ऐरे के आरंभ का नाम मान लेना है। p के साथ q को भी चलाना कॉपी किए पॉइंटर मान को स्थायी जुड़ाव समझने की गलती होगी।

अभ्यास 3 का हल

values + 3 बनाना वैध है। पहली लूप-जाँच सही है, लेकिन मूल बॉडी का पहला स्टेटमेंट उसी अंतिम सीमा के पॉइंटर से पढ़ने की कोशिश करता है। उस पठन का व्यवहार अपरिभाषित है। C इस असुरक्षित अंश के लिए कोई योग निर्धारित नहीं करता।

पढ़ने से पहले पीछे चलें:

```
#include <stdio.h>

int main(void)
{
    int values[3] = {3, 4, 6};
    int *p = values + 3;
    int total = 0;
    while (p != values) {
        p = p - 1;
        total += *p;
    }
    printf("%d %d\n", total, p == values);
    return 0;
}
```

बॉडी संख्या	बॉडी से पहले स्थिति	चलने के बाद पढ़ी स्थिति	पढ़ा मान	बॉडी के बाद योग
1	3	2	6	6
2	2	1	4	10
3	1	0	3	13

हर सफल जाँच पर वर्तमान स्थिति 1 से 3 के बीच है। पीछे जाने पर 0 से 2 के बीच पढ़ने योग्य स्थिति मिलती है। तीसरी बॉडी के बाद p, values के बराबर है; फिर घटाने से पहले ही अगली जाँच असफल हो जाती है। इसलिए लूप कभी स्थिति -1 नहीं बनाता। इसकी तीन बॉडी और चार जाँचें हैं। आउटपुट: 13 1, फिर नई पंक्ति; ऐसे अपरिवर्तित है।

यदि सुधरा लूप values से शुरू हो, तो पहली जाँच ही असफल है: कोई बॉडी नहीं, चाल नहीं, पठन नहीं, योग 0। वास्तविक तीन-तत्वों वाला ऐसे अब भी मौजूद है। शर्त पर दोबारा विचार किए बिना मूल शुरुआती पॉइंटर को केवल अंतिम तत्व पर रखने से पहला तत्व छूट जाएगा। किसी संभव लगने वाले योग से अनुमान लगाने के बजाय शुरुआती स्थिति, जाँच, चाल और पठन का आपसी संबंध सुधारें।

अभ्यास 4 का हल

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

int final_value(int data[], size_t length)
{
    return data[length - 1];
}

int main(void)
{
    int data[5] = {2, 4, 6, 8, 9};
    size_t length = sizeof data / sizeof data[0];
    int answer = final_value(data, length);
    printf("%zu %d\n", length, answer);
    return 0;
}
```

main में अनुपात $5 * S / S = 5$ है, जहाँ S, sizeof data[0] को दर्शाता है। कॉल पहले तत्व का पॉइंटर और गिनती 5 की कॉपी देती है। सहायक फ़ंक्शन में length - 1, 4 है, जो वैध इंडेक्स है, और वहाँ इनिशियलाइज़ किया मान 9 है। आउटपुट: 5 9, फिर नई पंक्ति। ऐसे और कॉलर की गिनती दोनों अपरिवर्तित रहते हैं।

सहायक फ़ंक्शन के भीतर data का समायोजित प्रकार int * है। उसका आकार उसी पॉइंटर प्रकार को मापता है, जबकि sizeof data[0], int को मापता है। उनका अनुपात चुनी हुई लंबाई या ऐसे की क्षमता नहीं खोजता। दो विशेष कॉलों में पॉइंटर आकार समान होने से उनके ऐसे बराबर लंबे नहीं हो जाते; इसी सहायक फ़ंक्शन को उपयुक्त तीन-तत्वों वाला ऐसे भी दिया जा सकता है। हम सभी पॉइंटर प्रकारों का आकार समान होने का दावा नहीं करते।

अनुबंध का length >= 1 अंतिम चुने तत्व का अस्तित्व स्थापित करता है और 1 घटाना उचित बनाता है। शून्य गिनती इस सहायक फ़ंक्शन के अनुबंध को तोड़ती है; परिणाम देखने के लिए तत्व तक पहुँचने का मूल्यांकन न करें। क्षमता के भीतर धनात्मक गिनती के लिए इंडेक्स 0 से क्षमता घटा 1 तक है। इसी ऐसे को लंबाई 3 देने पर इंडेक्स 2 चुना जाएगा और 6 लौटेगा। कॉलर का आकार-अनुपात उसकी क्षमता देता है; छोटा चुना हुआ हिस्सा चाहिए तो उसकी लंबाई जानबूझकर देनी होगी।

त्वरित संदर्भ

- ऐसे की पहचान और वर्तमान पॉइंटर स्थिति को तत्वों के मानों से अलग दर्ज करें
- int * में 1 जोड़ने पर एक int तत्व आगे जाते हैं; बाइट आकार मानने की जरूरत नहीं है

- केवल $p + k$ का मान निकालने से p नहीं बदलता; असाइनमेंट या p को इन्क्रिमेंट करना उसमें रखा पॉइंटर मान अपडेट कर सकता है
- वैध लक्ष्यों के लिए $a[i]$, $*(a + i)$ चुनता है; $p[i]$, p की वर्तमान स्थिति से सापेक्ष है
- निर्माण के लिए 0 से गिनती तक स्थितियाँ अनुमत हैं; तत्व पढ़ने या लिखने के लिए 0 से गिनती घटा 1 तक चाहिए
- one-past रुकने की वैध स्थिति है, कोई अतिरिक्त पढ़ने या लिखने योग्य तत्व नहीं
- ट्रैवर्सल की शुरुआत, जाँच और कदम को साथ प्रमाणित करें; अकेली तुलना सुरक्षा सिद्ध नहीं करती
- उसी ऐरे के पॉइंटर्स का अंतर तत्वों के signed कदम गिनता है; `ptrdiff_t` और `%td` लें
- `sizeof` का परिणाम `size_t` है; `%zu` लें और गिनती से 1 घटाने से पहले उसका धनात्मक होना जाँचें
- ऐरे-आकार अनुपात केवल वास्तविक ऐरे की गिनती देता है; समायोजित पॉइंटर पैरामीटर को अब भी स्पष्ट लंबाई चाहिए

आगे बढ़ने से पहले

आउटपुट का अनुमान लगाने से पहले आपको हर बनाई गई स्थिति का औचित्य बताना आना चाहिए। यह भी बता सकें कि हर स्टेटमेंट पॉइंटर चलाता है या पूर्णतः बदलता है। ठीक उस `sizeof` व्यंजक पर यह पहचानें कि नाम वास्तविक ऐरे है या समायोजित पॉइंटर पैरामीटर। यह पाठ एक-आयामी स्थिर आकार वाले ऐरे तक सीमित है; किसी भी मनमाने पॉइंटर की वैधता का सामान्य परीक्षण नहीं देता।

अर्थ-संबंधी तथ्य-जाँच: [WG14 N1570](#), खंड 6.3.2.1, 6.5.2.1, 6.5.3.4, 6.5.6, 6.7.6.3, 7.19 और 7.21.6.1। व्याख्याएँ, उदाहरण, ट्रेस और अभ्यास मौलिक हैं। यह उल्लेख तथ्य-जाँच का स्रोत पहचानता है; पुराने स्रोत-उपयोग का हिसाब वापस मिल जाने या सभी अधिकारों की समग्र मंजूरी का दावा नहीं है।

Lesson 06 · English

Rows and columns in two-dimensional arrays

One array can contain smaller arrays

A one-dimensional array gives each element one index. Now suppose each row contains three integers and there are two rows. We still select one object at a time: first a row, then an integer within that row. Keeping those two selections separate makes the loops, pointer types and boundaries much easier to explain.

In this lesson you will read a two-dimensional initializer, trace a nested loop, place an accumulator reset correctly, scan a chosen column and distinguish movement between rows from movement between integers. You will explain the row type instead of treating every pointer as an interchangeable address.

Prerequisites: initialized arrays and valid indices; loop tests, updates and accumulators; pointer targets and copied values; array-to-pointer conversion; element-unit pointer arithmetic, one-past boundaries, `sizeof`, `size_t` and `%zu`. The earlier call model remains unchanged, but this lesson needs no new helper interface.

C11 is our teaching convention, not a version specified by the GATE syllabus. All actual arrays here have positive constant dimensions and initialized `int` elements. All `int` calculations remain within -32767 through 32767 . Each complete program is independent. Explicitly marked classification fragments must never be compiled or executed. We assume no particular `int` byte size or pointer width. Variable-length arrays, allocation, general double-pointer programming and matrix algorithms are outside this lesson.

Read both dimensions and both initializer levels

For `int grid[2][3] = {{4, 1, 7}, {2, 6, 3}};`, read the declaration as an array of two elements, where each element is itself an array of three `int` objects. The outer array's element is a whole row. The row's element is one integer.

Row index	Column 0	Column 1	Column 2
0	4	1	7
1	2	6	3

The first inner brace group initializes row 0, and the second initializes row 1. The dimensions are counts: two rows and three columns in every row. They are not the last valid indices.

- `grid[0]` selects the first three-integer row
- `grid[1][2]` selects column 2 within row 1, whose current value is 3
- `grid[0][1]` selects a different integer, currently 1
- A read or write of `grid[r][c]` requires both $0 \leq r \leq 1$ and $0 \leq c \leq 2$

The first index is not “whichever dimension is being scanned.” It always selects a row in this declaration. A column scan changes the row index while holding the column index fixed. Swapping the indices changes the selected object and can violate a different bound.

Brace grouping also makes omitted values clear. `int ready[2][3] = {{5}, {2, 4}};` starts with rows `{5, 0, 0}` and `{2, 4, 0}`. The zeros come from the initializer rule, not from a general promise about automatic storage. An automatic declaration without an initializer does not acquire these guaranteed zeros. Our executable programs provide initialized elements before any read.

A nested loop finishes its inner work before advancing the outer index

A nested loop is a loop inside another loop's body. For a row scan, keep `r` fixed while `c` visits 0, 1 and 2. Only after that entire inner loop ends does the outer loop advance `r`.

Read the first worked program with this execution order:

1. Initialize outer `r` to 0 and test `r < 2`
2. Enter that row's outer body, initialize `row_total` to 0, and reach the inner loop
3. Initialize inner `c` to 0, test `c < 3`, process the element, then update `c`
4. Repeat the inner test/body/update until the test fails with `c = 3`
5. Continue with the remaining statements in the outer body, including saving this row's total
6. Finish the outer body, update `r`, and test the outer condition again

On the next successful outer test, the body reaches `int row_total = 0;` again. This starts the new row's total at zero. Reaching for `(int c = 0; ...)` again also starts that inner scan at column 0. Do not carry the previous inner scan's ending value 3 into the next row. Each declared `c` is local to its own execution of the inner loop; it is not available for use after that loop. These are source-level execution steps, not claims about machine instructions.

The reset point expresses the accumulator's purpose. Reset once per row to get separate row totals. Reset inside every inner body and you discard earlier values in the same row. Reset only before the outer loop and the value continues across rows. All three placements can stay within array bounds; only one meets the separate-row-total objective.

Worked example 1 Separate row totals and a whole-grid total

Predict the two row totals and the final combined total. The initialized one-dimensional `totals` array keeps the completed results; it is separate storage from `grid`.

```
#include <stdio.h>

int main(void)
{
    int grid[2][3] = {{4, 1, 7}, {2, 6, 3}};
    int totals[2] = {0, 0};
    int grand_total = 0;
    for (int r = 0; r < 2; r++) {
        int row_total = 0;
        for (int c = 0; c < 3; c++) {
            row_total += grid[r][c];
        }
        totals[r] = row_total;
        grand_total += row_total;
    }
    printf("%d %d | %d\n", totals[0], totals[1], grand_total);
    return 0;
}
```

The inner-body visit order and row totals are:

Visit number	r	c	Value read	row_total after addition
1	0	0	4	4
2	0	1	1	5
3	0	2	7	12
4	1	0	2	2
5	1	1	6	8
6	1	2	3	11

After row 0's inner loop, store 12 in `totals[0]` and add 12 to `grand_total`. After row 1's inner loop, store 11 in `totals[1]` and add 11 to the current 12. Output: 12 11 | 23, followed by a newline. `grid` is unchanged.

At the checkpoint just before an inner test, `r` identifies the current row and `row_total` is the sum of that row's columns already visited, from 0 through `c - 1`. At `c = 0`, this set is empty and the total is zero. At the failed test with `c = 3`, it covers the whole row. At each outer-test checkpoint, `grand_total` is the sum of all completed earlier rows; `totals` stores their individual totals. The unprocessed entries of `totals` retain their initialized zeros. These meanings explain both resets: `row_total` restarts per row, whereas `grand_total` retains completed rows.

The outer loop has two bodies and three condition tests. Each of its bodies runs an inner loop with three bodies and four condition tests. Altogether there are six inner bodies and eight inner condition tests, including a separate final failed test for each row. Both indices advance by one toward their finite bounds. No body runs at row 2 or column 3.

A second row total of 23 would carry the first row's total into the second. A first row total of 7 would discard earlier entries in that row. An order starting (0,0), (1,1) would incorrectly advance both indices together: the actual inner loop finishes all three columns before the outer update.

Worked example 2 Fix a column, then read before a selected update

This time only column 1 is selected. Add each visited integer's old value to `before_total`, and increase it by 4 only when it is below 5. The chosen column is within 0 through 2, and the loop visits the two existing rows. The shown small additions are representable in `int`.

```
#include <stdio.h>

int main(void)
{
    int grid[2][3] = {{3, 8, 1}, {6, 2, 5}};
    int column = 1;
    int before_total = 0;
    int changed = 0;
    for (int r = 0; r < 2; r++) {
        before_total += grid[r][column];
        if (grid[r][column] < 5) {
            grid[r][column] += 4;
            changed++;
        }
    }
    printf("%d %d | %d %d\n",
        before_total, changed, grid[0][column], grid[1][column]);
    return 0;
}
```

Visited row	Fixed column	Old value	before_total after addition	Value after possible store	changed
0	1	8	8	8	0
1	1	2	10	6	1

At row 0, add 8. The condition $8 < 5$ is false, so that element stays 8. At row 1, add the old 2, making `before_total = 10`; the condition is true, so store 6 and increase `changed` to 1. The array ends with rows {3, 8, 1} and {6, 6, 5}. Output: 10 1 | 8 6, followed by a newline.

Before each loop test, `before_total` contains the original values from the already visited rows of this selected column. `changed` counts how many of those positions received a store. Each selected position is visited once; other columns have not been read or changed by this loop. After the failed test at $r = 2$, these statements cover the whole chosen column. There are two bodies and three loop tests.

The new column total is 14, but `before_total` remains 10. A saved integer total does not automatically follow later element changes. Predicting 14 for that variable reverses the stated read/store order. Predicting three visits uses the column count as a row bound. No nested loop is needed for one fixed column: only one index varies.

Layout describes storage; loop order describes visits

The six integers in a 2-by-3 array are laid out in this order: `grid[0][0]`, `grid[0][1]`, `grid[0][2]`, `grid[1][0]`, `grid[1][1]`, `grid[1][2]`. This is row-major layout. Each row is contiguous, and the two rows are consecutive elements of the outer array. The right-hand index advances through a row before the next row begins.

If one `int` occupies `S` bytes, one row occupies `3 * S`, and the whole grid occupies `2 * 3 * S`. This is a symbolic size description, not executable pointer arithmetic or a claim that `S` is 4. It also does not authorize treating the inner arrays as a single six-element `int` array through a scalar pointer.

A program can choose a column-first visit order while storage remains row-major. A column-first scan would visit `(0,0)`, `(1,0)`, `(0,1)`, `(1,1)`, `(0,2)`, `(1,2)`. Interchanging loop nesting changes the sequence of accesses, not the declared shape or layout. Each access must still satisfy its own row and column bounds. We make no hardware timing claim from these small traces.

A pointer to a row is different from a pointer to an integer

Use the known shape `int grid[2][3]`. The declaration `int (*row)[3] = grid;` creates a pointer to an array of three `int` objects. The parentheses around `*row` matter: this declaration says that following row gives a three-integer array. We need only this one new pointer declaration, not general complicated declarators.

In this initializer, `grid` converts to a pointer to its first element. Its first element is the row `grid[0]`, so the result has type `int (*)[3]`. It does not skip directly to an `int *` or turn into an `int **`.

By contrast, `int *cell = grid[0];` selects row 0 and then converts that row expression to a pointer to its first integer, `grid[0][0]`. The conversion has happened at a different array level.

Expression in this stated context	Type or selected object
Actual object <code>grid</code>	Array of two arrays of three <code>int</code> objects
<code>grid</code> as initializer for row	<code>int (*)[3]</code> , pointing to row 0
<code>grid[0]</code> as initializer for cell	<code>int *</code> , pointing to row 0, column 0
<code>&grid[0]</code>	<code>int (*)[3]</code> , pointer to the first whole row
<code>&grid[0][0]</code>	<code>int *</code> , pointer to the first integer
<code>grid[1][2]</code>	The <code>int</code> object at row 1, column 2

Initially, with `row = grid`, `row + 1` points to row 1. One step counts one pointed-to object, which here is an entire three-integer row. With `cell = grid[0]`, `cell + 1` points to row 0, column 1. Its step counts one `int` within that inner array. Neither expression alone changes its pointer variable or any stored integer. An assignment such as `row = row + 1`; saves the new pointer position.

Following `row` gives the selected row; `(*row)[c]` then selects integer `c` in that row. With `row` initially at row 0, `row[1][2]` selects row 1, column 2. After moving `row` to row 1, use `(*row)[2]` or `row[0][2]` for its column 2. Pointer-relative subscripts are relative to the current pointer position, just as in the previous lesson.

The two layers also have different one-past positions. `grid + 2` is the valid endpoint after the two-row outer array, with no row there to access. `grid[0] + 3` is the valid endpoint after the three integers of row 0, with no integer there to read or write through that pointer. To reach the next row's first integer, select it through `grid[1][0]` or a valid row pointer. Do not dereference the scalar endpoint to cross that inner-array boundary. In particular, reading `grid[0][3]` is invalid even though another row follows. Contiguous placement does not cancel the subarray's bound.

Finally, `int **` means pointer to a pointer-to-`int` object. This `grid` stores rows of integers; it does not store an array of `int *` objects for a double pointer to follow. The converted row-pointer type `int (*)[3]` and `int **` are incompatible. Do not substitute one for the other or add a cast to hide that mismatch. Recognizing the mismatch is the full double-pointer scope of this lesson.

Count rows and columns where the actual array is available

At the actual `grid` declaration, `sizeof grid / sizeof grid[0]` is 2: whole-grid size divided by one-row size. `sizeof grid[0] / sizeof grid[0][0]` is 3: one-row size divided by one-integer size. The array operands of these `sizeof` expressions keep their array types instead of converting to pointers.

Below we store both counts in `size_t` variables, as introduced in the previous lesson; use `%zu` to print those variables. A pointer variable called `row` is still a pointer, so `sizeof row` does not give the grid's total size or recover its number of rows. The fixed three-column row type determines the kind of step; it does not store a runtime count of available rows. We use no two-dimensional function parameters in this lesson.

Practice before opening the solutions

These four original transfer tasks are unscored. Give the reasoning checkpoint, selected indices or pointer targets as well as the numeric answer. The text-fenced material in Practices 3 and 4 is for classification only; never compile or execute it.

Practice 1 A row total and a column total

Give the output and the two sequences of visited positions. Which position participates in both calculations? Does that make it count twice within either individual calculation?

```
#include <stdio.h>

int main(void)
{
    int grid[2][3] = {{5, 2, 9}, {4, 7, 1}};
    int row_total = 0;
    for (int c = 0; c < 3; c++) {
        row_total += grid[1][c];
    }
    int column_total = 0;
    for (int r = 0; r < 2; r++) {
        column_total += grid[r][2];
    }
    printf("%d %d\n", row_total, column_total);
    return 0;
}
```

Practice 2 Diagnose a misplaced reset in a column-first scan

The intention is three separate column totals. This program stays within bounds, but does it meet that intention? Give its actual output, exact inner-body visit order and the counts of outer and inner loop-condition tests. Then move the initialization of total to the correct place and give the repaired output.

```
#include <stdio.h>

int main(void)
{
    int grid[2][3] = {{2, 4, 1}, {6, 3, 5}};
    int totals[3] = {0, 0, 0};
    int total = 0;
    for (int c = 0; c < 3; c++) {
        for (int r = 0; r < 2; r++) {
            total += grid[r][c];
        }
        totals[c] = total;
    }
    printf("%d %d %d\n", totals[0], totals[1], totals[2]);
    return 0;
}
```

Practice 3 Repair the varying dimension's bound

Read this unsafe fragment only. The intention is the sum of column 2. Identify the first out-of-bounds pair it would attempt. Repair only the loop bound, then give the repaired total and explain why replacing the column by 1 would not repair the row bound.

```
int grid[2][3] = {{1, 3, 2}, {7, 4, 6}};
int column = 2;
int total = 0;
for (int r = 0; r < 3; r++) {
    total += grid[r][column];
}
```

Practice 4 Keep the row pointer and scalar pointer separate

Give the full output and the final targets of row and cell. Explain both size ratios and each pointer step.

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

int main(void)
{
    int grid[2][3] = {{5, 1, 4}, {2, 8, 6}};
    int (*row)[3] = grid;
    int *cell = grid[0] + 1;
    row = row + 1;
    (*row)[0] = *cell + 3;
    cell = cell + 1;
    *cell = (*row)[1] - 2;
    size_t rows = sizeof grid / sizeof grid[0];
    size_t columns = sizeof grid[0] / sizeof grid[0][0];
    printf("%zu %zu | %d %d %d | %d %d %d\n",
           rows, columns, grid[0][0], grid[0][1], grid[0][2],
           grid[1][0], grid[1][1], grid[1][2]);
    return 0;
}
```

Separately classify each of the following in the context of that live 2-by-3 grid. They are independent alternatives, not subsequent statements. Distinguish a valid endpoint, an invalid element read and an incompatible declaration.

```
grid + 2
grid[0] + 3
grid[0][3]
grid[1][3]
int **wrong = grid;
```

Full solutions and wrong-turn feedback

Practice 1 solution

The first loop fixes row 1. It visits $(1,0)$, $(1,1)$, $(1,2)$, reading 4, 7 and 1. Its running totals are 4, 11 and 12. At each test, `row_total` describes the visited prefix of row 1. The second loop fixes column 2, visiting $(0,2)$, $(1,2)$ and reading 9 then 1. Its independent running totals are 9 and 10.

Output: 12 10, followed by a newline. The grid is unchanged. Position $(1,2)$ participates once in each calculation; it is not visited twice by either loop. The accumulators answer two different questions and do not share their running state.

A first total of 10 would calculate a column instead of the requested row. A second total of 22 would start with the first loop's completed 12 instead of the new zero. Each loop's varying index must use its own dimension's bound: three columns for the first loop, two rows for the second.

Practice 2 solution

The visits are $(0,0)$, $(1,0)$, $(0,1)$, $(1,1)$, $(0,2)$, $(1,2)$. The values read are 2, 6, 4, 3, 1 and 5. Without a per-column reset, the running totals are 2, 8, 12, 15, 16 and 21. The completed columns therefore store 8, 15 and 21. Actual output: 8 15 21, followed by a newline.

The outer loop runs three bodies and four tests. Each inner loop runs two bodies and three tests, so there are six inner bodies and nine inner tests overall. Reaching the inner declaration restarts `r` at 0 for each new column. The failed inner test at `r = 2` does not end the outer loop.

Move `int total = 0;` inside the outer body, before the inner loop:

```
#include <stdio.h>

int main(void)
{
    int grid[2][3] = {{2, 4, 1}, {6, 3, 5}};
    int totals[3] = {0, 0, 0};
    for (int c = 0; c < 3; c++) {
        int total = 0;
        for (int r = 0; r < 2; r++) {
            total += grid[r][c];
        }
        totals[c] = total;
    }
    printf("%d %d %d\n", totals[0], totals[1], totals[2]);
    return 0;
}
```

Now each inner-test checkpoint says: `total` is the sum of the already visited rows of the current column only. Column 0 has totals 2 then 8; column 1 starts afresh, giving 4 then 7; column 2 starts afresh, giving 1 then 6. Repaired output: 8 7 6, followed by a newline. Visit order, loop-test counts and grid values are unchanged.

Resetting inside the inner body would instead keep only each column's last value, producing 6, 3 and 5. Swapping the loop bounds without changing what the indices select could create invalid accesses. The original issue is accumulator lifetime and reset placement, not layout or a pointer conversion.

Practice 3 solution

The valid row indices are 0 and 1, while the chosen column 2 is valid in each row. The proposed loop would first attempt an invalid access at (2, 2): a third row does not exist. It has undefined behavior, so C11 prescribes no numeric output or guaranteed partial total for the original fragment. Moving to column 1 still leaves a nonexistent row 2.

Use `r < 2`. The repaired complete program is:

```
#include <stdio.h>

int main(void)
{
    int grid[2][3] = {{1, 3, 2}, {7, 4, 6}};
    int column = 2;
    int total = 0;
    for (int r = 0; r < 2; r++) {
        total += grid[r][column];
    }
    printf("%d\n", total);
    return 0;
}
```

It reads 2 and 6, with running totals 2 and 8. Output: 8, followed by a newline. It has two bodies and three tests; the failed test at row 2 prevents an element access there. The fix meets the full chosen-column objective. A bound of 1 would avoid the invalid access but would omit the second row. A bound of 3 came from the wrong dimension, not from a lack of a third integer in each existing row.

Practice 4 solution

Initially `row` points to row 0, while `cell` points to row 0, column 1. Moving `row` by 1 selects the next whole row, row 1, and leaves `cell` unchanged. The store `(*row)[0] = *cell + 3` reads the 1 in row 0, column 1, and writes 4 into row 1, column 0.

Moving `cell` by 1 stays within row 0 and selects column 2. Then `(*row)[1]` reads row 1, column 1, whose value is 8. The subtraction gives 6, which is stored through `cell` into row 0, column 2. Final rows: {5, 1, 6} and {4, 8, 6}. The final pointer targets are the whole row 1 for `row` and row 0, column 2 for `cell`.

The first size ratio divides two-row storage by one-row storage and yields 2. The second divides a three-integer row by one integer and yields 3. Neither requires a numeric byte size. Both counts are stored in `size_t` variables, so their print formats are `%zu`; the six element values are `int`, printed with `%d`. Output: 2 3 | 5 1 6 | 4 8 6, followed by a newline.

Independent classifications:

- `grid + 2`: valid one-past pointer for the outer array of rows; no row exists there to access

- `grid[0] + 3`: valid one-past pointer for row 0's integer array; no integer is readable or writable through this endpoint
- `grid[0][3]`: invalid integer read beyond row 0; the next row's contiguous storage does not make column 3 valid
- `grid[1][3]`: invalid integer read beyond row 1; there is no fourth integer in that row either
- `int **wrong = grid;` incompatible pointer initialization that violates a C constraint and requires a diagnostic. `grid` supplies `int (*)[3]`, not `int **`. This is not a valid C11 declaration to run for an output

Predicting that the first store changes row 0 ignores the row-sized step. Predicting that moving `cell` also moves `row` confuses independent pointer objects. Treating the two invalid column reads as part of a six-integer flat scan discards the inner-array bounds. Keep the type, current target and governing array together.

Quick reference

- `int grid[2][3]` is an array of two rows; each row is an array of three `int` objects
- `grid[r][c]` selects a row, then an integer; check both dimension bounds separately
- In a row-outer nested loop, finish the inner column loop before the next outer row update
- Reaching an inner loop again restarts its initializer; reset a separate total once per row or column, as intended
- State an accumulator's meaning at a precise checkpoint, including whether it contains old or updated values
- A fixed-column scan varies the row index and uses the row count as its bound
- Row-major storage does not force row-first visits or permit scalar-pointer reads across inner-array boundaries
- `grid` converts to `int (*)[3]`; a selected row such as `grid[0]` can convert to `int *`; neither makes this `grid` an `int **`
- One row-pointer step selects the next row; one scalar-pointer step selects the next integer within its own row
- At the actual array, whole size / row size counts rows, and row size / integer size counts columns; use `size_t` and `%zu`

Before moving on

Trace a nested loop without advancing both indices together. Explain the chosen reset point, the two access bounds and the type that determines each pointer step. You are ready for the integrated learning review when you can combine those explanations with the earlier lessons' copied values, caller-visible stores and one-past reasoning. The review adds no new teaching topic. This module remains introductory array and pointer reasoning, not complete C or the complete GATE CS syllabus.

Semantic fact checks: [WG14 N1570](#), clauses 5.1.1.3p1, 6.2.5p20, 6.3.2.1p3, 6.5.2.1p2–3, 6.5.3.2p3–4, 6.5.3.4, 6.5.6p8, 6.5.16.1p1, 6.7.6.2, 6.7.9p11, p17–21, 6.8p3, 6.8.5.3p1, 7.19 and 7.21.6.1. All teaching explanations, examples, traces and practice are original. This attribution does not certify historical source accounting or global rights clearance.

Lesson 06 · हिन्दी

द्वि-आयामी ऐरे में पंक्तियाँ और कॉलम

एक ऐरे के भीतर छोटे ऐरे हो सकते हैं

एक-आयामी ऐरे में हर तत्व का एक इंडेक्स होता है। अब मान लें कि हर पंक्ति में तीन पूर्णांक हैं और ऐसी दो पंक्तियाँ हैं। हम अब भी एक समय पर एक ऑब्जेक्ट चुनते हैं: पहले पंक्ति, फिर उस पंक्ति के भीतर पूर्णांक। इन दोनों चयनों को अलग रखने से लूप, पॉइंटर के प्रकार और सीमाएँ समझना आसान होता है।

इस पाठ में आप द्वि-आयामी इनिशियलाइज़र पढ़ेंगे, नेस्टेड लूप का ट्रेस करेंगे, संचयक को सही जगह रीसेट करेंगे, चुने हुए कॉलम का स्कैन करेंगे और पंक्तियों के बीच की चाल को पूर्णांकों के बीच की चाल से अलग पहचानेंगे। हर पॉइंटर को एक-दूसरे की जगह इस्तेमाल होने वाला एड्रेस मानने के बजाय पंक्ति का प्रकार समझाएँगे।

पहले से आवश्यक बातें: इनिशियलाइज़र किए ऐरे और वैध इंडेक्स; लूप-जाँच, अपडेट और संचयक; पॉइंटर के लक्ष्य और कॉपी हुए मान; ऐरे से पॉइंटर में रूपांतरण; तत्वों में गिनी पॉइंटर की चाल, one-past सीमा, sizeof, size_t और %zu। पहले का कॉल-मॉडल नहीं बदलता, लेकिन इस पाठ में नए सहायक फ़ंक्शन के इंटरफ़ेस की जरूरत नहीं है।

C11 पढ़ाने के लिए हमारा चुना संस्करण है, GATE पाठ्यक्रम का निर्धारित संस्करण नहीं। यहाँ सभी वास्तविक ऐरे के आयाम धनात्मक स्थिरांक हैं और उनके int तत्व इनिशियलाइज़ किए गए हैं। int की सभी गणनाएँ -32767 से 32767 के भीतर रहती हैं। हर पूरा प्रोग्राम स्वतंत्र है। स्पष्ट रूप से चिह्नित वर्गीकरण-अंश कभी कंपाइल या निष्पादित न करें। int का कोई विशेष बाइट आकार या पॉइंटर चौड़ाई नहीं मानी गई है। परिवर्तनीय लंबाई के ऐरे, मेमोरी आवंटन, सामान्य डबल-पॉइंटर प्रोग्रामिंग और मैट्रिक्स एल्गोरिदम इस पाठ के बाहर हैं।

दोनों आयाम और इनिशियलाइज़र के दोनों स्तर पढ़ें

`int grid[2][3] = {{4, 1, 7}, {2, 6, 3}};` की घोषणा ऐसे पढ़ें: दो तत्वों का ऐरे, जिसका हर तत्व खुद तीन int ऑब्जेक्टों का ऐरे है। बाहरी ऐरे का तत्व पूरी पंक्ति है। पंक्ति का तत्व एक पूर्णांक है।

पंक्ति इंडेक्स	कॉलम 0	कॉलम 1	कॉलम 2
0	4	1	7
1	2	6	3

भीतर के पहले ब्रेस-समूह से पंक्ति 0 इनिशियलाइज़ होती है और दूसरे से पंक्ति 1। आयाम गिनतियाँ हैं: दो पंक्तियाँ और हर पंक्ति में तीन कॉलम। वे अंतिम वैध इंडेक्स नहीं हैं।

- `grid[0]` तीन पूर्णांकों वाली पहली पंक्ति चुनता है
- `grid[1][2]` पंक्ति 1 का कॉलम 2 चुनता है, जिसका वर्तमान मान 3 है
- `grid[0][1]` दूसरा पूर्णांक चुनता है, जिसका वर्तमान मान 1 है
- `grid[r][c]` पढ़ने या लिखने के लिए $0 \leq r \leq 1$ और $0 \leq c \leq 2$, दोनों चाहिए

पहले इंडेक्स का अर्थ “जिस आयाम का स्कैन हो रहा है” नहीं है। इस घोषणा में वह हमेशा पंक्ति चुनता है। कॉलम का स्कैन करते समय कॉलम इंडेक्स स्थिर रखकर पंक्ति इंडेक्स बदलते हैं। दोनों इंडेक्स की जगह बदलने से चुना ऑब्जेक्ट बदलता है और दूसरी सीमा भी टूट सकती है।

ब्रेस का समूह छोड़े हुए मान भी स्पष्ट करता है। `int ready[2][3] = {{5}, {2, 4}};` की शुरुआती पंक्तियाँ `{5, 0, 0}` और `{2, 4, 0}` हैं। शून्य इनिशियलाइज़र के नियम से आते हैं, automatic स्टोरेज के बारे में किसी सामान्य गारंटी से नहीं। बिना इनिशियलाइज़र वाली automatic घोषणा को ये गारंटी वाले शून्य नहीं मिलते। हमारे निष्पादन योग्य प्रोग्रामों में पढ़ने से पहले तत्व इनिशियलाइज़र हैं।

नेस्टेड लूप में बाहरी इंडेक्स बढ़ने से पहले भीतरी काम पूरा होता है

नेस्टेड लूप किसी दूसरे लूप की बॉडी के भीतर का लूप है। पंक्ति स्कैन करते समय `r` स्थिर रखें और `c` को 0, 1 और 2 पर जाने दें। पूरा भीतरी लूप समाप्त होने के बाद ही बाहरी लूप `r` बढ़ाता है।

पहला हल किया हुआ प्रोग्राम इस निष्पादन-क्रम से पढ़ें:

1. बाहरी `r` को 0 से इनिशियलाइज़ करें और `r < 2` जाँचें
2. उस पंक्ति की बाहरी बॉडी में जाएँ, `row_total` को 0 से इनिशियलाइज़ करें और भीतरी लूप तक पहुँचें
3. भीतरी `c` को 0 से इनिशियलाइज़ करें, `c < 3` जाँचें, तत्व संसाधित करें और फिर `c` अपडेट करें
4. भीतरी जाँच/बॉडी/अपडेट तब तक दोहराएँ जब तक `c = 3` पर जाँच असफल न हो
5. बाहरी बॉडी के बाकी स्टेटमेंट चलाएँ, जिनमें इस पंक्ति का योग सहेजना भी है
6. बाहरी बॉडी पूरी करें, `r` अपडेट करें और बाहरी शर्त फिर जाँचें

अगली सफल बाहरी जाँच पर बॉडी फिर `int row_total = 0;` तक पहुँचती है। इससे नई पंक्ति का योग शून्य से शुरू होता है। फिर से `for (int c = 0; ...)` तक पहुँचने पर उस भीतरी स्कैन की शुरुआत भी कॉलम 0 से होती है। पिछले भीतरी स्कैन का अंतिम मान 3 अगली पंक्ति में आगे न ले जाएँ। हर घोषित `c` भीतरी लूप के अपने निष्पादन तक स्थानीय है; उस लूप के बाद उसका उपयोग नहीं कर सकते। ये स्रोत-स्तर के निष्पादन चरण हैं, मशीन निर्देशों के बारे में दावे नहीं।

रीसेट की जगह संचयक का उद्देश्य बताती है। अलग-अलग पंक्ति-योग के लिए हर पंक्ति पर एक बार रीसेट करें। हर भीतरी बॉडी में रीसेट करेंगे तो उसी पंक्ति के पहले मान छूट जाएँगे। केवल बाहरी लूप से पहले रीसेट करेंगे तो मान अगली पंक्तियों में भी चलता रहेगा। तीनों स्थान ऐसे की सीमा के भीतर रह सकते हैं; केवल एक अलग-अलग पंक्ति-योग का उद्देश्य पूरा करता है।

हल किया हुआ उदाहरण 1 अलग पंक्ति-योग और पूरे ग्रिड का योग

दोनों पंक्ति-योग और अंत का संयुक्त योग बताएँ। इनिशियलाइज़ किया एक-आयामी totals ऐरे पूरे हुए परिणाम रखता है; उसका स्टोरेज grid से अलग है।

```
#include <stdio.h>

int main(void)
{
    int grid[2][3] = {{4, 1, 7}, {2, 6, 3}};
    int totals[2] = {0, 0};
    int grand_total = 0;
    for (int r = 0; r < 2; r++) {
        int row_total = 0;
        for (int c = 0; c < 3; c++) {
            row_total += grid[r][c];
        }
        totals[r] = row_total;
        grand_total += row_total;
    }
    printf("%d %d | %d\n", totals[0], totals[1], grand_total);
    return 0;
}
```

भीतरी बॉडी में देखने का क्रम और पंक्ति-योग हैं:

देखने की संख्या	r	c	पढ़ा मान	जोड़ के बाद row_total
1	0	0	4	4
2	0	1	1	5
3	0	2	7	12
4	1	0	2	2
5	1	1	6	8
6	1	2	3	11

पंक्ति 0 का भीतरी लूप पूरा होने पर totals[0] में 12 रखें और grand_total में 12 जोड़ें। पंक्ति 1 का भीतरी लूप पूरा होने पर totals[1] में 11 रखें और वर्तमान 12 में 11 जोड़ें। आउटपुट: 12 11 | 23, फिर नई पंक्ति। grid अपरिवर्तित है।

भीतरी जाँच से ठीक पहले के बिंदु पर r वर्तमान पंक्ति पहचानता है और row_total उस पंक्ति के पहले ही देखे गए कॉलम 0 से c - 1 तक का योग है। c = 0 पर यह समूह खाली है और योग शून्य है। c = 3 वाली असफल जाँच पर इसमें पूरी पंक्ति शामिल है। हर बाहरी-जाँच बिंदु पर grand_total पहले पूरी हो चुकी सभी पंक्तियों का योग है; totals उनके अलग-अलग योग रखता है। totals की अभी संसाधित नहीं हुई प्रविष्टियाँ अपने शुरुआती शून्य रखती हैं। इन अर्थों से दोनों शुरुआती व्यवस्थाएँ स्पष्ट होती हैं: row_total हर पंक्ति पर फिर शुरू होता है, जबकि grand_total पूरी हो चुकी पंक्तियों को सँभालकर रखता है।

बाहरी लूप में दो बॉडी और शर्त की तीन जाँचें हैं। उसकी हर बॉडी में भीतरी लूप की तीन बॉडी और चार जाँचें होती हैं। कुल मिलाकर छह भीतरी बॉडी और आठ भीतरी शर्त-जाँचें हैं; हर पंक्ति के लिए अलग अंतिम असफल जाँच भी गिनी गई है। दोनों इंडेक्स अपनी सीमित सीमा की ओर एक-एक बढ़ते हैं। पंक्ति 2 या कॉलम 3 पर कोई बॉडी नहीं चलती।

दूसरी पंक्ति का योग 23 बताना पहली पंक्ति का योग दूसरी में आगे ले जाना होगा। पहली पंक्ति का योग 7 बताना उस पंक्ति की पिछली प्रविष्टियाँ छोड़ देता है। (0,0), (1,1) से शुरू होने वाला क्रम दोनों इंडेक्स को गलत तरह से एक साथ बढ़ाता है: वास्तविक भीतरी लूप बाहरी अपडेट से पहले तीनों कॉलम पूरे करता है।

हल किया हुआ उदाहरण 2 कॉलम स्थिर रखें, फिर चुने बदलाव से पहले पढ़ें

इस बार केवल कॉलम 1 चुना गया है। हर देखे पूर्णांक का पुराना मान `before_total` में जोड़ें, और केवल 5 से कम होने पर उसमें 4 बढ़ाएँ। चुना कॉलम 0 से 2 के भीतर है और लूप दोनों मौजूद पंक्तियों पर जाता है। दिखाए गए छोटे जोड़ `int` में समाते हैं।

```
#include <stdio.h>

int main(void)
{
    int grid[2][3] = {{3, 8, 1}, {6, 2, 5}};
    int column = 1;
    int before_total = 0;
    int changed = 0;
    for (int r = 0; r < 2; r++) {
        before_total += grid[r][column];
        if (grid[r][column] < 5) {
            grid[r][column] += 4;
            changed++;
        }
    }
    printf("%d %d | %d %d\n",
           before_total, changed, grid[0][column], grid[1][column]);
    return 0;
}
```

देखी पंक्ति	स्थिर कॉलम	पुराना मान	जोड़ के बाद <code>before_total</code>	संभव लेखन के बाद मान	<code>changed</code>
0	1	8	8	8	0
1	1	2	10	6	1

पंक्ति 0 पर 8 जोड़ें। शर्त $8 < 5$ असत्य है, इसलिए तत्व 8 रहता है। पंक्ति 1 पर पुराना 2 जोड़ने से `before_total = 10` होता है; शर्त सत्य है, इसलिए 6 लिखें और `changed` को 1 करें। अंतिम ऐरे की पंक्तियाँ {3, 8, 1} और {6, 6, 5} हैं। आउटपुट: 10 1 | 8 6, फिर नई पंक्ति।

हर लूप-जाँच से पहले `before_total` में चुने कॉलम की पहले ही देखी गई पंक्तियों के मूल मानों का योग है। `changed` गिनता है कि उनमें कितनी स्थितियों पर लेखन हुआ। हर चुनी स्थिति केवल एक बार देखी जाती है; इस लूप ने दूसरे कॉलम न पढ़े हैं, न बदले हैं। $r = 2$ पर असफल जाँच के बाद ये बातें पूरे चुने कॉलम के लिए सही हैं। दो बॉडी और तीन लूप-जाँचें हैं।

कॉलम का नया योग 14 है, लेकिन `before_total` 10 रहता है। सहेजा हुआ पूर्णांक योग बाद के तत्व-बदलावों के साथ अपने आप नहीं बदलता। इस चर का मान 14 बताना बताए हुए पढ़ने/लिखने के क्रम को उलट देता है। तीन बार जाना मानना कॉलम की गिनती को पंक्ति की सीमा बना देता है। एक स्थिर कॉलम के लिए नेस्टेड लूप आवश्यक नहीं है: केवल एक इंडेक्स बदलता है।

लेआउट स्टोरेज बताता है; लूप का क्रम देखने का क्रम बताता है

2-बाय-3 ऐरे के छह पूर्णांक इस क्रम में रखे होते हैं: `grid[0][0]`, `grid[0][1]`, `grid[0][2]`, `grid[1][0]`, `grid[1][1]`, `grid[1][2]`। इसे row-major लेआउट कहते हैं। हर पंक्ति लगातार स्टोरेज में है और दोनों पंक्तियाँ बाहरी ऐरे के लगातार तत्व हैं। अगली पंक्ति शुरू होने से पहले दायीं इंडेक्स एक पंक्ति में आगे बढ़ता है।

यदि एक `int` का आकार `S` बाइट है, तो एक पंक्ति का आकार $3 * S$ और पूरे ग्रिड का आकार $2 * 3 * S$ है। यह आकार का प्रतीकात्मक वर्णन है, निष्पादन योग्य पॉइंटर अंकगणित या `S` को 4 मानने का दावा नहीं। इससे scalar पॉइंटर के रास्ते भीतरी ऐरे को छह-तत्वों वाला एक ही `int` ऐरे मानने की अनुमति भी नहीं मिलती।

प्रोग्राम कॉलम-पहले देखने का क्रम चुन सकता है, जबकि स्टोरेज row-major ही रहता है। कॉलम-पहले स्कैन का क्रम होगा $(0,0)$, $(1,0)$, $(0,1)$, $(1,1)$, $(0,2)$, $(1,2)$ । लूपों की नेस्टिंग बदलने से अभिगम का क्रम बदलता है, घोषित आकार या लेआउट नहीं। हर अभिगम को उसकी पंक्ति और कॉलम, दोनों की सीमाओं के भीतर रहना चाहिए। इन छोटे ट्रेस से हम हार्डवेयर के समय का कोई दावा नहीं करते।

पंक्ति का पॉइंटर पूर्णांक के पॉइंटर से अलग है

ज्ञात आकार `int grid[2][3]` लें। घोषणा `int (*row)[3] = grid;` तीन `int` ऑब्जेक्टों के ऐरे का पॉइंटर बनाती है। `*row` के चारों ओर गोल कोष्ठक महत्वपूर्ण हैं: घोषणा कहती है कि `row` के लक्ष्य तक जाने पर तीन पूर्णाकों का ऐरे मिलता है। हमें केवल यह एक नई पॉइंटर घोषणा चाहिए, सामान्य जटिल घोषणाएँ नहीं।

इस इनिशियलाइज़र में `grid` अपने पहले तत्व के पॉइंटर में बदलता है। उसका पहला तत्व पंक्ति `grid[0]` है, इसलिए परिणाम का प्रकार `int (*)[3]` है। वह सीधे `int *` तक नहीं पहुँच जाता और न `int **` बनता है।

इसके विपरीत, `int *cell = grid[0];` पहले पंक्ति 0 चुनता है और फिर उस पंक्ति के व्यंजक को उसके पहले पूर्णांक `grid[0][0]` के पॉइंटर में बदलता है। रूपांतरण ऐरे के दूसरे स्तर पर हुआ है।

बताए हुए संदर्भ का व्यंजक	प्रकार या चुना ऑब्जेक्ट
वास्तविक ऑब्जेक्ट <code>grid</code>	दो ऐरे का ऐरे; प्रत्येक में तीन <code>int</code> ऑब्जेक्ट
<code>row</code> के इनिशियलाइज़र के रूप में <code>grid</code>	<code>int (*)[3]</code> , पंक्ति 0 की ओर
<code>cell</code> के इनिशियलाइज़र के रूप में <code>grid[0]</code>	<code>int *</code> , पंक्ति 0, कॉलम 0 की ओर
<code>&grid[0]</code>	<code>int (*)[3]</code> , पूरी पहली पंक्ति का पॉइंटर
<code>&grid[0][0]</code>	<code>int *</code> , पहले पूर्णांक का पॉइंटर
<code>grid[1][2]</code>	पंक्ति 1, कॉलम 2 का <code>int</code> ऑब्जेक्ट

शुरुआत में `row = grid` होने पर `row + 1` पंक्ति 1 की ओर जाता है। एक कदम उस प्रकार का एक पूरा ऑब्जेक्ट गिनता है जिसकी ओर पॉइंटर है; यहाँ वह तीन पूर्णांकों की पूरी पंक्ति है। `cell = grid[0]` होने पर `cell + 1` पंक्ति 0, कॉलम 1 की ओर जाता है। उसका कदम उसी भीतरी ऐरे में एक `int` गिनता है। इन दोनों व्यंजकों में से कोई भी अकेले अपने पॉइंटर चर या रखे हुए पूर्णांक नहीं बदलता।
`row = row + 1`; जैसा असाइनमेंट पॉइंटर की नई स्थिति सहेजता है।

`row` के लक्ष्य तक जाने पर चुनी हुई पंक्ति मिलती है; फिर `(*row)[c]` उस पंक्ति का पूर्णांक `c` चुनता है। शुरुआत में `row` पंक्ति 0 पर हो, तो `row[1][2]` पंक्ति 1, कॉलम 2 चुनता है। `row` को पंक्ति 1 पर ले जाने के बाद उसके कॉलम 2 के लिए `(*row)[2]` या `row[0][2]` लें। पॉइंटर के सापेक्ष सबस्क्रिप्ट वर्तमान पॉइंटर स्थिति से गिना जाता है, ठीक पिछले पाठ की तरह।

दोनों स्तरों की one-past स्थितियाँ भी अलग हैं। `grid + 2` दो-पंक्तियों वाले बाहरी ऐरे के बाद का वैध सिरा है; वहाँ अभिगम के लिए कोई पंक्ति नहीं है। `grid[0] + 3` पंक्ति 0 के तीन पूर्णांकों के बाद का वैध सिरा है; उस पॉइंटर के रास्ते पढ़ने या लिखने के लिए वहाँ कोई पूर्णांक नहीं है। अगली पंक्ति के पहले पूर्णांक तक पहुँचने के लिए `grid[1][0]` या वैध पंक्ति-पॉइंटर से चुनें। भीतरी ऐरे की सीमा पार करने के लिए `scalar` सिरों को डीरेफ़रेंस न करें। विशेष रूप से, `grid[0][3]` पढ़ना अवैध है, भले ही अगली पंक्ति उसके बाद रखी हो। लगातार स्टोरेज होने से उप-ऐरे की सीमा खत्म नहीं होती।

अंत में, `int **` का अर्थ है किसी `int`-पॉइंटर ऑब्जेक्ट का पॉइंटर। यह ग्रिड पूर्णांकों की पंक्तियाँ रखता है; यह `int *` ऑब्जेक्टों का ऐरे नहीं रखता जिनके रास्ते डबल पॉइंटर चले। रूपांतरण से मिला पंक्ति-पॉइंटर प्रकार `int (*)[3]` और `int **` असंगत हैं। एक को दूसरे की जगह न रखें और प्रकार का अंतर छिपाने के लिए कास्ट न जोड़ें। इस अंतर की पहचान ही इस पाठ में डबल-पॉइंटर का पूरा दायरा है।

जहाँ वास्तविक ऐरे उपलब्ध हो, वहाँ पंक्तियाँ और कॉलम गिनें

वास्तविक `grid` की घोषणा वाले स्थान पर `sizeof grid / sizeof grid[0]` का मान 2 है: पूरे ग्रिड का आकार भाग एक पंक्ति का आकार। `sizeof grid[0] / sizeof grid[0][0]` का मान 3 है: एक पंक्ति का आकार भाग एक पूर्णांक का आकार। इन `sizeof` व्यंजकों के ऐरे ऑपरेंड पॉइंटर में बदलने के बजाय अपने ऐरे प्रकार रखते हैं।

नीचे दोनों गिनतियाँ पिछले पाठ में बताए `size_t` चरों में रखेंगे; उन चरों को छापने के लिए `%zu` लें। `row` नाम का पॉइंटर चर अब भी पॉइंटर है, इसलिए `sizeof row` ग्रिड का कुल आकार नहीं देता और न उसकी पंक्ति-संख्या खोजता है। तीन कॉलम वाला स्थिर पंक्ति-प्रकार कदम का प्रकार निर्धारित करता है; वह उपलब्ध पंक्तियों की रनटाइम गिनती नहीं रखता। इस पाठ में द्वि-आयामी फ़ंक्शन पैरामीटर नहीं लेते।

हल देखने से पहले अभ्यास करें

ये चार मौलिक ट्रांसफ़र अभ्यास बिना अंक के हैं। संख्यात्मक उत्तर के साथ तर्क का बिंदु, चुने इंडेक्स या पॉइंटर के लक्ष्य भी दें। अभ्यास 3 और 4 के पाठ-ब्लॉक केवल वर्गीकरण के लिए हैं; उन्हें कभी कंपाइल या निष्पादित न करें।

अभ्यास 1 एक पंक्ति का योग और एक कॉलम का योग

आउटपुट और देखी गई स्थितियों के दोनों क्रम बताएँ। कौन-सी स्थिति दोनों गणनाओं में आती है? क्या इससे किसी एक गणना के भीतर वह दो बार गिनी जाती है?

```
#include <stdio.h>

int main(void)
{
    int grid[2][3] = {{5, 2, 9}, {4, 7, 1}};
    int row_total = 0;
    for (int c = 0; c < 3; c++) {
        row_total += grid[1][c];
    }
    int column_total = 0;
    for (int r = 0; r < 2; r++) {
        column_total += grid[r][2];
    }
    printf("%d %d\n", row_total, column_total);
    return 0;
}
```

अभ्यास 2 कॉलम-पहले स्कैन में गलत जगह रखे रीसेट की पहचान

उद्देश्य तीन अलग-अलग कॉलम-योग निकालना है। यह प्रोग्राम सीमाओं के भीतर रहता है, लेकिन क्या वह उद्देश्य पूरा करता है? वास्तविक आउटपुट, भीतरी बॉडी का ठीक देखने का क्रम और बाहरी तथा भीतरी लूप-शर्त की जाँचों की गिनती दें। फिर total का इनिशियलाइज़ेशन सही जगह ले जाएँ और सुधरा आउटपुट दें।

```
#include <stdio.h>

int main(void)
{
    int grid[2][3] = {{2, 4, 1}, {6, 3, 5}};
    int totals[3] = {0, 0, 0};
    int total = 0;
    for (int c = 0; c < 3; c++) {
        for (int r = 0; r < 2; r++) {
            total += grid[r][c];
        }
        totals[c] = total;
    }
    printf("%d %d %d\n", totals[0], totals[1], totals[2]);
    return 0;
}
```

अभ्यास 3 बदलते आयाम की सीमा सुधारें

इस असुरक्षित अंश को केवल पढ़ें। उद्देश्य कॉलम 2 का योग है। पहली सीमा-बाहरी जोड़ी पहचानें जिस तक यह पहुँचने की कोशिश करेगा। केवल लूप की सीमा सुधारें, फिर सुधरा योग दें और समझाएँ कि कॉलम को 1 करने से पंक्ति की सीमा क्यों नहीं सुधरेगी।

```
int grid[2][3] = {{1, 3, 2}, {7, 4, 6}};
int column = 2;
int total = 0;
for (int r = 0; r < 3; r++) {
    total += grid[r][column];
}
```

अभ्यास 4 पंक्ति-पॉइंटर और scalar पॉइंटर अलग रखें

पूरा आउटपुट और row तथा cell के अंतिम लक्ष्य बताएँ। दोनों आकार-अनुपात और हर पॉइंटर कदम समझाएँ।

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

int main(void)
{
    int grid[2][3] = {{5, 1, 4}, {2, 8, 6}};
    int (*row)[3] = grid;
    int *cell = grid[0] + 1;
    row = row + 1;
    (*row)[0] = *cell + 3;
    cell = cell + 1;
    *cell = (*row)[1] - 2;
    size_t rows = sizeof grid / sizeof grid[0];
    size_t columns = sizeof grid[0] / sizeof grid[0][0];
    printf("%zu %zu | %d %d %d | %d %d %d\n",
           rows, columns, grid[0][0], grid[0][1], grid[0][2],
           grid[1][0], grid[1][1], grid[1][2]);
    return 0;
}
```

अलग से, उसी जीवित 2-बाय-3 ग्रिड के संदर्भ में नीचे की हर चीज़ का वर्गीकरण करें। ये स्वतंत्र विकल्प हैं, बाद में क्रम से चलने वाले स्टेटमेंट नहीं। वैध अंतिम सिरा, अवैध तत्व-पठन और असंगत घोषणा अलग पहचानें।

```
grid + 2
grid[0] + 3
grid[0][3]
grid[1][3]
int **wrong = grid;
```

पूरे हल और गलत रास्तों की प्रतिक्रिया

अभ्यास 1 का हल

पहला लूप पंक्ति 1 स्थिर रखता है। वह (1,0), (1,1), (1,2) पर जाकर 4, 7 और 1 पढ़ता है। उसके क्रमिक योग 4, 11 और 12 हैं। हर जाँच पर row_total पंक्ति 1 के पहले ही देखे शुरुआती हिस्से का वर्णन करता है। दूसरा लूप कॉलम 2 स्थिर रखकर (0,2), (1,2) पर जाता है और 9 फिर 1 पढ़ता है। उसके स्वतंत्र क्रमिक योग 9 और 10 हैं।

आउटपुट: 12 10, फिर नई पंक्ति। ग्रिड अपरिवर्तित है। स्थिति (1,2) हर गणना में एक बार आती है; कोई भी लूप उसे दो बार नहीं देखता। संचयक दो अलग प्रश्नों के उत्तर देते हैं और उनकी चलती हुई अवस्था साझा नहीं है।

पहला योग 10 बताना माँगी पंक्ति के बजाय कॉलम की गणना करता है। दूसरा योग 22 बताना नए शून्य के बजाय पहले लूप के पूरे हुए 12 से शुरू करता है। हर लूप के बदलते इंडेक्स को अपने आयाम की सीमा चाहिए: पहले के लिए तीन कॉलम, दूसरे के लिए दो पंक्तियाँ।

अभ्यास 2 का हल

देखने का क्रम है (0,0), (1,0), (0,1), (1,1), (0,2), (1,2)। पढ़े मान 2, 6, 4, 3, 1 और 5 हैं। हर कॉलम पर रीसेट न होने से क्रमिक योग 2, 8, 12, 15, 16 और 21 हैं। इसलिए कॉलम पूरे होने पर 8, 15 और 21 सहेजे जाते हैं। वास्तविक आउटपुट: 8 15 21, फिर नई पंक्ति।

बाहरी लूप की तीन बॉडी और चार जाँचें हैं। हर भीतरी लूप की दो बॉडी और तीन जाँचें हैं, इसलिए कुल छह भीतरी बॉडी और नौ भीतरी जाँचें हैं। भीतरी घोषणा तक फिर पहुँचने पर हर नए कॉलम के लिए r दोबारा 0 से शुरू होता है। r = 2 पर भीतरी जाँच असफल होने से बाहरी लूप समाप्त नहीं होता।

int total = 0; को बाहरी बॉडी के भीतर, भीतरी लूप से पहले ले जाएँ:

```
#include <stdio.h>

int main(void)
{
    int grid[2][3] = {{2, 4, 1}, {6, 3, 5}};
    int totals[3] = {0, 0, 0};
    for (int c = 0; c < 3; c++) {
        int total = 0;
        for (int r = 0; r < 2; r++) {
            total += grid[r][c];
        }
        totals[c] = total;
    }
    printf("%d %d %d\n", totals[0], totals[1], totals[2]);
    return 0;
}
```

अब हर भीतरी-जाँच बिंदु पर यह अर्थ सही है: total केवल वर्तमान कॉलम की पहले ही देखी गई पंक्तियों का योग है। कॉलम 0 के योग 2 फिर 8 हैं; कॉलम 1 नए सिरे से शुरू होकर 4 फिर 7 देता है; कॉलम 2 नए सिरे से शुरू होकर 1 फिर 6 देता है। सुधरा आउटपुट: 8 7 6, फिर नई पंक्ति। देखने का क्रम, लूप-जाँचों की गिनती और ग्रिड के मान नहीं बदलते।

भीतरी बॉडी के भीतर रीसेट करने पर हर कॉलम का केवल अंतिम मान बचेगा, जिससे 6, 3 और 5 मिलेंगे। इंडेक्स क्या चुनते हैं यह बदले बिना लूप की सीमाएँ बदलने से अवैध अभिगम हो सकते हैं। मूल समस्या संचयक के जीवनकाल और रीसेट की जगह की है, लेआउट या पॉइंटर रूपांतरण की नहीं।

अभ्यास 3 का हल

वैध पंक्ति इंडेक्स 0 और 1 हैं, जबकि चुना कॉलम 2 हर पंक्ति में वैध है। प्रस्तावित लूप का पहला अवैध अभिगम (2, 2) होगा: तीसरी पंक्ति मौजूद नहीं है। उसका व्यवहार अपरिभाषित है, इसलिए C11 मूल अंश के लिए कोई संख्यात्मक आउटपुट या गारंटी वाला आंशिक योग निर्धारित नहीं करता। कॉलम 1 पर जाने से भी पंक्ति 2 का अस्तित्व नहीं बनता।

$r < 2$ लें। पूरा सुधरा प्रोग्राम है:

```
#include <stdio.h>

int main(void)
{
    int grid[2][3] = {{1, 3, 2}, {7, 4, 6}};
    int column = 2;
    int total = 0;
    for (int r = 0; r < 2; r++) {
        total += grid[r][column];
    }
    printf("%d\n", total);
    return 0;
}
```

यह 2 और 6 पढ़ता है; क्रमिक योग 2 और 8 हैं। आउटपुट: 8, फिर नई पंक्ति। दो बॉडी और तीन जाँचें हैं; पंक्ति 2 पर असफल जाँच वहाँ तत्व तक पहुँचने से रोकती है। सुधार पूरे चुने कॉलम का उद्देश्य पूरा करता है। सीमा 1 करने से अवैध अभिगम तो बचेगा, लेकिन दूसरी पंक्ति छूट जाएगी। सीमा 3 गलत आयाम से आई थी, किसी मौजूद पंक्ति में तीसरा पूर्णांक न होने से नहीं।

अभ्यास 4 का हल

शुरुआत में row पंक्ति 0 की ओर है, जबकि cell पंक्ति 0, कॉलम 1 की ओर है। row में 1 का कदम अगली पूरी पंक्ति, यानी पंक्ति 1, चुनता है और cell नहीं बदलता। लेखन $(*row)[0] = *cell + 3$ पंक्ति 0, कॉलम 1 से 1 पढ़कर पंक्ति 1, कॉलम 0 में 4 लिखता है।

cell को 1 आगे बढ़ाने से वह पंक्ति 0 के भीतर ही कॉलम 2 चुनता है। फिर $(*row)[1]$ पंक्ति 1, कॉलम 1 पढ़ता है, जिसका मान 8 है। घटाने से 6 मिलता है, जो cell के रास्ते पंक्ति 0, कॉलम 2 में लिखा जाता है। अंतिम पंक्तियाँ: {5, 1, 6} और {4, 8, 6}। अंतिम पॉइंटर लक्ष्य हैं: row के लिए पूरी पंक्ति 1 और cell के लिए पंक्ति 0, कॉलम 2।

पहला आकार-अनुपात दो पंक्तियों के स्टोरेज को एक पंक्ति के स्टोरेज से भाग देकर 2 देता है। दूसरा तीन-पूर्णांक वाली पंक्ति को एक पूर्णांक से भाग देकर 3 देता है। दोनों में संख्यात्मक बाइट आकार की जरूरत नहीं। दोनों गिनतियाँ `size_t` चरों में रखी हैं, इसलिए उनके प्रिंट फ़ॉर्मैट `%zu` हैं; छह तत्वों के मान `int` हैं, जिन्हें `%d` से छापते हैं। आउटपुट: 2 3 | 5 1 6 | 4 8 6, फिर नई पंक्ति।

स्वतंत्र वर्गीकरण:

- `grid + 2`: पंक्तियों के बाहरी एरे का वैध one-past पॉइंटर; वहाँ अभिगम के लिए कोई पंक्ति नहीं है
- `grid[0] + 3`: पंक्ति 0 के पूर्णांक एरे का वैध one-past पॉइंटर; इस सिरे के रास्ते कोई पूर्णांक पढ़ना या लिखना वैध नहीं है

- `grid[0][3]`: पंक्ति 0 के बाहर अवैध पूर्णांक-पठन; अगली पंक्ति का लगातार स्टोरेज कॉलम 3 को वैध नहीं बनाता
- `grid[1][3]`: पंक्ति 1 के बाहर अवैध पूर्णांक-पठन; उस पंक्ति में भी चौथा पूर्णांक नहीं है
- `int **wrong = grid;` असंगत पॉइंटर इनिशियलाइज़ेशन, जो C की constraint तोड़ता है और जिसके लिए diagnostic आवश्यक है। `grid, int (*)[3]` देता है, `int **` नहीं। आउटपुट जानने के लिए चलाने वाली यह वैध C11 घोषणा नहीं है

पहले लेखन से पंक्ति 0 बदलना बताने पर पंक्ति के आकार वाला कदम छूट जाता है। `cell` को चलाने पर `row` भी चलना मानना स्वतंत्र पॉइंटर ऑब्जेक्टों को मिलाता है। दोनों अवैध कॉलम-पठनों को छह-पूर्णांकों के सपाट स्कैन का हिस्सा मानना भीतरी ऐरे की सीमाएँ हटा देता है। प्रकार, वर्तमान लक्ष्य और लागू ऐरे को साथ रखें।

त्वरित संदर्भ

- `int grid[2][3]` दो पंक्तियों का ऐरे है; हर पंक्ति तीन `int` ऑब्जेक्टों का ऐरे है
- `grid[r][c]` पहले पंक्ति, फिर पूर्णांक चुनता है; दोनों आयामों की सीमाएँ अलग जाँचें
- पंक्ति-बाहरी नेस्टेड लूप में अगली बाहरी पंक्ति-अपडेट से पहले भीतरी कॉलम लूप पूरा करें
- भीतरी लूप तक फिर पहुँचने पर उसका इनिशियलाइज़र फिर शुरू होता है; उद्देश्य के अनुसार हर पंक्ति या कॉलम पर अलग योग एक बार रीसेट करें
- किसी निश्चित बिंदु पर संचयक का अर्थ बताएँ, जिसमें पुराने या अपडेट किए मानों का अंतर भी हो
- स्थिर कॉलम के स्कैन में पंक्ति इंडेक्स बदलता है और सीमा पंक्तियों की गिनती होती है
- row-major स्टोरेज पंक्ति-पहले देखने को अनिवार्य नहीं करता और scalar पॉइंटर से भीतरी ऐरे की सीमा पार करके पढ़ने की अनुमति नहीं देता
- `grid, int (*)[3]` में बदलता है; `grid[0]` जैसी चुनी पंक्ति `int *` में बदल सकती है; इनमें से कोई इस ग्रिड को `int **` नहीं बनाता
- पंक्ति-पॉइंटर का एक कदम अगली पंक्ति चुनता है; scalar पॉइंटर का एक कदम उसकी अपनी पंक्ति का अगला पूर्णांक चुनता है
- वास्तविक ऐरे पर पूरा आकार / पंक्ति का आकार पंक्तियाँ गिनता है, और पंक्ति का आकार / पूर्णांक का आकार कॉलम गिनता है; `size_t` और `%zu` लें

आगे बढ़ने से पहले

दोनों इंडेक्स को एक साथ बढ़ाए बिना नेस्टेड लूप का ट्रेस करें। चुनी रीसेट-जगह, अभिगम की दोनों सीमाएँ और हर पॉइंटर कदम तय करने वाला प्रकार समझाएँ। जब इन व्याख्याओं को पिछले पाठों के कॉपी हुए मानों, कॉलर में दिखने वाले लेखन और one-past तर्क से जोड़ सकें तब आप एकीकृत पुनराभ्यास के लिए तैयार हैं। पुनराभ्यास कोई नया पढ़ाई-विषय नहीं जोड़ता। यह मॉड्यूल ऐरे और पॉइंटर के शुरुआती तर्क तक सीमित है, पूरी C या पूरा GATE CS पाठ्यक्रम नहीं।

अर्थ-संबंधी तथ्य-जाँच: [WG14 N1570](#), खंड 5.1.1.3p1, 6.2.5p20, 6.3.2.1p3, 6.5.2.1p2-3, 6.5.3.2p3-4, 6.5.3.4, 6.5.6p8, 6.5.16.1p1, 6.7.6.2, 6.7.9p11, p17-21, 6.8p3, 6.8.5.3p1, 7.19 और 7.21.6.1। पढ़ाने की सभी व्याख्याएँ, उदाहरण, ट्रेस और अभ्यास मौलिक हैं। यह उल्लेख पुराने स्रोत-हिसाब या सभी अधिकारों की समग्र मंजूरी का प्रमाण नहीं है।