Arrays and pointers learning review

REVIEW COMPANION · पुनराभ्यास पुस्तिका

English questions · 2
English answers and solutions · 9
हिंदी प्रश्न · 15
हिंदी उत्तर और पूरे हल · 23

Use the review to find a reasoning gap

Solve these twelve original questions after M2-L01 through M2-L06. They revisit array state, scans, pointer targets, calls, boundaries and two-dimensional rows. The review introduces no new examinable concept. It samples each lesson's core outcomes twice; it does not test every individual objective or replace the teaching exercises.

This is an untimed educational review. The maximum is 12 marks: +1 for each correct answer and 0 for an incorrect or unattempted answer. There are no negative marks, no partial marks and no pass threshold. These local learning settings are not a statement of official GATE marking rules. The questions are original, not official previous-year questions, and this is not a full GATE mock.

- MCQ means choose exactly one option. There are five MCQs
- MSQ means choose the complete set of true options. There are four MSQs. A missing correct option or an extra incorrect option gives 0 for that item
- NAT means supply one signed decimal integer. There are three NATs. Use ordinary digits 0–9 and the keyboard minus sign `-` for negative values; surrounding spaces, a leading plus sign and leading zeros are accepted. Decimals, exponents, arithmetic expressions, units and multiple values are not accepted. The integer value must match exactly; tolerance is zero

Use the same response format in either language. The Hindi section is the counterpart of the same twelve items, not twelve additional questions. Answers and full explanations follow all questions so you can attempt the review first.

Conditions for the code

C11 is the course's teaching convention, not a claimed GATE-mandated version. Every complete program is independent, uses positive fixed-size arrays where shown and starts from its own initialized state. Live target and bound assumptions are stated or established by the displayed code. Every evaluated integer calculation fits within −32767 through 32767; pointer differences shown fit in `ptrdiff_t`. No `int` or pointer byte size is assumed. No dynamic allocation or variable-length array is used.

The complete `c`-fenced programs are defined for the displayed inputs. Text-fenced classification material and invalid expressions mentioned in options must not be compiled or executed. Do not invent numeric outputs for undefined behavior. Any output triple or sequence shown in an answer is followed by the program's newline.

Questions

01 MCQ Keep the saved integer separate

Choose the printed triple. The three values appear in the order `saved`, `a[1]`, `a[2]`.

```
#include <stdio.h>

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

A. `8 9 3`

B. `3 4 3`

C. `8 9 8`

D. `8 3 9`

View solution

02 NAT Start from the complete initialized state

Enter the one integer printed by this program.

```
#include <stdio.h>

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

View solution

03 NAT Total only qualifying selected values

Enter the printed `total`. Keep the selected prefix, qualifying count and value sum distinct.

```
#include <stdio.h>

int main(void)
{
    int a[6] = {4, 9, 2, 7, 5, 8};
    int length = 5;
    int count = 0;
    int total = 0;
    for (int i = 0; i < length; i++) {
        if (a[i] >= 5) {
            count++;
            total += a[i];
        }
    }
    printf("%d\n", total);
    return 0;
}
```

View solution

04 MSQ Keep the first match

Select every true statement. Each proposed change in an option is considered separately from the original program.

```
#include <stdio.h>

int main(void)
{
    int values[5] = {6, 3, 6, 2, 6};
    int length = 4;
    int target = 6;
    int first = -1;
    for (int i = 0; i < length; i++) {
        if (first == -1 && values[i] == target) {
            first = i;
        }
    }
    printf("%d\n", first);
    return 0;
}
```

A. The displayed program prints 0 and still executes four loop bodies.

B. Once the first match is recorded, the guard ends the whole loop immediately.

C. Changing only `target` to 9 leaves `first` at −1; that marker must not be used as an array index.

D. Removing only `first == -1 &&` makes the displayed target 6 leave `first` at 2.

View solution

05 MCQ Retarget one pointer

Choose the output after both pointers have been traced separately.

```
#include <stdio.h>

int main(void)
{
    int x = 4;
    int y = 9;
    int *p = &x;
    int *q = p;
    *q = 6;
    p = &y;
    *p = *q + 1;
    *q = *p + 2;
    printf("%d %d %d\n", x, y, *q);
    return 0;
}
```

A. `6 12 12`

B. `9 7 9`

C. `6 7 6`

D. `9 9 9`

View solution

06 MSQ Check the target and its lifetime

Select every true statement under this module's C11 convention. These are behavior and lifetime claims, not instructions to execute invalid accesses. "Automatic integer" here means an ordinary local integer whose lifetime ends when its block ends.

A. For any pointer whatsoever, testing `p != NULL` proves that reading `*p` is safe.

B. A pointer to a live initialized integer containing 0 can be dereferenced; it is different from a null pointer.

C. When an automatic integer's lifetime ends, every pointer still holding its address automatically becomes null.

D. An integer value copied into an outer live variable before the inner source object ends can remain usable after that inner block ends.

View solution

07 MCQ Keep the caller and helper records separate

Choose the output. The selected length is valid for the shown array.

```
#include <stdio.h>

void raise_prefix(int a[], int length)
{
    for (int i = 0; i < length; i++) {
        a[i] += 2;
    }
    length = 1;
}

int main(void)
{
    int data[4] = {1, 4, 6, 3};
    int selected = 2;
    raise_prefix(data, selected);
    printf("%d %d %d | %d\n",
           data[0], data[1], data[2], selected);
    return 0;
}
```

A. `1 4 6 | 2`

B. `3 6 6 | 1`

C. `3 6 6 | 2`

D. `3 6 8 | 2`

View solution

08 MSQ Follow stores before and after local retargeting

Select every true statement about this complete program after the call returns.

```
#include <stdio.h>

void change_targets(int *p, int *q)
{
    *p = *p + 1;
    p = q;
    *p = *p + 2;
}

int main(void)
{
    int x = 2;
    int y = 5;
    int *chosen = &x;
    change_targets(chosen, &y);
    printf("%d %d %d\n", x, y, *chosen);
    return 0;
}
```

A. Caller `chosen` still identifies `x`.

B. The first store in the helper changes caller `x` to 3.

C. The assignment `p = q` makes caller `chosen` identify `y`.

D. The final store in the helper changes caller `y` to 7.

View solution

09 NAT Measure the signed element distance

Enter the printed signed integer. Both pointers belong to the same live array, and the difference fits in `ptrdiff_t`. Do not assume any byte size for `int` or a pointer.

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

int main(void)
{
    int a[6] = {2, 5, 1, 8, 4, 7};
    int *p = a + 1;
    int *q = p;
    p = p + 3;
    ptrdiff_t gap = q - p;
    printf("%td\n", gap);
    return 0;
}
```

View solution

10 MCQ Classify the end boundary before reading

This is classification-only material. Do not compile or execute the fragment or the invalid expressions mentioned in the choices. With the declaration below, choose the single correct statement. Consider each proposed expression independently.

```
int a[3] = {7, 2, 5};
int *end = a + 3;
```

A. `end` points to an extra integer whose value is implicitly zero.

B. Reading `*end` must produce the same value as reading `a[2]`.

C. Forming `end + 1` is permitted as long as the result is never dereferenced.

D. Forming `end` is permitted, and `*(end - 1)` reads 5; reading `*end` has undefined behavior.

View solution

11 MCQ Sum and then update a selected column

Choose the output. The column index remains 1 while the row index changes.

```
#include <stdio.h>

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

A. `9 1 9`

B. `10 1 9`

C. `6 1 8`

D. `9 0 1`

View solution

12 MSQ Keep rows and elements distinct

Select every true statement about the declarations below. The array and pointer are live. The visit-order statement refers to nested loops with `r` starting at 0 and increasing by 1 while `r < 2`, and, inside each outer body, a fresh `c` starting at 0 and increasing by 1 while `c < 3`. The inner body visits `grid[r][c]`. Claims involving an invalid read are for classification only.

```
int grid[2][3] = {{4, 1, 6}, {2, 8, 3}};
int (*row)[3] = grid;
```

A. `row` is a pointer to a three-`int` row; `row + 1` identifies the second row, not merely the next integer.

B. The declaration makes `grid` an `int **`, so each row is stored as a pointer variable.

C. The stated nested loops visit `(0,0), (0,1), (0,2), (1,0), (1,1), (1,2)` in that order.

D. Reading `grid[1][3]` is valid because the complete array contains six integers.

View solution

Answers and full solutions

Answer key: 01 A; 02 9; 03 21; 04 A, C, D; 05 B; 06 B, D; 07 C; 08 A, B, D; 09 -3; 10 D; 11 A; 12 A, C

01 Keep the saved integer separate

Answer A. `saved` copies 8 before either element update. The first store computes `5 - 2 = 3` and changes only `a[2]`. The second computes `8 + 1 = 9` and changes only `a[1]`. Final array: `{5, 9, 3, 1}`; separate `saved = 8`. Output: `8 9 3`, followed by a newline. All accessed indices lie from 0 through 3.

A matches those three current values in the requested order. B lets `saved` follow the changed element and then uses that imagined 3 in the next calculation. C omits the store to `a[2]`. D interchanges the two printed element positions.

First check if wrong: draw `saved` outside the array record and update only the assignment's actual destination. Revisit M2-L01, “Apply the familiar assignment rule to one element” and worked example 1.

Back to question · Contents

02 Start from the complete initialized state

Answer 9. The positive fixed-size array has an initializer list, so its full starting state is `{3, 2, 0, 0, 0}`. The three stores give `{3, 2, 0, 0, 6}`, then `{3, 2, 6, 0, 6}`, then `{3, 9, 6, 0, 6}`. Thus `a[1]` is 9. The read of `a[3]` uses its implicit initial zero, not an uninitialized value. Index 4 is the last valid position of this five-element array.

An answer of 6 stops before the final addition of `a[0]`. An answer of 7 can result from incorrectly repeating the last initializer 2 into omitted positions. An answer of 2 ignores the later stores. None of the updates reruns the initializer.

First check if wrong: write all five initial entries before tracing. Revisit M2-L01, “Partial initialization supplies the remaining zeros” and worked example 2.

Back to question · Contents

03 Total only qualifying selected values

Answer 21. Only indices 0 through 4 are selected. The visited values are 4, 9, 2, 7 and 5; the value 8 at index 5 is outside the selected prefix. After each body, `(count, total)` is `(0, 0)`, `(1, 9)`, `(1, 9)`, `(2, 16)`, `(3, 21)`. The loop then fails its test with `i = 5` without accessing that position.

Before each test, `total` is the sum of qualifying values already visited, and `count` is how many qualified. The array is unchanged. There are five bodies and six loop tests; three inner branches update the accumulators.

An answer of 3 gives the count, not the requested total. 29 incorrectly includes the unselected final value 8. 8 adds the qualifying indices 1, 3 and 4 instead of their values. 5 retains only the last qualifying value.

First check if wrong: draw a boundary after index 4, then record both accumulators at each checkpoint. Revisit M2-L02, worked example 1 and “Say what an accumulator means at a checkpoint”.

Back to question · Contents

04 Keep the first match

Answer A, C and D. In the original program, index 0 matches and stores 0. The remaining three bodies still execute. Their left operand `first == -1` is false, so short circuit skips the element comparison and no later assignment occurs. Output: `0`, followed by a newline. The loop has four bodies and five loop tests; it does not visit index 4.

A is true: retaining a result and stopping a loop are different actions. B is false: the guard controls the inner assignment, not the outer loop. C is true: none of the selected values equals 9, so −1 remains a not-found marker, never a valid position. D is true: without the first-match guard, indices 0 and 2 each overwrite `first`; the last selected match is 2. Index 4 also holds 6, but it is outside this chosen length.

First check if wrong: mark which condition skips which operation. Do not infer a loop exit from the variable name `first`. Revisit M2-L02, worked example 2 and practice 3–4.

Back to question · Contents

05 Retarget one pointer

Answer B. Initially both pointers identify `x`. The first store through `q` makes `x = 6`. Retargeting `p` gives `p → y`, while `q → x` remains. Next `*p = *q + 1` makes `y = 6 + 1 = 7`. Finally `*q = *p + 2` makes `x = 7 + 2 = 9`. The final targets are still different, so `*q` reads `x = 9`. Output: `9 7 9`, followed by a newline.

A can result from incorrectly making `q` follow the later retargeting of `p`: that mistaken trace keeps `x = 6` and makes both stores reach `y`. B is the valid trace. C omits the final store through `q`. D leaves `y` at its old 9 despite the store through `p`.

First check if wrong: after `p = &y`, redraw only the arrow for `p`. A copied pointer value does not make the other variable follow future assignments. Revisit M2-L03, worked example 2.

Back to question · Contents

06 Check the target and its lifetime

Answer B and D. A is false. A null comparison only rules out the known null case under suitable prior assumptions; it does not establish that an arbitrary pointer has a valid live target. An uninitialized pointer or one whose target has ended is not repaired by that comparison. We do not execute such a comparison to “check” it.

B is true. The target's integer value 0 and the pointer's nullness are different facts. A valid pointer to that initialized object supplies a readable integer zero.

C is false. A pointer value still referring to an object whose lifetime has ended becomes indeterminate under our C11 convention, not automatically null. Keeping the pointer variable alive does not prolong its target.

D is true. Copying a known integer into a separate longer-lived integer object preserves that value independently. This is different from retaining a pointer to the ended object. The outer copy must itself still be alive and initialized, as stated.

First check if wrong: draw separate lifetime intervals for the source object, any pointer variable and the copied integer. Revisit M2-L03, “Null is an explicit absence of a target” and “A target has to remain alive”.

Back to question · Contents

07 Keep the caller and helper records separate

Answer C. The array argument supplies a pointer to `data[0]`; helper parameter `a` is adjusted to `int *`. Helper `length` receives the copied integer 2. The two loop bodies change `data` to `{3, 4, 6, 3}` and then `{3, 6, 6, 3}`. The test fails at index 2, so no later element is changed.

The final helper assignment sets its own `length` to 1 after the loop. It does not change caller `selected = 2`. The void helper returns control without a result value, but the element stores have already reached caller storage. Output: `3 6 6 | 2`, followed by a newline.

A incorrectly invents a private array copy. B transfers a local parameter assignment back into caller `selected`. C keeps both records correct. D processes the third element even though the selected prefix has length 2. No hidden array length is supplied by the helper's brackets.

First check if wrong: draw the real array once and put helper parameters in separate boxes. Revisit M2-L04, worked example 1 and the helper access contract.

Back to question · Contents

08 Follow stores before and after local retargeting

Answer A, B and D. At entry, caller `chosen → x`, helper `p → x`, and helper `q → y` are three separate pointer records. The first body statement reads 2 through `p` and stores 3 into `x`. Then `p = q` retargets only the helper parameter `p` to `y`. The final store reads 5 from `y` and replaces it with 7. Output: `3 7 3`, followed by a newline.

A is true because caller `chosen` was never assigned a new pointer value. B is true because a copied pointer can still identify shared caller storage. C is false because assignment to the helper's pointer object does not assign to the caller's separate pointer object. D is true because the last dereference follows the helper's current target, `y`.

First check if wrong: mark each statement as a store to an integer or an assignment to a pointer variable. Do not use "passed by reference" as a substitute for the copied-value trace. Revisit M2-L04, worked example 2.

Back to question · Contents

09 Measure the signed element distance

Answer −3. Both pointers start at position 1. The assignment advances only `p` by three elements, taking it to position 4; `q` remains at position 1. Thus `q - p` is `1 - 4 = -3`. This counts signed element steps in the stated order, not values stored in the array and not bytes. No dereference occurs, and all formed positions are within this six-element array.

`ptrdiff_t` is the signed result type used here and `%td` prints it. An answer of 3 reverses the subtraction. Zero incorrectly moves `q` together with `p`. An answer based on 5 and 4 subtracts stored element values rather than positions. Multiplying by an assumed byte size answers a different question.

First check if wrong: label each pointer with `(array identity, position)` and subtract the position of the right operand from that of the left. Revisit M2-L05, "A distance between positions also has a type" and practice 2.

Back to question · Contents

10 Classify the end boundary before reading

Answer D. A three-element array has readable positions 0, 1 and 2. Pointer position 3 is the permitted one-past boundary. It is useful as a stopping position but supplies no fourth integer. Subtracting 1 from that end pointer returns to position 2, where the initialized value is 5.

A is false: partial-initializer zero filling never creates an extra element. B is false: an evaluated read through the end pointer is undefined, not a second spelling of the last element. C is false: `end + 1` attempts position 4, outside even the permitted pointer-formation range 0 through 3. Avoiding a later dereference does not make that formation valid. D states both boundaries correctly.

The unsafe expressions have no prescribed numeric output; neither a plausible observed value nor a compiler accepting a fragment would establish correctness. First check if wrong: mark a separate end boundary after the last element. Revisit M2-L05, “One past is a stopping position” and practice 1.

Back to question · Contents

11 Sum and then update a selected column

Answer A. The selected positions are `(0, 1)` and `(1, 1)`. At row 0 the program first adds 1, making `total = 1`, then adds row index 0 to that element, leaving it 1. At row 1 it first adds the current 8, making `total = 9`, then adds row index 1 to that element, making it 9. The final rows are `{4, 1, 6}` and `{2, 9, 3}`. Output: `9 1 9`, followed by a newline.

A follows the statement order and both indices. B totals the updated elements, as if the two body statements were reversed; the shown code sums before updating. C traces column 0 instead of the specified column 1, leaving the printed column-1 entries unchanged. D replaces each selected value with `r`, misreading `+= r` as `= r`. Other columns are unchanged.

First check if wrong: write every access as a pair `(row, column)` and record the total before the element update. Revisit M2-L06, the column scan worked example, and M2-L02’s current-state tracing.

Back to question · Contents

12 Keep rows and elements distinct

Answer A and C. `grid` is an array of two rows, each itself an array of three `int` elements. In the initializer for `row`, it supplies a pointer to its first row. `row` has type `int (*)[3]`, and its one-step movement selects the next whole row. A is true. It does not require a numerical byte size for `int`.

B is false. The real object contains row arrays, not an array of pointer variables; a pointer to a row is not `int **`. C is true: for each fixed `r`, the inner loop finishes all three columns before the outer update changes `r`. Its fresh `c = 0` starts the next row at column 0.

D is false. Row 1 exists, but it has only column indices 0, 1 and 2. Column index 3 supplies no element to read. The total of six integers does not remove the separate row boundary. Do not try to justify an invalid access by treating an `int *` into one row as a flat traversal of all rows.

First check if wrong: record the type at each level and pair each loop bound with its own dimension. Revisit M2-L06, its nested-loop explanation, row type and row-versus-element stepping sections.

Back to question · Contents

Use your result

The score is a location aid, not a readiness certificate. Pair items 1–2 with M2-L01, 3–4 with M2-L02, 5–6 with M2-L03, 7–8 with M2-L04, 9–10 with M2-L05 and 11–12 with M2-L06. If one answer is wrong, find the first state, target or bound that diverged from the solution, then redo that lesson's corresponding transfer exercise without looking at its answer. A correct guess is a reason to check the trace too.

This closes the introductory arrays-and-pointers module. It does not complete C, recursion, data structures or GATE CS. Recursion belongs in a later separate module.

Source note

The questions, code, distractors, bilingual explanations and traces are originally authored for this module; they are not reproductions of official examination questions. Language facts were verified against WG14 N1570, including clauses 6.2.4, 6.2.5, 6.3.2.1, 6.5.2.1, 6.5.2.2, 6.5.3.2, 6.5.6, 6.5.13, 6.7.6.3, 6.7.9 and 6.8.5.3. C11 remains a teaching convention.

Contents / विषय सूची

ऐरे और पॉइंटर पुनराभ्यास

तर्क की कमी पहचानने के लिए पुनराभ्यास करें

M2-L01 से M2-L06 पढ़ने के बाद ये बारह मौलिक प्रश्न हल करें। इनमें ऐरे की अवस्था, स्कैन, पॉइंटर के लक्ष्य, कॉल, सीमाएँ और द्वि-आयामी पंक्तियाँ दोबारा आती हैं। पुनराभ्यास में कोई नई परीक्षा-योग्य अवधारणा पहली बार नहीं आती। यह हर पाठ के मुख्य परिणामों को दो प्रश्नों से जाँचता है; हर छोटे उद्देश्य को नहीं जाँचता और शिक्षण अभ्यासों की जगह नहीं लेता।

यह बिना समय-सीमा का शैक्षिक पुनराभ्यास है। अधिकतम 12 अंक हैं: हर सही उत्तर पर +1 और गलत या छोड़े हुए उत्तर पर 0। न नकारात्मक अंक हैं, न आंशिक अंक और न उत्तीर्ण होने की सीमा। ये स्थानीय सीखने के नियम हैं, आधिकारिक GATE अंकन नियमों का दावा नहीं। प्रश्न मौलिक हैं, आधिकारिक पिछले वर्षों के प्रश्न नहीं हैं और यह पूरा GATE मॉक नहीं है।

- MCQ में ठीक एक विकल्प चुनें। ऐसे पाँच प्रश्न हैं
- MSQ में सभी सही विकल्पों का पूरा समूह चुनें। ऐसे चार प्रश्न हैं। कोई सही विकल्प छोड़ने या अतिरिक्त गलत विकल्प चुनने पर उस प्रश्न के 0 अंक हैं
- NAT में एक चिह्नित दशमलव पूर्णांक दें। ऐसे तीन प्रश्न हैं। साधारण अंक 0–9 और ऋणात्मक मान के लिए कीबोर्ड का ऋण चिह्न - इस्तेमाल करें; आसपास के खाली स्थान, शुरुआत में धन चिह्न और शुरुआती शून्य स्वीकार हैं। दशमलव-बिंदु वाले रूप, घातांक, गणितीय व्यंजक, इकाइयाँ या कई मान स्वीकार नहीं हैं। पूर्णांक का मान बिल्कुल सही होना चाहिए; सहनशीलता शून्य है

दोनों भाषाओं में उत्तर का प्रारूप एक जैसा रखें। हिंदी भाग इन्हीं बारह प्रश्नों का समकक्ष है, बारह अतिरिक्त प्रश्न नहीं। उत्तर और पूरे हल सभी प्रश्नों के बाद हैं, ताकि पहले आप स्वयं प्रयास कर सकें।

कोड की शर्तें

C11 पाठ्यक्रम की शिक्षण परंपरा है, GATE द्वारा अनिवार्य किया गया संस्करण होने का दावा नहीं। हर पूरा प्रोग्राम स्वतंत्र है, जहाँ ऐरे हैं वहाँ उनके निश्चित आकार धनात्मक हैं और हर प्रोग्राम अपनी इनिशियलाइज़्ड अवस्था से शुरू होता है। जीवित लक्ष्य और सीमा की धारणाएँ लिखी गई हैं या दिखाए कोड से स्थापित होती हैं। हर वास्तव में की गई पूर्णांक गणना −32767 से 32767 के बीच समाती है; दिखाए पॉइंटर अंतर `ptrdiff_t` में समाते हैं। `int` या पॉइंटर का कोई बाइट आकार नहीं माना गया। डायनेमिक आवंटन या परिवर्ती-लंबाई ऐरे नहीं है।

`c` चिह्न वाले पूरे प्रोग्राम दिखाए इनपुट के लिए परिभाषित हैं। `text` चिह्न वाली वर्गीकरण सामग्री और विकल्पों में बताए गलत व्यंजक कंपाइल या रन न करें। अपरिभाषित व्यवहार के लिए संख्यात्मक आउटपुट न गढ़ें। उत्तर में दिया गया कोई आउटपुट-क्रम प्रोग्राम की नई पंक्ति के बाद समाप्त होता है।

प्रश्न

01 MCQ सहेजे गए पूर्णांक को अलग रखें

छपने वाले तीन मान चुनें। उनका क्रम `saved`, `a[1]`, `a[2]` है।

```
#include <stdio.h>

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

A. `8 9 3`

B. `3 4 3`

C. `8 9 8`

D. `8 3 9`

हल देखें

02 NAT आरंभ की पूरी अवस्था से शुरू करें

इस प्रोग्राम से छपने वाला एक पूर्णांक लिखें।

```
#include <stdio.h>

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

हल देखें

03 NAT केवल चुने हुए योग्य मानों का योग करें

छपने वाला `total` लिखें। चुने हुए शुरुआती हिस्से, योग्य तत्वों की गिनती और उनके मानों के योग को अलग रखें।

```
#include <stdio.h>

int main(void)
{
    int a[6] = {4, 9, 2, 7, 5, 8};
    int length = 5;
    int count = 0;
    int total = 0;
    for (int i = 0; i < length; i++) {
        if (a[i] >= 5) {
            count++;
            total += a[i];
        }
    }
    printf("%d\n", total);
    return 0;
}
```

हल देखें

04 MSQ पहला मिलान सुरक्षित रखें

सभी सही कथन चुनें। किसी विकल्प में दिया गया बदलाव मूल प्रोग्राम से अलग-अलग मानें; बदलावों को एक साथ लागू न करें।

```
#include <stdio.h>

int main(void)
{
    int values[5] = {6, 3, 6, 2, 6};
    int length = 4;
    int target = 6;
    int first = -1;
    for (int i = 0; i < length; i++) {
        if (first == -1 && values[i] == target) {
            first = i;
        }
    }
    printf("%d\n", first);
    return 0;
}
```

A. दिखाया गया प्रोग्राम 0 छापता है और फिर भी लूप की बॉडी चार बार चलती है।

B. पहला मिलान दर्ज होते ही गार्ड पूरा लूप तुरंत समाप्त कर देता है।

C. केवल `target` को 9 करने पर `first` −1 रहता है; इस संकेतक को ऐरे इंडेक्स की तरह इस्तेमाल नहीं करना चाहिए।

D. केवल `first == -1 &&` हटाने पर दिखाए गए लक्ष्य 6 के लिए अंत में `first` का मान 2 होता है।

हल देखें

05 MCQ एक पॉइंटर का लक्ष्य बदलें

दोनों पॉइंटर का अलग-अलग ट्रेस करके आउटपुट चुनें।

```
#include <stdio.h>

int main(void)
{
    int x = 4;
    int y = 9;
    int *p = &x;
    int *q = p;
    *q = 6;
    p = &y;
    *p = *q + 1;
    *q = *p + 2;
    printf("%d %d %d\n", x, y, *q);
    return 0;
}
```

A. `6 12 12`

B. `9 7 9`

C. `6 7 6`

D. `9 9 9`

हल देखें

06 MSQ लक्ष्य और उसके जीवनकाल की जाँच करें

इस मॉड्यूल की C11 परंपरा के अनुसार सभी सही कथन चुनें। ये व्यवहार और जीवनकाल के बारे में कथन हैं; गलत एक्सेस चलाने के निर्देश नहीं हैं। यहाँ "ऑटोमैटिक पूर्णांक" का अर्थ साधारण लोकल पूर्णांक है जिसका जीवनकाल उसका ब्लॉक समाप्त होने पर खत्म होता है।

A. किसी भी पॉइंटर के लिए `p != NULL` जाँचना यह सिद्ध कर देता है कि `*p` पढ़ना सुरक्षित है।

B. जीवित और इनिशियलाइज़्ड पूर्णांक में 0 रखा हो तो उसकी ओर के पॉइंटर को डीरेफ़रेंस कर सकते हैं; वह नल पॉइंटर से अलग है।

C. ऑटोमैटिक पूर्णांक का जीवनकाल समाप्त होते ही उसका पता रखने वाले सभी पॉइंटर अपने आप नल हो जाते हैं।

D. अंदर का स्रोत ऑब्जेक्ट समाप्त होने से पहले उसका पूर्णांक मान बाहर के जीवित चर में कॉपी कर लें तो अंदर का ब्लॉक समाप्त होने के बाद भी उस कॉपी का इस्तेमाल किया जा सकता है।

हल देखें

07 MCQ कॉलर और सहायक के रिकॉर्ड अलग रखें

आउटपुट चुनें। चुनी गई लंबाई दिखाए गए ऐरे के लिए सही है।

```
#include <stdio.h>

void raise_prefix(int a[], int length)
{
    for (int i = 0; i < length; i++) {
        a[i] += 2;
    }
    length = 1;
}

int main(void)
{
    int data[4] = {1, 4, 6, 3};
    int selected = 2;
    raise_prefix(data, selected);
    printf("%d %d %d | %d\n",
           data[0], data[1], data[2], selected);
    return 0;
}
```

A. `1 4 6 | 2`

B. `3 6 6 | 1`

C. `3 6 6 | 2`

D. `3 6 8 | 2`

हल देखें

08 MSQ लोकल लक्ष्य बदलाव से पहले और बाद के असाइनमेंट देखें

कॉल लौटने के बाद इस पूरे प्रोग्राम के बारे में सभी सही कथन चुनें।

```
#include <stdio.h>

void change_targets(int *p, int *q)
{
    *p = *p + 1;
    p = q;
    *p = *p + 2;
}

int main(void)
{
    int x = 2;
    int y = 5;
    int *chosen = &x;
    change_targets(chosen, &y);
    printf("%d %d %d\n", x, y, *chosen);
    return 0;
}
```

A. कॉलर का `chosen` अभी भी `x` की ओर है।

B. सहायक का पहला असाइनमेंट कॉलर के `x` को 3 करता है।

C. `p = q` असाइनमेंट कॉलर के `chosen` को `y` की ओर कर देता है।

D. सहायक का अंतिम असाइनमेंट कॉलर के `y` को 7 करता है।

हल देखें

09 NAT तत्वों की चिह्नित दूरी निकालें

छपने वाला चिह्नित पूर्णांक लिखें। दोनों पॉइंटर एक ही जीवित ऐरे के हैं और अंतर `ptrdiff_t` में समा जाता है। `int` या पॉइंटर का कोई बाइट आकार न मानें।

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

int main(void)
{
    int a[6] = {2, 5, 1, 8, 4, 7};
    int *p = a + 1;
    int *q = p;
    p = p + 3;
    ptrdiff_t gap = q - p;
    printf("%td\n", gap);
    return 0;
}
```

हल देखें

10 MCQ पढ़ने से पहले अंतिम सीमा का वर्गीकरण करें

यह केवल वर्गीकरण की सामग्री है। इस अंश या विकल्पों में दिए गलत व्यंजकों को कंपाइल या रन न करें। नीचे दी घोषणा के लिए एक सही कथन चुनें। हर प्रस्तावित व्यंजक को अलग-अलग मानें।

```
int a[3] = {7, 2, 5};
int *end = a + 3;
```

A. `end` एक अतिरिक्त पूर्णांक की ओर है जिसका मान निहित रूप से शून्य है।

B. `*end` पढ़ने पर अनिवार्य रूप से वही मान मिलता है जो `a[2]` पढ़ने पर मिलता है।

C. `end + 1` बनाना अनुमत है, बशर्ते परिणाम को कभी डीरेफ़रेंस न करें।

D. `end` बनाना अनुमत है और `*(end - 1)` से 5 पढ़ा जाता है; `*end` पढ़ने का व्यवहार अपरिभाषित है।

हल देखें

11 MCQ चुने हुए कॉलम का योग करें फिर अपडेट करें

आउटपुट चुनें। पंक्ति इंडेक्स बदलता है, पर कॉलम इंडेक्स 1 रहता है।

```
#include <stdio.h>

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

A. `9 1 9`

B. `10 1 9`

C. `6 1 8`

D. `9 0 1`

हल देखें

12 MSQ पंक्तियों और तत्वों को अलग रखें

नीचे की घोषणाओं के बारे में सभी सही कथन चुनें। ऐरे और पॉइंटर जीवित हैं। भ्रमण-क्रम वाला कथन ऐसे नेस्टेड लूप के लिए है जिसमें `r` 0 से शुरू होकर `r < 2` तक हर बार 1 बढ़ता है; प्रत्येक बाहरी बॉडी के अंदर नया `c` 0 से शुरू होकर `c < 3` तक हर बार 1 बढ़ता है। अंदर की बॉडी `grid[r][c]` देखती है। गलत रीड वाले कथन केवल वर्गीकरण के लिए हैं।

```
int grid[2][3] = {{4, 1, 6}, {2, 8, 3}};
int (*row)[3] = grid;
```

A. `row` तीन `int` वाली पंक्ति का पॉइंटर है; `row + 1` दूसरी पंक्ति की ओर है, केवल अगले पूर्णांक की ओर नहीं।

B. घोषणा से `grid` एक `int **` बनता है, इसलिए हर पंक्ति एक पॉइंटर चर के रूप में रखी जाती है।

C. बताए गए नेस्टेड लूप क्रम से `(0,0), (0,1), (0,2), (1,0), (1,1), (1,2)` देखते हैं।

D. `grid[1][3]` पढ़ना सही है क्योंकि पूरे ऐरे में छह पूर्णांक हैं।

हल देखें

उत्तर और पूरे हल

उत्तर कुंजी: 01 A; 02 9; 03 21; 04 A, C, D; 05 B; 06 B, D; 07 C; 08 A, B, D; 09 -3; 10 D; 11 A; 12 A, C

01 सहेजे गए पूर्णांक को अलग रखें

उत्तर A। दोनों तत्वों के अपडेट से पहले `saved` में 8 की कॉपी बनती है। पहला असाइनमेंट `5 - 2 = 3` निकालकर केवल `a[2]` बदलता है। दूसरा `8 + 1 = 9` निकालकर केवल `a[1]` बदलता है। अंतिम ऐरे `{5, 9, 3, 1}` है और अलग चर `saved = 8` है। आउटपुट `8 9 3` है, जिसके बाद नई पंक्ति आती है। इस्तेमाल किए गए सभी इंडेक्स 0 से 3 के बीच हैं।

A माँगे गए क्रम में इन्हीं वर्तमान मानों को देता है। B में `saved` को बदले हुए तत्व के साथ बदलता मान लिया गया है और फिर कल्पित 3 से अगली गणना की गई है। C में `a[2]` वाला असाइनमेंट छोड़ दिया गया है। D में छपने वाले दोनों तत्वों के स्थान आपस में बदल दिए गए हैं।

गलती होने पर पहले जाँचें: ऐरे के रिकॉर्ड से बाहर `saved` का अलग रिकॉर्ड बनाएँ और केवल असाइनमेंट का वास्तविक गंतव्य बदलें। M2-L01 में एक तत्व पर असाइनमेंट का नियम और हल किया गया उदाहरण 1 दोबारा देखें।

प्रश्न पर लौटें · विषय सूची

02 आरंभ की पूरी अवस्था से शुरू करें

उत्तर 9। धनात्मक निश्चित आकार वाले इस ऐरे में इनिशियलाइज़र सूची है, इसलिए उसकी पूरी शुरुआती अवस्था `{3, 2, 0, 0, 0}` है। तीन असाइनमेंट क्रम से `{3, 2, 0, 0, 6}`, फिर `{3, 2, 6, 0, 6}` और अंत में `{3, 9, 6, 0, 6}` बनाते हैं। इसलिए `a[1]` का मान 9 है। `a[3]` को पढ़ते समय उसका निहित शुरुआती शून्य मिलता है; वह अनइनिशियलाइज़्ड मान नहीं है। पाँच तत्वों वाले इस ऐरे में इंडेक्स 4 अंतिम सही स्थान है।

6 लिखने पर आप शायद अंत में `a[0]` जोड़ने से पहले रुक गए हैं। 7 तब मिल सकता है जब अंतिम लिखे इनिशियलाइज़र 2 को छूटे हुए स्थानों में भी दोहरा दिया जाए। 2 लिखना बाद के असाइनमेंट छोड़ देता है। किसी अपडेट से इनिशियलाइज़र दोबारा नहीं चलता।

गलती होने पर पहले जाँचें: ट्रेस से पहले सभी पाँच शुरुआती मान लिखें। M2-L01 में आंशिक इनिशियलाइज़ेशन से बाकी शून्य मिलने वाला भाग और हल किया गया उदाहरण 2 दोबारा देखें।

प्रश्न पर लौटें · विषय सूची

03 केवल चुने हुए योग्य मानों का योग करें

उत्तर 21। केवल इंडेक्स 0 से 4 तक चुने गए हैं। देखे गए मान 4, 9, 2, 7 और 5 हैं; इंडेक्स 5 का मान 8 चुने हुए शुरुआती हिस्से से बाहर है। हर बॉडी के बाद `(count, total)` क्रम से `(0, 0)`, `(1, 9)`, `(1, 9)`, `(2, 16)`, `(3, 21)` है। फिर `i = 5` पर लूप की शर्त असफल होती है और उस स्थान का एक्सेस नहीं होता।

हर शर्त की जाँच से पहले `total` में पहले देखे गए योग्य मानों का योग है और `count` में उनकी गिनती है। ऐरे नहीं बदलता। बॉडी पाँच बार और लूप की शर्त छह बार चलती है; तीन चुनी हुई अंदर की शाखाएँ संचायक अपडेट करती हैं।

3 उत्तर देने पर योग की जगह गिनती दी गई है। 29 में अचयनित अंतिम मान 8 भी जोड़ दिया गया है। 8 में योग्य तत्वों के मानों के बजाय इंडेक्स 1, 3 और 4 जोड़े गए हैं। 5 में केवल अंतिम योग्य मान बचा है।

गलती होने पर पहले जाँचें: इंडेक्स 4 के बाद सीमा बनाएँ और हर जाँच-बिंदु पर दोनों संचायक लिखें। M2-L02 का हल किया गया उदाहरण 1 और जाँच-बिंदु पर संचायक का अर्थ वाला भाग दोबारा देखें।

प्रश्न पर लौटें · विषय सूची

04 पहला मिलान सुरक्षित रखें

उत्तर A, C और D। मूल प्रोग्राम में इंडेक्स 0 पर मिलान होता है और 0 दर्ज होता है। बाकी तीन बॉडी फिर भी चलती हैं। उनमें बायाँ ऑपरैंड `first == -1` असत्य है, इसलिए शॉर्ट सर्किट से तत्व की तुलना छोड़ दी जाती है और बाद में कोई असाइनमेंट नहीं होता। आउटपुट `0` है, फिर नई पंक्ति आती है। बॉडी चार बार और लूप की शर्त पाँच बार चलती है; इंडेक्स 4 नहीं देखा जाता।

A सही है: परिणाम सुरक्षित रखना और लूप रोकना अलग क्रियाएँ हैं। B गलत है: गार्ड अंदर के असाइनमेंट को नियंत्रित करता है, बाहर के लूप को नहीं। C सही है: चुने हुए किसी मान का 9 से मिलान नहीं होता, इसलिए −1 केवल न मिलने का संकेतक रहता है; वह सही स्थान नहीं है। D सही है: पहले मिलान का गार्ड हटाने पर इंडेक्स 0 और 2 दोनों `first` को बदलते हैं; अंतिम चुना हुआ मिलान 2 है। इंडेक्स 4 में भी 6 है, पर वह चुनी गई लंबाई से बाहर है।

गलती होने पर पहले जाँचें: कौन सी शर्त कौन सी क्रिया छोड़ती है, यह चिह्नित करें। चर का नाम `first` देखकर लूप से बाहर निकलना न मानें। M2-L02 का हल किया गया उदाहरण 2 और अभ्यास 3–4 दोबारा देखें।

प्रश्न पर लौटें · विषय सूची

05 एक पॉइंटर का लक्ष्य बदलें

उत्तर B। शुरुआत में दोनों पॉइंटर `x` की ओर हैं। `q` से पहला असाइनमेंट `x = 6` करता है। `p` का लक्ष्य बदलने पर `p → y` हो जाता है, जबकि `q → x` रहता है। फिर `*p = *q + 1` से `y = 6 + 1 = 7` होता है। अंत में `*q = *p + 2` से `x = 7 + 2 = 9` होता है। अंतिम लक्ष्य अभी भी अलग हैं, इसलिए `*q` से `x = 9` पढ़ा जाता है। आउटपुट `9 7 9` है, फिर नई पंक्ति आती है।

A तब मिल सकता है जब गलत ढंग से मान लें कि `p` का बाद वाला लक्ष्य बदलाव `q` पर भी लागू होता है; उस गलत ट्रेस में `x = 6` रहता है और दोनों असाइनमेंट `y` तक पहुँचते हैं। B सही ट्रेस है। C में `q` से अंतिम असाइनमेंट छोड़ दिया गया है। D में `p` से असाइनमेंट होने के बावजूद `y` पुराना 9 रखा गया है।

गलती होने पर पहले जाँचें: `p = &y` के बाद केवल `p` का तीर बदलें। कॉपी किया गया पॉइंटर मान दूसरे चर को भविष्य के असाइनमेंट के साथ नहीं चलाता। M2-L03 का हल किया गया उदाहरण 2 दोबारा देखें।

प्रश्न पर लौटें · विषय सूची

06 लक्ष्य और उसके जीवनकाल की जाँच करें

उत्तर B और D। A गलत है। उचित पहले से ज्ञात धारणाओं में नल की तुलना केवल ज्ञात नल स्थिति को अलग करती है; वह किसी भी मनमाने पॉइंटर के लिए जीवित सही लक्ष्य सिद्ध नहीं करती। इस तुलना से अनइनिशियलाइज़्ड पॉइंटर या समाप्त लक्ष्य वाला पॉइंटर ठीक नहीं हो जाता। उसे “जाँचने” के लिए हम ऐसी तुलना चलाते नहीं हैं।

B सही है। लक्ष्य पूर्णांक का मान 0 होना और पॉइंटर का नल होना अलग बातें हैं। उस इनिशियलाइज़्ड ऑब्जेक्ट का सही पॉइंटर पढ़ने योग्य पूर्णांक शून्य देता है।

C गलत है। हमारी C11 परंपरा में समाप्त जीवनकाल वाले ऑब्जेक्ट को अभी भी संदर्भित करने वाला पॉइंटर मान अनिर्धारित यानी indeterminate हो जाता है; वह अपने आप नल नहीं होता। पॉइंटर चर को जीवित रखने से लक्ष्य का जीवनकाल नहीं बढ़ता।

D सही है। ज्ञात पूर्णांक को अलग और अधिक देर तक जीवित रहने वाले पूर्णांक ऑब्जेक्ट में कॉपी करना उसका मान स्वतंत्र रूप से सुरक्षित रखता है। यह समाप्त ऑब्जेक्ट की ओर पॉइंटर रखने से अलग है। जैसा कथन में कहा गया है, बाहर की कॉपी स्वयं भी जीवित और इनिशियलाइज़्ड होनी चाहिए।

गलती होने पर पहले जाँचें: स्रोत ऑब्जेक्ट, पॉइंटर चर और कॉपी किए गए पूर्णांक के जीवनकाल अलग-अलग बनाएँ। M2-L03 में नल से लक्ष्य के अभाव और लक्ष्य के जीवित रहने वाले भाग दोबारा देखें।

प्रश्न पर लौटें · विषय सूची

07 कॉलर और सहायक के रिकॉर्ड अलग रखें

उत्तर C। ऐरे आर्ग्युमेंट से `data[0]` का पॉइंटर मिलता है; सहायक पैरामीटर `a` का प्रकार समायोजित होकर `int *` होता है। सहायक के `length` में पूर्णांक 2 की कॉपी आती है। लूप की दो बॉडी `data` को पहले `{3, 4, 6, 3}` और फिर `{3, 6, 6, 3}` करती हैं। इंडेक्स 2 पर शर्त असफल होती है, इसलिए बाद का कोई तत्व नहीं बदलता।

लूप के बाद अंतिम असाइनमेंट सहायक के अपने `length` को 1 करता है। इससे कॉलर का `selected = 2` नहीं बदलता। void सहायक कोई परिणाम मान दिए बिना नियंत्रण लौटाता है, लेकिन तत्वों के असाइनमेंट पहले ही कॉलर के स्टोरेज तक पहुँच चुके हैं। आउटपुट `3 6 6 | 2` है, फिर नई पंक्ति आती है।

A में गलत ढंग से ऐरे की निजी कॉपी बना दी गई है। B में लोकल पैरामीटर का बदलाव कॉलर के `selected` में भी पहुँचाया गया है। C दोनों रिकॉर्ड सही रखता है। D तीसरा तत्व भी बदल देता है, जबकि चुने हुए शुरुआती हिस्से की लंबाई 2 है। सहायक के ब्रैकेट किसी छिपी हुई ऐरे लंबाई को नहीं देते।

गलती होने पर पहले जाँचें: वास्तविक ऐरे केवल एक बार बनाएँ और सहायक पैरामीटर अलग रिकॉर्ड में रखें। M2-L04 का हल किया गया उदाहरण 1 और सहायक के एक्सेस की शर्तें दोबारा देखें।

प्रश्न पर लौटें · विषय सूची

08 लोकल लक्ष्य बदलाव से पहले और बाद के असाइनमेंट देखें

उत्तर A, B और D। प्रवेश पर कॉलर का `chosen → x`, सहायक का `p → x` और सहायक का `q → y` तीन अलग पॉइंटर रिकॉर्ड हैं। बॉडी का पहला कथन `p` से 2 पढ़कर `x` में 3 रखता है। फिर `p = q` केवल सहायक पैरामीटर `p` का लक्ष्य `y` करता है। अंतिम असाइनमेंट `y` से 5 पढ़कर उसकी जगह 7 रखता है। आउटपुट `3 7 3` है, फिर नई पंक्ति आती है।

A सही है क्योंकि कॉलर के `chosen` में नया पॉइंटर मान असाइन नहीं किया गया। B सही है क्योंकि कॉपी किया गया पॉइंटर अभी भी कॉलर के साझा स्टोरेज की ओर हो सकता है। C गलत है क्योंकि सहायक के पॉइंटर ऑब्जेक्ट में असाइनमेंट कॉलर के अलग पॉइंटर ऑब्जेक्ट में असाइनमेंट नहीं है। D सही है क्योंकि अंतिम डीरेफ़रेंस सहायक के वर्तमान लक्ष्य `y` तक जाता है।

गलती होने पर पहले जाँचें: हर कथन को पूर्णांक में असाइनमेंट या पॉइंटर चर में असाइनमेंट के रूप में चिह्नित करें। कॉपी किए मानों का ट्रेस करने के बजाय “पास बाय रेफ़रेंस” कहकर बात पूरी न मानें। M2-L04 का हल किया गया उदाहरण 2 दोबारा देखें।

प्रश्न पर लौटें · विषय सूची

09 तत्वों की चिह्नित दूरी निकालें

उत्तर −3। दोनों पॉइंटर शुरुआत में स्थान 1 पर हैं। असाइनमेंट केवल `p` को तीन तत्व आगे ले जाकर स्थान 4 पर रखता है; `q` स्थान 1 पर रहता है। इसलिए `q - p` से `1 - 4 = -3` मिलता है। यह दिए गए क्रम में तत्वों के चिह्नित कदम गिनता है, ऐरे में रखे मान या बाइट नहीं। कोई डीरेफ़रेंस नहीं होता और बने हुए सभी स्थान इसी छह-तत्व वाले ऐरे में हैं।

यहाँ `ptrdiff_t` चिह्नित परिणाम का प्रकार है और `%td` उसे छापता है। 3 उत्तर देने पर घटाव का क्रम उलट गया है। शून्य में `q` को भी `p` के साथ चला दिया गया है। 5 और 4 से गणना करने पर स्थानों की जगह रखे हुए तत्व मान घटाए जा रहे हैं। माने हुए बाइट आकार से गुणा करना किसी दूसरे प्रश्न का उत्तर होगा।

गलती होने पर पहले जाँचें: हर पॉइंटर पर (ऐरे की पहचान, स्थान) लिखें और बाएँ ऑपरैंड के स्थान से दाएँ ऑपरैंड का स्थान घटाएँ। M2-L05 में दो स्थानों के अंतर के प्रकार वाला भाग और अभ्यास 2 दोबारा देखें।

प्रश्न पर लौटें · विषय सूची

10 पढ़ने से पहले अंतिम सीमा का वर्गीकरण करें

उत्तर D। तीन तत्वों वाले ऐरे में पढ़ने योग्य स्थान 0, 1 और 2 हैं। पॉइंटर का स्थान 3 अंतिम तत्व के ठीक बाद की अनुमत सीमा है। रुकने के स्थान के रूप में वह उपयोगी है, लेकिन चौथा पूर्णांक नहीं देता। उस अंतिम सीमा पॉइंटर से 1 घटाने पर स्थान 2 मिलता है, जहाँ इनिशियलाइज़्ड मान 5 है।

A गलत है: आंशिक इनिशियलाइज़र से शून्य भरना कभी अतिरिक्त तत्व नहीं बनाता। B गलत है: अंतिम सीमा पॉइंटर से वास्तव में पढ़ना अपरिभाषित है, अंतिम तत्व को पढ़ने का दूसरा तरीका नहीं। C गलत है: `end + 1` स्थान 4 बनाने का प्रयास है, जो पॉइंटर बनाने की अनुमत सीमा 0 से 3 के भी बाहर है। बाद में डीरेफ़रेंस न करने से उसका बनना सही नहीं हो जाता। D दोनों सीमाएँ सही बताता है।

गलत व्यंजकों का कोई निर्धारित संख्यात्मक आउटपुट नहीं है; न कोई संभव दिखने वाला मान और न कंपाइलर का अंश स्वीकार करना सही व्यवहार सिद्ध करता है। गलती होने पर पहले जाँचें: अंतिम तत्व के बाद अलग सीमा बनाएँ। M2-L05 में अंतिम तत्व के बाद रुकने वाली स्थिति और अभ्यास 1 दोबारा देखें।

प्रश्न पर लौटें · विषय सूची

11 चुने हुए कॉलम का योग करें फिर अपडेट करें

उत्तर A। चुने गए स्थान `(0, 1)` और `(1, 1)` हैं। पंक्ति 0 पर प्रोग्राम पहले 1 जोड़ता है, इसलिए `total = 1` होता है; फिर उसी तत्व में पंक्ति इंडेक्स 0 जोड़ता है, इसलिए वह 1 रहता है। पंक्ति 1 पर पहले वर्तमान 8 जोड़ने से `total = 9` होता है; फिर उस तत्व में पंक्ति इंडेक्स 1 जोड़ने से वह 9 हो जाता है। अंतिम पंक्तियाँ `{4, 1, 6}` और `{2, 9, 3}` हैं। आउटपुट `9 1 9` है, फिर नई पंक्ति आती है।

A कथनों के क्रम और दोनों इंडेक्स का पालन करता है। B अपडेट किए तत्वों का योग करता है, मानो बॉडी के दोनों कथन उलट दिए गए हों; दिखाया कोड अपडेट से पहले जोड़ता है। C दिए गए कॉलम 1 के बजाय कॉलम 0 का ट्रेस करता है, जिससे छपने वाले कॉलम-1 के मान नहीं बदलते। D `+= r` को `= r` पढ़कर हर चुने मान की जगह `r` रख देता है। दूसरे कॉलम नहीं बदलते।

गलती होने पर पहले जाँचें: हर एक्सेस को (पंक्ति, कॉलम) जोड़ी में लिखें और तत्व के अपडेट से पहले योग दर्ज करें। M2-L06 का कॉलम स्कैन वाला हल किया गया उदाहरण और M2-L02 का वर्तमान अवस्था का ट्रेस दोबारा देखें।

प्रश्न पर लौटें · विषय सूची

12 पंक्तियों और तत्वों को अलग रखें

उत्तर A और C। `grid` दो पंक्तियों का ऐरे है और प्रत्येक पंक्ति स्वयं तीन `int` तत्वों का ऐरे है। `row` के इनिशियलाइज़र में उससे पहली पंक्ति का पॉइंटर मिलता है। `row` का प्रकार `int (*)[3]` है और उसका एक कदम अगली पूरी पंक्ति चुनता है। A सही है। इसके लिए `int` का कोई संख्यात्मक बाइट आकार जानना ज़रूरी नहीं।

B गलत है। वास्तविक ऑब्जेक्ट में पंक्ति ऐरे हैं, पॉइंटर चरों का ऐरे नहीं; पंक्ति का पॉइंटर `int **` नहीं है। C सही है: हर निश्चित `r` के लिए अंदर का लूप तीनों कॉलम पूरा करता है, फिर बाहरी अपडेट `r` बदलता है। नया `c = 0` अगली पंक्ति को कॉलम 0 से शुरू करता है।

D गलत है। पंक्ति 1 मौजूद है, पर उसमें केवल कॉलम इंडेक्स 0, 1 और 2 हैं। कॉलम इंडेक्स 3 पढ़ने के लिए कोई तत्व नहीं देता। कुल छह पूर्णांक होने से अलग पंक्ति की सीमा नहीं मिटती। किसी एक पंक्ति में मौजूद `int *` को सभी पंक्तियों का सपाट भ्रमण मानकर गलत एक्सेस को सही न ठहराएँ।

गलती होने पर पहले जाँचें: हर स्तर का प्रकार लिखें और हर लूप सीमा को उसके अपने आयाम से जोड़ें। M2-L06 में नेस्टेड लूप की व्याख्या, पंक्ति के प्रकार और पंक्ति तथा तत्व के अलग कदम वाले भाग दोबारा देखें।

प्रश्न पर लौटें · विषय सूची

अपने परिणाम का इस्तेमाल करें

अंक कमी का स्थान पहचानने में मदद करते हैं, तैयारी का प्रमाणपत्र नहीं हैं। प्रश्न 1–2 को M2-L01, 3–4 को M2-L02, 5–6 को M2-L03, 7–8 को M2-L04, 9–10 को M2-L05 और 11–12 को M2-L06 से जोड़ें। कोई उत्तर गलत हो तो पहला ऐसा अवस्था-रिकॉर्ड, लक्ष्य या सीमा ढूँढें जो हल से अलग हुआ; फिर उस पाठ का संबंधित अभ्यास उसका उत्तर देखे बिना दोबारा करें। सही अनुमान लग गया हो तब भी ट्रेस जाँचें।

यह शुरुआती ऐरे और पॉइंटर मॉड्यूल का समापन है। इससे C, रिकर्शन, डेटा स्ट्रक्चर या GATE CS पूरा नहीं होता। रिकर्शन बाद के अलग मॉड्यूल का विषय है।

स्रोत संबंधी टिप्पणी

प्रश्न, कोड, गलत विकल्प, द्विभाषी व्याख्याएँ और ट्रेस इस मॉड्यूल के लिए मौलिक रूप से लिखे गए हैं; वे आधिकारिक परीक्षा प्रश्नों की प्रतियाँ नहीं हैं। भाषा के तथ्य WG14 N1570 से सत्यापित किए गए, जिनमें खंड 6.2.4, 6.2.5, 6.3.2.1, 6.5.2.1, 6.5.2.2, 6.5.3.2, 6.5.6, 6.5.13, 6.7.6.3, 6.7.9 और 6.8.5.3 शामिल हैं। C11 शिक्षण परंपरा ही है।