

C foundations teaching workbook

C बुनियाद अध्ययन पुस्तिका

Five lessons on tracing C programs with integers

पूर्णार्कों वाले C प्रोग्राम का क्रमवार विश्लेषण सीखने के पाँच पाठ

Start with current values. Follow each update, branch, loop and function call. Attempt the practice before checking the fully explained solutions.

वर्तमान मानों से शुरू करें। हर अपडेट, शाखा, लूप और फ़ंक्शन कॉल को क्रम से समझें। पूरे हल देखने से पहले अभ्यास स्वयं करें।

Includes 10 worked examples, 20 practice questions, analogies and quick references in English and Hindi.

अंग्रेज़ी और हिंदी में 10 हल किए हुए उदाहरण, 20 अभ्यास प्रश्न, उपमाएँ और त्वरित संदर्भ।

Original bilingual study material. Explicit C11 teaching convention. This is a bounded foundation module, not full C or full GATE CS coverage. Hindi supports learning; GATE examination content is in English.

मौलिक द्विभाषी अध्ययन सामग्री। शिक्षण के लिए C11 स्पष्ट रूप से चुना गया है। यह सीमित बुनियादी मॉड्यूल है, पूरा C या पूरा GATE CS पाठ्यक्रम नहीं। हिंदी सीखने में सहायता करती है; GATE परीक्षा की सामग्री अंग्रेज़ी में होती है।

Contents

विषय सूची

Choose a language for each lesson. Each section includes its own explanations, practice and solutions.

हर पाठ के लिए भाषा चुनें। हर भाग में उसकी व्याख्या, अभ्यास और हल हैं।

1 Variables, assignments and a trace · English	3
1 चर, असाइनमेंट और क्रमवार ट्रेस · हिंदी.....	10
2 Expression values and safe updates · English	17
2 व्यंजक का मान और सुरक्षित अपडेट · हिंदी	25
3 Conditions and selected branches · English	33
3 शर्तें और चुनी गई शाखाएँ · हिंदी.....	40
4 Loops, updates and stopping · English	47
4 लूप, अपडेट और रुकने की शर्त · हिंदी	55
5 Functions, copied arguments and local state · English	63
5 फ़ंक्शन, आर्गुमेंट की कॉपी और स्थानीय अवस्था · हिंदी	70

Internal links and PDF bookmarks support navigation. Code blocks are independent examples. Never run the two classification-only unsafe originals in the teaching workbook.

अंदरूनी लिंक और PDF बुकमार्क से आगे जाएँ। हर कोड ब्लॉक स्वतंत्र उदाहरण है। अध्ययन पुस्तिका में केवल वर्गीकरण के लिए दिए गए दो असुरक्षित मूल प्रोग्राम कभी न चलाएँ।

Variables, assignments and a trace

What will this program print?

C code 1

```
#include <stdio.h>

int main(void)
{
    int x = 6;
    int y = 4;
    x = x + 3;
    y = x - y;
    printf("%d %d\n", x, y);
    return 0;
}
```

The two numbers are not obvious until we follow the changes. By the end of this lesson, you will be able to name the value held by each variable after every statement and explain the printed result.

Read the small amount of syntax we need

A variable is a named place for a value. Here, `x` and `y` each hold an `int`: an integer such as 6, 0 or -3. `int x = 6;` declares `x` and initializes it to 6. Initialization supplies its first value. Later, `x = x + 3;` changes the value of the existing variable; it does not declare another `x`.

The semicolon ends each declaration or expression statement shown here. The braces surround the body of `main`. `int main(void)` is the entry function for these complete programs; `void` here says it takes no parameters. `#include <stdio.h>` supplies the declaration needed for `printf`. The include line does not end in a semicolon. `return 0;` ends this successful execution. You can trace the statements inside the braces before learning functions in detail. A declaration may also introduce several variables of the same type: `int a=8, b=5;` gives `a` the value 8 and `b` the value 5. The comma separates the variable names and their initializers in that declaration.

`printf("%d %d\n", x, y);` prints two integer values. The first `%d` corresponds to `x`, the second to `y`. The space inside the quotes separates the numbers, and `\n` ends the output line. The quotes, `%d`, and `\n` themselves are not printed. In this call both arguments merely read existing values; neither changes a variable. The order in which a C implementation evaluates these arguments therefore cannot change this result.

Only ordinary whole-number arithmetic is needed: for example, $17 - 9$ is 8, and $6 * 4$ is 24. No earlier programming knowledge is assumed. In the code here, `*` means multiplication; the letter `x` is a variable name, not a multiplication sign.

The module's code contract

We use ISO C11 as our explicit teaching convention. Every deterministic example starts its variables with known values, uses ordinary signed `int`, and keeps every intermediate result between -32767 and 32767. All evaluated divisors are nonzero. These choices avoid relying on a particular machine's integer size or representation. We do not assume that `int` is 32 bits. Inputs are the values written in the declarations; no keyboard input is required.

State: the values that exist now

The state is the collection of current variable values at a particular point. A trace is a written record of those states. Make a new record after each statement. If a statement does not assign to a variable, carry that variable's value forward unchanged.

For an assignment of the simple form `x = x + 3;`, use this method:

1. Read the current values on the right: here the old `x`.
2. Calculate the right-hand expression: old `x` plus 3.
3. Store that result in the variable on the left: `x`.
4. Use the new state for the next statement.

The two appearances of `x` have different jobs: the one on the right supplies a value, while the one on the left names the destination. `=` is an instruction to store a value. It does not assert a mathematical equation that must remain true forever. With `x` currently 6, `x = x + 3;` is a valid update to 9; it is not the impossible equation "6 equals 9". This reading rule applies to the simple assignments we use, not to every possible expression with side effects.

Worked example 1: use the current state

Return to the opening program. Keep both variable values visible, including the one that did not change.

1. After `int x = 6;` `x=6`; `y` has not been declared yet.
2. After `int y = 4;` `x=6`, `y=4`.
3. For `x = x + 3;`, calculate `6 + 3 = 9`. After the statement: `x=9`, `y=4`.
4. For `y = x - y;`, calculate `9 - 4 = 5`. After the statement: `x=9`, `y=5`.
5. `printf` reads `x=9` and `y=5`. It prints `9 5` followed by a newline. The variables remain `x=9`, `y=5`.

The second assignment sees the updated `x=9`. Using the starting `x=6` again would incorrectly produce `y=2`. Each statement begins with the state left by the preceding statement; the assignments are not simultaneous.

A copy is not a permanent connection

A declaration can take its starting value from an existing variable. In the following body, `copy` receives the value that `source` has at that moment. It does not become a live formula for `source`.

All shorter code blocks in this lesson belong inside the same main body, with `#include <stdio.h>` above it and `return 0;` at the end. Each block is a separate example, starting afresh; do not concatenate the blocks.

C code 2

```
int source = 6;
int copy = source;
source = 10;
printf("%d %d\n", source, copy);
```

After the declarations: source=6, copy=6. After source = 10;: source=10, copy=6. The output is 10 6. Assigning a new value to source does not also write to copy. A trace records stored numbers, not continuing algebraic relationships.

Arithmetic expressions and grouping

An expression describes a value to calculate. For the integer expressions in this lesson:

- + adds, - subtracts, and * multiplies
- / gives an integer quotient; % gives the corresponding remainder
- Parentheses explicitly group a subexpression
- *, /, and % bind more tightly than + and -

Binding means which operands belong to each operator. It is not a general promise about the timing of operand evaluation. Our arithmetic examples read values without updating them inside the expression, so their grouping determines a unique result.

C code 3

```
int plain = 2 + 3 * 4;
int grouped = (2 + 3) * 4;
printf("%d %d\n", plain, grouped);
```

For plain, group the expression as $2 + (3 * 4)$: $3 * 4$ is 12, then $2 + 12$ is 14. For grouped, the parentheses require $(2 + 3)$ to supply 5, then $5 * 4$ is 20. The output is 14 20. C does not calculate every arithmetic expression by simply moving from the leftmost character to the right.

Whole groups and leftovers

If 23 items go into groups of 5, four full groups use 20 items and 3 remain. Thus $23 / 5$ is 4 and $23 \% 5$ is 3. The quotient is an integer, not 4.6 rounded to the nearest integer. With a nonnegative total and a positive group size, the remainder is at least zero and smaller than the group size. A zero remainder means the division is exact.

The symbols have different roles: % between two integer expressions is remainder; %d inside a printf format is a request to print an integer. Neither means “percent” in these examples. Never divide by zero or take remainder with zero as the divisor. Negative operands are taught in the next lesson; do not extend a “leftover must be positive” shortcut to them.

Worked example 2: reconstruct the total

C code 4

```
int total = 29;
int batch = 6;
int q = total / batch;
int r = total % batch;
int rebuilt = q * batch + r;
printf("%d %d %d\n", q, r, rebuilt);
```

1. After `int total = 29;`: `total=29`.
2. After `int batch = 6;`: `total=29, batch=6`.
3. `q = total / batch` uses $29 / 6$. Four complete batches fit, since $4 * 6 = 24$, while $5 * 6 = 30$ is too large. Now `q=4`.
4. `r = total % batch` is the amount left after those four batches: $29 - 24 = 5$. Now `r=5`; `total`, `batch`, and `q` have not changed.
5. `rebuilt = q * batch + r` groups as $(q * batch) + r$: $4 * 6 + 5 = 29$. Now `rebuilt=29`.
6. The output is `4 5 29` followed by a newline, in the argument order `q, r, rebuilt`.

The check `q * batch + r = total` explains how the two results fit together. A claimed quotient and remainder must both satisfy this relation and, for these positive operands, leave less than one full batch. Merely choosing two numbers whose sum is 29 is not enough.

Practice: try all four before opening the solutions

These are original learning exercises. They are unscored; write intermediate states as well as an answer.

Practice question 1: two successive assignments

Trace both assignments and give the final `a` and `b`. What line is printed?

C code 5

```
int a=8, b=5;
a=a-b;
b=b+a;
printf("%d %d\n",a,b);
```

Practice question 2: quotient and remainder

Find the whole groups and leftover items. Give the printed pair and check that it reconstructs the total.

C code 6

```
int total=47, size=8;
printf("%d %d\n",total/size,total%size);
```

Practice question 3: arithmetic grouping

Evaluate the expression using C's grouping rules. Write the grouping that justifies your answer.

C code 7

```
int value=7+2*5-3;
printf("%d\n",value);
```

Practice question 4: an attempted swap

Does this exchange the starting values of a and b? Give the final pair and explain where the original value of a goes.

C code 8

```
int a=4,b=9;
a=b;
b=a;
printf("%d %d\n",a,b);
```

Practice solutions and wrong-turn feedback

Practice question 1 solution

1. Initially $a=8$, $b=5$.
2. $a=a-b$ calculates $8-5=3$; now $a=3$, $b=5$.
3. $b=b+a$ calculates $5+3=8$; now $a=3$, $b=8$.
4. Output: 3 8 followed by a newline.

If you got 3 13, you reused the starting $a=8$ in the last addition. Cross out that obsolete state after the first assignment. If you got 8 13, you treated the first assignment as if it did not store its result. Both statements change the named destination.

Practice question 2 solution

1. Five full groups of 8 use 40 items; a sixth would require 48, more than 47.
 2. The quotient is 5; the remainder is $47-40=7$.
 3. Output: 5 7 followed by a newline. Check: $5*8+7=47$, with $0 \leq 7$ and $7 < 8$.
- 6 -1 is wrong because integer division does not round $47/8$ up. 5 0 discards the leftover items. 5.875 0 uses a fractional quotient, whereas both operands here are integers and / produces an integer result.

Practice question 3 solution

1. Multiplication binds more tightly: group as $7 + (2 * 5) - 3$.
2. The product is 10.
3. The remaining addition and subtraction group from the left: $(7 + 10) - 3 = 14$.
4. Output: 14 followed by a newline.

42 comes from incorrectly grouping the expression as $(7+2)*5-3$. 11 would come from changing it to $7+2*(5-3)$. Neither grouping is written in the question. Insert parentheses to reveal the actual grouping, not to invent a different one.

Practice question 4 solution

1. Initially $a=4$, $b=9$.
2. $a=b$ copies 9 into a ; now $a=9$, $b=9$. No variable in this program keeps the old $a=4$.
3. $b=a$ reads the current $a=9$ and writes 9 to b ; the pair remains 9, 9.
4. Output: 9 9 followed by a newline. This did not swap the values.

9 4 assumes simultaneous exchange. 4 9 assumes neither assignment takes effect. A simple repair saves the value that would otherwise be overwritten:

C code 9

```
int a = 4;
int b = 9;
int saved = a;
a = b;
b = saved;
printf("%d %d\n", a, b);
```

Here $saved=4$ before a changes. After $a=b$, $a=9$, $b=9$, $saved=4$. After $b=saved$, $a=9$, $b=4$, $saved=4$, and the output is 9 4. The repair works because the old number was explicitly stored, not because C remembers every overwritten value.

Before moving on

You can now trace a current state, distinguish initialization from later assignment, and group small integer calculations. Next, we will separate an expression's returned value from a change it makes to a variable, and extend division to negative integers.

Source notes

The code and traces are original. The language reference is the freely available [WG14 N1570 C11 committee draft](#): entry point §5.1.2.2.1p1; minimum integer range §5.2.4.2.1; initialization §6.7.9p11; simple assignment §6.5.16.1p2; grouping §6.5.5p1 and §6.5.6p1; integer division/remainder §6.5.5p5-6; `printf` §7.21.6.3 and `%d` §7.21.6.1p8. These are semantic references, not a requirement to read a standards document before learning.

Analogy and its limits

Imagine two labelled erasable cards. The x card says 6 and the y card says 4. The instruction $x = x + 3$ means “read 6, calculate 9, erase the x card and write 9.” The instruction $y = x - y$ then reads the cards as they are now: 9 and 4. It writes 5 on y. Copying a number onto a second card does not tie the cards together. If you need an old number after erasing it, first put it on a third card. The cards represent stored values; this analogy makes no claim about how physical memory is laid out.

Quick reference

- `int x = 6;` declares and initializes x; `x = 9;` updates the existing x
- State means current values. Record it after each statement
- In a simple assignment, calculate the right-hand value from the current state, then store it in the left-hand variable
- A copied value does not keep following the source variable
- Parentheses set grouping; `*` / `%` bind more tightly than `+` -
- For nonnegative a and positive b: `a / b` is the number of full groups; `a % b` is the leftover
- Check: `(a / b) * b + a % b` equals a; the remainder is below b
- Match each `%d` in `printf` with one `int` argument; `\n` ends the line
- C11 convention: initialized int values, nonzero evaluated divisors, every intermediate within -32767 to 32767

[Back to contents](#)

चर, असाइनमेंट और क्रमवार ट्रेस

यह प्रोग्राम क्या प्रिंट करेगा?

C कोड 1

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

```
int main(void)
{
    int x = 6;
    int y = 4;
    x = x + 3;
    y = x - y;
    printf("%d %d\n", x, y);
    return 0;
}
```

इन दो संख्याओं को जानने के लिए हमें मानों में होने वाले बदलाव क्रम से देखने होंगे। इस पाठ के अंत तक आप हर स्टेटमेंट के बाद प्रत्येक चर का मान बता सकेंगे और समझा सकेंगे कि वही परिणाम क्यों प्रिंट होता है।

जितना सिंटैक्स अभी चाहिए, उतना समझें

चर (variable) किसी मान को रखने की नाम वाली जगह है। यहाँ x और y में एक-एक `int` है: यानी 6, 0 या -3 जैसा पूर्णांक। `int x = 6;` में x की घोषणा होती है और उसका शुरुआती मान 6 रखा जाता है। इसे इनिशियलाइज़ेशन (initialization) कहते हैं। बाद का `x = x + 3;` उसी चर का मान बदलता है; वह दूसरा x घोषित नहीं करता।

यहाँ हर घोषणा या व्यंजक वाले स्टेटमेंट के अंत में सेमीकोलन है। ब्रेसिज़ के बीच `main` का मुख्य भाग है। इन पूरे प्रोग्रामों में `int main(void)` वह फ़ंक्शन है जहाँ से काम शुरू होता है; यहाँ `void` बताता है कि इसमें कोई पैरामीटर नहीं है। `#include <stdio.h>` से `printf` के उपयोग के लिए आवश्यक घोषणा मिलती है। इस `include` पंक्ति के अंत में सेमीकोलन नहीं आता। `return 0;` सफल निष्पादन समाप्त करता है। फ़ंक्शन विस्तार से सीखने से पहले भी आप ब्रेसिज़ के भीतर के स्टेटमेंट का ट्रेस कर सकते हैं। एक घोषणा में समान प्रकार के कई चर भी आ सकते हैं: `int a=8, b=5;` से a को 8 और b को 5 मिलता है। कॉमा उसी घोषणा में चरों के नाम और उनके शुरुआती मान अलग करता है।

`printf("%d %d\n", x, y);` दो पूर्णांक प्रिंट करता है। पहले `%d` की जगह x का मान और दूसरे की जगह y का मान आता है। उद्धरण-चिह्नों के भीतर का स्पेस संख्याओं के बीच छपता है और `\n` आउटपुट की पंक्ति समाप्त करता है। उद्धरण-चिह्न, `%d` और `\n` खुद नहीं छपते। इस कॉल में दोनों आर्गुमेंट केवल मौजूदा मान पढ़ते हैं; किसी चर को बदलते नहीं हैं। इसलिए C इम्प्लीमेंटेशन इन आर्गुमेंट के मान जिस क्रम में भी निकाले, इस परिणाम पर असर नहीं पड़ता।

केवल साधारण पूर्णांक गणना चाहिए: जैसे $17 - 9$ का मान 8 और $6 * 4$ का मान 24 है। पहले से प्रोग्रामिंग आना आवश्यक नहीं है। कोड में `*` गुणा का ऑपरेटर है; अक्षर x चर का नाम है, गुणा का चिह्न नहीं।

इस मॉड्यूल के कोड की सीमाएँ

पढ़ाने के लिए हम स्पष्ट रूप से ISO C11 के नियम लेते हैं। निश्चित परिणाम वाले हर उदाहरण में चरों को ज्ञात शुरुआती मान दिए गए हैं, सामान्य `signed int` लिया गया है और गणना के हर बीच के परिणाम को -32767 से 32767 के भीतर रखा गया है। जिस भाग या शेषफल की गणना वास्तव में होती है, उसका भाजक शून्य नहीं है। इससे उत्तर किसी खास मशीन के पूर्णांक आकार या निरूपण पर निर्भर नहीं करता। हम `int` को 32 बिट का मानकर नहीं चलते। इनपुट घोषणा में लिखे मान हैं; कीबोर्ड से इनपुट लेना आवश्यक नहीं है।

अवस्था: अभी कौन-से मान मौजूद हैं

अवस्था (state) किसी क्षण सभी चरों के वर्तमान मानों का समूह है। ट्रेस उन अवस्थाओं को क्रम से लिखने का तरीका है। हर स्टेटमेंट के बाद नया रिकॉर्ड लिखें। यदि स्टेटमेंट किसी चर को नया मान नहीं देता, तो उस चर का पिछला मान ही आगे लिखें।

$x = x + 3$; जैसे सरल असाइनमेंट (assignment) को पढ़ने का तरीका है:

1. दाईं ओर के वर्तमान मान पढ़ें: यहाँ x का पुराना मान।
2. दाईं ओर के व्यंजक का मान निकालें: पुराना x और 3 जोड़ें।
3. परिणाम बाईं ओर लिखे चर में रखें: यहाँ x में।
4. अगले स्टेटमेंट के लिए इसी नई अवस्था का उपयोग करें।

दोनों जगह लिखे x की भूमिकाएँ अलग हैं: दाईं ओर वाला मान देता है और बाईं ओर वाला परिणाम रखने की जगह बताता है। = किसी मान को रखने का निर्देश है। यह ऐसा गणितीय समीकरण नहीं है जिसे हमेशा सही रहना हो। यदि अभी x का मान 6 है, तो $x = x + 3$; उसे ठीक तरह से 9 करता है; यह "6 बराबर 9" वाला असंभव समीकरण नहीं है। पढ़ने का यह तरीका यहाँ के सरल असाइनमेंट के लिए है, हर तरह के बदलाव करने वाले जटिल व्यंजक के लिए नहीं।

हल किया हुआ उदाहरण 1: वर्तमान अवस्था लें

शुरुआती प्रोग्राम पर लौटें। दोनों चरों के मान सामने रखें, उस चर का भी जिसका मान नहीं बदला।

1. `int x = 6`; के बाद: $x=6$; अभी y की घोषणा नहीं हुई है।
2. `int y = 4`; के बाद: $x=6$, $y=4$ ।
3. $x = x + 3$; में $6 + 3 = 9$ निकालें। स्टेटमेंट के बाद: $x=9$, $y=4$ ।
4. $y = x - y$; में $9 - 4 = 5$ निकालें। स्टेटमेंट के बाद: $x=9$, $y=5$ ।
5. `printf` में $x=9$ और $y=5$ पढ़े जाते हैं। आउटपुट 9 5 है, जिसके बाद नई पंक्ति आती है। चरों के मान $x=9$, $y=5$ ही रहते हैं।

दूसरा असाइनमेंट अपडेट हुए $x=9$ को पढ़ता है। यदि फिर से शुरुआती $x=6$ लें, तो गलत $y=2$ मिलेगा। हर स्टेटमेंट पिछले स्टेटमेंट द्वारा छोड़ी गई अवस्था से शुरू होता है; ये असाइनमेंट एक साथ नहीं होते।

कॉपी का अर्थ स्थायी जुड़ाव नहीं है

किसी नए चर का शुरुआती मान पहले से मौजूद चर से लिया जा सकता है। नीचे `copy` को वही मान मिलता है जो उस समय `source` में है। वह `source` के साथ हमेशा बदलने वाला सूत्र नहीं बनता।

इस पाठ के सभी छोटे कोड ब्लॉक उसी `main` के भीतर रखे जाने वाले अलग-अलग उदाहरण हैं। ऊपर `#include <stdio.h>` और अंत में `return 0`; रहेगा। हर ब्लॉक नई शुरुआत वाला उदाहरण है; सभी ब्लॉक एक साथ न जोड़ें।

C कोड 2

```
int source = 6;
int copy = source;
source = 10;
printf("%d %d\n", source, copy);
```

घोषणाओं के बाद: `source=6, copy=6`। `source = 10`; के बाद: `source=10, copy=6`। आउटपुट 10 6 है। `source` को नया मान देने से `copy` में भी नया मान नहीं लिखा जाता। ट्रेस में रखी हुई संख्याएँ दर्ज होती हैं, लगातार कायम रहने वाले बीजगणितीय संबंध नहीं।

अंकगणितीय व्यंजक और समूह बनाना

व्यंजक (expression) बताता है कि कौन-सा मान निकालना है। इस पाठ के पूर्णांक व्यंजकों में:

- + जोड़ता है, - घटाता है और * गुणा करता है
- / पूर्णांक भागफल देता है; % उसी भाग का शेषफल देता है
- कोष्ठक किसी छोटे व्यंजक का समूह स्पष्ट करते हैं
- *, / और % की प्राथमिकता + और - से अधिक है

प्राथमिकता तय करती है कि किस ऑपरेटर के साथ कौन-से ऑपरेंड जुड़ते हैं। यह ऑपरेंड का मान निकालने के समय-क्रम का सामान्य वादा नहीं है। हमारे अंकगणितीय उदाहरण व्यंजक के भीतर चर बदले बिना उनके मान पढ़ते हैं, इसलिए सही समूह से एक निश्चित परिणाम मिलता है।

C कोड 3

```
int plain = 2 + 3 * 4;
int grouped = (2 + 3) * 4;
printf("%d %d\n", plain, grouped);
```

`plain` का समूह $2 + (3 * 4)$ है: $3 * 4$ से 12, फिर $2 + 12$ से 14 मिलता है। `grouped` के कोष्ठक $(2 + 3)$ से 5 लेने को कहते हैं, फिर $5 * 4$ से 20 मिलता है। आउटपुट 14 20 है। C हर अंकगणितीय व्यंजक को बस सबसे बाएँ चिह्न से सबसे दाएँ चिह्न की ओर बढ़ते हुए नहीं निकालता।

पूरे समूह और बची वस्तुएँ

यदि 23 वस्तुओं के 5 वस्तुओं वाले समूह बनाएँ, तो चार पूरे समूहों में 20 वस्तुएँ लगेंगी और 3 बचेंगी। इसलिए $23 / 5$ का मान 4 और $23 \% 5$ का मान 3 है। भागफल पूर्णांक है; वह 4.6 को निकटतम पूर्णांक में राउंड करके नहीं मिलता। जब कुल संख्या ऋणात्मक नहीं है और समूह का आकार धनात्मक है, तो शेषफल कम-से-कम शून्य और समूह के आकार से छोटा होगा। शेषफल शून्य होने पर भाग पूरा लगता है।

चिह्नों की भूमिकाएँ अलग रखें: दो पूर्णांक व्यंजकों के बीच % शेषफल है; `printf` के फॉर्मेट में %d पूर्णांक प्रिंट करने का निर्देश है। इन उदाहरणों में इनमें से किसी का अर्थ प्रतिशत नहीं है। कभी शून्य से भाग न दें और शून्य भाजक के साथ शेषफल न निकालें। ऋणात्मक मान अगले पाठ में आएँगे; उन पर “शेषफल हमेशा धनात्मक होगा” वाला अनुमान न लगाएँ।

हल किया हुआ उदाहरण 2: कुल संख्या फिर बनाएँ

C कोड 4

```
int total = 29;
int batch = 6;
int q = total / batch;
int r = total % batch;
int rebuilt = q * batch + r;
printf("%d %d %d\n", q, r, rebuilt);
```

1. `int total = 29`; के बाद: `total=29`।
2. `int batch = 6`; के बाद: `total=29, batch=6`।

3. $q = \text{total} / \text{batch}$ में 29 / 6 लिया जाता है। चार पूरे समूह बनते हैं, क्योंकि $4 * 6 = 24$, जबकि $5 * 6 = 30$ ज्यादा है। अब $q=4$ ।
4. $r = \text{total} \% \text{batch}$ उन चार समूहों के बाद बची मात्रा है: $29 - 24 = 5$ । अब $r=5$; total , batch और q नहीं बदले।
5. $\text{rebuilt} = q * \text{batch} + r$ का समूह $(q * \text{batch}) + r$ है: $4 * 6 + 5 = 29$ । अब $\text{rebuilt}=29$ ।
6. आर्गुमेंट के क्रम $q, r, \text{rebuilt}$ के अनुसार आउटपुट 4 5 29 है, जिसके बाद नई पंक्ति आती है।

जाँच $q * \text{batch} + r = \text{total}$ दिखाती है कि दोनों परिणाम कैसे जुड़ते हैं। बताए गए भागफल और शेषफल को यह संबंध सही करना चाहिए और इन धनात्मक मानों के लिए शेष एक पूरे समूह से कम होना चाहिए। केवल ऐसी दो संख्याएँ चुनना जिनका जोड़ 29 हो, पर्याप्त नहीं है।

अभ्यास: हल देखने से पहले चारों प्रश्न करें

ये सीखने के लिए बनाए गए मौलिक प्रश्न हैं। इनके अंक नहीं हैं; उत्तर के साथ बीच की अवस्थाएँ भी लिखें।

अभ्यास प्रश्न 1: क्रम से दो असाइनमेंट

दोनों असाइनमेंट का ट्रेस करके a और b के अंतिम मान बताइए। कौन-सी पंक्ति प्रिंट होती है?

C कोड 5

```
int a=8, b=5;
a=a-b;
b=b+a;
printf("%d %d\n",a,b);
```

अभ्यास प्रश्न 2: भागफल और शेषफल

पूरे समूहों और बची वस्तुओं की संख्या निकालिए। प्रिंट होने वाली जोड़ी लिखिए और जाँचिए कि उससे कुल संख्या फिर बनती है।

C कोड 6

```
int total=47, size=8;
printf("%d %d\n",total/size,total%size);
```

अभ्यास प्रश्न 3: अंकगणित का समूह

C के समूह-नियमों से व्यंजक का मान निकालिए। अपने उत्तर को समझाने वाला समूह भी लिखिए।

C कोड 7

```
int value=7+2*5-3;
printf("%d\n",value);
```

अभ्यास प्रश्न 4: मान बदलने की कोशिश

क्या यह शुरुआती a और b के मान आपस में बदलता है? अंतिम जोड़ी बताइए और समझाइए कि a का मूल मान कहाँ जाता है।

C कोड 8

```
int a=4,b=9;
a=b;
b=a;
printf("%d %d\n",a,b);
```

अभ्यास के हल और गलत उत्तरों का कारण

अभ्यास प्रश्न 1 का हल

1. शुरुआत में $a=8$, $b=5$ ।
2. $a=a-b$ में $8-5=3$ मिलता है; अब $a=3$, $b=5$ ।
3. $b=b+a$ में $5+3=8$ मिलता है; अब $a=3$, $b=8$ ।
4. आउटपुट: 3 8, फिर नई पंक्ति।

यदि 3 13 मिला, तो अंतिम जोड़ में फिर से शुरुआती $a=8$ लिया गया। पहले असाइनमेंट के बाद उस पुरानी अवस्था को काट दें। यदि 8 13 मिला, तो पहले असाइनमेंट का परिणाम चर में रखा ही नहीं गया। दोनों स्टेटमेंट अपने बाईं ओर लिखे चर का मान बदलते हैं।

अभ्यास प्रश्न 2 का हल

1. 8 वाले पाँच पूरे समूहों में 40 वस्तुएँ लगती हैं; छठे के लिए 48 चाहिए, जो 47 से अधिक है।
2. भागफल 5 है; शेषफल $47-40=7$ है।
3. आउटपुट: 5 7, फिर नई पंक्ति। जाँच: $5*8+7=47$, साथ में $0 \leq 7$ और $7 < 8$ ।

6 -1 गलत है, क्योंकि पूर्णांक भाग में $47/8$ को ऊपर राउंड नहीं किया जाता। 5 0 बची हुई वस्तुओं को छोड़ देता है। 5.875 0 में भिन्न वाला भागफल लिया गया है, जबकि यहाँ दोनों ऑपरेंड पूर्णांक हैं और / का परिणाम भी पूर्णांक है।

अभ्यास प्रश्न 3 का हल

1. गुणा की प्राथमिकता अधिक है: समूह $7 + (2 * 5) - 3$ बनाएँ।
2. गुणनफल 10 है।
3. बचे जोड़ और घटाव का समूह बाईं ओर से है: $(7 + 10) - 3 = 14$ ।
4. आउटपुट: 14, फिर नई पंक्ति।

42 गलत समूह $(7+2)*5-3$ से मिलता है। 11 तब मिलेगा जब व्यंजक को बदलकर $7+2*(5-3)$ कर दें। प्रश्न में इनमें से कोई समूह नहीं दिया गया है। कोष्ठक लगाकर वास्तविक समूह दिखाएँ, नया व्यंजक न बनाएँ।

अभ्यास प्रश्न 4 का हल

1. शुरुआत में $a=4$, $b=9$ ।
2. $a=b$ से 9 की कॉपी a में जाती है; अब $a=9$, $b=9$ । इस प्रोग्राम का कोई चर पुराना $a=4$ नहीं रखता।
3. $b=a$ वर्तमान $a=9$ पढ़कर b में 9 रखता है; जोड़ी 9, 9 ही रहती है।
4. आउटपुट: 9 9, फिर नई पंक्ति। मान आपस में नहीं बदले।

9 4 मान लेता है कि दोनों मान एक साथ बदलते हैं। 4 9 मान लेता है कि किसी असाइनमेंट का असर नहीं हुआ। सरल सुधार यह है कि जिस पुराने मान के ऊपर नया मान लिखा जाएगा, उसे पहले बचा लें:

C कोड 9

```
int a = 4;
int b = 9;
int saved = a;
a = b;
b = saved;
printf("%d %d\n", a, b);
```

यहाँ a बदलने से पहले $saved=4$ हो जाता है। $a=b$ के बाद $a=9$, $b=9$, $saved=4$ । $b=saved$ के बाद $a=9$, $b=4$, $saved=4$, और आउटपुट 9 4 है। सुधार इसलिए काम करता है क्योंकि पुरानी संख्या को साफ़ तौर पर अलग चर में रखा गया; C अपने-आप हर मिटा हुआ पुराना मान याद नहीं रखता।

आगे बढ़ने से पहले

अब आप वर्तमान अवस्था का ट्रेस कर सकते हैं, इनिशियलाइज़ेशन को बाद के असाइनमेंट से अलग पहचान सकते हैं और छोटे पूर्णांक व्यंजकों के समूह बना सकते हैं। अगले पाठ में हम व्यंजक से मिलने वाले मान और उससे चर में होने वाले बदलाव को अलग करेंगे, तथा ऋणात्मक पूर्णांकों के भाग को समझेंगे।

स्रोत संबंधी नोट

कोड और ट्रेस मौलिक हैं। भाषा का संदर्भ सार्वजनिक [WG14 N1570 C11 समिति-ड्राफ्ट](#) है: प्रवेश फ़ंक्शन §5.1.2.2.1p1; पूर्णांक की न्यूनतम सीमा §5.2.4.2.1; इनिशियलाइज़ेशन §6.7.9p11; सरल असाइनमेंट §6.5.16.1p2; समूह §6.5.5p1 और §6.5.6p1; पूर्णांक भाग/शेषफल §6.5.5p5-6; printf §7.21.6.3 और %d §7.21.6.1p8। ये नियमों के संदर्भ हैं; सीखना शुरू करने से पहले मानक दस्तावेज़ पढ़ना आवश्यक नहीं है।

उपमा और उसकी सीमाएँ

दो नाम लिखे मिटाए जा सकने वाले कार्ड सोचें। x वाले कार्ड पर 6 और y वाले पर 4 है। $x = x + 3$ का अर्थ है "6 पढ़ें, 9 निकालें, x वाले कार्ड का पुराना मान मिटाकर 9 लिखें।" इसके बाद $y = x - y$ कार्डों के अभी के मान, यानी 9 और 4, पढ़ता है और y पर 5 लिखता है। दूसरे कार्ड पर संख्या की कॉपी करने से दोनों कार्ड आपस में बँध नहीं जाते। मिटाने के बाद भी पुरानी संख्या चाहिए तो पहले उसे तीसरे कार्ड पर रख लें। कार्ड रखे हुए मानों का उदाहरण हैं; इससे वास्तविक मेमोरी की बनावट के बारे में कोई दावा नहीं किया जा रहा।

त्वरित संदर्भ

- `int x = 6;` में x की घोषणा और इनिशियलाइज़ेशन है; $x = 9;$ मौजूदा x को बदलता है
- अवस्था का अर्थ वर्तमान मान हैं। हर स्टेटमेंट के बाद उन्हें लिखें
- सरल असाइनमेंट में वर्तमान अवस्था से दाईं ओर का मान निकालें, फिर बाईं ओर के चर में रखें
- कॉपी किया गया मान स्रोत चर के साथ अपने-आप नहीं बदलता
- कोष्ठक समूह तय करते हैं; $*$ / $%$ की प्राथमिकता $+$ - से अधिक है
- गैर-ऋणात्मक a और धनात्मक b के लिए: a / b पूरे समूह और $a \% b$ बची मात्रा है
- जाँच: $(a / b) * b + a \% b$ का मान a है; शेषफल b से छोटा है
- `printf` के हर `%d` के लिए एक `int` आर्गुमेंट रखें; `\n` पंक्ति समाप्त करता है
- C11 की सीमा: ज्ञात शुरुआती `int` मान, वास्तव में गणना होने पर भाजक शून्य नहीं, हर बीच का परिणाम -32767 से 32767 के भीतर

[विषय सूची पर लौटें](#)

Expression values and safe updates

What happens when an integer is negative?

C code 1

```
#include <stdio.h>

int main(void)
{
    int a = -19;
    int b = 4;
    int q = a / b;
    int r = a % b;
    int z = 2 + q * 3;
    printf("%d %d %d\n", q, r, z);
    return 0;
}
```

The quotient is not obtained by always rounding downward, and the remainder need not be positive. To trace this program, we need the exact integer rule. Later in this lesson we will keep an expression's value separate from any change it makes to a variable.

Carry forward the tracing rules

You need the previous lesson's current-state method, initialization, simple assignment, arithmetic grouping, and %d output. Shorter code blocks below are separate bodies for the same complete wrapper shown above: #include <stdio.h>, int main(void), braces, and a final return 0;. Start each block with its own initial state.

We continue to use ISO C11, initialized signed int values, and intermediate results within -32767 to 32767. Every evaluated divisor is nonzero and each quotient fits in int. We will label one unsafe expression for recognition only; it is not an output-prediction program to run.

Integer division: move toward zero

For integer operands, / discards the fractional part of the mathematical quotient. "Toward zero" tells us which integer remains on the negative side. The mathematical quotient of -19 by 4 is -4.75; discarding its fractional part gives -4, which is nearer zero. It does not give -5.

Use three steps rather than guessing the remainder's sign:

1. Find the integer quotient q by truncating toward zero.
2. Calculate the remainder using $r = a - q * b$.
3. Check that $q * b + r$ reconstructs a .

These arithmetic equalities describe the values; an equality written here is not a C assignment statement to paste into the program. For this lesson's bounded operands, the multiplication and subtraction in the check also fit safely in int.

The remainder is zero or has the sign of the dividend a . Its magnitude is smaller than the magnitude of the divisor b . The divisor's sign affects the quotient, but it does not determine the sign of a nonzero remainder. This is C remainder; do not replace it with a convention from another language that always gives a nonnegative result for a positive divisor.

Worked example 1: quotient, remainder, then an expression

Trace the opening program from the declarations.

1. After the first two declarations: $a = -19$, $b = 4$.
2. $q = a / b$ yields -4 , because the fractional part of -4.75 is discarded toward zero. Now $q = -4$.
3. $r = a \% b$ gives -3 . Check: $(-4) * 4 = -16$, and $-19 - (-16) = -3$. Now $r = -3$.
4. For $z = 2 + q * 3$, multiplication binds more tightly: $q * 3 = -12$, then $2 + (-12) = -10$. Now $z = -10$.
5. The output is $-4 -3 -10$ followed by a newline, in the order q, r, z . Neither a nor b has changed.

The pair $-5, 1$ reconstructs -19 , but it uses downward rounding, so it is not C11's quotient/remainder pair. Reconstruction alone is not sufficient: it must agree with truncation toward zero. The pair $-4, 3$ uses the right quotient but fails reconstruction: $-16 + 3$ is -13 .

Check the sign rule in three nearby cases

C code 2

```
printf("%d %d\n", 19 / -4, 19 % -4);
printf("%d %d\n", -19 / -4, -19 % -4);
printf("%d %d\n", -20 / 4, -20 % 4);
```

1. $19 / -4$ gives -4 ; $19 - (-4) * (-4) = 3$. The first line is $-4 \ 3$.
2. $-19 / -4$ gives 4 ; $-19 - 4 * (-4) = -3$. The second line is $4 \ -3$.
3. $-20 / 4$ gives -5 ; $-20 - (-5) * 4 = 0$. The third line is $-5 \ 0$.

A negative dividend can still have remainder zero. "The remainder follows the dividend's sign" needs that zero exception. These calls only read constants, so argument-evaluation order cannot change them.

Grouping equal-priority operations

$*$, $/$, and $\%$ share a precedence level and group from the left. For example, $18 / 4 * 3$ groups as $(18 / 4) * 3$. The first quotient is 4 , then the product is 12 . It does not group as $18 / (4 * 3)$, whose result would be 1 .

Discarding a fraction can happen at an intermediate division. You cannot generally do all calculations using fractions and truncate only once at the end: $(18 / 4) * 3$ in C has already lost the fractional part before the multiplication uses that quotient. Parentheses can change which division is requested. They do not generally create a left-to-right evaluation order for side effects. Our examples in this section have no side effects inside the arithmetic expression.

A value calculation is different from an update

C code 3

```
int x = 4;
int next = x + 1;
printf("%d %d\n", x, next);
```

$x + 1$ produces 5, which initializes next. It does not store anything into x. The output is 4 5. In contrast, $x = x + 1$; would store the result back into x. Ask two questions whenever an expression might update a variable: "What value does it supply?" and "What is the state when this statement is finished?"

Compound assignments to one variable

For the ordinary int variable used here, $\text{score} += 5$; has the same numerical effect as $\text{score} = \text{score} + 5$;. Similarly, $\text{score} -= 4$; subtracts 4 and stores the result, and $\text{score} /= 5$; stores the integer quotient. These operators combine arithmetic with assignment; $+=$ is not just a different way to write $+$.

C code 4

```
int score = 17;
score += 5;
score -= 4;
score /= 5;
printf("%d\n", score);
```

1. Initialization gives $\text{score}=17$.
2. After $\text{score} += 5$;: $\text{score}=22$.
3. After $\text{score} -= 4$;: $\text{score}=18$.
4. After $\text{score} /= 5$;: $\text{score}=3$, since integer $18 / 5$ is 3.
5. The output is 3 followed by a newline.

The compound form evaluates its left-hand destination only once. We limit this lesson to a plain variable, so there are no extra address calculations to track. Do not treat the expanded form as a permission to duplicate more complicated expressions. A compound assignment still needs the same arithmetic checks: no evaluated division by zero, and no out-of-range signed result.

Increment: two questions, not one slogan

The increment operator $++$ adds one to a variable. Its position determines the value supplied by the expression:

- In $x++$, called postfix increment, the expression supplies the old value of x. The variable is also increased by one
- In $++x$, called prefix increment, the expression supplies the increased value of x

These are not two different amounts of increase: each increases x by one. If the old value is 3, the state after a separate $x++$; or $++x$; statement has $x=4$. The difference matters when another variable receives the expression's value.

We place each update in its own full expression. An initializer such as the one in `int before = x++;` is a full expression; so is the expression in a standalone update statement. Its required updates are complete before the next full expression begins. Postfix does not mean “wait until some later statement to update”. Do not turn several separate statements into a combined expression while assuming their meaning will be preserved.

Worked example 2: the supplied value and the final state

C code 5

```
int x = 3;
int before = x++;
int after = ++x;
printf("%d %d %d\n", before, after, x);
```

1. After `int x = 3;` `x=3`.
2. In `int before = x++;`, the postfix expression supplies 3 to initialize `before`. By the end of this declaration, `x` has increased to 4. State: `x=4`, `before=3`.
3. In `int after = ++x;`, `x` increases from 4 to 5, and the prefix expression supplies 5 to initialize `after`. State: `x=5`, `before=3`, `after=5`.
4. The output is 3 5 5 followed by a newline, because the arguments are `before`, `after`, then `x`.

`before` stays 3; it is a stored copy of a value, not another name for `x`. If you got 3 4 5, you gave the prefix expression an old value. If you got 4 5 5, you gave the postfix expression a new value. Record the supplied value and resulting state separately to avoid both mistakes.

A precise safety boundary

C does not promise to evaluate the operands of ordinary `+` from left to right. When an expression changes a variable and also reads that variable elsewhere without the required sequencing, it can have undefined behavior. Undefined behavior means C11 gives no prescribed result for the program; it is not merely a choice between two normal answers, and a crash is not required.

Keep the updates in separate full expressions as in worked example 2. Parentheses that change arithmetic grouping do not, by themselves, establish sequencing between operands. A single compiler run is not a rule for an unsafe expression. Practice question 4 asks you to recognize this boundary instead of guessing a number.

Practice: attempt before the solutions

These four original exercises are unscored. For each defined example, write the expression values and final state. For the unsafe example, give a semantic classification instead of a numeric answer.

Practice question 1: negative dividend

Find the C11 quotient and remainder. Give the output and show the reconstruction check.

C code 6

```
int a=-23,b=6;
printf("%d %d\n",a/b,a%b);
```

Practice question 2: two different increment forms

Find the final n and the values stored in p and q . Pay attention to the printed argument order.

C code 7

```
int n=4;
int p=++n;
int q=n++;
printf("%d %d %d\n",n,p,q);
```

Practice question 3: the effect of parentheses

Find both results and explain why they differ.

C code 8

```
int a=24/5*2;
int b=24/(5*2);
printf("%d %d\n",a,b);
```

Practice question 4: classify, do not execute

Does C11 give a defined value for r in this fragment? Explain whether a single numeric output could be required if r were printed afterward. Do not run the fragment to choose a number.

C code 9

```
int i=4;
int r=i++ + i;
```

Practice solutions and wrong-turn feedback

Practice question 1 solution

1. The mathematical quotient of -23 and 6 lies between -4 and -3 ; truncation toward zero gives $q = -3$.
 2. $q * b = (-3) * 6 = -18$.
 3. The remainder is $a - q * b = -23 - (-18) = -5$.
 4. Check: $(-3) * 6 + (-5) = -23$; the remainder's magnitude 5 is below 6 , and its sign agrees with the negative dividend.
 5. Output: -3 -5 followed by a newline.
- -4 1 applies floor division instead of C11 truncation. -3 5 incorrectly forces the remainder positive and reconstructs -13 , not -23 . -3 -1 has a plausible sign but fails the arithmetic check: $-18 - 1 = -19$.

Practice question 2 solution

1. Initially $n=4$.
 2. `int p=++n;` first increases n to 5 and supplies 5 to p . State: $n=5$, $p=5$.
 3. `int q=n++;` supplies the current 5 to q and increases n to 6 by the end of the declaration. State: $n=6$, $p=5$, $q=5$.
 4. The print order is n , p , q , so the output is 6 5 5 followed by a newline.
- 6 5 6 gives postfix the new value. 5 5 5 forgets that postfix still changes n . 6 4 5 gives prefix the old value. Both updates have already happened when `printf` runs.

Practice question 3 solution

1. In the first declaration, equal-priority `/` and `*` group from the left: $a=(24/5)*2$.
 2. Integer $24/5$ is 4; then $4*2=8$. Thus $a=8$.
 3. In the second declaration, parentheses form the divisor $5*2=10$. Integer $24/10$ is 2. Thus $b=2$.
 4. Output: 8 2 followed by a newline.
- 2 2 silently adds parentheses to the first expression. 9 2 uses the fractional intermediate 4.8 and truncates only after multiplying; the C expression already used an integer quotient of 4. 8 8 ignores the parentheses in the second declaration. Grouping and integer division are both needed for the explanation.

Practice question 4 solution

The fragment has undefined behavior under C11. Within `i++ + i`, the increment's modification of `i` is unsequenced relative to the other operand's read of `i`. Ordinary `+` does not supply the sequencing needed to make this safe. The rule is N1570 §6.5p2.

There is no C11-prescribed numeric value for `r`, and no single output a conforming trace question may require. 8 and 9 are tempting calculations based on imagined evaluation timings, not valid standard-defined answers. "Either 8 or 9" is also incorrect: undefined behavior does not restrict the implementation to those two outcomes. "It must fail to compile" and "it must crash" are not guaranteed either. Adding parentheses around `i++` or around the whole sum does not repair the sequencing.

If the intended operation is "keep the old value, increment, then add old and new", express those steps separately:

C code 10

```
int i = 4;
int old = i;
i = i + 1;
int r = old + i;
printf("%d %d\n", r, i);
```

1. Initially `i=4`.
2. `old=i` stores 4 in `old`.
3. `i=i+1` makes `i=5`.
4. `r=old+i` uses `4+5=9`.
5. Output: 9 5 followed by a newline, for `r` and `i`.

This is a new, defined program with an explicitly chosen intention. It does not prove that the unsafe fragment "really meant 9". The unsafe fragment had no defined meaning to preserve.

Before moving on

You can now distinguish a calculation from a stored update, trace prefix and postfix in separate full expressions, and apply C11's integer quotient/remainder rule. Next, conditions will use expression values to decide which statements run.

Source notes

The code, calculations, and feedback are original. Precise references in the [WG14 N1570 C11 committee draft](#): integer division/remainder §6.5.5p5-6; left grouping §6.5.5p1; compound assignment §6.5.16.2p3; postfix §6.5.2.4p2; prefix §6.5.3.1p2; full expressions §6.8p4 and their sequencing §5.1.2.3p3; unsequenced modification/read §6.5p2; out-of-range results §6.5p5. Compiler output may corroborate the defined examples, but it is not the language definition.

Analogy and its limits

Imagine a counter and a small receipt. With postfix, the receipt records the old counter reading while the counter advances by one; with prefix, the counter advances and the receipt records the new reading. After the whole transaction, the counter has advanced in either case, but the receipts can differ. Make one transaction complete before starting the next. This picture helps with separate full expressions; it does not define a safe order for several updates squeezed into one C expression.

Quick reference

- C11 integer / truncates toward zero: $-19 / 4$ is -4
- For nonzero b and a representable quotient: $(a / b) * b + a \% b$ equals a
- A nonzero remainder has the dividend's sign; its magnitude is below the divisor's magnitude
- $*$ / $\%$ group from the left: $24 / 5 * 2$ means $(24 / 5) * 2$
- For a plain int variable, $x += k$, $x -= k$, and $x /= k$ store the corresponding arithmetic result
- $x + 1$ calculates a value without changing x ; $x = x + 1$ stores it
- $x++$ supplies the old value; $++x$ supplies the new value; both increase x by one
- Separate full expressions make the shown updates finish before the next begins
- $i++ + i$ has undefined behavior, not a prescribed numeric answer
- Parentheses control grouping; they do not generally sequence side effects
- Keep every intermediate in range and every evaluated divisor nonzero

[Back to contents](#)

व्यंजक का मान और सुरक्षित अपडेट

पूर्णांक ऋणात्मक होने पर क्या बदलता है?

C कोड 1

`#include <stdio.h>``int main(void)``{` `int a = -19;` `int b = 4;` `int q = a / b;` `int r = a % b;` `int z = 2 + q * 3;` `printf("%d %d %d\n", q, r, z);` `return 0;``}`

भागफल हमेशा नीचे की ओर राउंड करके नहीं मिलता और शेषफल हमेशा धनात्मक नहीं होता। इस प्रोग्राम का ट्रेस करने के लिए पूर्णांक भाग का सही नियम चाहिए। इसी पाठ में आगे हम व्यंजक से मिलने वाले मान को उससे चर में होने वाले बदलाव से अलग रखेंगे।

पिछले ट्रेस के नियम साथ रखें

पिछले पाठ की वर्तमान अवस्था वाली विधि, इनिशियलाइज़ेशन, सरल असाइनमेंट, अंकगणितीय समूह और %d वाला आउटपुट आना चाहिए। नीचे के छोटे कोड ब्लॉक ऊपर दिखाए गए पूरे ढाँचे के अलग-अलग मुख्य भाग हैं: `#include <stdio.h>`, `int main(void)`, ब्रेसिज़ और अंत में `return 0;`। हर ब्लॉक अपनी शुरुआती अवस्था से शुरू होता है।

हम ISO C11, ज्ञात शुरुआती signed int मान और -32767 से 32767 के भीतर के बीच के परिणाम लेते रहेंगे। वास्तव में होने वाली हर भाग-गणना का भाजक शून्य नहीं है और भागफल int में आ सकता है। एक असुरक्षित व्यंजक को केवल पहचानने के लिए अलग चिह्नित किया जाएगा; उसे चलाकर आउटपुट का अनुमान नहीं लगाना है।

पूर्णांक भाग: शून्य की ओर जाएँ

पूर्णांक ऑपरेंड के लिए / गणितीय भागफल का भिन्न वाला हिस्सा हटा देता है। ऋणात्मक तरफ कौन-सा पूर्णांक बचेगा, यह “शून्य की ओर” से समझ आता है। -19 को 4 से भाग देने पर गणितीय भागफल -4.75 है; भिन्न वाला हिस्सा हटाने पर -4 मिलता है, जो शून्य के अधिक पास है। -5 नहीं मिलता।

शेषफल का चिह्न अंदाज़ से चुनने के बजाय तीन कदम लें:

1. शून्य की ओर काटकर पूर्णांक भागफल q निकालें।
2. $r = a - q * b$ से शेषफल निकालें।
3. जाँचें कि $q * b + r$ से फिर a मिलता है।

ये गणितीय बराबरियाँ मानों के संबंध बताती हैं; यहाँ लिखी हर बराबरी प्रोग्राम में चिपकाने योग्य C असाइनमेंट नहीं है। इस पाठ की सीमित संख्याओं के लिए जाँच में होने वाला गुणा और घटाव भी int के भीतर सुरक्षित हैं।

शेषफल या तो शून्य है या उसका चिह्न भाज्य a के चिह्न जैसा है। उसका परिमाण भाजक b के परिमाण से छोटा है। भाजक का चिह्न भागफल को प्रभावित करता है, लेकिन गैर-शून्य शेषफल का चिह्न उससे तय नहीं होता। यह C का शेषफल है; इसे किसी दूसरी भाषा के ऐसे नियम से न बदलें जिसमें धनात्मक भाजक के साथ परिणाम हमेशा गैर-ऋणात्मक होता हो।

हल किया हुआ उदाहरण 1: भागफल, शेषफल, फिर व्यंजक

शुरुआती प्रोग्राम को घोषणाओं से ट्रेस करें।

1. पहली दो घोषणाओं के बाद: $a = -19$, $b = 4$ ।
2. $q = a / b$ से -4 मिलता है, क्योंकि -4.75 का भिन्न वाला हिस्सा शून्य की ओर हटाया जाता है। अब $q = -4$ ।
3. $r = a \% b$ से -3 मिलता है। जाँचें: $(-4) * 4 = -16$ और $-19 - (-16) = -3$ । अब $r = -3$ ।
4. $z = 2 + q * 3$ में गुणा की प्राथमिकता अधिक है: $q * 3 = -12$, फिर $2 + (-12) = -10$ । अब $z = -10$ ।
5. q, r, z के क्रम में आउटपुट $-4 -3 -10$ है, फिर नई पंक्ति आती है। a और b में से किसी का मान नहीं बदला।

जोड़ी $-5, 1$ से फिर -19 बनता है, लेकिन उसमें नीचे की ओर राउंड किया गया है, इसलिए वह $C11$ की भागफल-शेषफल जोड़ी नहीं है। केवल कुल संख्या फिर बना लेना पर्याप्त नहीं है: भागफल शून्य की ओर काटने वाले नियम से भी मेल खाना चाहिए। जोड़ी $-4, 3$ का भागफल सही है, लेकिन उससे मूल संख्या नहीं बनती: $-16 + 3$ का मान -13 है।

पास के तीन मामलों में चिह्न का नियम जाँचें

C कोड 2

```
printf("%d %d\n", 19 / -4, 19 % -4);  
printf("%d %d\n", -19 / -4, -19 % -4);  
printf("%d %d\n", -20 / 4, -20 % 4);
```

1. $19 / -4$ से -4 मिलता है; $19 - (-4)*(-4) = 3$ । पहली पंक्ति $-4\ 3$ है।
2. $-19 / -4$ से 4 मिलता है; $-19 - 4*(-4) = -3$ । दूसरी पंक्ति $4\ -3$ है।
3. $-20 / 4$ से -5 मिलता है; $-20 - (-5)*4 = 0$ । तीसरी पंक्ति $-5\ 0$ है।

ऋणात्मक भाज्य के साथ भी शेषफल शून्य हो सकता है। “शेषफल का चिह्न भाज्य जैसा है” कहते समय शून्य वाला अपवाद ज़रूरी है। ये कॉल केवल स्थिर संख्याएँ पढ़ते हैं, इसलिए आर्ग्युमेंट के मान निकालने का क्रम परिणाम नहीं बदल सकता।

समान प्राथमिकता वाले ऑपरेटर्स का समूह

$*$, $/$ और $\%$ की प्राथमिकता समान है और उनका समूह बाईं ओर से बनता है। जैसे $18 / 4 * 3$ का समूह $(18 / 4) * 3$ है। पहले भागफल 4 है, फिर गुणनफल 12 है। समूह $18 / (4 * 3)$ नहीं है, जिसका परिणाम 1 होगा।

भिन्न वाला हिस्सा बीच की किसी भाग-गणना में ही हट सकता है। सामान्यतः सभी गणनाएँ भिन्नो के साथ करके केवल अंत में हिस्सा हटाना सही नहीं है: C में $(18 / 4) * 3$ का भागफल गुणा में इस्तेमाल होने से पहले ही अपना भिन्न वाला हिस्सा खो चुका है। कोष्ठक यह बदल सकते हैं कि कौन-सा भाग माँगा गया है। वे सामान्यतः चर बदलने वाले प्रभावों का बाएँ-से-दाएँ समय-क्रम तय नहीं करते। इस हिस्से के उदाहरणों में अंकगणितीय व्यंजक के भीतर कोई चर नहीं बदलता।

मान निकालना और चर बदलना अलग हैं

C कोड 3

```
int x = 4;
int next = x + 1;
printf("%d %d\n", x, next);
```

$x + 1$ से 5 मिलता है, जिससे next का शुरुआती मान रखा जाता है। इससे x में कुछ नहीं लिखा जाता। आउटपुट 4 5 है। इसके विपरीत, $x = x + 1$; परिणाम को वापस x में रखेगा। जब भी व्यंजक से चर बदल सकता हो, दो प्रश्न पूछें: “यह कौन-सा मान देता है?” और “यह स्टेटमेंट पूरा होने पर अवस्था क्या है?”

एक चर पर संयुक्त असाइनमेंट

यहाँ लिए गए साधारण int चर के लिए $\text{score} += 5$; का संख्यात्मक प्रभाव $\text{score} = \text{score} + 5$; जैसा है। इसी तरह $\text{score} -= 4$; में 4 घटाकर परिणाम रखा जाता है और $\text{score} /= 5$; में पूर्णांक भागफल रखा जाता है। इन ऑपरेटरों में गणना और असाइनमेंट दोनों हैं; $+=$ केवल + लिखने का दूसरा तरीका नहीं है।

C कोड 4

```
int score = 17;
score += 5;
score -= 4;
score /= 5;
printf("%d\n", score);
```

1. इनिशियलाइज़ेशन से $\text{score}=17$ ।
2. $\text{score} += 5$; के बाद: $\text{score}=22$ ।
3. $\text{score} -= 4$; के बाद: $\text{score}=18$ ।
4. $\text{score} /= 5$; के बाद: $\text{score}=3$, क्योंकि पूर्णांक $18 / 5$ का मान 3 है।
5. आउटपुट 3 है, फिर नई पंक्ति आती है।

संयुक्त रूप में बाईं ओर की परिणाम रखने वाली जगह का मूल्यांकन केवल एक बार होता है। इस पाठ में वह एक साधारण चर है, इसलिए जगह ढूँढ़ने की कोई अतिरिक्त गणना ट्रेस नहीं करनी पड़ती। विस्तार वाले रूप को अधिक जटिल व्यंजकों की कॉपी करने की अनुमति न समझें। संयुक्त असाइनमेंट पर भी वही गणितीय जाँचें लागू हैं: वास्तव में शून्य से भाग नहीं होना चाहिए और signed परिणाम सीमा से बाहर नहीं जाना चाहिए।

एक बढ़ाना: केवल नारा नहीं, दो अलग प्रश्न

इन्क्रिमेंट ऑपरेटर ++ किसी चर में एक जोड़ता है। उसकी जगह तय करती है कि व्यंजक कौन-सा मान देगा:

- $x++$ को postfix increment कहते हैं। व्यंजक x का पुराना मान देता है। साथ ही चर का मान एक बढ़ता है
- $++x$ को prefix increment कहते हैं। व्यंजक x का बढ़ा हुआ मान देता है

बढ़ने की मात्रा दोनों में अलग नहीं है: दोनों x को एक बढ़ाते हैं। पुराना मान 3 हो तो अलग $x++$; या $++x$; स्टेटमेंट पूरा होने के बाद $x=4$ होगा। फर्क तब महत्वपूर्ण है जब व्यंजक का मान किसी दूसरे चर में रखा जाए।

हर अपडेट को हम अपने अलग पूर्ण व्यंजक (full expression) में रखते हैं। `int before = x++;` का इनिशियलाइज़र एक पूर्ण व्यंजक है; अलग अपडेट स्टेटमेंट का व्यंजक भी ऐसा ही है। अगले पूर्ण व्यंजक के शुरू होने से पहले इसके आवश्यक बदलाव पूरे हो जाते हैं। Postfix का अर्थ “किसी बाद के स्टेटमेंट तक अपडेट टाल दो” नहीं है। कई अलग स्टेटमेंट को एक संयुक्त व्यंजक में बदलकर यह न मानें कि अर्थ वही रहेगा।

हल किया हुआ उदाहरण 2: मिलने वाला मान और अंतिम अवस्था

C कोड 5

```
int x = 3;
int before = x++;
int after = ++x;
printf("%d %d %d\n", before, after, x);
```

1. `int x = 3;` के बाद: `x=3`।
2. `int before = x++;` में postfix व्यंजक `before` को शुरू करने के लिए 3 देता है। इस घोषणा के अंत तक `x` बढ़कर 4 हो गया है। अवस्था: `x=4, before=3`।
3. `int after = ++x;` में `x`, 4 से बढ़कर 5 होता है और prefix व्यंजक `after` को शुरू करने के लिए 5 देता है। अवस्था: `x=5, before=3, after=5`।
4. आउटपुट 3 5 5 है, फिर नई पंक्ति आती है, क्योंकि आर्गुमेंट का क्रम `before, after, फिर x` है।

`before` में 3 ही रहता है; वह रखे गए मान की कॉपी है, `x` का दूसरा नाम नहीं। यदि 3 4 5 मिला, तो prefix व्यंजक को पुराना मान दे दिया गया। यदि 4 5 5 मिला, तो postfix व्यंजक को नया मान दे दिया गया। दोनों गलतियों से बचने के लिए व्यंजक से मिला मान और उसके बाद की अवस्था अलग-अलग लिखें।

सुरक्षा की सही सीमा

C साधारण + के ऑपरेंड को बाएँ से दाएँ मूल्यांकित करने का वादा नहीं करता। यदि कोई व्यंजक एक चर को बदलता भी है और उसी चर को दूसरी जगह आवश्यक क्रम के बिना पढ़ता भी है, तो उसमें अपरिभाषित व्यवहार (undefined behavior) हो सकता है। इसका अर्थ है कि C11 प्रोग्राम का कोई निर्धारित परिणाम नहीं देता। यह केवल दो सामान्य उत्तरों में से एक चुनने की बात नहीं है और क्रैश होना भी आवश्यक नहीं है।

हल किए हुए उदाहरण 2 की तरह अपडेट अलग-अलग पूर्ण व्यंजकों में रखें। अंकगणितीय समूह बदलने वाले कोष्ठक अपने-आप ऑपरेंड के बीच आवश्यक क्रम नहीं बनाते। असुरक्षित व्यंजक के लिए किसी कंपाइलर पर एक बार मिला परिणाम नियम नहीं बन जाता। अभ्यास प्रश्न 4 में संख्या का अनुमान लगाने के बजाय यही सीमा पहचाननी है।

अभ्यास: हल पढ़ने से पहले प्रयास करें

इन चार मौलिक प्रश्नों के अंक नहीं हैं। हर निश्चित उदाहरण में व्यंजकों के मान और अंतिम अवस्था लिखें। असुरक्षित उदाहरण के लिए संख्या की जगह उसके व्यवहार का वर्गीकरण करें।

अभ्यास प्रश्न 1: ऋणात्मक भाज्य

C11 के अनुसार भागफल और शेषफल निकालिए। आउटपुट लिखिए और मूल संख्या फिर बनाकर जाँच दिखाइए।

C कोड 6

```
int a=-23,b=6;
printf("%d %d\n",a/b,a%b);
```

अभ्यास प्रश्न 2: इन्क्रिमेंट के दो रूप

अंतिम n और p, q में रखे मान निकालिए। प्रिंट होने वाले आर्गुमेंट के क्रम का ध्यान रखें।

C कोड 7

```
int n=4;
int p=++n;
int q=n++;
printf("%d %d %d\n",n,p,q);
```

अभ्यास प्रश्न 3: कोष्ठक का प्रभाव

दोनों परिणाम निकालिए और समझाइए कि वे अलग क्यों हैं।

C कोड 8

```
int a=24/5*2;
int b=24/(5*2);
printf("%d %d\n",a,b);
```

अभ्यास प्रश्न 4: वर्गीकरण करें, चलाएँ नहीं

क्या इस अंश में C11, r का परिभाषित मान देता है? समझाइए कि यदि बाद में r प्रिंट किया जाए, तो क्या एक निश्चित संख्यात्मक आउटपुट माँगना सही होगा। संख्या चुनने के लिए इस अंश को न चलाएँ।

C कोड 9

```
int i=4;
int r=i++ + i;
```

अभ्यास के हल और गलत उत्तरों का कारण

अभ्यास प्रश्न 1 का हल

1. -23 और 6 का गणितीय भागफल -4 और -3 के बीच है; शून्य की ओर काटने पर $q = -3$ मिलता है।
 2. $q * b = (-3) * 6 = -18$ ।
 3. शेषफल $a - q * b = -23 - (-18) = -5$ है।
 4. जाँच: $(-3) * 6 + (-5) = -23$; शेषफल का परिमाण 5, 6 से छोटा है और उसका चिह्न ऋणात्मक भाज्य जैसा है।
 5. आउटपुट: -3 -5, फिर नई पंक्ति।
- 4 1 में C11 के शून्य की ओर काटने के बजाय floor division लिया गया है। -3 5 गलत तरीके से शेषफल को धनात्मक बनाता है और उससे -23 के बजाय -13 बनता है। -3 -1 का चिह्न संभव लगता है, लेकिन गणना की जाँच विफल होती है: $-18 - 1 = -19$ ।

अभ्यास प्रश्न 2 का हल

1. शुरुआत में $n=4$ ।
2. `int p=++n`; पहले n को 5 करता है और p को 5 देता है। अवस्था: $n=5$, $p=5$ ।
3. `int q=n++`; वर्तमान 5 को q में देता है और घोषणा के अंत तक n को 6 करता है। अवस्था: $n=6$, $p=5$, $q=5$ ।
4. प्रिंट का क्रम n , p , q है, इसलिए आउटपुट 6 5 5 है, फिर नई पंक्ति।

6 5 6 postfix को नया मान देता है। 5 5 5 भूल जाता है कि postfix फिर भी n बदलता है। 6 4 5 prefix को पुराना मान देता है। `printf` चलने से पहले दोनों अपडेट हो चुके हैं।

अभ्यास प्रश्न 3 का हल

1. पहली घोषणा में समान प्राथमिकता वाले `/` और `*` का समूह बाईं ओर से है: $a=(24/5)*2$ ।
2. पूर्णांक $24/5$ से 4 मिलता है; फिर $4*2=8$ । इसलिए $a=8$ ।
3. दूसरी घोषणा में कोष्ठक भाजक $5*2=10$ बनाते हैं। पूर्णांक $24/10$ से 2 मिलता है। इसलिए $b=2$ ।
4. आउटपुट: 8 2, फिर नई पंक्ति।

2 2 पहले व्यंजक में बिना कारण नए कोष्ठक जोड़ देता है। 9 2 बीच में भिन्न वाला 4.8 रखता है और गुणा के बाद ही हिस्सा हटाता है; C व्यंजक में पहले ही पूर्णांक भागफल 4 आ चुका था। 8 8 दूसरी घोषणा के कोष्ठक अनदेखे करता है। समझाने के लिए समूह और पूर्णांक भाग, दोनों ज़रूरी हैं।

अभ्यास प्रश्न 4 का हल

इस अंश में C11 के अनुसार अपरिभाषित व्यवहार है। $i++ + i$ में इन्क्रिमेंट द्वारा i को बदलना और दूसरे ऑपरेंड द्वारा i को पढ़ना एक-दूसरे के सापेक्ष unsequenced हैं, यानी इनके बीच आवश्यक क्रम निर्धारित नहीं है। साधारण $+$ इसे सुरक्षित बनाने वाला क्रम नहीं देता। नियम N1570 §6.5p2 में है।

r के लिए C11 द्वारा निर्धारित कोई संख्यात्मक मान नहीं है और सही ट्रेस-प्रश्न एक निश्चित आउटपुट नहीं माँग सकता। 8 और 9 कल्पना किए गए मूल्यांकन-समय पर आधारित लुभावने उत्तर हैं, मानक से परिभाषित उत्तर नहीं। “8 या 9 में से कोई एक” भी गलत है: अपरिभाषित व्यवहार इम्प्लीमेंटेशन को केवल इन दो परिणामों तक सीमित नहीं करता। “कंपाइल होना ज़रूर विफल होगा” और “क्रैश होना ही चाहिए” की भी गारंटी नहीं है। $i++$ के चारों ओर या पूरे जोड़ के चारों ओर कोष्ठक लगाने से क्रम की समस्या ठीक नहीं होती।

यदि इरादा “पुराना मान रखें, एक बढ़ाएँ, फिर पुराने और नए मान को जोड़ें” है, तो ये चरण अलग-अलग लिखें:

C कोड 10

```
int i = 4;
int old = i;
i = i + 1;
int r = old + i;
printf("%d %d\n", r, i);
```

1. शुरुआत में $i=4$ ।
2. $old=i$ से old में 4 रखा जाता है।
3. $i=i+1$ से $i=5$ होता है।
4. $r=old+i$ में $4+5=9$ लिया जाता है।
5. r और i के लिए आउटपुट 9 5 है, फिर नई पंक्ति।

यह स्पष्ट रूप से चुने गए इरादे वाला नया, परिभाषित प्रोग्राम है। इससे यह सिद्ध नहीं होता कि असुरक्षित अंश का “वास्तविक अर्थ 9 था”। उस असुरक्षित अंश का कोई परिभाषित अर्थ था ही नहीं जिसे बचाया जा सके।

आगे बढ़ने से पहले

अब आप गणना और चर में रखे जाने वाले अपडेट का अंतर समझते हैं, अलग पूर्ण व्यंजकों में prefix और postfix का ट्रेस कर सकते हैं तथा C11 का भागफल-शेषफल नियम लगा सकते हैं। अगले पाठ में शर्तें व्यंजकों के मान का उपयोग करके तय करेंगी कि कौन-से स्टेटमेंट चलेंगे।

स्रोत संबंधी नोट

कोड, गणनाएँ और गलतियों की व्याख्याएँ मौलिक हैं। [WG14 N1570 C11 समिति-ड्राफ्ट](#) में सटीक संदर्भ: पूर्णांक भाग/शेषफल §6.5.5p5-6; बाई ओर से समूह §6.5.5p1; संयुक्त असाइनमेंट §6.5.16.2p3; postfix §6.5.2.4p2; prefix §6.5.3.1p2; पूर्ण व्यंजक §6.8p4 और उनका क्रम §5.1.2.3p3; बिना आवश्यक क्रम के बदलाव/पढ़ना §6.5p2; सीमा से बाहर के परिणाम §6.5p5। कंपाइलर का आउटपुट परिभाषित उदाहरणों की जाँच में सहायक है, लेकिन वही भाषा की परिभाषा नहीं है।

उपमा और उसकी सीमाएँ

एक काउंटर और छोटी रसीद सोचें। Postfix में रसीद पर काउंटर का पुराना मान आता है और काउंटर एक बढ़ता है; prefix में काउंटर बढ़ता है और रसीद पर नया मान आता है। पूरा काम होने पर दोनों रूपों में काउंटर एक बढ़ चुका है, लेकिन रसीदों के मान अलग हो सकते हैं। अगला काम शुरू करने से पहले एक काम पूरा करें। यह उदाहरण अलग-अलग पूर्ण व्यंजक समझने में मदद करता है; एक ही C व्यंजक में कई अपडेट भर देने पर उनका कोई सुरक्षित क्रम तय नहीं करता।

त्वरित संदर्भ

- C11 में पूर्णांक / शून्य की ओर काटता है: $-19 / 4$ का मान -4 है
- गैर-शून्य b और सीमा में आने वाले भागफल के लिए: $(a / b) * b + a \% b$ का मान a है
- गैर-शून्य शेषफल का चिह्न भाज्य जैसा है; उसका परिमाण भाजक के परिमाण से छोटा है
- $*$ / $\%$ का समूह बाईं ओर से है: $24 / 5 * 2$ का अर्थ $(24 / 5) * 2$ है
- साधारण `int` चर पर $x += k$, $x -= k$ और $x /= k$ संबंधित गणना का परिणाम रखते हैं
- $x + 1$ मान निकालता है, x नहीं बदलता; $x = x + 1$ उसे रखता है
- $x++$ पुराना मान देता है; $++x$ नया मान देता है; दोनों x को एक बढ़ाते हैं
- अलग पूर्ण व्यंजकों में दिखाए गए अपडेट अगले व्यंजक से पहले पूरे होते हैं
- $i++ + i$ में अपरिभाषित व्यवहार है, कोई निर्धारित संख्यात्मक उत्तर नहीं
- कोष्ठक समूह तय करते हैं; वे सामान्यतः बदलावों का समय-क्रम तय नहीं करते
- हर बीच का परिणाम सीमा में रखें और वास्तव में उपयोग होने वाला हर भाजक गैर-शून्य रखें

[विषय सूची पर लौटें](#)

Conditions and selected branches

Which branch changes bonus?

The program below starts with score=12. Does it give the bonus for the first satisfied condition, or keep looking for a larger bonus? Predict both printed numbers before reading the trace.

C code 1

```
#include <stdio.h>

int main(void)
{
    int score = 12;
    int bonus = 0;
    if (score >= 10) {
        bonus = 2;
    } else if (score >= 12) {
        bonus = 5;
    } else {
        bonus = 1;
    }
    printf("%d %d\n", bonus, score + bonus);
    return 0;
}
```

You already know how an assignment changes the current state. A condition now decides whether an assignment is reached. This lesson uses C11 and small signed int values; every value read is initialized. All evaluated arithmetic stays within $-32767 \dots 32767$. Each displayed complete program can be compiled separately. The original practice question 3 is deliberately unsafe and must only be read, not run.

A condition has a value

A condition is an expression used to decide what happens next. With the integers in this lesson, zero means false and every nonzero value means true. Thus 0 is false, while 5 and -3 are both true. A negative number is not automatically false. Testing a variable does not change its stored value: if x contains -3 , testing x does not replace x with 1.

A comparison computes an int result: 1 when its relation holds and 0 otherwise. Use $<$ and $>$ for strict comparisons, $<=$ and $>=$ to include equality, $==$ to test equality, and $!=$ to test inequality. For score=12, score $>=$ 10 gives 1, score $==$ 10 gives 0, and score $!=$ 10 gives 1. These tests read score; none assigns a new score.

Keep assignment and equality separate. $x = 5$ stores 5 in x. $x == 5$ asks whether the current x is 5. An assignment is permitted in a C condition, but it still assigns: if the condition is $x = 5$, x becomes 5 and the condition is true. That is not an equality test. Write $x == 5$ when equality is intended, and keep ordinary updates in their own statements as in the earlier lessons.

Source rule: comparison results and integer conditions are specified in [C11 draft N1570](#), §§6.5.8p6, 6.5.9p3 and 6.8.4.1p2; assignment value is covered by §6.5.16p3.

Choose one route with if, else if and else

Read the program in worked example 1 in this order. The parentheses contain each condition. The braces contain the statements belonging to that branch.

1. Test `score >= 10`. If it is true, run that first braced body and skip the entire else part.
2. Only if the first test is false, reach the second if inside the else part. Test `score >= 12` then.
3. Only if both reached tests are false, run the final else body. A final else has no condition of its own.
4. Continue with `printf` after the whole chain.

An else-if chain is one connected choice. It does not rank conditions or choose the largest assigned value. This particular ordering makes the `score >= 12` branch unreachable: any score that reaches it has already failed `score >= 10` and therefore is below 10. If the intended rule were to prefer the higher threshold, test `score >= 12` first. We trace the program actually written before deciding whether its rule is the intended one.

Worked example 1: first match wins

1. After the declarations: `score=12, bonus=0`.
2. First test: `12 >= 10` gives 1, so enter the first body.
3. Execute `bonus=2`. State: `score=12, bonus=2`.
4. Skip the entire else part. The expression `score >= 12` is not evaluated on this run, and neither later assignment is executed.
5. `printf` reads `bonus=2` and computes `score+bonus` as `12+2=14`. Output: `2 14`, followed by a newline.

The output is not `5 17`: a later mathematically true condition is irrelevant when execution never reaches its test. It is not `1 13` either: else is the alternative to a false condition, not a statement that always runs last.

By contrast, two separate if statements are two decisions. After the first one finishes, execution reaches the second one and evaluates it using the current state. Both bodies may run. If the first body changes a variable used by the second test, use the new value there. Practice question 1 below makes this distinction concrete without changing its tested variable.

Build a condition with !, && and ||

Logical operators also produce the int value 0 or 1. They do not return an arbitrary nonzero operand. For example, the logical expression `(-3) && 5` is true and produces 1, not 5 or -3.

- `!` means logical NOT. `!0` is 1, while `!5` and `!(-3)` are 0. It reverses truth, not the sign of a number.
- With left `&&` right, evaluate the left operand first. If it is zero, the whole result is 0 and do not evaluate the right operand. If left is nonzero, evaluate right; the result is 1 exactly when right is also nonzero.
- With left `||` right, evaluate the left operand first. If it is nonzero, the whole result is 1 and do not evaluate the right operand. If left is zero, evaluate right; the result is 1 exactly when right is nonzero.

The skipped work is called short-circuit evaluation. “Skipped” is stronger than “calculated but ignored”: the expression is not evaluated at all. For `&&` and `||`, the first operand is sequenced before the second when the second is needed. This sequencing rule belongs to these logical operators; it is not a rule that every C expression evaluates from left to right. See [N1570](#), §6.5.3.3p5 and §6.5.13p3-4, 6.5.14p3-4.

Worked example 2: prove which divisions are skipped

C code 2

```
#include <stdio.h>

int main(void)
{
    int divisor = 0;
    int value = 15;
    int all = (divisor != 0) && (value / divisor > 2);
    int any = (value > 10) || (value / divisor > 2);
    printf("%d %d\n", all, any);
    return 0;
}
```

1. Initial state: divisor=0, value=15.
2. To initialize all, first evaluate divisor!=0. The comparison 0!=0 gives 0.
3. The left operand of `&&` is false, so value/divisor>2 is skipped. No division occurs. Store all=0.
4. To initialize any, first evaluate value>10. The comparison 15>10 gives 1.
5. The left operand of `||` is true, so value/divisor>2 is skipped here too. Store any=1.
6. printf prints 0 1, followed by a newline. divisor and value still contain 0 and 15.

The zero stored in divisor does not itself make this run undefined: no evaluated operation divides by it. Reversing either pair of operands would make the division occur first for these initial values and would lose that protection. The `&&` guard divisor!=0 also protects the division when the starting divisor changes: if zero it skips, and if nonzero the quotient with numerator 15 is representable. The `||` expression shown is safe for value=15 because its left operand is true; it is not a general zero-divisor guard. For value=3 and divisor=0, that particular `||` left operand would be false and its right operand would divide by zero. A true result in one run is not a safety proof for all inputs.

Test an inclusive interval with two comparisons

To ask whether *m* is from 2 through 5 including both ends, require both facts: *m*≥2 and *m*≤5. Write `(m>=2)&&(m<=5)`. At *m*=2 both comparisons hold; at *m*=5 both hold; at *m*=1 the first fails, and at *m*=6 the second fails. Use parentheses to make the two comparisons easy to read.

Do not import the mathematical shorthand $2 \leq m \leq 5$ into C. C groups it as `(2<=m)<=5`. With *m*=8, the first comparison gives 1 and then `1<=5` gives 1, incorrectly accepting 8. In fact either first result, 0 or 1, is `<=5`. Parenthesizing the whole chained expression does not repair its meaning. The correct two-comparison expression must join the conditions with `&&`. Also, `(m>2)&&(m<5)` is a different interval because it excludes 2 and 5.

Practice: attempt all four before opening the solutions

The questions use new values or a different control-flow demand from the worked traces. They are unscored practice.

Practice question 1: two independent decisions

Find the final flags value. State whether each body runs and explain why this is different from an else-if chain.

C code 3

```
#include <stdio.h>

int main(void)
{
    int m=9, flags=0;
    if(m%3==0){flags+=1;}
    if(m>5){flags+=2;}
    printf("%d\n", flags);
    return 0;
}
```

Practice question 2: negation followed by AND

Find z. Name the value of !a, say whether the right operand of && is evaluated, and give the logical result.

C code 4

```
#include <stdio.h>

int main(void)
{
    int a=0, b=5;
    int z=(!a)&&(b>2);
    printf("%d\n", z);
    return 0;
}
```

Practice question 3: repair the guard order

For d=0 and n=14, is the condition below safe? Rewrite only the order of its two operands so that it safely guards division. Do not execute the original to guess an output; this is a behavior-classification and repair question.

C code 5

```
#include <stdio.h>

int main(void)
{
    int d=0, n=14;
    int result=(n/d>1)&&(d!=0);
    return 0;
}
```

Practice question 4: keep the endpoints

For a general int m, which expression tests the inclusive interval [2,5]?

- A: $(m \geq 2) \&\& (m \leq 5)$
- B: $(m > 2) \&\& (m < 5)$

Choose A or B, then predict both printed results in the m=5 check.

C code 6

```
#include <stdio.h>

int main(void)
{
    int m=5;
    printf("%d %d\n", (m>=2)&&(m<=5), (m>2)&&(m<5));
    return 0;
}
```

Practice solutions and wrong-answer feedback

Practice question 1 solution

1. Start with m=9, flags=0.
2. Compute m%3 as 9%3=0; 0==0 gives 1. The first body adds 1, leaving flags=1.
3. Reach the second independent if. The test 9>5 gives 1, so its body adds 2 to the current flags=1.
4. Final flags=3; output is 3 and a newline. Both tests were evaluated and both bodies ran.

A result of 1 wrongly skips the second if as though it were an else-if. A result of 2 keeps only the second change, but += adds to the current flags instead of replacing it. A result of 0 overlooks that both comparisons are true. m remains 9 throughout.

Practice question 2 solution

1. a=0 and b=5.
2. !a is !0, which gives 1.
3. Since the left operand of && is nonzero, evaluate b>2. The comparison 5>2 gives 1.
4. Both operands are true, so z=1 and the output is 1 followed by a newline.

The answer 0 confuses !0 with 0. The answer 5 confuses a logical result with b's stored value: && yields 0 or 1. Neither a nor b changes, and the right operand is not skipped on this run.

Practice question 3 solution

The original has undefined behavior. Its left operand evaluates n/d before there is any check of d ; at these values that is $14/0$. Under C11 there is no prescribed output, and a crash is not a guaranteed outcome. The fact that the right comparison would be false cannot undo the already evaluated invalid division.

The repaired condition is $(d!=0)\&\&(n/d>1)$. A complete safe check is:

C code 7

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

```
int main(void)
{
    int d=0,n=14;
    int result=(d!=0)&\&(n/d>1);
    printf("%d\n",result);
    return 0;
}
```

1. $d!=0$ evaluates as $0!=0$ and yields 0.
2. $\&\&$ skips $n/d>1$, so no division occurs.
3. result becomes 0; this repaired program prints 0 and a newline.

Merely adding parentheses around the original condition does not change operand order. Replacing $\&\&$ with $||$ does not repair the original leftmost division. “The original returns 0 because d is zero” is also wrong: the guard is too late. The unsafe classification follows from [N1570](#), §6.5.5p5; the repair uses §6.5.13p4.

Practice question 4 solution

A is correct. At $m=5$, the first expression checks $5\geq 2$ and $5\leq 5$, both true, and produces 1. The second checks $5>2$ (true), then $5<5$ (false), and produces 0. The printed pair is 1 0, followed by a newline.

B would also reject $m=2$, so it cannot describe an interval that includes both endpoints. The pair 1 1 misses the strict upper comparison; the pair 0 0 wrongly treats equality as disallowed by $\leq$. For a lower boundary check, $m=1$ makes A's first comparison false, so its right comparison is skipped; that still correctly gives 0.

Carry the state forward

Before moving on, explain worked example 1's skipped second test and worked example 2's skipped divisions without evaluating them. A branch chooses whether to run a body once. The next lesson repeatedly tests a condition so that a body can run zero, one or several times, with a new state on each pass.

Analogy and its limits

Imagine a clerk following a written routing card. "If the parcel weighs at least 10 units, place a blue sticker; otherwise, check the next rule." Once the first route is taken, the clerk does not also follow the alternative route. Two separate cards saying "if" would instead require two decisions, so both actions could happen. That distinction mirrors the first-match chain and the independent if statements.

A guard is like "Is there a nonzero number of boxes? If so, calculate items per box." Asking whether the boxes exist must come before division. The comparison does not secretly calculate the quotient. This picture helps with `&&`, but the actual C rule is exact: a zero left operand skips the right operand. For `||`, a nonzero left operand skips the right operand. The analogy does not permit reordering the checks.

Quick reference

- An int condition is false at 0 and true at every nonzero value, including negative values.
- `<`, `>`, `<=`, `>=`, `==` and `!=` produce int 0 or 1. `x=5` assigns; `x==5` compares.
- `!0` is 1; `!` of a nonzero value is 0.
- left `&&` right: evaluate left first; skip right when left is zero.
- left `||` right: evaluate left first; skip right when left is nonzero.
- `&&` and `||` produce 0 or 1. Their sequencing is not a general left-to-right rule for all operators.
- In an if/else-if/else chain, take the first reached true branch and skip its alternatives. Separate if statements are separate decisions.
- Inclusive `[2,5]`: `(m>=2)&&(m<=5)`. The chained form `2<=m<=5` does not express that interval in C.
- Before dividing by `d`, put a protecting `d!=0` test on the left of `&&`. A skipped division is not evaluated; an already evaluated division by zero cannot be repaired by a later test.
- To trace: record the current state, each reached condition, the selected body, the skipped operands or branches, and the resulting state.

[Back to contents](#)

शर्तें और चुनी गई शाखाएँ

bonus को कौन-सी शाखा बदलेगी?

नीचे के प्रोग्राम में शुरुआत में score=12 है। क्या वह पहली सही शर्त का बोनस देगा, या उसके बाद और बड़ा बोनस ढूँढ़ता रहेगा? ट्रेस पढ़ने से पहले छपने वाली दोनों संख्याओं का अनुमान लगाइए।

C कोड 1

```
#include <stdio.h>

int main(void)
{
    int score = 12;
    int bonus = 0;
    if (score >= 10) {
        bonus = 2;
    } else if (score >= 12) {
        bonus = 5;
    } else {
        bonus = 1;
    }
    printf("%d %d\n", bonus, score + bonus);
    return 0;
}
```

पिछले पाठ में आपने देखा कि असाइनमेंट वर्तमान अवस्था (state) कैसे बदलता है। अब शर्त (condition) तय करेगी कि किसी असाइनमेंट तक पहुँचना है या नहीं। यहाँ C11 और छोटे signed int मानों का उपयोग है। पढ़े जाने से पहले हर चर को मान दिया गया है, और मूल्यांकित गणनाओं के सभी मान -32767...32767 में रहते हैं। प्रत्येक पूरा प्रोग्राम अलग संकलित किया जा सकता है। मूल अभ्यास प्रश्न 3 जानबूझकर असुरक्षित है; उसे केवल पढ़ना है, चलाना नहीं।

शर्त का भी एक मान होता है

शर्त एक व्यंजक (expression) है जिससे अगला कदम तय होता है। इस पाठ के पूर्णाकों में शून्य का अर्थ गलत और हर गैर-शून्य मान का अर्थ सही है। इसलिए 0 गलत है, लेकिन 5 और -3 दोनों सही हैं। ऋणात्मक संख्या अपने आप गलत नहीं होती। किसी चर को शर्त की तरह जाँचने से उसका संग्रहित मान नहीं बदलता: x में -3 हो तो x की जाँच उसे 1 में नहीं बदलती।

तुलना का परिणाम int होता है: संबंध सही हो तो 1, गलत हो तो 0। सख्त तुलना के लिए < और >, बराबरी को भी शामिल करने के लिए <= और >=, बराबरी जाँचने के लिए ==, और बराबर न होने की जाँच के लिए != लिखते हैं। score=12 पर score>=10 का मान 1, score==10 का मान 0, और score!=10 का मान 1 है। ये जाँच score को पढ़ती हैं; कोई नया score नहीं रखती।

असाइनमेंट और बराबरी को अलग रखें। x = 5, x में 5 रखता है। x == 5 पूछता है कि x का वर्तमान मान 5 है या नहीं। C की शर्त में असाइनमेंट लिखना वैध है, पर वह तब भी असाइनमेंट ही करता है: शर्त x = 5 हो तो x में 5 रखा जाता है और शर्त सही होती है। यह बराबरी की जाँच नहीं है। बराबरी पूछनी हो तो x == 5 लिखें, और सामान्य अपडेट को पहले के पाठों की तरह अलग कथन में रखें।

स्रोत-नियम: तुलना के परिणाम और पूर्णाक शर्तों के लिए [C11 का N1570 मसौदा](#), §6.5.8p6, 6.5.9p3 और 6.8.4.1p2 देखें। असाइनमेंट के मान का नियम §6.5.16p3 में है।

if, else if और else से एक रास्ता चुनना

उदाहरण 1 को इस क्रम में पढ़ें। गोल कोष्ठकों में शर्त है। बड़े कोष्ठकों में उस शाखा के कथन हैं।

1. `score >= 10` जाँचें। सही हो तो पहले बड़े कोष्ठकों वाली body चलाएँ और पूरा else भाग छोड़ दें।
2. पहली जाँच गलत होने पर ही else के अंदर वाले दूसरे if तक पहुँचें। तभी `score >= 12` जाँचें।
3. दोनों तक पहुँची हुई जाँच गलत हों तो अंतिम else की body चलाएँ। अंतिम else की अपनी कोई शर्त नहीं होती।
4. पूरी श्रृंखला के बाद printf पर आगे बढ़ें।

else-if की श्रृंखला आपस में जुड़ा एक चुनाव है। वह शर्तों की रैंकिंग नहीं करती और सबसे बड़ा असाइन होने वाला मान नहीं चुनती। इस विशेष क्रम में `score >= 12` की शाखा कभी नहीं पहुँच सकती: वहाँ तक पहुँचने वाला score पहले ही `score >= 10` में असफल हुआ है, इसलिए वह 10 से छोटा होगा। यदि नियम में बड़े threshold को प्राथमिकता देनी हो तो `score >= 12` को पहले जाँचें। पहले वास्तव में लिखे प्रोग्राम का ट्रेस करें; उसके बाद तय करें कि लिखा नियम इच्छित नियम है या नहीं।

हल किया हुआ उदाहरण 1: पहली सही शाखा चुनी जाती है

1. घोषणाओं के बाद: `score=12, bonus=0`।
2. पहली जाँच: `12 >= 10` से 1 मिलता है, इसलिए पहली body में प्रवेश होता है।
3. `bonus=2` चलता है। अवस्था: `score=12, bonus=2`।
4. पूरा else भाग छूट जाता है। इस बार `score >= 12` का मूल्यांकन नहीं होता और बाद वाले दोनों असाइनमेंट भी नहीं चलते।
5. printf, `bonus=2` पढ़ता है और `score+bonus` को `12+2=14` निकालता है। आउटपुट: 2 14, फिर नई पंक्ति।

आउटपुट 5 17 नहीं है: बाद की शर्त गणित की दृष्टि से सही हो, तब भी उस तक पहुँचे बिना उसकी जाँच नहीं होती। 1 13 भी नहीं है: else गलत शर्त का वैकल्पिक रास्ता है, अंत में हमेशा चलने वाला कथन नहीं।

इसके विपरीत, दो अलग if कथन दो अलग निर्णय हैं। पहला पूरा होने के बाद दूसरे तक पहुँचकर वर्तमान अवस्था से उसकी जाँच की जाती है। दोनों की body चल सकती है। पहली body उस चर को बदल दे जिसे दूसरी शर्त पढ़ती है, तो दूसरी जाँच में नया मान लें। नीचे अभ्यास प्रश्न 1 में जाँचे जा रहे चर को बदले बिना यही अंतर दिखता है।

!, && और || से शर्त बनाना

तार्किक ऑपरेटरों का परिणाम भी int मान 0 या 1 होता है। वे किसी भी गैर-शून्य operand को सीधे वापस नहीं देते।

उदाहरण के लिए `(-3) && 5` तार्किक रूप से सही है और 1 देता है, 5 या -3 नहीं।

- ! तार्किक NOT है। !0 से 1, जबकि !5 और !(-3) से 0 मिलता है। यह सत्यता बदलता है, संख्या का चिह्न नहीं।
- left && right में पहले बायाँ operand जाँचें। वह शून्य हो तो पूरा परिणाम 0 है और दाएँ operand का मूल्यांकन न करें। बायाँ गैर-शून्य हो तभी दायाँ जाँचें; दायाँ भी गैर-शून्य हो तो परिणाम 1 है, अन्यथा 0।
- left || right में पहले बायाँ operand जाँचें। वह गैर-शून्य हो तो पूरा परिणाम 1 है और दाएँ operand का मूल्यांकन न करें। बायाँ शून्य हो तभी दायाँ जाँचें; दायाँ गैर-शून्य हो तो परिणाम 1 है, अन्यथा 0।

इस तरह अनावश्यक भाग छोड़ने को short-circuit evaluation कहते हैं। “छोड़ दिया” का अर्थ “गणना करके परिणाम अनदेखा किया” नहीं है: उस व्यंजक की गणना होती ही नहीं। && और || में, दाएँ operand की जरूरत हो तो भी बाएँ operand का मूल्यांकन उसके पहले क्रमित होता है। यह नियम इन तार्किक ऑपरेटर्स का है; इससे यह निष्कर्ष न निकालें कि C का हर व्यंजक बाएँ से दाएँ मूल्यांकित होता है। [N1570](#), §6.5.3.3p5 और §6.5.13p3-4, 6.5.14p3-4 देखें।

हल किया हुआ उदाहरण 2: कौन-से भाग छूटे, यह दिखाएँ

C कोड 2

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

```
int main(void)
{
    int divisor = 0;
    int value = 15;
    int all = (divisor != 0) && (value / divisor > 2);
    int any = (value > 10) || (value / divisor > 2);
    printf("%d %d\n", all, any);
    return 0;
}
```

1. शुरुआती अवस्था: divisor=0, value=15।
2. all को मान देने के लिए पहले divisor!=0 निकालें। 0!=0 की तुलना से 0 मिलता है।
3. && का बायाँ operand गलत है, इसलिए value/divisor>2 छूट जाता है। कोई भाग नहीं होता। all=0 रखें।
4. any को मान देने के लिए पहले value>10 निकालें। 15>10 की तुलना से 1 मिलता है।
5. || का बायाँ operand सही है, इसलिए यहाँ भी value/divisor>2 छूट जाता है। any=1 रखें।
6. printf से 0 1 और फिर नई पंक्ति छपती है। divisor और value के मान अभी भी 0 और 15 हैं।

divisor में शून्य होने मात्र से इस बार undefined behavior नहीं होता: वास्तव में मूल्यांकित किसी गणना में उससे भाग नहीं हुआ। इनमें से किसी भी operand जोड़ी का क्रम उलटने पर इन शुरुआती मानों के लिए भाग पहले होगा और सुरक्षा खत्म हो जाएगी। && वाला divisor!=0 गार्ड शुरुआती divisor बदलने पर भी भाग की रक्षा करता है: शून्य हो तो भाग छूटता है; गैर-शून्य हो तो अंश 15 वाले इस भाग का परिणाम int में निरूपित हो सकता है। दिखाया गया || व्यंजक value=15 पर सुरक्षित है क्योंकि उसका बायाँ operand सही है; यह शून्य divisor से बचाने वाला सामान्य गार्ड नहीं है। value=3 और divisor=0 पर इसी || का बायाँ operand गलत होगा और दाएँ में शून्य से भाग होगा। एक बार सही परिणाम आना सभी इनपुट के लिए सुरक्षा का प्रमाण नहीं है।

दोनों सिरों सहित अंतराल के लिए दो तुलना करें

m, 2 से 5 तक है और दोनों सिरों शामिल हैं, यह पूछने के लिए दोनों बातें चाहिए: $m \geq 2$ और $m \leq 5$ । लिखें $(m \geq 2) \&\& (m \leq 5)$ । $m=2$ पर दोनों तुलना सही हैं; $m=5$ पर भी दोनों सही हैं; $m=1$ पर पहली गलत है और $m=6$ पर दूसरी गलत है। दोनों तुलना साफ दिखाने के लिए कोष्ठक लगाएँ।

गणित के छोटे रूप $2 \leq m \leq 5$ को C में वैसे ही न लिखें। C उसे $(2 \leq m) \leq 5$ की तरह समूहित करता है। $m=8$ पर पहली तुलना से 1 मिलता है, फिर $1 \leq 5$ से 1 मिलता है, इसलिए 8 गलत ढंग से स्वीकार हो जाता है। वास्तव में पहली तुलना का 0 या 1, दोनों ही ≤ 5 हैं। पूरे chained व्यंजक के बाहर कोष्ठक लगाने से उसका अर्थ नहीं सुधरता। सही रूप में दोनों तुलना को && से जोड़ना जरूरी है। $(m > 2) \&\& (m < 5)$ अलग अंतराल है, क्योंकि वह 2 और 5 को छोड़ता है।

अभ्यास: हल देखने से पहले चारों प्रश्न करें

इन प्रश्नों में हल किए ट्रेस से अलग मान या अलग नियंत्रण-प्रवाह का अभ्यास है। इनके अंक नहीं हैं।

अभ्यास प्रश्न 1: दो स्वतंत्र निर्णय

flags का अंतिम मान निकालिए। बताइए कि हर body चलती है या नहीं, और यह else-if की श्रृंखला से अलग क्यों है।

C कोड 3

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

```
int main(void)
{
    int m=9, flags=0;
    if(m%3==0){flags+=1;}
    if(m>5){flags+=2;}
    printf("%d\n", flags);
    return 0;
}
```

अभ्यास प्रश्न 2: NOT के बाद AND

z का मान निकालिए। !a का मान बताइए, && का दायाँ operand जाँचा जाता है या नहीं बताइए, और अंतिम तार्किक परिणाम दीजिए।

C कोड 4

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

```
int main(void)
{
    int a=0, b=5;
    int z=(!a)&&(b>2);
    printf("%d\n", z);
    return 0;
}
```

अभ्यास प्रश्न 3: गार्ड का क्रम सुधारें

d=0 और n=14 के लिए नीचे की शर्त सुरक्षित है या नहीं? उसके केवल दोनों operands का क्रम बदलकर भाग को सुरक्षित करने वाला गार्ड बनाइए। आउटपुट का अनुमान लगाने के लिए मूल प्रोग्राम न चलाएँ; यहाँ व्यवहार का वर्गीकरण और सुधार करना है।

C कोड 5

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

```
int main(void)
{
    int d=0, n=14;
    int result=(n/d>1)&&(d!=0);
    return 0;
}
```

अभ्यास प्रश्न 4: सिरों को शामिल रखें

सामान्य int m के लिए कौन-सा व्यंजक दोनों सिरों सहित [2,5] अंतराल जाँचता है?

- A: (m>=2)&&(m<=5)
- B: (m>2)&&(m<5)

A या B चुनिए, फिर m=5 वाली जाँच में छपने वाले दोनों परिणाम बताइए।

C कोड 6

```
#include <stdio.h>

int main(void)
{
    int m=5;
    printf("%d %d\n", (m>=2)&&(m<=5), (m>2)&&(m<5));
    return 0;
}
```

अभ्यास के पूरे हल और गलत उत्तरों का कारण

अभ्यास प्रश्न 1 का हल

1. शुरुआत में m=9, flags=0।
2. m%3 से 9%3=0 मिलता है; 0==0 से 1 मिलता है। पहली body, 1 जोड़ती है और flags=1 हो जाता है।
3. दूसरे स्वतंत्र if तक पहुँचते हैं। 9>5 से 1 मिलता है, इसलिए उसकी body वर्तमान flags=1 में 2 जोड़ती है।
4. अंत में flags=3 है; आउटपुट 3 और नई पंक्ति है। दोनों जाँच हुई और दोनों body चलीं।

1 उत्तर देने पर दूसरे if को else-if की तरह गलत ढंग से छोड़ दिया गया है। 2 उत्तर केवल दूसरा बदलाव रखता है, जबकि += वर्तमान flags में जोड़ता है, उसे 2 से बदलता नहीं। 0 उत्तर में दोनों तुलना सही होने की बात छूट गई है। पूरे समय m का मान 9 रहता है।

अभ्यास प्रश्न 2 का हल

1. a=0 और b=5 हैं।
2. !a अर्थात् !0 से 1 मिलता है।
3. && का बायाँ operand गैर-शून्य है, इसलिए b>2 जाँचा जाता है। 5>2 से 1 मिलता है।
4. दोनों operands सही हैं, इसलिए z=1 है और आउटपुट 1, फिर नई पंक्ति है।

0 उत्तर में !0 को गलती से 0 माना गया है। 5 उत्तर में तार्किक परिणाम और b के संग्रहित मान को एक माना गया है: && केवल 0 या 1 देता है। a और b में कोई बदलाव नहीं होता, और इस बार दायाँ operand छोड़ा नहीं जाता।

अभ्यास प्रश्न 3 का हल

मूल में undefined behavior है। उसका बायाँ operand d की जाँच से पहले n/d निकालता है; यहाँ वह 14/0 है। C11 इसके लिए कोई निश्चित आउटपुट नहीं बताता, और crash होना भी निश्चित परिणाम नहीं है। दाईं तुलना बाद में गलत होती, इससे पहले हो चुका अमान्य भाग रद्द नहीं हो जाता।

सुधरी हुई शर्त $(d \neq 0) \&\& (n/d > 1)$ है। इसकी पूरी सुरक्षित जाँच:

C कोड 7

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

```
int main(void)
{
    int d=0,n=14;
    int result=(d!=0)&&(n/d>1);
    printf("%d\n",result);
    return 0;
}
```

1. $d \neq 0$ अर्थात् $0 \neq 0$ से 0 मिलता है।
2. $\&\&$, $n/d > 1$ को छोड़ देता है; इसलिए कोई भाग नहीं होता।
3. result में 0 रखा जाता है; सुधरा प्रोग्राम 0 और नई पंक्ति छापता है।

मूल शर्त के चारों ओर केवल कोष्ठक लगाने से operands का क्रम नहीं बदलता। $\&\&$ को $||$ में बदल देने से भी सबसे बाएँ वाला भाग नहीं सुधरता। “d शून्य है इसलिए मूल 0 लौटाता है” भी गलत है: गार्ड बहुत बाद में आता है। असुरक्षित व्यवहार का नियम [N1570](#), §6.5.5p5 में है; सुधार §6.5.13p4 के नियम पर आधारित है।

अभ्यास प्रश्न 4 का हल

A सही है। $m=5$ पर पहला व्यंजक $5 \geq 2$ और $5 \leq 5$ जाँचता है; दोनों सही हैं, इसलिए 1 मिलता है। दूसरा $5 > 2$ को सही और फिर $5 < 5$ को गलत पाता है, इसलिए 0 मिलता है। छपने वाली जोड़ी 1 0, फिर नई पंक्ति है।

B, $m=2$ को भी अस्वीकार करेगा, इसलिए वह दोनों सिरों को शामिल करने वाला अंतराल नहीं है। 1 1 उत्तर में सख्त ऊपरी तुलना छूट गई है; 0 0 उत्तर में $\leq$ को बराबरी अस्वीकार करने वाला मान लिया गया है। निचले सिरे के बाहर की जाँच में $m=1$ पर A की पहली तुलना गलत है, इसलिए उसकी दाईं तुलना छूटती है; फिर भी सही परिणाम 0 मिलता है।

अगली जाँच में वर्तमान अवस्था लें

आगे बढ़ने से पहले उदाहरण 1 की छूटी दूसरी जाँच और उदाहरण 2 के छूटे भाग को बिना मूल्यांकित किए समझाइए। शाखा तय करती है कि body एक बार चलनी है या नहीं। अगले पाठ में शर्त बार-बार जाँची जाएगी, ताकि body शून्य, एक या कई बार चले और हर बार नई अवस्था का उपयोग करे।

उपमा और उसकी सीमाएँ

मान लें कोई कर्मचारी लिखे हुए निर्देश से पार्सल पर स्टिकर लगाता है: “वजन कम-से-कम 10 इकाई हो तो नीला स्टिकर लगाएँ; नहीं तो अगला नियम जाँचें।” पहला रास्ता लेने के बाद वह वैकल्पिक रास्ते पर भी नहीं चलता। इसके बदले दो अलग पर्वियों पर अलग-अलग “यदि” निर्देश हों तो दो निर्णय करने होंगे और दोनों काम हो सकते हैं। यही पहली-सही-शाखा वाली श्रृंखला और स्वतंत्र if कथनों का अंतर है।

गार्ड ऐसा है: “क्या डिब्बों की संख्या गैर-शून्य है? तभी प्रति डिब्बा वस्तुओं की संख्या निकालें।” भाग करने से पहले डिब्बों की संख्या जाँचना जरूरी है। तुलना छिपकर भागफल नहीं निकालती। यह चित्र && समझने में मदद करता है, लेकिन C का सटीक नियम यही है कि बायाँ operand शून्य हो तो दायाँ छूटता है। || में बायाँ गैर-शून्य हो तो दायाँ छूटता है। इस उपमा से जाँचों का क्रम बदलने की अनुमति नहीं मिलती।

त्वरित संदर्भ

- int शर्त 0 पर गलत और हर गैर-शून्य मान पर सही है; ऋणात्मक मान भी सही है।
- <, >, <=, >=, == और != से int मान 0 या 1 मिलता है। x=5 असाइन करता है; x==5 तुलना करता है।
- !0 से 1; गैर-शून्य मान पर ! से 0 मिलता है।
- left && right: बायाँ पहले जाँचें; वह शून्य हो तो दायाँ छोड़ें।
- left || right: बायाँ पहले जाँचें; वह गैर-शून्य हो तो दायाँ छोड़ें।
- && और ||, 0 या 1 देते हैं। उनका क्रम सभी ऑपरेटरों के लिए सामान्य बाएँ-से-दाएँ नियम नहीं है।
- if/else-if/else श्रृंखला में पहली पहुँची हुई सही शाखा लें और उसके विकल्प छोड़ें। अलग if कथन अलग निर्णय हैं।
- दोनों सिरों सहित [2,5]: (m>=2)&&(m<=5)। 2<=m<=5 का chained रूप C में यह अंतराल नहीं बताता।
- d से भाग करने से पहले सुरक्षा देने वाला d!=0, && के बाएँ रखें। छोड़ा गया भाग मूल्यांकित नहीं होता; पहले हो चुके शून्य से भाग को बाद की जाँच नहीं सुधारती।
- ट्रेस में लिखें: वर्तमान अवस्था, हर पहुँची शर्त, चुनी body, छूटे operands या शाखाएँ, और उसके बाद की अवस्था।

विषय सूची पर लौटें

Loops, updates and stopping

What happens after the last square is added?

This program adds squares. Predict both sum and i at printf. In particular, decide whether i is still 4 when the loop finishes.

C code 1

```
#include <stdio.h>

int main(void)
{
    int sum = 0;
    int i;
    for (i = 1; i <= 4; i++) {
        sum += i * i;
    }
    printf("%d %d\n", sum, i);
    return 0;
}
```

A loop repeats a body while its condition allows another pass. Each completed pass is an iteration. You already know how to read an assignment and interpret a zero/nonzero condition. Keep those same rules: each new test and update reads the current state, not the starting state. This lesson uses C11, initialized-before-read int values and intermediate results within $-32767 \dots 32767$. All displayed complete programs can be compiled separately.

Read the four stages of for

In `for (i=1; i<=4; i++)`, the first part sets the starting state, the middle part is a condition, and the last part is an update. The two semicolons separate those parts; they do not mean all three happen before the body.

1. Initialization: execute `i=1` once when the loop is reached.
2. Test: evaluate `i<=4` using the current `i`. If false, leave the loop immediately.
3. Body: if the test was true, run the braced statements in order.
4. Update: after this body completes, execute `i++`, then return to step 2, not step 1.

Here `i` is declared before the loop so it is still available to `printf` afterwards. Its declaration `int i;` does not give it a usable starting value, but the `for` initialization assigns 1 before any read. The earlier `sum=0` establishes an accumulator: a variable that retains a running total between iterations. `sum += i*i` means compute the square using the current `i`, then add it to the current sum. Do not reset `sum` to zero inside the body; doing so would discard earlier contributions.

The update `i++` is separate from the body's arithmetic. Its old expression value is unused; its important effect here is to increase `i` by 1. The final update still happens after the final successful body. Only the next test discovers that no more body should run.

Source rule: the order of the for stages is specified in [C11 draft N1570](#), §6.8.5.3p1. Its loop stopping rule is §6.8.5p4.

Worked example 1: total, body count and final state

1. Before the first test: $\text{sum}=0$ and initialization has set $i=1$.
2. Test 1: $1 \leq 4$ is true. Body: $i*i=1$; $\text{sum}=0+1=1$. Update: $i=2$.
3. Test 2: $2 \leq 4$ is true. Body: $i*i=4$; $\text{sum}=1+4=5$. Update: $i=3$.
4. Test 3: $3 \leq 4$ is true. Body: $i*i=9$; $\text{sum}=5+9=14$. Update: $i=4$.
5. Test 4: $4 \leq 4$ is true. Body: $i*i=16$; $\text{sum}=14+16=30$. Update: $i=5$.
6. Test 5: $5 \leq 4$ is false. Do not run the body or another update. Exit with $\text{sum}=30$, $i=5$.
7. Output: 30 5, followed by a newline.

There is one initialization, five condition evaluations, four body executions and four updates. The body sees $i=1,2,3,4$; the condition also sees the stopping value 5. The pair 30 4 misses the last update. The pair 55 5 wrongly adds the square of 5 even though its test fails.

Why the accumulator is correct and why the loop stops

A loop invariant is a statement about the state that remains true at a chosen checkpoint. Here choose “just before every test”: sum contains the squares of all positive integers already visited, from 1 through $i-1$. This is an ordinary description of what the variable means, not a new statement added to the code.

- At the first test, $i=1$ and no terms have been visited, so the total is 0. The statement holds before any body runs.
- If the statement holds and $i \leq 4$ is true, the body adds exactly the next square $i*i$. The update moves i to the next integer, so before the next test the total again covers exactly 1 through $i-1$.
- On exit $i=5$, so the preserved statement says sum covers 1 through 4. This agrees with $1+4+9+16=30$.

Correct accumulation and stopping are separate questions. This loop stops because i starts at 1 and increases by 1 on each completed pass until it becomes 5. Its test then fails. Every visited value, product and total is small and representable. A condition alone does not force progress: if the needed update is missing, you cannot assume the condition eventually becomes false. We only trace finite, safely bounded loops here.

while checks first; the body supplies the progress

A while loop has the form `while (condition)` followed by a body. Test the condition before every possible body execution. If false initially, the body runs zero times. A while statement has no automatic update field: the statements inside its body must change the state when progress requires it.

Worked example 2: halve an integer on every pass

C code 2

```
#include <stdio.h>

int main(void)
{
    int n = 41;
    int steps = 0;
    while (n > 3) {
        n = n / 2;
        steps++;
    }
    printf("%d %d\n", n, steps);
    return 0;
}
```

1. Start: $n=41$, $steps=0$.
2. Test 1: $41>3$ is true. Body: $n=41/2=20$; then steps becomes 1.
3. Test 2: $20>3$ is true. Body: $n=20/2=10$; then steps becomes 2.
4. Test 3: $10>3$ is true. Body: $n=10/2=5$; then steps becomes 3.
5. Test 4: $5>3$ is true. Body: $n=5/2=2$; then steps becomes 4.
6. Test 5: $2>3$ is false. Exit without another division or increment.
7. Output: 2 4, followed by a newline.

The body runs four times; the condition is evaluated five times. n is assigned four times by the body, and $steps++$ runs four times. Do not invent a fifth update after the final failed test. There is no separate for-style update stage in this while loop.

At each test, steps counts completed halvings and n is the integer obtained from the starting 41 by exactly those halvings. The description is true for zero steps; each body carries out one integer division and increments steps once, so it stays true. Integer division is performed on every pass: $41/2$ is 20, and $5/2$ is 2. Keeping 20.5 or 2.5 would trace a different kind of arithmetic.

For termination, while $n>3$, n is an integer at least 4. Dividing it by 2 gives a nonnegative integer strictly smaller than n . A strictly decreasing nonnegative integer cannot keep decreasing forever, so it eventually reaches a value at most 3 and the next test fails. This reasoning uses both the test and the update. With a starting n of 3 or 0, the first test is already false: zero bodies, one test, and steps remains 0. Source rules: [N1570](#), §6.8.5.1p1 and integer division §6.5.5p6.

Empty ranges and the post-test alternative

A for or while loop can have an empty set of body executions. For example, start a for loop at $i=6$ and use $i \leq 4$: initialization still sets i to 6, the first test fails, and neither body nor update runs. The accumulator keeps its starting value. “A loop must run once to test its condition” is false for these pre-test forms.

A do...while loop puts the condition after the body. Its sequence is body → test → body → test, until a test fails. There is a semicolon after its closing while(condition). With normal entry and completion of the body, the body runs once before any test, even when the corresponding initial pre-test condition would be false. Practice question 3 below compares this boundary directly. See [N1570](#), §6.8.5.2p1.

For the finite loops in this lesson, which leave only because the condition becomes false, use these counts:

- for: one initialization; k bodies and k updates; k+1 tests, including the last failed test. If k=0, there is one test and no updates.
- while: k bodies and k+1 tests. Count assignments and increments from its body; there is no implicit update.
- do...while: k bodies and k tests, with k at least 1 in the examples here. Each completed body is followed by one test, including the final false one.

These are counts of the source program's logical steps. They do not claim a particular number of machine instructions or physical tests after compiler optimization. The counts assume the simple normal flow shown here; early exits and other control-flow features are outside this lesson.

Practice: trace before checking the answers

These four questions are unscored. Record tests, bodies and updates separately rather than looking only at the last printed value.

Practice question 1: an update larger than one

List the i values seen by the body, the final sum and final i. Count the condition evaluations and updates.

C code 3

```
#include <stdio.h>

int main(void)
{
    int sum=0,i;
    for(i=2;i<9;i+=3){sum+=i;}
    printf("%d %d\n",sum,i);
    return 0;
}
```

Practice question 2: count integer divisions

How often does the body execute? Give count and n at the end, and include the final failed test in your trace.

C code 4

```
#include <stdio.h>

int main(void)
{
    int n=30, count=0;
    while(n>0){n=n/3; count++;}
    printf("%d %d\n", count, n);
    return 0;
}
```

Practice question 3: first test or first body?

Compare the two loops. Give a and b at the end, with the number of bodies and tests for each.

C code 5

```
#include <stdio.h>

int main(void)
{
    int a=0, b=0;
    while(a<0){a++;}
    do{b++;}while(b<0);
    printf("%d %d\n", a, b);
    return 0;
}
```

Practice question 4: choose the inclusive stopping condition

Fill the test in `for(i=1; __; i++)` so that the body `sum+=i` adds exactly 1 through 5. Give the final total and counts. Assume sum begins at 0 and the body contains only `sum+=i`.

Practice solutions and wrong-answer feedback

Practice question 1 solution

1. Start with `sum=0`; initialization sets `i=2`.
2. Test `2<9` is true. Body adds 2: `sum=2`. Update adds 3: `i=5`.
3. Test `5<9` is true. Body adds 5: `sum=7`. Update: `i=8`.
4. Test `8<9` is true. Body adds 8: `sum=15`. Update: `i=11`.
5. Test `11<9` is false. Exit. Output: 15 11, then a newline.

The body visits 2,5,8. There are three bodies, three updates and four tests. A final `i` of 8 omits the last update. A final `i` of 9 assumes the loop stops exactly on its bound, but the update jumps from 8 to 11. `sum=26` wrongly includes 11 after its failed test. The invariant is that `sum` contains exactly the visited terms before the current `i`; positive increments of 3 eventually move `i` beyond the bound.

Practice question 2 solution

1. Start: $n=30$, $\text{count}=0$. Test $30>0$ is true.
2. First body: $n=30/3=10$, $\text{count}=1$. Next test $10>0$ is true.
3. Second body: $n=10/3=3$, $\text{count}=2$. Next test $3>0$ is true.
4. Third body: $n=3/3=1$, $\text{count}=3$. Next test $1>0$ is true.
5. Fourth body: $n=1/3=0$, $\text{count}=4$. Final test $0>0$ is false.

There are four bodies and five tests; output is 4 0, then a newline. Three bodies is wrong because $n=1$ still satisfies $n>0$. Five bodies counts the final failed test as a body. A fractional final n is wrong because both operands of each division are int. While $n>0$, division by 3 strictly decreases this positive integer, including the step 1 to 0, so the loop stops. count records exactly how many divisions have completed.

Practice question 3 solution

1. Start: $a=0$, $b=0$.
2. The while test $a<0$ is $0<0$, false. Its body runs zero times; its test runs once; a remains 0.
3. The do body runs before its first test. $b++$ changes b from 0 to 1.
4. Its test $b<0$ is now $1<0$, false. The do loop has one body and one test.
5. Output: 0 1, followed by a newline.

The pair 0 0 incorrectly tests the do condition before its body. The pair 1 1 incorrectly grants the while loop an initial body execution. The do test reads the updated $b=1$, not the starting $b=0$. No extra update happens after either failed test.

Practice question 4 solution

C code 6

```
#include <stdio.h>

int main(void)
{
    int sum=0;
    for(int i=1;i<=5;i++){sum+=i;}
    printf("%d\n",sum);
    return 0;
}
```

The declaration `int i=1` in the `for` initializer supplies the starting value. This `i` belongs to the loop and is not available after it; `printf` therefore prints only `sum`. The next lesson returns to local variable scope.

Use `i<=5`; `i<6` is also correct for this integer sequence. The body sees `i=1,2,3,4,5` and the running sum becomes 1,3,6,10,15. After the fifth body, `i++` makes the loop's `i` equal to 6. The sixth test fails, so the output is 15 and a newline. There is one initialization, five bodies, five updates and six tests.

The condition `i<5` omits the term 5 and produces 10. The condition `i<=6` includes an unwanted 6 and produces 21. A proposal that only happens to print 15 but visits other terms does not meet “exactly 1 through 5.” The accumulator begins at 0 and never resets. Before each test it equals the sum of positive integers less than the current `i`; when the failing test is reached, the intended five terms have been added.

Carry a loop into a function

You can now distinguish the last successful body from the final failed test, describe what an accumulator means, and explain why a small loop stops. Next, place a finite loop inside a function and track its local variables separately from the caller's variables.

Analogy and its limits

Picture adding the prices of a short list to a running total. Before each addition, a marker says which item is next. A for loop first places the marker, checks whether that item belongs to the list, adds its price, and moves the marker. After the last addition the marker moves past the list; the next check stops the work. That is why the final marker position differs from the last position used for an addition.

The running total remembers earlier items. Clearing it before every new item would keep only the last price. A while loop is similar but leaves the marker movement to the body. A do...while loop is like doing one action before checking whether to repeat; use it only when that initial action is wanted. The analogy describes source-level order, not processor timing.

Quick reference

- for: initialization once → test → body → update → test again.
- A false for test skips both the next body and its update. A last successful body still gets its update.
- while: test → body → test. Any progress update must be supplied by the body or condition; our examples keep updates in the body.
- do...while: body → test. The closing while(condition) is followed by a semicolon.
- For normal finite loops that exit only on a false condition: for has k bodies, k updates and k+1 tests; while has k bodies and k+1 tests; do...while has k bodies and k tests, $k \geq 1$.
- A pre-test loop can have zero bodies and one test. An initially empty range leaves the accumulator unchanged.
- Initialize an accumulator before the loop; add to its current value rather than resetting it each pass.
- State an invariant at one checkpoint, then check its starting truth and how a body/update preserves it.
- Explain progress toward a failing condition separately from correct accumulation. Check intermediate arithmetic stays representable.
- Report body-visited values, the stopping value, body/test/update counts, and the final state. Source-step counts are not machine-instruction counts.

[Back to contents](#)

लूप, अपडेट और रुकने की शर्त

अंतिम वर्ग जोड़ने के बाद क्या होता है?

यह प्रोग्राम वर्गों का योग निकालता है। printf पर sum और i, दोनों का मान बताइए। खास तौर पर सोचिए कि लूप समाप्त होते समय i अभी भी 4 है या नहीं।

C कोड 1

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

```
int main(void)
{
    int sum = 0;
    int i;
    for (i = 1; i <= 4; i++) {
        sum += i * i;
    }
    printf("%d %d\n", sum, i);
    return 0;
}
```

लूप अपनी शर्त के अनुमति देने तक body दोहराता है। एक पूरा चक्कर पुनरावृत्ति (iteration) है। आप असाइनमेंट पढ़ना और शून्य/गैर-शून्य शर्त समझना जानते हैं। वही नियम रखें: अगली जाँच और अपडेट में वर्तमान अवस्था ली जाती है, शुरुआती अवस्था नहीं। यहाँ C11, पढ़ने से पहले मान पाए हुए int चर और -32767...32767 के भीतर रहने वाले मध्यवर्ती परिणाम हैं। दिखाए गए हर पूरे प्रोग्राम को अलग संकलित किया जा सकता है।

for के चार चरण पढ़ें

for (i=1; i<=4; i++) में पहला भाग शुरुआती अवस्था बनाता है, बीच का भाग शर्त है और अंतिम भाग अपडेट है। दो semicolon इन भागों को अलग करते हैं; उनका अर्थ यह नहीं है कि तीनों body से पहले चलेंगे।

1. आरंभ: लूप तक पहुँचने पर i=1 केवल एक बार चलाएँ।
2. जाँच: वर्तमान i से i<=4 निकालें। गलत हो तो तुरंत लूप छोड़ें।
3. body: जाँच सही हो तो बड़े कोष्ठकों के कथन क्रम से चलाएँ।
4. अपडेट: यह body पूरी होने पर i++ चलाएँ, फिर चरण 2 पर लौटें, चरण 1 पर नहीं।

यहाँ i की घोषणा लूप से पहले है, इसलिए बाद के printf में भी वह उपलब्ध है। int i; घोषणा अपने आप उपयोगी शुरुआती मान नहीं देती, लेकिन for का आरंभ किसी भी पढ़ने से पहले उसमें 1 रखता है। पहले का sum=0 एक संचायक (accumulator) बनाता है: ऐसा चर जो पुनरावृत्तियों के बीच अब तक का योग रखता है। sum += i*i का अर्थ है वर्तमान i का वर्ग निकालकर उसे वर्तमान sum में जोड़ना। body के अंदर sum को फिर शून्य न करें; ऐसा करने से पहले जोड़े गए पद खो जाएँगे।

अपडेट i++, body की गणना से अलग है। यहाँ इस व्यंजक का पुराना मान उपयोग नहीं होता; उसका जरूरी प्रभाव i को 1 बढ़ाना है। अंतिम सफल body के बाद भी अंतिम अपडेट होता है। उसके बाद वाली जाँच ही बताती है कि अब और body नहीं चलनी चाहिए।

स्रोत-नियम: for के चरणों का क्रम [C11 के N1570 मसौदे](#), §6.8.5.3p1 में है। लूप के रुकने का नियम §6.8.5p4 में है।

हल किया हुआ उदाहरण 1: योग, body की संख्या और अंतिम अवस्था

1. पहली जाँच से पहले: $sum=0$ और आरंभ ने $i=1$ किया है।
2. जाँच 1: $1 \leq 4$ सही है। body: $i*i=1$; $sum=0+1=1$ । अपडेट: $i=2$ ।
3. जाँच 2: $2 \leq 4$ सही है। body: $i*i=4$; $sum=1+4=5$ । अपडेट: $i=3$ ।
4. जाँच 3: $3 \leq 4$ सही है। body: $i*i=9$; $sum=5+9=14$ । अपडेट: $i=4$ ।
5. जाँच 4: $4 \leq 4$ सही है। body: $i*i=16$; $sum=14+16=30$ । अपडेट: $i=5$ ।
6. जाँच 5: $5 \leq 4$ गलत है। न body चलाएँ, न एक और अपडेट। $sum=30$, $i=5$ पर बाहर निकलें।
7. आउटपुट: 30 5, फिर नई पंक्ति।

आरंभ एक बार, शर्त का मूल्यांकन पाँच बार, body चार बार और अपडेट चार बार होता है। body को $i=1,2,3,4$ मिलता है; शर्त को रुकने वाला मान 5 भी मिलता है। 30 4 की जोड़ी में अंतिम अपडेट छूटा है। 5 5 में गलती से 5 का वर्ग भी जोड़ा गया है, जबकि उसकी जाँच गलत है।

संचायक सही क्यों है और लूप रुकता क्यों है?

लूप इनवेरिएंट (loop invariant) अवस्था के बारे में ऐसा कथन है जो चुने हुए जाँच-बिंदु पर हर बार सही रहता है। यहाँ बिंदु चुनें “हर जाँच के ठीक पहले”: sum में पहले आ चुके धनात्मक पूर्णाकों, यानी 1 से $i-1$ तक के वर्गों का योग है। यह चर का अर्थ बताने वाला सामान्य वर्णन है, कोड में जोड़ा जाने वाला नया कथन नहीं।

- पहली जाँच पर $i=1$ है और अभी कोई पद नहीं आया, इसलिए योग 0 है। body चलने से पहले ही वर्णन सही है।
- मान लें वर्णन सही है और $i \leq 4$ भी सही है। body ठीक अगला वर्ग $i*i$ जोड़ती है। अपडेट i को अगला पूर्णांक बनाता है; इसलिए अगली जाँच से पहले योग फिर ठीक 1 से $i-1$ तक है।
- बाहर निकलते समय $i=5$ है, इसलिए बना रहा वर्णन बताता है कि sum में 1 से 4 तक के वर्ग हैं। यह $1+4+9+16=30$ से मेल खाता है।

सही योग बनना और लूप रुकना अलग प्रश्न हैं। यह लूप इसलिए रुकता है कि i , 1 से शुरू होकर हर पूरे चक्कर पर 1 बढ़ता है और 5 बन जाता है। तब जाँच गलत होती है। बीच में आने वाला हर मान, गुणनफल और योग छोटा तथा int में निरूपित करने योग्य है। केवल शर्त लिखने से प्रगति निश्चित नहीं होती: जरूरी अपडेट न हो तो यह नहीं मान सकते कि शर्त कभी अपने आप गलत हो जाएगी। यहाँ हम केवल समाप्त होने वाले, सुरक्षित छोटी सीमा के लूप का ट्रेस करते हैं।

while पहले जाँचता है; प्रगति body करती है

while में while (condition) के बाद body होती है। हर संभावित body से पहले शर्त जाँचें। शुरुआत में गलत हो तो body शून्य बार चलती है। while कथन में अपने आप चलने वाला अपडेट भाग नहीं है: प्रगति के लिए अवस्था बदलनी हो तो body के कथनों को वह करना होगा।

हल किया हुआ उदाहरण 2: हर बार पूर्णांक को आधा करें

C कोड 2

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

```
int main(void)
{
    int n = 41;
    int steps = 0;
    while (n > 3) {
        n = n / 2;
        steps++;
    }
    printf("%d %d\n", n, steps);
    return 0;
}
```

1. शुरुआत: $n=41$, $steps=0$ ।
2. जाँच 1: $41>3$ सही है। body: $n=41/2=20$; फिर $steps$ में 1 हो जाता है।
3. जाँच 2: $20>3$ सही है। body: $n=20/2=10$; फिर $steps$ में 2 हो जाता है।
4. जाँच 3: $10>3$ सही है। body: $n=10/2=5$; फिर $steps$ में 3 हो जाता है।
5. जाँच 4: $5>3$ सही है। body: $n=5/2=2$; फिर $steps$ में 4 हो जाता है।
6. जाँच 5: $2>3$ गलत है। एक और भाग या बढ़ोतरी किए बिना बाहर निकलें।
7. आउटपुट: 2 4, फिर नई पंक्ति।

body चार बार और शर्त का मूल्यांकन पाँच बार होता है। body में n का असाइनमेंट चार बार तथा $steps++$ चार बार चलता है। अंतिम गलत जाँच के बाद पाँचवाँ अपडेट न जोड़ें। इस while में for जैसा अलग अपडेट चरण नहीं है।

हर जाँच पर $steps$ पूरे हो चुके आधा करने वाले चरण गिनता है और n , शुरुआती 41 पर उतने पूर्णांक भाग करके मिला मान है। शून्य चरण पर यह वर्णन सही है; हर body एक पूर्णांक भाग करती है और $steps$ एक बार बढ़ाती है, इसलिए वर्णन सही बना रहता है। हर चक्कर में पूर्णांक भाग ही करें: $41/2$ से 20 और $5/2$ से 2 मिलता है। 20.5 या 2.5 रखने पर आप अलग प्रकार के अंकगणित का ट्रेस करेंगे।

रुकने के लिए देखें: $n>3$ सही रहने तक n कम-से-कम 4 वाला पूर्णांक है। उसे 2 से भाग देने पर गैर-ऋणात्मक पूर्णांक मिलता है जो n से सख्ती से छोटा है। लगातार घटता गैर-ऋणात्मक पूर्णांक अनंत बार नहीं घट सकता, इसलिए अंततः वह 3 या उससे कम होगा और अगली जाँच गलत होगी। इस तर्क में शर्त और अपडेट दोनों जरूरी हैं। शुरुआती $n=3$ या $n=0$ हो तो पहली जाँच ही गलत है: शून्य body, एक जाँच, और $steps$ का मान 0 रहता है। स्रोत-नियम: [N1570](#), §6.8.5.1p1 और पूर्णांक भाग के लिए §6.5.5p6।

खाली सीमा और body के बाद जाँचने वाला रूप

for या while में body एक भी बार न चलना संभव है। उदाहरण के लिए for में $i=6$ से शुरू करें और $i\leq 4$ जाँचें: आरंभ फिर भी i में 6 रखता है, लेकिन पहली जाँच गलत होती है और n body चलती है न अपडेट। संचायक का शुरुआती मान बना रहता है। “शर्त जाँचने के लिए लूप को एक बार चलना ही पड़ेगा” इन पहले-जाँच वाले रूपों के लिए गलत है।

do...while में शर्त body के बाद होती है। उसका क्रम body → जाँच → body → जाँच है, जब तक कोई जाँच गलत न हो। अंत वाले while(condition) के बाद semicolon होता है। सामान्य प्रवेश और body पूरी होने पर, पहली जाँच से पहले body एक बार चलती है, भले ही शुरू में की जाने वाली वैसी ही शर्त गलत होती। नीचे अभ्यास प्रश्न 3 इसी सीमा की तुलना करता है। [N1570](#), §6.8.5.2p1 देखें।

इस पाठ के सीमित लूप केवल शर्त गलत होने पर निकलते हैं। उनके लिए ये गिनतियाँ रखें:

- for: एक आरंभ; k body और k अपडेट; अंतिम गलत जाँच सहित k+1 जाँच। k=0 हो तो एक जाँच और कोई अपडेट नहीं।
- while: k body और k+1 जाँच। असाइनमेंट और बढ़ोतरी उसकी body से गिनें; कोई छिपा अपडेट नहीं है।
- do...while: k body और k जाँच; यहाँ के उदाहरणों में k कम-से-कम 1 है। हर पूरी body के बाद एक जाँच है, जिसमें अंतिम गलत जाँच भी शामिल है।

ये मूल प्रोग्राम के तार्किक चरणों की गिनतियाँ हैं। इनसे compiler optimization के बाद मशीन निर्देशों या वास्तव में चलाए गए परीक्षणों की कोई निश्चित संख्या नहीं बताई जा रही। गिनतियाँ यहाँ के सामान्य सरल प्रवाह पर लागू हैं; बीच से निकलना और अन्य नियंत्रण-प्रवाह सुविधाएँ इस पाठ से बाहर हैं।

अभ्यास: उत्तर जाँचने से पहले ट्रेस करें

इन चार प्रश्नों के अंक नहीं हैं। केवल अंतिम छपा मान देखने के बजाय जाँच, body और अपडेट अलग-अलग लिखें।

अभ्यास प्रश्न 1: एक से बड़ा अपडेट

body में आने वाले i के मान, अंतिम sum और अंतिम i लिखिए। शर्त के मूल्यांकन और अपडेट की संख्या गिनिए।

C कोड 3

```
#include <stdio.h>

int main(void)
{
    int sum=0,i;
    for(i=2;i<9;i+=3){sum+=i;}
    printf("%d %d\n",sum,i);
    return 0;
}
```

अभ्यास प्रश्न 2: पूर्णांक भाग की संख्या

body कितनी बार चलती है? अंत में count और n बताइए, और ट्रेस में अंतिम गलत जाँच भी शामिल कीजिए।

C कोड 4

```
#include <stdio.h>

int main(void)
{
    int n=30,count=0;
    while(n>0){n=n/3;count++;}
    printf("%d %d\n",count,n);
    return 0;
}
```

अभ्यास प्रश्न 3: पहले जाँच या पहले body?

दोनों लूप की तुलना कीजिए। अंत में a और b बताइए तथा दोनों में body और जाँच की संख्या दीजिए।

C कोड 5

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

```
int main(void)
{
    int a=0,b=0;
    while(a<0){a++;}
    do{b++;}while(b<0);
    printf("%d %d\n",a,b);
    return 0;
}
```

अभ्यास प्रश्न 4: अंतिम पद शामिल करने वाली शर्त

for(i=1; __; i++) में ऐसी शर्त भरिए कि body sum+=i ठीक 1 से 5 तक जोड़े। अंतिम योग और गिनतियाँ दीजिए। मान लें कि शुरुआत में sum=0 है और body में केवल sum+=i है।

अभ्यास के पूरे हल और गलत उत्तरों का कारण

अभ्यास प्रश्न 1 का हल

1. sum=0 से शुरू करें; आरंभ में i=2 होता है।
2. 2<9 सही है। body, 2 जोड़ती है: sum=2। अपडेट, 3 जोड़ता है: i=5।
3. 5<9 सही है। body, 5 जोड़ती है: sum=7। अपडेट: i=8।
4. 8<9 सही है। body, 8 जोड़ती है: sum=15। अपडेट: i=11।
5. 11<9 गलत है। बाहर निकलें। आउटपुट: 15 11, फिर नई पंक्ति।

body में 2,5,8 आते हैं। तीन body, तीन अपडेट और चार जाँच हैं। अंतिम i को 8 कहना अंतिम अपडेट छोड़ता है। अंतिम i को 9 कहना मान लेता है कि लूप ठीक सीमा पर रुकेगा, जबकि अपडेट 8 से 11 पर छलाँग लगाता है। sum=26 में असफल जाँच के बाद 11 को भी जोड़ दिया गया है। इनवेरिफ़ेन्स यह है कि sum में वर्तमान i से पहले body में आए ठीक उन्हीं पदों का योग है। 3 की धनात्मक बढ़ोतरी अंततः i को सीमा के पार ले जाती है।

अभ्यास प्रश्न 2 का हल

1. शुरुआत: $n=30$, $count=0$ । जाँच $30>0$ सही है।
2. पहली body: $n=30/3=10$, $count=1$ । अगली जाँच $10>0$ सही है।
3. दूसरी body: $n=10/3=3$, $count=2$ । अगली जाँच $3>0$ सही है।
4. तीसरी body: $n=3/3=1$, $count=3$ । अगली जाँच $1>0$ सही है।
5. चौथी body: $n=1/3=0$, $count=4$ । अंतिम जाँच $0>0$ गलत है।

चार body और पाँच जाँच हैं; आउटपुट 4 0, फिर नई पंक्ति है। तीन body कहना गलत है क्योंकि $n=1$ पर भी $n>0$ सही है। पाँच body कहने में अंतिम गलत जाँच को body गिन लिया गया है। अंतिम n को भिन्न मान देना गलत है, क्योंकि हर भाग के दोनों operands int हैं। $n>0$ रहने पर 3 से भाग इस धनात्मक पूर्णांक को सख्ती से घटाता है, 1 से 0 वाला चरण भी ऐसा करता है, इसलिए लूप रुकता है। count ठीक पूरे हो चुके भागों की संख्या रखता है।

अभ्यास प्रश्न 3 का हल

1. शुरुआत: $a=0$, $b=0$ ।
2. while की जाँच $a<0$ अर्थात $0<0$ गलत है। उसकी body शून्य बार और जाँच एक बार होती है; a में 0 रहता है।
3. do की body पहली जाँच से पहले चलती है। $b++$ से b , 0 से 1 होता है।
4. उसकी जाँच $b<0$ अब $1<0$ है, जो गलत है। do में एक body और एक जाँच हुई।
5. आउटपुट: 0 1, फिर नई पंक्ति।

0 0 की जोड़ी में do की शर्त को body से पहले जाँच लिया गया है। 1 1 में while को भी गलती से एक शुरुआती body दे दी गई है। do की जाँच अपडेट हुआ $b=1$ पढ़ती है, शुरुआती $b=0$ नहीं। किसी भी गलत जाँच के बाद अतिरिक्त अपडेट नहीं होता।

अभ्यास प्रश्न 4 का हल

C कोड 6

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

```
int main(void)
{
    int sum=0;
    for(int i=1;i<=5;i++){sum+=i;}
    printf("%d\n",sum);
    return 0;
}
```

for के आरंभ में int i=1 की घोषणा ही शुरुआती मान देती है। यह i उसी लूप का है और उसके बाद उपलब्ध नहीं है; इसलिए printf केवल sum छापता है। अगले पाठ में स्थानीय चर के scope पर फिर आएँगे।

i<=5 लिखें; इस पूर्णांक क्रम के लिए i<6 भी सही है। body में i=1,2,3,4,5 आते हैं और चलता योग 1,3,6,10,15 बनता है। पाँचवीं body के बाद i++ से लूप वाला i, 6 हो जाता है। छठी जाँच गलत होती है, इसलिए आउटपुट 15 और नई पंक्ति है। एक आरंभ, पाँच body, पाँच अपडेट और छह जाँच हैं।

i<5 शर्त में पद 5 छूटता है और योग 10 आता है। i<=6 में अनचाहा 6 जुड़ता है और योग 21 आता है। किसी और पद-क्रम से संयोग से 15 छप जाए, तब भी “ठीक 1 से 5 तक” की माँग पूरी नहीं होती। संचायक 0 से शुरू होता है और फिर शून्य पर रीसेट नहीं होता। हर जाँच से पहले वह वर्तमान i से छोटे धनात्मक पूर्णांकों का योग है; असफल जाँच तक पहुँचने पर इच्छित पाँच पद जुड़ चुके हैं।

लूप को फ़ंक्शन में ले जाना

अब आप अंतिम सफल body और अंतिम गलत जाँच अलग कर सकते हैं, संचायक का अर्थ बता सकते हैं और समझा सकते हैं कि छोटा लूप क्यों रुकता है। अगले पाठ में सीमित लूप को फ़ंक्शन के अंदर रखेंगे और उसके स्थानीय चर, caller के चरों से अलग ट्रेस करेंगे।

उपमा और उसकी सीमाएँ

मान लें छोटी सूची की कीमतें एक चलते योग में जोड़नी हैं। हर जोड़ से पहले एक निशान बताता है कि अगली वस्तु कौन-सी है। `for` पहले निशान रखता है, जाँचता है कि वह वस्तु सूची में है या नहीं, उसकी कीमत जोड़ता है, फिर निशान आगे बढ़ाता है। अंतिम जोड़ के बाद भी निशान सूची के पार बढ़ता है; अगली जाँच काम रोकती है। इसलिए निशान की अंतिम जगह और जोड़ के लिए उपयोग हुई अंतिम जगह अलग हैं।

चलता योग पहले की वस्तुओं को याद रखता है। हर नई वस्तु से पहले उसे मिटाने पर केवल अंतिम कीमत बचेगी। `while` में भी यही विचार है, लेकिन निशान आगे बढ़ाने का काम `body` पर है। `do...while` ऐसा है जैसे एक काम करने के बाद पूछना कि उसे दोहराना है या नहीं; उसे तभी चुनें जब वह पहला काम सच में चाहिए। यह उपमा स्रोत-कोड का क्रम समझाती है, `processor` का समय नहीं।

त्वरित संदर्भ

- `for`: एक बार आरंभ → जाँच → `body` → अपडेट → फिर जाँच।
- `for` की गलत जाँच अगली `body` और उसका अपडेट, दोनों छोड़ती है। अंतिम सफल `body` के बाद भी उसका अपडेट होता है।
- `while`: जाँच → `body` → जाँच। प्रगति का अपडेट `body` या शर्त में दिया जाना चाहिए; हमारे उदाहरणों में अपडेट `body` में हैं।
- `do...while`: `body` → जाँच। अंत के `while(condition)` के बाद semicolon है।
- केवल गलत शर्त पर निकलने वाले सामान्य सीमित लूप में: `for` के `k` `body`, `k` अपडेट और `k+1` जाँच; `while` के `k` `body` और `k+1` जाँच; `do...while` के `k` `body` और `k` जाँच, जहाँ $k \geq 1$ है।
- पहले-जाँच वाले लूप में शून्य `body` और एक जाँच हो सकती है। शुरुआत से खाली सीमा में संचायक नहीं बदलता।
- संचायक को लूप से पहले मान दें; हर बार रीसेट करने के बजाय उसके वर्तमान मान में जोड़ें।
- एक जाँच-बिंदु पर इनवेरिपेंट बताएँ; शुरू में उसका सही होना और `body`/अपडेट से बना रहना जाँचें।
- सही योग से अलग यह समझाएँ कि गलत शर्त की ओर प्रगति कैसे होती है। बीच की गणनाएँ भी `int` में निरूपित होनी चाहिए।
- `body` में आए मान, रुकने वाला मान, `body`/जाँच/अपडेट की गिनती और अंतिम अवस्था दें। स्रोत के चरणों की गिनती मशीन निर्देशों की गिनती नहीं है।

[विषय सूची पर लौटें](#)

Functions, copied arguments and local state

What changes when a function is called

This program changes an `x` inside `transform`. Does the `x` in `main` change too? Predict the two printed numbers before reading the trace.

C code 1

```
#include <stdio.h>

int transform(int x) {
    x = x + 4;
    return 2 * x;
}

int main(void) {
    int x = 3;
    int y = transform(x);
    printf("%d %d\n", x, y);
    return 0;
}
```

A function packages a calculation under a name. In `int transform(int x)`, the first `int` says that a call returns an integer. `transform` is the function's name, and `int x` declares its integer parameter. The braces contain its body. The definition appears before `main`, so its name and parameter type are known when it is called.

In `transform(x)`, the expression between parentheses is an argument. Here it reads the caller's `x`. The parameter is the separate variable declared by the function; the argument supplies its starting value. The return value is the result sent back by `return`. These are three roles: supplied value, local receiving variable, and result of the call.

An ordinary `int` argument is passed by value: the parameter receives a copy of the argument's value. Changing the copy does not change the caller's integer variable. A repeated name does not join the two variables. Even if the parameter had been named `p`, the call would still supply 3.

Use two separate state records whenever control enters a function: one for the caller and one for the current call. While the helper runs, leave the caller's record visible but unchanged unless a later caller statement assigns a new value.

Worked trace 1 A copied parameter

1. Before the call, the caller is `main` and its state is `x = 3`. The initialization of `y` is waiting for a value.
2. For `transform(x)`, read the caller's current `x`: the argument value is 3. Enter `transform` with a separate parameter also named `x`, initialized to 3. Record two boxes: caller `x = 3`; callee `x = 3`.
3. In `transform`, `x = x + 4` reads the callee's 3 and stores 7 in that same callee parameter. The caller's box still contains 3.

4. `return 2 * x` computes $2 * 7 = 14$ and ends this call. It returns the number 14; it does not print 14 or assign to the caller's `x`.
5. Back in `main`, the call expression has value 14, so the waiting initialization completes with `y = 14`. The caller's state is now `x = 3, y = 14`.
6. `printf` reads the caller's two variables in the written output order. The output line is `3 14`.

The decisive check is where each assignment happens. `x = x + 4` belongs to `transform`; `int y = transform(x)` belongs to `main`. The latter stores the returned number in `y`. Returning a value and assigning it to a chosen caller variable are separate steps. No “copy back” happens when the helper finishes.

`return` ends the current function call immediately. If an `if` branch executes a `return`, statements after that `return` are not executed during that call. The integer helpers here provide a return value on every reachable path. `printf` is the operation that displays output; the return in a helper only supplies the call's value.

Worked trace 2 A loop inside a helper

The next helper adds the positive odd integers no larger than its argument. It is called twice. Predict whether the second call begins with the first call's total.

C code 2

```
#include <stdio.h>

int odd_sum(int n) {
    int total = 0;
    for (int k = 1; k <= n; k += 2) {
        total += k;
    }
    return total;
}

int main(void) {
    int n = 6;
    int first = odd_sum(n);
    int second = odd_sum(3);
    printf("%d %d %d\n", n, first, second);
    return 0;
}
```

1. Caller: `n = 6`. The first call receives parameter `n = 6` and initializes its own `total = 0`.
2. First call, test `k = 1`: `1 <= 6` is true. Add 1, so `total = 1`; then `k += 2` makes `k = 3`.
3. First call, test `k = 3`: true. Add 3, so `total = 4`; update to `k = 5`.
4. First call, test `k = 5`: true. Add 5, so `total = 9`; update to `k = 7`.
5. First call, test `k = 7`: `7 <= 6` is false. The body has run 3 times and the test 4 times. Return 9; caller stores `first = 9`.
6. Second call: the argument is the constant 3. A new parameter has `n = 3`, and this call's new local `total` is initialized to 0. Caller `n` remains 6.
7. Second call, test `k = 1`: true. Add 1, giving `total = 1`; update to `k = 3`.

8. Second call, test `k = 3: true`. Add 3, giving `total = 4`; update to `k = 5`.
9. Second call, test `k = 5: false`. The body has run 2 times and the test 3 times. Return 4; caller stores `second = 4`.
10. Caller state: `n = 6`, `first = 9`, `second = 4`. The output line is `6 9 4`.

At the start of each loop test, `total` contains the sum of the odd positive integers already processed. Initially none have been processed, so 0 is correct. Each successful body adds the current odd `k`; the update moves to the next odd integer. This explains the accumulator's meaning, not just the final number.

For the supplied inputs, `k` increases by 2 until it exceeds `n`, so the loop terminates. At the boundary `odd_sum(0)`, the first test is `1 <= 0`, which is false: no body executes and the function returns 0. A helper with a loop can still return normally when its loop executes zero times.

The local variable `total` is initialized by `int total = 0` every time this declaration is reached on a new call. "Local" does not mean "automatically zero": the initializer is what supplies 0. The second call has its own initialized local; it does not continue from 9. Here `total` can be named from its declaration through the rest of the helper's body, including the loop body. The `k` declared in the `for` initializer can be named in that loop's test, update and body; it is outside its scope after the loop. `main` cannot name either helper-local variable. It receives a number through the return value instead.

The caller's `n` and the helper's parameter `n` are also separate. The second call uses the literal 3, so it does not even read caller `n` for that argument. The first call's returned 9 remains in caller `first` while the second call computes its result.

Combining small functions

One function's returned value can be another function's argument. Read `outer(inner(value))` by finding the inner call's result, then using that number as the argument of the outer call. This follows the data needed by the calls; it is not a rule that unrelated C arguments always run left to right.

In the practice below, `twice` multiplies its integer parameter by 2 and `plus3` adds 3. Applying the two calculations in a different order can change the result. Keep each call's argument and returned value visible instead of mentally merging the two function bodies. These helpers use only their integer parameters and local calculations, so there are no other state changes to track.

A reliable call trace

1. Read the current value of the argument expression in the caller.
2. Start a separate callee state and give the parameter that value.
3. Execute the helper's statements in order, including tests, loop updates and local initializers.
4. Stop that call at the executed return and record its value.
5. Resume the caller's pending expression and store the value only where the caller's statement says to store it.

All programs here use C11, initialized ordinary signed `int` values and small, representable intermediate results. The shown helpers are defined before their calls. Trace only the supplied inputs and the explicitly stated small boundaries; a loop formula is not a promise of safe arithmetic for every possible `int`.

Practice

For each program, record the caller state, the argument received by each helper, and the returned value. Try all four before opening their solutions.

Practice 1

Give the caller value and returned result.

C code 3

```
#include <stdio.h>

int decrease(int p){p-=2;return p;}

int main(void) {
    int z=9;
    int ans=decrease(z);
    printf("%d %d\n",z,ans);
    return 0;
}
```

Practice 2

Compare the two compositions.

C code 4

```
#include <stdio.h>

int twice(int p){return p*2;}
int plus3(int p){return p+3;}

int main(void) {
    int a=twice(plus3(4));
    int b=plus3(twice(4));
    printf("%d %d\n",a,b);
    return 0;
}
```

Practice 3

Find results at a negative input, zero, and a positive input.

C code 5

```
#include <stdio.h>

int nonnegative(int p){if(p>=0){return 1;}return 0;}

int main(void) {
    int a=nonnegative(-2);
    int b=nonnegative(0);
    int c=nonnegative(3);
    printf("%d %d %d\n",a,b,c);
    return 0;
}
```

Practice 4

Trace two calls and explain why the second total starts at zero.

C code 6

```
#include <stdio.h>

int sum_to(int n){int total=0;for(int k=1;k<=n;k++){total+=k;}return total;}

int main(void) {
    int a=sum_to(4);
    int b=sum_to(2);
    printf("%d %d\n",a,b);
    return 0;
}
```

Practice solutions

Practice 1

1. Caller starts with $z = 9$. The call `decrease(z)` supplies argument value 9.
2. Inside `decrease`, the separate parameter starts at $p = 9$. $p -= 2$ gives $p = 7$.
3. `return p` returns 7. Caller stores $ans = 7$; caller z is still 9.
4. The output line is 9 7.

7 7 would incorrectly copy the parameter's change back to z . 9 9 would ignore the executed subtraction and returned value. 7 9 makes both mistakes. Track the variable that each assignment actually names.

Practice 2

1. For a, the inner call is `plus3(4)`. Its parameter is 4 and it returns $4 + 3 = 7$.
 2. The outer call is therefore `twice(7)`. It returns $7 * 2 = 14$, stored in a.
 3. For b, the inner call is `twice(4)`, which returns 8.
 4. The outer call is therefore `plus3(8)`, which returns 11, stored in b.
 5. The output line is 14 11. Each helper uses a new parameter for its own call.
- 11 14 reverses which composition belongs to a and b. 14 14 assumes adding 3 and multiplying by 2 can be interchanged; they cannot, because in the first composition the added 3 is also doubled. 7 8 stops after the inner calls and omits the outer calls.

Practice 3

1. With argument -2, parameter `p = -2`. The test `p >= 0` is false, so execution reaches `return 0`. Thus `a = 0`.
 2. With argument 0, the test `0 >= 0` is true. `return 1` ends this call immediately, so the later `return 0` is not executed. Thus `b = 1`.
 3. With argument 3, the test is true and the call returns 1. Thus `c = 1`.
 4. The output line is 0 1 1. Both routes return an integer.
- 0 0 1 treats `>=` as `>` and wrongly excludes zero. 0 0 0 assumes the final return always overwrites an earlier return, but a returned call has already ended. -2 0 3 repeats the arguments even though the function returns a classification value.

Practice 4

1. First call: parameter `n = 4`, local `total = 0`.
 2. For `k = 1, 2, 3, 4`, the successive totals are 1, 3, 6, 10. After each body, `k` increases by 1. At `k = 5`, the test fails. There are 4 body executions and 5 tests.
 3. Return 10 and store `a = 10` in the caller.
 4. Second call: parameter `n = 2`, a fresh local `total = 0`. For `k = 1, 2`, totals are 1, 3; at `k = 3`, the test fails. There are 2 body executions and 3 tests.
 5. Return 3 and store `b = 3`. The output line is 10 3.
- 10 13 incorrectly starts the second accumulator at 10. 6 1 omits the endpoint despite `k <= n`. 15 6 wrongly executes the body for the first failed-test values 5 and 3. The update prepares the next test; it does not guarantee another body execution.

Check your explanation

A complete explanation distinguishes the caller's variable, the current call's parameter, and the returned number. If a helper has a loop, include its first failed test. Use these same records in the separate C foundation learning review.

Source note

C11 semantics are referenced to the [WG14 N1570 committee draft](#). For copied arguments, local initialization, loop scope and return values, see §§6.5.2.2, 6.2.1-6.2.4, 6.8.5.3 and 6.8.6.4.

Analogy and its limits

Think of a helper as a calculation desk. You write the number 3 on a request slip while keeping your own record of 3. The desk writes 3 in its own working box, changes that box to 7, and sends back a result slip showing 14. Your record still says 3. If you file the result under *y*, then *y* becomes 14.

On the next visit, the desk begins a new worksheet. In `odd_sum`, the written instruction `int total = 0` starts that worksheet's total at zero. An old worksheet's total is not reused. The separate boxes model distinct integer variables; the paper story is only a tracing aid, not a claim about how a computer physically stores them.

Quick reference

- Function definition: return type, function name, typed parameters, then a braced body; place these definitions before calls.
- Argument: the value supplied by the caller. Parameter: a separate receiving variable. Return value: the result of the call expression.
- An ordinary `int` parameter receives a copy. Assigning to it does not assign to the caller's variable.
- `return expression`; evaluates the expression, ends this call and supplies its result. It does not print.
- A caller stores the returned number only when its own statement does so.
- When a local declaration is reached in a new call, its initializer runs for that call. `int total = 0` supplies zero; locality alone does not.
- A variable declared in a block is usable only within its scope. The `int k` in a `for` initializer belongs to that loop, not the statements after it.
- For a nested single-argument call, compute the inner return value to obtain the outer argument. Do not infer a general left-to-right argument-order rule.
- Trace loops inside helpers exactly as before: test, body, update, next test, including the final false test.
- Worked trace 1 prints 3 14; Worked trace 2 prints 6 9 4. Restrict arithmetic claims to the stated small inputs.

[Back to contents](#)

फ़ंक्शन, आर्गुमेंट की कॉपी और स्थानीय अवस्था

फ़ंक्शन कॉल होने पर क्या बदलता है

यह प्रोग्राम transform के अंदर एक x बदलता है। क्या main का x भी बदलता है? ट्रेस पढ़ने से पहले छपने वाली दोनों संख्याओं का अनुमान लगाइए।

C कोड 1

```
#include <stdio.h>

int transform(int x) {
    x = x + 4;
    return 2 * x;
}

int main(void) {
    int x = 3;
    int y = transform(x);
    printf("%d %d\n", x, y);
    return 0;
}
```

फ़ंक्शन किसी गणना को एक नाम देता है। `int transform(int x)` में पहला `int` बताता है कि कॉल एक पूर्णांक लौटाती है। `transform` फ़ंक्शन का नाम है और `int x` उसका पूर्णांक पैरामीटर घोषित करता है। ब्रेस के अंदर उसकी body है। परिभाषा `main` से पहले दी गई है, इसलिए कॉल के स्थान पर फ़ंक्शन का नाम और पैरामीटर का प्रकार ज्ञात हैं।

`transform(x)` में कोष्ठक के अंदर का व्यंजक आर्गुमेंट है। यहाँ वह caller का `x` पढ़ता है। पैरामीटर वह अलग चर है जिसे फ़ंक्शन की परिभाषा घोषित करती है; आर्गुमेंट उसका शुरुआती मान देता है। लौटाया गया मान वह परिणाम है जो `return` वापस भेजता है। ये तीन भूमिकाएँ अलग हैं: दिया गया मान, उसे ग्रहण करने वाला स्थानीय चर और कॉल का परिणाम।

सामान्य `int` आर्गुमेंट मान की कॉपी के रूप में दिया जाता है (pass by value): पैरामीटर को आर्गुमेंट के मान की कॉपी मिलती है। कॉपी बदलने से caller का पूर्णांक चर नहीं बदलता। एक जैसा नाम दोनों चरों को एक नहीं बना देता। पैरामीटर का नाम `p` होता, तब भी इस कॉल से उसे 3 ही मिलता।

फ़ंक्शन में प्रवेश करते समय अवस्था के दो अलग रिकॉर्ड बनाएँ: एक caller का और दूसरा वर्तमान कॉल का। helper के चलने के दौरान caller का रिकॉर्ड सामने रखें, लेकिन उसे न बदलें; बाद में caller का कोई कथन नया मान असाइन करे, तभी उसका रिकॉर्ड बदलें।

हल किया ट्रेस 1 पैरामीटर में मान की कॉपी

1. कॉल से पहले बुलाने वाला फ़ंक्शन (caller) `main` है और उसकी अवस्था `x = 3` है। `y` का आरंभीकरण अभी मान मिलने की प्रतीक्षा कर रहा है।
2. `transform(x)` के लिए caller के वर्तमान `x` को पढ़ें: आर्गुमेंट का मान 3 है। `transform` में प्रवेश करने पर अलग पैरामीटर `x` को 3 मिलता है। दो अलग खाने लिखें: caller में `x = 3`; बुलाए गए फ़ंक्शन (callee) में `x = 3`।
3. `transform` के अंदर `x = x + 4` callee का 3 पढ़कर उसी पैरामीटर में 7 रखता है। caller के खाने में अभी भी 3 है।

4. `return 2 * x` से $2 * 7 = 14$ निकलता है और यह कॉल समाप्त होती है। यह संख्या 14 लौटाता है; न 14 छापता है, न caller के `x` में असाइन करता है।
5. `main` में लौटने पर कॉल व्यंजक का मान 14 है, इसलिए रुका हुआ आरंभीकरण पूरा होता है: $y = 14$ । अब caller की अवस्था $x = 3, y = 14$ है।
6. `printf` caller के दोनों चरों को लिखे गए आउटपुट क्रम में पढ़ता है। छपने वाली पंक्ति 3 14 है।

मुख्य जाँच यह है कि हर असाइनमेंट कहाँ होता है। $x = x + 4$ का कथन `transform` में है;

`int y = transform(x)` का कथन `main` में है। दूसरा कथन लौटे मान को `y` में रखता है। मान लौटाना और caller के चुने गए चर में उसे रखना अलग चरण हैं। `helper` के समाप्त होने पर पैरामीटर का मान अपने आप caller के चर में वापस कॉपी नहीं होता।

`return` वर्तमान फ़ंक्शन कॉल तुरंत समाप्त कर देता है। यदि किसी `if` शाखा में `return` चलता है, तो उस कॉल में उसके बाद के कथन नहीं चलते। यहाँ के पूर्णांक `helpers` हर संभव निष्पादन-पथ पर मान लौटाते हैं। आउटपुट दिखाने का काम `printf` करता है; `helper` का `return` केवल कॉल का मान देता है।

हल किया ट्रेस 2 helper के अंदर लूप

अगला `helper` अपने आर्गुमेंट से छोटी या उसके बराबर धनात्मक विषम संख्याएँ जोड़ता है। इसे दो बार बुलाया गया है। पहले अनुमान लगाइए: क्या दूसरी कॉल पहली कॉल के योग से शुरू होगी?

C कोड 2

```
#include <stdio.h>

int odd_sum(int n) {
    int total = 0;
    for (int k = 1; k <= n; k += 2) {
        total += k;
    }
    return total;
}

int main(void) {
    int n = 6;
    int first = odd_sum(n);
    int second = odd_sum(3);
    printf("%d %d %d\n", n, first, second);
    return 0;
}
```

1. caller में $n = 6$ है। पहली कॉल के पैरामीटर को $n = 6$ मिलता है और उसका अपना स्थानीय `total = 0` से शुरू होता है।
2. पहली कॉल में $k = 1$ की जाँच: $1 \leq 6$ सही है। 1 जोड़ने पर `total = 1`; फिर $k += 2$ से $k = 3$ ।
3. पहली कॉल में $k = 3$ की जाँच सही है। 3 जोड़ने पर `total = 4`; अपडेट के बाद $k = 5$ ।
4. पहली कॉल में $k = 5$ की जाँच सही है। 5 जोड़ने पर `total = 9`; अपडेट के बाद $k = 7$ ।
5. पहली कॉल में $k = 7$ की जाँच: $7 \leq 6$ गलत है। `body` 3 बार चली और शर्त 4 बार जाँची गई। 9 लौटता है; caller में `first = 9` रखा जाता है।

6. दूसरी कॉल का आर्गुमेंट स्थिरांक 3 है। नए पैरामीटर में $n = 3$ है और इस कॉल का नया स्थानीय total फिर 0 से आरंभ होता है। caller का n अभी भी 6 है।
7. दूसरी कॉल में $k = 1$ की जाँच सही है। 1 जोड़ने पर $total = 1$; अपडेट के बाद $k = 3$ ।
8. दूसरी कॉल में $k = 3$ की जाँच सही है। 3 जोड़ने पर $total = 4$; अपडेट के बाद $k = 5$ ।
9. दूसरी कॉल में $k = 5$ की जाँच गलत है। body 2 बार चली और शर्त 3 बार जाँची गई। 4 लौटता है; caller में $second = 4$ रखा जाता है।
10. caller की अवस्था: $n = 6$, $first = 9$, $second = 4$ । छपने वाली पंक्ति 6 9 4 है।

लूप की हर जाँच के शुरू में total में पहले से संसाधित धनात्मक विषम संख्याओं का योग है। शुरू में ऐसी कोई संख्या नहीं है, इसलिए 0 सही है। हर सफल body वर्तमान विषम k जोड़ती है; अपडेट अगली विषम संख्या तक पहुँचाता है। इससे केवल अंतिम संख्या नहीं, योग जमा करने वाले चर का अर्थ भी स्पष्ट होता है।

दिए गए inputs पर k हर बार 2 बढ़ता है और अंत में n से बड़ा हो जाता है, इसलिए लूप रुकता है। सीमा-मान `odd_sum(0)` पर पहली जाँच $1 \leq 0$ गलत है: body एक बार भी नहीं चलती और फ़ंक्शन 0 लौटाता है। helper का लूप शून्य बार चलने पर भी helper सामान्य रूप से मान लौटा सकता है।

हर नई कॉल में `int total = 0` घोषणा तक पहुँचने पर स्थानीय चर total फिर आरंभ होता है। “स्थानीय” होने का अर्थ “अपने आप शून्य” नहीं है: 0 देने वाला initializer है। दूसरी कॉल का अपना आरंभ किया हुआ चर है; वह 9 से आगे नहीं बढ़ती। यहाँ total का नाम उसकी घोषणा के बाद helper की बाकी body में, लूप की body सहित, उपयोग किया जा सकता है। for के आरंभीकरण में घोषित k का नाम उसी लूप की शर्त, अपडेट और body में उपयोग होता है; लूप के बाद वह अपने scope के बाहर है। main इन दोनों helper-स्थानीय चरों का नाम लेकर उन्हें नहीं पढ़ सकता। उसे return value के माध्यम से संख्या मिलती है।

caller का n और helper का पैरामीटर n भी अलग हैं। दूसरी कॉल में स्थिरांक 3 है, इसलिए उस आर्गुमेंट के लिए caller का n पढ़ा भी नहीं जाता। पहली कॉल से लौटा 9 caller के first में रहता है, जबकि दूसरी कॉल अपना परिणाम निकालती है।

छोटे फ़ंक्शन जोड़कर गणना करना

एक फ़ंक्शन का लौटाया मान दूसरे का आर्गुमेंट हो सकता है। `outer(inner(value))` पढ़ते समय पहले अंदर वाली कॉल का परिणाम निकालें, फिर वही संख्या बाहर वाली कॉल के आर्गुमेंट के रूप में लें। यह कॉल के लिए आवश्यक मान का क्रम है; इसका अर्थ यह नहीं कि C के असंबंधित आर्गुमेंट हमेशा बाएँ से दाएँ चलते हैं।

नीचे के अभ्यास में `twice` अपने पूर्णांक पैरामीटर को 2 से गुणा करता है और `plus3` उसमें 3 जोड़ता है। दोनों गणनाओं का क्रम बदलने से परिणाम बदल सकता है। दोनों फ़ंक्शन की body को मन में मिलाने के बजाय हर कॉल का आर्गुमेंट और लौटाया मान अलग लिखें। ये helpers केवल अपने पूर्णांक पैरामीटर और स्थानीय गणना उपयोग करते हैं, इसलिए किसी दूसरी अवस्था में बदलाव का ट्रेस नहीं करना है।

कॉल का भरोसेमंद ट्रेस

1. caller में आर्गुमेंट व्यंजक का वर्तमान मान पढ़ें।
2. callee की अलग अवस्था शुरू करें और उसके पैरामीटर को वही मान दें।
3. helper के कथन क्रम से चलाएँ; शर्तें, लूप अपडेट और स्थानीय आरंभीकरण भी लिखें।

4. जिस return का निष्पादन हो, वहीं उस कॉल को रोकें और उसका मान लिखें।
5. caller का अधूरा व्यंजक पूरा करें और मान केवल उसी चर में रखें जहाँ caller का कथन कहता है।

यहाँ सभी प्रोग्राम C11, आरंभ किए गए सामान्य signed int और छोटी, निरूपित की जा सकने वाली मध्यवर्ती संख्याएँ उपयोग करते हैं। दिखाए गए helpers अपनी कॉल से पहले परिभाषित हैं। केवल दिए गए inputs और स्पष्ट रूप से बताए छोटे सीमा-मानों का ट्रेस करें; लूप का कोई सूत्र हर संभव int के लिए सुरक्षित अंकगणित की गारंटी नहीं है।

अभ्यास

हर प्रोग्राम में caller की अवस्था, हर helper को मिला आर्गुमेंट और लौटाया मान लिखें। हल देखने से पहले चारों प्रश्न आजमाएँ।

अभ्यास 1

caller के चर और लौटे परिणाम का मान बताइए।

C कोड 3

```
#include <stdio.h>

int decrease(int p){p-=2;return p;}

int main(void) {
    int z=9;
    int ans=decrease(z);
    printf("%d %d\n",z,ans);
    return 0;
}
```

अभ्यास 2

दोनों फ़ंक्शन-रचनाओं की तुलना कीजिए।

C कोड 4

```
#include <stdio.h>

int twice(int p){return p*2;}
int plus3(int p){return p+3;}

int main(void) {
    int a=twice(plus3(4));
    int b=plus3(twice(4));
    printf("%d %d\n",a,b);
    return 0;
}
```

अभ्यास 3

ऋणात्मक मान, शून्य और धनात्मक मान पर परिणाम बताइए।

C कोड 5

```
#include <stdio.h>

int nonnegative(int p){if(p>=0){return 1;}return 0;}

int main(void) {
    int a=nonnegative(-2);
    int b=nonnegative(0);
    int c=nonnegative(3);
    printf("%d %d %d\n",a,b,c);
    return 0;
}
```

अभ्यास 4

दो कॉल का ट्रेस कीजिए और बताइए कि दूसरी बार total शून्य से क्यों शुरू होता है।

C कोड 6

```
#include <stdio.h>

int sum_to(int n){int total=0;for(int k=1;k<=n;k++){total+=k;}return total;}

int main(void) {
    int a=sum_to(4);
    int b=sum_to(2);
    printf("%d %d\n",a,b);
    return 0;
}
```

अभ्यास के हल

अभ्यास 1

1. caller में शुरू में $z = 9$ है। $\text{decrease}(z)$ कॉल आर्गुमेंट मान 9 देती है।
2. decrease के अंदर अलग पैरामीटर $p = 9$ से शुरू होता है। $p -= 2$ से $p = 7$ होता है।
3. $\text{return } p$ से 7 लौटता है। caller में $\text{ans} = 7$ रखा जाता है; caller का z अभी भी 9 है।
4. आउटपुट पंक्ति 9 7 है।

7 7 मानने पर पैरामीटर का बदलाव गलत ढंग से z में वापस कॉपी किया गया है। 9 9 मानने पर घटाव और लौटे मान को छोड़ा गया है। 7 9 में दोनों गलतियाँ हैं। हर असाइनमेंट वास्तव में जिस चर का नाम लेता है, उसी का ट्रेस करें।

अभ्यास 2

1. a के लिए अंदर वाली कॉल `plus3(4)` है। उसका पैरामीटर 4 है और वह $4 + 3 = 7$ लौटाती है।
2. इसलिए बाहर वाली कॉल `twice(7)` है। वह $7 * 2 = 14$ लौटाती है, जो a में रखा जाता है।
3. b के लिए अंदर वाली कॉल `twice(4)` है, जो 8 लौटाती है।
4. इसलिए बाहर वाली कॉल `plus3(8)` है, जो 11 लौटाती है; यह b में रखा जाता है।
5. आउटपुट पंक्ति 14 11 है। हर helper अपनी कॉल के लिए नया पैरामीटर उपयोग करता है।

11 14 में a और b की फ़ंक्शन-रचनाएँ उलट दी गई हैं। 14 14 मानता है कि 3 जोड़ने और 2 से गुणा करने का क्रम बदला जा सकता है; ऐसा नहीं है, क्योंकि पहली रचना में जोड़ा गया 3 भी दुगुना होता है। 7 8 में अंदर वाली कॉल के मान रखकर बाहर वाली कॉल छोड़ दी गई है।

अभ्यास 3

1. आर्गुमेंट -2 पर पैरामीटर `p = -2` है। `p >= 0` गलत है, इसलिए निष्पादन `return 0` तक पहुँचता है। अतः `a = 0`।
2. आर्गुमेंट 0 पर `0 >= 0` सही है। `return 1` यह कॉल तुरंत समाप्त करता है, इसलिए बाद का `return 0` नहीं चलता। अतः `b = 1`।
3. आर्गुमेंट 3 पर शर्त सही है और कॉल 1 लौटाती है। अतः `c = 1`।
4. आउटपुट पंक्ति 0 1 1 है। दोनों रास्ते पूर्णांक लौटाते हैं।

0 0 1 में `>=` को `>` मानकर शून्य को गलत ढंग से बाहर किया गया है। 0 0 0 मानता है कि अंतिम `return` पहले `return` को बदल देगा, लेकिन `return` हो चुकी कॉल समाप्त हो चुकी है। -2 0 3 आर्गुमेंट दोहराता है, जबकि फ़ंक्शन वर्गीकरण का मान लौटाता है।

अभ्यास 4

1. पहली कॉल: पैरामीटर `n = 4`, स्थानीय `total = 0`।
2. `k = 1, 2, 3, 4` पर क्रमशः योग 1, 3, 6, 10 होते हैं। हर `body` के बाद `k` में 1 जुड़ता है। `k = 5` पर शर्त गलत है। `body` 4 बार चलती है और शर्त 5 बार जाँची जाती है।
3. 10 लौटता है और caller में `a = 10` रखा जाता है।
4. दूसरी कॉल: पैरामीटर `n = 2`, नया स्थानीय `total = 0`। `k = 1, 2` पर योग 1, 3 हैं; `k = 3` पर शर्त गलत है। `body` 2 बार चलती है और शर्त 3 बार जाँची जाती है।
5. 3 लौटता है और `b = 3` रखा जाता है। आउटपुट पंक्ति 10 3 है।

10 13 में दूसरी बार योग गलत ढंग से 10 से शुरू किया गया है। 6 1 में `k <= n` होने के बावजूद अंतिम मान छोड़ दिया गया है। 15 6 में पहली असफल जाँच के मानों 5 और 3 पर भी `body` चला दी गई है। अपडेट अगली जाँच की तैयारी करता है; वह अगली `body` चलने की गारंटी नहीं है।

अपनी व्याख्या जाँचें

पूरी व्याख्या caller के चर, वर्तमान कॉल के पैरामीटर और लौटाई संख्या को अलग रखती है। helper में लूप हो तो उसकी पहली असफल जाँच भी शामिल करें। अलग C बुनियाद पुनराभ्यास में भी यही रिकॉर्ड उपयोग करें।

स्रोत टिप्पणी

C11 के अर्थ-नियमों का संदर्भ [WG14 का N1570 समिति मसौदा](#) है। आर्ग्युमेंट की कॉपी, स्थानीय आरंभीकरण, लूप के scope और लौटाए मान के लिए §§6.5.2.2, 6.2.1-6.2.4, 6.8.5.3 और 6.8.6.4 देखें।

उपमा और उसकी सीमाएँ

helper को गणना करने वाली मेज़ समझें। आप पर्ची पर संख्या 3 लिखकर देते हैं और अपने रिकॉर्ड में 3 बनाए रखते हैं। मेज़ अपने अलग काम वाले खाने में 3 लिखती है, उसे 7 करती है और परिणाम की पर्ची पर 14 लौटाती है। आपके रिकॉर्ड में अभी भी 3 है। यदि आप परिणाम को y के नाम से रखते हैं, तो y का मान 14 होता है।

अगली बार मेज़ नया कार्यपत्र शुरू करती है। odd_sum में लिखा निर्देश `int total = 0` उस कार्यपत्र का योग शून्य से शुरू करता है। पुराने कार्यपत्र का योग दोबारा उपयोग नहीं होता। अलग खाने अलग पूर्णांक चरों को समझाते हैं; कागज़ की यह तुलना ट्रेस में मदद के लिए है, कंप्यूटर के वास्तविक भंडारण का दावा नहीं।

त्वरित संदर्भ

- फ़ंक्शन परिभाषा: लौटने का प्रकार, फ़ंक्शन का नाम, प्रकार सहित पैरामीटर, फिर ब्रेस में body; इन परिभाषाओं को कॉल से पहले रखें।
- आर्ग्युमेंट: caller से दिया गया मान। पैरामीटर: उसे ग्रहण करने वाला अलग चर। लौटाया मान: कॉल व्यंजक का परिणाम।
- सामान्य `int` पैरामीटर को कॉपी मिलती है। उसमें असाइन करने से caller के चर में असाइन नहीं होता।
- `return expression;` व्यंजक का मान निकालता है, वर्तमान कॉल समाप्त करता है और परिणाम देता है। वह छापता नहीं है।
- caller लौटाई संख्या तभी किसी चर में रखता है जब उसका अपना कथन ऐसा कहता है।
- नई कॉल में स्थानीय घोषणा तक पहुँचने पर उसका initializer उस कॉल के लिए चलता है। `int total = 0` शून्य देता है; केवल स्थानीय होना पर्याप्त नहीं है।
- ब्लॉक में घोषित चर केवल अपने scope में उपयोग होता है। `for` के आरंभीकरण का `int k` उस लूप का है, उसके बाद के कथनों का नहीं।
- एक-आर्ग्युमेंट वाली नेस्टेड कॉल में अंदर का लौटाया मान बाहर का आर्ग्युमेंट बनता है। इससे आर्ग्युमेंट के सामान्य बाएँ-से-दाएँ क्रम का नियम न निकालें।
- helper के लूप का ट्रेस पहले की तरह करें: शर्त, body, अपडेट, अगली शर्त; अंतिम गलत शर्त भी शामिल करें।
- हल किया ट्रेस 1 में 3 14 और हल किया ट्रेस 2 में 6 9 4 छपता है। अंकगणित के दावे बताए गए छोटे inputs तक सीमित रखें।

[विषय सूची पर लौटें](#)