

Level 0 Training

Session 1

Intro to Problem Solving

Data types • Conditions • Loops

A study reference for your first steps
in programming and problem solving.

How to use this reference

This is a book to **learn from after the session**, not a set of slides to memorize. You do not need previous programming experience. Start with small examples, make predictions, and use your own compiler to check them.

A simple way to study

1. **Read one idea.** Explain it aloud in your own words. Do not move on just because the code looks familiar.
2. **Trace the example.** Write the values of the variables after each important step, before running the program.
3. **Type and change it.** Run the code, then change an input or a condition. Predict what will change.
4. **Solve without copying.** Try the practice problems before looking at hints or solutions. Return to a solution later and write it again from memory of the idea, not the exact lines.

What the labels mean

Complete program means the code can be saved as a `.cpp` file and compiled on its own. **Inside main** means the fragment belongs between the braces of the starter program. A **wrong example** is deliberately incorrect and is never a template to copy.

Pause and predict questions have explained answers nearby. Cover the answer first. **Optional extension** material is useful for a second reading; it is not a prerequisite for solving the core practice.

The language used in this book

The examples are written for **C++23**. We use ordinary, readable C++ rather than advanced features simply because they are new. Select C++23 in your compiler or use the build command in Section 1.2. All complete example and solution files are included in the source package.

You are not expected to learn everything in one sitting

First study input/output, variables, arithmetic, `if`, and `for`. Then return to `while`, nested loops, common patterns, and the worked problems. The quick reference is for revision *after* you understand the explanations.

The Level 0 learning path

Session	Topics
1	Intro to problem solving: data types, conditions, and loops.
2	Arrays and strings.
3	Functions and complexity analysis.
4	STL, part 1: vector, pair, deque, queue, priority queue, and stack.
5	STL, part 2: set, multiset, ordered set, ordered map, and unordered map.
6	Static range queries: prefix sum, partial sum, and frequency array.

What is included in this edition

This booklet contains **Session 1 only**. Its chapter numbering, typography, and source structure are designed to carry over into the collected book. Sessions 2–6 are shown here only to explain where the training is going.

What you should be able to do after Session 1

Read a short problem statement and identify its input, output, and constraints. Turn a small example into clear steps. Write and run a C++ program that reads values, calculates an answer, makes decisions, and repeats work. Trace that program and test the cases most likely to break it.

You will also practise choosing between `int`, `long long`, and `double`; avoiding integer-division and overflow mistakes; using conditions accurately; and reading several values without needing an array yet.

Unless a task says otherwise, assume its input follows the specified format and constraints. The solutions do not ask the user to re-enter invalid data.

What can wait until later

Arrays, string processing, custom functions, recursion, formal complexity analysis, and STL containers are not required here. A brief mention of `std::string` only distinguishes text from a single character. We will not ask you to use untaught containers to solve Session 1 exercises.

Contents

How to use this reference	i
The Level 0 learning path	ii
1 Intro to Problem Solving	1
1.1 From a problem statement to a plan	2
1.2 Your first C++ program	4
1.3 Reading input and printing output	6
1.4 Variables and data types	8
1.5 Arithmetic, conversions, and safe calculations	12
1.6 Conditions: choosing what happens	16
1.7 Combining conditions with logic	19
1.8 Other ways to choose an alternative	21
1.9 For loops: repeating a known pattern	23
1.10 While and do-while loops	26
1.11 Break and continue	29
1.12 Loop patterns you will reuse	31
1.13 Nested loops: rows and columns	36
1.14 Working with the digits of a number	38
1.15 Handling multiple test cases	40
1.16 Worked problems: combine the building blocks	42
1.17 Debugging and testing your solution	47
1.18 Practice problems	50
1.19 Hints before solutions	54
1.20 Explained practice solutions	55
1.21 Quick reference and revision checklist	72
1.22 Technical notes and references	76

SESSION 1

Intro to Problem Solving

Data types, conditions, and loops

A program is a set of instructions that a computer can follow. Problem solving is the work you do *before and while* writing those instructions: understanding the task, choosing steps that work, and checking that they work for every allowed input.

You do not need to be fast yet. Your first goal is to write a small correct program and explain why it is correct.

Three building blocks

Building block	Question it answers	Example
Data and variables	What information do I need to remember?	A score, a total, or a number of students.
Conditions	Which action should happen?	Print PASS when a score is at least 50.
Loops	Which action should be repeated?	Read each student's score and add it to a total.

Before you begin

Have a C++ compiler ready, somewhere to type code, and paper for tracing. The program you type is called *source code*; the compiler checks and translates it into a program you can run.

When you feel lost, use the smallest possible example. A loop with three repetitions is easier to understand than a loop with a million repetitions, but the underlying idea can be the same.

The habit that matters most

Before pressing Run, write what you expect the program to print. After running it, compare the result with your prediction. A difference is a useful clue, not a reason to give up.

1.1 From a problem statement to a plan

Input, processing, and output

Imagine this task: two boxes contain some pencils. You are given the number of pencils in each box. Find the total.

The **input** is the information given to the program. The **processing** is the work done using that information. The **output** is the answer the program must produce.

Input	Processing	Output
7 5	$7 + 5$	12

The numbers 7 and 5 are only one example. A correct solution must also work when the boxes contain 0 and 0, or 3 and 9. We therefore read the input instead of writing an answer such as 12 directly in the program.

Read these four parts of a statement

The task tells you what to calculate or decide. Rewrite it as a short sentence: “Read two counts and print their sum.”

The input format tells you how many values to read and in which order. Reading two values in the wrong order matters when the operation is not symmetric, such as subtraction.

The output format tells you exactly what to print. Printing an explanation when the task asks for one number can make the output incorrect.

The constraints tell you the allowed values. For our pencil task, suppose $0 \leq a, b \leq 1000$. This means both counts are whole numbers between 0 and 1000, including the endpoints. The largest possible total is 2000.

What is an algorithm?

An **algorithm** is a clear sequence of steps for solving a task. You can describe it in ordinary language before translating it into C++.

For the pencil task, our algorithm is: read the first count, read the second count, add them, and print the result. These instructions also work for the other examples, so they describe a general solution.

```
Read a and b.  
Let total be a + b.  
Print total.
```

This is **pseudocode**: an informal way to write the idea. It is not C++ and is not meant to be compiled.

A repeatable problem-solving routine

1. **Understand.** What is given? What must be produced? Which values are allowed?
2. **Try by hand.** Solve one small example without code. Notice the steps you used.
3. **Plan.** Write those steps so they work for any allowed input.
4. **Implement.** Translate the plan into C++.
5. **Test.** Check ordinary inputs and boundary cases, not just the sample.

A **boundary case** lies at or near a limit. If a score of 50 is the first passing score, then 49, 50, and 51 are useful tests. Boundaries help reveal small mistakes in comparisons.

Why a sample is not proof

Suppose a program always prints 12. It passes the sample 7 5, and also passes 3 9, which happens to have the same sum. But it fails for 1 2. Two passing examples can be misleading when they happen to have the same answer.

Testing should challenge your idea. For our sum problem, try 0 0, 0 8, 3 9, and 1000 1000. Also explain why adding the two input values must give the total; this reasoning is more general than a few passing tests.

Pause and predict – 1

A task gives the length and width of a rectangle and asks for its area. What are the input, processing, and output? What would be wrong with always printing the sample answer?

Check your prediction. The input is the length and width. Multiply them to obtain the area, then print it. A fixed answer only works for inputs whose area happens to equal that answer.

1.2 Your first C++ program

Complete program – printing a message

```
#include <iostream>

int main() {
    std::cout << "Hello, world!" << '\n';
    return 0;
}
```

The output is:

```
Hello, world!
```

Read the program one piece at a time

`#include <iostream>` makes the standard input/output tools available. A **header** provides declarations for tools your program can use. This include line does not end with a semicolon.

`int main()` marks the starting function for the ordinary console programs in this book. For now, think of it as the starting point. Functions are studied properly in Session 3.

The braces `{` and `}` enclose the body of `main`. Instructions inside this body run in order unless a condition or loop changes that order.

`std::cout` is the standard output stream. The insertion operator `<<` sends the following value to it. The prefix `std::` identifies a name from the standard library. Keep this prefix in the examples; you do not need to understand namespaces deeply yet.

`"Hello, world!"` is text surrounded by double quotes. The quotes mark the text in the source code; the quotes themselves are not printed. `'\n'` represents a newline, so later output starts on a new line.

`return 0;` ends `main` and reports successful completion. It does **not** print 0. Reaching the end of `main` also returns 0, but we write it explicitly to make the ending clear.^[13]

Punctuation is part of the language

A semicolon `;` ends an ordinary statement such as an assignment or output instruction. Braces group statements. Parentheses are used in `main()` and later around conditions and expressions. These symbols are not interchangeable.

C++ is case-sensitive: `main`, `Main`, and `MAIN` are different names. The correct standard output name is `std::cout`, not `std::Cout`.

Save, compile, and run

Save the code in a file named `main.cpp`. In an editor with a Run button, choose a C++23 configuration and run the file. With a GNU C++ compiler, the command below selects C++23 and enables useful warnings.^[1]

```
g++ -std=c++23 -Wall -Wextra -Wpedantic main.cpp -o main
```

Then run the resulting program. On Linux or macOS, a terminal command is `./main`. On Windows PowerShell, a typical executable is run with `.\main.exe`. The exact executable name depends on your compiler command and setup.

Compile means translate and check the source. **Run** means execute the compiled program. If you change the source, compile it again before trusting the next run. Otherwise, you may accidentally run an older version.

A starter program you can reuse

```
#include <iostream>

int main() {
    // Write your solution here.
    return 0;
}
```

The line beginning with `//` is a **comment**. It is for people reading the source and is not executed. A multi-line comment can use `/* comment */`. Do not put a solution inside a comment by mistake.

Indentation makes structure visible

The spaces at the beginning of a line are called indentation. They help you see which statements belong inside a block. Use four spaces for each nesting level. Ordinary indentation is not what groups instructions; **the braces do that**.

When the compiler reports an error

Start with the first useful error message. Check the named line and the line immediately above it. A missing semicolon or quote can make the compiler complain later than the actual mistake. Fix one cause, then compile again.

Pause and predict – 2

Does `return 0;` print a zero? Does a comment become part of the program's output?

Check your prediction. Neither prints anything. Use an output instruction such as `std::cout << 0;` to display a zero.

1.3 Reading input and printing output

Read values instead of hard-coding them

The following program solves the pencil task for any allowed input, rather than only for one example.

Complete program – sum of two counts

```
#include <iostream>

int main() {
    int a = 0;
    int b = 0;
    std::cin >> a >> b;
    int sum = a + b;
    std::cout << sum << '\n';
    return 0;
}
```

Sample input

7 5

Sample output

12

`int a = 0;` creates an integer variable named `a` and gives it an initial value. We will study variables in the next topic. `std::cin >> a >> b;` reads the next value into `a`, then the next value into `b`.

For input `7 5`, the program remembers 7 in `a` and 5 in `b`. The next line calculates 12 and remembers it in `sum`. Finally, the program prints the value of `sum`.

Spaces and newlines in input

For the numeric input used here, `>>` skips leading whitespace, such as spaces, tabs, and newlines.^[10] Therefore, the following arrangements provide the same two values:

7 5

7
5

The important thing is the **order of the values**, not the visual position on the page. If a statement says that the first value is a count, do not treat it as the first item in a list.

Why does my program seem to be waiting?

It may be waiting for input. If the program reads two values and you supplied only one, type the second value. In an online compiler, provide the input in its input area before running.

Output does not add separators automatically

`std::cout << a << b;` prints one value immediately after the other. If `a` is 4 and `b` is 9, the output is 49, not 4 9.

Complete program – spaces, text, and newlines

```
#include <iostream>

int main() {
    int a = 4;
    int b = 9;
    std::cout << a << ' ' << b << '\n';
    std::cout << "Total: " << a + b << '\n';
    return 0;
}
```

```
4 9
Total: 13
```

Here `' '` is a space character. `'\n'` is a newline character. `"Total: "` is literal text, whereas `a + b` is an expression that is evaluated before its result is printed.

Text in quotes versus a variable name

Inside `main`, after declaring `int score = 80;`, compare:

```
std::cout << "score" << '\n'; // Prints the word score.
std::cout << score << '\n';   // Prints the value 80.
```

Double quotes are used for text such as `"YES"`. Single quotes are used for one character such as `'Y'` or `'\n'`. A letter and its lowercase form are different characters: `'A'` is not `'a'`.

Output for a judge

In a programming problem, print exactly the required information. If the output should be a single number, do not add `"Enter a:"`, `"The answer is"`, or debugging messages. Such messages are useful during local experiments but should not be in the submitted answer.

When a statement asks for YES, use the requested spelling and capitalization. Separate multiple answers as specified. Ending a line with `'\n'` is a clear default in this book. `std::endl` also adds a newline, but additionally flushes the stream; that extra operation is usually unnecessary for these tasks.

Pause and predict – 3

With `a = 4` and `b = 9`, what do these print: `std::cout << a + b;` and `std::cout << "a + b";`?

Check your prediction. The first prints 13. The second prints the text `a + b`, because everything between its double quotes is literal text.

1.4 Variables and data types

A variable remembers a value

Think of a variable as a named storage location. Its **name** lets you refer to it. Its **type** determines the kinds of values and operations it supports. Its **value** is the information it currently holds.

```
int score = 10;
score = 15;
std::cout << score << '\n';
```

The first line **declares** the variable and **initializes** it with 10. The second line **assigns** a new value, replacing 10 with 15. The output is 15. The variable does not automatically keep a history of its older values.

Assignment is an action, not an equation

```
int score = 10;
score = score + 5;
```

Read the second line from right to left: calculate the old value of `score` plus 5, then store the result back in `score`. It is not the impossible mathematical claim that a number equals itself plus 5.

If you write `int copy = score;`, `copy` receives the *current* value. Changing `score` later does not automatically change `copy`.

Types you will use most often

Type	Use it for	Example declaration
<code>int</code>	Whole numbers in a suitable range.	<code>int count = 12;</code>
<code>long long</code>	Larger whole numbers and large totals.	<code>long long sum = 0;</code>
<code>double</code>	Values with a fractional part.	<code>double x = 2.5;</code>
<code>char</code>	One ordinary character.	<code>char letter = 'A';</code>
<code>bool</code>	A true/false fact.	<code>bool found = false;</code>

A `std::string`, introduced fully in Session 2, stores a sequence of characters and needs `<string>`. It is not a fundamental type like `int`. For now, remember that one character such as `'A'` is different from a piece of text such as `"Ahmed"`.

Complete program – several types together

```
#include <iostream>

int main() {
    int solved = 3;
    long long population = 8000000000LL;
    double distance = 2.5;
    char group = 'A';
    bool ready = true;
    std::cout << solved << '\n';
    std::cout << population << '\n';
    std::cout << distance << '\n';
    std::cout << group << '\n';
    std::cout << ready << '\n';
    return 0;
}
```

```
3
8000000000
2.5
A
1
```

The suffix LL in 8000000000LL explicitly marks the integer literal as `long long`. A **literal** is a value written directly in the source, such as 3, 2.5, or 'A'.

By default, printing a `bool` with `std::cout` produces 1 for `true` and 0 for `false`. Use a condition when the problem requires words such as YES and NO.

How large can an integer be?

The exact size of an integer type depends on the implementation. C++23 guarantees a width of at least 16 bits for `int` and at least 64 for `long long`; it does not require every `int` to be 32 bits.^[2]

For the common 32-bit `int` / 64-bit `long long` environment used to test this book:

Type	Inclusive range in this environment
<code>int</code>	−2,147,483,648 to 2,147,483,647
<code>long long</code>	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

You do not need to memorize every digit. For common contest environments, remember roughly 2×10^9 for positive `int` values and 9×10^{18} for positive `long long` values. Choose the type from the **largest intermediate result**, not just from the largest input.

For example, two values of 10^9 each fit in a 32-bit `int`, but their product is 10^{18} . That product needs a wider type. The next topic explains why even the type of the multiplication matters.

Whole numbers and decimal numbers

`int items = 3;` is appropriate for counting three items. `double distance = 3.75;` is appropriate when a fractional value is allowed. Changing an integer variable's type to `double` does not automatically fix a calculation that already used integer division.

Floating-point values are usually approximations, not exact decimal storage. For example, many decimal fractions cannot be represented exactly in the binary format used by common implementations. Use integer arithmetic when a task asks for exact whole-number answers, and use `double` when fractional answers are required.

Naming variables clearly

For the simple names in this book, use letters, digits, and underscores, without starting with a digit. Do not use spaces or language keywords such as `int` and `for`. Prefer names such as `totalScore`, `count`, and `price` over names whose meaning you will forget.

Good beginner names

`score`, `total_score`, `number2`

Do not use

`2number`, `total score`, `int`

`score` and `Score` are different names. Avoid leading underscores while learning; some underscore-based names are reserved for the implementation.

Initialize before you use

```
int total = 0;
int count = 0;
bool found = false;
```

These starting values are deliberate. A sum starts at zero because adding zero changes nothing. A fact such as “have we found a negative number?” starts as false because no values have been checked yet.

Do not read an uninitialized local integer. Writing `int total;` does *not* give a local variable a dependable initial zero. In this book, initialize your variables, then read or calculate the values you need.^[14]

Constants

A value that should not change can be declared with `const`:

```
const int passMark = 50;
int score = 72;
bool passed = (score >= passMark);
```

Trying to assign a new value to `passMark` later is an error. A named constant can communicate the meaning of a number better than repeating the number itself.

Where a variable name can be used

Scope describes where a name can be used. A variable declared inside a block is normally used from its declaration to the end of that block. If you need a result after an `if` or loop, declare the result before that block. We will revisit this with actual conditions and loops.

Optional detail: other type names

You may encounter `short`, `long`, `float`, `long double`, and unsigned integer types. Their exact sizes or precision can vary. Do not assume `long` is always 64 bits, or change every integer to unsigned just because the input is nonnegative. The five core types above are enough for this chapter.

Pause and predict – 4

After `int a = 5; int b = a; a = 9;`, what are the values of `a` and `b`?

Check your prediction. `a` is 9 and `b` is 5. The assignment to `b` copied the value 5; it did not create a permanent connection between the variables.

1.5 Arithmetic, conversions, and safe calculations

Basic arithmetic operators

An **expression** is a piece of code that produces a value, such as `a + b`. An **operator** tells C++ which operation to perform.

Operator	Meaning	Example result
+	Addition	<code>7 + 2</code> gives 9
-	Subtraction	<code>7 - 2</code> gives 5
*	Multiplication	<code>7 * 2</code> gives 14
/	Division	<code>7 / 2</code> gives 3
%	Integer remainder	<code>7 % 2</code> gives 1

The remainder operator applies to integral values, not `double`. If 7 objects are placed into groups of 2, there are 3 complete groups and 1 object left over. That is why `7 / 2` is 3 and `7 % 2` is 1.

Integer division discards the fractional part

When both operands are integers, `/` performs integer division. It truncates toward zero, rather than rounding to the nearest integer.^[3]

Complete program – integer and floating-point division

```
#include <iostream>

int main() {
    int a = 7;
    int b = 2;
    std::cout << a / b << '\n';
    std::cout << a % b << '\n';
    std::cout << static_cast<double>(a) / b << '\n';
    return 0;
}
```

```
3
1
3.5
```

`static_cast<double>(a)` explicitly converts the value of `a` to `double` for that expression. It does not change the declared type of `a`. With a floating-point operand, the last division can produce a fractional result.

Inside main:

```
double wrong = 7 / 2;    // Integer division first: 3.
double correct = 7 / 2.0; // Floating-point division: 3.5.
```

The name `wrong` means wrong for the intended result of 3.5; the statement is legal C++. The destination type does not travel backward and change how `7 / 2` was computed.

For negative inputs, `-7 / 2` is `-3`, not `-4`. The corresponding remainder `-7 % 2` is `-1`. When the operations are valid and the quotient is representable:

$$a = (a/b) \times b + (a \bmod b).$$

Do not assume every nonzero remainder is positive.

Two important safety rules

Never divide or take a remainder by zero. Also keep the result within the chosen type: the most negative signed integer divided by `-1` cannot be represented in that same signed type. Our exercises use bounds that avoid that exceptional case.

Remainders answer divisibility questions

For nonzero `k`, a whole number `n` is divisible by `k` exactly when `n % k == 0`. In particular, `n % 2 == 0` checks whether `n` is even, including zero and negative even numbers.

To check oddness for both positive and negative integers, use `n % 2 != 0`, not `n % 2 == 1`. A negative odd integer can have remainder `-1`.

Order of operations

Multiplication, division, and remainder have higher precedence than addition and subtraction. Parentheses make your intention explicit.

```
int first = 2 + 3 * 4;    // 14
int second = (2 + 3) * 4; // 20
int third = 20 / 5 * 2;   // 8: (20 / 5) * 2
```

Do not translate mathematical multiplication as `2(n + 1)`. Write `2 * (n + 1)`. Also, `^` is **not** a power operator in C++; `2 ^ 3` does not mean `23`. For a small square, write `x * x`. Repeated multiplication can calculate small integer powers later.

Changing a variable

```
int x = 10;
x += 3; // x = x + 3; now x is 13.
x -= 2; // x = x - 2; now x is 11.
x *= 4; // x = x * 4; now x is 44.
x /= 5; // x = x / 5; now x is 8.
x %= 3; // x = x % 3; now x is 2.
x++;    // Add 1; now x is 3.
x--;    // Subtract 1; now x is 2.
```

These are updates: every statement uses the value left by the previous statement. They do not all use the original 10.

When used alone, `x++` and `++x` both increase `x` by one. Inside an expression they differ: postfix `x++` produces the old value, while prefix `++x` produces the new value. Keep increments on separate lines or in loop updates while learning; clever combinations are harder to reason about.

Prevent overflow before it happens

Overflow means a result does not fit in its intended representation. Signed integer overflow has undefined behavior: there is no valid answer you may rely on, and assuming it will simply wrap is unsafe.^[2]

Deliberately unsafe example on a 32-bit int system:

```
int a = 1000000000;
int b = 1000000000;
long long product = a * b; // WRONG: int multiplication.
```

The multiplication uses two `int` operands. It can overflow *before* the value is assigned to `product`. A large destination does not rescue an already-invalid calculation.

Complete program – widen before multiplying

```
#include <iostream>

int main() {
    int a = 1000000000;
    int b = 1000000000;
    long long product = 1LL * a * b;
    std::cout << product << '\n';
    return 0;
}
```

Here `1LL * a` uses `long long` arithmetic, and the following multiplication stays wide. The result is 10^{18} , which fits in a 64-bit `long long`. Another clear solution is to read both inputs directly into `long long` variables.

Bounds are still necessary. A `long long` is large, not unlimited. If your largest possible intermediate value can exceed its range, you need a different plan; merely using this type everywhere is not a proof of safety.

Printing a decimal answer

Suppose two scores are integers from 0 to 100, and the required output is their average with two digits after the decimal point.

Complete program – a decimal average

```
#include <iostream>
#include <iomanip>

int main() {
    int a = 0;
    int b = 0;
    std::cin >> a >> b;
    double average = (a + b) / 2.0;
    std::cout << std::fixed << std::setprecision(2);
    std::cout << average << '\n';
    return 0;
}
```

Sample input

7 8

Sample output

7.50

`<iomanip>` provides formatting tools. Together, `std::fixed` and `std::setprecision(2)` request two digits after the decimal point. Without `fixed`, precision has a different meaning: it normally controls significant digits. These settings affect later floating-point output on the stream until changed.^[11]

Formatting changes the displayed answer, not the underlying stored value. Converting a floating-point value to an integer also does not round to the nearest whole number; it discards the fractional part, and the result must fit the destination type.^[4]

Optional detail: comparing computed decimal values

For exact counts, use integer comparisons. For computed floating-point results, exact equality can be too strict because of representation and calculation errors. When a task defines an acceptable error, use that specification; there is no single tolerance suitable for every problem. We keep this session's branching examples integer-based to avoid introducing a separate numerical-analysis topic.

Pause and predict – 5

What is the value of `double x = 5 / 2;`? Give two ways to obtain 2.5 instead.

Check your prediction. The value is 2.0 because the division is integer division. Use `5 / 2.0` or `static_cast<double>(5) / 2`. Changing only the destination type is not enough.

1.6 Conditions: choosing what happens

A program does not always perform the same actions. A **condition** is a question that can be answered with **true** or **false**. An **if** statement uses that answer to decide whether to execute a block.^[5]

Comparison operators

Operator	Meaning	Example
==	Equal to	5 == 5 is true
!=	Not equal to	5 != 3 is true
<	Less than	5 < 3 is false
<=	Less than or equal to	5 <= 5 is true
>	Greater than	5 > 3 is true
>=	Greater than or equal to	5 >= 7 is false

Inside main:

```
int score = 65;
if (score >= 50) {
    std::cout << "PASS" << '\n';
}
std::cout << "DONE" << '\n';
```

With a score of 65, the program prints **PASS** and then **DONE**. With a score of 40, it skips the block and prints only **DONE**. The statement after the block is not part of the decision.

Two alternatives: if and else

Use **else** when one action should happen if the condition is true and a different action should happen otherwise. Exactly one of the two branches runs.

Complete program – even or odd

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    if (n % 2 == 0) {
        std::cout << "EVEN" << '\n';
    } else {
        std::cout << "ODD" << '\n';
    }
    return 0;
}
```

Sample input	Sample output
8	EVEN

The condition checks the remainder after division by 2. If it is zero, print **EVEN**. Otherwise, print **ODD**. Zero is even because dividing zero by 2 leaves no remainder.

Several alternatives: an else-if chain

Complete program – positive, negative, or zero

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    if (n > 0) {
        std::cout << "POSITIVE" << '\n';
    } else if (n < 0) {
        std::cout << "NEGATIVE" << '\n';
    } else {
        std::cout << "ZERO" << '\n';
    }
    return 0;
}
```

For input -6 , the program first checks $n > 0$, which is false. It then checks $n < 0$, which is true, and prints **NEGATIVE**. The final **else** is not executed.

For input 0 , both comparisons are false, so the final branch prints **ZERO**. There is no need to repeat $n == 0$ there, because the earlier failures already tell us that the integer is neither positive nor negative.

Independent if statements are different

```
int score = 95;
if (score >= 50) {
    std::cout << "Passed" << '\n';
}
if (score >= 90) {
    std::cout << "Excellent" << '\n';
}
```

Both messages print because the two conditions are checked independently. That is useful when several facts can be true at once. An **if / else if / else** chain is useful when you want one category.

Ordering conditions matters

Suppose grades are defined as A for 90–100, B for 75–89, C for 50–74, and F for 0–49. The input is guaranteed to be between 0 and 100.

Complete program – first matching grade

```
#include <iostream>

int main() {
    int score = 0;
    std::cin >> score;
    if (score >= 90) {
        std::cout << 'A' << '\n';
    } else if (score >= 75) {
        std::cout << 'B' << '\n';
    } else if (score >= 50) {
        std::cout << 'C' << '\n';
    } else {
        std::cout << 'F' << '\n';
    }
    return 0;
}
```

The strongest threshold is checked first. If you begin with `score >= 50`, a score of 95 matches it immediately and never reaches the A branch. In this version, reaching the B test already tells us the score is below 90.

Test each boundary: 49 and 50, 74 and 75, 89 and 90. A single mistaken `>` instead of `>=` can misclassify an exact threshold.

Common condition mistakes

Assignment instead of comparison. `x = 5` stores 5 in `x`; `x == 5` asks whether `x` equals 5. An assignment can be legal inside a condition, which makes this mistake dangerous. Use `==` when you mean equality.

An accidental semicolon. Writing `if (score >= 50);` ends the controlled statement immediately. A block written after that semicolon can run regardless of the condition. Do not place a semicolon after an ordinary `if (...)` header.

Missing braces. Without braces, an `if` controls only one statement. Use braces even for a one-line branch; adding another line later will then be safer. Braces also make it clear which `if` owns an `else`.

Using a name outside its scope. A variable declared inside a branch is not available after the branch. Declare a result beforehand and assign it in the branches if you need to print it afterward.

Pause and predict – 6

A score of 90 is the first A grade. Should the test be `score > 90` or `score >= 90`? Why?

Check your prediction. Use `score >= 90`. The boundary value 90 belongs in the A category, so equality must be included.

1.7 Combining conditions with logic

Sometimes one comparison is not enough. A valid score might need to be both at least 0 *and* at most 100. A discount might apply on Saturday *or* Sunday.

AND, OR, and NOT

`&&` means logical AND: both sides must be true. `||` means logical OR: at least one side must be true, including the case where both are true. `!` means logical NOT: it reverses a true/false result.

A	B	A && B	A B	!A
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

This is a **truth table**. It lists all possible true/false combinations, so you can check the meaning of the operators rather than memorizing a vague description.

Inside an inclusive interval

To check $10 \leq x \leq 20$, combine the two comparisons:

Complete program – checking both boundaries

```
#include <iostream>

int main() {
    int x = 0;
    std::cin >> x;
    if (x >= 10 && x <= 20) {
        std::cout << "INSIDE" << '\n';
    } else {
        std::cout << "OUTSIDE" << '\n';
    }
    return 0;
}
```

Sample input

20

Sample output

INSIDE

$x \geq 10$ prevents values below 10. $x \leq 20$ prevents values above 20. Both conditions must hold, so the correct operator is AND.

Do not write $10 \leq x \leq 20$. C++ does not interpret it as the mathematical chained comparison. It compares $10 \leq x$ first, producing a boolean value, and then compares that value with 20. Write the two comparisons explicitly.

Outside the interval

A value is outside $[10, 20]$ when it is below 10 *or* above 20:

```
bool outside = (x < 10 || x > 20);
```

This fragment assumes `x` has already been declared. Requiring `x < 10 && x > 20` would be impossible: the same number cannot lie on both sides of the interval at once.

For two equalities, repeat the variable: `choice == 'A' || choice == 'B'`. Do not write `choice == 'A' || 'B'`. The second part must be a complete question, not a character standing on its own.

NOT and parentheses

If `inside` is a boolean variable, `!inside` means “not inside.” You can also negate a whole expression:

```
bool notInRange = !(x >= 10 && x <= 20);
```

Use parentheses to show what is being negated. `!(a == b)` means the same as `a != b`. More generally, “not (A and B)” means “not A or not B,” and “not (A or B)” means “not A and not B.” Try the truth table to verify those rules.

Short-circuit evaluation

For built-in `&&`, C++ checks the left side first. If it is false, the right side is not evaluated.^[6] For built-in `||`, if the left side is true, the right side is not evaluated.^[12]

Inside `main`, with already-declared integers `n` and `k`:

```
if (k != 0 && n % k == 0) {  
    std::cout << "DIVISIBLE" << '\n';  
}
```

If `k` is zero, the first comparison is false, so `n % k` is never evaluated. The order is important. Reversing the conditions would attempt the unsafe remainder first. Assume the integer values otherwise allow the remainder operation.

Prefer obvious conditions to clever ones

Use `&&` and `||` for logic. The single-character operators `&` and `|` are different operators, not shorter spellings of the same thing. Bitwise operations are outside this session.

Pause and predict – 7

To accept a score from 0 through 100, should you use `score >= 0 && score <= 100` or `score >= 0 || score <= 100`?

Check your prediction. Use AND. With OR, a score of 150 would pass because it is at least 0. You need both limits to hold.

1.8 Other ways to choose an alternative

Optional extension

You can solve every core exercise with `if` and `else`. Learn these alternatives after the basic conditions feel comfortable.

Switch: choose from exact values

A `switch` is useful when one integer or character is compared with several fixed values. The following example reads one character: A for addition or S for subtraction.

Complete program – a small command menu

```
#include <iostream>

int main() {
    char choice = ' ';
    std::cin >> choice;
    switch (choice) {
        case 'A':
            std::cout << "Add" << '\n';
            break;
        case 'S':
            std::cout << "Subtract" << '\n';
            break;
        default:
            std::cout << "Unknown" << '\n';
    }
    return 0;
}
```

The matching `case` is the starting point. `break` leaves the switch. The `default` branch handles input that does not match either listed case. Without a suitable `break`, execution can continue into the following case; this is called **fallthrough**.^[8]

Use `switch` for exact choices such as 'A' or 2, not range tests such as `score >= 50`. It does not directly switch on a `std::string` or a `double`. An `if` chain is often clearer for compound conditions or ranges.

The conditional operator chooses a value

Inside `main`, after declaring integers `a` and `b`:

```
int larger = (a > b) ? a : b;
```

Read this as: if `a > b` is true, use `a`; otherwise, use `b`. The expression has the form *condition ? value-if-true : value-if-false*.

The same idea with ordinary conditions is:

```
int larger = 0;
if (a > b) {
    larger = a;
} else {
    larger = b;
}
```

These are two alternative fragments, not code to paste into the same block together. Prefer the longer version when it is easier to understand. Avoid nesting several conditional operators while learning.

Pause and predict – 8

Why does the menu example use `break` after printing `Add`?

Check your prediction. The `break` exits the switch. Otherwise, execution would continue into the later statements and could print additional messages. Matching a case does not automatically stop execution at the next case label.

1.9 For loops: repeating a known pattern

A **loop** repeats a block of instructions. One execution of its body is an **iteration**. A **counter** often records which iteration is taking place.

Suppose we want to print 1, 2, and 3. We could write three output statements. But if the task asks for the numbers from 1 to an input value n , we need a method that adapts to the input.

Complete program – a first for loop

```
#include <iostream>

int main() {
    for (int i = 1; i <= 3; i++) {
        std::cout << i << '\n';
    }
    return 0;
}
```

```
1
2
3
```

The three parts of a for header

In `for (int i = 1; i <= 3; i++)`, the semicolons separate three parts:

1. **Initialization:** `int i = 1` runs once at the beginning.
2. **Condition:** `i <= 3` is checked before every possible iteration. If it is false, the loop ends.
3. **Update:** `i++` runs after each completed iteration of the body.

The execution order is initialization, condition, body, update, condition, body, update, and so on. The update is *not* performed before the first body execution.^[7]

Trace the loop instead of guessing

i	i <= 3	Action	After update
1	true	Print 1.	2
2	true	Print 2.	3
3	true	Print 3.	4
4	false	Exit; do not print 4.	No update.

The condition is checked four times, but the body runs only three times. The final failed check is what stops the loop.

Two common counting styles

Inside `main`, after reading a nonnegative integer `n`:

```
for (int i = 1; i <= n; i++) {
    std::cout << i << '\n';
}
```

This visits 1, 2, ..., `n`, so it prints the actual numbers from 1 to `n`.

```
for (int i = 0; i < n; i++) {
    std::cout << "Hello" << '\n';
}
```

This visits 0, 1, ..., `n - 1`, so it performs the body exactly `n` times. The two styles both repeat `n` times, but the counter values differ. Later, zero-based counting will be useful with arrays.

When `n` is zero, both conditions are false on the first check, so neither loop body runs. A `for` loop is allowed to run zero times.

Counting backward and changing the step

Complete program – count down to one

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    for (int i = n; i >= 1; i--) {
        std::cout << i << '\n';
    }
    return 0;
}
```

Sample input

3

Sample output

3
2
1

The counter starts at `n`, decreases by one, and stops when it is no longer at least 1. A decreasing loop needs a condition and update that agree with that direction.

To visit positive even numbers up to `n`, an alternative is:

```
for (int i = 2; i <= n; i += 2) {
    std::cout << i << '\n';
}
```

For $n = 7$, the counter values are 2, 4, and 6. The next update produces 8, which fails the condition. We use small bounded counters in this chapter so that the final update also fits the type.

Scope of a loop counter

A variable declared in the **for** initializer is available in that loop's condition, update, and body, but not after the loop. It is fine for a later, separate loop to declare its own `i`.

If a total is needed after the loop, declare the total before the loop. If you declare and initialize it inside the loop, it will start over on every iteration instead of remembering previous work.

Off-by-one errors

An **off-by-one error** means a loop performs one repetition too many or too few, or includes the wrong endpoint. With `i = 0`, the condition `i <= n` repeats $n + 1$ times. With `i = 1`, the condition `i < n` omits n .

Write the intended counter values on paper before writing the header. For the values 1, 2, 3, 4, the natural header is `for (int i = 1; i <= 4; i++)`. Check the first value, the last value, and the value that makes the loop stop.

Pause and predict – 9

What are the counter values in `for (int i = 0; i < 3; i++)`? What changes if `<` becomes `<=`?

Check your prediction. The original values are 0, 1, and 2: three iterations. With `<=`, they are 0, 1, 2, and 3: four iterations.

1.10 While and do-while loops

While: continue while a condition is true

A **while** loop checks its condition before each iteration. It is often useful when the number of repetitions is controlled by changing data rather than a simple fixed count. It can also express the same counting pattern as a **for** loop.

Complete program – the same count with while

```
#include <iostream>

int main() {
    int i = 1;
    while (i <= 3) {
        std::cout << i << '\n';
        i++;
    }
    return 0;
}
```

The initialization is before the loop, and the update is inside the body. With **i = 1**, the body prints 1, 2, and 3. When **i** becomes 4, the condition is false and execution continues after the loop.

Make sure the loop can stop

If the update **i++** is removed, the value stays at 1. The condition **i <= 3** stays true, so the program keeps printing 1. This is an **infinite loop** caused by failing to make progress.

For every loop, ask: “What changes, and why will that eventually make the condition false or trigger an exit?” A loop that waits for a special input can stop because the next input changes the condition.

A sentinel-controlled loop

A **sentinel** is a special value that marks the end of input data. Suppose a task gives integers ending with 0, asks for the sum before that 0, and guarantees the ending 0 is present. Assume at most 10^5 values, each between -10^9 and 10^9 .

Complete program – sum until zero

```
#include <iostream>

int main() {
    long long sum = 0;
    int x = 0;
    std::cin >> x;
    while (x != 0) {
        sum += x;
        std::cin >> x;
    }
    std::cout << sum << '\n';
    return 0;
}
```

Sample input

4 7 -2 0

Sample output

9

The first read happens before the loop so that there is a value to test. The next read happens inside the body to prepare for the next check. The sentinel is not included: when `x` becomes zero, the body is skipped.

Trace: read 4, add it; read 7, add it; read -2, add it; read 0, stop. If the first value is 0, there is nothing to add, and the output is 0.

Do not invent an input format

Use a sentinel loop only when the statement defines a sentinel. A list that begins with a count `n`, a list ending with zero, and a list continuing until end-of-file are three different formats.

Do-while: run once, then check

A **do-while** loop executes the body first and checks its condition afterward. Therefore, it runs at least once. A **while** loop can run zero times.^[7]

Complete program – one execution even with a false check

```
#include <iostream>

int main() {
    int i = 5;
    do {
        std::cout << i << '\n';
        i++;
    } while (i <= 3);
    return 0;
}
```

This program prints 5 once. Only afterward does it check `i <= 3`, which is false after `i` has become 6. Notice the required semicolon after `while (i <= 3);`. This is different from an ordinary **while** header.

How do you choose a loop?

Use **for** when an initialization, a counter test, and an update describe the repetition clearly. Use **while** when you want to emphasize “keep going while this situation holds.” Use **do-while** when the body must happen once before the first check.

The choice is mainly about clarity, not magic performance differences. More than one loop form can solve the same problem. Choose the one whose start, continuation rule, and stopping behavior you can explain.

Pause and predict – 10

If `i = 5`, how many times does `while (i <= 3)` execute its body? How many times does the do-while example above execute its body?

Check your prediction. The **while** body runs zero times because the first check is false. The **do-while** body runs once because the check happens after the body.

1.11 Break and continue

Break exits the nearest enclosing loop

Sometimes you know the answer before a loop reaches its normal stopping condition. A **break** ends the nearest enclosing loop or switch immediately. It does not end the entire program.^[9]

Complete program – stop when the counter reaches four

```
#include <iostream>

int main() {
    for (int i = 1; i <= 6; i++) {
        if (i == 4) {
            break;
        }
        std::cout << i << '\n';
    }
    std::cout << "DONE" << '\n';
    return 0;
}
```

```
1
2
3
DONE
```

When *i* is 4, the program reaches **break** before the output statement in the body. Therefore, 4 is not printed. Execution continues after the loop, which is why **DONE** is printed.

Continue skips the rest of this iteration

A **continue** ends the current iteration, not the whole loop. In a **for** loop, control proceeds to the update and then the next condition check. In a **while** loop, it proceeds directly to the next condition check.^[9]

Complete program – skip the even counter values

```
#include <iostream>

int main() {
    for (int i = 1; i <= 6; i++) {
        if (i % 2 == 0) {
            continue;
        }
        std::cout << i << '\n';
    }
    return 0;
}
```

This prints 1, 3, and 5 on separate lines. For an even value of `i`, `continue` skips the output statement, but the `for` update still increments the counter.

A `continue` can accidentally skip your update

Deliberately incorrect fragment – do not copy:

```
int i = 1;
while (i <= 5) {
    if (i == 3) {
        continue; // i never changes after reaching 3.
    }
    i++;
}
```

Once `i` reaches 3, the update below `continue` is never reached. The loop gets stuck. A clear correction is to put the update where it cannot be skipped, or to use a `for` loop whose update happens after `continue`.

Break, continue, and return are not interchangeable

`break` leaves the nearest loop or switch. `continue` moves to the next iteration of the nearest loop. `return 0;` inside `main` ends the program's work in `main` entirely.^[9]

Do not stop reading a fixed-length input list early just because you found an answer. In a multi-test-case problem, the unread values may then be mistaken for the next case's input. You can record the answer in a boolean flag and keep consuming the required values.

Pause and predict – 11

In the first example, what changes if `break` becomes `continue`?

Check your prediction. Only the iteration with `i == 4` is skipped. The program prints 1, 2, 3, 5, 6, and then DONE, each on a separate line.

1.12 Loop patterns you will reuse

Many beginner problems share the same structure. Rather than memorize whole solutions, learn what each pattern remembers and why its initial value makes sense.

Pattern 1: accumulate a sum

To sum the integers from 1 to n , start at zero and add each next integer. Assume $0 \leq n \leq 10^6$.

Complete program – sum from one to n

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    long long sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    std::cout << sum << '\n';
    return 0;
}
```

Sample input

4

Sample output

10

i	Sum before	Update	Sum after
1	0	$0 + 1$	1
2	1	$1 + 2$	3
3	3	$3 + 3$	6
4	6	$6 + 4$	10

The total is declared *outside* the loop so it survives between iterations. It is printed *after* the loop because the task asks for the final answer, not every intermediate total.

Why it works. After processing a value i , `sum` equals the sum of every integer from 1 through i . At the end, the last processed value is n , so the total is the required answer. For $n = 0$, no additions occur and the sum remains zero.

This kind of statement about what remains true as a loop progresses is called a **loop invariant**. You do not need a formal proof format yet; being able to explain what your variables mean after each iteration is already valuable.

Pattern 2: count matching values

A **sum** adds the values themselves. A **count** adds one whenever a value meets a condition. Suppose a task gives $0 \leq n \leq 10^5$, followed by n integers, each between -10^9 and 10^9 .

Complete program – how many input values are even?

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    int count = 0;
    for (int i = 0; i < n; i++) {
        int x = 0;
        std::cin >> x;
        if (x % 2 == 0) {
            count++;
        }
    }
    std::cout << count << '\n';
    return 0;
}
```

Sample input	Sample output
5	3
2 7 4 0 -3	

The first 5 is the number of data values, not one of the values to test. The even values are 2, 4, and 0. We increment **count** by one for each, so the final count is 3. We do not add 2, 4, and 0 to obtain the count.

Why it works. After reading any number of values, **count** equals the number of even values among those already read. Each new even value adds exactly one; an odd value adds nothing.

Pattern 3: remember the largest value so far

Suppose $1 \leq n \leq 10^5$ and each input value lies between -10^9 and 10^9 . Start with the first actual value as the best candidate, then compare the remaining values with it.

Complete program – maximum without an array

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    long long best = 0;
    std::cin >> best;
    for (int i = 1; i < n; i++) {
        long long x = 0;
        std::cin >> x;
        if (x > best) {
            best = x;
        }
    }
    std::cout << best << '\n';
    return 0;
}
```

Sample input

4
-8 -3 -10 -5

Sample output

-3

The first value is read before the loop. Therefore, only $n - 1$ more values remain: $i = 1; i < n$ performs exactly that many reads. If $n = 1$, the loop runs zero times and the first value is already the answer.

For the sample, **best** starts as -8 , becomes -3 , and stays -3 when -10 and -5 are read. Initializing the candidate to zero *without replacing it with the first input* would incorrectly make zero the maximum of an all-negative list.

Why it works. After each read, **best** is the largest value seen so far. Replacing it only when a larger value arrives preserves this meaning. To find a minimum instead, replace the comparison with $x < \text{best}$.

Pattern 4: accumulate a product

A product starts at **one**, not zero. Multiplying by one preserves a value; multiplying by zero destroys the entire product.

The factorial of a nonnegative integer is the product $1 \times 2 \times \cdots \times n$, written $n!$. By definition, $0! = 1$. For this exercise, $0 \leq n \leq 20$.

Complete program – factorial with a bounded result

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    long long factorial = 1;
    for (int i = 1; i <= n; i++) {
        factorial *= i;
    }
    std::cout << factorial << '\n';
    return 0;
}
```

Sample input	Sample output
5	120

The running product is 1, 2, 6, 24, and 120. With $n = 0$, the body does not run and the answer stays 1. The bound 20 matters: $20!$ fits in a signed 64-bit integer, while $21!$ does not.

Optional extension: a small nonnegative integer power

To calculate a^b by repeated multiplication, start with **answer** = 1 and multiply by **a** exactly **b** times. For example, 3^4 gives $1 \rightarrow 3 \rightarrow 9 \rightarrow 27 \rightarrow 81$. This requires $b \geq 0$ and bounds that keep all products in range. Do not use $^$ for exponentiation.

Pattern 5: remember whether something happened

A **flag** is a boolean variable recording a fact. Here it records whether any negative value has appeared. Assume $0 \leq n \leq 10^5$ and valid integer data.

Complete program – did any input value contain a negative number?

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    bool found = false;
    for (int i = 0; i < n; i++) {
        int x = 0;
        std::cin >> x;
        if (x < 0) {
            found = true;
        }
    }
    if (found) {
        std::cout << "YES" << '\n';
    } else {
        std::cout << "NO" << '\n';
    }
    return 0;
}
```

Sample input

4
5 -2 7 1

Sample output

YES

Once `found` becomes true, it stays true. A later positive number does not erase an earlier negative number. We still read all `n` values, so later input remains correctly positioned.

You do not always need to store the whole list

A running sum, count, minimum, maximum, or flag can often be updated one input value at a time. This is why these problems do not require arrays. Arrays become useful when you need to keep many values for later access.

Pause and predict – 12

Why does a sum usually start at 0, a product at 1, and a maximum at the first actual input?

Check your prediction. Zero does not change a sum; one does not change a product; the first input gives a genuine maximum candidate that works even when every value is negative. Each initial value matches what the variable is meant to remember.

1.13 Nested loops: rows and columns

A **nested loop** is a loop inside another loop. The outer loop often chooses a row, and the inner loop performs the work within that row. The inner loop completes its iterations each time the outer loop reaches it.

Printing a rectangle

Suppose the input gives between 1 and 20 rows and between 1 and 20 columns. Print that rectangle using star characters.

Complete program – a rectangle of stars

```
#include <iostream>

int main() {
    int rows = 0;
    int columns = 0;
    std::cin >> rows >> columns;
    for (int row = 1; row <= rows; row++) {
        for (int column = 1; column <= columns; column++) {
            std::cout << '*';
        }
        std::cout << '\n';
    }
    return 0;
}
```

Sample input

2 3

Sample output

```
***
***
```

For row 1, the inner loop prints three stars. Then the newline ends that row. For row 2, a *new execution* of the inner loop starts its column counter at 1 and prints three more stars.

Row	Column	Action
1	1, 2, 3	Print three stars, then a newline.
2	1, 2, 3	Print three stars, then a newline.

Where the newline belongs

The newline is **inside the outer loop but outside the inner loop**. If it is placed inside the inner loop, every star starts a separate line. If it is placed after the outer loop, the rows run together on one line.

Read the indentation, then verify the braces. The braces determine which loop owns the newline; indentation only helps your eyes see that structure.

Printing a growing triangle

For a triangle with n rows, row 1 needs one star, row 2 needs two, and row n needs n . Assume $1 \leq n \leq 20$.

Complete program – let the row number determine the width

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    for (int row = 1; row <= n; row++) {
        for (int column = 1; column <= row; column++) {
            std::cout << '*';
        }
        std::cout << '\n';
    }
    return 0;
}
```

Sample input

3

Sample output

```
*
**
***
```

The inner condition is `column <= row`, not `column <= n`. That is the only change needed to make the width grow by one on each row.

Count the work before using large limits

A rectangle with 3 rows and 4 columns prints $3 \times 4 = 12$ stars. A triangle with 4 rows prints $1 + 2 + 3 + 4 = 10$ stars. The total work depends on how many times the inner body runs, not just on the outer-loop count.

We will study complexity notation in Session 3. For now, notice that very large nested loops can do far more work than a single loop. Do not assume that a short piece of source code means a small amount of repeated work.

Pause and predict – 13

For the triangle program with $n = 4$, how many stars are printed in total, and how many newline characters are printed?

Check your prediction. There are $1 + 2 + 3 + 4 = 10$ stars and 4 newlines. One newline is printed after each completed row.

1.14 Working with the digits of a number

This is an application of division, remainder, and loops. It is not a new kind of loop. We restrict the input to **nonnegative integers** so that the decimal-digit interpretation stays straightforward.

Two operations reveal one digit at a time

For a nonnegative integer n , $n \% 10$ gives its last decimal digit, and integer division $n / 10$ removes that last digit.

$$5072 \bmod 10 = 2, \quad 5072/10 = 507 \quad (\text{integer division}).$$

Repeating the process visits the digits from right to left: 2, then 7, then 0, then 5. It does not visit them in their written left-to-right order.

Sum of digits

Assume $0 \leq n \leq 10^{18}$. Keep removing the last digit and adding it to a running sum.

Complete program – sum the digits without a string

```
#include <iostream>

int main() {
    long long n = 0;
    std::cin >> n;
    long long remaining = n;
    int sum = 0;
    do {
        int digit = static_cast<int>(remaining % 10);
        sum += digit;
        remaining /= 10;
    } while (remaining > 0);
    std::cout << sum << '\n';
    return 0;
}
```

Sample input

5072

Sample output

14

The variable `remaining` starts as a copy of `n`, preserving the original input. The remainder is between 0 and 9, so converting it to `int` with `static_cast<int>` is safe here.

Leading zeros are not stored. Reading 0072 as a decimal integer stores 72. Counting all four characters in "0072" is a string-processing task for Session 2.

Remaining before	be-	Digit	Sum after	Remaining after
5072		2	2	507
507		7	9	50
50		0	9	5
5		5	14	0

After the remaining number becomes zero, every original digit has been processed, so the running sum is complete.

Why do-while helps with zero

The number 0 has one decimal digit: 0. A `while (n > 0)` loop skips its body for this input. That happens to leave a digit *sum* of zero, but it would incorrectly produce a digit *count* of zero if the count also started at zero.

A `do-while` handles the zero case naturally by processing one digit before checking whether more digits remain.

Complete program – count decimal digits, including zero

```
#include <iostream>

int main() {
    long long n = 0;
    std::cin >> n;
    int count = 0;
    do {
        count++;
        n /= 10;
    } while (n > 0);
    std::cout << count << '\n';
    return 0;
}
```

For input 5072, the answer is 4. For input 0, the answer is 1. The count increases once for every removed digit.

Pause and predict – 14

What are $730 \% 10$ and $730 / 10$? Why must the digit zero not make the whole process stop immediately?

Check your prediction. They are 0 and 73. A zero digit can occur before other digits remain. Stop according to whether the remaining number is zero, not according to whether the last digit was zero.

1.15 Handling multiple test cases

Some problems ask you to solve the same task for several independent inputs. The statement may begin with a number t , followed by t test cases. Read that count once, then run the single-case logic once per case.

Start from a working single-case solution

Our even-or-odd program already solves one case. For $1 \leq t \leq 10^4$, place its input, condition, and output inside an outer loop:

Complete program – one parity answer per test case

```
#include <iostream>

int main() {
    int t = 0;
    std::cin >> t;
    for (int test = 0; test < t; test++) {
        int n = 0;
        std::cin >> n;
        if (n % 2 == 0) {
            std::cout << "EVEN" << '\n';
        } else {
            std::cout << "ODD" << '\n';
        }
    }
    return 0;
}
```

Sample input

3
4
7
0

Sample output

EVEN
ODD
EVEN

The first 3 is the case count. It is not a number whose parity should be printed. The three actual cases are 4, 7, and 0.

Reset per-case state inside the case loop

If every case gives a list whose sum is required, declare the sum *inside* the outer loop:

```
for (int test = 0; test < t; test++) {
    int n = 0;
    std::cin >> n;
    long long sum = 0; // Fresh total for this case.
    for (int i = 0; i < n; i++) {
        long long x = 0;
        std::cin >> x;
        sum += x;
    }
    std::cout << sum << '\n';
}
```

This fragment assumes `t` has already been read and that every case's sum fits `long long`. Declaring the sum outside the outer loop without resetting it would carry one case's total into the next case.

Distinguish the two counters

The outer counter counts **test cases**. The inner counter counts **values within one case**. Their jobs are different. Naming them `test` and `i`, rather than reusing one variable, helps you keep the input structure clear.

Always read exactly the data belonging to the current case. Extra reads steal values from the next case; missing reads leave values behind. Both mistakes can make later cases look wrong even if the single-case calculation is correct.

Optional extension: read until end-of-file

A different statement may ask you to process values until input ends, without a count or sentinel. In that format, `while (std::cin >> x)` reads a value and continues only while extraction succeeds. It also stops on invalid input, not just end-of-file. Use this form only when it matches the stated input format.

Pause and predict – 15

Two cases have sums 12 and 3. A program prints 12 and then 15. What is a likely cause?

Check your prediction. The accumulator was probably not reset between cases. A sum initialized inside the test-case loop starts at zero for each case.

1.16 Worked problems: combine the building blocks

The goal of a worked solution is not to memorize its source. Notice the repeated sequence: understand the required result, choose variables, write a plan, trace an example, and test the boundary cases.

Worked problem A – checkout total

Task. Read a unit price p and quantity q , both integers between 1 and 10^9 . The item total is $p \times q$. Add a delivery fee of 50 when the item total is *less than* 1000. Print the final total.

Reasoning. First multiply, then decide whether to add the fee. The fee rule applies to the original item total. At exactly 1000, there is no fee. The largest product is 10^{18} , so both inputs and the total use `long long`.

Complete program – multiplication followed by a decision

```
#include <iostream>

int main() {
    long long price = 0;
    long long quantity = 0;
    std::cin >> price >> quantity;
    long long total = price * quantity;
    if (total < 1000) {
        total += 50;
    }
    std::cout << total << '\n';
    return 0;
}
```

Sample input

120 3

Sample output

410

Trace. The item total is $120 \times 3 = 360$. Since $360 < 1000$, add 50 to obtain 410.

Why it works. The multiplication gives the cost of all items. The condition separates the only two cases in the delivery rule. Each branch leaves the required final total in the same variable.

Tests. 999 1 gives 1049. 100 10 gives 1000. 1001 1 gives 1001. These tests make the exact threshold visible.

Worked problem B – split a duration

Task. Given $0 \leq s \leq 10^9$ seconds, print the number of complete hours, then remaining complete minutes, then remaining seconds. Print three integers separated by spaces. Hours are not restricted to 0–23 because this is a duration, not a clock time.

Reasoning. One hour contains 3600 seconds, so $s / 3600$ gives the complete hours. The remainder $s \% 3600$ is the part that still needs to be split into minutes and seconds.

Complete program – quotient and remainder in two stages

```
#include <iostream>

int main() {
    long long seconds = 0;
    std::cin >> seconds;
    long long hours = seconds / 3600;
    long long remaining = seconds % 3600;
    long long minutes = remaining / 60;
    long long lastSeconds = remaining % 60;
    std::cout << hours << ' ' << minutes << ' ';
    std::cout << lastSeconds << '\n';
    return 0;
}
```

Sample input

3671

Sample output

1 1 11

Trace. $3671/3600 = 1$ hour, with 71 seconds left. $71/60 = 1$ minute, with 11 seconds left.

Why it works. Division extracts the complete units; remainder preserves precisely what those complete units did not use. After both stages, the printed components add back to the original duration:

$$3600 \times \text{hours} + 60 \times \text{minutes} + \text{lastSeconds} = s.$$

Tests. 0 gives 0 0 0; 59 gives 0 0 59; 60 gives 0 1 0; 3600 gives 1 0 0. Try 86400 as well: the answer is 24 0 0, not 0 0 0.

A useful debugging check

When you split a quantity into units, reconstruct the original quantity from the output. If the reconstruction is wrong, a quotient or remainder probably used the wrong value.

Worked problem C – classify three side lengths

Task. Read three positive integer side lengths $a, b, c \leq 10^9$. Print **INVALID** if they cannot form a nondegenerate triangle. Otherwise print **EQUILATERAL** if all are equal, **ISOSCELES** if exactly two are equal, and **SCALENE** if all are different.

Given mathematical rule. A valid triangle requires each pair of side lengths to have a sum strictly greater than the remaining side. Equality gives a flattened shape, not a nondegenerate triangle. We use this rule as part of the problem specification.

Reasoning. Check validity first. Among valid triangles, test all-equal before testing any equal pair. That ensures the categories are exclusive even though an all-equal triangle also contains equal pairs.

Complete program – validation followed by classification

```
#include <iostream>

int main() {
    long long a = 0;
    long long b = 0;
    long long c = 0;
    std::cin >> a >> b >> c;
    if (a + b > c && a + c > b && b + c > a) {
        if (a == b && b == c) {
            std::cout << "EQUILATERAL" << '\n';
        } else if (a == b || a == c || b == c) {
            std::cout << "ISOSCELES" << '\n';
        } else {
            std::cout << "SCALENE" << '\n';
        }
    } else {
        std::cout << "INVALID" << '\n';
    }
    return 0;
}
```

Sample input

3 4 5

Sample output

SCALENE

Why it works. The outer condition enforces all three required inequalities. Inside it, the ordered equality tests distinguish all-equal, exactly-two-equal, and all-different cases. Positive input bounds are guaranteed, so the program does not need a separate positivity test.

Tests. 2 2 2 is equilateral; 3 3 5 is isosceles; 1 2 3 is invalid. Also test a repeated pair in a different order, such as 5 3 3.

Worked problem D – a class score report

Task. Read $1 \leq n \leq 10^5$, followed by n integer scores from 0 to 100. Print the total and the number of passing scores on the first line. A score passes when it is at least 50. On the second line, print the average with two digits after the decimal point.

Reasoning. Each score contributes to the sum. Only passing scores contribute one to the count. After reading everything, divide the sum by n using floating-point division. The guarantee $n \geq 1$ makes the divisor nonzero.

Complete program – sum, count, and average together

```
#include <iostream>
#include <iomanip>

int main() {
    int n = 0;
    std::cin >> n;
    long long sum = 0;
    int passed = 0;
    for (int i = 0; i < n; i++) {
        int score = 0;
        std::cin >> score;
        sum += score;
        if (score >= 50) {
            passed++;
        }
    }
    double average = static_cast<double>(sum) / n;
    std::cout << sum << ' ' << passed << '\n';
    std::cout << std::fixed << std::setprecision(2);
    std::cout << average << '\n';
    return 0;
}
```

Sample input

4
40 50 80 90

Sample output

260 3
65.00

Trace. After the four scores, the sum progresses through 40, 90, 170, and 260. The passing count progresses through 0, 1, 2, and 3. The average is $260/4 = 65$.

Why it works. The sum and count patterns maintain their separate meanings while processing the same input stream. Reading a score once is enough to update both. No array is needed.

Tests. One score of 0 gives total 0, count 0, average 0.00. One score of 50 gives total 50, count 1, average 50.00. Try scores whose average is fractional, such as 50 and 51.

Optional worked problem E – is a number prime?

Task. Given $0 \leq n \leq 1000$, print YES if n is prime and NO otherwise. A prime number is an integer at least 2 whose only positive divisors are 1 and itself.

Reasoning. Numbers below 2 are not prime. For any other number, look for a divisor from 2 through $n - 1$. Finding one proves the answer is no, so further checks are unnecessary.

Complete program – a flag and an early exit

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    bool prime = (n >= 2);
    for (int divisor = 2; divisor < n; divisor++) {
        if (n % divisor == 0) {
            prime = false;
            break;
        }
    }
    if (prime) {
        std::cout << "YES" << '\n';
    } else {
        std::cout << "NO" << '\n';
    }
    return 0;
}
```

Sample input

17

Sample output

YES

Trace. For $n = 9$, divisor 2 does not work, but divisor 3 does. The flag becomes false and the loop exits. For $n = 2$, the loop body never runs, and the initial true flag correctly remains true.

Why it works. The loop checks every possible positive divisor other than 1 and n . If none divides n , the definition is satisfied. The initial $n \geq 2$ condition handles 0 and 1 separately.

Limit of this approach. The intentionally simple loop is suitable for the small bound given here. It is not an efficient plan for extremely large input values. We will discuss how to compare algorithms later; do not increase a problem's constraints without reconsidering the method.

1.17 Debugging and testing your solution

Debugging means finding and correcting the cause of a program's incorrect behavior. Do not change random lines until the output happens to look right. Try to find the first point where the program's actual state differs from your intended state.

Recognize the kind of problem

Compilation error. The source is not accepted. Common causes include missing semicolons, unmatched braces, misspelled names, undeclared variables, and using a name outside its scope. Read the first relevant message, fix the cause, and compile again.

Invalid operation or runtime failure. The program starts but encounters an operation it is not allowed to perform, such as integer division by zero. Invalid behavior is not guaranteed to produce a clean error message or a crash. Prevent it using constraints and appropriate checks.

Wrong answer. The program runs and prints something, but the answer or its format is wrong. Check the algorithm, boundary conditions, initialization, integer division, overflow, and reading order.

Too much time or endless output. A loop may not be making progress, or it may be doing far more work than the input limit permits. Trace a small input and count the repetitions.

Use a trace table

Consider this fragment:

```
int total = 0;
for (int i = 1; i <= 4; i++) {
    if (i % 2 == 0) {
        total += i;
    }
}
std::cout << total << '\n';
```

i	Even?	total after this iteration
1	No	0
2	Yes	2
3	No	2
4	Yes	6

The program prints 6. Writing only the final output is less informative than showing *why* the total changes at each iteration.

Debugging messages are temporary

While testing locally, you can inspect values:

```
std::cerr << "i=" << i << " total=" << total << '\n';
```

This fragment assumes `i` and `total` exist at that location, for example inside the loop. `std::cerr` is a diagnostic stream, separate from ordinary `std::cout`. Some tools display both in the same console. Remove temporary diagnostics before submitting; do not depend on how a particular platform displays them.

Build a useful test set

Start with the statement's sample, then add a case that checks each important choice in your code. For a passing threshold of 50, use 49, 50, and 51. For a maximum problem, use all-negative values, equal values, and a list of length one. For digit processing, include 0 and a value with internal zeros.

Case to try	What it can reveal
Smallest allowed input	Wrong initialization or zero-iteration behavior.
Largest allowed input	Overflow or excessive repeated work.
Exact threshold	Incorrect <code><</code> versus <code><=</code> .
One value only	Extra reads or incorrect loop bounds.
All negative values	A maximum incorrectly initialized to zero.
Two consecutive cases	State carried from one case into the next.

Fix the cause, then keep the failing test

Suppose the task is to print the sum from 1 to `n`, but your program prints several values. Check whether the output statement is inside the loop. Moving it after the loop fixes the placement of the output, not the arithmetic.

If your maximum program prints zero for `-8 -3 -5`, the comparisons may be fine. The initial candidate is the problem. Start from a real input value.

After fixing a bug, rerun the input that revealed it, the sample, and at least one boundary case. A change can solve one case and break another.

A short debugging exercise

Intended task: read $1 \leq n \leq 10^5$ and print $1 + 2 + \dots + n$. **Deliberately incorrect fragment:**

```
int n = 0;
std::cin >> n;
for (int i = 1; i < n; i++) {
    long long sum = 0;
    sum += i;
    std::cout << sum << '\n';
}
```

Find three independent mistakes before reading the explanation.

Explained correction. The condition should include `n`, so use `i <= n`. Initialize `sum` before the loop so earlier additions are remembered. Print after the loop so only the final total appears. Compare your correction with `examples/20_sum_to_n.cpp`.

Before you submit

Check that the program matches the input format, prints only the requested output, initializes every accumulator, handles boundaries, uses safe arithmetic types, and stops every loop correctly. A program that compiles has passed a syntax check, not a correctness proof.

1.18 Practice problems

Try these in order. Each problem is independent and needs only ideas already taught in this chapter. Use `if` instead of `switch` or the conditional operator whenever it is clearer to you. Unless a problem explicitly says otherwise, input is valid and there is **one** test case.

Write your idea in one or two sentences before coding. Hints are in Section 1.19; explained solutions are in Section 1.20. Use them after a genuine attempt.

Stage A – input and arithmetic

P01. Sum and product

Read two integers $-10^9 \leq a, b \leq 10^9$. Print their sum and their product, in that order, separated by one space. Solution 1.20, P01

Sample input	Sample output
4 7	11 28

P02. Rectangle measurements

Read integer length and width, each from 1 to 10^9 . Print the area, then the perimeter, separated by a space. Use $\text{area} = \text{length} \times \text{width}$ and $\text{perimeter} = 2(\text{length} + \text{width})$.

Sample input	Sample output
5 3	15 16

P03. Average of three scores

Read three integer scores from 0 to 100. Print their average with exactly two digits after the decimal point.

Sample input	Sample output
10 11 13	11.33

P04. The last two digits

Read $0 \leq n \leq 10^9$. Print its tens digit and ones digit, separated by a space. For a one-digit number, the tens digit is 0. For $n = 0$, print 0 0.

Sample input	Sample output
5072	7 2

Stage B – decisions**P05. Maximum of three integers**

Read three integers, each between -10^9 and 10^9 . Print the largest value. Equal values are allowed.

Sample input	Sample output
-8 -3 -5	-3

P06. Sign and parity

Read $-10^9 \leq n \leq 10^9$. On the first line, print POSITIVE, NEGATIVE, or ZERO. On the second, print EVEN or ODD. Remember that zero is even.

Sample input	Sample output
-7	NEGATIVE ODD

P07. Inside the interval?

Read three integers `left`, `right`, and `x`, each between -10^9 and 10^9 , with `left` $\leq$ `right`. Print YES if `x` belongs to the inclusive interval from `left` to `right`; otherwise print NO.

Sample input	Sample output
10 20 20	YES

P08. A small integer calculator

Read an integer `a`, an operator character, and an integer `b`, in that order. Both integers lie between -10^6 and 10^6 . The operator is one of `+`, `-`, `*`, or `/`. Print the result, using C++ integer division for `/`. If division by zero is requested, print INVALID instead.

Sample input	Sample output
7 / 2	3

Stage C – repetition**P09. Print one through n**

Read $0 \leq n \leq 1000$. Print every integer from 1 through n , in increasing order, one per line. If $n = 0$, print nothing.

Sample input	Sample output
3	1 2 3

P10. Sum the positive even integers

Read $0 \leq n \leq 10^6$. Print the sum of all positive even integers no larger than n . Print 0 if there are none.

Sample input	Sample output
7	12

P11. Count signs in a list

Read $1 \leq n \leq 10^5$, then n integers from -10^9 to 10^9 . Print three counts separated by spaces: the number of positive values, the number of negative values, and the number of zeros, in that order.

Sample input	Sample output
6 -2 0 5 -1 0 8	2 2 2

P12. Maximum of a list

Read $1 \leq n \leq 10^5$, then n integers from -10^9 to 10^9 . Print the largest value. Solve it without storing the entire list.

Sample input	Sample output
4 -8 -3 -10 -5	-3

Stage D – combine the ideas**P13. Factorial**

Read $0 \leq n \leq 20$. Print $n!$, the product $1 \times 2 \times \cdots \times n$. For zero, print 1.

Sample input	Sample output
5	120

P14. Sum of decimal digits

Read $0 \leq n \leq 10^{18}$. Print the sum of its decimal digits. For zero, print 0. Do not use a string or array.

Sample input	Sample output
5072	14

P15. A growing triangle

Read $1 \leq n \leq 20$. Print n rows. Row 1 contains one star, row 2 two stars, and so on. Use no spaces between stars.

Sample input	Sample output
3	* ** ***

P16. Even counts in several cases

Read $1 \leq t \leq 100$. Each case starts with $1 \leq n \leq 10^5$, followed by n integers from -10^9 to 10^9 . The total number of integers across all cases is at most 10^5 . For each case, print the number of even values on a separate line.

Sample input	Sample output
2	2
3	0
2 3 4	
2	
1 5	

A useful standard for finishing a problem

You are finished when you can explain the idea, trace the code, justify the chosen types, and name a boundary case your solution handles. An accepted-looking sample output alone is not enough.

1.19 Hints before solutions

Read only the hint for the problem you are working on. Then close this page and try again. A hint should help you take the next step, not replace your attempt.

Arithmetic and decisions

P01. Estimate the largest product before choosing the types of *both operands*. You do not need a loop.

P02. Use separate variables for the area and perimeter. Parentheses matter in the perimeter formula.

P03. Is the division performed before the result is stored in a `double`? Make at least one operand floating-point.

P04. Remainder by 10 extracts a last digit. Divide by 10 once before extracting the tens digit.

P05. Start with one of the three values as your candidate. Let the second and third values replace it when appropriate.

P06. Sign and parity are independent questions. A sign decision should not prevent you from also printing parity.

P07. Both interval limits must hold. Check the endpoints, not just a value in the middle.

P08. Compare the operator with single-quoted characters. Handle a zero divisor before performing division.

Loops and combined problems

P09. Write the desired counter values for `n = 3`. Check what the first condition does for `n = 0`.

P10. Use a running sum. Either test each integer for evenness or visit only even counter values.

P11. Use three counters starting at zero. Every input value belongs to exactly one sign category.

P12. Your initial candidate must still work for an all-negative list. After reading the first value, how many reads remain?

P13. What starting value preserves a product? What should happen when there are no factors to multiply?

P14. Keep taking the last digit and removing it. Check the special input zero and a number containing zero digits.

P15. The row number tells you how many times the inner loop should print a star. Print the newline after that inner loop.

P16. Use one loop for cases and one loop for values. The even counter belongs inside the case loop, but outside the value loop.

1.20 Explained practice solutions

These are reference solutions, not the only possible solutions. Each page explains the idea and includes a complete C++23 program. Type your own version after understanding the explanation; do not make learning depend on copying exactly the same variable names.

Problem	Main idea	Page
P01	Safe sum and product	56
P02	Translate two formulas	57
P03	Floating-point division and formatting	58
P04	Division and remainder	59
P05	Update a best candidate	60
P06	Two independent classifications	61
P07	Combine inclusive comparisons	62
P08	Select one operation safely	63
P09	A bounded counter	64
P10	Conditional accumulation	65
P11	Three exclusive categories	66
P12	A maximum over streamed input	67
P13	Product initialization	68
P14	Process one digit at a time	69
P15	A row-dependent inner loop	70
P16	Independent state for each test case	71

How to learn from a solution

Ask what each variable represents, why its initial value is correct, and where it changes. For a loop, identify the first value, the stopping condition, and the final state. For a condition, explain why each possible input reaches the right branch.

Then change one detail of the task. For example, turn a maximum into a minimum, count odd values instead of even values, or print a decreasing triangle. Changing the requirement tests whether you understood the idea.

Solution 1 – Sum and product

Idea. Both operations use the same two inputs. Read them as `long long`, so the multiplication itself is wide enough. The maximum product magnitude is 10^{18} , which fits in a signed 64-bit integer. The sum is smaller, so it also fits.

Complete program – P01

```
#include <iostream>

int main() {
    long long a = 0;
    long long b = 0;
    std::cin >> a >> b;
    std::cout << a + b << ' ' << a * b << '\n';
    return 0;
}
```

Why it works. The program prints exactly the two operations requested, in the requested order. For input 4 and 7, the addition is 11 and the multiplication is 28. Since both operands are already `long long`, no extra `1LL` is needed.

Check these cases

Input	Output
4 7	11 28
0 -8	-8 0
-3 -2	-5 6

A variable named `product` is optional. You may print an expression directly when that makes the solution clearer.

Solution 2 – Rectangle measurements

Idea. Translate each formula separately. The parentheses in `2 * (length + width)` ensure that the sum of the two dimensions is doubled. Reading both dimensions as `long long` keeps both the area and perimeter arithmetic safe for the given bounds.

Complete program – P02

```
#include <iostream>

int main() {
    long long length = 0;
    long long width = 0;
    std::cin >> length >> width;
    long long area = length * width;
    long long perimeter = 2 * (length + width);
    std::cout << area << ' ' << perimeter << '\n';
    return 0;
}
```

Why it works. The area counts the unit squares in a rectangle. Its perimeter includes two sides of each length, giving $2(l + w)$. For $l = 5, w = 3$, these produce 15 and 16. The largest allowed perimeter is 4×10^9 , which is too large for a signed 32-bit `int`.

Check these cases

Input	Output
5 3	15 16
1 1	1 4
2 7	14 18

Check both formulas. A type that is large enough for one output is not automatically large enough for every other output.

Solution 3 – Average of three scores

Idea. Add the three scores, then divide by 3.0. The fractional literal makes the division floating-point. The sum is at most 300, so integer addition is safe before that division. Use `fixed` and `setprecision(2)` to print two decimal places.

Complete program – P03

```
#include <iostream>
#include <iomanip>

int main() {
    int a = 0;
    int b = 0;
    int c = 0;
    std::cin >> a >> b >> c;
    double average = (a + b + c) / 3.0;
    std::cout << std::fixed << std::setprecision(2);
    std::cout << average << '\n';
    return 0;
}
```

Why it works. The arithmetic mean is the sum divided by the number of scores, which is three. For 10, 11, and 13, the sum is 34 and the mean is approximately 11.3333. Formatting displays 11.33. It does not turn the stored number into an exact decimal fraction.

Check these cases

Input	Output
10 11 13	11.33
0 0 0	0.00
100 100 100	100.00

Changing the destination to `double` would not fix `(a + b + c) / 3`; the operands of the division determine how it is performed.

Solution 4 – The last two digits

Idea. The ones digit is the remainder after division by 10. Remove that digit with integer division, then take another remainder to obtain the tens digit. The task asks for tens first and ones second, even though extracting ones first is convenient.

Complete program – P04

```
#include <iostream>

int main() {
    long long n = 0;
    std::cin >> n;
    long long ones = n % 10;
    long long tens = (n / 10) % 10;
    std::cout << tens << ' ' << ones << '\n';
    return 0;
}
```

Why it works. For 5072, the ones digit is 2. Dividing by 10 gives 507, whose last digit is 7. For a one-digit input such as 5, division by 10 gives 0, so the tens digit naturally becomes 0.

Check these cases

Input	Output
5072	7 2
5	0 5
0	0 0

The expressions are written for nonnegative input, as required by the statement. Always check that a digit trick matches its input assumptions.

Solution 5 – Maximum of three integers

Idea. Start with **a** as the best candidate. Compare **b** with that candidate and replace it if necessary. Then do the same with **c**. The two comparisons must be independent: a larger **b** does not mean **c** can be ignored.

Complete program – P05

```
#include <iostream>

int main() {
    long long a = 0;
    long long b = 0;
    long long c = 0;
    std::cin >> a >> b >> c;
    long long best = a;
    if (b > best) {
        best = b;
    }
    if (c > best) {
        best = c;
    }
    std::cout << best << '\n';
    return 0;
}
```

Why it works. After the first comparison, **best** is the maximum of **a** and **b**. After the second, it is the maximum of all three. Equal values do not need an update because the candidate is already correct. Starting from a real input also handles all-negative triples.

Check these cases

Input	Output
-8 -3 -5	-3
4 4 2	4
1 7 9	9

Trace 1 7 9. Replacing the second **if** with **else if** would skip checking 9 after updating the candidate to 7.

Solution 6 – Sign and parity

Idea. Use one exclusive chain for positive, negative, and zero. After that chain finishes, perform a separate even/odd decision. Every input has one sign category and one parity category, so exactly two output lines are required.

Complete program – P06

```
#include <iostream>

int main() {
    long long n = 0;
    std::cin >> n;
    if (n > 0) {
        std::cout << "POSITIVE" << '\n';
    } else if (n < 0) {
        std::cout << "NEGATIVE" << '\n';
    } else {
        std::cout << "ZERO" << '\n';
    }
    if (n % 2 == 0) {
        std::cout << "EVEN" << '\n';
    } else {
        std::cout << "ODD" << '\n';
    }
    return 0;
}
```

Why it works. The sign comparisons cover all integers without overlap. The remainder test then covers both possible parity results. A negative odd number can have remainder -1 , which is still nonzero, so the `else` branch correctly prints ODD.

Check these cases

Input	Output
-7	NEGATIVE ODD
0	ZERO EVEN
8	POSITIVE EVEN

Two properties can both need an answer. Do not force every decision in a program into one large else-if chain.

Solution 7 – Inside the interval?

Idea. A number is inside the inclusive interval only when it satisfies both limits. Use `x >= left` and `x <= right`, joined by `&&`. The input guarantees that the left endpoint does not exceed the right endpoint.

Complete program – P07

```
#include <iostream>

int main() {
    long long left = 0;
    long long right = 0;
    long long x = 0;
    std::cin >> left >> right >> x;
    if (x >= left && x <= right) {
        std::cout << "YES" << '\n';
    } else {
        std::cout << "NO" << '\n';
    }
    return 0;
}
```

Why it works. The first comparison rejects values below the interval. The second rejects values above it. If both succeed, no outside case remains. The equality signs include the endpoints, including the single-point interval where left equals right.

Check these cases

Input	Output
10 20 20	YES
10 20 21	NO
4 4 4	YES

Test the exact endpoints as well as values just outside them. These tests distinguish inclusive comparisons from strict comparisons.

Solution 8 – A small integer calculator

Idea. Read the input in its actual order: number, character, number. An else-if chain selects one operation. The division branch requires a nonzero divisor. Since all permitted operators have been covered, reaching the final branch means the requested division cannot be performed.

Complete program – P08

```
#include <iostream>

int main() {
    long long a = 0;
    long long b = 0;
    char operation = ' ';
    std::cin >> a >> operation >> b;
    if (operation == '+') {
        std::cout << a + b << '\n';
    } else if (operation == '-') {
        std::cout << a - b << '\n';
    } else if (operation == '*') {
        std::cout << a * b << '\n';
    } else if (operation == '/' && b != 0) {
        std::cout << a / b << '\n';
    } else {
        std::cout << "INVALID" << '\n';
    }
    return 0;
}
```

Why it works. For valid division, C++ truncates toward zero: $7 / 2$ is 3 and $-7 / 2$ is -3 . The input bounds also avoid the signed minimum divided by -1 case. Products of the permitted operands fit `long long`.

Check these cases

Input	Output
7 / 2	3
5 / 0	INVALID
4 * 5	20

A zero divisor is a property of the input, not something to discover by trying the division and hoping it works.

Solution 9 – Print one through n

Idea. The first required value is 1, so initialize the counter to 1. The last required value is n , so use $i \leq n$. Increasing by one produces every integer in between. Put the output inside the loop because one output is needed for every counter value.

Complete program – P09

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    for (int i = 1; i <= n; i++) {
        std::cout << i << '\n';
    }
    return 0;
}
```

Why it works. Each iteration prints exactly the current required integer. For $n = 3$, the values are 1, 2, and 3, and the first rejected counter is 4. If $n = 0$, the initial comparison is false and nothing is printed, as requested.

Check these cases

Input	Output
1	1
3	1 2 3
0	No output

Check the empty-output case intentionally. Printing the number 0 would not mean the same thing as printing nothing.

Solution 10 – Sum the positive even integers

Idea. Visit the positive integers from 1 to n . Add a value only when its remainder by 2 is zero. The sum can be much larger than the counter, so use a `long long` accumulator. The counter can remain an `int` under the stated bounds.

Complete program – P10

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    long long sum = 0;
    for (int i = 1; i <= n; i++) {
        if (i % 2 == 0) {
            sum += i;
        }
    }
    std::cout << sum << '\n';
    return 0;
}
```

Why it works. After processing each counter, the sum contains exactly the even values seen so far. For $n = 7$, it adds 2, 4, and 6, giving 12. If there are no positive even values, the initial zero is already the correct answer.

Check these cases

Input	Output
7	12
1	0
6	12

An equally valid loop starts at 2 and uses `i += 2`. You can explain either method using the same running-sum idea.

Solution 11 – Count signs in a list

Idea. Create three counters, all initially zero. Read one value per iteration, determine its sign, and increment exactly one counter. The maximum of any count is n , which is at most 10^5 , so `int` is sufficient in the tested environment.

Complete program – P11

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    int positive = 0;
    int negative = 0;
    int zero = 0;
    for (int i = 0; i < n; i++) {
        int x = 0;
        std::cin >> x;
        if (x > 0) {
            positive++;
        } else if (x < 0) {
            negative++;
        } else {
            zero++;
        }
    }
    std::cout << positive << ' ' << negative << ' ';
    std::cout << zero << '\n';
    return 0;
}
```

Why it works. The three branches are mutually exclusive and cover every integer. Therefore, each input contributes exactly once. As a useful check, the three final counts must add up to n . The sample contains two positives, two negatives, and two zeros.

Check these cases

Input	Output
3	1 1 1
1 -2 0	
1	0 0 1
0	
2	0 2 0
-5 -5	

The counters measure how many values occur, not their sum. Reading -5 increases the negative counter by one, not by -5 .

Solution 12 – Maximum of a list

Idea. The guarantee $n \geq 1$ lets us read one actual value before the loop. Use that value as the first maximum candidate. Then read the remaining $n - 1$ values, updating the candidate whenever a larger value appears.

Complete program – P12

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    long long best = 0;
    std::cin >> best;
    for (int i = 1; i < n; i++) {
        long long x = 0;
        std::cin >> x;
        if (x > best) {
            best = x;
        }
    }
    std::cout << best << '\n';
    return 0;
}
```

Why it works. After each processed value, the candidate equals the maximum of everything read so far. At the end, everything has been read, so the candidate is the answer. The first input is not read again, and the loop makes exactly the remaining number of reads.

Check these cases

Input	Output
4 -8 -3 -10 -5	-3
1 -8	-8
3 4 4 4	4

The initial `best = 0` only creates the variable. The immediate input operation replaces it with the first real value before any comparison is made.

Solution 13 – Factorial

Idea. Start with 1 and multiply by each integer from 1 to n . A product accumulator must not start at zero. Use `long long`; the exercise's upper bound of 20 was chosen so that every intermediate product fits a signed 64-bit integer.

Complete program – P13

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    long long answer = 1;
    for (int i = 1; i <= n; i++) {
        answer *= i;
    }
    std::cout << answer << '\n';
    return 0;
}
```

Why it works. After multiplying by i , the accumulator equals $i!$. Thus the final value is $n!$. For zero, there are no iterations, leaving 1, which matches the defined value of $0!$. For 5, the products are 1, 2, 6, 24, and 120.

Check these cases

Input	Output
0	1
1	1
5	120

At the upper bound, $20! = 2,432,902,008,176,640,000$. Increasing the bound to 21 would require rethinking the numeric representation.

Solution 14 – Sum of decimal digits

Idea. Use a do-while loop to process at least one digit. Add the remainder by 10, then remove the last digit with integer division. The remainder lies from 0 through 9, so its explicit conversion to `int` is safe. The original input is not needed afterward, so this solution updates `n` directly.

Complete program – P14

```
#include <iostream>

int main() {
    long long n = 0;
    std::cin >> n;
    int sum = 0;
    do {
        sum += static_cast<int>(n % 10);
        n /= 10;
    } while (n > 0);
    std::cout << sum << '\n';
    return 0;
}
```

Why it works. Each iteration adds exactly one decimal digit and removes exactly that digit. No digit is skipped or counted twice. When no digits remain, the sum is complete. For zero, the first iteration adds zero and stops. For 5072, the added digits are 2, 7, 0, and 5.

Check these cases

Input	Output
5072	14
0	0
1000	1

If you need to use the original input after digit processing, work on a copy as the earlier complete example did.

Solution 15 – A growing triangle

Idea. The outer loop chooses a row from 1 to n . The inner loop prints exactly as many stars as that row's number. A newline after the inner loop completes one row before the next row starts.

Complete program – P15

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    for (int row = 1; row <= n; row++) {
        for (int column = 1; column <= row; column++) {
            std::cout << '*';
        }
        std::cout << '\n';
    }
    return 0;
}
```

Why it works. For row r , the inner counter visits 1 through r , so exactly r stars are printed. The outer loop visits every required row once. For $n = 3$, the row widths are 1, 2, and 3, producing six stars in total.

Check these cases

Input	Output
1	*
2	* **
3	* ** ***

Follow the braces around the newline. Moving it into the inner loop changes the output from rows of stars to one star per line.

Solution 16 – Even counts in several cases

Idea. Use an outer loop for the test cases. Inside it, read the case length and initialize a fresh even counter. Use an inner loop to read exactly that many values. Print one count after the inner loop, while still inside the outer loop.

Complete program – P16

```
#include <iostream>

int main() {
    int t = 0;
    std::cin >> t;
    for (int test = 0; test < t; test++) {
        int n = 0;
        std::cin >> n;
        int count = 0;
        for (int i = 0; i < n; i++) {
            int x = 0;
            std::cin >> x;
            if (x % 2 == 0) {
                count++;
            }
        }
        std::cout << count << '\n';
    }
    return 0;
}
```

Why it works. Within each case, the counter always equals the number of even values read in that case. A new declaration of `count = 0` is executed at the start of each case, so previous answers do not affect later answers. Every value is consumed, preserving the input boundaries.

Check these cases

Input	Output
2	2
3	0
2 3 4	
2	
1 5	
1	1
1	
0	

The sample's second answer is zero. It is a useful test specifically because an unreset counter would incorrectly print two instead.

1.21 Quick reference and revision checklist

Use this section to *recall* ideas you have already studied. If a line feels unfamiliar, follow the section reference and read its explanation instead of memorizing syntax without meaning.

Data, arithmetic, and conditions

Need	Pattern or reminder
Whole-number count	<code>int count = 0;</code> (Section 1.4)
Large whole-number total	<code>long long sum = 0;</code>
Fractional value	<code>double average = 0.0;</code>
One character	<code>char choice = 'A';</code>
A true/false fact	<code>bool found = false;</code>
Read a value	<code>std::cin >> x;</code> (Section 1.3)
Print a value and newline	<code>std::cout << x << '\n';</code>
Floating-point division	<code>static_cast<double>(a) / b</code> , with a valid nonzero divisor.
Widen a product first	<code>1LL * a * b</code> , when the final result fits.
Evenness	<code>n % 2 == 0</code>
Oddness for signed integers	<code>n % 2 != 0</code>
Both conditions	<code>A && B</code> (Section 1.7)
At least one condition	<code>A B</code>
Negate a condition	<code>!A</code> or <code>!(expression)</code>
Inclusive interval	<code>x >= left && x <= right</code>

Read these symbols correctly

= assigns; == compares. * multiplies; ^ does not mean exponentiation. && and || are logical operators; single & and | are different operators. 'A' is one character; "A" is text.

The numeric-safety questions

What is the largest possible intermediate value? Are the *operands* wide enough before the operation begins? Do you need exact integers or a fractional result? Could a divisor be zero? Did you accidentally perform integer division before storing a `double`?

Remember the two different limits

Input limits constrain the values you read. Calculation limits constrain what happens when you add, multiply, divide, or repeat operations using those values. Check both.

Loop templates to recognize

The fragments below belong inside `main`. They assume `n` is a small enough nonnegative integer and any other used variables have been declared and initialized.

Visit 1 through `n`:

```
for (int i = 1; i <= n; i++) {
    std::cout << i << '\n';
}
```

Perform `n` reads and additions:

```
long long sum = 0;
for (int i = 0; i < n; i++) {
    long long x = 0;
    std::cin >> x;
    sum += x;
}
```

Repeat while a situation holds:

```
int i = 1;
while (i <= n) {
    std::cout << i << '\n';
    i++;
}
```

Process at least one decimal digit of a nonnegative `n`:

```
int digits = 0;
do {
    digits++;
    n /= 10;
} while (n > 0);
```

Initialization and control

A sum begins at 0. A product begins at 1. A maximum or minimum can begin at the first actual value, provided at least one value exists. A flag begins with the fact that is true before processing starts.

`break` exits the nearest loop or switch. `continue` skips to the next loop iteration; in a `for`, its update still runs. `return 0`; in `main` ends the program's work there.

A small glossary

Term	Meaning in this chapter
Algorithm	A sequence of steps that solves a task for its allowed inputs.
Compiler	A tool that checks and translates source code into a runnable program.
Constraint	A stated limit or guarantee on the input.
Declaration	Introducing a variable and its type.
Initialization	Giving a variable its first value.
Assignment	Storing a value in an existing variable.
Expression	Code that produces a value, such as <code>a + b</code> .
Condition	An expression used as a true/false question.
Block	A group of statements enclosed in braces.
Scope	The part of the program where a name can be used.
Iteration	One execution of a loop's body.
Accumulator	A variable that combines results over time, such as a sum.
Flag	A boolean variable remembering a fact.
Sentinel	A special input value that marks the end of data.
Trace	A record of the changing state as instructions are executed.
Boundary case	A case at or near a limit or decision threshold.
Overflow	A result outside the relevant representation's capacity; signed overflow is not safe to rely on.

What to do when you get stuck

Reduce the input to a tiny example. Write what each variable should mean. Trace one iteration or one branch at a time. Find the earliest difference between your plan and your program. Fix that difference, then test again.

When asking a mentor for help, include the task, your current code, the input that fails, the output you expected, and the output you actually obtained. This gives the mentor something concrete to reason about.

Readiness check: explain, do not just recognize

Try these without looking at the chapter. Then compare with the answer key below.

1. Why does `double x = 9 / 2;` not give 4.5?
2. What is the difference between `x = 5` and `x == 5`?
3. Write a condition meaning “x is from 1 through 10, inclusive.”
4. How many times does `for (int i = 0; i <= 4; i++)` run?
5. Why must a running sum usually be initialized outside its value-processing loop?
6. Why is zero an unsafe initial maximum candidate for an arbitrary nonempty list?
7. What does `continue` do to the update of a `for` loop?
8. What is the decimal digit count of zero?
9. Why can a wide destination variable fail to prevent an overflowing multiplication?
10. Where should a per-case counter be initialized in a multi-case problem?

Answer key

1. Integer division happens first, giving 4; it is then converted to `double`. Use a floating-point operand before division.
2. The first assigns 5. The second compares the current value with 5.
3. `x >= 1 && x <= 10`.
4. Five times: 0, 1, 2, 3, and 4.
5. The accumulator must remember earlier iterations instead of being reset for every value.
6. If every input is negative, zero is not even an input value and is larger than the true maximum.
7. The update still happens before the next condition check.
8. One digit. A do-while loop can account for this naturally.
9. The operation can overflow using its operand types before assignment happens. Widen the operands first.
10. Inside the outer case loop, before the inner value loop.

Ready for Session 2

You should be able to write a small program from a blank file, explain every line you wrote, trace its conditions and loops, and solve P01–P16 with understanding. Next, arrays and strings will let you store and revisit collections of values and characters.

1.22 Technical notes and references

For students

You do not need to read a language standard to study this chapter. The explanations, examples, traces, and practice solutions are the main learning material. The references below let mentors and curious students check the exact language rules behind important details.

Edition and verification

This chapter targets **C++23**. The standalone PDF and the collected-book edition include the same chapter source. The book edition currently contains Session 1 only; the other sessions are a roadmap, not completed material.

All **50 complete example and solution programs** were compiled using GNU C++ 14.2.0 with `-std=c++23`, warnings treated as errors, and undefined-behavior checks enabled. They passed **1,549 test executions**: 124 specified examples/boundary tests and 1,425 additional generated or exhaustive tests. The prime example was checked for every input in its stated range, 0–1000. The test script and machine-readable report are included in the source package.

These checks cover the complete files under `examples/` and `practice/`. Deliberately incorrect teaching fragments are excluded. Testing supports the checks made here, but it is not a mathematical guarantee that no mistake can exist.

Reference convention

Sources [2]–[14] reproduce clauses from the C++23 working draft N4950, with editorial updates, in an unofficial HTML rendering. They are not the current evolving C++ draft, so the language-version basis is explicit. The explanations and exercise programs in this chapter are newly written teaching material, not copied examples from the standard.

[1] GNU Compiler Collection: selecting the language standard

<https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html>

[2] C++23 draft: fundamental types, integer widths, and signed overflow

<https://timsong-cpp.github.io/cppwp/n4950/basic.fundamental>

[3] C++23 draft: multiplication, integer division, and remainder

<https://timsong-cpp.github.io/cppwp/n4950/expr.mul>

[4] C++23 draft: floating-point and integer conversions

<https://timsong-cpp.github.io/cppwp/n4950/conv.fpint>

Language and library references, continued

[5] C++23 draft: the if statement

<https://timsong-cpp.github.io/cppwp/n4950/stmt.if>

[6] C++23 draft: logical AND and short-circuit evaluation

<https://timsong-cpp.github.io/cppwp/n4950/expr.log.and>

[7] C++23 draft: iteration statements

<https://timsong-cpp.github.io/cppwp/n4950/stmt.iter>

[8] C++23 draft: the switch statement

<https://timsong-cpp.github.io/cppwp/n4950/stmt.switch>

[9] C++23 draft: break, continue, and return

<https://timsong-cpp.github.io/cppwp/n4950/stmt.jump>

[10] C++23 draft: input sentry and whitespace skipping

<https://timsong-cpp.github.io/cppwp/n4950/istream.sentry>

[11] C++23 draft: formatting and manipulators

<https://timsong-cpp.github.io/cppwp/n4950/ostream.format>

[12] C++23 draft: logical OR and short-circuit evaluation

<https://timsong-cpp.github.io/cppwp/n4950/expr.log.or>

[13] C++23 draft: the main function

<https://timsong-cpp.github.io/cppwp/n4950/basic.start.main>

[14] C++23 draft: initialization

<https://timsong-cpp.github.io/cppwp/n4950/dcl.init.general>

[15] GNU Compiler Collection: warning options

<https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>