Module 3 of 20

Control flow

if/else and the conditional operator, switch with fallthrough and enums, the three loops and the unsigned countdown trap, range-based for with auto& and structured bindings, and the loop patterns every program is built from.

Control flow is where a C++ program stops being a formula and starts being a program: it decides, it repeats, and it stops. The statements are the ones C had and Java copied — `if`, `switch`, `for`, `while`, `do … while` — but C++ evaluates them by its own rules. A condition is *converted* to `bool` rather than required to be one, so `if (n = 5)` compiles; `0 < x < 10` parses and is always true; a `switch` falls through unless told not to and refuses a `std::string` outright; and a countdown on an unsigned index never reaches `-1`, because unsigned arithmetic wraps. Every one of those is a bug that compiles, and this module is where you stop writing them.

The five lessons move from the branch to the loop to the loop *pattern*. Branching covers `if` / `else`, the C++17 `if` with initialiser, the conditional operator's type rules and short-circuit evaluation as a guard. Switch covers labels, `[[fallthrough]]`, switching on an `enum class` without a `default` so the compiler reports a missing case, and what to do instead of switching on a string. Loops fixes the three shapes, the half-open range convention, `break` / `continue`, the unsigned countdown trap and how to leave two nested loops. Range-based `for` settles `auto` versus `auto&` versus `const auto&` — the silent copy trap — structured bindings over a `std::map`, and why a container must not change while it is being walked. Loop patterns catalogues the shapes every later exercise reuses: search with early exit, accumulate and min/max, sentinel-terminated and until-EOF input, the invariant comment, the two-pointer sweep and building output once.

The exercises are whole programs that read standard input the way Module 1 taught — `std::cin >>` for tokens, `std::getline` for lines, `std::cin.ignore()` between them — and each one is built on the construct its lesson teaches: a grade table of nested ifs, a days-in-month calculator, a command interpreter that maps words to an enum and switches, a privilege ladder with `[[fallthrough]]`, a countdown that survives `n = 0`, a nested pair search, a word-frequency table printed in map order, an in-place clamp through `auto&`, a sentinel-terminated statistics reader and a two-pointer pair sweep. The checkpoint adds a run-length encoder, a bank ledger that reads until the input ends, and a prime lister with nested loops and an early exit.

CONTENTS

1	Branching — if, else and the conditional operator	13 min
2	switch — labels, fallthrough and what to switch on	13 min
3	Loops — for, while, do…while and the traps between them	14 min
4	Range-based for — iterating without indices	13 min
5	Loop patterns — search, accumulate, sentinel, EOF and two pointers	14 min
✓	Checkpoint — Control flow	25 min

Exercises in this module compile on Clang 18 · C++20. Reference solutions and the answer key follow each lesson.

LESSON 1 · 13 MIN

Branching — if, else and the conditional operator

Every program that does more than evaluate one formula has to decide, and in C++ deciding means `if`. The statement looks exactly like C's and Java's, which is the problem: the condition is *converted* to `bool` rather than required to be one, a chain such as `a < b < c` compiles and means something else, `if` can carry its own initialiser since C++17, and the conditional operator has type rules of its own. This lesson settles how a condition is evaluated, how branches nest, and how `&&` and `||` make an expression safe to evaluate at all.

---

if, else if, else

```
#include <iostream>

int main() {
    int score;
    std::cin >> score;
    if (score ⩾ 90) {
        std::cout << "A\n";
    } else if (score ⩾ 80) {
        std::cout << "B\n";
    } else if (score ⩾ 70) {
        std::cout << "C\n";
    } else {
        std::cout << "below C\n";
    }
    return 0;
}
```

The branches are tested top to bottom and the first true one runs; nothing after it is considered, which is why the chain can test `⩾ 80` without also saying `< 90`. `else if` is not a keyword pair — it is an `else` whose statement happens to be another `if` — so a chain is really a nest that the indentation flattens.

Braces are optional around a single statement and you should write them anyway. The famous reason is Apple's 2014 `goto fail` bug: a duplicated line after an unbraced `if` ran

unconditionally and skipped a certificate check for every user of the library. The everyday reason is the dangling `else`:

```
if (a > 0)
    if (b > 0)
        std::cout << "both positive\n";
else
    std::cout << "a is not positive\n";   // actually belongs to `if (b > 0)`
```

An `else` binds to the nearest `if` that does not yet have one, whatever the indentation says. Braces make your intent the compiler's reading.

The condition is converted, not checked

Java requires a `boolean` in an `if`; C++ takes any expression that converts to `bool`. Arithmetic values convert with zero meaning `false` and everything else `true`; pointers convert with `nullptr` meaning `false`; a stream such as `std::cin` converts through its `explicit operator bool`, which is why `if (std::cin >> x)` reads as "if the read succeeded". So these are all legal:

```
int count = 3;
if (count) { /* count ≠ 0 */ }
const char* p = nullptr;
if (!p) { /* p == nullptr */ }
if (std::cin >> count) { /* a number was read */ }
```

Conventional style spells the comparison out for numbers (`count ≠ 0`) and leaves the shorthand for pointers and streams, where "is there one?" is what the code means.

The conversion has a dark side. `if (n = 5)` is an assignment whose value is 5, converts to `true`, compiles, and runs the branch every time. GCC and Clang warn under `-Wall` ("suggest parentheses around assignment used as truth value"). The same rules are why `if (score ⩾ 90);` — note the semicolon — is a complete, empty `if` followed by a block that runs unconditionally.

Comparison chains that lie

```cpp
int x = 15;
if (0 < x < 10) std::cout << "single digit\n";   // prints it — for every x
```

There is no chained comparison in C++. `0 < x < 10` parses as `(0 < x) < 10`; the inner comparison yields a `bool`, the `bool` promotes to `int` (`0` or `1`), and both are less than 10. The compiler sees nothing wrong with the types (GCC's `-Wparentheses` does flag this particular shape, but not `x == 1 || 2`, which is always true for the same reason: `2` converts to `true` on its own). Write what you mean:

```cpp
if (0 < x && x < 10) { /* … */ }
if (x == 1 || x == 2) { /* … */ }
```

---

if with an initialiser (C++17)

A variable that exists only to be tested can be declared in the `if` itself:

```cpp
std::map<std::string, int> stock{{"apple", 4}};

if (auto it = stock.find("apple"); it != stock.end()) {
    it->second -= 1;
} else {
    std::cout << "not stocked\n";
}
// `it` no longer exists here
```

The initialiser runs first, the condition is tested, and the variable is in scope in both branches and nowhere after them. Before C++17 `it` had to be declared in the enclosing scope, where it lingered for the rest of the function. The form also removes the temptation to look up twice — `if (stock.count(k)) use(stock[k])` searches the map twice; this searches once. `switch` accepts the same syntax (next lesson).

The conditional operator

`cond ? a : b` is an *expression*: it produces a value, so it can sit inside another expression or initialise a variable, which an `if` statement cannot do.

```
const int larger = a > b ? a : b;
const char* unit = count == 1 ? "item" : "items";
std::cout << count << ' ' << unit << '\n';
```

The first line is the idiom's real justification: `larger` is `const`, which an `if` cannot manage without a mutable variable assigned in each branch. Two rules follow from it being an expression. First, both arms are converted to one common type — `flag ? 1 : 2.5` has type `double`, so the `1` becomes `1.0`. Second, precedence: `?:` binds more loosely than almost everything, so `std::cout << n == 1 ? "one" : "many"` parses as `(std::cout << n) == 1 …` and does not compile; put the whole conditional in parentheses when it is an operand of `<<`. Nested conditionals are legal and unreadable; at two levels, switch to an `if` chain.

Short-circuit evaluation as a guard

`&&` evaluates its left operand and stops if it is `false`; `||` stops if the left operand is `true`. The right operand is not evaluated at all — it is not merely ignored — which makes the left operand a guard for the right one:

```
if (i < v.size() && v[i] == target) { /* v[i] is never read out of range */ }
if (p != nullptr && p->ready) { /* p is never dereferenced when null */ }
if (line.empty() || line[0] == '#') continue;   // line[0] only when there is
```

Order matters: swap the operands and the guard runs after the thing it was guarding. The bitwise `&` and `|` always evaluate both sides, so `i < v.size() & v[i] == target` reads out of range. The same rule puts cheap tests before expensive ones: `if (cache.contains(k) || slowLookup(k))` calls `slowLookup` only on a miss.

What goes wrong

Mistake	What happens
`if (x = 5)`	Assigns, then tests 5: always true. `-Wall` warns.
`if (0 < x < 10)`	`(0 < x)` is 0 or 1, both `< 10`: always true.
`if (x == 1 || 2)`	`2` is true on its own: always true.
`if (cond);`	Empty statement; the block below runs unconditionally.
Unbraced nested `if` with `else`	The `else` binds to the inner `if`.
`if (s == "yes")` with `const char* s`	Compares two addresses, not the text (`std::string` compares content).
`if (d == 0.1)` with `double d`	Representation error (Module 2, lesson 3); compare with a tolerance.

Key takeaways

A condition is converted to `bool`: zero and `nullptr` are false, a stream is true when its last operation succeeded.

`a < b < c` compiles and is wrong; combine comparisons with `&&` and `||`.

`if (init; cond)` scopes a variable to the branches — use it for lookups.

`?:` is an expression with a common type; parenthesise it inside `<<`.

`&&` and `||` stop early, so the left operand can make the right one safe to evaluate; `&` and `|` do not.

Always brace; `else` binds to the nearest unmatched `if`.

EXERCISE 1

Letter grades with modifiers

Read an integer `n`, then `n` lines each holding a name (one word) and an integer score. For each line print `<name>: <grade>`.

The band comes from an `if` / `else if` chain: 90–100 is `A`, 80–89 `B`, 70–79 `C`, 60–69 `D`, 0–59 `F`. Inside a band other than `F`, a **nested** `if` adds a modifier: `+` when the last digit of the score is 7, 8 or 9 (and always for 100), `-` when the last digit is 0, 1 or 2 (never for 100), nothing otherwise. A score below 0 or above 100 prints `<name>: invalid`.

Input: `n`, then `n` lines of `name score`. **Output:** `n` lines.

```
4
ada 93
bob 100
cy 65
dee 41
```

prints

```
ada: A
bob: A+
cy: D
dee: F
```

INPUT 1

```
4
ada 93
bob 100
cy 65
dee 41
```

OUTPUT 1

```
ada: A
bob: A+
cy: D
dee: F
```

INPUT 2

```
3
eve 87
fay 70
gus 59
```

OUTPUT 2

```
eve: B+
fay: C-
gus: F
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <string>

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string name;
        int score;
        std::cin >> name >> score;
        std::string grade;
        if (score < 0 || score > 100) {
            grade = "invalid";
        } else {
            if (score ⩾ 90) grade = "A";
            else if (score ⩾ 80) grade = "B";
            else if (score ⩾ 70) grade = "C";
            else if (score ⩾ 60) grade = "D";
            else grade = "F";
            if (grade ≠ "F") {
                const int lastDigit = score % 10;
                if (score == 100 || lastDigit ⩾ 7) grade += '+';
                else if (lastDigit ⩽ 2) grade += '-';
            }
        }
        std::cout << name << ": " << grade << '\n';
    }
    return 0;
}
```

EXERCISE 2

Days in a month

Read an integer `n`, then `n` queries of `year month`. For each, print `<year>/<month>: <days>` — the number of days in that month — or `<year>/<month>: invalid` when the month is not 1–12.

A year is a leap year when it is divisible by 4 but not by 100, or divisible by 400; February has 29 days in a leap year and 28 otherwise. April, June, September and November have 30 days; the rest have 31. Compute the leap flag once per query — the C++17 `if` with initialiser is a good fit — and use the conditional operator to choose between 29 and 28.

Input: `n`, then `n` lines of `year month`. **Output:** `n` lines.

Example: `2024 2` → `2024/2: 29`; `1900 2` → `1900/2: 28`; `2023 13` → `2023/13: invalid`.

INPUT 1

```
4
2024 2
2023 2
2024 4
2024 12
```

OUTPUT 1

```
2024/2: 29
2023/2: 28
2024/4: 30
2024/12: 31
```

INPUT 2

```
2
1900 2
2000 2
```

OUTPUT 2

```
1900/2: 28
2000/2: 29
```

REFERENCE SOLUTION

```cpp
#include <iostream>

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        int year, month;
        std::cin >> year >> month;
        std::cout << year << '/' << month << ": ";
        if (month < 1 || month > 12) {
            std::cout << "invalid\n";
        } else if (const bool leap = (year % 4 == 0 && year % 100 != 0) || year % 40
            std::cout << (leap ? 29 : 28) << '\n';
        } else if (month == 4 || month == 6 || month == 9 || month == 11) {
            std::cout << 30 << '\n';
        } else {
            std::cout << 31 << '\n';
        }
    }
    return 0;
}
```

QUIZ

1. What does this print?

```cpp
int x = 5;
if (0 < x < 3) std::cout << "in";
else std::cout << "out";
```

A. `in`

B. `out`

C. Compile error

D. Undefined behaviour

2. What does this print?

```cpp
int n = 0;
if (n = 2) std::cout << "yes";
else std::cout << "no";
```

A. `yes`

B. `no`

C. Compile error

D. Nothing

3. In `if (auto it = m.find(k); it ≠ m.end()) { A } else { B }`, where is `it` in scope?

A. Only in block `A`

B. In both `A` and `B`, and nowhere after the statement

C. From the `if` to the end of the enclosing block

D. Only inside the condition

4. Which condition reads `v[i]` only when `i` is a valid index?

A. `if (v[i] == 7 && i < v.size())`

B. `if (i < v.size() && v[i] == 7)`

C. `if (i < v.size() & v[i] == 7)`

D. `if (v[i] == 7 || i < v.size())`

5. What happens here?

```
int n = 1;
std::cout << n == 1 ? "one" : "many";
```

A. Prints `one`

B. Prints `many`

C. Prints `1`

D. Compile error

6. What does this print?

```
int a = 0, b = 1;
if (a)
    if (b) std::cout << "A";
else std::cout << "B";
```

A. `A`

B. `B`

C. Nothing

D. Compile error

7. What does this print?

```
int a = 3;
double d = a > 2 ? 1 : 2.5;
std::cout << d;
```

A. `1`

B. `1.0`

C. `2.5`

D. Compile error

ANSWER KEY · BRANCHING — IF, ELSE AND THE CONDITIONAL OPERATOR

1. A `0 < x < 3` is `(0 < x) < 3`. The inner comparison is `true`, which promotes to `1`, and `1 < 3` holds — so the branch runs for every `x`. Write `0 < x && x < 3`.

2. A `n = 2` is an assignment whose value is `2`, and `2` converts to `true`. It compiles; `-Wall` warns "suggest parentheses around assignment used as truth value", which is the warning that catches a missing `=`.

3. B The C++17 initialiser is scoped to the whole `if` statement: both branches can use it and it is destroyed after the `else`. Before C++17 the variable had to be declared outside and lived on for the rest of the block.

4. B `&&` evaluates left to right and stops at the first `false`, so the bounds check guards the subscript. Reversing the operands reads out of range first; the bitwise `&` always evaluates both sides; `||` does not guard at all.

5. D `<<` binds tighter than `==` and `?:`, so this parses as `(std::cout << n) == 1 ? …`, and there is no `==` between an `ostream` and an `int`. Parenthesise the whole conditional: `std::cout << (n == 1 ? "one" : "many")`.

6. C The `else` pairs with the nearest unmatched `if` — `if (b)` — whatever the indentation suggests. `a` is 0, so the entire inner statement, its `else` included, is skipped and nothing prints. Read as indented it would print `B`; braces would make the two readings agree.

7. A Both arms of `?:` are converted to a common type — here `double` — so the result is `1.0`, and `std::cout` prints a `double` with no fractional part as `1`. The arms must convert to one type; they need not already have it.

LESSON 2 · 13 MIN

switch — labels, fallthrough and what to switch on

A `switch` compares one integral value against a set of constants and jumps straight to the matching label. That is a different thing from an `if` chain, not a shorthand for one: the compiler may turn it into a jump table or a binary search, the labels must be compile-time constants, and control *falls through* from one label to the next unless you stop it. Every one of those properties is a feature when the switch is written for it and a bug when it is not. This lesson covers the anatomy, fallthrough, variables declared inside cases, and what to do instead of switching on a string.

Anatomy

```
#include <iostream>

int main() {
    char op;
    int a, b;
    std::cin >> a >> op >> b;
    switch (op) {
        case '+': std::cout << a + b << '\n'; break;
        case '-': std::cout << a - b << '\n'; break;
        case '*': std::cout << a * b << '\n'; break;
        case '/':
            if (b == 0) {
                std::cout << "division by zero\n";
            } else {
                std::cout << a / b << '\n';
            }
            break;
        default:
            std::cout << "unknown operator " << op << '\n';
            break;
    }
    return 0;
}
```

The controlling expression must have an integral or enumeration type — `int`, `char`, `long long`, `bool` (pointless), an `enum` or `enum class`, or a class type convertible to one of those. `double` is rejected, and so is `std::string`. Each `case` label is a constant expression of that type: a literal, a `constexpr` variable, an enumerator, `'a' + 1`. Two labels with the same value are a compile error. `default` catches everything else and may appear anywhere, though convention puts it last.

`break` leaves the switch. Without it, execution continues into the statements under the next label — the switch is a list of labelled statements with one entry point per label, not a set of separate blocks.

Fallthrough, intended and accidental

Empty labels stacked on one body are the legitimate use of fallthrough and need no annotation:

```
switch (c) {
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowels;
        break;
    case ' ': case '\t':
        ++blanks;
        break;
    default:
        ++others;
        break;
}
```

A body that runs and then continues into the next body is where the bugs live — a forgotten `break` after `case 1:` silently adds case 2's work to case 1. GCC and Clang warn (`-Wimplicit-fallthrough`, part of `-Wextra`). When the fallthrough is what you want, say so with the C++17 attribute:

```
switch (level) {
    case 3:
        std::cout << "delete ";
        [[fallthrough]];
    case 2:
        std::cout << "write ";
        [[fallthrough]];
    case 1:
        std::cout << "read\n";
        break;
    default:
        std::cout << "no access\n";
        break;
}
```

`[[fallthrough]];` is a statement with no run-time effect: it documents the intent and tells the compiler not to warn, which a comment cannot do. Level 3 prints all three words — the ladder accumulates.

Enums and the missing default

Switching on a scoped enum is the most common use in modern code, and it comes with a compiler check worth keeping:

```
enum class Shape { Circle, Square, Triangle };

int sides(Shape s) {
    switch (s) {
        case Shape::Circle: return 0;
        case Shape::Square: return 4;
        case Shape::Triangle: return 3;
    }
    return -1;   // reached only for a value that is none of the enumerators
}
```

With no `default`, `-Wswitch` (in `-Wall`) warns when an enumerator is not handled — so adding `Shape::Hexagon` later produces a warning at every switch that forgot it. Write a `default` and the check is gone. The price is the line after the switch: an `enum class` can hold a value that is none of its enumerators (`static_cast<Shape>(7)`), the compiler knows it,

and a function without the trailing `return` "may reach the end of a non-void function". Handle every enumerator, omit `default`, keep the fallback after the switch.

Declaring a variable inside a case

```
switch (n) {                        // does not compile
    case 1:
        int doubled = n * 2;        // error: jump to case label crosses initial
        std::cout << doubled;
        break;
    case 2:
        std::cout << n;
        break;
}
```

Every label is inside one scope — the switch body — so `doubled` is in scope at `case 2` as well, but jumping to `case 2` skips its initialisation. C++ forbids jumping past the initialisation of a variable, and the compiler says so. The fix is a block per case:

```
switch (n) {
    case 1: {
        int doubled = n * 2;
        std::cout << doubled;
        break;
    }
    case 2:
        std::cout << n;
        break;
}
```

Now `doubled` belongs to the block and no label can jump into it. The rule covers anything with an initialiser, including `std::string s;` (a constructor counts); a bare `int x;` would be allowed, and reading it would be undefined behaviour — brace the case instead.

No strings in a switch

`switch (command)` with `std::string command` does not compile: the labels must be constant expressions, and comparing strings is a run-time operation on text that lives on the

heap. Three honest options:

An `if` chain — `if (cmd == "add") … else if (cmd == "sub") …`. Fine for a handful of words.

Map the word to an enum, then switch — an `if` chain (or a `std::map<std::string, Cmd>`) does the string work once, and the `switch` on the enum gets the exhaustiveness check and the jump table:

```
enum class Cmd { Add, Sub, Show, Unknown };

Cmd cmd = Cmd::Unknown;
if (word == "add") cmd = Cmd::Add;
else if (word == "sub") cmd = Cmd::Sub;
else if (word == "show") cmd = Cmd::Show;

switch (cmd) {
    case Cmd::Add: /* … */ break;
    case Cmd::Sub: /* … */ break;
    case Cmd::Show: /* … */ break;
    case Cmd::Unknown: std::cout << "unknown: " << word << '\n'; break;
}
```

Switch on a character when the vocabulary is single letters (`case 'q':`) — the shape of most menu loops.

You will meet code that switches on a `constexpr` hash of the string: it compiles, it is fast, and two words can hash to the same label without the compiler noticing. Leave it where you found it.

---

switch with an initialiser (C++17)

Like `if`, `switch` can declare a variable scoped to itself:

```
switch (const int code = status(); code) {
    case 0: std::cout << "ok\n"; break;
    case 1: std::cout << "retry\n"; break;
    default: std::cout << "failed with " << code << '\n'; break;
}
```

`code` is visible in every case and gone afterwards. It is most useful when the `default` branch wants to print the value that matched nothing.

switch or if?

Use `switch` when	Use `if` when
one integral or enum value is compared against constants	the tests are ranges (`score ⩾ 90`) or compound (`a && b`)
several values share one body (stacked labels)	the value is a `double` , a `std::string` or a pointer
you want `-Wswitch` to catch a missing enumerator	the branches test different variables

What goes wrong

A missing `break` runs the next case's body too. Compile with `-Wextra` ; write `[[fallthrough]]` when you mean it.

`break` inside a `switch` inside a loop leaves the switch, not the loop. Use a flag or restructure — the command-interpreter exercise makes you.

Duplicate case values and non-constant labels (`case n:`) are compile errors.

`case 1 ... 5:` is a GCC extension; it will not compile on other compilers. Use an `if` for ranges.

A `default` on an enum switch hides the missing-enumerator warning.

Key takeaways

`switch` dispatches on one integral or enum value to constant labels; `break` ends a case, and without it control falls through.

Stack empty labels freely; mark a body-to-body fallthrough with `[[fallthrough]];` .

A variable initialised inside a case needs braces around the case.

Strings cannot be switched on: map them to an enum first, or use an `if` chain.

Handle every enumerator and omit `default` so the compiler warns when the enum grows.

EXERCISE 1

A command interpreter

Write an interpreter for a one-register machine. The register starts at 0 and holds a `long long`. Read commands one per line until `quit` or the end of input:

- `add N`, `sub N`, `mul N` — apply the operation with the integer `N`
- `neg` — negate the register
- `show` — print `value=<register>`
- `reset` — set the register to 0
- `quit` — stop; nothing after it is processed
- anything else — print `unknown command: <word>`

Map the word to an `enum class` first, then `switch` on the enum. Remember that `break` inside the `switch` leaves the switch, not the reading loop — `quit` needs a flag the loop condition checks.

Input: commands, one per line. **Output:** one line per `show` and per unknown command.

```
add 5
mul 3
show
sub 20
neg
show
quit
```

prints

```
value=15
value=5
```

INPUT 1

```
add 5
mul 3
show
sub 20
neg
show
quit
```

OUTPUT 1

```
value=15
value=5
```

INPUT 2

```
show
fly
add 2
show
```

OUTPUT 2

```
value=0
unknown command: fly
value=2
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <string>

enum class Op { Add, Sub, Mul, Neg, Show, Reset, Quit, Unknown };

int main() {
    long long acc = 0;
    bool running = true;
    std::string word;
    while (running && std::cin >> word) {
        Op op = Op::Unknown;
        if (word == "add") op = Op::Add;
        else if (word == "sub") op = Op::Sub;
        else if (word == "mul") op = Op::Mul;
        else if (word == "neg") op = Op::Neg;
        else if (word == "show") op = Op::Show;
        else if (word == "reset") op = Op::Reset;
        else if (word == "quit") op = Op::Quit;

        switch (op) {
            case Op::Add: {
                long long value;
                std::cin >> value;
                acc += value;
                break;
            }
            case Op::Sub: {
                long long value;
                std::cin >> value;
                acc -= value;
                break;
            }
            case Op::Mul: {
                long long value;
                std::cin >> value;
                acc *= value;
                break;
            }
            case Op::Neg:
                acc = -acc;
                break;
            case Op::Show:
                std::cout << "value=" << acc << '\n';
                break;
            case Op::Reset:
                acc = 0;
                break;
            case Op::Quit:
                running = false;   // break alone would only leave the switch
```

```
                break;
            case Op::Unknown:
                std::cout << "unknown command: " << word << '\n';
                break;
        }
    }
    return 0;
}
```

EXERCISE 2

A privilege ladder

Read an integer `n`, then `n` lines each holding a name and a one-letter role. Roles are cumulative: an admin (`a` or `A`) may delete, write and read; an editor (`e` or `E`) may write and read; a viewer (`v` or `V`) may read. Print `<name>: <rights>` with the rights in that order, separated by single spaces — `delete write read`, `write read` or `read` — or `<name>: no access` for any other letter.

Write one `switch` on the role character: stack the two letters of each role as empty labels, and let the cases fall into each other with `[[fallthrough]];` so each level adds its word before the level below adds the next.

Input: `n`, then `n` lines of `name role`. **Output:** `n` lines.

```
3
ada A
bob e
cy v
```

prints

```
ada: delete write read
bob: write read
cy: read
```

INPUT 1

```
3
ada A
bob e
cy v
```

OUTPUT 1

```
ada: delete write read
bob: write read
cy: read
```

INPUT 2

```
2
dee x
eve E
```

OUTPUT 2

```
dee: no access
eve: write read
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <string>

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string name;
        char role;
        std::cin >> name >> role;
        std::string rights;
        switch (role) {
            case 'a':
            case 'A':
                rights += "delete ";
                [[fallthrough]];
            case 'e':
            case 'E':
                rights += "write ";
                [[fallthrough]];
            case 'v':
            case 'V':
                rights += "read";
                break;
            default:
                rights = "no access";
                break;
        }
        std::cout << name << ": " << rights << '\n';
    }
    return 0;
}
```

QUIZ

1. What does this print?

```
int x = 2;
switch (x) {
    case 1: std::cout << "one";
    case 2: std::cout << "two";
    case 3: std::cout << "three";
    default: std::cout << "other";
}
```

A. `two`

B. `twothree`

C. `twothreeother`

D. `other`

2. Which of these can be the controlling expression of a `switch`?

A. A `std::string`

B. A `double`

C. An `enum class` value

D. A `float`

3. Does this compile?

```
switch (n) {
    case 1:
        int y = n * 2;
        std::cout << y;
        break;
    case 2:
        std::cout << n;
        break;
}
```

A. Yes, and it works

B. No: jumping to `case 2` crosses the initialisation of `y`

C. Yes, but reading `y` in `case 2` is undefined behaviour

D. No: variables cannot be declared inside a `switch`

4. What does `[[fallthrough]];` do?

A. Makes execution continue into the next case where it otherwise would stop

B. Documents an intentional fallthrough and silences the compiler's warning; no run-time effect

C. Ends the `switch`, like `break`

D. Jumps to `default`

5. Why omit `default` when switching on an `enum class` whose every enumerator has a case?

A. `default` is not allowed with scoped enums

B. So `-Wswitch` warns when a new enumerator is added and no case handles it

C. It makes the jump table smaller

D. The compiler inserts a `default` automatically

6. What does this print?

```
for (int i = 0; i < 3; ++i) {
    switch (i) {
        case 1: break;
        default: std::cout << i;
    }
}
```

A. `02`

B. `0`

C. `012`

D. Nothing

7. You have `std::string cmd` with a dozen possible words. What is the sound way to dispatch on it?

A. `switch (cmd)` with string case labels

B. Map the word to an `enum class` with an `if` chain or a `std::map`, then `switch` on the enum

C. `switch (cmd.size())`

D. `switch` on a `constexpr` hash of the string

ANSWER KEY · SWITCH — LABELS, FALLTHROUGH AND WHAT TO SWITCH ON

1. C Control jumps to `case 2` and, with no `break`, falls through every following label including `default`. `-Wimplicit-fallthrough` (in `-Wextra`) warns on each of these; `[[fallthrough]];` would mark them as deliberate.

2. C The controlling expression must have an integral or enumeration type (or convert to one). Floating-point types and `std::string` are rejected at compile time — map a string to an enum first.

3. B `y` is in scope at `case 2`, but jumping there skips its initialiser, which C++ forbids ("jump to case label crosses initialization"). Wrap the `case 1` body in braces so `y`'s scope ends before the next label.

4. B Fallthrough is the default behaviour whenever `break` is absent; the attribute changes nothing at run time. It exists so `-Wimplicit-fallthrough` can tell a deliberate fallthrough from a forgotten `break`.

5. B With no `default`, the compiler can see that every enumerator is (or is not) handled and warns about missing ones — a check that survives the enum growing. A `default` swallows every future enumerator silently. Keep a fallback after the switch for out-of-range values.

6. A `break` leaves the innermost `switch` or loop — here the `switch` — so the `for` continues. `i = 1` prints nothing; `0` and `2` reach `default`. To leave the loop from inside a `switch`, set a flag the loop condition tests.

7. B Case labels must be integral constant expressions, so strings cannot be labels. Converting to an enum keeps the string comparison in one place and gives the `switch` its exhaustiveness check. Hashing works until two words collide, and `size()` is not a name.

LESSON 3 · 14 MIN

Loops — for, while, do...while and the traps between them

A loop is the one place where a program's correctness depends on a boundary: one iteration too many reads past the end of an array, one too few drops the last element, and a condition that never turns false runs until the judge kills the process. C++ has the three loops C had — `for`, `while`, `do … while` — and one trap Java programmers do not carry: unsigned arithmetic wraps, so a countdown on `std::size_t` never reaches `-1`. This lesson fixes the shape of each loop, the half-open convention that makes off-by-one errors rare, `break` and `continue`, and how to leave two nested loops at once.

Three loops, three shapes

```
for (int i = 0; i < n; ++i) { /* … */ }   // a known number of steps, or an i

while (balance > 0) { /* … */ }            // test first; may run zero times

do {
    std::cout << "enter a positive number: ";
    std::cin >> x;
} while (x ⩽ 0);                           // body first; always runs at leas
```

`for` is for counting and indexing; `while` is for "as long as", where the number of steps is not known in advance; `do … while` is the rare one — a body that must run at least once before its condition makes sense, the classic case being "ask, then check, repeat while wrong".

Anatomy of for

```
for (init; condition; step) body
```

`init` runs once. Then, per iteration: test `condition`, stop if false; run `body`; run `step`. All three parts are optional — `for (;;)` is an infinite loop — and `init` may declare several variables of one type:

```cpp
for (int lo = 0, hi = n - 1; lo < hi; ++lo, --hi) {
    std::swap(a[lo], a[hi]);
}
```

The variable declared in `init` is scoped to the loop: `i` does not exist after the closing brace. If the code after the loop needs the index at which it stopped, declare the variable outside:

```cpp
int i = 0;
for (; i < n && a[i] ≠ target; ++i) { }
// i is n when target is absent, otherwise its index
```

`++i` and `i++` do the same thing to an `int`; `++i` is the habit worth forming, because for an iterator (Module 13) `i++` has to produce a copy of the old value that nobody uses.

Half-open ranges and the fencepost

The C++ convention for a range is *half-open*: `[begin, end)` includes `begin` and excludes `end`. Every standard algorithm takes its ranges that way, and every counting loop should:

```cpp
for (int i = 0; i < n; ++i) { /* exactly n iterations: 0, 1, …, n-1 */ }
```

Three properties make the convention worth keeping. The number of elements is `end - begin`, with no `+ 1`. An empty range is `begin == end`, which needs no special case. Two adjacent ranges `[a, b)` and `[b, c)` share their boundary without overlapping. The loop that inclusively counts `1` to `n` — `for (int i = 1; i ⩽ n; ++i)` — is also correct, but the moment `⩽` and a zero-based index meet (`i ⩽ n` over `a[i]`), the last iteration reads `a[n]`, one past the end, which is undefined behaviour.

Printing `n` values separated by spaces with no trailing space is the fencepost problem in miniature — `n` values need `n - 1` separators, so print the separator *before* every value except the first: `if (i > 0) std::cout << ' ';`.

The unsigned countdown trap

`std::size_t` — the type of `v.size()` and of every container index — is unsigned. Unsigned arithmetic does not go negative; it wraps (Module 2, lesson 2). So the obvious countdown never stops:

```
std::vector<int> v{1, 2, 3};
for (std::size_t i = v.size() - 1; i ⩾ 0; --i) {   // i ⩾ 0 is always true
    std::cout << v[i];
}
```

After `i` reaches 0, `--i` wraps to 18446744073709551615, `v[i]` reads far outside the vector, and the behaviour is undefined. (GCC does warn: "comparison of unsigned expression in `⩾ 0` is always true".) There is a second trap in the same line: when `v` is empty, `v.size() - 1` already wraps and the loop begins out of range.

Two shapes are correct. Test before decrementing, so the body sees the decremented value:

```
for (std::size_t i = v.size(); i-- > 0;) {
    std::cout << v[i];        // last index first, ends after index 0
}
```

Or keep `i` one above the index you use:

```
for (std::size_t i = v.size(); i > 0; --i) {
    std::cout << v[i - 1];
}
```

Both are correct for an empty vector. The alternative — `int` indices with `static_cast<int>(v.size())` — works, but comparing a bare `int i` with `v.size()` draws `-Wsign-compare`, and that warning is right often enough to keep.

---

break and continue

`break` leaves the innermost enclosing loop *or switch*; `continue` skips the rest of the body and goes to the next iteration. In a `for` loop the step still runs after `continue`; in a `while` loop nothing runs but the condition, so a `continue` placed before the increment loops forever:

```
int i = 0;
while (i < 5) {
    if (i % 2 == 0) continue;     // i is 0, stays 0: never ends
    std::cout << i;
    ++i;
}
```

The `for` version — `for (int i = 0; i < 5; ++i) { if (i % 2 == 0) continue; … }` — is fine, because the `++i` is in the step. That difference is the practical reason to prefer `for` whenever there is a counter at all.

Nested loops and leaving both

`break` in an inner loop leaves the inner loop only; the outer one carries on. To stop both, put the fact that you are done somewhere the outer loop can see:

```
bool found = false;
int bi = 0, bj = 0;
for (int i = 0; i < n && !found; ++i) {
    for (int j = i + 1; j < n; ++j) {
        if (a[i] + a[j] == target) {
            bi = i;
            bj = j;
            found = true;
            break;                // leaves the inner loop; the outer conditio
        }
    }
}
```

When the search is a function of its own, `return` from the inner loop is cleaner than the flag (Module 4). `goto` can jump to a label after both loops; modern code does not use it.

Termination

Every loop needs something that moves toward the exit: an index growing toward `n`, a number halved each step, input running out. `while (true)` with a `break` is legitimate when the exit test belongs mid-body, provided the `break` is reachable on every path. `for (double x = 0; x != 1.0; x += 0.1)` has no such guarantee — ten additions of `0.1` do not make exactly `1.0` (Module 2, lesson 3); count in integers and scale.

What goes wrong

Mistake	What happens
`for (…);` with a stray semicolon	Empty body; the block below runs once, after the loop.
`i ⩽ n` over a zero-based array	Reads `a[n]`: undefined behaviour.
`size_t` countdown with `i ⩾ 0`	Wraps at 0; never terminates, reads out of range.
`continue` before the increment in a `while`	Infinite loop.
`break` inside a `switch` inside a loop	Leaves the switch only.
`while (!std::cin.eof())`	Runs once more after the last value (lesson 5).

Key takeaways

`for` counts, `while` waits for a condition, `do … while` runs first and asks afterwards.

Ranges are half-open: `[0, n)` with `<`, `n` iterations, no `+ 1`, empty when `begin == end`.

Never write `size_t i ⩾ 0`; count down with `i-- > 0` or `i > 0` and `v[i - 1]`.

`continue` in a `while` skips the increment; `break` leaves only the innermost loop or switch.

Leave nested loops with a flag in the outer condition or a `return`.

EXERCISE 1

Launch countdown

Read an integer `n` (0 ≤ n ≤ 20), then `n` stage names (single words, whitespace-separated). Store them in a `std::vector<std::string>` and print them in **reverse** order, one per line as `<index>: <name>` where `index` is the position in the input (0-based), so the last stage prints first and `0:` prints last. Then print `liftoff`.

The loop variable must be a `std::size_t` — the type of `stages.size()` — and the program must be correct when `n` is 0: a countdown written as `for (std::size_t i = n - 1; i ⩾ 0; --i)` never terminates, because an unsigned value cannot go below zero.

Input: `n`, then `n` names. **Output:** `n` lines, then `liftoff`.

```
3
fuel
ignition
boosters
```

prints

```
2: boosters
1: ignition
0: fuel
liftoff
```

INPUT 1

```
3
fuel
ignition
boosters
```

OUTPUT 1

```
2: boosters
1: ignition
0: fuel
liftoff
```

INPUT 2

```
1
check
```

OUTPUT 2

```
0: check
liftoff
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <string>
#include <vector>

int main() {
    int n;
    std::cin >> n;
    std::vector<std::string> stages;
    for (int i = 0; i < n; ++i) {
        std::string stage;
        std::cin >> stage;
        stages.push_back(stage);
    }
    for (std::size_t i = stages.size(); i-- > 0;) {
        std::cout << i << ": " << stages[i] << '\n';
    }
    std::cout << "liftoff\n";
    return 0;
}
```

EXERCISE 2

First pair with the target sum

Read integers `n` and `target`, then `n` integers. Find the first pair of positions `i < j` such that `a[i] + a[j] == target`, trying `i` from 0 upwards and, for each `i`, `j` from `i + 1` upwards. Print `<i> <j>` for the first pair found, or `none`.

Use two nested `for` loops, and stop **both** as soon as a pair is found — a `break` leaves only the inner loop, so the outer loop's condition must also test a flag. Store the values as `long long` so the sum cannot overflow.

Input: `n target`, then `n` integers. **Output:** one line.

Example: `5 9` then `2 7 11 15 4` → `0 1` (2 + 7). `4 10` then `1 2 3 4` → `none`.

INPUT 1

```
5 9
2 7 11 15 4
```

OUTPUT 1

```
0 1
```

INPUT 2

```
4 10
1 2 3 4
```

OUTPUT 2

```
none
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <vector>

int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<long long> a(n);
    for (int i = 0; i < n; ++i) {
        std::cin >> a[i];
    }
    bool found = false;
    int bi = 0;
    int bj = 0;
    for (int i = 0; i < n && !found; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (a[i] + a[j] == target) {
                bi = i;
                bj = j;
                found = true;
                break;   // leaves the inner loop; the outer condition sees found
            }
        }
    }
    if (found) std::cout << bi << ' ' << bj << '\n';
    else std::cout << "none\n";
    return 0;
}
```

QUIZ

1. What happens?

```cpp
std::vector<int> v{1, 2, 3};
for (std::size_t i = v.size() - 1; i ⩾ 0; --i) std::cout << v[i];
```

A. Prints `321` and stops

B. Prints `123`

C. Prints `321`, then `--i` wraps to a huge value and `v[i]` reads out of range: undefined behaviour

D. Compile error

2. How many times does the body of `for (int i = 1; i ⩽ 10; i += 3)` run?

A. 3

B. 4

C. 10

D. It never ends

3. What happens?

```
int i = 0;
while (i < 5) {
    if (i % 2 == 0) continue;
    std::cout << i;
    ++i;
}
```

A. Prints `13`

B. Prints `1234`

C. Never terminates

D. Prints `024`

4. What does this print?

```
int i;
for (i = 0; i < 4; ++i);
std::cout << i;
```

A. `0123`

B. `4`

C. `3`

D. Compile error

5. A match is found in the inner of two nested loops. Which correctly stops both?

A. `break;` in the inner loop

B. `break; break;`

C. A `bool` flag set before the inner `break` and tested in the outer loop's condition (or `return` from a function)

D. `continue;` in the outer loop

6. How does `do { body } while (cond);` differ from `while (cond) { body }`?

A. They are identical

B. `do … while` runs the body once before testing, so it always executes at least once

C. `do … while` cannot contain `break`

D. `while` is faster

7. Compiled with `-Wall -Wextra`, what does `for (int i = 0; i < v.size(); ++i)` produce, `v` being a `std::vector<int>`?

A. Nothing — it is clean

B. A warning: comparison of integer expressions of different signedness

C. A compile error

D. Undefined behaviour

ANSWER KEY · LOOPS — FOR, WHILE, DO…WHILE AND THE TRAPS BETWEEN THEM

1. C `std::size_t` is unsigned, so `i ⩾ 0` is always true (GCC warns exactly that). After index 0 the decrement wraps to 18446744073709551615 and the subscript is out of bounds. Count down with `for (std::size_t i = v.size(); i-- > 0;)`.

2. B `i` takes the values 1, 4, 7 and 10 — four iterations; 13 fails `⩽ 10`. Counting the values a loop variable actually takes is the reliable way to answer fencepost questions.

3. C `i` is 0, which is even, so `continue` jumps back to the condition without reaching `++i`; `i` stays 0 forever. In a `for` loop the step would still run — which is why counters belong in `for`.

4. B The semicolon after the `for` is an empty body, so the loop simply counts `i` up until `i < 4` fails, leaving `i` at 4. Because `i` was declared outside the loop it is still in scope for the print.

5. C `break` leaves only the innermost loop, and a second `break` after it is unreachable. Either the outer loop tests a flag (`i < n && !found`) or the search lives in a function and `return`s from the inner loop.

6. B `do … while` tests after the body, so the body runs at least once — the shape for "ask, then check". A `while` loop tests first and may run zero times. Both accept `break` and `continue`.

7. B `v.size()` is `std::size_t`, `i` is `int`, and `-Wsign-compare` (in `-Wall` for C++) flags the mixed comparison. Use `std::size_t i`, or `static_cast<int>(v.size())` when an `int` index is needed.

LESSON 4 · 13 MIN

Range-based for — iterating without indices

Most loops over a container do not need an index; they need each element in turn. The range-based `for`, added in C++11, says exactly that — `for (element : container)` — and removes the three places an index loop can be wrong: the start, the bound and the subscript. What it does not remove is the question of *how* each element is handed to you: by copy, by reference or by const reference, spelled `auto`, `auto&` and `const auto&`. Choosing wrongly is silent — a loop that modifies copies compiles and does nothing. This lesson covers what the loop expands to, the three declarations and when each is right, structured bindings over a `std::map`, when an index loop is still the answer, and why a container must not change while it is being iterated.

The form and what it expands to

```
#include <iostream>
#include <string>
#include <vector>

int main() {
    std::vector<std::string> names{"ada", "bob", "cy"};
    for (const std::string& name : names) {
        std::cout << name << '\n';
    }
    for (char c : std::string{"abc"}) std::cout << c << ' ';
    for (int x : {10, 20, 30}) std::cout << x << ' ';
    int fib[] = {1, 1, 2, 3, 5};
    for (int f : fib) std::cout << f << ' ';
    std::cout << '\n';
    return 0;
}
```

The compiler rewrites `for (decl : range) body` into roughly:

```
{
    auto&& __range = range;
    auto __begin = std::begin(__range);
    auto __end = std::end(__range);
    for (; __begin != __end; ++__begin) {
        decl = *__begin;
        body
    }
}
```

Three consequences follow. Anything with `begin()` and `end()` works: every standard container, `std::string`, a built-in array whose size the compiler knows, an initialiser list, a `std::string_view`. The end is computed once, before the first iteration. And `decl` is initialised from `*__begin` each time — which is where the copy question comes from.

---

auto, auto& and const auto&

```
std::vector<std::string> words{"one", "two"};

for (auto w : words) w += "!";                 // w is a copy; words is unchar
for (auto& w : words) w += "!";                // w refers to the element; wor
for (const auto& w : words) std::cout << w;    // read-only, no copy
```

Declaration	Meaning	Use when
`auto x`	copy of each element	the element is small (`int`, `char`, a pointer) or you want a copy to alter
`auto& x`	reference to the element	modifying in place
`const auto& x`	read-only reference	reading anything bigger than a register — strings, vectors, structs

The first line above is the trap: it compiles, runs, and leaves `words` as it was, because each `w` is a fresh `std::string` destroyed at the end of its iteration. There is no warning. The reverse mistake, `auto` where `const auto&` was meant, is invisible too and merely slow — a loop over a `std::vector<std::string>` with `auto` copies every string. The rule that avoids both:

`const auto&` by default, `auto&` to modify, plain `auto` only for small types. Writing the type instead of `auto` (`const std::string&`, `char&`) is equally correct and often clearer.

```
std::string s = "hello";
for (char& c : s) {
    c = static_cast<char>(std::toupper(static_cast<unsigned char>(c)));
}
```

`<cctype>` functions take and return `int`, and the `unsigned char` cast is the correct way to hand them a `char` (Module 5, lesson 3).

A map, in order, with structured bindings

A `std::map<K, V>` iterates in key order — that is what makes it a map and not an `unordered_map` — and each element is a `std::pair<const K, V>`. C++17 structured bindings unpack it in the declaration:

```
std::map<std::string, int> freq{{"pear", 2}, {"apple", 5}, {"fig", 1}};
for (const auto& [word, count] : freq) {
    std::cout << word << ' ' << count << '\n';   // apple 5, fig 1, pear 2
}
for (auto& [word, count] : freq) count *= 2;     // values are modifiable; ke
```

The key is `const` in the pair because changing it would break the tree's ordering; `auto&` therefore gives a mutable `count` and an immutable `word`. `auto [k, v]` without the reference copies every pair. Before C++17 the same loop read `entry.first` and `entry.second`. Sorted output for free is the reason the frequency-count exercise uses `std::map`: an `unordered_map` would have to be copied into a vector and sorted first, because its iteration order is unspecified and differs between runs and library versions.

When an index is still the answer

Range-for gives you elements, not positions. Reach for an index loop when the body needs:

- the position itself — `std::cout << i << ": " << v[i]`;
- a neighbour — `v[i] == v[i - 1]` for runs, or `v[i] < v[i + 1]` for "is sorted";
- two ranges in lockstep — `a[i] * b[i]` for a dot product;

a direction or stride — backwards, every second element, from index 3;

to stop early at a position you then use after the loop.

A counter declared beside a range-for (`std::size_t i = 0; for (const auto& x : v) { /* … */ ++i; }`) is a legitimate middle ground when only the position is missing. C++20 `std::views::reverse` (`for (int x : v | std::views::reverse)`) covers "backwards"; `std::views::enumerate` is C++23 and not available on this track's runtime — Module 14 covers views.

Do not modify the container you are iterating

```
std::vector<int> v{1, 2, 3};
for (int x : v) {
    if (x == 2) v.push_back(4);   // undefined behaviour
}
```

The loop holds `__begin` and `__end` from before the first iteration. `push_back` may reallocate the vector's storage, after which both point into freed memory; even without reallocation, `__end` still points at the old end. `erase` is the same story with iterators that now skip or repeat. The rule: a range-for reads a container whose *size does not change* in the body. To build a filtered or extended result, write into a second container; to delete while walking, use explicit iterators and the `it = v.erase(it)` idiom (Module 13, lesson 6). Modifying the *elements* through `auto&` is fine — the container's shape is what must hold still.

What goes wrong

Mistake	What happens
`for (auto s : strings) s += "x";`	Alters copies; the container is unchanged, no warning.
`for (auto s : strings) read(s);`	Correct, and copies every string. Use `const auto&`.
`for (auto& x : {1, 2, 3}) x *= 2;`	Compile error: the elements of an initialiser list are `const`.
`for (auto [k, v] : m)`	Copies every pair; `const auto&` reads, `auto&` modifies `v`.
`for (int x : longLongs)`	Narrows each element silently; use `auto` or the right type.
`push_back` / `erase` inside the loop	Invalidated iterators: undefined behaviour.
`for (auto x : make().items())`	If `items()` returns a reference, the temporary `make()` is gone before the loop runs (C++23 fixes this); bind `make()` to a variable first.

Key takeaways

`for (decl : range)` walks `begin()` to `end()`, computing the end once and initialising `decl` from each element.

`const auto&` to read, `auto&` to modify, `auto` only for small types — the copy trap is silent.

A `std::map` iterates in key order; `const auto& [key, value]` unpacks each entry, and the key is always `const`.

When the body needs a position, a neighbour or two ranges at once, use an index loop.

Never grow or shrink the container inside its own range-for.

EXERCISE 1

Word frequencies in order

Read an integer `n`, then `n` words (whitespace-separated, possibly across several lines). Count how often each word appears in a `std::map<std::string, int>` and print one line per distinct word as

`<word> <count>`, in the map's own order — a range-based `for` with a structured binding `const auto& [word, count]`. Finish with `distinct=<number of distinct words>`.

Note that a `std::map` orders its keys with `<` on `std::string`, so uppercase letters sort before lowercase ones.

Input: `n`, then `n` words. **Output:** one line per distinct word, then `distinct=<k>`.

Example: `6` then `the cat saw the dog the` →

```
cat 1
dog 1
saw 1
the 3
distinct=4
```

INPUT 1

```
6
the cat saw the dog the
```

OUTPUT 1

```
cat 1
dog 1
saw 1
the 3
distinct=4
```

INPUT 2

```
4
b a b a
```

OUTPUT 2

```
a 2
b 2
distinct=2
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    int n;
    std::cin >> n;
    std::map<std::string, int> freq;
    for (int i = 0; i < n; ++i) {
        std::string word;
        std::cin >> word;
        ++freq[word];
    }
    for (const auto& [word, count] : freq) {
        std::cout << word << ' ' << count << '\n';
    }
    std::cout << "distinct=" << freq.size() << '\n';
    return 0;
}
```

EXERCISE 2

Clamp in place

Read two integers `lo hi` (lo ≤ hi), then an integer `n` (n ≥ 1), then `n` integers into a `std::vector<int>`. Clamp every element into the range `[lo, hi]` **in place** with a range-based `for` whose loop variable is a reference (`int&` or `auto&`) — a loop over copies would leave the vector unchanged. Then print the clamped values on one line separated by single spaces, and on the next line `sum=<sum of the clamped values>` ; read the values for that with `const auto&` and keep the sum in a `long long` .

Input: `lo hi` , then `n` , then `n` integers. **Output:** two lines.

Example: `0 10` / `5` / `-3 4 12 10 0` →

```
0 4 10 10 0
sum=24
```

INPUT 1

```
0 10
5
-3 4 12 10 0
```

OUTPUT 1

```
0 4 10 10 0
sum=24
```

INPUT 2

```
1 5
3
1 2 3
```

OUTPUT 2

```
1 2 3
sum=6
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <vector>

int main() {
    int lo, hi, n;
    std::cin >> lo >> hi >> n;
    std::vector<int> values;
    for (int i = 0; i < n; ++i) {
        int x;
        std::cin >> x;
        values.push_back(x);
    }
    for (int& x : values) {
        if (x < lo) x = lo;
        else if (x > hi) x = hi;
    }
    long long sum = 0;
    bool first = true;
    for (const auto& x : values) {
        if (!first) std::cout << ' ';
        std::cout << x;
        first = false;
        sum += x;
    }
    std::cout << '\n' << "sum=" << sum << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```
std::vector<std::string> names{"ann", "bo"};
for (auto n : names) n += "!";
for (const auto& n : names) std::cout << n << ' ';
```

A. `ann! bo!`

B. `ann bo`

C. Compile error

D. Undefined behaviour

2. Does `for (auto& x : {1, 2, 3}) x *= 2;` compile?

A. Yes, and doubles each value

B. No: the elements of an initialiser list are `const`, so `x` is a `const int&` and cannot be assigned

C. Yes, but has no effect

D. No: range-for does not accept braces

3. In `for (const auto& [k, v] : m)` over a `std::map<std::string, int>`, what is the type of `k`?

A. `std::string`

B. `const std::string`

C. `int`

D. `std::pair<std::string, int>`

4. What does this print?

```
std::map<std::string, int> m{{"pear", 1}, {"apple", 2}, {"fig", 3}};
for (const auto& [k, v] : m) std::cout << k << ' ';
```

A. `pear apple fig`

B. `apple fig pear`

C. The order is unspecified

D. `fig apple pear`

5. Which loop doubles every element of `std::vector<int> v` in place?

A. `for (auto x : v) x *= 2;`

B. `for (auto& x : v) x *= 2;`

C. `for (const auto& x : v) x *= 2;`

D. `for (int x : v) v[x] *= 2;`

6. What is wrong here?

```
std::vector<int> v{1, 2, 3};
for (int x : v) {
    if (x == 2) v.push_back(4);
}
```

A. Nothing — `v` becomes `{1, 2, 3, 4}`

B. Undefined behaviour: `push_back` may reallocate and invalidate the iterators the loop holds

C. Compile error: `v` is const inside the loop

D. The loop runs forever

7. When is an index loop the right choice over a range-based `for`?

A. Whenever the container is a `std::vector`

B. When the body needs the position, a neighbour, a direction, or two ranges walked together

C. Never — range-for replaces every loop

D. Only for built-in arrays

ANSWER KEY · RANGE-BASED FOR — ITERATING WITHOUT INDICES

1. B `auto n` declares a copy of each element; the `+=` modifies the copy, which is destroyed at the end of the iteration. The vector is unchanged and no warning is issued. `auto& n` would modify in place.

2. B A braced list is a `std::initializer_list<int>` whose elements are `const int`; `auto&` deduces `const int&`, and `x *= 2` is an assignment to a read-only reference. Reading them with `const auto&` or `int` is fine.

3. B A map's element type is `std::pair<const K, V>` — the key is `const` because changing it would break the ordering. So even `auto& [k, v]` lets you modify `v` but never `k`.

4. B `std::map` is an ordered container: iteration follows the key order, `apple < fig < pear`, regardless of insertion order. It is `std::unordered_map` whose order is unspecified.

5. B Only a non-const reference reaches the element. The copy version changes nothing; the `const` version does not compile; the last one uses the *value* as an index, which is wrong and usually out of range.

6. B The range-for captured `begin()` and `end()` before the first iteration; a reallocation moves the elements and leaves both dangling, and even without one the cached end is stale. Never change a container's size inside its own range-for.

7. B Range-for hands you elements, not positions. Comparing `v[i]` with `v[i - 1]`, walking backwards, striding, or pairing `a[i]` with `b[i]` all need an index. For plain "each element", range-for is safer.

LESSON 5 · 14 MIN

Loop patterns — search, accumulate, sentinel, EOF and two pointers

Almost every loop you will write in this track is one of about six loops with the names changed. Recognising which one you are writing is what lets you write it right the first time: a search knows what it prints when nothing matches, an accumulation knows what it starts from, a reading loop knows what ends it. This lesson catalogues them, states the invariant each keeps, and adds two that separate a working solution from a fast one: the two-pointer sweep over sorted data and building output once.

Linear search with an early exit

```
int found = -1;
for (int i = 0; i < n; ++i) {
    if (a[i] == target) {
        found = i;
        break;
    }
}
if (found == -1) std::cout << "absent\n";
else std::cout << "at " << found << '\n';
```

The three decisions are made before the loop: what "not found" looks like (`-1` , a flag, `n`), that the *first* match wins (`break` on it), and that the answer is a position rather than the value. Keeping "not found" as a value outside the valid range is the standard library's own convention — `std::string::npos` , `end()` — and Module 14's `std::find` replaces this loop with one call.

Accumulate, count, min and max

```cpp
long long sum = 0;
int evens = 0;
int lo = a[0], hi = a[0];          // n ⩾ 1
for (int i = 0; i < n; ++i) {
    sum += a[i];
    if (a[i] % 2 == 0) ++evens;
    if (a[i] < lo) lo = a[i];
    if (a[i] > hi) hi = a[i];
}
```

Two rules. The sum of `int`s goes in a `long long`: a thousand values near two billion overflow `int`, and signed overflow is undefined behaviour, not a wrong answer. And a minimum or maximum is initialised from the *data*, never from `0` — `int hi = 0` reports 0 as the maximum of `{-3, -7}`. When the loop reads its data as it goes and cannot look at `a[0]` first, use a count (`if (count == 0) lo = hi = x;`) or start from `std::numeric_limits<int>::max()` for a minimum and `::min()` for a maximum (`<limits>`, Module 2). To find the *position* of the maximum, compare `a[i] > a[best]` — the strict `>` keeps the first of several equal maxima, `⩾` keeps the last.

Sentinel-terminated input

Some inputs end with a value that is not data: a `0`, a `-1`, a line reading `end`. The read and the test belong in one condition:

```cpp
int x;
int count = 0;
while (std::cin >> x && x ≠ 0) {
    ++count;
    // x is real data here
}
```

`std::cin >> x` runs first; if it fails (end of input, or a non-number) the stream converts to `false` and `&&` stops without testing `x`, which would otherwise hold a stale value. If it succeeds, `x ≠ 0` decides. The sentinel itself is consumed and never counted. The alternative — read, test in the body, `break` — is the same loop with the exit in the middle, but the one-line condition cannot forget to test the read. A sentinel must be a value the data cannot take.

Reading until the input ends

```cpp
int x;
while (std::cin >> x) { /* every whitespace-separated number */ }

std::string line;
while (std::getline(std::cin, line)) { /* every line, including empty ones */
```

Both stop when the read fails, which at the end of standard input is exactly when there is nothing left. The version people write from other languages is wrong:

```cpp
while (!std::cin.eof()) {     // wrong
    std::cin >> x;
    use(x);                   // runs once more with a failed read
}
```

`eof()` becomes true only after a read has *tried* to go past the end, so after the last number the loop runs once more: the read fails and `use(x)` sees what the failed read left — since C++11, `0`. The "n then n values" format sidesteps the question with a counted loop; "until EOF" is for formats that do not say how much is coming.

State the invariant

An invariant is a statement that is true at the top of every iteration. Write it as a comment on the loop; it is how you check a loop without running it, and it is what an interviewer means by "walk me through why this is correct":

```cpp
// invariant: best is the largest of a[0..i)
int best = a[0];
for (int i = 1; i < n; ++i) {
    if (a[i] > best) best = a[i];
}
// i == n, so best is the largest of a[0..n)
```

Establish it before the loop (`best` is the largest of `a[0..1)`), show the body keeps it (after comparing `a[i]`, `best` is the largest of `a[0..i+1)`), and the exit condition turns it into the result. The one-line habit catches the `hi = 0` bug above and every off-by-one in a binary search.

The two-pointer sweep

When data is sorted, a nested pair of loops is usually one loop with two indices walking toward each other:

```
// a is sorted ascending; find two elements that sum to target
int lo = 0;
int hi = n - 1;
// invariant: any pair that sums to target has both indices in [lo, hi]
while (lo < hi) {
    const long long sum = static_cast<long long>(a[lo]) + a[hi];
    if (sum == target) { std::cout << lo << ' ' << hi << '\n'; break; }
    if (sum < target) ++lo;      // a[lo] is too small for every partner up t
    else --hi;                   // a[hi] is too large for every partner from
}
```

Each step discards one index for a reason the invariant states: if the sum is too small, `a[lo]` paired with anything at or below `hi` is smaller still, so `lo` can never be part of the answer. The loop does at most `n - 1` steps where the nested version did about `n²/2`. The same shape reverses a string in place, tests a palindrome, merges two sorted arrays (one index each, moving forward) and removes duplicates from a sorted vector (a *write* index trailing a *read* index). Lesson 3's nested pair search and this one solve the same problem; the difference is the sort.

Build the output once

```
std::string out;
for (int i = 0; i < n; ++i) {
    if (!out.empty()) out += ' ';
    out += std::to_string(a[i]);
}
std::cout << out << '\n';
```

Two things improve. The separator logic lives in one place and cannot produce a trailing space — the test is "is there something before me", not "am I last". And a program that prints a hundred thousand values makes one write instead of two hundred thousand stream operations; with `std::endl` in the loop it would also flush every line (Module 1, lesson 4). `std::ostringstream` (Module 5) is the same idea with `<<` formatting.

What goes wrong

Mistake	What happens
`int max = 0;` before the loop	Wrong for all-negative data. Start from the data or the type's limit.
Summing `int`s into an `int`	Overflow: undefined behaviour. Use `long long`.
Testing the sentinel after counting	The sentinel is counted. Test in the condition.
`while (!std::cin.eof())`	One extra iteration on a failed read.
`while (lo ⩽ hi)` in the pair sweep	Pairs an element with itself. Distinct indices need `<`.

Key takeaways

- Decide the "not found" value before writing a search; `break` on the first match.
- Sums in `long long`; min/max from the data, never from 0.
- Put the read in the condition: `while (std::cin >> x && x ≠ sentinel)` and `while (std::cin >> x)` for EOF; never loop on `eof()`.
- Write the invariant as a comment; it is the proof the loop is right.
- Sorted data turns a nested search into a two-pointer sweep; build output in a string and print once.

EXERCISE 1

Sentinel-terminated statistics

Read integers until the sentinel `0` or the end of input, whichever comes first. The sentinel is not data. Print `count=<c> sum=<s> min=<m> max=<M>` on one line, or `no data` when no value precedes the sentinel.

Put the read and the sentinel test in the loop condition: `while (std::cin >> x && x ≠ 0)`. Initialise the minimum and maximum from the first value read, not from 0 — the data may be entirely negative. Keep the sum in a `long long`.

Input: integers separated by whitespace, normally ending with `0`. **Output:** one line.

Example: `4 8 -2 15 0` → `count=4 sum=25 min=-2 max=15`. `0` → `no data`.

INPUT 1

```
4 8 -2 15 0
```

OUTPUT 1

```
count=4 sum=25 min=-2 max=15
```

INPUT 2

```
0
```

OUTPUT 2

```
no data
```

REFERENCE SOLUTION

```cpp
#include <iostream>

int main() {
    int x;
    int count = 0;
    long long sum = 0;
    int lo = 0;
    int hi = 0;
    while (std::cin >> x && x != 0) {
        if (count == 0) {
            lo = x;
            hi = x;
        } else {
            if (x < lo) lo = x;
            if (x > hi) hi = x;
        }
        sum += x;
        ++count;
    }
    if (count == 0) {
        std::cout << "no data\n";
    } else {
        std::cout << "count=" << count << " sum=" << sum << " min=" << lo << " max="
    }
    return 0;
}
```

EXERCISE 2

Two-pointer pair in a sorted list

Read integers `n` and `target`, then `n` integers in **non-decreasing** order. Find two distinct positions whose values sum to `target` with a two-pointer sweep: `lo` starts at 0 and `hi` at `n - 1`; while `lo < hi`, if `a[lo] + a[hi]` equals `target` print `<lo> <hi>` and stop; if it is less, move `lo` up; otherwise move `hi` down. Print `none` if the pointers meet without a match.

Follow the sweep exactly as described so the pair you find is the one the judge expects. Store the values as `long long`. Write the loop's invariant as a comment above it.

Input: `n target`, then `n` sorted integers. **Output:** one line.

Example: `6 10` then `1 2 4 6 8 9` → `0 5`. `5 12` then `1 3 5 7 9` → `1 4`.

INPUT 1

```
6 10
1 2 4 6 8 9
```

OUTPUT 1

```
0 5
```

INPUT 2

```
5 12
1 3 5 7 9
```

OUTPUT 2

```
1 4
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <vector>

int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<long long> a(n);
    for (int i = 0; i < n; ++i) {
        std::cin >> a[i];
    }
    int lo = 0;
    int hi = n - 1;
    // invariant: every pair summing to target has both positions inside [lo, hi]
    while (lo < hi) {
        const long long sum = a[lo] + a[hi];
        if (sum == target) {
            std::cout << lo << ' ' << hi << '\n';
            return 0;
        }
        if (sum < target) ++lo;
        else --hi;
    }
    std::cout << "none\n";
    return 0;
}
```

QUIZ

1. `int best = 0; for (int x : v) if (x > best) best = x;` — what is wrong?

A. Nothing

B. It reports 0 when every element is negative; initialise from the first element or from `std::numeric_limits<int>::min()`

C. It fails when `v` has one element

D. It does not compile

2. With input `3 4 5` and no zero, what does `while (std::cin >> x && x ≠ 0) sum += x;` do?

A. Sums 12 and stops when the input ends

B. Loops forever waiting for the sentinel

C. Sums 0

D. Runtime error

3. Why is `while (!std::cin.eof()) { std::cin >> x; use(x); }` wrong?

A. `eof()` is not a member of `std::cin`

B. After the last value is read `eof()` is still false, so the loop runs again with a failed read and `use(x)` sees a zeroed `x`

C. It reads only the first value

D. It is correct

4. In a two-pointer sweep over a sorted array for `a[lo] + a[hi] == target`, the sum is less than `target`. What moves?

A. `--hi`

B. `++lo`

C. Both pointers

D. Restart from `lo = 0`

5. Compared with `std::cout << x << ' '` inside the loop, building the output in a `std::string` and printing once…

A. Is required by the judge

B. Makes one write instead of thousands and keeps the separator logic in one place, so no trailing space

C. Uses less memory

D. Is always ten times faster

6. What does this print?

```
int a[] = {3, 9, 4, 9};
int best = 0;
for (int i = 1; i < 4; ++i) if (a[i] > a[best]) best = i;
std::cout << best;
```

A. `1`

B. `3`

C. `9`

D. `0`

7. What is a loop invariant?

A. A variable the loop never changes

B. A statement that holds at the top of every iteration and, together with the exit condition, proves the result

C. The loop's condition

D. A `const` loop variable

ANSWER KEY · LOOP PATTERNS — SEARCH, ACCUMULATE, SENTINEL, EOF AND TWO POINTERS

1. B 0 is not an element of the data, so the answer is wrong whenever every value is below it. A maximum starts from the data (`v[0]` , or the first value read) or from the type's minimum.

2. A After 5, the next `std::cin >> x` fails at end of input; the stream converts to `false` and `&&` stops before testing `x` . A sentinel loop written this way ends correctly with or without the sentinel.

3. B `eof()` only becomes true after a read has tried to go past the end. Test the read itself — `while (std::cin >> x)` — so the body runs only for values that were actually read.

4. B `a[lo]` paired with any element at or below `hi` gives a sum no larger than the current one, so `lo` cannot be part of a solution and moves up. A sum that is too large discards `hi` for the mirror reason.

5. B The test `if (!out.empty()) out += ' '` puts a separator only before a value that has a predecessor, and one final `<<` replaces a stream call per token. The judge ignores trailing spaces, but not every judge does.

6. A The strict `>` replaces `best` only for a strictly larger value, so the first 9 (index 1) wins over the second. `⩾` would keep the last maximum, index 3. The loop tracks a *position*, hence `a[best]` .

7. B Establish it before the loop, show the body preserves it, and the exit condition turns it into the answer — `best` is the largest of `a[0..i)` becomes the largest of `a[0..n)` when `i = n` . It is how a loop is checked without running it.

CHECKPOINT · 25 MIN

Checkpoint — Control flow

This checkpoint covers the whole module: branching and the conditional operator, `switch` with fallthrough and enums, the three loops and the unsigned countdown, range-based `for` with `auto&` and structured bindings, and the loop patterns from search to two pointers.

How it works. Fourteen questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- Why is `if (0 < x < 10)` true for every `x`?
- What does `[[fallthrough]];` change at run time, and what does it change at compile time?
- Why does `for (std::size_t i = v.size() - 1; i ⩾ 0; --i)` never terminate?
- What is the difference between `for (auto s : v)` and `for (auto& s : v)`?
- Why is `while (!std::cin.eof())` one iteration too many?
- When is a pair of indices moving toward each other better than two nested loops?

The three programs are a run-length encoder that reads lines with `std::getline` after a count, a bank ledger that reads commands until the input ends, and a prime lister built from nested loops with an early exit. Read the input format carefully before writing the loop.

EXERCISE 1

Run-length encoding

Read an integer `n`, then `n` lines of text (each at least one character; a line may contain spaces). For each line print its run-length encoding: every maximal run of one repeated character becomes the character followed by the run's length.

Read `n` with `std::cin >>`, then call `std::cin.ignore()` before the first `std::getline` — the newline after `n` would otherwise be read as an empty line. Walk each line with two indices: `i` marks the start of a run and `j` advances while the character repeats. `std::to_string(k)` turns a count into text.

Input: `n`, then `n` lines. **Output:** `n` lines.

Example: `2` / `aaabccdddd` / `abc` →

```
a3b1c2d4
a1b1c1
```

INPUT 1

```
2
aaabccdddd
abc
```

OUTPUT 1

```
a3b1c2d4
a1b1c1
```

INPUT 2

```
1
mississippi
```

OUTPUT 2

```
m1i1s2i1s2i1p2i1
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <string>

int main() {
    int n;
    std::cin >> n;
    std::cin.ignore();
    for (int k = 0; k < n; ++k) {
        std::string line;
        std::getline(std::cin, line);
        std::string out;
        std::size_t i = 0;
        // invariant: line[0..i) has been encoded into out
        while (i < line.size()) {
            std::size_t j = i;
            while (j < line.size() && line[j] == line[i]) ++j;
            out += line[i];
            out += std::to_string(j - i);
            i = j;
        }
        std::cout << out << '\n';
    }
    return 0;
}
```

EXERCISE 2

A bank ledger

Read commands until the end of input — `while (std::cin >> word)` — and keep a balance in a `long long` that starts at 0:

- `deposit N` — add `N`
- `withdraw N` — subtract `N`, unless `N` exceeds the balance, in which case print `insufficient funds` and leave it
- `balance` — print `balance=<value>`
- any other word — print `unknown: <word>`

Map each word to an `enum class` and `switch` on it, handling every enumerator without a `default`. After the input ends print `final=<balance> deposits=<number of deposits> rejected=<number of refused withdrawals>`.

Input: commands, one per line, until end of input (possibly none). **Output:** one line per `balance`, refusal and unknown word, then the final line.

```
deposit 100
withdraw 30
balance
withdraw 500
balance
```

prints

```
balance=70
insufficient funds
balance=70
final=70 deposits=1 rejected=1
```

INPUT 1

```
deposit 100
withdraw 30
balance
withdraw 500
balance
```

OUTPUT 1

```
balance=70
insufficient funds
balance=70
final=70 deposits=1 rejected=1
```

INPUT 2

```
balance
foo
deposit 5
deposit 5
balance
```

OUTPUT 2

```
balance=0
unknown: foo
balance=10
final=10 deposits=2 rejected=0
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <string>

enum class Cmd { Deposit, Withdraw, Balance, Unknown };

int main() {
    long long balance = 0;
    int deposits = 0;
    int rejected = 0;
    std::string word;
    while (std::cin >> word) {
        Cmd cmd = Cmd::Unknown;
        if (word == "deposit") cmd = Cmd::Deposit;
        else if (word == "withdraw") cmd = Cmd::Withdraw;
        else if (word == "balance") cmd = Cmd::Balance;

        switch (cmd) {
            case Cmd::Deposit: {
                long long amount;
                std::cin >> amount;
                balance += amount;
                ++deposits;
                break;
            }
            case Cmd::Withdraw: {
                long long amount;
                std::cin >> amount;
                if (amount > balance) {
                    std::cout << "insufficient funds\n";
                    ++rejected;
                } else {
                    balance -= amount;
                }
                break;
            }
            case Cmd::Balance:
                std::cout << "balance=" << balance << '\n';
                break;
            case Cmd::Unknown:
                std::cout << "unknown: " << word << '\n';
                break;
        }
    }
    std::cout << "final=" << balance << " deposits=" << deposits << " rejected=" <<
    return 0;
}
```

EXERCISE 3

Primes up to n

Read an integer `n` (1 ≤ n ≤ 10 000) and print every prime number from 2 to `n` on one line separated by single spaces — or `none` when there is no prime in range — followed by a line `count=<how many>`.

Test each candidate by trial division in a nested loop: divisors `d` from 2 while `d * d ⩽ candidate`; `break` at the first divisor and `continue` the outer loop for a composite. Build the first line in a `std::string` and print it once, so no trailing space is produced.

Input: `n`. **Output:** two lines.

Example: `20` →

```
2 3 5 7 11 13 17 19
count=8
```

INPUT 1

```
20
```

OUTPUT 1

```
2 3 5 7 11 13 17 19
count=8
```

INPUT 2

```
2
```

OUTPUT 2

```
2
count=1
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <string>

int main() {
    int n;
    std::cin >> n;
    std::string out;
    int count = 0;
    for (int candidate = 2; candidate <= n; ++candidate) {
        bool prime = true;
        for (int d = 2; d * d <= candidate; ++d) {
            if (candidate % d == 0) {
                prime = false;
                break;
            }
        }
        if (!prime) continue;
        if (!out.empty()) out += ' ';
        out += std::to_string(candidate);
        ++count;
    }
    if (count == 0) out = "none";
    std::cout << out << '\n' << "count=" << count << '\n';
    return 0;
}
```

CHECKPOINT QUESTIONS

1. `if (x == 1 || 2)` is…

A. True when `x` is 1 or 2

B. Always true, because `2` converts to `true` on its own

C. A compile error

D. Always false

2. What does `int n = 7; std::cout << (n > 5 ? 1 : 2.0);` print?

A. `1`

B. `1.0`

C. `2`

D. Compile error

3. `double d = 1.5; switch (d) { … }`

A. Works, matching `case 1.5:`

B. Compile error: the controlling expression must be of integral or enumeration type

C. Truncates `d` to 1

D. Undefined behaviour

4. What does this print?

```
char c = 'b';
switch (c) {
    case 'a': std::cout << 1; [[fallthrough]];
    case 'b': std::cout << 2; [[fallthrough]];
    case 'c': std::cout << 3; break;
    default: std::cout << 0;
}
```

A. `2`

B. `23`

C. `230`

D. `123`

5. What does `for (unsigned i = 3; i ⩾ 0; --i) std::cout << i << ' ';` do?

A. Prints `3 2 1 0` and stops

B. Never terminates: after 0, `--i` wraps to 4294967295 and `i ⩾ 0` is still true

C. Compile error

D. Prints `3 2 1`

6. How many times does the body of `for (int i = 10; i > 0; i -= 3)` run?

A. 3

B. 4

C. 10

D. It never ends

7. What does this print?

```
for (int i = 0; i < 5; ++i) {
    if (i == 1) continue;
    if (i == 3) break;
    std::cout << i;
}
```

A. `02`

B. `024`

C. `0`

D. `0234`

8. Which statement about `for (auto s : words)` versus `for (auto& s : words)` over a `std::vector<std::string>` is correct?

A. They are identical

B. The first copies each string and any change to `s` is lost; the second refers to the element in place

C. The first is faster

D. The second does not compile

9. What does this print?

```
std::map<std::string, int> m{{"b", 1}, {"a", 2}};
for (auto& [k, v] : m) v *= 10;
for (const auto& [k, v] : m) std::cout << k << v << ' ';
```

A. `b10 a20`

B. `a20 b10`

C. `a2 b1`

D. Compile error: `k` is const

10. Input is `5 -1 7`. What does `long long sum = 0; int x; while (std::cin >> x && x != -1) sum += x;` leave in `sum`?

A. 5

B. 11

C. 12

D. -1

11. Which is not a valid `case` label (`n` is a non-const `int`, `Op` an `enum class`)?

A. `case 1 + 1:`

B. `case 'x':`

C. `case n:`

D. `case Op::Add:`

12. `while (std::getline(std::cin, line))` ends when…

A. `line` is empty

B. `getline` fails at the end of input and the stream converts to `false`

C. A line contains only whitespace

D. It never ends

13. Input is `3⏎hello⏎`. After `int n; std::cin >> n; std::string line; std::getline(std::cin, line);` what is `line`?

A. `hello`

B. `` (empty)

C. `3`

D. Undefined

14. In a nest of unbraced `if` statements, which `if` does an `else` belong to?

A. The first `if` in the nest

B. The `if` at the same indentation

C. The nearest preceding `if` that does not already have an `else`

D. It is ambiguous and does not compile

ANSWER KEY · CHECKPOINT — CONTROL FLOW

1. B `||` takes two operands, `x == 1` and `2`; a non-zero integer is `true`, so the whole condition is. Write `x == 1 || x == 2`.

2. A The arms of `?:` are converted to a common type, `double`, so the value is `1.0`; `std::cout` prints an integral-valued `double` as `1`. The parentheses are needed because `<<` binds tighter than `?:`.

3. B A `switch` dispatches on integer-like values only; `double`, `float` and `std::string` are rejected. Use an `if` chain for those.

4. B Control enters at `case 'b'`, prints 2, falls through to `case 'c'`, prints 3, and `break` ends the switch before `default`. `case 'a'` is never entered.

5. B An unsigned value is always `>= 0`, so the condition can never fail; the decrement past 0 wraps to the maximum. Count down with `i-- > 0` in the condition, or loop while `i > 0` and use `i - 1`.

6. B `i` takes 10, 7, 4 and 1 — four iterations; −2 fails `> 0`. List the values the variable takes rather than dividing.

7. A `continue` skips the print for 1 but the step still runs; `break` at 3 ends the loop before 3 and 4 print. Only 0 and 2 reach the output.

8. B `auto` deduces a value, so each iteration constructs and destroys a copy — slower, and modifications vanish. `auto&` (or `const auto&` for reading) binds to the element itself.

9. B `auto&` binds to the real pair, so `v *= 10` changes the map; the key is `const` but is never assigned here. A `std::map` iterates in key order, `a` before `b`, regardless of insertion order.

10. A The loop reads 5 and adds it, reads −1, and stops because it is the sentinel — the sentinel is consumed but not added, and the 7 after it is never read.

11. C Labels must be constant expressions. `1 + 1`, a character literal and an enumerator are all constants; a run-time variable is not.

12. B `getline` returns the stream, which is `true` while the read succeeded. An empty line is a successful read of an empty string; only running out of input ends the loop.

13. B `>>` stops at the newline after `3` and leaves it in the stream, so `getline` reads up to that newline and returns an empty string. Call `std::cin.ignore()` after the `>>` before switching to `getline`.

14. C The dangling-else rule pairs an `else` with the closest unmatched `if`, whatever the indentation suggests. Braces around every branch make the pairing explicit.
