Module 2 of 20

Fundamental types

The built-in types and their exact sizes on this platform; integer overflow, wrap-around and bits; floating-point representation and deterministic printing; conversions and the four casts; const, constexpr and auto; literals and scoped enums.

C++ hands you the machine's types with the machine's sizes and none of the guard rails a managed language wraps around them. Every value is a fixed number of bytes with a fixed meaning, and the compiler emits exactly the instruction for that type — which is where the speed comes from, and why the rules at the edges belong to the programmer: what a signed overflow does (nothing you may rely on), what an unsigned one does (wraps), what `0.1 + 0.2` is (not `0.3`), and what `-1 < 1u` says (false).

The module takes the six topics in order. The fundamental types, their sizes on this track's x86-64 runtime, `<cstdint>`, `<limits>` and why brace-initialisation is the default. Integer arithmetic: overflow as undefined behaviour and how to detect it first, unsigned wrap-around, integer promotion, truncating division, the bit operators and C++20's `<bit>`. Floating point: IEEE 754 in enough detail to predict `0.1 + 0.2`, tolerance comparison, accumulation error, printing with a fixed precision, NaN and infinity, and when `long long` cents is the right answer. Conversions and casts: narrowing and how braces reject it, the signed/unsigned trap, `static_cast`, `reinterpret_cast`, `const_cast` and why the C-style cast is retired. `const`, `constexpr` and the four deduction rules of `auto`. Literals in every base and suffix, escape and raw strings, and unscoped versus scoped enumerations.

Every exercise is a judged program on the Clang 18 / C++20 runtime: a platform report from `std::numeric_limits`, a smallest-type classifier, a checked sum with `__builtin_add_overflow`, an eight-bit flag register, two-decimal statistics and a tolerance comparer, a Caesar shift written in `char` arithmetic, signed/unsigned comparisons done right, `constexpr` power and primality proved by `static_assert`, `auto`-typed accumulators, a literal decoder and a scoped weekday enum. The checkpoint adds a checked running product, an invoice kept in integer cents and a literal classifier that names each token's type.

CONTENTS

1	The fundamental types — bool, char, integers and floating point	14 min
2	Integer arithmetic and bits	15 min
3	Floating point — representation, error and printing	15 min
4	Conversions and casts	14 min
5	const, constexpr and auto	14 min
6	Literals and enumerations	14 min
✓	Checkpoint — Fundamental types	25 min

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

LESSON 1 · 14 MIN

The fundamental types — bool, char, integers and floating point

Every C++ variable has a type fixed at compile time, and the type decides three things: how many bytes the object occupies, which values those bytes can mean, and which machine operations the compiler emits for it. Java pins all of this in its specification; C++ leaves the sizes to the platform, which is why "how big is an `int`?" has the honest answer "it depends" and the practical answer "4 bytes everywhere this track runs". This lesson lays out the built-in types, their exact sizes on the x86-64 runtime the exercises use, where the limits live, and the initialisation form that keeps the compiler on your side.

What a type decides

```
int count = 3;          // 4 bytes, -2147483648 … 2147483647, integer instruct
double ratio = 0.75;    // 8 bytes, IEEE 754 binary64, floating-point instruct
bool done = false;      // 1 byte, true or false
char grade = 'B';       // 1 byte, a small integer the streams print as a char
```

Because the size is part of the type, the compiler can keep a local in a register or at a fixed stack offset and compute on it directly — the zero-overhead principle from Module 1 in its smallest form. The other side of the bargain is that nothing checks at run time whether a value fits. C++ trusts the type you chose, and the next lesson is about what happens when that trust is misplaced.

bool

`bool` holds `true` or `false`, printed as `1` and `0` unless you switch the stream with `std::cout << std::boolalpha`. Any arithmetic or pointer value converts to `bool` — zero is `false`, everything else `true` — which is what makes `if (n)` and `if (ptr)` legal. Arithmetic on a `bool` promotes it to `int`: `true + true` is `2`, and `std::count_if` results are routinely summed that way.

char and the character types

`char` is one byte and, on this platform, **signed**: it holds −128 to 127. It is an integer type that the streams happen to print as a character, so `'A' + 1` is the `int` 66 and `static_cast<char>('A' + 1)` is `'B'` — Module 5 builds digit and case arithmetic on exactly this. When you mean *a byte* rather than *a character*, say so with `unsigned char` or `std::uint8_t`; the value 200 does not fit in a signed `char`, and storing it there gives −56. `wchar_t`, `char16_t`, `char32_t` and C++20's `char8_t` exist for wide and Unicode text; this track stays with `char` and UTF-8 bytes.

The integer family

Type	Bytes on this platform	Range
`short`	2	−32 768 … 32 767
`int`	4	±2.1 × 10^{9}
`long`	8	±9.2 × 10^{18}
`long long`	8	±9.2 × 10^{18}

Each comes in `signed` (the default) and `unsigned` flavours, and `unsigned int` is usually written `unsigned`. The standard guarantees only an ordering and minimums (`int` at least 16 bits, `long` at least 32, `long long` at least 64); the numbers above are the x86-64 Linux ABI. `long` is the one that moves — it is 4 bytes on 64-bit Windows — which is why portable code writes `long long` when it needs 64 bits and reserves `long` for APIs that demand it.

Unsigned types hold 0 … $2^{n} - 1$ and never overflow: arithmetic wraps modulo 2^{n}, which lesson 2 covers. Use them for bit patterns and for values that are counts *by definition* — `std::size_t`, the type of `sizeof` and of every container's `.size()`, is an 8-byte unsigned type here. Do not reach for `unsigned` merely because a value "can't be negative": the difference of two counts is negative half the time, and an unsigned result cannot say so.

Fixed widths: `<cstdint>`

```cpp
#include <cstdint>
std::int32_t  id = 7;         // exactly 32 bits, two's complement
std::uint8_t  byte = 200;     // exactly 8 bits, 0 … 255
std::int64_t  total = 0;      // exactly 64 bits
std::uint64_t hash = 0;
```

When the width is part of the meaning — a file format, a network packet, a hash, a bitmask — name it. The aliases buy certainty, not new types: `std::int32_t` *is* `int` here, `std::int64_t` is `long`, and `std::uint8_t` is `unsigned char`. That last one is a trap: `std::cout << std::uint8_t{65}` prints `A`, because the stream sees a character type. Print bytes as numbers with `static_cast<int>(b)` or with the unary plus, `+b`, which promotes to `int`. `std::intmax_t`, `std::uintptr_t` (an integer wide enough for a pointer) and `std::ptrdiff_t` (the signed type of a pointer difference) round out the header.

Floating point

`float` is 4 bytes with 24 bits of precision (about 7 decimal digits), `double` 8 bytes with 53 bits (15–17 digits), and `long double` 16 bytes on this platform, holding the x87 80-bit format with 64 bits of precision. Default to `double`: it is what the literals are (`0.5` is a `double`, `0.5f` a `float`), what `<cmath>` is written for, and no slower than `float` on a modern CPU unless you are filling memory with millions of them. Lesson 3 is about what these types cannot represent.

`<limits>`: ask, don't remember

```cpp
#include <limits>
std::numeric_limits<int>::max()          // 2147483647
std::numeric_limits<int>::min()          // -2147483648
std::numeric_limits<double>::max()       // 1.79769e+308
std::numeric_limits<double>::lowest()    // -1.79769e+308  (min() is the smal
std::numeric_limits<double>::digits10    // 15
std::numeric_limits<char>::is_signed     // true here
```

`std::numeric_limits<T>` is a template you query with a type. `max()` and `min()` are `constexpr`, so they cost nothing and belong in range checks and in "start the maximum search from the smallest possible value" initialisers. The C macros `INT_MAX` and `LLONG_MIN` from `<climits>` still work and are what older code uses.

`sizeof` and the platform table

`sizeof(T)` is a compile-time constant of type `std::size_t` giving an object's size in bytes; `sizeof x` works on an expression too. Print it to describe the platform, never to build logic on:

```
std::cout << "int " << sizeof(int) << ", long long " << sizeof(long long)
          << ", double " << sizeof(double) << ", pointer " << sizeof(void*) <<
```

```
int 4, long long 8, double 8, pointer 8
```

On this track's runtime: `bool` and `char` 1, `int` 4, `long` and `long long` 8, `double` 8, `long double` 16, `std::size_t` 8, every pointer 8.

Initialisation forms — and why braces

```
int a;          // no initialiser: an indeterminate value; reading it is undej
int b = 5;      // copy-initialisation, the C form
int c(5);       // direct-initialisation
int d{5};       // list-initialisation ("brace-init"), since C++11
int e{};        // value-initialisation: zero
```

The first line is the one that bites. A local of a fundamental type with no initialiser holds whatever the memory held before, the compiler is allowed to assume you never read it, and the optimiser will cheerfully produce a program that prints 0 today and garbage after the next edit. Globals and `static` locals are zeroed before `main` runs; automatic locals are not.

Braces are the modern default for two reasons. `int e{}` gives a definite zero for any type, which makes "declare now, fill in later" safe. And brace-init refuses **narrowing**: `int x{3.7}`, `char c{300}` and `unsigned u{-1}` are compile errors where `int x = 3.7` silently stores 3 (lesson 4 has the full list). Write `T name{value};` unless you have a reason not to. The usual reason is a container: `std::vector<int> v{10}` is one element holding 10, not ten elements, so sizes still go in parentheses (Module 13).

Choosing a type

`int` for counts and indices that fit; `long long` for anything summed or multiplied; `std::size_t` when comparing with `.size()`.

`double` for measurements; `float` only when memory says so.

`std::uint8_t` or `unsigned char` for bytes, `char` for text, `bool` for flags — never `int` for any of them.

A fixed-width type whenever the width is part of a format.

Pitfalls

`int x;` followed by `x += 1;` is undefined behaviour, and `-Wall` warns only when the compiler can see the read. Initialise.

`std::cout << std::uint8_t{7}` prints a control character, not `7`.

`long` is 8 bytes here and 4 on Windows; say `long long` when you mean 64 bits.

`std::numeric_limits<double>::min()` is tiny and positive; the most negative double is `lowest()`.

`char` is signed here: a byte value above 127 stored in a `char` is negative.

Key takeaways

A type fixes size, value set and operations at compile time; the sizes are the platform's, and on this track `int` is 4 bytes and `long long` is 8.

`char` is a signed one-byte integer that prints as a character; `std::uint8_t` prints the same way, so cast bytes to `int` to see numbers.

Use `<cstdint>` when the width matters and `<limits>` instead of remembering constants.

An uninitialised local holds garbage and reading it is undefined behaviour; `T x{}` is the safe default, and braces reject narrowing.

`int` by default, `long long` for sums and products, `std::size_t` for sizes, `double` for real numbers.

EXERCISE 1

Platform report

Read an integer `n`, then `n` lines each naming one type: `bool`, `char`, `int`, `long`, `long long`, `size_t`, `double` or `long double`. For each of the six integer types print `<name> size=<bytes> min=<min> max=<max>`; for the two floating types print `<name> size=<bytes> digits10=<digits>`; for any other name print `<name> unknown`.

Take every number from `sizeof` and `std::numeric_limits` — never type a constant in. Print `char`'s limits as numbers, not characters.

Input: `n`, then `n` type names, one per line (`long long` contains a space, so read lines with `std::getline`). **Output:** `n` lines.

```
3
int
char
double
```

→

```
int size=4 min=-2147483648 max=2147483647
char size=1 min=0 max=255
double size=8 digits10=15
```

INPUT 1

```
3
int
char
double
```

OUTPUT 1

```
int size=4 min=-2147483648
max=2147483647
char size=1 min=0 max=255
double size=8 digits10=15
```

INPUT 2

```
2
long long
string
```

OUTPUT 2

```
long long size=8
min=-9223372036854775808
max=9223372036854775807
string unknown
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    std::cin.ignore();
    for (int i = 0; i < n; ++i) {
        std::string name;
        std::getline(std::cin, name);
        if (name == "bool") {
            std::cout << name << " size=" << sizeof(bool)
                      << " min=" << std::numeric_limits<bool>::min()
                      << " max=" << std::numeric_limits<bool>::max() << '\n';
        } else if (name == "char") {
            // A char prints as a character; promote to int to see the number.
            std::cout << name << " size=" << sizeof(char)
                      << " min=" << static_cast<int>(std::numeric_limits<char>::min(
                      << " max=" << static_cast<int>(std::numeric_limits<char>::max(
        } else if (name == "int") {
            std::cout << name << " size=" << sizeof(int)
                      << " min=" << std::numeric_limits<int>::min()
                      << " max=" << std::numeric_limits<int>::max() << '\n';
        } else if (name == "long") {
            std::cout << name << " size=" << sizeof(long)
                      << " min=" << std::numeric_limits<long>::min()
                      << " max=" << std::numeric_limits<long>::max() << '\n';
        } else if (name == "long long") {
            std::cout << name << " size=" << sizeof(long long)
                      << " min=" << std::numeric_limits<long long>::min()
                      << " max=" << std::numeric_limits<long long>::max() << '\n';
        } else if (name == "size_t") {
            std::cout << name << " size=" << sizeof(std::size_t)
                      << " min=" << std::numeric_limits<std::size_t>::min()
                      << " max=" << std::numeric_limits<std::size_t>::max() << '\n';
        } else if (name == "double") {
            std::cout << name << " size=" << sizeof(double)
                      << " digits10=" << std::numeric_limits<double>::digits10 << '\
        } else if (name == "long double") {
            std::cout << name << " size=" << sizeof(long double)
                      << " digits10=" << std::numeric_limits<long double>::digits10
        } else {
            std::cout << name << " unknown\n";
        }
    }
```

```
    return 0;
}
```

EXERCISE 2

The smallest type that fits

Read an integer `n`, then `n` integers, each within the range of a signed 64-bit integer. For each value print `<value> fits in <type>`, where the type is the narrowest of `int8_t`, `int16_t`, `int32_t` and `int64_t` whose range contains it.

Read the values into a `long long` and compare against `std::numeric_limits<std::int8_t>::min()` / `max()` and the other fixed-width types from `<cstdint>` — no hand-typed constants.

Input: `n`, then `n` integers separated by whitespace. **Output:** `n` lines.

Example: input `3 100 -129 40000` →

```
100 fits in int8_t
-129 fits in int16_t
40000 fits in int32_t
```

INPUT 1

```
5
100
-129
40000
2147483648
0
```

OUTPUT 1

```
100 fits in int8_t
-129 fits in int16_t
40000 fits in int32_t
2147483648 fits in int64_t
0 fits in int8_t
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        long long v;
        std::cin >> v;
        std::string type = "int64_t";
        if (v ⩾ std::numeric_limits<std::int8_t>::min() && v ⩽ std::numeric_limits
            type = "int8_t";
        } else if (v ⩾ std::numeric_limits<std::int16_t>::min() && v ⩽ std::numeri
            type = "int16_t";
        } else if (v ⩾ std::numeric_limits<std::int32_t>::min() && v ⩽ std::numeri
            type = "int32_t";
        }
        std::cout << v << " fits in " << type << '\n';
    }
    return 0;
}
```

QUIZ

1. What does this print?

```cpp
int x;
std::cout << x;
```

A. `0` — locals start at zero

B. Whatever was in that memory, reliably

C. Nothing reliable: reading an uninitialised local is undefined behaviour

D. A compile error

2. What happens with `int x{3.5};`?

A. Compile error: narrowing conversion

B. `x` is `3`

C. `x` is `4`

D. Undefined behaviour

3. What does this print?

```
std::uint8_t b = 65;
std::cout << b;
```

A. `65`

B. `A`

C. A compile error

D. `'A'`

4. What is the type of `sizeof(int)` and of `v.size()` for a `std::vector`?

A. `int`

B. `unsigned`

C. `std::size_t` — an unsigned type, 8 bytes on this platform

D. `long long`

5. Which statement about `long` is correct?

A. It is 8 bytes on every platform

B. It is 8 bytes on this platform and 4 on 64-bit Windows, so portable code says `long long` for 64 bits

C. It is always the same size as `int`

D. The standard fixes it at 32 bits

6. What is `std::numeric_limits<double>::min()`?

A. The most negative double

B. The smallest positive normalised double; the most negative is `lowest()`

C. `-1.79769e+308`

D. Zero

7. Which declaration gives a definite zero for any fundamental type `T`?

A. `T x;`

B. `T x{};`

C. `T x();`

D. `T x = NULL;`

ANSWER KEY · THE FUNDAMENTAL TYPES — BOOL, CHAR, INTEGERS AND FLOATING POINT

1. C An automatic local of a fundamental type has an indeterminate value and reading it is undefined behaviour — the optimiser may assume it never happens. Only globals and `static` locals are zero-initialised. It compiles; `-Wall` may or may not warn.

2. A List-initialisation rejects narrowing conversions, and `double` → `int` loses information. `int x = 3.5;` would silently store 3 — which is why braces are the recommended form.

3. B `std::uint8_t` is an alias for `unsigned char`, and the stream prints character types as characters. Use `static_cast<int>(b)` or `+b` to print the number.

4. C Both are `std::size_t`, the unsigned type sized to index any object in memory. Mixing it with `int` in comparisons is the source of the signed/unsigned warnings in lesson 4.

5. B The standard only requires `long` ≥ 32 bits. The Linux x86-64 ABI makes it 8 bytes; Windows keeps it at 4. `long long` is guaranteed at least 64 bits everywhere.

6. B For floating types `min()` is the smallest positive normal value (about 2.2×10^{-308}). Use `lowest()` to start a maximum search; `min()` would make every negative number lose.

7. B Empty braces value-initialise: zero for every arithmetic type, `false` for `bool`, `nullptr` for pointers. `T x;` is indeterminate, `T x();` declares a *function* returning `T` — the "most vexing parse" — and `NULL` is a null pointer constant meant for pointers, not a general zero.

LESSON 2 · 15 MIN

Integer arithmetic and bits

Integer arithmetic in C++ compiles to the CPU's own add, multiply and shift instructions and nothing more: no range check, no promotion to a big integer, no exception. That is where the speed comes from, and it is why the rules at the edges are yours to know. Signed and unsigned types behave differently when a result does not fit, small types quietly become `int` before any arithmetic happens, division rounds in a definite direction, and the shift operators have holes the standard refuses to fill. This lesson states each rule, shows how to detect overflow without committing it, and covers the bit operators and the C++20 `<bit>` header that make masks and flags readable.

Two's complement

Since C++20 signed integers are two's complement by definition: an n-bit type holds -2^{n-1} … $2^{n-1} - 1$, the top bit is the sign, and negating a value is "invert every bit, add one". The range is asymmetric — one more negative value than positive — so `-x` and `std::abs(x)` have exactly one input that does not fit: `std::numeric_limits<int>::min()`. Every rule below follows from that picture.

Signed overflow is undefined behaviour

```
int big = std::numeric_limits<int>::max();    // 2147483647
int next = big + 1;                           // undefined behaviour — not "wra
```

The hardware would wrap, and an unoptimised build usually shows the wrapped value, but the language says the program has no meaning from that point on. The optimiser takes the promise literally: `if (x + 1 > x)` is folded to `true`, a loop bounded by `i * 2 < n` is rewritten on the assumption that `i * 2` never wrapped, and a check you wrote *after* the overflow is deleted as dead code. Overflow has to be detected **before** it happens, in one of three ways:

```cpp
// 1. Widen: do the arithmetic in a type that cannot overflow, then range-chec
long long wide = static_cast<long long>(a) + b;
bool fits = wide ⩾ std::numeric_limits<int>::min() && wide ⩽ std::numeric_li

// 2. Compare against the limit first — the textbook form.
bool willOverflow = b > 0 ? a > std::numeric_limits<int>::max() - b
                          : a < std::numeric_limits<int>::min() - b;

// 3. Ask the compiler. GCC and Clang both provide this; the wrapped result is
int result;
if (__builtin_add_overflow(a, b, &result)) { /* overflowed; result holds the w
```

`__builtin_sub_overflow` and `__builtin_mul_overflow` are the siblings, and they work for `long long` too — multiplication is where widening runs out, because the product of two 64-bit values needs 128 bits. C++26 adds `std::add_sat` and friends, which clamp at the limit instead of overflowing; treat that as reading for now.

Unsigned arithmetic wraps

Unsigned types are defined modulo 2^n: `0u - 1` is 4 294 967 295, `4294967295u + 1` is 0, and neither is an error. That is the behaviour hashes, checksums and wrapping counters want. It is also the behaviour that turns `v.size() - 1` into 18 446 744 073 709 551 615 when the vector is empty, and makes `for (std::size_t i = n - 1; i ⩾ 0; --i)` loop for ever, because an unsigned value is always ≥ 0 (Module 3 has the idioms). Prefer `std::ssize(v)` — C++20, returns a signed count — or `i + 1 < v.size()` over subtracting from a size.

Integer promotion and the usual arithmetic conversions

No arithmetic is ever done in a type narrower than `int`. `bool`, `char`, `short` and their unsigned forms are **promoted** to `int` first:

```cpp
unsigned char a = 200, b = 100;
auto sum = a + b;      // int 300, not an unsigned char that wrapped to 44
short s = 30000;
int product = s * s;   // 900000000 — computed in int, fits
```

Then the **usual arithmetic conversions** bring two operands to one common type: the wider wins; at equal width the unsigned one wins, so `3u + (-5)` converts `-5` to unsigned and

yields 4 294 967 294; a floating operand converts the integer one. The "unsigned wins" rule is what makes `-1 < 1u` false — lesson 4 returns to it.

Division and remainder

`/` on integers truncates toward zero, `%` gives a remainder with the sign of the dividend, and `(a / b) * b + a % b == a` always holds:

`a`	`b`	`a / b`	`a % b`
7	2	3	1
−7	2	−3	−1
7	−2	−3	1
−7	−2	3	−1

Two cases are undefined: dividing by zero, and `std::numeric_limits<int>::min() / -1`, whose answer does not fit. A negative remainder is rarely what an index or a clock wants; the floor-modulo idiom `((a % m) + m) % m` gives 0 … m − 1 for any `a` and positive `m`.

The bitwise operators

Operator	Meaning	With `5` (`101`) and `3` (`011`)
`a & b`	and	`1`
`a | b`	or	`7`
`a ^ b`	exclusive or	`6`
`~a`	complement	`~5` is `-6` as `int`, `4294967290` as `unsigned`
`a << n`	shift left	`5 << 1` is `10`
`a >> n`	shift right	`5 >> 1` is `2`

Precedence bites: `&` and `|` bind more loosely than `==`, so `x & 1 == 0` parses as `x & (1 == 0)` and is always 0. Parenthesise every bitwise test.

Shifts and their holes

`x << n` multiplies by 2^n and `x >> n` divides by 2^n. C++20 nailed down two formerly grey cases: left-shifting a signed value is defined modulo 2^n, so `1 << 31` is `-2147483648`; and right-shifting a negative value is arithmetic — the sign bit is copied in, so `-16 >> 2` is `-4`. Still undefined: a shift count that is negative or at least the width of the promoted type. `1 << 32` is undefined; so is `1 << 40` even when the result is assigned to a `long long`, because the `1` is an `int` — write `1LL << 40` or `1ull << 40`. The safe habit is to do bit work on unsigned types of a known width.

Masks

```
std::uint32_t flags = 0;
flags |= 1u << 3;                                  // set bit 3
flags &= ~(1u << 3);                               // clear bit 3
flags ^= 1u << 3;                                  // toggle bit 3
bool on = (flags >> 3) & 1u;                       // test bit 3
std::uint32_t x = 40;                              // 101000
bool powerOfTwo = x != 0 && (x & (x - 1)) == 0;    // x & (x - 1) clears the
std::uint32_t low = x & (~x + 1u);                 // isolates the lowest se
```

Name the bits — `constexpr std::uint32_t kVerbose = 1u << 3;` — and the same lines read as English. Lesson 6 shows the same idea with an enumeration.

`<bit>` (C++20)

```
#include <bit>
std::popcount(255u)        // 8  — set bits
std::bit_width(255u)       // 8  — bits needed; bit_width(256u) is 9
std::has_single_bit(64u)   // true — exactly one bit set, i.e. a power of two
std::countl_zero(1u)       // 31 — leading zeros in a 32-bit value
std::countr_zero(8u)       // 3  — trailing zeros
std::bit_ceil(17u)         // 32 — next power of two up; bit_floor(17u) is 16
std::rotl(0x80000001u, 1)  // 3  — rotate; the bit that falls off comes back
```

Every one of them takes an **unsigned** argument — pass an `int` and the call does not compile, which is the header's way of enforcing the rule above. Print a pattern with

`std::format("{:b}", x)`, zero-padded with `{:08b}`, and in hex with `{:#x}`.

Pitfalls

`int` products: `n * (n + 1) / 2` for `n = 100000` overflows before the division. Compute in `long long`.

`std::abs(std::numeric_limits<int>::min())` is undefined; so is `-x` for that one value.

`x & 1 == 0` — precedence. Write `(x & 1) == 0`.

`1 << 40` in an `int` expression, even when assigned to a `long long`.

`-7 % 2` is `-1`, so `x % 2 == 1` misses negative odd numbers; test `x % 2 != 0`.

Key takeaways

Signed overflow is undefined behaviour and the optimiser assumes it never happens; detect it before it happens by widening, by comparing against `numeric_limits`, or with `__builtin_add_overflow`.

Unsigned arithmetic wraps modulo 2^n — right for hashes, fatal for `size() - 1`.

Everything narrower than `int` is promoted to `int`; at equal width the unsigned operand wins.

`/` truncates toward zero and `%` follows the dividend; `((a % m) + m) % m` for a non-negative result.

Do bit manipulation on unsigned types; `<bit>` gives `popcount`, `bit_width`, `has_single_bit` and friends; parenthesise every mask test.

EXERCISE 1

Checked sum

Read an integer `n`, then `n` integers that each fit in an `int`. Add them into an `int` accumulator one at a time, checking every addition with `__builtin_add_overflow(sum, x, &next)` (available on GCC and Clang). If the whole sum fits, print `sum=<sum>`. If adding the value at 0-based index `i` would overflow, print `overflow at index <i> (wrapped value <w>)` — `w` is the wrapped result the builtin stores — and stop.

The point is to detect overflow *without* committing it: a plain `sum + x` that overflows is undefined behaviour, and widening to `long long` is the other correct approach; here, use the builtin.

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

Example: input `2 2147483647 1` → `overflow at index 1 (wrapped value -2147483648)`

INPUT 1

```
3
1 2 3
```

OUTPUT 1

```
sum=6
```

INPUT 2

```
2
2147483647 1
```

OUTPUT 2

```
overflow at index 1 (wrapped value
-2147483648)
```

REFERENCE SOLUTION

```cpp
#include <iostream>

int main() {
    int n;
    std::cin >> n;
    int sum = 0;
    for (int i = 0; i < n; ++i) {
        int x;
        std::cin >> x;
        int next = 0;
        if (__builtin_add_overflow(sum, x, &next)) {
            std::cout << "overflow at index " << i << " (wrapped value " << next <<
            return 0;
        }
        sum = next;
    }
    std::cout << "sum=" << sum << '\n';
    return 0;
}
```

EXERCISE 2

Eight-bit flag register

Keep an eight-bit register in a `std::uint8_t`. The first line is its starting value (0–255), the second an integer `n`, then `n` operations, one per line: `set i`, `clear i`, `toggle i` or `test i`, with `0 ⩽ i ⩽ 7`. Apply each with a mask (`1u << i`); for every `test` print `bit <i> = 0` or `bit <i> = 1`. After the last operation print `value=<decimal> bits=<8 binary digits> count=<number of set bits>` — use `std::format("{:08b}", reg)` for the binary field and `std::popcount` from `<bit>` for the count.

Input: the start value, `n`, then `n` operations. **Output:** one line per `test`, then the summary line.

```
0
4
set 0
set 3
test 3
toggle 0
```

→

```
bit 3 = 1
value=8 bits=00001000 count=1
```

INPUT 1

```
0
4
set 0
set 3
test 3
toggle 0
```

OUTPUT 1

```
bit 3 = 1
value=8 bits=00001000 count=1
```

INPUT 2

```
5
2
test 0
test 1
```

OUTPUT 2

```
bit 0 = 1
bit 1 = 0
value=5 bits=00000101 count=2
```

REFERENCE SOLUTION

```cpp
#include <bit>
#include <cstdint>
#include <format>
#include <iostream>
#include <string>

int main() {
    int start;
    int n;
    std::cin >> start >> n;
    std::uint8_t reg = static_cast<std::uint8_t>(start);
    for (int k = 0; k < n; ++k) {
        std::string op;
        int i;
        std::cin >> op >> i;
        std::uint8_t mask = static_cast<std::uint8_t>(1u << i);
        // Every bitwise expression below is computed in int (promotion) and narrowe
        if (op == "set") {
            reg = static_cast<std::uint8_t>(reg | mask);
        } else if (op == "clear") {
            reg = static_cast<std::uint8_t>(reg & ~mask);
        } else if (op == "toggle") {
            reg = static_cast<std::uint8_t>(reg ^ mask);
        } else if (op == "test") {
            std::cout << "bit " << i << " = " << ((reg >> i) & 1u) << '\n';
        }
    }
    std::cout << std::format("value={} bits={:08b} count={}\n", reg, reg, std::popcc
    return 0;
}
```

QUIZ

1. What does `std::cout << -7 / 2 << ' ' << -7 % 2;` print?

A. `-4 1`

B. `-3 -1`

C. `-3 1`

D. `-4 -1`

2. `int big = std::numeric_limits<int>::max(); big + 1` …

A. Wraps to `-2147483648`

B. Is undefined behaviour — the optimiser may assume it never happens

C. Throws `std::overflow_error`

D. Saturates at `2147483647`

3. What does this print?

```
unsigned u = 0;
std::cout << u - 1;
```

A. `-1`

B. `4294967295`

C. Undefined behaviour

D. A compile error

4. What are the type and value of `s`?

```
unsigned char a = 200, b = 100;
auto s = a + b;
```

A. `unsigned char`, 44

B. `int`, 300

C. `unsigned int`, 300

D. A compile error

5. In C++20, with `int` operands, what are `1 << 31` and `1 << 32`?

A. Both undefined behaviour

B. `-2147483648` and `0`

C. `-2147483648` (defined since C++20) and undefined behaviour (shift by the type's width)

D. Both compile errors

6. What are `std::popcount(0b1011u)` and `std::bit_width(0b1011u)`?

A. 3 and 4

B. 4 and 3

C. 3 and 3

D. 4 and 4

7. How does the compiler parse `x & 1 == 0`?

A. `(x & 1) == 0`

B. `x & (1 == 0)` — always 0, because `==` binds tighter than `&`

C. A compile error

D. `x && (1 == 0)`

ANSWER KEY · INTEGER ARITHMETIC AND BITS

1. B Integer division truncates toward zero, so `-7 / 2` is `-3`, and the remainder takes the dividend's sign so that `(a / b) * b + a % b == a`: `-3 * 2 + (-1) == -7`. Floor division (`-4 1`) is what Python does, not C++.

2. B Signed overflow is undefined. The hardware would wrap and a debug build often shows that, but the compiler is entitled to fold `big + 1 > big` to `true` and delete checks written after the fact. Detect overflow before it happens.

3. B Unsigned arithmetic is defined modulo 2^{32}, so `0 - 1` wraps to the largest `unsigned`. Well-defined, and the reason `v.size() - 1` on an empty vector is a huge number rather than `-1`.

4. B Both operands are promoted to `int` before the addition, so nothing wraps: `s` is an `int` holding 300. Assigning the result *back* to an `unsigned char` would narrow it to 44.

5. C C++20 defines left shift of a signed value modulo 2^n, so `1 << 31` is `INT_MIN`. A shift count that is negative or ≥ the width remains undefined, whatever type you assign the result to — write `1LL << 32`.

6. A `popcount` counts set bits: three. `bit_width` is the number of bits needed to represent the value — the position of the highest set bit plus one: four.

7. B The comparison operators have higher precedence than the bitwise `&`, `^` and `|`, a C inheritance everyone regrets. Always parenthesise the mask test: `(x & 1) == 0`.

LESSON 3 · 15 MIN

Floating point — representation, error and printing

A `double` is not a real number. It is a 64-bit pattern that stands for one of a finite set of fractions, and every value between two of them is rounded to the nearer one before you ever get to use it. Almost every floating-point surprise — `0.1 + 0.2 ≠ 0.3`, a total that is off by a cent, a loop that never terminates — is that rounding, seen from a distance. This lesson gives the representation in enough detail to predict the surprises, the comparison idiom that replaces `==`, the printing controls that make output deterministic, and the two cases where the right answer is not a `double` at all.

The shape of a double

IEEE 754 binary64 spends its 64 bits as 1 sign bit, 11 exponent bits and 52 fraction bits; a hidden leading 1 makes 53 bits of precision. The value is ±1.*fraction* × 2^exponent. `float` (binary32) has 1, 8 and 23 bits for 24 bits of precision, and `long double` on this platform is the 80-bit x87 format with 64. Since $2^{53} \approx 9 \times 10^{15}$, a `double` carries between 15 and 17 significant decimal digits: `std::numeric_limits<double>::digits10` is 15 (any 15-digit decimal survives a round trip) and `max_digits10` is 17 (the digits needed to print a double so that it reads back exactly). `float` gives 6 and 9.

```
#include <bit>
std::uint32_t bits = std::bit_cast<std::uint32_t>(1.0f);      // 0×3F800000
std::cout << std::format("{:032b}\n", bits);                  // 0 01111111 000
```

`std::bit_cast` (C++20) copies the bytes of one value into another type of the same size; it is the well-defined way to look at a float's bits.

Representation error

Only fractions whose denominator is a power of two are exact: 0.5, 0.25, 0.375, 3.75. Everything else is rounded. The literal `0.1` is stored as 0.1000000000000000055511151231257827…, `0.2` is a little above 0.2 as well, and their sum

rounds to 0.30000000000000004, while the literal `0.3` rounds to 0.2999999999999999. So:

```
std::cout << (0.1 + 0.2 == 0.3) << '\n';                    // 0
std::cout << 0.1 + 0.2 << '\n';                              // 0.3        (6 s
std::cout << std::setprecision(17) << 0.1 + 0.2 << '\n';     // 0.300000000000(
```

The default stream precision prints 6 significant digits, which is why the error is invisible until you look for it. Integers up to 2^{53} are exact, so a `double` is a fine container for a whole number that size — and a terrible one beyond it.

Comparing: a tolerance, not `==`

```
#include <cmath>
bool nearlyEqual(double a, double b, double relTol = 1e-9, double absTol = 1e-
    double scale = std::max(std::fabs(a), std::fabs(b));
    return std::fabs(a - b) ⩽ std::max(relTol * scale, absTol);
}
```

A *relative* tolerance scales with the magnitude — 1 part in 10^9 is the right question for both 3.0 and 3×10^{12} — but it collapses to nothing near zero, which is what the *absolute* term is for. `std::numeric_limits<double>::epsilon()` (2.22×10^{-16}) is the gap between 1.0 and the next double, not a general tolerance: near 10^6 the gap is already 1.16×10^{-10}, so comparing values that size against `epsilon` is the same as `==`. The same rounding is why `for (double x = 0; x ≠ 1.0; x += 0.1)` never terminates: count in integers and scale, `for (int i = 0; i ⩽ 10; ++i) { double x = i / 10.0; … }`.

Big integers and absorption

Beyond 2^{53} = 9 007 199 254 740 992 doubles are spaced two apart, so `9007199254740993.0` prints as …992 and `1e16 + 1.0 == 1e16` is true: the 1 is *absorbed*. `float` loses whole numbers at 2^{24} = 16 777 216 — `16777216.0f + 1.0f == 16777216.0f`. Never keep an id, a count or a currency amount in a floating type.

Accumulation order

```
float total = 0.0f;
for (int i = 0; i < 1'000'000; ++i) total += 0.1f;
std::cout << std::fixed << std::setprecision(1) << total << '\n';   // 100958.
```

Every addition rounds, and once `total` is large each 0.1 loses most of its bits before it is added. Rules that follow: sum in `double` even when the inputs are `float`; add small values before large ones; for serious work use compensated (Kahan) summation, or a `long double` accumulator, which on this platform has 11 more bits than `double`. Ten additions of `0.1` in `double` give 0.99999999999999989 — wrong, but wrong by 10^{-16} rather than 10^{3}.

Printing deterministically

The default `std::cout` format is "6 significant digits, scientific when the exponent gets large", which is not what a judge or a user wants:

```
std::cout << 1234567.0 << ' ' << 1e6 << ' ' << 2.50 << '\n';
std::cout << std::setprecision(3) << 1234.5678 << '\n';
std::cout << std::fixed << std::setprecision(2) << 1e6 << ' ' << 1.0 / 3 << '
std::cout << std::format("{:.2f} {:.3e} {}\n", 2.5, 1234.5, 0.1 + 0.2);
```

`std::fixed` makes the precision mean *decimals* and, like every stream manipulator, sticks until changed; `std::defaultfloat` restores the default. `std::format` sets the precision per placeholder: `{:.2f}` fixed, `{:.3e}` scientific, `{:g}` general. A bare `{}` prints the *shortest* text that reads back to the same double — `0.30000000000000004`, and `1e+05` for `100000.0` — so judged output always carries an explicit precision.

The printed digit is decided on the exact stored value. `2.675` is stored just below 2.675, so `{:.2f}` prints `2.67`; a genuine tie such as `0.125` (exactly representable) goes to the even digit, `0.12`. This is why money should not be a `double` — the fix is at the end of the lesson.

Rounding functions

Call	Result	Rule
`std::round(2.5)` / `std::round(-2.5)`	`3` / `-3`	half away from zero
`std::floor(-2.5)`	`-3`	toward −∞
`std::ceil(-2.5)`	`-2`	toward +∞
`std::trunc(-2.5)`	`-2`	toward zero
`static_cast<int>(-2.5)`	`-2`	toward zero; undefined if out of range
`std::lround(2.5)`	`3L`	`round`, returned as `long`

All live in `<cmath>` and, except `lround` / `llround`, return a `double` — `std::round(2.5)` prints `3`, but it is `3.0`.

NaN and infinity

`1.0 / 0.0` is `inf` (floating division by zero is *defined*; integer division by zero is not), and `0.0 / 0.0` or `std::sqrt(-1.0)` is NaN, "not a number". NaN compares false with everything, including itself, so `x ≠ x` is the classic test and `std::isnan(x)`, `std::isinf(x)` and `std::isfinite(x)` are the readable ones. A NaN silently poisons every calculation it touches, and printing one gives `nan` or `-nan` depending on a sign bit you did not choose — test for it, never print it. Negative zero exists too: `-0.0 == 0.0` is true, but it prints as `-0`.

When the answer is not a `double`

`long double` buys headroom on x86-64 — 18 significant digits, sums of a million doubles without visible drift — at the cost of portability (on MSVC it is just `double`). When the values are exact by nature, use integers: money in `long long` cents, time in integer nanoseconds, coordinates in integer micro-degrees. The arithmetic is exact, the comparisons are `==`, and the output is assembled from integers:

```
long long cents = 1725;
std::cout << std::format("{}.{:02}\n", cents / 100, cents % 100);   // 17.25
```

Pitfalls

`==` on computed doubles; `std::setprecision` without `std::fixed` (significant digits, not decimals).

`float` accumulators and `float` in general — `double` costs nothing extra.

A `double` holding an id or a count above 2^{53}, or a `float` above 2^{24}.

Printing a NaN, or a value the default format will show in scientific notation.

Rounding money in a `double` and being surprised by `2.67`.

Key takeaways

A `double` has 53 bits of precision, 15–17 decimal digits; only dyadic fractions are exact, so `0.1 + 0.2 != 0.3`.

Compare with a relative tolerance plus an absolute floor; `epsilon()` is the spacing at 1.0, not a tolerance.

Whole numbers are exact up to 2^{53} (`float`: 2^{24}); beyond that small addends are absorbed. Sum small to large, in `double` or `long double`.

Always print with an explicit precision: `std::fixed << std::setprecision(2)` or `std::format("{:.2f}")`; ties round on the exact stored value.

NaN is unequal to itself and must be tested, not printed; exact quantities such as money belong in `long long` cents.

EXERCISE 1

Statistics to two decimals

Read an integer `n` (at least 1), then `n` real numbers. Print four lines — `sum=`, `mean=`, `min=` and `max=` — each value with exactly two decimals, using `std::fixed << std::setprecision(2)` from `<iomanip>` (or `std::format("{:.2f}", x)`).

Accumulate in a `double`; start the minimum from `std::numeric_limits<double>::max()` and the maximum from `lowest()`.

Input: `n`, then `n` numbers (they may be written as `3`, `4.2` or `1e6`). **Output:** four lines.

Example: input `4 1.5 2.5 3 4.2` →

```
sum=11.20
mean=2.80
min=1.50
max=4.20
```

INPUT 1

```
4
1.5 2.5 3 4.2
```

OUTPUT 1

```
sum=11.20
mean=2.80
min=1.50
max=4.20
```

INPUT 2

```
3
10 20 33
```

OUTPUT 2

```
sum=63.00
mean=21.00
min=10.00
max=33.00
```

REFERENCE SOLUTION

```cpp
#include <iomanip>
#include <iostream>
#include <limits>

int main() {
    int n;
    std::cin >> n;
    double sum = 0.0;
    double lo = std::numeric_limits<double>::max();
    double hi = std::numeric_limits<double>::lowest();
    for (int i = 0; i < n; ++i) {
        double x;
        std::cin >> x;
        sum += x;
        if (x < lo) lo = x;
        if (x > hi) hi = x;
    }
    std::cout << std::fixed << std::setprecision(2);
    std::cout << "sum=" << sum << '\n'
              << "mean=" << sum / n << '\n'
              << "min=" << lo << '\n'
              << "max=" << hi << '\n';
    return 0;
}
```

EXERCISE 2

Near enough

Write the tolerance comparison from the lesson. The first line holds two tolerances, `rel` and `abs`; the second an integer `n`; then `n` lines each with two numbers `a` and `b`. Two numbers are *equal* when `|a - b| ≤ max(rel * max(|a|, |b|), abs)`. For the `i`-th pair (1-based) print `<i>: equal` or `<i>: different`. Use `std::fabs` from `<cmath>` and `std::max` from `<algorithm>`; never compare the doubles with `==`.

Input: `rel abs`, then `n`, then `n` pairs. **Output:** `n` lines.

```
1e-9 1e-12
3
0.30000000000000004 0.3
1000000000000001 1000000000000000
100 100.5
```

→

```
1: equal
2: equal
3: different
```

The second pair differs by 1, which is within one part in 10^9 of 10^{15} — a relative tolerance scales with the magnitude.

INPUT 1

```
1e-9 1e-12
3
0.30000000000000004 0.3
1000000000000001 1000000000000000
100 100.5
```

OUTPUT 1

```
1: equal
2: equal
3: different
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>

int main() {
    double relTol;
    double absTol;
    int n;
    std::cin >> relTol >> absTol >> n;
    for (int i = 1; i ⩽ n; ++i) {
        double a;
        double b;
        std::cin >> a >> b;
        double scale = std::max(std::fabs(a), std::fabs(b));
        bool equal = std::fabs(a - b) ⩽ std::max(relTol * scale, absTol);
        std::cout << i << ": " << (equal ? "equal" : "different") << '\n';
    }
    return 0;
}
```

QUIZ

1. What does `std::cout << (0.1 + 0.2 == 0.3);` print?

A. `1`

B. `0`

C. Undefined behaviour

D. A compile error

2. What does `std::cout << 1234567.0;` print with the default stream settings?

A. `1234567`

B. `1.23457e+06`

C. `1234567.0`

D. `1234567.000000`

3. What does `std::cout << std::setprecision(2) << 3.14159;` print?

A. `3.14`

B. `3.1`

C. `3.14159`

D. `3`

4. What are `std::round(-2.5)` and `static_cast<int>(-2.5)`?

A. `-3` and `-2`

B. `-2` and `-2`

C. `-3` and `-3`

D. `-2` and `-3`

5. Which expression is true exactly when `x` is NaN?

A. `x == NAN`

B. `x != x`

C. `x == std::numeric_limits<double>::quiet_NaN()`

D. `std::isinf(x)`

6. Which type should hold an amount of money?

A. `double`

B. `float`

C. `long long` holding cents (or the smallest unit)

D. `long double`

7. What does this print?

```
float f = 16777216.0f;
std::cout << (f + 1.0f == f);
```

A. `0`

B. `1` — 16777217 is not representable in 24 bits of precision, so the 1 is absorbed

C. Undefined behaviour

D. A compile error

ANSWER KEY · FLOATING POINT — REPRESENTATION, ERROR AND PRINTING

1. B Neither 0.1 nor 0.2 nor 0.3 is exactly representable; the rounded sum is 0.30000000000000004 and the literal 0.3 rounds to 0.29999999999999999. Compare with a tolerance, or keep such values as integers.

2. B The default is six significant digits in general format, which switches to scientific notation once the exponent is at least 6. `std::fixed` (or `std::format("{:.0f}")`) prints `1234567`.

3. B Without `std::fixed`, the precision is the number of *significant* digits, so two of them give `3.1`. Add `std::fixed` to make it two decimals: `3.14`.

4. A `std::round` rounds halves away from zero, giving `-3.0`. A cast truncates toward zero, giving `-2`. `std::floor(-2.5)` would be `-3` and `std::ceil(-2.5)` `-2`.

5. B NaN compares unequal to everything, including itself, so `x ≠ x` is the classic test and `std::isnan(x)` the readable one. Any `==` against a NaN is false, which is why the first and third options never succeed.

6. C Money is exact by nature and decimal by convention; a binary floating type cannot hold 0.10 exactly, and totals drift. Integer cents are exact, compare with `==`, and are formatted from `cents / 100` and `cents % 100`.

7. B A `float` has 24 bits of precision, so above 2^{24} = 16 777 216 consecutive floats are two apart. `16777217.0f` rounds back to 16 777 216 and the comparison is true. `double` has the same problem above 2^{53}.

LESSON 4 · 14 MIN

Conversions and casts

C++ converts between types constantly and silently. Every mixed-type expression, every argument passed to a differently typed parameter, every `if` on a number is a conversion. Most are harmless promotions; a handful lose information, and two of those — narrowing a floating value to an integer, and mixing signed with unsigned — are behind a large share of real bugs. This lesson lists the implicit conversions and what each does to a value, shows how brace-initialisation refuses the lossy ones, and explains the four named casts and why the C-style cast has no place in new code.

Where implicit conversions happen

```
double half = 1 / 2;          // 0.0: the int division happens first, then 0 c
int n = 2.9;                  // 2: truncated, no warning under -Wall
if (n) { }                    // n converts to bool
void f(long long x);
f(n);                         // n widens to long long — a promotion, value pr
```

Initialisation and assignment, the operands of an operator, arguments, return values and conditions are all conversion sites. A **promotion** goes to a wider type and preserves the value; a **conversion** may not. The compiler reports neither unless you ask (`-Wconversion` is not part of `-Wall`).

Narrowing conversions

Written	Stored	Why
`int i = 3.99;`	`3`	floating → integer truncates toward zero
`int i = 3e10;`	undefined	out of range for `int`
`unsigned u = -1;`	`4294967295`	signed → unsigned is modular, always defined
`int k = 3000000000LL;`	`-1294967296`	wider → narrower keeps the low bits (modular since C++20)
`char c = 300;`	`44`	300 mod 256
`float f = 16777217;`	`16777216.0f`	integer → float rounds to the nearest representable
`bool b = 0.1;`	`true`	anything non-zero

Only one row is undefined behaviour — a floating value that does not fit the target integer — but every row loses something. Braces make the compiler say so:

```
int i{3.99};        // error: narrowing conversion of '3.99' from 'double' to
unsigned u{-1};     // error
char c{300};        // error
char d{65};         // fine: a constant that fits is not narrowing
long long big = 5;
int j{big};         // error: a run-time long long might not fit, even though t
```

When you *want* the conversion, say so with `static_cast`; the reader then knows the loss was considered and `-Wall` stops second-guessing you.

---

bool

Zero and null are `false`; everything else is `true`. `if (x = 5)` compiles — assignment, then the 5 converts to `true` — which is why `-Wall` warns about an assignment used as a condition and why some people write `5 == x`. Going the other way, `true` is `1` in arithmetic and `false` is `0`; `std::cout << std::boolalpha` prints the words instead.

---

char and int

A `char` in arithmetic becomes an `int`, which is the whole basis of text processing by hand:

```
char c = 'q';
int code = c;                              // 113
int digit = '7' - '0';                     // 7
char next = static_cast<char>(c + 1);      // 'r' — c + 1 is an int and must be
bool lower = c ⩾ 'a' && c ⩽ 'z';
```

Because `char` is signed here, a byte ≥ 128 is a negative `int`; before passing one to `<cctype>` or using it as an index, convert through `unsigned char` (Module 5 makes this a habit).

Signed and unsigned

`int` → `unsigned` wraps modulo 2^{32}, and since C++20 so does the reverse. The trouble is not the conversion itself but that it happens *inside comparisons* without asking:

```
unsigned count = 3;
int delta = -5;
std::cout << count + delta << '\n';   // 4294967294 — delta became unsigned
std::cout << (delta < count) << '\n'; // 0 — "-5 < 3" is false, because -5 bec
```

The usual arithmetic conversions (lesson 2) convert the signed operand, and `-Wall` warns with `-Wsign-compare`. The classic sighting is `for (int i = 0; i < v.size(); ++i)`. Fixes, in order of preference:

```
for (std::size_t i = 0; i < v.size(); ++i)   // match the type
for (int i = 0; i < std::ssize(v); ++i)      // C++20: a signed size
std::cmp_less(delta, count)                  // C++20 <utility>: compares mathl
std::in_range<int>(someUnsigned)             // does the value fit the target
```

`std::cmp_equal`, `cmp_not_equal`, `cmp_less`, `cmp_greater`, `cmp_less_equal` and `cmp_greater_equal` are the whole family. Casting the unsigned side to `long long` also works when it is known to fit.

Floating to integer

`static_cast<int>(2.9)` is `2` and `static_cast<int>(-2.9)` is `-2` — truncation toward zero, the same as `std::trunc`. If the truncated value does not fit the target, the behaviour is undefined, so `static_cast<int>(1e10)` is not "some big number", it is a bug. Range-check first, use `std::llround` when rounding is wanted, and reach for `long long` when the magnitude is uncertain.

The four casts

```
double ratio = static_cast<double>(hits) / total;              // numeric; for
auto level = static_cast<Level>(2);                            // int → enum (
const unsigned char* bytes = reinterpret_cast<const unsigned char*>(&ratio);
void legacyPrint(char* text);
legacyPrint(const_cast<char*>(name.c_str()));                  // only if lega
```

`static_cast` is the everyday cast: every numeric conversion, enum ↔ integer, `void*` → `T*`, and up or down a class hierarchy without a run-time check. If a conversion is *plausible*, `static_cast` does it.

`reinterpret_cast` reinterprets the bits of a pointer or an integer as another pointer type. It is for byte-level I/O (Module 18) and talking to hardware; almost every other use is undefined behaviour. For *values* — the bits of a `float` as a `uint32_t` — use `std::bit_cast`.

`const_cast` adds or removes `const`. Removing it to call a badly declared legacy function is legitimate; writing through the result to an object that was born `const` is undefined behaviour.

`dynamic_cast` is a checked cast down a polymorphic hierarchy, in Module 10.

Each cast does one thing, is easy to search for, and refuses conversions outside its job.

Why not `(int)x`

A C-style cast `(T)x` — and its functional twin `T(x)` — tries `const_cast`, then `static_cast`, then `static_cast` plus `const_cast`, then `reinterpret_cast`, and silently uses the first that compiles. `(int)somePointer`, `(char*)someConstString` and `(Base*)unrelatedPointer` all compile and all mean something different from what a reader expects. It cannot be grepped for, and it keeps compiling when the types around it change into something the cast should have refused. Treat one in new code as a review comment waiting to happen.

Pitfalls

`double avg = sum / n;` with integer operands: the division is integral before the conversion. Cast one operand first.

`long long area = width * height;` where both are `int`: the product overflows in `int` before the widening. Cast one operand first.

`if (index < v.size())` with a negative `index`: the comparison is unsigned and the bounds check passes.

`static_cast<int>(hugeDouble)` is undefined, not saturated.

`(int)x` where `x` turns out to be a pointer.

Key takeaways

Conversions happen at every assignment, operand, argument, return and condition; promotions preserve values, conversions may not, and `-Wall` stays quiet.

Brace-initialisation rejects narrowing; `int x{3.7}` and `unsigned u{-1}` are compile errors.

Mixed signed/unsigned expressions convert the signed side; use matching types, `std::ssize`, or `std::cmp_less` and friends.

Floating → integer truncates toward zero and is undefined when out of range.

`static_cast` for value conversions, `reinterpret_cast` for bytes and hardware, `const_cast` for legacy signatures, `std::bit_cast` for bit views; never a C-style cast.

EXERCISE 1

Caesar shift in char arithmetic

Read an integer `k` — any sign, any size — then one line of text. Shift every letter `k` places along the alphabet, wrapping (`z` + 1 is `a`) and keeping the case; leave every other character unchanged. Print the shifted line.

Do the arithmetic in `int` (`c - 'a'` gives the letter's index) and narrow back to `char` with `static_cast<char>`, never a C-style cast. Normalise `k` to 0…25 first with the floor-modulo idiom `((k % 26) + 26) % 26`. Read `k` with `>>`, then `std::cin.ignore();` before `std::getline` — the pattern from Module 1.

Input: `k`, then one line. **Output:** the shifted line.

Example: input `3` / `Hello, World!` → `Khoor, Zruog!`

INPUT 1

```
3
Hello, World!
```

OUTPUT 1

```
Khoor, Zruog!
```

INPUT 2

```
-3
Khoor, Zruog!
```

OUTPUT 2

```
Hello, World!
```

REFERENCE SOLUTION

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

int main() {
    long long k;
    std::cin >> k;
    std::cin.ignore();
    std::string line;
    std::getline(std::cin, line);
    int shift = static_cast<int>(((k % 26) + 26) % 26);
    for (char& c : line) {
        if (c >= 'a' && c <= 'z') {
            c = static_cast<char>('a' + (c - 'a' + shift) % 26);
        } else if (c >= 'A' && c <= 'Z') {
            c = static_cast<char>('A' + (c - 'A' + shift) % 26);
        }
    }
    std::cout << line << '\n';
    return 0;
}
```

EXERCISE 2

Compare across the sign divide

Read an integer `n`, then `n` lines each holding an `int a` (possibly negative) and an `unsigned b` (possibly above `2147483647`). For each pair print `<a> vs <b>: naive <r1>, correct <r2>`, where each result is `less`, `equal` or `greater`.

The *naive* result is what the built-in `<` does — it converts `a` to `unsigned` first, so compute it with `static_cast<unsigned>(a)` against `b`. The *correct* result compares the mathematical values with `std::cmp_less` and `std::cmp_equal` from `<utility>` (C++20).

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

Example: input `2` / `-1 1` / `5 5` →

```
-1 vs 1: naive greater, correct less
5 vs 5: naive equal, correct equal
```

INPUT 1

```
3
-1 1
5 5
7 3
```

OUTPUT 1

```
-1 vs 1: naive greater, correct less
5 vs 5: naive equal, correct equal
7 vs 3: naive greater, correct greater
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        int a;
        unsigned b;
        std::cin >> a >> b;
        // What the usual arithmetic conversions do to a < b: the int becomes unsigr
        unsigned converted = static_cast<unsigned>(a);
        std::string naive = converted < b ? "less" : converted == b ? "equal" : "gre
        std::string correct = std::cmp_less(a, b) ? "less" : std::cmp_equal(a, b) ?
        std::cout << a << " vs " << b << ": naive " << naive << ", correct " << corr
    }
    return 0;
}
```

QUIZ

1. What happens here?

```
int x = 3.99;
int y{3.99};
```

A. `x` is 3; the second line is a compile error

B. Both are 3

C. Both are 4

D. Both lines are compile errors

2. What is the value of `u` after `unsigned u = -1;`?

A. A compile error

B. `4294967295`

C. Undefined behaviour

D. `-1`

3. `std::vector<int> v;` is empty and `int i = -1;`. What does `if (i < v.size())` do?

A. Enters the branch: −1 is less than 0

B. Skips the branch: `i` converts to `std::size_t` 18446744073709551615, which is not less than 0

C. A compile error

D. Undefined behaviour

4. What is `static_cast<int>(1e10)`?

A. `1410065408` — the low 32 bits

B. `2147483647` — saturated

C. Undefined behaviour: the truncated value does not fit an `int`

D. A compile error

5. Which cast converts a `double d` to an `int`?

A. `static_cast<int>(d)`

B. `reinterpret_cast<int>(d)`

C. `const_cast<int>(d)`

D. `dynamic_cast<int>(d)`

6. `(char*)text.c_str()` compiles. Which named cast did the C-style cast silently perform?

A. `static_cast`

B. `const_cast` — it removed `const`, which is exactly the kind of thing a reader should have been told

C. `reinterpret_cast`

D. None; it is a compile error

7. What does `std::cmp_less(-1, 1u)` return?

A. `false`, like `-1 < 1u`

B. `true` — it compares the mathematical values

C. A compile error

D. Unspecified

ANSWER KEY · CONVERSIONS AND CASTS

1. A Copy-initialisation performs the narrowing conversion silently (truncating to 3, with no warning under `-Wall`). List-initialisation rejects narrowing, so `int y{3.99}` does not compile.

2. B Signed → unsigned conversion is always defined and modular: −1 mod 2^{32} is 4 294 967 295. Only the braced form `unsigned u{-1}` is refused.

3. B The usual arithmetic conversions turn the signed operand unsigned, so the bounds check passes a negative index through. `-Wall` warns (`-Wsign-compare`); use `std::cmp_less(i, v.size())` or a `std::size_t` index.

4. C Floating → integer conversion is undefined when the truncated value is out of range; it is the one narrowing in the table that is not merely lossy. Range-check first, or convert to `long long`.

5. A `static_cast` performs value conversions. `reinterpret_cast` cannot take a `double` at all (it works on pointers and integers), `const_cast` only changes constness, and `dynamic_cast` is for polymorphic class pointers and references.

6. B `c_str()` returns `const char*`; a C-style cast tries the named casts in turn and used `const_cast`. Written as `const_cast<char*>(...)` the intent is visible and searchable; the C-style form hides it and would also happily reinterpret an unrelated pointer.

7. B The `<utility>` comparison functions (C++20) are defined on the values, not on the converted bits, so `cmp_less(-1, 1u)` is `true` while the built-in `-1 < 1u` converts −1 to 4294967295 and is `false`.

LESSON 5 · 14 MIN

const, constexpr and auto

Three keywords decide how much the compiler knows about a variable. `const` promises the value will not change after initialisation; `constexpr` promises it is known while compiling; `auto` asks the compiler to work the type out from the initialiser. Used well, each removes a class of mistakes — an accidental assignment, a run-time computation that should have been folded, a silently wrong numeric type. Used carelessly, each hides something. This lesson gives the rules, including the deduction rules for `auto` that most people learn by tripping over them.

`const` variables

```
const int limit = 100;
limit = 200;                      // error: assignment of read-only variable
const int rounds = readCount();   // fine: const is about after initialisatio
const std::vector<int> table{1, 2, 3};
table.push_back(4);               // error: only const member functions may be
```

A `const` object must be initialised and cannot be assigned afterwards. The initial value may come from anywhere, including input. The habit worth forming is *const by default*: declare every local `const` unless it changes, so that the ones that do change are the ones a reader looks at. It also turns a whole family of typos into compile errors, and it lets the compiler keep the value in a register without ever checking memory again.

`const` references

```
int x = 5;
const int& view = x;      // an alias that cannot be used to modify x
const double& d = 2.5;    // binds to a temporary and extends its lifetime to
int& bad = 2.5;           // error: a non-const lvalue reference cannot bind t
```

A `const T&` binds to anything — a variable, a literal, the result of a function call — without copying, which is why it is the default parameter type for anything bigger than a register (Module 4). A plain `T&` binds only to something that already has a name.

`constexpr` variables

```
constexpr int kMaxUsers = 1'000;
constexpr double kPi = 3.141592653589793;
int buckets[kMaxUsers / 10];               // a constant expression is require
static_assert(kMaxUsers % 10 == 0);
constexpr int fromInput = readCount();     // error: not a constant expressio
```

`constexpr` on a variable means "initialised by a constant expression and usable wherever the compiler needs a number": array bounds, template arguments, `static_assert`, `case` labels. Every `constexpr` variable is also `const`; the reverse is not true.

`constexpr` functions

```
constexpr long long factorial(int n) {
    long long result = 1;
    for (int i = 2; i <= n; ++i) result *= i;
    return result;
}

static_assert(factorial(20) == 2432902008176640000LL);   // evaluated by the c
int n;
std::cin >> n;
std::cout << factorial(n) << '\n';                        // evaluated at run
```

A `constexpr` function is an ordinary function that the compiler is *allowed* to evaluate during compilation when every argument is a constant and the result is needed in a constant expression. Otherwise it is called at run time like any other. C++20 permits loops, local variables, `if`, and even `std::vector` and `std::string` inside the compile-time evaluation (as long as they are gone by the end of it); it forbids undefined behaviour, so a `constexpr` call that overflows is a compile error rather than a silent wrap — a useful side effect. `consteval` (C++20) marks a function that *must* be evaluated at compile time, and `constinit` a variable that must be initialised then; Module 16 goes deeper.

	`const`	`constexpr`
Value known at	run time is fine	compile time, always
Can be initialised from input	yes	no
Usable as an array bound or in `static_assert`	only if the initialiser was a constant	yes
Applies to functions	(member functions: Module 8)	yes — "may run at compile time"

`auto`: the deduction rules

`auto` uses the same rules as template argument deduction (Module 12), which come down to four cases:

```
const int cx = 10;
int x = 5;

auto a = cx;          // int — a COPY; the reference-ness and top-level const
a = 11;               // fine, a is a mutable int
auto& r = cx;         // const int& — auto& keeps the reference and therefore
const auto& cr = x;   // const int& — binds to anything, never copies, never m
auto* p = &x;         // int* — say the star when you mean a pointer
```

Rule 1 is the one that matters: `auto x = expr;` is always a copy. That is safe for an `int` and expensive for a `std::vector`, and it is the reason the range-for idiom is `for (const auto& item : items)` when reading and `for (auto& item : items)` when modifying — `for (auto item : items) item *= 2;` doubles a copy and leaves the container untouched.

Literals fix the deduced type, so the literal's suffix is the type annotation:

```
auto i = 0;            // int
auto u = 0u;           // unsigned
auto l = 0L;           // long
auto ll = 0LL;         // long long
auto d = 0.0;          // double
auto f = 0.0f;         // float
auto c = 'c';          // char
auto s = "text";       // const char* — NOT std::string
auto big = 10'000'000'000;   // long: the literal did not fit an int
auto n = v.size();     // std::size_t — the right type without spelling it
```

`auto list = {1, 2, 3};` deduces `std::initializer_list<int>`, which is almost never wanted; write `std::vector<int>`.

`decltype` in brief

`decltype(expr)` names the type of an expression without evaluating it. `decltype(x)` for a variable is its declared type; `decltype((x))`, with the extra parentheses, is the type of the *expression* `x`, an lvalue, so it is `int&`. It matters in templates (Module 12) and in `decltype(auto)` (Module 16); in everyday code you will meet it in `decltype(v)::value_type` and similar spellings.

When `auto` helps, and when it hides

`auto` earns its place when the type is obvious from the right-hand side (`auto p = std::make_unique<Widget>()`), when only the compiler can name it (a lambda, an iterator over a nested map), and when it prevents a silent conversion (`auto n = v.size()` is `std::size_t`; `int n = v.size()` narrows). It hides when the numeric type is the point:

```
auto total = 0;                // int — overflows at 2 billion; write 0LL or
auto mean = sum / count;       // integer division if both are integers; nobo
auto name = "Ada";             // const char*; name.size() does not compile
```

The working rule: `auto` for types that are obvious or unnameable; the explicit type for numbers whose width or signedness matters, and for anything a reader would otherwise have to look up.

Pitfalls

`auto x = someConstRef;` makes a mutable copy — surprising when the copy is a large container.

`auto& r = 5;` does not compile; `const auto& r = 5;` does.

`for (auto e : v) e = 0;` modifies nothing.

`auto sum = 0;` then summing millions of values: it is an `int`.

`constexpr int n = readInt();` — a run-time value can be `const`, never `constexpr`.

Key takeaways

`const` is a promise not to modify after initialisation; the value may come from run time. Make locals `const` by default.

`const T&` binds to anything, including temporaries, without a copy; `T&` needs an lvalue.

`constexpr` variables are compile-time constants; `constexpr` functions run at compile time when their arguments are constants and at run time otherwise, and `static_assert` proves it.

`auto x = expr` copies and drops references and top-level `const`; `auto&` and `const auto&` keep them; literal suffixes choose the numeric type.

Use `auto` when the type is obvious or unnameable; spell the type when a number's width or signedness matters.

EXERCISE 1

constexpr power and primes

Implement two `constexpr` functions: `long long power(long long base, int exponent)` (`exponent ⩾ 0`; the result fits in a `long long`) and `bool isPrime(long long n)` (trial division while `d * d ⩽ n`; `n ⩾ 0`). Then uncomment the two `static_assert` lines in the starter — they only compile if the functions really are usable at compile time — and answer queries: read `q`, then `q` lines that are either `pow <base> <exponent>` (print the value) or `prime <n>` (print `yes` or `no`).

Input: `q`, then `q` queries. **Output:** `q` lines.

Example: input `3` / `pow 2 10` / `prime 97` / `prime 1` →

```
1024
yes
no
```

INPUT 1

```
4
pow 2 10
prime 97
prime 1
pow 3 0
```

OUTPUT 1

```
1024
yes
no
1
```

REFERENCE SOLUTION

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

constexpr long long power(long long base, int exponent) {
    long long result = 1;
    for (int i = 0; i < exponent; ++i) result *= base;
    return result;
}

constexpr bool isPrime(long long n) {
    if (n < 2) return false;
    for (long long d = 2; d * d <= n; ++d) {
        if (n % d == 0) return false;
    }
    return true;
}

// Evaluated by the compiler: a wrong implementation fails to build.
static_assert(power(2, 10) == 1024);
static_assert(isPrime(97) && !isPrime(91));

int main() {
    int q;
    std::cin >> q;
    for (int i = 0; i < q; ++i) {
        std::string kind;
        std::cin >> kind;
        if (kind == "pow") {
            long long base;
            int exponent;
            std::cin >> base >> exponent;
            std::cout << power(base, exponent) << '\n';
        } else {
            long long n;
            std::cin >> n;
            std::cout << (isPrime(n) ? "yes" : "no") << '\n';
        }
    }
    return 0;
}
```

EXERCISE 2

Deduced accumulators

Read an integer `n` (at least 1), then `n` integers in `int` range. Print `sum=<exact sum>` (it may exceed an `int`), `int_mean=<sum / n in integer division>`, `mean=<sum / n with two decimals>` and

`max=<largest value>`.

Declare the accumulators with `auto` and let the initialiser fix the type: `auto sum = 0LL;` (a `long long` — `auto sum = 0;` would be an `int` and overflow) and `auto best = std::numeric_limits<int>::min();`. Make the derived values `const auto`, and cast to `double` for the real mean — `sum / n` with two integers is integer division no matter what `auto` deduces.

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

Example: input `4 1 2 3 4` →

```
sum=10
int_mean=2
mean=2.50
max=4
```

INPUT 1

```
4
1 2 3 4
```

OUTPUT 1

```
sum=10
int_mean=2
mean=2.50
max=4
```

INPUT 2

```
3
2000000000 2000000000 2000000000
```

OUTPUT 2

```
sum=6000000000
int_mean=2000000000
mean=2000000000.00
max=2000000000
```

REFERENCE SOLUTION

```cpp
#include <format>
#include <iostream>
#include <limits>

int main() {
    int n;
    std::cin >> n;
    auto sum = 0LL;                                    // long long: the LL suffix dec
    auto best = std::numeric_limits<int>::min();       // int
    for (int i = 0; i < n; ++i) {
        int x;
        std::cin >> x;
        sum += x;
        if (x > best) best = x;
    }
    const auto intMean = sum / n;                          // long long, truncated t
    const auto mean = static_cast<double>(sum) / n;        // double
    std::cout << "sum=" << sum << '\n'
              << "int_mean=" << intMean << '\n'
              << std::format("mean={:.2f}\n", mean)
              << "max=" << best << '\n';
    return 0;
}
```

QUIZ

1. What happens here?

```cpp
const int cx = 10;
auto a = cx;
a = 5;
```

A. Compile error: `a` is `const int`

B. It compiles: `auto` drops top-level `const`, so `a` is a mutable `int` copy

C. Undefined behaviour

D. `cx` becomes 5

2. What happens here?

```
auto s = "hello";
std::cout << s.size();
```

A. Prints `5`

B. Compile error: `s` is a `const char*`, which has no `size()`

C. Prints `6`

D. Undefined behaviour

3. `int readInt();` reads a number at run time. What does `constexpr int n = readInt();` do?

A. Compiles; `n` is set when the program runs

B. Compile error: the initialiser is not a constant expression

C. `n` is 0

D. Undefined behaviour

4. After `for (auto x : v) x *= 2;`, what has changed in `v`?

A. Every element is doubled

B. Nothing: `auto x` copies each element

C. A compile error

D. Only the first element

5. A `constexpr` function is called with an argument read from `std::cin`. What happens?

A. A compile error

B. It runs at run time, like any other function

C. It returns 0

D. Its result is cached across runs

6. Which of these compile?

```
auto& r1 = 5;
const auto& r2 = 5;
```

A. Neither

B. Only `r2`: a `const` reference binds to a temporary and extends its lifetime

C. Both

D. Only `r1`

7. What is the type of `r` in `int x = 1; decltype((x)) r = x;`?

A. `int`

B. `int&` — the parenthesised expression `x` is an lvalue

C. `const int`

D. A compile error

ANSWER KEY · CONST, CONSTEXPR AND AUTO

1. B `auto x = expr` deduces like a by-value template parameter: references and top-level `const` are dropped and `a` is a fresh `int`. To keep the constness write `const auto a = cx;` or `auto& a = cx;`.

2. B A string literal is a `const char` array and `auto` deduces the decayed pointer type. Write `std::string s = "hello";` or `auto s = "hello"s;` with `using namespace std::string_literals;`.

3. B `constexpr` demands a compile-time value. A run-time value can be `const` — `const int n = readInt();` is fine — but never `constexpr`.

4. B `auto` without `&` copies. Write `for (auto& x : v)` to modify in place and `for (const auto& x : v)` to read without copying.

5. B `constexpr` means the function *may* be evaluated at compile time when its arguments are constants. With run-time arguments it is an ordinary call — which is exactly why one implementation serves both `static_assert` and the input loop.

6. B A non-const lvalue reference needs an lvalue; `5` is a temporary. A `const T&` may bind to a temporary, which then lives as long as the reference — the rule that makes `const T&` parameters accept literals.

7. B `decltype(x)` gives the declared type `int`; with the extra parentheses the operand is an *expression*, and for an lvalue expression `decltype` yields a reference. A subtle rule that mostly matters inside templates and `decltype(auto)`.

LESSON 6 · 14 MIN

Literals and enumerations

A literal is a value spelled straight into the source, and each spelling has a type that the compiler settles before your variable gets a say — which is why `auto big = 3000000000;` is a `long`, why `1 << 40` is undefined but `1LL << 40` is fine, and why `"text"` is not a `std::string`. Enumerations give names to a set of integer constants, and the C++11 scoped form fixes the two problems the C form always had. This lesson is the reference for both.

Integer literals

```
int dec = 255;
int hex = 0xFF;           // 255
int bin = 0b1111'1111;    // 255  (binary since C++14; the ' is a digit separa
int oct = 0377;           // 255  — a LEADING ZERO means octal; 017 is 15, and
long long big = 1'000'000'000'000LL;
```

The suffix fixes the type: `u` / `U` unsigned, `l` / `L` long, `ll` / `LL` long long, and combinations such as `ul` and `ull`. Without a suffix a decimal literal gets the first of `int`, `long`, `long long` that can hold it, so on this platform `2147483647` is an `int` and `2147483648` is a `long`. Hexadecimal and binary literals may also become `unsigned` before growing, which is why `0xFFFFFFFF` is an `unsigned int`, not a `long`. `-1` is not a literal at all; it is unary minus applied to `1`, which is why `-2147483648` does not have the type you expect (the `2147483648` is a `long` first).

Floating literals

`1.5`, `1e3`, `1.5e-3`, `.5` and `5.` are all `double`. `1.5f` is a `float` and `1.5L` a `long double`. A literal with no dot and no exponent is an integer, so `1 / 2` is `0` and `1.0 / 2` is `0.5`; the fix for integer division is one `.0`. C++17 added hexadecimal floats (`0x1p-3` is 0.125), which you will read in numeric libraries and rarely write.

Character literals and escape sequences

`'a'` is a `char` with the value 97. Characters that cannot be typed use an escape:

Escape	Meaning	Escape	Meaning
`\n`	newline	`\\`	backslash
`\t`	tab	`\'` `\"`	quote characters
`\r`	carriage return	`\0`	the null character, value 0
`\x41`	hex code: `'A'`	`\101`	octal code: `'A'`

`'\0'` terminates C strings and is what an empty `char` variable holds after `char c{}`. A literal with several characters, `'ab'`, is an `int` with an implementation-defined value; compilers warn, and it is never what you meant. `u8'a'`, `u'a'`, `U'a'` and `L'a'` spell the other character types.

String literals

```
const char* s = "abc";                  // "abc" has type const char[4] — three
std::cout << "line one\n" "line two\n"; // adjacent literals are joined by the
using namespace std::string_literals;
auto owned = "abc"s;                    // std::string, thanks to the s suffix
```

A string literal is an array of `const char` that ends in `'\0'`; it lives for the whole program and decays to a `const char*` when you take it by pointer (Module 6). `sizeof("abc")` is 4 and `std::strlen("abc")` is 3. Writing into a string literal is undefined behaviour, which is why the pointer type is `const char*`. The `s` suffix from `<string>` gives a real `std::string`, and `sv` a `std::string_view` (Module 5).

Raw string literals

```
std::cout << R"(C:\temp\new "quoted")" << '\n';      // C:\temp\new "quoted"
std::cout << R"delim(a )" b)delim" << '\n';          // a )" b  — a custom del
```

`R"(...)"` keeps backslashes, quotes and even newlines exactly as written. Regular expressions, Windows paths, JSON and SQL snippets go in raw literals; the alternative is a thicket of `\\`.

The other literals

`true` and `false` are `bool`. `nullptr` is the null pointer literal, of type `std::nullptr_t`, and the only spelling you should use (Module 6 says why `0` and `NULL` are worse). User-defined literals such as `250ms` and `2h` come from `<chrono>` and from your own `operator""` (Module 11).

Unscoped enumerations

```
enum Colour { Red, Green = 5, Blue };   // Red 0, Green 5, Blue 6

Colour c = Green;
int n = Blue;              // implicit conversion to int: 6
bool same = c == 5;        // compiles — compares as int
Colour d = 5;              // error: no implicit int → enum
Colour e = static_cast<Colour>(5);   // fine, and not checked: static_cast<Co
```

The C-style `enum` has two problems. The names spill into the enclosing scope, so two enumerations in the same namespace cannot both have a `None`. And the values convert to `int` at the slightest excuse, so `Colour` and `Weekday` compare and add without complaint. The underlying type is implementation-chosen (`int` here unless every value fits a narrower type and you ask for it) and can be fixed: `enum Colour : std::uint8_t { … }`.

Scoped enumerations

```
enum class Level : std::uint8_t { Low = 1, Mid, High };   // Mid 2, High 3; or

Level lv = Level::Mid;
int n = lv;                                  // error: no implicit conversion
int m = static_cast<int>(lv);                // 2 — on purpose, in writing
Level from = static_cast<Level>(3);          // High; validate the integer fi
bool ok = lv < Level::High;                  // comparison within one enum is
std::underlying_type_t<Level> raw = static_cast<std::underlying_type_t<Level>>
```

`enum class` qualifies every name (`Level::Mid`), converts in neither direction implicitly, and defaults its underlying type to `int` unless you fix it. That is the whole difference, and it is enough to make `enum class` the default choice. C++23 adds `std::to_underlying(lv)` for

the last line above; this track's runtime is C++20, so spell the cast. C++20's `using enum Level;` inside a block or a `switch` lets you write the bare names where they are unambiguous.

Switching on an enumeration

```
const char* label(Level lv) {
    switch (lv) {
        case Level::Low:  return "low";
        case Level::Mid:  return "mid";
        case Level::High: return "high";
    }
    return "?";   // reachable: a cast can produce any value
}
```

`switch` is Module 3's subject; the enum-specific points are these. With `-Wall`, `-Wswitch` warns when a `switch` over an enumeration without a `default` omits an enumerator — leave the `default` out and the compiler tells you when someone adds `Level::Critical`. And C++ has no built-in way to print an enumerator's *name*; a `switch` or an array of `const char*` indexed by the value is how every codebase does it.

Enumerations as flags

```
enum Permission : unsigned { Exec = 1u << 0, Write = 1u << 1, Read = 1u << 2 }

unsigned mode = Read | Write;                 // 6 — unscoped, so | works on t
bool canWrite = (mode & Write) != 0;
mode &= ~static_cast<unsigned>(Exec);         // clear a flag
```

One bit per enumerator, powers of two, combined with `|` and tested with `&` — lesson 2's masks with names. An unscoped enum with a fixed underlying type is the pragmatic choice because the operators work out of the box; with `enum class` you overload `operator|` and `operator&` yourself (Module 11) or cast at every use.

Pitfalls

`017` is 15, and `08` does not compile — a leading zero is octal.

`1 << 40` is undefined; the `1` needs a suffix: `1LL << 40`.

`"a"` is a `const char[2]`; `'a'` is a `char`. `"abc" + 1` is pointer arithmetic (Module 6), not concatenation.

`static_cast<Level>(99)` compiles and produces a `Level` holding 99; validate before casting.

Comparing two different unscoped enums compiles and means nothing.

Key takeaways

Every literal has a type: unsuffixed decimals are the smallest of `int` / `long` / `long long` that fits, `u` / `L` / `LL` fix it, `0x` / `0b` /leading-`0` set the base, `'` separates digits.

`'a'` is a one-byte integer; `"a"` is a `const char` array ending in `'\0'`; raw strings `R"(...)"` take no escapes.

Unscoped enums leak names and convert to `int`; `enum class` does neither and needs `static_cast` both ways.

Fix the underlying type when the size matters; validate an integer before casting it to an enum.

Flags are powers of two combined with `|` and tested with `&`; a `switch` without `default` lets `-Wswitch` catch a missing case.

EXERCISE 1

Literal decoder

Read an integer `n`, then `n` tokens, each a C++ integer literal with no suffix: decimal, hexadecimal (`0x` or `0X`), binary (`0b` or `0B`) or octal (a leading `0`), possibly containing `'` digit separators. Every value fits in a `long long`. Print `<token> = <decimal value>` for each.

Decode by hand: drop the separators, choose the base from the prefix, then accumulate `value = value * base + digit` where the digit comes from `char` arithmetic (`c - '0'`, `c - 'a' + 10`, `c - 'A' + 10`).

Input: `n`, then `n` tokens separated by whitespace. **Output:** `n` lines.

Example: input `4 0x1F 0b1010 017 1'000'000` →

```
0x1F = 31
0b1010 = 10
017 = 15
1'000'000 = 1000000
```

INPUT 1

```
5
0x1F
0b1010
017
1'000'000
0
```

OUTPUT 1

```
0x1F = 31
0b1010 = 10
017 = 15
1'000'000 = 1000000
0 = 0
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string token;
        std::cin >> token;
        std::string digits;
        for (char c : token) {
            if (c != '\'') digits += c;   // digit separators are ignored by the com
        }
        int base = 10;
        std::size_t start = 0;
        if (digits.size() > 1 && digits[0] == '0') {
            char p = digits[1];
            if (p == 'x' || p == 'X') {
                base = 16;
                start = 2;
            } else if (p == 'b' || p == 'B') {
                base = 2;
                start = 2;
            } else {
                base = 8;   // a leading zero means octal: 017 is 15
                start = 1;
            }
        }
        long long value = 0;
        for (std::size_t k = start; k < digits.size(); ++k) {
            char c = digits[k];
            int digit = 0;
            if (c >= '0' && c <= '9') {
                digit = c - '0';
            } else if (c >= 'a' && c <= 'f') {
                digit = c - 'a' + 10;
            } else {
                digit = c - 'A' + 10;
            }
            value = value * base + digit;
        }
        std::cout << token << " = " << value << '\n';
    }
    return 0;
}
```

EXERCISE 2

A scoped weekday enum

Define `enum class Weekday : std::uint8_t { Mon = 1, Tue, Wed, Thu, Fri, Sat, Sun };`. Read an integer `n`, then `n` integers. For each value from 1 to 7, convert it with `static_cast<Weekday>` and print `<value> → <Name> weekday` or `<value> → <Name> weekend` (`Sat` and `Sun` are the weekend); for any other value print `<value> → invalid` — validate *before* casting, because the cast itself checks nothing.

Map the enumerator to its name with a `switch` on the enum (Module 3 covers `switch` in full; it is the standard way to give an enumerator a name). Compare enumerators with `==`, never through their integer values.

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

Example: input `3 1 6 9` →

```
1 → Mon weekday
6 → Sat weekend
9 → invalid
```

INPUT 1

```
4
1
6
9
3
```

OUTPUT 1

```
1 → Mon weekday
6 → Sat weekend
9 → invalid
3 → Wed weekday
```

REFERENCE SOLUTION

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

enum class Weekday : std::uint8_t { Mon = 1, Tue, Wed, Thu, Fri, Sat, Sun };

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        int v;
        std::cin >> v;
        if (v < 1 || v > 7) {
            std::cout << v << " -> invalid\n";   // static_cast<Weekday>(256) would
            continue;
        }
        Weekday day = static_cast<Weekday>(v);
        std::string name;
        switch (day) {
            case Weekday::Mon: name = "Mon"; break;
            case Weekday::Tue: name = "Tue"; break;
            case Weekday::Wed: name = "Wed"; break;
            case Weekday::Thu: name = "Thu"; break;
            case Weekday::Fri: name = "Fri"; break;
            case Weekday::Sat: name = "Sat"; break;
            case Weekday::Sun: name = "Sun"; break;
        }
        bool weekend = day == Weekday::Sat || day == Weekday::Sun;
        std::cout << v << " -> " << name << (weekend ? " weekend" : " weekday") << '
    }
    return 0;
}
```

QUIZ

1. What does `std::cout << 017;` print?

A. `17`

B. `15`

C. A compile error

D. `7`

2. What is the type of `x` in `auto x = 3000000000;` on this platform?

A. `int`, with the value wrapped

B. `long` — the first of `int`, `long`, `long long` that holds the value

C. `unsigned`

D. A compile error

3. What does `std::cout << '\101';` print?

A. `\101`

B. `A`

C. `101`

D. A compile error

4. Given `enum Colour { Red, Green = 5, Blue };`, what is `int n = Blue;`?

A. `6`

B. `2`

C. A compile error

D. `5`

5. Given `enum class Level { Low, High };`, what does `int n = Level::High;` do?

A. `n` is 1

B. Compile error: a scoped enum has no implicit conversion to `int` — write `static_cast<int>(Level::High)`

C. `n` is 0

D. Undefined behaviour

6. What is `sizeof(Level)` for `enum class Level : std::uint8_t { Low, High };`?

A. 1

B. 4

C. 8

D. Unspecified

7. What does `std::cout << R"(a\nb)";` print?

A. `a`, a newline, `b`

B. The four characters `a\nb` — a raw string processes no escapes

C. A compile error

D. `ab`

ANSWER KEY · LITERALS AND ENUMERATIONS

1. B A leading zero makes an octal literal: 0·8 + 1·8 + 7 = 15. `08` would not compile, since 8 is not an octal digit. Zero-padded numbers copied from data are the classic way to meet this.

2. B An unsuffixed decimal literal takes the narrowest of `int`, `long`, `long long` that fits. 3 000 000 000 exceeds `int`, so it is a `long` (8 bytes here). Add `LL` if you mean `long long` regardless of platform.

3. B A backslash followed by up to three octal digits is an octal escape: $101_8 = 65$ = `'A'`. The hexadecimal form `'\x41'` is the same character.

4. A Each enumerator without an initialiser is one more than the previous, so `Blue` is 6. An unscoped enum converts implicitly to `int`, so the initialisation compiles — the leak that `enum class` closes.

5. B Scoped enumerations convert in neither direction implicitly. That is the point: a `Level` cannot be added to a `Colour` by accident. The cast makes the conversion visible.

6. A A fixed underlying type sets the enum's size and representation; `std::uint8_t` is one byte. Without the `: type`, a scoped enum defaults to `int` (4 bytes).

7. B Inside `R"(...)"` a backslash is just a backslash, which is why raw strings suit regular expressions and Windows paths. `"a\nb"` without the `R` would print two lines.

CHECKPOINT · 25 MIN

Checkpoint — Fundamental types

This checkpoint covers the whole module: the built-in types and their sizes on this platform, integer arithmetic with its overflow rules and bit operators, floating-point representation and printing, conversions and the four casts, `const`, `constexpr` and `auto`, and literals and scoped enumerations.

How it works. Fourteen questions and three programs; 70% on the questions and every program accepted clears 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:

- What happens when an `int` holding `2147483647` is incremented, and how do you detect it first?
- What are `-7 / 2` and `-7 % 2`?
- Why is `0.1 + 0.2 == 0.3` false, and what do you compare with instead?
- Which initialisation form rejects `int x = 3.7`?
- What does `auto s = "text";` deduce?
- Why is `-1 < 1u` false, and which C++20 function gets it right?
- What does `enum class` change about `int n = Red;`?

The three programs are a checked running product, an invoice kept in integer cents, and a literal classifier that names each token's type and value — read its input format twice; the cases include suffixes and a `char` literal.

EXERCISE 1

Checked product

Read an integer `n`, then `n` integers that each fit in a `long long`. Multiply them into a `long long` accumulator that starts at 1, checking every step with `__builtin_mul_overflow(product, x, &next)`. If everything fits print `product=<value>`; if multiplying by the factor at 0-based index `i` would overflow, print `overflow at index <i>` and stop. Widening cannot help here — the product of two 64-bit values needs 128 bits — so the checked builtin is the tool.

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

Example: input `3 2 3 4` → `product=24`; input `2 3037000500 3037000500` → `overflow at index 1`

INPUT 1

```
3
2 3 4
```

OUTPUT 1

```
product=24
```

INPUT 2

```
2
3037000500 3037000500
```

OUTPUT 2

```
overflow at index 1
```

REFERENCE SOLUTION

```cpp
#include <iostream>

int main() {
    int n;
    std::cin >> n;
    long long product = 1;
    for (int i = 0; i < n; ++i) {
        long long x;
        std::cin >> x;
        long long next = 0;
        if (__builtin_mul_overflow(product, x, &next)) {
            std::cout << "overflow at index " << i << '\n';
            return 0;
        }
        product = next;
    }
    std::cout << "product=" << product << '\n';
    return 0;
}
```

EXERCISE 2

Invoice in cents

Read an integer `n`, then `n` lines `quantity unit_cents`, then one line with an integer tax percentage. Compute the subtotal as the sum of `quantity * unit_cents`, the tax as `subtotal * percent / 100` rounded half up — in integers, `(subtotal * percent + 50) / 100` — and the total. Print `subtotal=`, `tax=` and `total=`, each as `<dollars>.<two-digit cents>` built from integers with `std::format("{}.{:02}", cents / 100, cents % 100)`.

No `double` anywhere: every amount is a `long long` number of cents, so the arithmetic is exact and the rounding rule is yours.

Input: `n`, `n` item lines, then the tax percentage. **Output:** three lines.

Example: input `2` / `3 199` / `1 1000` / `8` →

```
subtotal=15.97
tax=1.28
total=17.25
```

INPUT 1

```
2
3 199
1 1000
8
```

OUTPUT 1

```
subtotal=15.97
tax=1.28
total=17.25
```

INPUT 2

```
1
3 333
15
```

OUTPUT 2

```
subtotal=9.99
tax=1.50
total=11.49
```

REFERENCE SOLUTION

```cpp
#include <format>
#include <iostream>

int main() {
    int n;
    std::cin >> n;
    long long subtotal = 0;
    for (int i = 0; i < n; ++i) {
        long long quantity;
        long long unitCents;
        std::cin >> quantity >> unitCents;
        subtotal += quantity * unitCents;
    }
    long long percent;
    std::cin >> percent;
    long long tax = (subtotal * percent + 50) / 100;   // half up: 12.5 cents become
    long long total = subtotal + tax;
    std::cout << std::format("subtotal={}.{:02}\n", subtotal / 100, subtotal % 100)
              << std::format("tax={}.{:02}\n", tax / 100, tax % 100)
              << std::format("total={}.{:02}\n", total / 100, total % 100);
    return 0;
}
```

EXERCISE 3

What literal is that?

Read an integer `n`, then `n` tokens. Classify each as a C++ literal and print `<token>: <type> <value>`:

- digits only → `int` if the value is at most `2147483647`, otherwise `long`;
- digits with a suffix `u` → `unsigned` if at most `4294967295`, otherwise `unsigned long`; `l` → `long`; `ll` → `long long`; `ul` → `unsigned long`; `ull` → `unsigned long long`;
- digits with a decimal point → `double`, or `float` with an `f` suffix, or `long double` with an `L` suffix — print the value with exactly three decimals;
- `'x'` (three characters) → `char` and the character's code;
- `true` / `false` → `bool 1` / `bool 0`;
- anything else → `<token>: unknown`.

Parse by hand: walk the token once, accumulating the integer part in an `unsigned long long` and the fraction in a `double` (`fraction = fraction * 10 + digit`, `scale *= 10`), then treat everything after the digits as the suffix. Every value fits the type named.

Input: `n`, then `n` tokens separated by whitespace. **Output:** `n` lines.

Example: input `4 42 42u 3.5 'A'` →

```
42: int 42
42u: unsigned 42
3.5: double 3.500
'A': char 65
```

INPUT 1

```
6
42
42u
3.5
2.5f
'A'
true
```

OUTPUT 1

```
42: int 42
42u: unsigned 42
3.5: double 3.500
2.5f: float 2.500
'A': char 65
true: bool 1
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string t;
        std::cin >> t;
        std::cout << t << ": ";
        if (t == "true" || t == "false") {
            std::cout << "bool " << (t == "true" ? 1 : 0) << '\n';
            continue;
        }
        if (t.size() == 3 && t[0] == '\'' && t[2] == '\'') {
            std::cout << "char " << static_cast<int>(t[1]) << '\n';   // a char is a
            continue;
        }
        unsigned long long whole = 0;
        double fraction = 0.0;
        double scale = 1.0;
        bool seenDot = false;
        bool anyDigit = false;
        std::size_t pos = 0;
        while (pos < t.size() && ((t[pos] >= '0' && t[pos] <= '9') || t[pos] == '.')
            char c = t[pos];
            if (c == '.') {
                if (seenDot) break;
                seenDot = true;
            } else if (seenDot) {
                fraction = fraction * 10 + (c - '0');
                scale *= 10;
            } else {
                whole = whole * 10 + static_cast<unsigned long long>(c - '0');
                anyDigit = true;
            }
            if (c != '.') anyDigit = true;
            ++pos;
        }
        std::string suffix;
        while (pos < t.size()) suffix += t[pos++];
        if (!anyDigit) {
            std::cout << "unknown\n";
            continue;
        }
        if (seenDot) {
            double value = static_cast<double>(whole) + fraction / scale;
```

```
            if (suffix.empty()) {
                std::cout << std::format("double {:.3f}\n", value);
            } else if (suffix == "f") {
                std::cout << std::format("float {:.3f}\n", static_cast<float>(value)
            } else if (suffix == "L") {
                std::cout << std::format("long double {:.3f}\n", static_cast<long do
            } else {
                std::cout << "unknown\n";
            }
            continue;
        }
        if (suffix.empty()) {
            std::cout << (whole <= 2147483647ULL ? "int " : "long ") << whole << '\r
        } else if (suffix == "u") {
            std::cout << (whole <= 4294967295ULL ? "unsigned " : "unsigned long ") <
        } else if (suffix == "l") {
            std::cout << "long " << whole << '\n';
        } else if (suffix == "ll") {
            std::cout << "long long " << whole << '\n';
        } else if (suffix == "ul") {
            std::cout << "unsigned long " << whole << '\n';
        } else if (suffix == "ull") {
            std::cout << "unsigned long long " << whole << '\n';
        } else {
            std::cout << "unknown\n";
        }
    }
    return 0;
}
```

CHECKPOINT QUESTIONS

1. On this platform, what does `char c = 200; std::cout << static_cast<int>(c);` print?

A. `200`

B. `-56` — `char` is signed here and 200 wraps

C. A compile error

D. `72`

2. What does `int x{}; std::cout << x;` print?

A. An indeterminate value

B. `0`

C. A compile error

D. Nothing

3. What does `std::cout << -9 % 4;` print?

A. `-1`

B. `3`

C. `1`

D. `-3`

4. Is `if (a + b < a)` a valid way to detect that adding two positive `int`s overflowed?

A. Yes, on two's-complement hardware

B. No: the overflow is undefined behaviour before the check runs, and the optimiser may delete the test; use `__builtin_add_overflow` or compare against `numeric_limits` first

C. Only with `-O0`

D. Only for `unsigned`

5. What happens with `std::popcount(5)`?

A. Returns 2

B. Compile error: the `<bit>` functions take unsigned types only

C. Returns 1

D. Returns 3

6. What does `std::cout << std::fixed << std::setprecision(2) << 2.675;` print?

A. `2.68`

B. `2.67` — the stored value is slightly below 2.675, so it rounds down

C. `2.7`

D. `2.675`

7. What does `std::cout << (1e16 + 1.0 == 1e16);` print?

A. `0`

B. `1` — above 2^{53} doubles are spaced 2 apart, so the 1 is absorbed

C. Undefined behaviour

D. A compile error

8. `int sum = 7, n = 2; double avg = sum / n;` — what is `avg`?

A. `3.5`

B. `3.0` — the division is done in `int` before the conversion

C. `4.0`

D. A compile error

9. Which is the well-defined way to look at the bits of a `float f` as a `std::uint32_t`?

A. `reinterpret_cast<std::uint32_t>(f)`

B. `std::bit_cast<std::uint32_t>(f)`

C. `static_cast<std::uint32_t>(f)`

D. `*(std::uint32_t*)&f`

10. What is the type of `n` in `auto n = v.size();` for a `std::vector<int> v`?

A. `int`

B. `std::size_t`

C. `unsigned`

D. `long long`

11. `static_assert(square(4) == 16);` fails to compile with "non-constant condition". What is missing?

A. `#include <cassert>`

B. `square` must be declared `constexpr` so the compiler may evaluate it

C. `square` must be `inline`

D. A `const` on the parameter

12. Which expression is undefined behaviour?

A. `1LL << 40`

B. `1 << 40`

C. `1u << 31`

D. `1ull << 63`

13. Given `enum Colour { Red, Green, Blue };`, which line compiles?

A. `Colour c = 1;`

B. `Colour c = static_cast<Colour>(1);`

C. `Colour c{1.0};`

D. `Colour c = "Green";`

14. `for (int i = 0; i < v.size(); ++i)` compiles with a warning under `-Wall`. Which fix is wrong?

A. `for (std::size_t i = 0; i < v.size(); ++i)`

B. `for (int i = 0; i < std::ssize(v); ++i)`

C. `for (int i = 0; std::cmp_less(i, v.size()); ++i)`

D. `for (int i = 0; i < (int)v.size(); ++i)` — a C-style cast, hiding a narrowing that can fail for huge containers

ANSWER KEY · CHECKPOINT — FUNDAMENTAL TYPES

1. B `char` is signed on x86-64 Linux, so 200 is converted modulo 256 into the range −128…127: −56. Keep byte values in `unsigned char` or `std::uint8_t`; the braced form `char c{200}` would refuse to compile.

2. B Empty braces value-initialise the object, which for an `int` means zero — unlike `int x;`, whose value is indeterminate and whose read is undefined behaviour.

3. A `-9 / 4` truncates to `-2`, and the remainder keeps the dividend's sign so that `-2 * 4 + (-1) == -9`. Use `((a % m) + m) % m` for a result in 0…m−1.

4. B By the time `a + b` has wrapped the program has already invoked undefined behaviour, and the compiler is entitled to reason that `a + b < a` is impossible for positive `b`. Overflow must be prevented, not detected after the fact. (For `unsigned`, wrapping is defined and the check does work.)

5. B Every function in `<bit>` is constrained to unsigned integer types, which is deliberate — signed bit manipulation has holes. `std::popcount(5u)` returns 2.

6. B 2.675 has no exact binary representation; the nearest double is 2.67499999999999982…, and rounding happens on that value. This is the standard argument for keeping money in integer cents.

7. B A `double` has 53 bits of precision; 10^{16} exceeds $2^{53} \approx 9 \times 10^{15}$, so $10^{16} + 1$ rounds back to 10^{16}. Whole numbers that large belong in `long long`.

8. B Both operands are `int`, so `/` is integer division yielding 3, which is then converted to `3.0`. Cast one operand first: `static_cast<double>(sum) / n`.

9. B `std::bit_cast` (C++20, `<bit>`) copies the object representation into a same-sized type. `static_cast` converts the *value* (1.0f → 1), `reinterpret_cast` cannot take a non-pointer, and the pointer pun violates aliasing rules.

10. B `auto` deduces exactly what `size()` returns, `std::size_t`. That avoids the narrowing of `int n = v.size();` and keeps later comparisons with `.size()` warning-free.

11. B `static_assert` needs a constant expression, and only a `constexpr` (or `consteval`) function can be evaluated at compile time. Marking it `constexpr` costs nothing at run time.

12. B `1` is an `int` (32 bits); a shift count of 40 is at least the width of the type, which is undefined regardless of where the result is stored. The other three shift within their types' widths.

13. B There is no implicit conversion from `int` to an enumeration, even an unscoped one; only the reverse is implicit. `static_cast` performs the conversion — and performs no validation, so check the integer first.

14. D The first three keep the comparison mathematically correct. The C-style cast silences the warning by narrowing `size()` to `int`, which is wrong for sizes above 2^{31} and uses the cast form the lesson retires.

CodeKairo · C++ study plan · Module 2: Fundamental types. Solve the exercises online to track your progress and earn XP.