Module 12 of 20

Templates and generic programming

Function templates and deduction, Stack<T> and Pair<A, B> with CTAD, full and partial specialisation and the traits pattern, non-type parameters and fold expressions, C++20 concepts with constrained auto, and the type traits behind if constexpr dispatch.

Every container, algorithm and smart pointer you have used so far is a template: `std::vector<int>` is one class stamped out of `std::vector`, and `std::sort` is one function generated for each element type it meets. A template is a recipe the compiler follows the first time a type is supplied, and the result is ordinary code — a `max_of<int>` that is exactly the function you would have written by hand, with no run-time dispatch, no boxing and no cost. That is the zero-overhead principle applied to genericity, and it is why C++ can offer one `std::sort` for `int`, `std::string` and your own types and still compile each to the tightest loop. This module is where you stop only *using* templates and start writing them, and where you learn to read the errors they produce when a type does not fit.

The six lessons move from the mechanism to the tools that tame it. Function templates settle deduction — what the compiler reads from the arguments, why `max_of(3, 2.5)` is refused, when you name `T` yourself, how a template shares its name with plain overloads, and why the whole definition must sit in a header. Class templates build a `Stack<T>` and a `Pair<A, B>` with members defined outside the body, default arguments and C++17 class template argument deduction, and explain the `typename` the parser demands for dependent names. Specialisation writes the exception next to the rule — one exact type, or a pattern such as every pointer or every `std::pair<T, T>` — and fixes the rule that function templates are overloaded, never specialised. Non-type and variadic templates put a number or a whole pack of arguments in the parameter list and replace recursion with fold expressions. Concepts move a template's requirements into its signature so a bad argument fails at the call in two lines, and type traits close the module with `if constexpr` dispatch, `std::conditional_t`, `decltype`, and the SFINAE you must be able to read in older code.

The exercises are whole programs that read a type word from standard input and let the rest of the input follow from it: a `max_of` / `min_of` pair and a `clamp_to` served by one template for `int`,

`double` and `std::string`; a command-driven `Stack<T>` with out-of-class members; a `Pair<A, B>` built through CTAD with a defaulted second argument; a `Describe<T>` that tags `bool`, `const char*` and `std::string` with your own labels and recurses through vectors and pairs by partial specialisation; a fixed-capacity `Ring<T, N>` whose size is chosen among three instantiations at run time; a variadic report built entirely from fold expressions; an `average` whose overload is picked by `std::integral` and `std::floating_point`; a `Shape` concept written as a `requires` expression; and an `if constexpr` dispatcher that strips a forwarding reference with `std::remove_cvref_t`. The checkpoint adds a `Queue<T>` instantiated for the type on the first line, a statistics report whose overloads are chosen by concepts, and a `Table<K, V>` whose header comes from a `TypeName` trait.

CONTENTS

1	Function templates — one definition, every type	14 min
2	Class templates — Stack<T>, Pair<A, B> and deduction	14 min
3	Specialisation — when one type needs its own version	13 min
4	Non-type and variadic templates — numbers and packs as parameters	14 min
5	Concepts and constraints — saying what T must be	15 min
6	Type traits and metaprogramming — computing with types	15 min
✓	Checkpoint — Templates and generic programming	28 min

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

LESSON 1 · 14 MIN

Function templates — one definition, every type

A function fixes the types of its parameters when it is written; a function template takes the types as parameters too, and the compiler writes the function you need the first time you call it. `std::max`, `std::swap`, `std::sort` and `std::make_unique` are all function templates, which is why one `std::sort` sorts `int`s, `std::string`s and your own types without a line of dispatch. This lesson settles how a template is declared, how the compiler deduces `T` from a call, when you must spell the argument out, how a template coexists with ordinary overloads, and why a template lives in a header rather than a `.cpp` file.

A function that is a recipe

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

template <typename T>
T max_of(const T& a, const T& b) {
    return b < a ? a : b;
}

int main() {
    std::cout << max_of(3, 9) << '\n';
    std::cout << max_of(2.5, -1.0) << '\n';
    std::cout << max_of(std::string("pear"), std::string("apple")) << '\n';
    return 0;
}
```

`template <typename T>` introduces a *template parameter*: a placeholder type named `T` that the rest of the declaration may use as if it were a real type. `typename` and `class` mean the same thing in this position; older code writes `class`. The name `max_of` is not a function — it is a recipe, and `max_of<int>`, `max_of<double>` and `max_of<std::string>` are the three functions the compiler generated from it above. Generating one is called *instantiation*, and it happens on demand: a template that is never called produces no code, and a template that is called with four types produces four functions.

The body uses only `<`, not `>` or `==`. That is deliberate and it is the standard library's convention: the fewer operations the body needs, the more types can use it. A type that

defines only `operator<` works with `std::sort`, `std::set` and this `max_of` alike.

Deduction: the compiler reads the arguments

The call `max_of(3, 9)` never mentions `int`. The compiler matches each parameter's pattern (`const T&`) against the argument's type (`int`) and solves for `T`. The rules mirror `auto` from Module 2, lesson 5: a by-value `T` drops references and top-level `const`, and a `const T&` binds to anything and deduces the referenced type. Every parameter that mentions `T` must agree:

```
error: no matching function for call to 'max_of(int, double)'
note: candidate: 'template<class T> T max_of(const T&, const T&)'
note:   deduced conflicting types for parameter 'T' ('int' and 'double')
```

`max_of(3, 2.5)` is the call that produced that. Nothing is converted during deduction — the compiler does not promote the `3` to `double` the way it would for a plain `double max_of(double, double)` — so the two deductions disagree and the call fails. The fixes are to convert one argument yourself or to name `T` explicitly, next.

The trap that compiles is the string literal. `max_of("pear", "apple")` deduces `T = const char*`, and `b < a` then compares two addresses, not two words: it returns whichever literal the linker placed later in memory. Pass `std::string` values, or write `"pear"s` with `using namespace std::literals;`.

Explicit template arguments

Angle brackets after the name set the parameters by hand, and deduction is skipped for the ones you name:

```
max_of<double>(3, 2.5);   // T = double; the 3 converts to 3.0 like any argume

template <typename T>
T read_one() {            // nothing to deduce from: T appears only in the re
    T value;
    std::cin >> value;
    return value;
}
int n = read_one<int>();

template <typename To, typename From>
To convert(const From& from) { return static_cast<To>(from); }
double d = convert<double>(7);   // To named, From deduced as int
```

Explicit arguments fill the parameter list from the left; anything after them is still deduced. That is why `To` comes before `From` in `convert` — the caller names the one that cannot be deduced and lets the argument supply the other — and it is exactly how `std::make_unique<Widget>(a, b)` names the type to build while deducing the constructor arguments.

More than one type parameter

```
template <typename A, typename B>
auto add(const A& a, const B& b) {
    return a + b;               // return type deduced from the return stateme
}
add(3, 2.5);                    // A = int, B = double, returns double
```

Independent parameters deduce independently, so mixed types are fine here where `max_of` refused them. With `auto` as the return type the compiler takes the type of the `return` expression; every `return` in the body must then agree, or the function does not compile. When the return type must be spelled from the parameters, the trailing form `auto add(const A& a, const B& b) → decltype(a + b)` says so (lesson 6 covers `decltype`).

Templates and overloads together

A template can share its name with ordinary functions, and the compiler chooses among all of them:

```cpp
void describe(int) { std::cout << "int overload\n"; }

template <typename T>
void describe(const T&) { std::cout << "template\n"; }

describe(5);      // int overload: a non-template that matches exactly wins
describe(5L);     // template, T = long: an exact template match beats a conve
describe(2.5);    // template, T = double
```

The rule is short: deduce the template, then run ordinary overload resolution over every candidate; when a non-template and a template instantiation are equally good, the non-template wins. That tie-break is what lets you handle one awkward type — `const char*`, say — with a plain overload while the template covers everything else, and it is the recommended tool for the job; lesson 3 explains why specialising a function template is the wrong one.

Abbreviated function templates (C++20)

When you never need to name `T`, the `auto` parameter of a generic lambda (Module 4, lesson 5) is available on an ordinary function too:

```cpp
void show(const auto& value) {          // template <typename T> void show(cor
    std::cout << value << '\n';
}

auto add(auto a, auto b) { return a + b; }   // two parameters: template <type
```

Each `auto` is its own template parameter, so `add(auto, auto)` accepts mixed types. The long form is still required when the body needs `T` — to declare a local `T sum{}`, or to write `std::vector<T>` — and lesson 5 adds a concept in front of the `auto` to say what the type must support.

A template lives in a header

The compiler generates `max_of<int>` where it sees the call, so the whole definition — not just a declaration — must be visible in that translation unit. Put the template's body in the header. A declaration in `max.h` with the definition in `max.cpp` compiles both files and fails at link time with `undefined reference to 'int max_of<int>(int const&, int const&)'`, because

nothing in `max.cpp` asked for the `int` version and so it was never generated. The one-definition rule (Module 1, lesson 2) permits identical template definitions in many translation units; the linker keeps one copy of each instantiation.

Pitfalls

Mistake	What happens
`max_of(3, 2.5)`	Deduction conflict: `int` vs `double`. Name `T` or convert one argument.
`max_of("pear", "apple")`	`T = const char*`; compares addresses, compiles, wrong.
`read_one()` without `<int>`	`T` appears only in the return type: nothing to deduce from.
Definition in a `.cpp`, declaration in the header	Undefined reference at link time.
`auto` return with `return 1;` in one branch and `return 2.5;` in another	Inconsistent deduction: compile error.
Trusting an uncalled template	Only its syntax is checked until something instantiates it.

Key takeaways

A function template is a recipe; each set of template arguments instantiates a separate function, on demand.

Deduction matches parameter patterns to argument types and never converts: mismatched arguments to one `T` fail.

`f<double>(…)` names the arguments explicitly, filling from the left; it is the only way when `T` is not deducible.

Non-template overloads win exact-match ties; use an overload, not a specialisation, for the special case.

`void f(const auto& x)` is an abbreviated template; each `auto` is its own parameter.

Templates go in headers, definition and all.

EXERCISE 1

max_of and min_of over int, double and std::string

Write two function templates, `max_of(const T& a, const T& b)` and `min_of(const T& a, const T& b)`, each using only `operator<` in its body (so any type that defines `<` works). Read an integer `n`, then `n` lines of `<type> <a> <b>` where `<type>` is `int`, `double` or `string`. Both values on a line are read as that type, so one instantiation serves the line, and the three type words instantiate the templates three times in total.

Print `<type>: min=<min> max=<max>` per line. Doubles print with two decimals (`std::fixed` and `std::setprecision(2)`); strings compare the way `std::string` does, so `Zebra` is less than `apple`. Put the reading in a helper such as `answer<T>(type)` — `T` appears only in its body, so the call must name it: `answer<int>(type)`.

Input: `n`, then `n` lines of `type a b`. **Output:** `n` lines.

```
3
int 3 9
double 2.5 -1
string pear apple
```

prints

```
int: min=3 max=9
double: min=-1.00 max=2.50
string: min=apple max=pear
```

INPUT 1

```
3
int 3 9
double 2.5 -1
string pear apple
```

OUTPUT 1

```
int: min=3 max=9
double: min=-1.00 max=2.50
string: min=apple max=pear
```

INPUT 2

```
2
int -4 -9
string Zebra apple
```

OUTPUT 2

```
int: min=-9 max=-4
string: min=Zebra max=apple
```

REFERENCE SOLUTION

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

template <typename T>
T max_of(const T& a, const T& b) {
    return b < a ? a : b;
}

template <typename T>
T min_of(const T& a, const T& b) {
    return b < a ? b : a;
}

template <typename T>
void answer(const std::string& type) {
    T a, b;
    std::cin >> a >> b;
    std::cout << type << ": min=" << min_of(a, b) << " max=" << max_of(a, b) << '\n'
}

int main() {
    std::cout << std::fixed << std::setprecision(2);
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string type;
        std::cin >> type;
        if (type == "int") answer<int>(type);
        else if (type == "double") answer<double>(type);
        else answer<std::string>(type);
    }
    return 0;
}
```

EXERCISE 2

clamp_to with read_one<T>() and a printing overload

Write three things. `read_one<T>()` reads one value of type `T` from standard input and returns it — `T` appears only in the return type, so every call names it explicitly. `clamp_to(const T& v, const T& lo, const T& hi)` returns `lo` when `v < lo`, `hi` when `hi < v`, and `v` otherwise. And two printers that share a name: a plain function `void show(double v)` that prints with two decimals, and an abbreviated function template `void show(const auto& v)` that prints the value as it is. Overload resolution sends a `double` to the plain function (an exact non-template match wins the tie) and an `int` or a `std::string` to the template (an exact template match beats the conversion to `double`).

Read `n`, then `n` lines of `<type> <value> <lo> <hi>` with `<type>` one of `int`, `double`, `string`. For each line print `<type>: <clamped>`. After the last line print `changed=<k>`, the number of lines whose value was outside `[lo, hi]`.

Input: `n`, then `n` lines of `type value lo hi`. **Output:** `n` lines, then `changed=<k>`.

```
3
int 15 0 10
double 2.5 1 5
string m a k
```

prints

```
int: 10
double: 2.50
string: k
changed=2
```

INPUT 1

```
3
int 15 0 10
double 2.5 1 5
string m a k
```

OUTPUT 1

```
int: 10
double: 2.50
string: k
changed=2
```

INPUT 2

```
2
int -3 -2 2
string cat cab dog
```

OUTPUT 2

```
int: -2
string: cat
changed=1
```

REFERENCE SOLUTION

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

template <typename T>
T read_one() {
    T value;
    std::cin >> value;
    return value;
}

template <typename T>
T clamp_to(const T& v, const T& lo, const T& hi) {
    if (v < lo) return lo;
    if (hi < v) return hi;
    return v;
}

void show(double v) { std::cout << std::fixed << std::setprecision(2) << v; }

void show(const auto& v) { std::cout << v; }

template <typename T>
bool clamp_line(const std::string& type) {
    const T v = read_one<T>();
    const T lo = read_one<T>();
    const T hi = read_one<T>();
    std::cout << type << ": ";
    show(clamp_to(v, lo, hi));
    std::cout << '\n';
    return v < lo || hi < v;
}

int main() {
    int n;
    std::cin >> n;
    int changed = 0;
    for (int i = 0; i < n; ++i) {
        std::string type;
        std::cin >> type;
        bool moved = false;
        if (type == "int") moved = clamp_line<int>(type);
        else if (type == "double") moved = clamp_line<double>(type);
        else moved = clamp_line<std::string>(type);
        if (moved) ++changed;
    }
    std::cout << "changed=" << changed << '\n';
```

```
    return 0;
}
```

QUIZ

1. What happens here?

```
template <typename T>
T max_of(const T& a, const T& b) { return b < a ? a : b; }

int main() { std::cout << max_of(3, 2.5); }
```

A. Prints `3`

B. Prints `2.5`

C. Compile error: conflicting deductions for `T`

D. Undefined behaviour

2. `max_of("pear", "apple")` compiles. What does it compare?

A. The two words, character by character

B. The lengths of the two literals

C. The addresses of the two literals

D. Nothing — it is a compile error

3. What does this print?

```
void describe(int) { std::cout << "int"; }

template <typename T>
void describe(const T&) { std::cout << "template"; }

int main() { describe(5L); }
```

A. `int`

B. `template`

C. Compile error: ambiguous call

D. `int` — non-templates always win

4. Why does this fail to compile?

```
template <typename T>
T zero() { return T{}; }

int main() { auto z = zero(); }
```

A. `T{}` is not valid for every `T`

B. `T` appears only in the return type, so there is nothing to deduce it from

C. `auto` cannot hold a template's result

D. A template with no parameters must be `constexpr`

5. `max.h` declares `template <typename T> T max_of(const T&, const T&);` and `max.cpp` defines it. `main.cpp` includes the header and calls `max_of(1, 2)`. What happens?

A. Everything compiles and links

B. `main.cpp` fails to compile: the body is missing

C. Both files compile; the link fails with an undefined reference to `max_of<int>`

D. `max.cpp` fails to compile: a template cannot live in a `.cpp` file

6. What does this print?

```
auto add(auto a, auto b) { return a + b; }

int main() { std::cout << add(3, 2.5); }
```

A. `5.5`

B. `5`

C. Compile error: the two `auto` parameters must have the same type

D. Compile error: `auto` parameters need a `template` header

7. In `template <typename To, typename From> To convert(const From& f);`, why is `To` declared first?

A. Return-type parameters must always come first

B. Explicit arguments fill the list from the left, so the caller can name `To` and let `From` be deduced

C. Deduction runs right to left

D. It makes no difference; the order is a convention

ANSWER KEY · FUNCTION TEMPLATES — ONE DEFINITION, EVERY TYPE

1. C Deduction never converts: `3` says `T = int` and `2.5` says `T = double`, and the two must agree. Either name the type — `max_of<double>(3, 2.5)` — or pass `3.0`. A plain `double max_of(double, double)` would have promoted the `3`; a template does not.

2. C Both literals decay to `const char*`, so `T = const char*` and `b < a` compares two pointers. The result depends on where the linker placed the strings. Pass `std::string` values, or `"pear"s` with `using namespace std::literals;`.

3. B The template deduces `T = long`, an exact match; the non-template needs a `long` to `int` conversion. Exact beats conversion, so the template runs. The non-template wins only when the two are *equally* good — `describe(5)` prints `int`.

4. B Deduction reads the *arguments*; a return type is never used for it. `zero<int>()` names the argument explicitly. `std::make_unique<Widget>(…)` and `std::get<0>(…)` are the same shape: the caller supplies what cannot be deduced.

5. C The compiler generates `max_of<int>` where it sees the call, and it can only do that with the definition in view. `main.cpp` compiles against the declaration, `max.cpp` never instantiates the `int` version because nothing there asks for it, and the linker finds no symbol. Put the whole template in the header.

6. A Each `auto` parameter is its own template parameter, so this is `template <typename A, typename B> auto add(A, B)` and mixed types are fine. `int + double` is `double`, the return type is deduced from it, and `5.5` prints. Only a single `T` used twice would force the arguments to agree.

7. B `convert<double>(7)` sets `To = double` and leaves `From` to be deduced from the argument. Declared the other way round, the caller would have to spell both. Putting the non-deducible parameters first is the pattern behind `std::make_unique<T>(args…)`.

LESSON 2 · 14 MIN

Class templates — Stack<T>, Pair<A, B> and deduction

`std::vector<int>` is not a class; `std::vector` is a class template and `std::vector<int>` is one class stamped out of it. Every container, smart pointer and vocabulary type in the standard library is built the same way, and writing your own is how you stop duplicating a `Stack` for `int` and another for `std::string`. This lesson covers declaring a class template, defining its members outside the class body, default template arguments, class template argument deduction (C++17), and the `typename` keyword the compiler demands when a name depends on `T`.

Stack<T>

```
#include <cstddef>
#include <stdexcept>
#include <vector>

template <typename T>
class Stack {
public:
    void push(const T& value) { items_.push_back(value); }
    void pop();                          // defined below
    const T& top() const;
    std::size_t size() const { return items_.size(); }
    bool empty() const { return items_.empty(); }

private:
    std::vector<T> items_;
};
```

`template <typename T>` in front of a class makes `T` usable anywhere inside it: as a member's type, a parameter, a return type, or an argument to another template such as `std::vector<T>`. `Stack<int>` and `Stack<std::string>` are two unrelated classes — there is no conversion between them, and a function taking `Stack<int>&` will not accept a `Stack<long>`. Inside the class body the bare name `Stack` means "the current instantiation", so a copy constructor is written `Stack(const Stack&)`, not `Stack(const Stack<T>&)`, though both are accepted.

Member functions are instantiated one by one, only when called. `Stack<Widget>` compiles even if `Widget` has no `operator<`, as long as nobody calls a member that needs it. This lazy rule is why `std::vector<T>` can offer `operator<` for every `T` and only complain when you actually compare two vectors of a type that cannot be compared.

Members defined outside the class

The body reads better as an interface when the long members live below it:

```
template <typename T>
void Stack<T>::pop() {
    if (items_.empty()) throw std::out_of_range("pop on empty stack");
    items_.pop_back();
}

template <typename T>
const T& Stack<T>::top() const {
    if (items_.empty()) throw std::out_of_range("top on empty stack");
    return items_.back();
}
```

Every out-of-class definition repeats the template header and qualifies the name with `Stack<T>::` — here the `<T>` is required, because outside the class there is no current instantiation to be implied. The definitions still belong in the header, next to the class: the compiler needs them wherever `Stack<int>::pop` is first used (lesson 1).

Pair<A, B> and default template arguments

```
template <typename A, typename B = A>
struct Pair {
    A first;
    B second;

    Pair<B, A> swapped() const { return {second, first}; }
};

Pair<int, std::string> label{1, "one"};
Pair<double> range{0.5, 2.5};              // B defaults to A: Pair<double, dou
Pair<std::string, int> back = label.swapped();
```

Two parameters, two independent types. A default template argument works like a default function argument — trailing ones only, and the caller may omit them — and it is why `std::map<std::string, int>` compiles with two arguments when the template declares four: `std::map<Key, T, Compare = std::less<Key>, Allocator = …>`. `swapped()` returns a *different instantiation* of the same template; a member of `Pair<A, B>` can name `Pair<B, A>` freely because the compiler generates it on demand.

A struct template with public members and no constructor is an aggregate, so brace initialisation fills the members in order exactly as for a plain struct (Module 8).

Class template argument deduction (C++17)

Before C++17 every instantiation was spelled out; since then the compiler can deduce the arguments from a constructor call or an aggregate initialiser:

```
Pair p{42, 2.5};                    // Pair<int, double>
std::vector v{1, 2, 3};             // std::vector<int>
std::pair q{std::string("k"), 7};   // std::pair<std::string, int>
std::lock_guard lock(mutex);        // std::lock_guard<std::mutex>
```

CTAD is all or nothing: `Pair<int> p{1, 2.5}` does not deduce `B` from the second member — it uses the default `B = A` and narrows `2.5` to `int`, which brace-init rejects. Either name every argument or none.

Two deductions surprise people. `Pair p{"k", 1}` deduces `A = const char*`, because a string literal is an array of `char` and decays to a pointer; a library can steer this with a *deduction guide* — the line `Pair(const char*, B) → Pair<std::string, B>;` after the struct tells the compiler what to deduce for that shape, and it is how `std::pair`-like types become friendlier. And `std::vector v{3}` is a vector holding the single element `3`, because braces prefer the `initializer_list` constructor; `std::vector v(3, 0)` with parentheses is three zeros. Whenever the deduced type is not obvious from the initialiser, write it out — CTAD saves typing, not thinking.

---

typename for dependent names

```cpp
template <typename C>
void print_first(const C& c) {
    typename C::value_type first = *c.begin();   // typename is required here
    std::cout << first << '\n';
}
```

`C::value_type` is a *dependent name*: what it means depends on `C`, which is unknown while the template is parsed. It might be a type (as in every container) or a static data member or a constant, and the grammar differs, so the parser assumes "not a type" unless you say `typename`. Leave it out and GCC reports `need 'typename' before 'C::value_type' because 'C' is a dependent scope`. The same logic gives the rarer `c.template get<0>()` when a dependent member is itself a template. In practice `auto first = *c.begin();` sidesteps the whole question, and most modern generic code prefers it; `typename` still appears in return types and member declarations where `auto` cannot.

Static members and member templates

A static data member belongs to each instantiation separately: `Stack<int>::count` and `Stack<double>::count` are two variables, each initialised once. A member function can itself be a template — `template <typename U> void push_converted(const U& u) { push(static_cast<T>(u)); }` — with its own parameter deduced at the call; `std::vector::assign` and the converting constructors of `std::pair` are member templates.

The standard library is class templates

Template	Parameters	Deduction you have used
`std::vector<T, Alloc>`	element, allocator (defaulted)	`std::vector v{1, 2}`
`std::map<K, V, Compare, Alloc>`	two named, two defaulted	—
`std::pair<A, B>`	two types	`std::pair{1, 2.5}`
`std::array<T, N>`	a type and a number (lesson 4)	`std::array a{1, 2, 3}`
`std::unique_ptr<T, Deleter>`	pointee, deleter (defaulted)	—
`std::basic_string<CharT>`	`std::string` is `basic_string<char>`	—

An *alias template* names a shape once: `template <typename T> using Grid = std::vector<std::vector<T>>;` lets you write `Grid<int>` everywhere (Module 6 used the long form).

Pitfalls

Mistake	What happens
Passing `Stack<int>` where `Stack<long>` is expected	Unrelated types: no conversion, compile error.
Out-of-class member without `template <typename T>`	`Stack` is not a class: compile error.
`Pair<int> p{1, 2.5}`	Partial CTAD does not exist; `B` defaults to `int` and the narrowing fails.
`std::vector v{3}`	One element, not three. Use parentheses for a count.
`Pair p{"k", 1}`	`A = const char*`, not `std::string`.
`C::value_type x;` without `typename`	Parsed as a non-type: compile error.
Member definitions in a `.cpp`	Undefined reference for every instantiation the `.cpp` never used.

Key takeaways

- A class template stamps out a distinct class per argument list; `Stack<int>` and `Stack<long>` are unrelated.
- Members are instantiated on first use, so a template can offer operations only some `T`s support.
- Out-of-class members repeat the template header and qualify with `Stack<T>::`; they live in the header.
- Default template arguments are trailing and explain why `std::map<K, V>` takes two arguments.
- CTAD deduces all arguments or none; watch `const char*` and `std::vector v{3}`.
- A name that depends on `T` needs `typename` to be read as a type.

EXERCISE 1

Stack<T> for the type named on the first line

Write a class template `Stack<T>` over a `std::vector<T>` with `push`, `pop`, `top`, `size` and `print`. Define `pop`, `top` and `print` **outside** the class body — each with its own `template <typename T>` header and the `Stack<T>::` qualifier. Then write a function template `run<T>()` that owns one `Stack<T>` and processes commands until the input ends; `main` reads the first line and calls `run<int>()` or `run<std::string>()`.

Commands, one per line: `push <v>`; `pop` (print `popped <v>` or `empty`); `top` (print `top=<v>` or `empty`); `size` (print `size=<k>`); `print` (print `stack: <bottom> … <top>` or `stack: (empty)`).

Input: the type word (`int` or `string`), then commands. **Output:** one line per `pop`, `top`, `size` and `print`.

```
int
push 1
push 2
push 3
print
pop
top
size
```

prints

```
stack: 1 2 3
popped 3
top=2
size=2
```

INPUT 1

```
int
push 1
push 2
push 3
print
pop
top
size
```

OUTPUT 1

```
stack: 1 2 3
popped 3
top=2
size=2
```

INPUT 2

```
string
push pear
push apple
top
pop
pop
pop
print
```

OUTPUT 2

```
top=apple
popped apple
popped pear
empty
stack: (empty)
```

REFERENCE SOLUTION

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

template <typename T>
class Stack {
public:
    void push(const T& value) { items_.push_back(value); }
    bool pop(T& out);
    bool top(T& out) const;
    std::size_t size() const { return items_.size(); }
    void print() const;

private:
    std::vector<T> items_;
};

template <typename T>
bool Stack<T>::pop(T& out) {
    if (items_.empty()) return false;
    out = items_.back();
    items_.pop_back();
    return true;
}

template <typename T>
bool Stack<T>::top(T& out) const {
    if (items_.empty()) return false;
    out = items_.back();
    return true;
}

template <typename T>
void Stack<T>::print() const {
    std::cout << "stack:";
    if (items_.empty()) {
        std::cout << " (empty)\n";
        return;
    }
    for (const T& item : items_) std::cout << ' ' << item;
    std::cout << '\n';
}

template <typename T>
void run() {
    Stack<T> stack;
    std::string cmd;
    while (std::cin >> cmd) {
```

```
        if (cmd == "push") {
            T value;
            std::cin >> value;
            stack.push(value);
        } else if (cmd == "pop") {
            T value;
            if (stack.pop(value)) std::cout << "popped " << value << '\n';
            else std::cout << "empty\n";
        } else if (cmd == "top") {
            T value;
            if (stack.top(value)) std::cout << "top=" << value << '\n';
            else std::cout << "empty\n";
        } else if (cmd == "size") {
            std::cout << "size=" << stack.size() << '\n';
        } else if (cmd == "print") {
            stack.print();
        }
    }
}

int main() {
    std::string type;
    std::cin >> type;
    if (type == "int") run<int>();
    else run<std::string>();
    return 0;
}
```

EXERCISE 2

Pair<A, B> with a default argument and CTAD

Write `template <typename A, typename B = A> struct Pair { A first; B second; Pair<B, A> swapped() const; }` and a free function template `print(const Pair<A, B>& p)` that prints `(<first>, <second>)` and deduces `A` and `B` from the argument.

Read `n` (at least 1), then `n` lines of `<name> <score>` (a word and an integer). Build each entry with class template argument deduction — `Pair entry{name, score};` deduces `Pair<std::string, int>` — and keep them in a `std::vector`. Then print: every entry, one per line; `swapped:` followed by the first entry's `swapped()` (a `Pair<int, std::string>`); `range:` followed by a `Pair<int>` holding the lowest and highest score — note the single argument, so `B` takes its default; and `best:` followed by the entry with the highest score (the first one on a tie).

Input: `n`, then `n` lines of `name score`. **Output:** `n` lines, then the `swapped`, `range` and `best` lines.

```
3
ada 93
bob 71
cy 88
```

prints

```
(ada, 93)
(bob, 71)
(cy, 88)
swapped: (93, ada)
range: (71, 93)
best: (ada, 93)
```

INPUT 1

```
3
ada 93
bob 71
cy 88
```

OUTPUT 1

```
(ada, 93)
(bob, 71)
(cy, 88)
swapped: (93, ada)
range: (71, 93)
best: (ada, 93)
```

INPUT 2

```
1
zed 50
```

OUTPUT 2

```
(zed, 50)
swapped: (50, zed)
range: (50, 50)
best: (zed, 50)
```

REFERENCE SOLUTION

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

template <typename A, typename B = A>
struct Pair {
    A first;
    B second;

    Pair<B, A> swapped() const { return {second, first}; }
};

template <typename A, typename B>
void print(const Pair<A, B>& p) {
    std::cout << '(' << p.first << ", " << p.second << ')';
}

int main() {
    int n;
    std::cin >> n;
    std::vector<Pair<std::string, int>> entries;
    for (int i = 0; i < n; ++i) {
        std::string name;
        int score;
        std::cin >> name >> score;
        Pair entry{name, score};          // CTAD: Pair<std::string, int>
        entries.push_back(entry);
    }
    for (const auto& e : entries) {
        print(e);
        std::cout << '\n';
    }
    std::cout << "swapped: ";
    print(entries.front().swapped());
    std::cout << '\n';

    Pair<int> range{entries.front().second, entries.front().second};   // B defaults
    std::size_t bestAt = 0;
    for (std::size_t i = 0; i < entries.size(); ++i) {
        const int s = entries[i].second;
        if (s < range.first) range.first = s;
        if (range.second < s) range.second = s;
        if (entries[bestAt].second < s) bestAt = i;
    }
    std::cout << "range: ";
    print(range);
    std::cout << "\nbest: ";
    print(entries[bestAt]);
```

```
    std::cout << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```
std::vector v{3};
std::cout << v.size();
```

A. `3`

B. `1`

C. `0`

D. Compile error: CTAD needs the element type

2. Which is the correct out-of-class definition of `pop` for `template <typename T> class Stack`?

A. `void Stack<T>::pop() { … }`

B. `template <typename T> void Stack::pop() { … }`

C. `template <typename T> void Stack<T>::pop() { … }`

D. `template <> void Stack<T>::pop() { … }`

3. Given `template <typename A, typename B = A> struct Pair { A first; B second; };`, what happens with `Pair<int> p{1, 2.5};`?

A. `Pair<int, double>` — `B` is deduced from `2.5`

B. `Pair<int, int>`, and the initialisation fails because `2.5` narrows

C. `Pair<int, int>` holding `1` and `2`

D. Compile error: a defaulted parameter cannot be omitted

4. Why does `typename C::value_type first = *c.begin();` need `typename` inside a template?

A. `value_type` is a reserved word

B. `C::value_type` depends on `C`, and the parser assumes a dependent name is not a type unless told

C. Every nested type needs `typename`, template or not

D. It is optional; GCC accepts the line without it

5. `struct Widget {};` has no `operator<`. Does `Stack<Widget> s; s.push(Widget{});` compile, given that `Stack<T>` has a member `bool contains_less(const T& x) const` that uses `<`?

A. No — every member is checked when `Stack<Widget>` is instantiated

B. Yes — members are instantiated only when called, and `contains_less` never is

C. No — `push` needs `<` to keep the stack ordered

D. Yes, but with a warning about the unusable member

6. `std::map` is declared with four template parameters. Why does `std::map<std::string, int>` compile with two?

A. The compiler deduces the other two from the first two

B. The last two have default arguments (`std::less<Key>` and an allocator)

C. `std::map` is a partial specialisation for the two-argument form

D. Two is a special case the standard allows only for `std::map`

7. What does `Pair p{"k", 1};` deduce for `A` in `template <typename A, typename B> struct Pair { A first; B second; };`?

A. `std::string`

B. `const char*`

C. `char[2]`

D. It is ambiguous and fails to compile

ANSWER KEY · CLASS TEMPLATES — STACK<T>, PAIR<A, B> AND DEDUCTION

1. B Braces prefer the `initializer_list` constructor, so this is a `std::vector<int>` holding the single element `3`. `std::vector v(3, 0)` with parentheses is three zeros. When the deduced type is not obvious from the initialiser, write it out.

2. C Outside the class there is no current instantiation to imply the arguments, so the definition repeats the template header and qualifies with `Stack<T>::`. Without the header `T` is undeclared; without `<T>` `Stack` is a template, not a class; `template <>` introduces an explicit specialisation, which needs a concrete type.

3. B Class template argument deduction is all or nothing: naming one argument switches it off, so `B` takes its default `int`. Brace initialisation then rejects `2.5` as narrowing. Either write `Pair<int, double>` or `Pair p{1, 2.5}` with no arguments at all.

4. B While the template is parsed, `C` is unknown: `C::value_type` might be a type, a static member or a constant, and the grammar differs. The parser assumes "not a type" and stops with `need 'typename' before …`. Outside a template, or for a non-dependent name, `typename` is not required.

5. B A class template's member functions are instantiated one by one on first use. `push` needs only `push_back`, so the program compiles; calling `contains_less` would be the moment the missing `<` is reported. This is why `std::vector<T>` can offer `operator<` for every `T`.

6. B Default template arguments work like default function arguments: trailing parameters may be omitted and take their defaults. There is no deduction involved — `std::less<Key>` and `std::allocator<…>` are spelled in the declaration.

7. B A string literal is an array of `char` that decays to `const char*` during deduction, so `A = const char*` — not the `std::string` most people expect. A deduction guide such as `Pair(const char*, B) → Pair<std::string, B>;` steers it, or pass a `std::string` explicitly.

LESSON 3 · 13 MIN

Specialisation — when one type needs its own version

A template is a rule that applies to every type; a specialisation is the exception written next to it. `std::vector<bool>` packs bits where every other `std::vector<T>` stores elements, `std::hash<std::string>` hashes characters where the primary template has no body at all, and every `std::is_integral<T>` is a specialisation that says `true` for a handful of types. This lesson covers full specialisation (one exact type, such as `const char*` or `bool`), partial specialisation (a pattern, such as every pointer or every `std::pair<T, T>`), the traits style that the next lessons build on, and the reason a function template is overloaded rather than specialised.

Full specialisation

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

template <typename T>
struct Describe {                                       // the primary template
    static void print(const T& v) { std::cout << "[value] " << v << '\n'; }
};

template <>                                             // no parameters left to d
struct Describe<bool> {
    static void print(bool v) { std::cout << "[bool] " << (v ? "true" : "false
};

template <>
struct Describe<const char*> {
    static void print(const char* v) { std::cout << "[C string] \"" << v << "\
};

template <>
struct Describe<std::string> {
    static void print(const std::string& v) { std::cout << "[string] \"" << v
};
```

`template <>` with an empty parameter list followed by the name with its arguments filled in — `Describe<bool>` — declares an *explicit* (full) specialisation. When the compiler needs `Describe<bool>` it uses that class and never instantiates the primary for `bool`. The specialisation is a separate class: it inherits nothing from the primary, so if the primary has five members and the specialisation defines two, the other three do not exist for `bool`. Keep the interface the same on purpose; the compiler does not make you.

A specialisation must be declared before the first use that would instantiate the primary. Use `Describe<bool>` on line 10 and specialise it on line 30 and GCC stops with `specialization of 'Describe<bool>' after instantiation`. Headers put the specialisations right after the primary for this reason.

You can also specialise a single member instead of the whole class — `template <> void Describe<int>::print(const int& v) { … }` — which keeps every other member from the primary. It is the neater choice when only one function differs.

Partial specialisation

A full specialisation names one type; a partial specialisation names a *pattern* and stays a template:

```cpp
#include <utility>
#include <vector>

template <typename T>
struct Describe<T*> {                                  // every pointer type
    static void print(T* p) {
        std::cout << "[pointer to] ";
        if (p) Describe<T>::print(*p);
        else std::cout << "null\n";
    }
};

template <typename T>
struct Describe<std::vector<T>> {                      // every vector, of anythi
    static void print(const std::vector<T>& v) {
        std::cout << "[vector of " << v.size() << "]\n";
        for (const T& item : v) Describe<T>::print(item);
    }
};

template <typename A, typename B>
struct Describe<std::pair<A, B>> {                     // any pair
    static void print(const std::pair<A, B>& p) { std::cout << "[pair]\n"; }
};

template <typename T>
struct Describe<std::pair<T, T>> {                     // both halves the same ty
    static void print(const std::pair<T, T>& p) { std::cout << "[homogeneous p
};
```

The parameter list after `template` is whatever is still free — `T` for pointers, `A` and `B` for a general pair — and the argument list after the name is the pattern to match. `Describe<int*>` matches `Describe<T*>` with `T = int`; `Describe<std::vector<std::string>>` matches the vector pattern and recurses into `Describe<std::string>`, which is the full specialisation above. Because the recursion goes through the template again, `Describe<std::vector<std::vector<int>>>` works with no further code.

When several partial specialisations match, the compiler picks the *most specialised* — the one whose pattern could be matched by the other but not the reverse. `std::pair<int, int>` matches both pair patterns; `pair<T, T>` is more specialised than `pair<A, B>` (every `pair<T, T>` is a `pair<A, B>`, not vice versa), so it wins. Two patterns where neither is more specialised make the use ambiguous and the compiler says so. The library uses this

machinery in `std::unique_ptr<T[]>`, a partial specialisation for arrays that adds `operator[]` and calls `delete[]`.

Traits: specialisation as a lookup table

```
template <typename T> struct TypeName { static constexpr const char* value = "
template <> struct TypeName<int>         { static constexpr const char* value
template <> struct TypeName<double>      { static constexpr const char* value
template <> struct TypeName<std::string> { static constexpr const char* value

template <typename T>
constexpr const char* type_name_v = TypeName<T>::value;     // a variable temp

std::cout << type_name_v<double> << '\n';                    // double
```

A *trait* is a class template whose members are compile-time facts about `T`, filled in by specialisation. The whole of `<type_traits>` (lesson 6) is this pattern: `std::is_integral<T>` derives from `std::false_type`, and specialisations for `int`, `long`, `char`, `bool` and their relatives derive from `std::true_type`; `std::is_pointer<T*>` is a partial specialisation. The `_v` variable template is the C++17 convenience so you write `std::is_integral_v<T>` instead of `std::is_integral<T>::value`.

Why a trait instead of `typeid(T).name()`? Because `name()` returns the *mangled* name — GCC and Clang print `i` for `int` and `PKc` for `const char*`, MSVC prints something else again — and it is never a string you want in output. A trait gives you the words you chose.

Function templates: overload, do not specialise

A function template can be fully specialised, but never partially:

```
template <typename T> void show(const T& v) { std::cout << v << '\n'; }

template <typename T> void show<std::vector<T>>(const std::vector<T>& v);   //
                                                                            //
template <typename T> void show(const std::vector<T>& v);                   //

template <> void show<const char*>(const char* const& v);     // legal full sp
void show(const char* v);                                     // the overload
```

The trap is that a full specialisation does not take part in overload resolution. The compiler first picks the best *primary* template or non-template for the call; only if the chosen primary has a matching specialisation is that used. The classic demonstration: `template <class T> void f(T);` specialised as `template <> void f(int*);` and then a second template `template <class T> void f(T*);` added later. `f(p)` with an `int*` chooses the second primary, because `T*` is the better match, and the specialisation of the first is never considered. An ordinary overload `void f(int*)` would have won outright. Overloads are the tool for functions; specialisation is the tool for class templates.

Where each tool fits

Need	Tool
A different representation for one type (`std::vector<bool>`)	Full specialisation of the class
Extra members for a family of types (`unique_ptr<T[]>`)	Partial specialisation
A compile-time fact per type (`is_integral`, `TypeName`)	Trait: primary default plus specialisations
One function that treats one type differently	A non-template overload
A few lines that differ inside one body	`if constexpr` (lesson 6)
"Only types that can do X"	A concept (lesson 5)

Pitfalls

Mistake	What happens
Specialising after the first use	`specialization … after instantiation` error.
Expecting the full specialisation to keep the primary's members	It has only what it declares.
`template <typename T> void f<T*>(T*)`	Partial specialisation of a function: syntax error. Write an overload.
Relying on `template <> void f<int*>(int*)` to win	It is not an overload candidate; a better-matching primary hides it.
Two partial specialisations, neither more specialised	Ambiguous instantiation error at the use.
Printing `typeid(T).name()`	Mangled, compiler-specific text.
`template <> struct TypeName { … }` (arguments missing)	Not a specialisation; redefinition error.

Key takeaways

`template <> struct X<Type>` replaces the primary for exactly that type; it inherits none of its members.

`template <typename T> struct X<T*>` matches a pattern and stays a template; the most specialised match wins.

Traits are class templates whose specialisations record a fact per type; `<type_traits>` is built this way.

Declare specialisations before the first use, in the header, next to the primary.

Function templates do not partially specialise, and a full specialisation is not an overload candidate: overload instead.

Use your own tag strings, never `typeid(T).name()`, for anything printed.

EXERCISE 1

Describe<T> with full specialisations

Write a class template `Describe<T>` with one static member `print`. The primary template prints `[value] <v>`. Add three **full specialisations**: `Describe<bool>` prints `[bool] true` or `[bool] false`; `Describe<const char*>` prints `[C string] "<v>"`; `Describe<std::string>` prints `[string] "<v>"`. The tags are yours — never print `typeid(T).name()`.

Read `n`, then `n` lines of `<kind> <token>` with `<kind>` one of `int`, `double`, `bool` (token `0`, `1`, `true` or `false`), `string` and `cstr`. For `cstr` call `Describe<const char*>::print` with the token's `c_str()`, so the pointer specialisation is what runs. Doubles print with the stream's default formatting (`2.5` prints as `2.5`).

Input: `n`, then `n` lines of `kind token`. **Output:** one line per input line.

```
5
int 42
double 2.5
bool 1
string hello
cstr hello
```

prints

```
[value] 42
[value] 2.5
[bool] true
[string] "hello"
[C string] "hello"
```

INPUT 1

```
5
int 42
double 2.5
bool 1
string hello
cstr hello
```

OUTPUT 1

```
[value] 42
[value] 2.5
[bool] true
[string] "hello"
[C string] "hello"
```

INPUT 2

```
2
bool 0
int -7
```

OUTPUT 2

```
[bool] false
[value] -7
```

REFERENCE SOLUTION

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

template <typename T>
struct Describe {
    static void print(const T& v) { std::cout << "[value] " << v << '\n'; }
};

template <>
struct Describe<bool> {
    static void print(bool v) { std::cout << "[bool] " << (v ? "true" : "false") <<
};

template <>
struct Describe<const char*> {
    static void print(const char* v) { std::cout << "[C string] \"" << v << "\"\n";
};

template <>
struct Describe<std::string> {
    static void print(const std::string& v) { std::cout << "[string] \"" << v << "\'
};

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string kind, token;
        std::cin >> kind >> token;
        if (kind == "int") Describe<int>::print(std::stoi(token));
        else if (kind == "double") Describe<double>::print(std::stod(token));
        else if (kind == "bool") Describe<bool>::print(token == "1" || token == "tru
        else if (kind == "cstr") Describe<const char*>::print(token.c_str());
        else Describe<std::string>::print(token);
    }
    return 0;
}
```

EXERCISE 2

Partial specialisations for vectors and pairs

Write a class template `Describe<T>` whose static `text(const T&)` **returns** a `std::string`, so specialisations can compose. The primary returns `[value] <v>`; the full specialisation `Describe<std::string>` returns `[string] "<v>"`. Then add three **partial specialisations**:

`Describe<std::vector<T>>` returns `[vector of <k>: <e1>, <e2>, …]` with each element described by `Describe<T>::text` — or `[vector of 0: empty]`.

`Describe<std::pair<A, B>>` returns `[pair: <a> | <b>]`.

`Describe<std::pair<T, T>>` returns `[same pair: <a> | <b>]` — the more specialised pattern, chosen whenever both halves have the same type.

Read `n`, then `n` lines: `ints <k> <v…>` (a `std::vector<int>`), `words <k> <w…>` (a `std::vector<std::string>`), `pair <int> <word>` (a `std::pair<int, std::string>`), `same <int> <int>` (a `std::pair<int, int>`), `pairs <k> <a1> <b1> …` (a `std::vector<std::pair<int, int>>`). Print the description of each.

Input: `n`, then `n` lines. **Output:** one line per input line.

```
5
ints 3 1 2 3
words 2 pear apple
pair 7 seven
same 4 5
pairs 2 1 2 3 4
```

prints

```
[vector of 3: [value] 1, [value] 2, [value] 3]
[vector of 2: [string] "pear", [string] "apple"]
[pair: [value] 7 | [string] "seven"]
[same pair: [value] 4 | [value] 5]
[vector of 2: [same pair: [value] 1 | [value] 2], [same pair: [value] 3 | [value] 4]
```

INPUT 1

```
5
ints 3 1 2 3
words 2 pear apple
pair 7 seven
same 4 5
pairs 2 1 2 3 4
```

OUTPUT 1

```
[vector of 3: [value] 1, [value] 2,
[value] 3]
[vector of 2: [string] "pear", [string]
"apple"]
[pair: [value] 7 | [string] "seven"]
[same pair: [value] 4 | [value] 5]
[vector of 2: [same pair: [value] 1 |
[value] 2], [same pair: [value] 3 |
[value] 4]]
```

INPUT 2

```
2
ints 0
words 1 solo
```

OUTPUT 2

```
[vector of 0: empty]
[vector of 1: [string] "solo"]
```

REFERENCE SOLUTION

```cpp
#include <cstddef>
#include <iostream>
#include <sstream>
#include <string>
#include <utility>
#include <vector>

template <typename T>
struct Describe {
    static std::string text(const T& v) {
        std::ostringstream out;
        out << "[value] " << v;
        return out.str();
    }
};

template <>
struct Describe<std::string> {
    static std::string text(const std::string& v) { return "[string] \"" + v + "\"";
};

template <typename T>
struct Describe<std::vector<T>> {
    static std::string text(const std::vector<T>& v) {
        std::string out = "[vector of " + std::to_string(v.size()) + ": ";
        if (v.empty()) out += "empty";
        for (std::size_t i = 0; i < v.size(); ++i) {
            if (i > 0) out += ", ";
            out += Describe<T>::text(v[i]);
        }
        return out + "]";
    }
};

template <typename A, typename B>
struct Describe<std::pair<A, B>> {
    static std::string text(const std::pair<A, B>& p) {
        return "[pair: " + Describe<A>::text(p.first) + " | " + Describe<B>::text(p.
    }
};

template <typename T>
struct Describe<std::pair<T, T>> {
    static std::string text(const std::pair<T, T>& p) {
        return "[same pair: " + Describe<T>::text(p.first) + " | " + Describe<T>::te
    }
};

template <typename T>
```

```cpp
std::vector<T> read_vector() {
    std::size_t k;
    std::cin >> k;
    std::vector<T> v(k);
    for (T& item : v) std::cin >> item;
    return v;
}

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string kind;
        std::cin >> kind;
        if (kind == "ints") {
            std::cout << Describe<std::vector<int>>::text(read_vector<int>()) << '\n
        } else if (kind == "words") {
            std::cout << Describe<std::vector<std::string>>::text(read_vector<std::s
        } else if (kind == "pair") {
            std::pair<int, std::string> p;
            std::cin >> p.first >> p.second;
            std::cout << Describe<std::pair<int, std::string>>::text(p) << '\n';
        } else if (kind == "same") {
            std::pair<int, int> p;
            std::cin >> p.first >> p.second;
            std::cout << Describe<std::pair<int, int>>::text(p) << '\n';
        } else if (kind == "pairs") {
            std::size_t k;
            std::cin >> k;
            std::vector<std::pair<int, int>> v(k);
            for (auto& [a, b] : v) std::cin >> a >> b;
            std::cout << Describe<std::vector<std::pair<int, int>>>::text(v) << '\n'
        }
    }
    return 0;
}
```

QUIZ

1. What does this print?

```
template <typename T> struct Tag { static constexpr const char* value = "generi
template <> struct Tag<int> { static constexpr const char* value = "int"; };
template <typename T> struct Tag<T*> { static constexpr const char* value = "po

int main() {
    std::cout << Tag<int*>::value << ' ' << Tag<const char*>::value << ' ' << T
}
```

A. `int pointer generic`

B. `pointer pointer generic`

C. `pointer generic generic`

D. `int generic generic`

2. The primary `template <typename T> struct Box` has members `get`, `set` and `reset`. The full specialisation `template <> struct Box<bool>` defines only `get`. Which members does `Box<bool>` have?

A. `get`, `set` and `reset` — the other two are inherited from the primary

B. Only `get`

C. `get` from the specialisation and `set` / `reset` from the primary, but only when called

D. It is a compile error to specialise with fewer members

3. Both `template <typename A, typename B> struct D<std::pair<A, B>>` and `template <typename T> struct D<std::pair<T, T>>` exist. What does `D<std::pair<int, int>>` use?

A. The `<A, B>` version — it was declared first

B. The `<T, T>` version — it is more specialised

C. Neither — the use is ambiguous and fails

D. The primary template, because two partial specialisations conflict

4. How do you give a function template `template <typename T> void show(const T&)` a different body for every `std::vector<T>`?

A. `template <typename T> void show<std::vector<T>>(const std::vector<T>&)`

B. Write a second template `template <typename T> void show(const std::vector<T>&)` — an overload

C. `template <> void show<std::vector>(…)`

D. It cannot be done; use a class template instead

5. Given, in this order: `template <class T> void f(T);`, `template <> void f<>(int*);`, `template <class T> void f(T*);` — what does `int* p; f(p);` call?

A. The specialisation `f<>(int*)`

B. The second primary, `f(T*)`, with `T = int`

C. The first primary, `f(T)`, with `T = int*`

D. It is ambiguous

6. Why does the lesson tell you never to print `typeid(T).name()`?

A. It throws for fundamental types

B. It returns the mangled, compiler-specific name (`i` for `int`, `PKc` for `const char*`)

C. It requires RTTI, which templates disable

D. It is only valid for polymorphic classes

7. A header uses `Describe<bool>::print(true)` on line 10 and declares `template <> struct Describe<bool>` on line 30. What happens?

A. It works; specialisations are found by name lookup at the end of the translation unit

B. Compile error: specialisation of `Describe<bool>` after instantiation

C. Line 10 uses the primary and line 30 silently defines an unused class

D. Undefined behaviour

ANSWER KEY · SPECIALISATION — WHEN ONE TYPE NEEDS ITS OWN VERSION

1. B `Tag<int*>` matches the pointer pattern with `T = int` — the full specialisation is for `int`, not `int*`. `const char*` is also a pointer (`T = const char`), and `long` matches nothing special, so the primary answers. Patterns match the *shape* of the argument.

2. B An explicit specialisation is a separate class that replaces the primary for that argument; it inherits nothing. `Box<bool>` has exactly what it declares. Keeping the interface identical is your discipline, not the compiler's — or specialise a single member instead of the whole class.

3. B When several partial specialisations match, the most specialised wins: every `pair<T, T>` is a `pair<A, B>` but not the reverse, so `<T, T>` is chosen. Ambiguity arises only when neither pattern subsumes the other.

4. B Function templates do not partially specialise — the first spelling is a syntax error. A second template with a more specific parameter is an ordinary overload and takes part in overload resolution, which is exactly what you want.

5. B A full specialisation is not an overload candidate. Resolution chooses among the primaries first — `f(T*)` is the better match for a pointer — and only then looks for a specialisation *of that primary*; the specialisation belongs to the other one and is never considered. A plain overload `void f(int*)` would have won outright.

6. B `name()` is whatever the implementation chooses — GCC and Clang return the mangled form, MSVC something else — so it is never a string you want in output. A trait such as `TypeName<T>::value` gives you the words you chose.

7. B Line 10 instantiates the primary for `bool`; a specialisation that arrives afterwards would contradict code already generated, so the compiler refuses it. Declare specialisations right after the primary, before any use.

LESSON 4 · 14 MIN

Non-type and variadic templates — numbers and packs as parameters

A template parameter does not have to be a type. `std::array<int, 4>` takes a number, `std::bitset<64>` takes a number, and `std::tuple<int, std::string, double>` takes as many types as you give it. Both ideas — a *non-type* parameter that is a compile-time value, and a *parameter pack* that stands for any number of arguments — are what let a container know its size without storing it and a `print` function accept three arguments of three types. This lesson covers non-type parameters, packs and `sizeof...`, recursion over a pack, and the C++17 fold expressions that replace most of that recursion with one line.

Non-type template parameters

```
#include <array>
#include <cstddef>

template <typename T, std::size_t N>
class Ring {                                  // a fixed-capacity buffer that ov
public:
    static constexpr std::size_t capacity = N;

    void push(const T& value) {
        data_[(start_ + size_) % N] = value;
        if (size_ < N) ++size_;
        else start_ = (start_ + 1) % N;       // full: the slot we just wrote wa
    }
    std::size_t size() const { return size_; }
    const T& at(std::size_t i) const { return data_[(start_ + i) % N]; }   //

private:
    std::array<T, N> data_{};
    std::size_t start_ = 0;
    std::size_t size_ = 0;
};

Ring<int, 4> recent;                          // N = 4, baked into the type
```

`std::size_t N` declares a parameter that is a value, and every use of `N` inside the class is a compile-time constant: it sizes the `std::array`, it is the modulus, and it can initialise a `static constexpr` member. There is no `size` field for the capacity because the capacity is part of the type — `sizeof(std::array<int, 4>)` is 16 bytes, exactly four `int`s, which is the reason `std::array` exists (Module 6). `Ring<int, 4>` and `Ring<int, 8>` are different types, as unrelated as `Stack<int>` and `Stack<long>` were in lesson 2.

The argument must be a constant expression. `int n; std::cin >> n; Ring<int, n> r;` is refused with `the value of 'n' is not usable in a constant expression`; a `const int n = 4;` is fine because its initialiser is a constant, while `const int n = read();` is not. When the size genuinely comes from input, either dispatch at run time to a few fixed instantiations — `if (n == 4) run<4>(); else run<8>();` — or use a `std::vector`.

Allowed kinds: integers and enumerations, pointers and references to objects with static storage, `std::nullptr_t`, and since C++20 floating-point values and class types whose members are all public. `template <auto V>` lets the parameter's type be deduced from the argument. Non-type parameters can also be deduced from a call: `template <typename T, std::size_t N> std::size_t length(const T (&)[N]) { return N; }` learns the length of a built-in array from the argument's type, which is how `std::size(arr)` works.

Parameter packs

```
#include <iostream>

template <typename... Ts>                    // Ts is a pack of types
void count_args(const Ts&... args) {         // args is a pack of parameters
    std::cout << sizeof...(args) << " arguments\n";
}

count_args(1, 2.5, "three");                 // Ts = {int, double, char[6]}: 3
count_args();                                // an empty pack is legal: 0 argu
```

`typename... Ts` declares a pack; `Ts...` or `args...` *expands* it — the compiler replaces the expansion with a comma-separated list, one entry per element. `sizeof...` gives the count without expanding. Expansion can carry a pattern: `f(args)...` becomes `f(a1), f(a2), f(a3)`, and `std::forward<Ts>(args)...` is the pattern `std::make_unique` uses to pass every constructor argument through unchanged. A pack can be empty, which matters when you write the base case below.

Recursion over a pack

Before C++17 the only way to *do something with each element* was to peel one off and recurse:

```
void print() { std::cout << '\n'; }                                  // base case: the

template <typename First, typename... Rest>
void print(const First& first, const Rest&... rest) {
    std::cout << first;
    if (sizeof...(rest) > 0) std::cout << ' ';
    print(rest...);                                                  // one shorter ca
}

print("total", 3, 2.5);                                             // total 3 2.5
```

`print("total", 3, 2.5)` instantiates `print<char[6], int, double>`, which calls `print<int, double>`, then `print<double>`, then the non-template `print()`. Four functions for one call, and the base case is not optional — without it the last call `print()` has no candidate and the error names it. `if constexpr (sizeof...(rest) > 0) print(rest...);` (lesson 6) removes the need for the base case by not compiling the recursive call when the pack is empty. The peel-one-off shape is still the right one when the first element is special: a separator argument, a format string, a stream.

Fold expressions (C++17)

A fold applies one binary operator across the whole pack in a single expression:

```cpp
template <typename... Ts>
auto sum(Ts... xs) { return (xs + ...); }             // unary right fold: x1

template <typename... Ts>
auto sum_from_zero(Ts... xs) { return (0 + ... + xs); } // binary left fold: (

template <typename... Ts>
bool all(Ts... flags) { return (... && flags); }      // true for an empty pac

template <typename... Ts>
void print_all(const Ts&... xs) {
    ((std::cout << xs << ' '), ...);                  // fold over the comma o
    std::cout << '\n';
}
```

Form	Spelling	Expands to
unary right fold	`(pack op ...)`	`x1 op (x2 op (x3))`
unary left fold	`(... op pack)`	`((x1 op x2) op x3)`
binary right fold	`(pack op ... op init)`	`x1 op (x2 op init)`
binary left fold	`(init op ... op pack)`	`(init op x1) op x2`

The parentheses are part of the syntax. Left and right differ only when the operator is not associative — `(xs - ...)` and `(... - xs)` give different answers — and the binary forms supply a starting value, which is what makes them legal on an empty pack: `(xs + ...)` with no arguments is a compile error, `(0 + ... + xs)` is `0`. Only `&&`, `||` and `,` may fold an empty pack unaided, giving `true`, `false` and nothing. The comma fold is the everyday one: it runs an expression once per element, in order, and replaces the whole recursive `print`. `(std::cout << ... << xs)` is a binary left fold over `<<` that prints every element with no separator.

Variadic class templates

`std::tuple<Ts...>` stores one member per pack element; `std::variant<Ts...>` stores one of them at a time (Module 15). The one variadic class you will write yourself is the visitor helper:

```cpp
template <typename... Fs>
struct Overloaded : Fs... {                    // inherit from every lambda
    using Fs::operator()...;                   // and pull in every call operator
};
```

Passed to `std::visit`, it lets a handful of lambdas act as one overload set; Module 15 shows the whole idiom.

Pitfalls

Mistake	What happens
`Ring<int, n>` with a run-time `n`	Not a constant expression: compile error.
Passing `Ring<int, 4>` to a function taking `Ring<int, 8>&`	Different types: no match.
`(xs + ...)` on an empty pack	Compile error; use `(0 + ... + xs)`.
Recursion without a base case	`no matching function for call to 'print()'`.
`f(args...)` when you meant `f(args)...`	Passes all arguments to one call instead of calling once per argument.
`sizeof(Ts)...` for `sizeof...(Ts)`	The former is a list of sizes, not a count.
Reading `(xs - ...)` as left to right	It is a right fold: `x1 - (x2 - x3)`.

Key takeaways

A non-type parameter is a compile-time value baked into the type; `std::array<T, N>` stores no size because `N` is the type.

A template argument must be a constant expression; dispatch run-time values to a few fixed instantiations.

`typename... Ts` declares a pack, `Ts...` expands it, `sizeof...(Ts)` counts it, and a pack may be empty.

Recursion peels one element per call and needs a base case; `if constexpr` can replace it.

Fold expressions apply one operator across a pack; the binary forms are safe on an empty pack, and the comma fold runs code per element.

EXERCISE 1

Ring<T, N> with the capacity chosen at run time

Write `template <typename T, std::size_t N> class Ring` — a fixed-capacity buffer over a `std::array<T, N>` that overwrites its oldest entry when full — with `push`, `size`, `at(i)` (`0` is the oldest) and a `static constexpr std::size_t capacity = N`. Then a function template `run<N>()` that prints `capacity=<N>` from `Ring<int, N>::capacity` and processes commands.

The first line is the capacity, one of `2`, `4` or `8`. It is read at run time, so it cannot be a template argument directly: dispatch to `run<2>()`, `run<4>()` or `run<8>()`. Commands: `push <v>`; `size` (print `size=<k>`); `oldest` (print `oldest=<v>` or `empty`); `print` (print `ring: <oldest> … <newest>` or `ring: (empty)`).

Input: the capacity, then commands. **Output:** `capacity=<N>` first, then one line per `size`, `oldest` and `print`.

```
4
push 1
push 2
push 3
push 4
push 5
print
size
oldest
```

prints

```
capacity=4
ring: 2 3 4 5
size=4
oldest=2
```

INPUT 1

```
4
push 1
push 2
push 3
push 4
push 5
print
size
oldest
```

OUTPUT 1

```
capacity=4
ring: 2 3 4 5
size=4
oldest=2
```

INPUT 2

```
2
print
push 10
push 20
push 30
print
```

OUTPUT 2

```
capacity=2
ring: (empty)
ring: 20 30
```

REFERENCE SOLUTION

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

template <typename T, std::size_t N>
class Ring {
public:
    static constexpr std::size_t capacity = N;

    void push(const T& value) {
        data_[(start_ + size_) % N] = value;
        if (size_ < N) ++size_;
        else start_ = (start_ + 1) % N;     // full: the slot just written was the o
    }
    std::size_t size() const { return size_; }
    const T& at(std::size_t i) const { return data_[(start_ + i) % N]; }

private:
    std::array<T, N> data_{};
    std::size_t start_ = 0;
    std::size_t size_ = 0;
};

template <std::size_t N>
void run() {
    Ring<int, N> ring;
    std::cout << "capacity=" << Ring<int, N>::capacity << '\n';
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "push") {
            int v;
            std::cin >> v;
            ring.push(v);
        } else if (cmd == "size") {
            std::cout << "size=" << ring.size() << '\n';
        } else if (cmd == "oldest") {
            if (ring.size() == 0) std::cout << "empty\n";
            else std::cout << "oldest=" << ring.at(0) << '\n';
        } else if (cmd == "print") {
            std::cout << "ring:";
            if (ring.size() == 0) std::cout << " (empty)";
            for (std::size_t i = 0; i < ring.size(); ++i) std::cout << ' ' << ring.a
            std::cout << '\n';
        }
    }
}

int main() {
```

```
    int n;
    std::cin >> n;
    if (n == 2) run<2>();
    else if (n == 8) run<8>();
    else run<4>();
    return 0;
}
```

EXERCISE 2

A variadic report built from fold expressions

Write `template <typename... Ts> void report(Ts... xs)` that prints one line: `args=<count> [<x1> <x2> …] sum=<s> all_positive=<yes|no>`. The count is `sizeof...(xs)`; the values are printed by a fold over the comma operator with a single space between them (and `[]` for no values); `sum` is a **binary** fold `(0LL + ... + xs)` so that an empty pack sums to 0 and large values do not overflow `int`; `all_positive` is the fold `(... && (xs > 0))`, which is `yes` for an empty pack. No loops over the pack, no recursion — folds only.

Read `n`, then `n` lines of `<k> <v1> … <vk>` with `k` from 0 to 4 and integer values. A pack's size is fixed at compile time, so read the values into an array and dispatch on `k`: `report()`, `report(v[0])`, `report(v[0], v[1])` and so on — five instantiations of one template.

Input: `n`, then `n` lines. **Output:** one line per input line.

```
4
3 1 2 3
0
2 -1 5
1 7
```

prints

```
args=3 [1 2 3] sum=6 all_positive=yes
args=0 [] sum=0 all_positive=yes
args=2 [-1 5] sum=4 all_positive=no
args=1 [7] sum=7 all_positive=yes
```

INPUT 1

```
4
3 1 2 3
0
2 -1 5
1 7
```

OUTPUT 1

```
args=3 [1 2 3] sum=6 all_positive=yes
args=0 [] sum=0 all_positive=yes
args=2 [-1 5] sum=4 all_positive=no
args=1 [7] sum=7 all_positive=yes
```

INPUT 2

```
1
4 10 20 30 40
```

OUTPUT 2

```
args=4 [10 20 30 40] sum=100
all_positive=yes
```

REFERENCE SOLUTION

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

template <typename... Ts>
void report(Ts... xs) {
    std::cout << "args=" << sizeof...(xs) << " [";
    [[maybe_unused]] std::size_t i = 0;
    ((std::cout << (i++ ? " " : "") << xs), ...);          // comma fold: one print
    const long long sum = (0LL + ... + xs);                  // binary left fold: le
    const bool allPositive = (... && (xs > 0));              // && folds an empty pa
    std::cout << "] sum=" << sum << " all_positive=" << (allPositive ? "yes" : "no")
}

int main() {
    int n;
    std::cin >> n;
    for (int line = 0; line < n; ++line) {
        int k;
        std::cin >> k;
        std::array<long long, 4> v{};
        for (int i = 0; i < k; ++i) std::cin >> v[i];
        if (k == 0) report();
        else if (k == 1) report(v[0]);
        else if (k == 2) report(v[0], v[1]);
        else if (k == 3) report(v[0], v[1], v[2]);
        else report(v[0], v[1], v[2], v[3]);
    }
    return 0;
}
```

QUIZ

1. What happens here?

```
int n;
std::cin >> n;
Ring<int, n> r;
```

A. A ring of `n` slots is created

B. Compile error: `n` is not usable in a constant expression

C. Undefined behaviour if `n` is 0

D. It compiles only with `-O2`

2. What does this print?

```
template <typename... Ts>
auto diff(Ts... xs) { return (xs - ...); }

int main() { std::cout << diff(10, 3, 2); }
```

A. `5`

B. `9`

C. `-5`

D. Compile error: `-` cannot be folded

3. What does this print?

```
template <typename... Ts>
bool all(Ts... flags) { return (... && flags); }

int main() { std::cout << all(); }
```

A. `1`

B. `0`

C. Compile error: an empty pack cannot be folded

D. Nothing — the function is never instantiated

4. Which expression gives the number of elements in the pack `Ts`?

A. `sizeof(Ts)...`

B. `sizeof...(Ts)`

C. `Ts::size()`

D. `sizeof(Ts...)`

5. Given a pack `args` of three values, what is the difference between `f(args...)` and `f(args)...`?

A. None — they are two spellings of the same expansion

B. `f(args...)` calls `f` once with three arguments; `f(args)...` expands to three calls, one per element

C. `f(args)...` is invalid outside a fold

D. `f(args...)` calls `f` three times

6. Why does `std::array<int, 4>` have no data member holding its size?

A. It stores the size in the first slot

B. The size is the non-type argument `4`, part of the type, so `size()` returns a constant

C. It computes the size from `sizeof` at run time

D. It does; `sizeof(std::array<int, 4>)` is 24 on this platform

7. What does the recursive `print(const First& first, const Rest&... rest)` need that the fold version does not?

A. A `template <>` specialisation for one argument

B. A non-template `print()` overload for the empty pack, or an `if constexpr` guard on the recursive call

C. A `sizeof...` check before every output

D. Nothing — recursion over a pack ends by itself

1. B A non-type template argument must be a constant expression, and a value read at run time is not one. Dispatch to a few fixed instantiations (`if (n == 4) run<4>();`) or use a `std::vector` when the size truly comes from input.

2. B `(xs - ...)` is a unary *right* fold: `10 - (3 - 2)` = 9. The left fold `(... - xs)` would give `(10 - 3) - 2` = 5. The two forms differ only for non-associative operators, which is exactly when you must read the dots carefully.

3. A Only `&&`, `||` and `,` may fold an empty pack unaided, and `&&` gives `true`. The same call with `(flags + ...)` would be a compile error; `(0 + ... + flags)` would print `0`.

4. B `sizeof...(Ts)` counts the pack. `sizeof(Ts)...` is a pack expansion producing a *list* of sizes, one per type, and the last option is a syntax error.

5. B The dots expand whatever pattern precedes them. `args...` expands the pack inside one argument list; `f(args)...` makes the whole call the pattern, so it becomes `f(a1), f(a2), f(a3)`. Writing one when you meant the other is the classic variadic mistake.

6. B `N` is baked into the type, so `std::array<int, 4>` is exactly four `int`s — 16 bytes on this platform — and `size()` is a `constexpr` function returning `N`. That is why `std::array<int, 4>` and `std::array<int, 8>` are unrelated types.

7. B Each call peels one element and recurses with one fewer; the last call is `print()` with an empty pack, and without a candidate for it the compiler reports `no matching function`. A comma fold `((std::cout << xs << ' '), ...)` does the same work in one expression and has no base case to forget.

LESSON 5 · 15 MIN

Concepts and constraints — saying what T must be

Every template has requirements: `max_of` needs `<`, `sum` needs `+`, `std::sort` needs a random-access range of things that compare. Before C++20 those requirements lived only in the body, so a wrong argument produced an error from deep inside the template — forty lines of notes ending in a line of library code you did not write. Concepts move the requirement to the signature, where the compiler can check it at the call and say which one failed. This lesson covers `requires` clauses and the standard concepts, writing your own concept with a `requires` expression, constrained `auto`, overloading on concepts and the subsumption rule, and `if constexpr` as the in-body companion.

The problem, measured

```
template <typename T>
T max_of(const T& a, const T& b) { return b < a ? a : b; }

struct Point { int x, y; };
max_of(Point{1, 2}, Point{3, 4});
```

```
error: no match for 'operator<' (operand types are 'const Point' and 'const Po
    return b < a ? a : b;
           ~~^~~
note: required from 'T max_of(const T&, const T&) [with T = Point]'
```

Two lines here; with `std::sort` on a vector of `Point` the same mistake produces a page, because the missing `<` is discovered six calls down inside `<bits/stl_algo.h>`. The template was honest about nothing: its signature said "any `T`", and only the body knew better.

---

requires clauses and the standard concepts

```cpp
#include <concepts>

template <typename T>
    requires std::integral<T>                // a requires clause after the te
T half(T v) { return v / 2; }

template <std::floating_point T>             // the same constraint, shorthand
T half(T v) { return v / 2; }

void show(std::integral auto v) {            // constrained auto: an abbreviat
    std::cout << v << '\n';
}
```

All three spellings declare a template whose parameter must satisfy a concept; choose by taste and keep one style per file. Now `half(Point{})` fails at the call: `error: no matching function … note: constraints not satisfied … 'std::integral<Point>' evaluated to 'false'`. Two lines, naming the requirement, pointing at your code.

`<concepts>` supplies the vocabulary. The ones you will use: `std::integral`, `std::signed_integral`, `std::unsigned_integral`, `std::floating_point`; `std::same_as<T, U>` and `std::convertible_to<From, To>`; `std::derived_from<D, B>`; `std::invocable<F, Args...>` and `std::predicate<F, Args...>` for callables; `std::equality_comparable` and `std::totally_ordered`; `std::copyable`, `std::movable`, `std::default_initializable`; and `std::ranges::range` from `<ranges>`. Two details bite: `bool` and `char` satisfy `std::integral` — they are integer types to the language — and there is no `std::arithmetic`; write `std::integral<T> || std::floating_point<T>` or name it yourself, next.

Writing a concept

A concept is a named compile-time predicate on types, and a `requires` expression is how you state what code must be valid:

```cpp
template <typename T>
concept Number = std::integral<T> || std::floating_point<T>;

template <typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;      // compound requirement: valid,
};

template <typename S>
concept Shape = requires(const S& s) {
    { s.area() } -> std::convertible_to<double>;
    { s.name() } -> std::same_as<std::string>;
};

template <typename C>
concept Container = requires(C c) {
    typename C::value_type;                   // type requirement: this neste
    c.begin();                                // simple requirement: this exp
    { c.size() } -> std::convertible_to<std::size_t>;
    requires std::copyable<C>;                // nested requirement: another
};

static_assert(Addable<int>);
static_assert(Addable<std::string>);          // + concatenates: satisfied
static_assert(!Addable<std::vector<int>>);    // no operator+: not satisfied,
```

The parameters of a `requires` expression are never created and its body is never executed — it is checked, not run, and the result is a `bool` you can `static_assert`, combine with `&&` and `||`, or test in `if constexpr`. Four kinds of requirement cover everything: a simple expression, a type name, a compound `{ expr } -> concept` that also checks the result's type, and a nested `requires`. Note the return-type check is itself a concept applied to the expression's type, so `-> std::same_as<std::string>` means "exactly `std::string`", while `-> std::convertible_to<double>` also admits `float` and `int`.

A concept checks syntax, not meaning. `Addable<std::string>` is true because `+` compiles; whether concatenation is what a `sum` should do to strings is a decision the concept cannot make for you.

Overloading on concepts and subsumption

```cpp
template <std::integral T>        void classify(T) { std::cout << "integral\n";
template <std::floating_point T> void classify(T) { std::cout << "floating\n";
template <typename T>             void classify(T) { std::cout << "something el

classify(7);      // integral
classify(2.5);    // floating
classify("s");    // something else
```

Overload resolution treats a satisfied constraint as a tie-breaker: when two templates match equally well, the constrained one beats the unconstrained one, and between two constrained ones the *more constrained* wins — provided the compiler can prove one subsumes the other. It can when the constraints are built from named concepts: `std::signed_integral<T>` is defined as `std::integral<T> && std::is_signed_v<T>`, so an overload constrained on `signed_integral` subsumes one constrained on `integral` and wins for `int`. It cannot see through a `requires` expression you wrote inline: two overloads with hand-written `requires(T a) { a + a; }` clauses are never equivalent even when the text is identical, and the call is ambiguous. Name your concepts and build the specific ones from the general ones.

Constrained auto everywhere

The `auto` in front of a variable, a return type or a lambda parameter accepts the same prefix:

```cpp
std::integral auto n = read_count();                // fails to compile if rea
std::floating_point auto mean(const auto& v) { … }  // the return type is chec
auto is_even = [](std::integral auto x) { return x % 2 == 0; };
```

The check happens when the type is deduced, so a constrained variable documents an assumption the compiler then enforces — the modern replacement for a comment saying `// must be an integer`.

---

if constexpr

A concept is a `bool`, and `if constexpr` chooses code at compile time, so the two fit together for the case where one function handles several kinds of type with a few lines of difference:

```cpp
template <typename T>
std::string describe(const T& v) {
    if constexpr (std::integral<T>) return "integral " + std::to_string(v);
    else if constexpr (std::floating_point<T>) return "floating " + std::to_st
    else return "something else";
}
```

The branch not taken is *discarded*: it is not instantiated for that `T`, so `std::to_string(v)` in the first branch is never compiled for a `std::string` argument. An ordinary `if` would try, and fail. Lesson 6 makes this the centre of a whole dispatch style.

Pitfalls

Mistake	What happens
`requires T` instead of `requires std::integral<T>`	A constraint must be a `bool` expression: compile error.
A `requires` clause in the wrong place	It goes after the template header or after the parameter list, not between the name and the parameters.
Two inline `requires` expressions on two overloads	Never subsume each other: ambiguous call.
Forgetting `#include <concepts>`	`std::integral` is undeclared.
Assuming `bool` is not `std::integral`	It is; add `&& !std::same_as<T, bool>` when that matters.
Reading `Addable<T>` as "addition makes sense"	It means only that `a + b` compiles.
`requires requires (T t) { … }` on a function	Legal ad-hoc constraint, but a sign the concept deserves a name.

Key takeaways

A concept moves a template's requirement into its signature, so a bad argument fails at the call with a two-line message.

`requires C<T>`, `template <C T>` and `C auto` are three spellings of the same constraint.

A `requires` expression lists code that must be valid — expressions, types, result types, nested constraints — and is a `bool`.

Constrained overloads beat unconstrained ones; more-constrained beats less, but only through named concepts.

`bool` and `char` are `std::integral`; there is no `std::arithmetic` concept.

`if constexpr` discards the branch not taken, so one body can serve several kinds of type.

EXERCISE 1

average chosen by std::integral and std::floating_point

Write three overloads of `average(const std::vector<T>& v)`. `template <std::integral T>` sums into a `long long` and prints `integral: sum=<sum> mean=<mean>`; `template <std::floating_point T>` sums into a `double` and prints `floating: sum=<sum> mean=<mean>`; the unconstrained `template <typename T>` prints `unconstrained: no average for <k> values`. Means and floating sums have two decimals. For an `int` vector the first and third overloads both match — the constrained one wins the tie.

Read `n`, then `n` lines of `<type> <k> <v1> … <vk>` where `<type>` is `int`, `double` or `string`; `k` is at least 1 for the numeric types. A helper `handle<T>()` reads the vector and calls `average`.

Input: `n`, then `n` lines. **Output:** one line per input line.

```
3
int 4 1 2 3 4
double 2 1.5 2.5
string 2 pear apple
```

prints

```
integral: sum=10 mean=2.50
floating: sum=4.00 mean=2.00
unconstrained: no average for 2 values
```

INPUT 1

```
3
int 4 1 2 3 4
double 2 1.5 2.5
string 2 pear apple
```

OUTPUT 1

```
integral: sum=10 mean=2.50
floating: sum=4.00 mean=2.00
unconstrained: no average for 2 values
```

INPUT 2

```
1
int 3 -1 -2 -3
```

OUTPUT 2

```
integral: sum=-6 mean=-2.00
```

REFERENCE SOLUTION

```cpp
#include <concepts>
#include <cstddef>
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>

template <std::integral T>
void average(const std::vector<T>& v) {
    long long sum = 0;
    for (T x : v) sum += x;
    std::cout << "integral: sum=" << sum << " mean=" << std::fixed << std::setprecis
              << static_cast<double>(sum) / static_cast<double>(v.size()) << '\n';
}

template <std::floating_point T>
void average(const std::vector<T>& v) {
    double sum = 0.0;
    for (T x : v) sum += x;
    std::cout << "floating: sum=" << std::fixed << std::setprecision(2) << sum
              << " mean=" << sum / static_cast<double>(v.size()) << '\n';
}

template <typename T>
void average(const std::vector<T>& v) {
    std::cout << "unconstrained: no average for " << v.size() << " values\n";
}

template <typename T>
void handle() {
    std::size_t k;
    std::cin >> k;
    std::vector<T> v(k);
    for (T& x : v) std::cin >> x;
    average(v);
}

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string type;
        std::cin >> type;
        if (type == "int") handle<int>();
        else if (type == "double") handle<double>();
        else handle<std::string>();
    }
```

```
    return 0;
}
```

EXERCISE 2

A Shape concept written as a requires expression

Write a concept `Shape` with a `requires` expression: for a `const S& s`, `s.area()` must be valid with a result convertible to `double`, and `s.name()` must return exactly `std::string` (→ `std::same_as<std::string>`). Define `Circle` (radius; area `pi * r * r` with `std::numbers::pi`), `Rect` (width and height) and `Square` (side) so they satisfy it, and add `static_assert(Shape<Circle>)` and `static_assert(!Shape<int>)`. Write `void report(const Shape auto& s)` — constrained `auto` — that prints `<name>: area=<area>`.

Read `n` (at least 1), then `n` lines: `circle <r>`, `rect <w> <h>` or `square <s>`. Report each shape as it is read; after the last, print `total=<sum of areas>` and `largest=#<index> <name>` for the shape with the largest area (1-based index; the first on a tie). All areas have two decimals.

Input: `n`, then `n` shape lines. **Output:** `n` report lines, then `total=…` and `largest=…`.

```
3
circle 2
rect 3 4
square 3
```

prints

```
circle: area=12.57
rect: area=12.00
square: area=9.00
total=33.57
largest=#1 circle
```

INPUT 1

```
3
circle 2
rect 3 4
square 3
```

OUTPUT 1

```
circle: area=12.57
rect: area=12.00
square: area=9.00
total=33.57
largest=#1 circle
```

INPUT 2

```
1
square 2.5
```

OUTPUT 2

```
square: area=6.25
total=6.25
largest=#1 square
```

REFERENCE SOLUTION

```cpp
#include <concepts>
#include <iomanip>
#include <iostream>
#include <numbers>
#include <string>

template <typename S>
concept Shape = requires(const S& s) {
    { s.area() } -> std::convertible_to<double>;
    { s.name() } -> std::same_as<std::string>;
};

struct Circle {
    double r;
    double area() const { return std::numbers::pi * r * r; }
    std::string name() const { return "circle"; }
};

struct Rect {
    double w, h;
    double area() const { return w * h; }
    std::string name() const { return "rect"; }
};

struct Square {
    double side;
    double area() const { return side * side; }
    std::string name() const { return "square"; }
};

static_assert(Shape<Circle> && Shape<Rect> && Shape<Square>);
static_assert(!Shape<int>);
static_assert(!Shape<std::string>);

void report(const Shape auto& s) {
    std::cout << s.name() << ": area=" << s.area() << '\n';
}

struct Tally {
    double total = 0.0;
    double largestArea = -1.0;
    std::string largest;
};

void account(const Shape auto& s, int index, Tally& tally) {
    report(s);
    const std::floating_point auto area = s.area();
    tally.total += area;
    if (tally.largestArea < area) {
```

```
        tally.largestArea = area;
        tally.largest = "#" + std::to_string(index) + " " + s.name();
    }
}

int main() {
    std::cout << std::fixed << std::setprecision(2);
    int n;
    std::cin >> n;
    Tally tally;
    for (int i = 1; i <= n; ++i) {
        std::string kind;
        std::cin >> kind;
        if (kind == "circle") {
            double r;
            std::cin >> r;
            account(Circle{r}, i, tally);
        } else if (kind == "rect") {
            double w, h;
            std::cin >> w >> h;
            account(Rect{w, h}, i, tally);
        } else {
            double s;
            std::cin >> s;
            account(Square{s}, i, tally);
        }
    }
    std::cout << "total=" << tally.total << '\n';
    std::cout << "largest=" << tally.largest << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```
template <std::integral T> void classify(T) { std::cout << "integral"; }
template <typename T>      void classify(T) { std::cout << "other"; }

int main() { classify(true); }
```

A. `integral`

B. `other`

C. Compile error: ambiguous call

D. Compile error: `bool` is not a template argument

2. What does this print?

```
template <std::integral T>        void f(T) { std::cout << "integral"; }
template <std::signed_integral T> void f(T) { std::cout << "signed"; }

int main() { f(7); }
```

A. `integral`

B. `signed`

C. Compile error: ambiguous call

D. Compile error: two templates cannot share constraints

3. Two overloads are constrained with identical inline expressions: `requires requires(T a) { a + a; }`. What happens when a call matches both?

A. The first declared wins

B. The call is ambiguous — inline `requires` expressions never subsume each other, even when textually identical

C. The compiler merges them into one overload

D. The last declared wins

4. `concept Addable = requires(T a, T b) { { a + b } -> std::convertible_to<T>; };` — is `Addable<std::string>` satisfied?

A. No — addition is a numeric operation

B. Yes — `+` concatenates and the result is a `std::string`

C. No — `std::string` is a class type

D. It is a compile error to test a concept on a class

5. Which three spellings declare the same constrained template?

A. `requires T`, `template <T>`, `T auto`

B. `template <typename T> requires std::integral<T>`, `template <std::integral T>`, `void f(std::integral auto v)`

C. `std::integral<T>`, `std::integral T`, `std::integral(T)`

D. `concept T`, `typename T`, `auto T`

6. In `describe`, why does the first branch not break compilation for a `std::string` argument?

```
template <typename T>
std::string describe(const T& v) {
    if constexpr (std::integral<T>) return "integral " + std::to_string(v);
    else return "something else";
}
```

A. `std::to_string` has an overload for `std::string`

B. `if constexpr` discards the branch whose condition is false, so `std::to_string(v)` is never instantiated for `std::string`

C. The compiler evaluates the condition at run time and skips the call

D. It does break; the function fails to compile for `std::string`

7. What is the difference between `{ s.name() } -> std::same_as<std::string>;` and `{ s.name() } -> std::convertible_to<std::string>;`?

A. None — both accept any type that can become a `std::string`

B. `same_as` requires exactly `std::string`; `convertible_to` also admits `const char*` and anything else that converts

C. `same_as` is a run-time check, `convertible_to` a compile-time one

D. `convertible_to` is stricter

ANSWER KEY · CONCEPTS AND CONSTRAINTS — SAYING WHAT T MUST BE

1. A `bool` (and `char`) satisfy `std::integral` — they are integer types to the language. Both templates match equally well and the constrained one wins the tie. Add `&& !std::same_as<T, bool>` when a boolean must not count.

2. B `std::signed_integral<T>` is defined as `std::integral<T> && std::is_signed_v<T>`, so it subsumes `std::integral` and the more constrained overload wins. `f(7u)` would print `integral`. Subsumption works only through named concepts built from one another.

3. B The subsumption rule compares *named* concept atoms; two hand-written `requires` expressions are always different atoms. Name the concept and build the specific one from the general one, and the compiler can prove which is more constrained.

4. B A concept checks that the code is *valid*, not that it means what you intend. `a + b` compiles for two strings and yields a `std::string`, so the requirement holds. Whether a generic `sum` should concatenate is a decision the concept cannot make for you.

5. B A `requires` clause after the template header, a concept in place of `typename`, and a concept in front of an `auto` parameter all attach the same constraint. `requires T` is an error — a constraint must be a `bool` expression — and the other options are not C++.

6. B For `T = std::string` the condition is `false` at compile time and the whole first branch is discarded — not instantiated at all. A plain `if` would compile both branches for every `T` and `std::to_string(std::string)` does not exist.

7. B The return-type check applies a concept to the expression's type. `std::same_as<std::string>` holds only for `std::string` itself; `std::convertible_to<std::string>` is satisfied by `const char*`, `std::string_view`-like types with a conversion, and so on.

LESSON 6 · 15 MIN

Type traits and metaprogramming — computing with types

A type trait is a class template that answers a question about a type at compile time — is it an integer, is it a pointer, what is it once the `const` and the `&` are stripped — or that manufactures a new type from an old one. `<type_traits>` holds a hundred of them, every one built from the specialisation machinery of lesson 3, and they are what makes `if constexpr` dispatch, forwarding references and generic containers work. This lesson covers the traits you will actually use, `decltype` and `std::declval` for naming the type of an expression, `if constexpr` as the dispatch tool, SFINAE and `std::enable_if` as the pre-C++20 technique you must be able to read, and why a `constexpr` function beats template recursion for computing values.

The predicates

```
#include <stdexcept>
#include <string>
#include <type_traits>

static_assert(std::is_integral_v<int>);
static_assert(std::is_integral_v<bool>);            // bool is an integer type
static_assert(std::is_floating_point_v<double>);
static_assert(std::is_same_v<int, signed int>);
static_assert(!std::is_same_v<int, long>);          // different types even wh
static_assert(std::is_pointer_v<const char*>);
static_assert(std::is_class_v<std::string>);
static_assert(std::is_base_of_v<std::exception, std::runtime_error>);
static_assert(std::is_convertible_v<int, double>);
```

Each `std::is_x<T>` is a struct with a `static constexpr bool value`, and `std::is_x_v<T>` is the C++17 variable template that saves writing `::value`. The implementation is lesson 3's trait pattern: a primary template deriving from `std::false_type`, specialisations for the matching types deriving from `std::true_type`, and a partial specialisation `is_pointer<T*>` for the pointer family. Two more you will meet: `std::is_arithmetic_v` (integral or floating), and `std::is_nothrow_move_constructible_v`, which `std::vector` consults before deciding whether to move or copy elements when it grows (Module 9).

The transformers

Traits that produce a type expose it as `::type`, with a `_t` alias:

Trait	`int` becomes	Used for
`std::remove_cvref_t<const int&>`	`int`	Naming the plain type behind a forwarding reference
`std::remove_pointer_t<int*>`	`int`	Pointee of a pointer parameter
`std::add_pointer_t<int>`	`int*`	Building a pointer type
`std::make_unsigned_t<int>`	`unsigned int`	Bit manipulation on a signed input
`std::decay_t<const char(&)[6]>`	`const char*`	What a by-value parameter would receive
`std::conditional_t<B, T, F>`	`T` if `B` else `F`	Choosing a type by a compile-time flag
`std::common_type_t<int, double>`	`double`	The type `?:` would produce
`std::underlying_type_t<E>`	the enum's integer type	Printing an `enum class`

`std::conditional_t` is a type-level `?:` and the everyday way to pick a representation:

```
template <typename T>
using Accumulator = std::conditional_t<std::is_integral_v<T>, long long, doubl

template <typename T>
Accumulator<T> total(const std::vector<T>& v) {
    Accumulator<T> sum{};
    for (const T& x : v) sum += x;
    return sum;
}
```

`total` on a `std::vector<int>` sums into a `long long` and cannot overflow on a few thousand values; on a `std::vector<float>` it sums into a `double`. Both branch types must

be valid — `conditional_t` selects, it does not discard — so it cannot choose between "a type that exists" and "one that does not".

`remove_cvref_t` earns its place with forwarding references. `template <typename T> void f(T&& x)` deduces `T` as `std::string&` for an lvalue `std::string`, so `std::is_same_v<T, std::string>` is `false` for the very argument you meant to catch. Strip first: `using U = std::remove_cvref_t<T>;` then test `U`. The same applies to `auto&&` in a range-for.

---

decltype and std::declval

`decltype(expr)` is the type of an expression, computed without evaluating it:

```
int i = 0;
decltype(i) j = 1;                  // int
decltype((i)) r = i;                // int& — an extra pair of parentheses make

template <typename A, typename B>
auto add(const A& a, const B& b) → decltype(a + b) { return a + b; }

template <typename C>
using SizeOf = decltype(std::declval<C>().size());     // the type size() retu
```

`std::declval<T>()` pretends to produce a `T` inside an unevaluated context — `decltype`, `sizeof`, a `requires` expression — so you can ask what an expression *would* be without needing an object, or even a constructor. It has no definition and calling it in real code is a link error by design. The `decltype(i)` versus `decltype((i))` distinction is a well-known interview question: a name gives the declared type, a parenthesised name is an lvalue expression and gives a reference.

---

if constexpr dispatch

Where lesson 5 used `if constexpr` with a concept, the traits version reads the same and predates it:

```cpp
template <typename T>
void describe(const T& v) {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "integral " << v << (v % 2 == 0 ? " even" : " odd") << '
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "floating " << std::llround(v) << '\n';
    } else if constexpr (std::is_same_v<T, std::string>) {
        std::cout << "text of " << v.size() << " chars\n";
    } else {
        std::cout << "other\n";
    }
}
```

Only the branch whose condition is true is instantiated for a given `T`; `v % 2` is never compiled for a `double` and `v.size()` never for an `int`. To make an unsupported type a compile error rather than an `"other"` line, the last branch can `static_assert` — but `static_assert(false)` inside a template is ill-formed on its own (the compiler may reject it before any instantiation), so the idiom is a dependent false: `template <typename> inline constexpr bool always_false = false;` and then `static_assert(always_false<T>, "unsupported type")`. C++23 relaxes this, but the judge is C++20.

SFINAE and enable_if — the old way

Before concepts, a template was switched on and off with a trick called *substitution failure is not an error*: when substituting a deduced `T` into a candidate's signature produces an invalid type, that candidate is dropped silently instead of ending compilation.

```cpp
template <typename T>
std::enable_if_t<std::is_integral_v<T>, T> half(T v) { return v / 2; }

template <typename T>
std::enable_if_t<std::is_floating_point_v<T>, T> half(T v) { return v / 2; }
```

`std::enable_if_t<false, T>` names a `::type` that does not exist, so for `half(2.5)` the first candidate fails substitution and vanishes, leaving the second. The technique ran a decade of library code and every error it produced was a list of "candidate template ignored: substitution failure" notes. You will read it in older codebases and in the standard library's own headers; you should write the concept version from lesson 5, which says the same thing in the signature and produces a message that names the unmet requirement.

Compile-time recursion versus constexpr

The original template metaprogramming computed values by recursion and specialisation:

```
template <unsigned N>
struct Factorial { static constexpr unsigned long long value = N * Factorial<N
template <>
struct Factorial<0> { static constexpr unsigned long long value = 1; };

static_assert(Factorial<10>::value == 3628800);

constexpr unsigned long long factorial(unsigned n) {      // the modern form
    unsigned long long result = 1;
    for (unsigned i = 2; i <= n; ++i) result *= i;
    return result;
}
static_assert(factorial(10) == 3628800);
```

The template version instantiates eleven classes to compute one number, cannot use a loop, and reports mistakes as a chain of instantiation notes. The `constexpr` function is ordinary C++ evaluated at compile time when its arguments are constants — C++20 allows loops, `std::vector` and `std::string` inside — and Module 16 covers it in depth. The rule that survives: compute *values* with `constexpr` functions, compute *types* with traits.

Pitfalls

Mistake	What happens
`std::is_same_v<T, std::string>` with `T&&` parameter	`T` is `std::string&`: false. Strip with `remove_cvref_t`.
Testing `std::is_integral_v<T>` to mean "a number to print"	`bool` and `char` are integral too.
`std::conditional_t<B, T, F>` with an invalid `F`	Both branches must be valid types.
`static_assert(false)` in the last `if constexpr` branch	Ill-formed before C++23; use a dependent false.
Plain `if` where `if constexpr` was needed	Every branch compiles for every `T`: `v.size()` on an `int` fails.
`decltype((x))` when `decltype(x)` was meant	A reference instead of the declared type.
Calling `std::declval<T>()` outside `decltype`	Undefined symbol at link time.

Key takeaways

`is_x_v<T>` predicates and `x_t<T>` transformers are the two halves of `<type_traits>`; both are traits built by specialisation.

`conditional_t` picks a type from a flag; `remove_cvref_t` recovers the plain type behind a forwarding reference.

`decltype` names an expression's type unevaluated; `std::declval<T>()` supplies a fake `T` for it.

`if constexpr` instantiates only the taken branch — the dispatch tool; end with a dependent-false `static_assert` when the rest must not compile.

SFINAE and `enable_if` are the pre-C++20 constraint; read them, write concepts.

Compute values with `constexpr` functions, types with traits; recursive template arithmetic is history.

EXERCISE 1

An if constexpr dispatcher over a forwarding reference

Write `template <typename T> void describe(T&& v)` — a **forwarding reference**, so that for an lvalue `std::string` the parameter `T` deduces as `std::string&`. Strip it first with `using U = std::remove_cvref_t<T>;` and dispatch on `U` with `if constexpr`, in this order: `std::is_same_v<U, bool>` prints `bool true` or `bool false`; `std::is_integral_v<U>` prints `integral <v> even` or `integral <v> odd`; `std::is_floating_point_v<U>` prints `floating <std::llround(v)>`; `std::is_same_v<U, std::string>` prints `text of <size> chars`; anything else prints `other`. The `bool` test must come before the integral one, because `bool` is integral.

Read `n`, then `n` lines of `<kind> <value>`: `int`, `double`, `string`, `bool` (`0` / `1`), `char` (one character — it is integral too, and prints as a character), and `pair <a> <b>` (a `std::pair<int, int>`, which is `other`). Read each value into a variable and call `describe(variable)` on it.

Input: `n`, then `n` lines. **Output:** one line per input line.

```
6
int 42
double 2.5
string hello
bool 1
char x
pair 1 2
```

prints

```
integral 42 even
floating 3
text of 5 chars
bool true
integral x even
other
```

INPUT 1

```
6
int 42
double 2.5
string hello
bool 1
char x
pair 1 2
```

OUTPUT 1

```
integral 42 even
floating 3
text of 5 chars
bool true
integral x even
other
```

INPUT 2

```
2
int -7
bool 0
```

OUTPUT 2

```
integral -7 odd
bool false
```

REFERENCE SOLUTION

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

template <typename T>
void describe(T&& v) {
    using U = std::remove_cvref_t<T>;          // T is std::string& for an lvalue: s
    if constexpr (std::is_same_v<U, bool>) {
        std::cout << "bool " << (v ? "true" : "false") << '\n';
    } else if constexpr (std::is_integral_v<U>) {
        std::cout << "integral " << v << (v % 2 == 0 ? " even" : " odd") << '\n';
    } else if constexpr (std::is_floating_point_v<U>) {
        std::cout << "floating " << std::llround(v) << '\n';
    } else if constexpr (std::is_same_v<U, std::string>) {
        std::cout << "text of " << v.size() << " chars\n";
    } else {
        std::cout << "other\n";
    }
}

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string kind;
        std::cin >> kind;
        if (kind == "int") {
            int v;
            std::cin >> v;
            describe(v);
        } else if (kind == "double") {
            double v;
            std::cin >> v;
            describe(v);
        } else if (kind == "string") {
            std::string v;
            std::cin >> v;
            describe(v);
        } else if (kind == "bool") {
            bool v;
            std::cin >> v;
            describe(v);
        } else if (kind == "char") {
            char v;
            std::cin >> v;
            describe(v);
        } else {
```

```
            std::pair<int, int> v;
            std::cin >> v.first >> v.second;
            describe(v);
        }
    }
    return 0;
}
```

EXERCISE 2

Pick the accumulator with std::conditional_t

Write an alias template `Accumulator<T>` = `std::conditional_t<std::is_integral_v<T>, long long, double>` and a function template `total(const std::vector<T>&)` that returns `Accumulator<T>` — the sum of the elements in the wider type. Print each result as `<kind>: sum=<sum> accumulator=<long long|double>`, deciding the label with `std::is_same_v<Accumulator<T>, long long>` in an `if constexpr`; a `double` sum has two decimals.

Read `n`, then `n` lines of `<kind> <k> <v1> … <vk>` where `<kind>` is `int`, `char` (single characters, summed by character code), `float` or `double`. `k` may be 0.

Input: `n`, then `n` lines. **Output:** one line per input line.

```
4
int 2 2000000000 2000000000
char 3 a b c
float 2 0.5 0.25
double 3 1.5 2.5 3
```

prints

```
int: sum=4000000000 accumulator=long long
char: sum=294 accumulator=long long
float: sum=0.75 accumulator=double
double: sum=7.00 accumulator=double
```

INPUT 1

```
4
int 2 2000000000 2000000000
char 3 a b c
float 2 0.5 0.25
double 3 1.5 2.5 3
```

OUTPUT 1

```
int: sum=4000000000 accumulator=long
long
char: sum=294 accumulator=long long
float: sum=0.75 accumulator=double
double: sum=7.00 accumulator=double
```

INPUT 2

```
1
int 0
```

OUTPUT 2

```
int: sum=0 accumulator=long long
```

REFERENCE SOLUTION

```cpp
#include <cstddef>
#include <iomanip>
#include <iostream>
#include <string>
#include <type_traits>
#include <vector>

template <typename T>
using Accumulator = std::conditional_t<std::is_integral_v<T>, long long, double>;

template <typename T>
Accumulator<T> total(const std::vector<T>& v) {
    Accumulator<T> sum{};
    for (const T& x : v) sum += x;
    return sum;
}

template <typename T>
void report(const std::string& kind) {
    std::size_t k;
    std::cin >> k;
    std::vector<T> v(k);
    for (T& x : v) std::cin >> x;
    const auto sum = total(v);
    std::cout << kind << ": sum=";
    if constexpr (std::is_same_v<Accumulator<T>, long long>) {
        std::cout << sum << " accumulator=long long\n";
    } else {
        std::cout << std::fixed << std::setprecision(2) << sum << " accumulator=doub
    }
}

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string kind;
        std::cin >> kind;
        if (kind == "int") report<int>(kind);
        else if (kind == "char") report<char>(kind);
        else if (kind == "float") report<float>(kind);
        else report<double>(kind);
    }
    return 0;
}
```

QUIZ

1. What does this print?

```
template <typename T>
void f(T&& x) { std::cout << std::is_same_v<T, std::string>; }

int main() {
    std::string s = "a";
    f(s);
}
```

A. `1`

B. `0`

C. Compile error: `T&&` cannot bind to an lvalue

D. Undefined behaviour

2. What does this print?

```
int i = 0;
decltype((i)) r = i;
r = 5;
std::cout << i;
```

A. `0`

B. `5`

C. Compile error: `decltype((i))` is not a type

D. Undefined behaviour

3. What type is `Accumulator<char>` given `template <typename T> using Accumulator = std::conditional_t<std::is_integral_v<T>, long long, double>;`?

A. `double`

B. `long long`

C. `char`

D. Compile error: `char` is neither

4. Why is `static_assert(false, "unsupported")` in the final `else` of an `if constexpr` chain a problem before C++23?

A. `static_assert` is not allowed inside templates

B. The condition does not depend on `T`, so the compiler may reject the template before any instantiation

C. It fires for every `T`, even the supported ones

D. It only warns and never stops compilation

5. With `template <typename T> std::enable_if_t<std::is_integral_v<T>, T> half(T v);` and a second `half` enabled for `std::is_floating_point_v<T>`, what happens to the first candidate when `half(2.5)` is called?

A. It is a compile error: `enable_if_t<false, T>` has no `type`

B. It is dropped silently — substitution failure is not an error — and the second candidate is chosen

C. It is chosen, and `2.5` is converted to `int`

D. Both are chosen and the call is ambiguous

6. Which is the recommended way to compute `factorial(10)` at compile time in C++20?

A. A recursive class template `Factorial<N>` with a specialisation for `0`

B. A `constexpr` function with a loop, used in a `static_assert` or a constant initialiser

C. A macro

D. A `std::conditional_t` chain

7. What happens if `std::declval<T>()` is called in ordinary code, outside `decltype`, `sizeof` or a `requires` expression?

   A. It returns a default-constructed `T`

   B. It compiles but fails to link — it has no definition by design

   C. It throws `std::bad_function_call`

   D. It returns a null reference

ANSWER KEY · TYPE TRAITS AND METAPROGRAMMING — COMPUTING WITH TYPES

1. B For an lvalue argument a forwarding reference deduces `T = std::string&`, and `std::string&` is not the same type as `std::string`. Strip first — `std::remove_cvref_t<T>` — and the test becomes true. `f(std::string("a"))` with an rvalue would have printed `1`.

2. B `decltype(i)` is the declared type `int`, but `decltype((i))` — a parenthesised name — is an lvalue expression and yields `int&`. `r` is a reference to `i`, so assigning through it changes `i`.

3. B `char` is an integral type, so the flag is `true` and `conditional_t` selects `long long`. It selects; it does not discard — both branch types must be valid regardless of the flag.

4. B A discarded branch is still checked for well-formedness where nothing depends on a template parameter, and `static_assert(false)` is ill-formed on its own. The idiom is a dependent false: `template <typename> inline constexpr bool always_false = false;` then `static_assert(always_false<T>, …)`.

5. B Substituting `T = double` makes the first signature name a `::type` that does not exist; SFINAE removes that candidate from the set instead of failing the compilation, leaving the floating-point overload. Concepts say the same thing in the signature with a readable error when nothing matches.

6. B The template version instantiates eleven classes, cannot use a loop and reports mistakes as a chain of instantiation notes; a `constexpr` function is ordinary C++ evaluated at compile time when its arguments are constants. Compute *values* with `constexpr` functions and *types* with traits.

7. B `std::declval` is declared, never defined, so it exists only to give unevaluated contexts an expression of type `T&&` without needing an object or a constructor. Odr-using it — actually calling it — is an undefined reference at link time, which is the intended guard.

CHECKPOINT · 28 MIN

Checkpoint — Templates and generic programming

This checkpoint covers the whole module: function templates and deduction, class templates with out-of-class members and CTAD, full and partial specialisation and the traits pattern, non-type parameters and packs with fold expressions, concepts and constrained `auto`, and the type traits behind `if constexpr` dispatch.

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

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

- Why does `max_of(3, 2.5)` fail to compile, and what are the two ways to fix it?
- Why must a template's definition live in the header rather than a `.cpp` file?
- What does `std::vector v{3}` deduce, and how many elements does it hold?
- Why is a function template overloaded rather than specialised?
- Which fold forms are legal on an empty pack?
- Is `bool` a `std::integral`, and why does that matter for an overload set?
- Why is `std::is_same_v<T, std::string>` false inside `f(T&& x)` for a `std::string` lvalue?

The three programs are a `Queue<T>` instantiated for the element type named on the first line, a statistics report whose overloads are chosen by concepts, and a `Table<K, V>` whose header line comes from a `TypeName` trait. Read the type word first; everything else follows from it.

EXERCISE 1

Queue<T> for the element type on the first line

Write a class template `Queue<T>` — first in, first out — over a `std::vector<T>` and a head index, with `push`, `pop`, `front`, `size` and `print`; define `pop`, `front` and `print` outside the class body with their own template headers. A function template `run<T>()` processes the commands; `main` reads the type word and calls `run<int>()` or `run<std::string>()`.

Commands, one per line: `push <v>`; `pop` (print `popped <v>` or `empty`); `front` (print `front=<v>` or `empty`); `size` (print `size=<k>`); `print` (print `queue: <front> … <back>` or `queue: (empty)`).

Input: the type word (`int` or `string`), then commands. **Output:** one line per `pop`, `front`, `size` and `print`.

```
int
push 1
push 2
push 3
print
pop
front
size
```

prints

```
queue: 1 2 3
popped 1
front=2
size=2
```

INPUT 1

```
int
push 1
push 2
push 3
print
pop
front
size
```

OUTPUT 1

```
queue: 1 2 3
popped 1
front=2
size=2
```

INPUT 2

```
string
push pear
push apple
pop
pop
pop
print
```

OUTPUT 2

```
popped pear
popped apple
empty
queue: (empty)
```

REFERENCE SOLUTION

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

template <typename T>
class Queue {
public:
    void push(const T& value) { items_.push_back(value); }
    bool pop(T& out);
    bool front(T& out) const;
    std::size_t size() const { return items_.size() - head_; }
    void print() const;

private:
    std::vector<T> items_;
    std::size_t head_ = 0;     // index of the front element
};

template <typename T>
bool Queue<T>::pop(T& out) {
    if (size() == 0) return false;
    out = items_[head_];
    ++head_;
    if (head_ == items_.size()) {      // drained: reclaim the storage
        items_.clear();
        head_ = 0;
    }
    return true;
}

template <typename T>
bool Queue<T>::front(T& out) const {
    if (size() == 0) return false;
    out = items_[head_];
    return true;
}

template <typename T>
void Queue<T>::print() const {
    std::cout << "queue:";
    if (size() == 0) {
        std::cout << " (empty)\n";
        return;
    }
    for (std::size_t i = head_; i < items_.size(); ++i) std::cout << ' ' << items_[i
    std::cout << '\n';
}
```

```cpp
template <typename T>
void run() {
    Queue<T> queue;
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "push") {
            T value;
            std::cin >> value;
            queue.push(value);
        } else if (cmd == "pop") {
            T value;
            if (queue.pop(value)) std::cout << "popped " << value << '\n';
            else std::cout << "empty\n";
        } else if (cmd == "front") {
            T value;
            if (queue.front(value)) std::cout << "front=" << value << '\n';
            else std::cout << "empty\n";
        } else if (cmd == "size") {
            std::cout << "size=" << queue.size() << '\n';
        } else if (cmd == "print") {
            queue.print();
        }
    }
}

int main() {
    std::string type;
    std::cin >> type;
    if (type == "int") run<int>();
    else run<std::string>();
    return 0;
}
```

EXERCISE 2

A statistics report chosen by concepts

Write a concept `Number` = `std::integral<T> || std::floating_point<T>`, a helper `template <Number T> T spread(const std::vector<T>&)` returning `max - min`, and three overloads of `stats(const std::vector<T>& v)`:

- `template <std::integral T>`: `integral n=<k> sum=<sum> min=<min> max=<max> spread=<spread>` — the sum in a `long long`.
- `template <std::floating_point T>`: `floating n=<k> mean=<mean> min=<min> max=<max> spread=<spread>` — all four with two decimals.

`template <typename T>` (unconstrained): `text n=<k> longest=<word> chars=<total characters>` — the first longest word on a tie.

Read `n`, then `n` lines of `<type> <k> <v1> … <vk>` with `<type>` one of `int`, `double`, `string` and `k` at least 1.

Input: `n`, then `n` lines. **Output:** one line per input line.

```
3
int 4 1 2 3 4
double 2 1.5 2.5
string 3 pear banana kiwi
```

prints

```
integral n=4 sum=10 min=1 max=4 spread=3
floating n=2 mean=2.00 min=1.50 max=2.50 spread=1.00
text n=3 longest=banana chars=14
```

INPUT 1

```
3
int 4 1 2 3 4
double 2 1.5 2.5
string 3 pear banana kiwi
```

OUTPUT 1

```
integral n=4 sum=10 min=1 max=4 spread=3
floating n=2 mean=2.00 min=1.50 max=2.50
spread=1.00
text n=3 longest=banana chars=14
```

INPUT 2

```
1
int 1 -9
```

OUTPUT 2

```
integral n=1 sum=-9 min=-9 max=-9
spread=0
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <concepts>
#include <cstddef>
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>

template <typename T>
concept Number = std::integral<T> || std::floating_point<T>;

template <Number T>
T spread(const std::vector<T>& v) {
    const auto [lo, hi] = std::minmax_element(v.begin(), v.end());
    return *hi - *lo;
}

template <std::integral T>
void stats(const std::vector<T>& v) {
    long long sum = 0;
    for (T x : v) sum += x;
    const auto [lo, hi] = std::minmax_element(v.begin(), v.end());
    std::cout << "integral n=" << v.size() << " sum=" << sum << " min=" << *lo << "
              << " spread=" << spread(v) << '\n';
}

template <std::floating_point T>
void stats(const std::vector<T>& v) {
    double sum = 0.0;
    for (T x : v) sum += x;
    const auto [lo, hi] = std::minmax_element(v.begin(), v.end());
    std::cout << std::fixed << std::setprecision(2) << "floating n=" << v.size()
              << " mean=" << sum / static_cast<double>(v.size()) << " min=" << *lo <
              << " spread=" << spread(v) << '\n';
}

template <typename T>
void stats(const std::vector<T>& v) {
    std::size_t longestAt = 0;
    std::size_t chars = 0;
    for (std::size_t i = 0; i < v.size(); ++i) {
        chars += v[i].size();
        if (v[longestAt].size() < v[i].size()) longestAt = i;
    }
    std::cout << "text n=" << v.size() << " longest=" << v[longestAt] << " chars=" <
}

template <typename T>
void handle() {
```

```
    std::size_t k;
    std::cin >> k;
    std::vector<T> v(k);
    for (T& x : v) std::cin >> x;
    stats(v);
}

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string type;
        std::cin >> type;
        if (type == "int") handle<int>();
        else if (type == "double") handle<double>();
        else handle<std::string>();
    }
    return 0;
}
```

EXERCISE 3

Table<K, V> with a TypeName header

Write a trait `TypeName<T>` — a primary template whose `value` is `"unknown"` and full specialisations giving `"int"`, `"double"` and `"std::string"` — plus a variable template `type_name_v<T>`. Then a class template `Table<K, V>` over a `std::map<K, V>` with `set(key, value)` (a repeated key replaces the earlier value) and `print()`, which writes the header `Table<<K name>, <V name>>` from the trait, then every row in key order as `<key> -> <value>` (a `double` value with two decimals — use `if constexpr` on `std::is_floating_point_v<V>`), then `rows=<count>`.

The first line names `K` (`int` or `string`) and `V` (`int`, `double` or `string`), so there are six possible instantiations; dispatch through a `run<K, V>()` function template. Then `n`, then `n` lines of `<key> <value>`. Note that integer keys order numerically while string keys order lexicographically — the same input prints differently under `Table<int, …>` and `Table<std::string, …>`.

Input: the two type words, `n`, then `n` rows. **Output:** the header, the rows in key order, then `rows=<count>`.

```
int string
3
10 ten
2 two
7 seven
```

prints

```
Table<int, std::string>
2 → two
7 → seven
10 → ten
rows=3
```

INPUT 1

```
int string
3
10 ten
2 two
7 seven
```

OUTPUT 1

```
Table<int, std::string>
2 → two
7 → seven
10 → ten
rows=3
```

INPUT 2

```
string double
3
pear 1.5
apple 0.25
pear 2
```

OUTPUT 2

```
Table<std::string, double>
apple → 0.25
pear → 2.00
rows=2
```

REFERENCE SOLUTION

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

template <typename T> struct TypeName { static constexpr const char* value = "unknow
template <> struct TypeName<int> { static constexpr const char* value = "int"; };
template <> struct TypeName<double> { static constexpr const char* value = "double";
template <> struct TypeName<std::string> { static constexpr const char* value = "sta

template <typename T>
constexpr const char* type_name_v = TypeName<T>::value;

template <typename K, typename V>
class Table {
public:
    void set(const K& key, const V& value) { rows_[key] = value; }
    void print() const {
        std::cout << "Table<" << type_name_v<K> << ", " << type_name_v<V> << ">\n";
        for (const auto& [key, value] : rows_) {
            std::cout << key << " -> ";
            if constexpr (std::is_floating_point_v<V>) std::cout << std::fixed << st
            else std::cout << value;
            std::cout << '\n';
        }
        std::cout << "rows=" << rows_.size() << '\n';
    }

private:
    std::map<K, V> rows_;
};

template <typename K, typename V>
void run() {
    Table<K, V> table;
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        K key;
        V value;
        std::cin >> key >> value;
        table.set(key, value);
    }
    table.print();
}

template <typename K>
void run_with_value(const std::string& v) {
```

```
    if (v == "int") run<K, int>();
    else if (v == "double") run<K, double>();
    else run<K, std::string>();
}

int main() {
    std::string k, v;
    std::cin >> k >> v;
    if (k == "int") run_with_value<int>(v);
    else run_with_value<std::string>(v);
    return 0;
}
```

CHECKPOINT QUESTIONS

1. Which call fixes `max_of(3, 2.5)` for `template <typename T> T max_of(const T&, const T&)`?

A. `max_of(3, (int)2.5)` — and it returns `3`

B. `max_of<double>(3, 2.5)` — the `3` converts once `T` is named

C. `max_of(static_cast<auto>(3), 2.5)`

D. Nothing; a template cannot take two arithmetic types

2. A template is declared in a header and defined in one `.cpp` file. Which error results when another file uses it?

A. A compile error in the header

B. A linker error: undefined reference to the instantiation

C. A run-time crash

D. No error — the linker instantiates it

3. `void f(int);` and `template <typename T> void f(const T&);` both exist. What do `f(5)` and `f('a')` call?

A. Both call the non-template

B. Both call the template

C. `f(5)` the non-template (an equal-match tie goes to it); `f('a')` the template (`T = char` is exact, `char` to `int` is a promotion)

D. `f(5)` the template; `f('a')` the non-template

4. What does this print?

```
std::vector a{3};
std::vector b(3, 0);
std::cout << a.size() << b.size();
```

A. `33`

B. `13`

C. `31`

D. Compile error

5. Inside `template <typename C> void g(const C& c)`, which line compiles?

A. `C::value_type x = *c.begin();`

B. `typename C::value_type x = *c.begin();`

C. `template C::value_type x = *c.begin();`

D. `auto C::value_type x = *c.begin();`

6. Which statement about a full specialisation `template <> struct Box<bool>` is correct?

A. It inherits every member of the primary `Box<T>` and may override some

B. It is a separate class: it has only the members it declares

C. It must declare the same members as the primary or the compiler rejects it

D. It applies to `bool` and to every type convertible to `bool`

7. You want `show(const char*)` to print differently from the template `template <typename T> void show(const T&)`. What should you write?

A. `template <> void show<const char*>(const char* const&)`

B. A non-template overload `void show(const char* v)`

C. `template <typename T> void show<T*>(T*)`

D. A `static_assert` in the template

8. `template <typename T, std::size_t N> class Ring` — which declaration compiles?

A. `int n = 4; Ring<int, n> r;`

B. `const int n = read(); Ring<int, n> r;`

C. `constexpr std::size_t n = 4; Ring<int, n> r;`

D. `std::size_t n; std::cin >> n; Ring<int, n> r;`

9. What does `sum()` return for `template <typename... Ts> auto sum(Ts... xs) { return (0 + ... + xs); }` — and what would `(xs + ...)` do instead?

A. `0`; `(xs + ...)` would also give `0`

B. `0`; `(xs + ...)` would be a compile error on an empty pack

C. A compile error in both cases

D. `0`; `(xs + ...)` would give an unspecified value

10. What does this print?

```
template <std::integral T>       void kind(T) { std::cout << "integral"; }
template <std::floating_point T> void kind(T) { std::cout << "floating"; }
template <typename T>            void kind(T) { std::cout << "other"; }

int main() { kind('x'); kind(1.0f); kind("s"); }
```

A. `otherfloatingother`

B. `integralfloatingother`

C. `integralotherother`

D. Compile error: ambiguous call for `'x'`

11. Overload A is constrained by `std::integral<T>`; overload B by `std::signed_integral<T>`. Overload C is constrained by an inline `requires(T a) { a + a; }` and overload D by an identical inline expression. Which pair resolves cleanly for `int`?

A. A and B — B subsumes A through the named concept and wins

B. C and D — identical text makes them equivalent

C. Neither pair; two constrained overloads are always ambiguous

D. Both pairs resolve to the first declared

12. Why does `std::is_same_v<T, std::string>` fail to detect a `std::string` lvalue passed to `template <typename T> void f(T&& x)`, and what fixes it?

A. `T` deduces as `std::string&`; test `std::remove_cvref_t<T>` instead

B. `T` deduces as `const std::string`; test `std::remove_const_t<T>`

C. `is_same_v` cannot compare class types; use `typeid`

D. `T&&` only accepts rvalues, so the call does not compile

13. Which one should compute a *value* at compile time in C++20, and which a *type*?

A. Values with recursive class templates; types with `constexpr` functions

B. Values with `constexpr` functions; types with traits such as `std::conditional_t`

C. Both with macros

D. Both with `if constexpr`

ANSWER KEY · CHECKPOINT — TEMPLATES AND GENERIC PROGRAMMING

1. B Naming `T` skips deduction, and `3` then converts to `double` like any function argument. The first option compiles too but throws away the `.5` — it answers a different question. `static_cast<auto>` is not C++.

2. B Each translation unit generates only the instantiations it uses, and the using file has only a declaration in view, so nothing generates the function and the linker finds no symbol. Templates live in headers, definition included.

3. C Resolution first ranks the conversions. `f(5)` is exact for both, and a tie goes to the non-template. `f('a')` is exact only for the template — `const char&` — while the non-template needs a promotion, so the template wins.

4. B Braces choose the `initializer_list` constructor — one element — while parentheses choose the count-and-value constructor — three zeros. CTAD deduces `std::vector<int>` for both; it is the initialiser syntax that differs.

5. B `C::value_type` is a dependent name and the parser assumes it is not a type unless `typename` says so. `auto x = *c.begin();` would also work and is what most modern code writes; the other spellings are syntax errors.

6. B An explicit specialisation replaces the primary for exactly that argument and inherits nothing from it. Keeping the interface identical is your job; the compiler enforces nothing. To change only one member, specialise that member alone.

7. B A non-template overload takes part in overload resolution and wins an exact-match tie. A full specialisation is not a candidate — a better-matching primary added later would hide it — and function templates cannot be partially specialised at all.

8. C A non-type argument must be a constant expression. A `constexpr` variable qualifies; a non-const `int`, a `const` initialised from a function call, and a value read at run time do not. Run-time sizes dispatch to a few fixed instantiations or use `std::vector`.

9. B A binary fold supplies the starting value, which is what makes it legal for zero arguments — `(0 + ... + xs)` is just `0`. A unary `+` fold over an empty pack has nothing to produce and is rejected; only `&&`, `||` and `,` may fold an empty pack unaided.

10. B `char` is `std::integral`, `float` is `std::floating_point`, and a string literal decays to `const char*`, which is neither. In each call the constrained overload that is satisfied beats the unconstrained one; no two constrained overloads are satisfied at once, so nothing is ambiguous.

11. A Subsumption is proved through named concepts: `signed_integral` is defined as `integral && is_signed_v`, so B is more constrained. Two inline `requires` expressions are different atoms even when identical, so C and D are ambiguous. Name your concepts.

12. A A forwarding reference deduces `T` as an lvalue reference for an lvalue argument, and `std::string&` is not `std::string`. `std::remove_cvref_t<T>` strips the reference and any `const` / `volatile`, giving the plain type to test. The same applies to `auto&&` in a range-for.

13. B A `constexpr` function is ordinary C++ — loops, locals, even `std::vector` in C++20 — evaluated at compile time when its arguments are constants; recursive template arithmetic instantiates a class per step and cannot loop. Types cannot be returned from functions, so traits and `conditional_t` do that job.
