Modern C++ Cheat Sheet

1. Common Data Types

Type	#include	Description
int	built-in	whole numbers
double	built-in	floating-point numbers
float	built-in	smaller floating-point (suffix f)
char	built-in	single character (numeric type, 'A' = 65)
bool	built-in	true / false
long	built-in	at least as large as int (suffix l)
long long	built-in	larger than long (suffix ll)
unsigned int	built-in	non-negative whole numbers (suffix u)
size_t	<cstddef>	unsigned type for sizes, returned by .size()
std::string	<string>	dynamic character string
std::string_view	<string_view>	non-owning read-only view of a string
auto	keyword	compiler deduces the type

Traps: auto x = "hello" gives const char*, not std::string – use "hello"s suffix with using namespace std::string_literals. Brace initialization int x{3.5} prevents narrowing conversions (compile error).

Idioms: prefer brace initialization {}; use const when the value won't change; use auto when the type is obvious or complex.

2. Operators

Precedence Table

Prec	Operator	Description	Assoc
1	::	scope resolution	L-R
2	() [] . -> ++ --	call, subscript, member access, postfix inc/dec	L-R
3	++ -- + - ! ~ * &	prefix inc/dec, unary, logical/bitwise NOT, deref, address-of	R-L
5	* / %	multiply, divide, modulus	L-R
6	+ -	add, subtract	L-R
7	<< >>	bitwise shift	L-R
8	< <= > >=	relational	L-R
9	== !=	equality	L-R
10	&	bitwise AND	L-R
11	^	bitwise XOR	L-R
12	|	bitwise OR	L-R
13	&&	logical AND (short-circuits)	L-R
14	||	logical OR (short-circuits)	L-R
15	?: = += -= *= /= %=	ternary, assignment, compound assignment	R-L
16	,	comma	L-R

Quick Reference

Operator	Example	Notes
%	10 % 3 → 1	integers only; remainder after division

Operator	Example	Notes
`++x` / `x++`	prefix returns new value; postfix returns old	prefer prefix in loops
`+= -= *= /= %=`	`x += 5` ≡ `x = x + 5`	compound assignment
`?:`	`cond ? a : b`	ternary; avoid nesting
`sizeof(T)`	`sizeof(int)` → 4 (typical)	size in bytes; compile-time
`static_cast<T>(x)`	`static_cast<double>(n)`	safe C++ cast (e.g. int to double)

Traps: `=` vs `==` in conditions (`if (x = 5)` assigns, always true). `!` binds tighter than `>`, so `!score > 80` is `(!score) > 80`. Bitwise operators have lower precedence than `==`: `x & 0xFF == 0` parses as `x & (0xFF == 0)`. Don't confuse `&&`/`||` (logical) with `&`/`|` (bitwise).

Idioms: use parentheses when precedence is not obvious; integer division truncates (`10/3` → `3`).

3. Formatting

`#include <print>` for `std::println` / `std::print`; `#include <format>` for `std::format`

Signature	Description
`std::println("{}", val)`	print with newline
`std::print("{}", val)`	print without newline
`std::format("{}", val)`	returns `std::string`
`std::println(file, "{}", val)`	print to an output file stream

Format Specification: `{index:fill align width .precision type}`

Spec	Example	Result
fixed 2 decimals	`{:.2f}`	`3.14`
right-justify, width 10	`{:>10}`	`·····hello`
left-justify, width 10	`{:<10}`	`hello·····`
center, width 10	`{:^10}`	`··hello···`
fill with `*`, right, width 6	`{:*>6}`	`***abc`
explicit arg index	`{0:} {1:}`	args by index
dynamic width	`{:>{}}`	2nd arg = width
decimal (default)	`{:d}`	`42`
hexadecimal	`{:x}` / `{:#x}`	`2a` / `0x2a`
octal	`{:o}` / `{:#o}`	`52` / `052`
binary	`{:b}` / `{:#b}`	`101010` / `0b101010`
always show sign	`{:+}`	`+42` or `-42`
space for positive	`{: }`	`·42` or `-42`

Traps: literal braces require doubling: `format("{{x}}")` produces `{x}`. `std::cout` prints `bool` as `0`/`1`; `std::println` prints `true`/`false`. Too few format args is a compile error; too many silently ignores extras. Displayed values are rounded, not truncated.

Idioms: prefer `std::format` over `+` for building strings; prefer `std::println` over `std::cout`.

4. Functions

```
int sum(int a, int b);                  // returns int by value
void foo(int x);                        // by value (copies)
void foo(const std::string & s);        // by const ref (no copy, read-only)
void foo(std::string & s);              // by ref (modifiable)
```

```
double avg(const std::vector<double> & v);  // returns double
std::string greet(std::string_view name);   // returns string
```

Pattern	When to Use
pass by value	fundamental types (`int`, `double`, `char`, `bool`), `string_view`
pass by `const &`	large objects you don't need to modify (`string`, `vector`)
pass by `&`	objects you need to modify, streams
return by value	the default; compiler optimizes away copies
return `void`	function has no return value

Traps: default parameters must be at the end; once you default one, all that follow must also be defaulted. A function in a header without `inline` breaks ODR if included in multiple files.

Idioms: pass small types by value, large types by `const &`; if the function needs its own copy, take by value to get an automatic copy.

5. iostreams

`#include <iostream>` for `cin`/`cout`/`cerr`; `#include <fstream>` for file streams; `#include <sstream>` for string streams

Stream	Description
`std::cin`	standard input (keyboard)
`std::cout`	standard output (buffered)
`std::cerr`	standard error (unbuffered)

Signature	Description
`std::cin >> var`	extract from standard input
`std::cout << val << '\n'`	insert to standard output
`std::getline(stream, str)`	read entire line into `string`
`std::getline(stream, str, delim)`	read until `delim` into `string`
`if (stream)`	check stream is in good state
`stream.clear()`	clear error flags
`stream.ignore(max, '\n')`	discard remaining input (see below)
`std::ifstream file{name}`	open for reading
`std::ofstream file{name}`	open for writing (truncates!)
`std::ofstream file{name, ios::app}`	open for appending
`std::istringstream iss{str}`	input from a string (useful for testing)
`std::ostringstream oss{}`	build a string via `<<`, extract with `oss.str()`

File Mode	Description
`std::ios::in`	input (default for `ifstream`)
`std::ios::out`	output (default for `ofstream`)
`std::ios::app`	append
`std::ios::binary`	binary mode

Full ignore call: `stream.ignore(std::numeric_limits<std::streamsize>::max(), '\n')` (needs `<limits>`)

Traps: `>>` leaves the newline in the buffer; a following `getline` reads an empty string. Fix: call `ignore()` between `>>` and `getline`, or use `std::getline(std::cin >> std::ws, str)`. `while (!stream.eof())` is almost always wrong – `eof()` is only set after a failed read, causing the last item to be processed twice; use `while (stream >>`

`var)` or `while (std::getline(stream, line))` instead. Non-numeric input puts `cin` in a fail state; must `clear()` then `ignore()` to recover. Default `ofstream` truncates the file. Always check `if (file)` after opening.

Idioms: write functions taking `std::istream &` / `std::ostream &` so they work with cin/cout, files, and stringstreams. Files close automatically via RAII when the stream goes out of scope.

6. Strings and string_view

`#include <string>`; `#include <string_view>`

std::string

Signature	Description
`std::string s{"abc"}`	construct from literal
`std::string s(3, 'A')`	`"AAA"` – n copies of a char
`s.find(ch)`	`size_t` position, or `npos` if not found
`std::string::npos`	`size_t` max value – "not found" sentinel
`s.substr(pos, count)`	returns `string` (copies)
`s.size()` / `s.length()`	`size_t` number of characters
`s.empty()`	`bool` – true if empty
`s.front()` / `s.back()`	`char &` – first / last character
`s.push_back(ch)`	append a character
`s[i]`	access by index (no bounds check)
`s + other`	concatenation (creates temporaries)
`s += str` / `s.append(str)`	append in place (no temporaries)
`s.insert(pos, str)`	insert string at position
`s.erase(pos, count)`	remove `count` chars starting at `pos`
`s.replace(pos, count, str)`	replace substring
`s.c_str()`	`const char *` for C interop
`s.starts_with(x)`	`bool` (C++20)
`s.ends_with(x)`	`bool` (C++20)
`s.contains(x)`	`bool` (C++23)

Conversions

Signature	Description
`std::stoi(str)`	`int` from string (throws on failure)
`std::stol(str)`	`long` from string
`std::stoll(str)`	`long long` from string
`std::stoul(str)`	`unsigned long` from string
`std::stoull(str)`	`unsigned long long` from string
`std::stof(str)`	`float` from string
`std::stod(str)`	`double` from string
`std::stold(str)`	`long double` from string
`std::to_string(num)`	`string` from numeric value

std::string_view

Signature	Description
`std::string_view sv{str}`	non-owning view of a string
`sv.substr(pos, count)`	returns `string_view` (no copy)

Traps: all `sto*` functions throw `std::invalid_argument` if the string is not a number and `std::out_of_range` if the value is too large. Modifying the underlying string invalidates any `string_view` of it (dangling, undefined behavior). String `+` concatenation creates temporaries; prefer `std::format`.

Idioms: use `std::string_view` by value for read-only string parameters; use `const std::string &` when you need a guaranteed `std::string`. Use `"hello"s` suffix (with `using namespace std::string_literals`) to get a `std::string` from a literal.

7. Arrays and Vectors

`#include <array>`; `#include <vector>`

std::array – fixed size

Signature	Description
`std::array<double, 5> a{}`	5 doubles, all 0.0
`std::array a{1, 2, 3}`	CTAD deduces type and size
`a[i]` / `a.at(i)`	`T &` – access (at throws on bad index)
`a.size()`	`size_t` number of elements
`a.begin()` / `a.end()`	iterator to first / past-end

std::vector – dynamic size

Signature	Description
`std::vector<int> v{}`	empty vector
`std::vector v{1, 2, 3}`	CTAD
`std::vector<int> v(10, 0)`	10 elements, all 0
`v.push_back(val)`	add to end (may reallocate)
`v.emplace_back(args...)`	construct in place at end
`v.insert(it, val)`	insert at position
`v.erase(it)`	erase single element
`v.erase(first, last)`	erase range
`v[i]` / `v.at(i)`	`T &` – access (at throws on bad index)
`v.size()`	`size_t` number of elements
`v.empty()`	`bool` – true if no elements
`v.front()` / `v.back()`	`T &` – first / last element
`v.clear()`	remove all elements (size becomes 0)
`v.resize(n)`	change element count (adds default or removes)
`v.pop_back()`	remove last element
`v.reserve(n)`	pre-allocate space (no new elements)
`v.capacity()`	`size_t` current allocated capacity
`v.data()`	`T *` pointer to underlying array (C interop)

std::span – non-owning view of contiguous data (C++20)

`#include <span>`

Signature	Description
`std::span<int> s{arr}`	view of array or vector
`std::span<int, 5> s{arr}`	fixed-extent (size known at compile time)
`s.size()` / `s[i]`	`size_t` count / element access
`s.subspan(offset, count)`	`span` – sub-view

Range-based for:

```
for (const auto & elem : container) { ... }   // read-only
for (auto & elem : container) { ... }         // modifiable
```

Traps: `vector<int>{2, 5}` has two elements (2 and 5); `vector<int>(2, 5)` has two elements both equal to 5 – braces vs parentheses matter. `push_back` may reallocate and invalidate all iterators. `[]` does not check bounds.

Idioms: use `vector` as the default container; use `array` when size is known at compile time; call `reserve()` before many `push_back` calls.

8. Algorithms

`#include <algorithm>`; `#include <numeric>` for `accumulate`

Range Algorithms (C++20, `std::ranges::`)

Signature	Description
`ranges::sort(cont)`	`void` – sort in place
`ranges::sort(cont, comp)`	sort with comparator (e.g. `greater<>{}`)
`ranges::sort(cont, {}, &T::mem)`	sort by member (projection)
`ranges::find(cont, val)`	iterator to first equal element, or `end()`
`ranges::find_if(cont, pred)`	iterator to first match, or `end()`
`ranges::count_if(cont, pred)`	`size_t` count of matching elements
`ranges::minmax(cont)`	`{.min, .max}` struct
`ranges::any_of(cont, pred)`	`bool` – true if any element matches
`ranges::all_of(cont, pred)`	`bool` – true if all match
`ranges::none_of(cont, pred)`	`bool` – true if none match
`ranges::for_each(cont, fn)`	apply `fn` to each element
`ranges::transform(in, out, fn)`	apply `fn`, write to `out` iterator
`ranges::copy(cont, out)`	copy elements to `out` iterator
`ranges::copy_if(cont, out, pred)`	copy matching elements
`ranges::reverse(cont)`	`void` – reverse in place
`ranges::max_element(cont)`	iterator to largest element
`ranges::min_element(cont)`	iterator to smallest element
`ranges::generate(begin, end, fn)`	`void` – fill range from a callable
`ranges::generate_n(inserter, n, fn)`	iterator past last generated
`ranges::shuffle(cont, gen)`	`void` – random reorder using engine

`std::back_inserter(container)` (`<iterator>`) – output iterator that calls `push_back`; used with `copy`, `copy_if`, `transform`, `generate_n`.

Classic Algorithms

Signature	Description
`std::count_if(begin, end, pred)`	`size_t` count of matching
`std::accumulate(begin, end, init)`	`T` sum of elements (`<numeric>`)
`std::iota(begin, end, start)`	fill with `start`, `start+1`, … (`<numeric>`)
`std::remove_if(begin, end, pred)`	iterator to new logical end

Erase (C++20)

Signature	Description
`std::erase_if(container, pred)`	`size_t` count of erased elements

Common Predicates and Comparators

`#include <functional>` for comparators; `#include <cctype>` for character classification

Predicate	Description
`std::less<>{}`	`a < b` (default for `sort`)
`std::greater<>{}`	`a > b` (sort descending)
`std::equal_to<>{}`	`a == b`
`std::isalpha(ch)`	`bool` – letter (`<cctype>`)
`std::isdigit(ch)`	`bool` – digit (`<cctype>`)
`std::isspace(ch)`	`bool` – whitespace (`<cctype>`)
`std::isupper(ch)` / `std::islower(ch)`	`bool` – case check (`<cctype>`)

`std::toupper(ch)` / `std::tolower(ch)` (`<cctype>`) convert case; commonly used with `ranges::transform`.

Traps: `std::ranges::minmax` on an empty range is undefined behavior. `std::remove_if` does NOT resize the container; must follow with `container.erase(new_end, container.end())`. `std::accumulate` is in `<numeric>`, not `<algorithm>`. `<cctype>` functions take `int`, not `char`; passing a negative `char` is undefined behavior – cast to `unsigned char` first.

Idioms: prefer `std::erase_if` (C++20) over the erase-remove idiom; prefer `std::ranges::` algorithms over iterator-pair versions; use `std::back_inserter(container)` with generate algorithms.

9. Lambdas

Syntax: `[captures](parameters) { body }`

Capture	Meaning
`[]`	nothing
`[x]`	`x` by value (copy)
`[&x]`	`x` by reference
`[x, &y]`	mix
`[=]`	all used variables by value
`[&]`	all used variables by reference
`[this]`	current object's `this` pointer
`[p = std::move(ptr)]`	init-capture: move into the lambda

Adding `mutable` after `()` allows modifying by-value captures (the copy, not the original).

Generic lambdas use `auto` parameters: `[](auto x, auto y) { return x + y; }`.

Trailing return type: `[](int x) -> double { return x / 2.0; }` – needed when return type is ambiguous.

`std::function<ReturnType(Params)>` (`<functional>`) stores any callable with a matching signature.

Traps: each lambda has a unique type; use `auto` to store one. Capturing by reference: if the referenced variable goes out of scope, the lambda has a dangling reference. `[=]` can accidentally capture unintended variables.

Idioms: capture explicitly by name rather than `[=]` or `[&]`. Use named functions for anything longer than a few lines. Use lambdas as algorithm predicates:

```
std::erase_if(v, [](double x) { return x < 0.0; });
```

10. Ranges

`#include <ranges>`

View	Description
`std::views::filter(range, pred)`	keep elements matching predicate
`std::views::transform(range, fn)`	apply `fn` to each element
`std::views::take(range, n)`	first n elements
`std::views::take_while(range, pred)`	take while predicate is true
`std::views::drop(range, n)`	skip first n elements
`std::views::drop_while(range, pred)`	skip while predicate is true
`std::views::reverse`	reverse the range
`std::views::iota(start)`	infinite sequence: `start`, `start+1`, ...
`std::views::iota(start, end)`	bounded sequence: `start` to `end-1`
`std::views::enumerate(range)`	`{index, value}` pairs (C++23)

Pipe Syntax

```
auto result = data | std::views::filter(pred) | std::views::take(5);
```

Converting to a Container

```
auto vec = std::ranges::to<std::vector>(view);          // C++23
std::vector<int> vec{view.begin(), view.end()};         // fallback
```

Traps: views are lazy – predicates are not called until the view is iterated. Views do not own data; if the underlying data goes out of scope, the view is invalid.

Idioms: use pipe `|` to chain views for readability; use views to avoid copying large datasets.

11. Random Numbers

```
#include <random>
```

Three components: seed, engine, distribution.

Signature	Description
`random_device rd{}`	non-deterministic seed source
`default_random_engine gen(rd())`	engine seeded randomly
`default_random_engine gen{42}`	fixed seed (reproducible)
`uniform_int_distribution dist{1, 6}`	integers in [1, 6]
`uniform_real_distribution dist{0.0, 1.0}`	doubles in [0.0, 1.0)
`normal_distribution dist{mean, stddev}`	normal/bell curve
`bernoulli_distribution dist{0.5}`	`bool` with given probability
`int roll = dist(gen)`	generate a number

All types above are in `std::`.

Traps: same seed → same sequence. Without `std::random_device`, you get the same numbers every run. Normal distribution can produce negative values.

Idioms: use `std::random_device` for varying seeds; use a fixed seed for reproducible testing; separate generation from usage by passing a `std::function<double()>` or lambda.

12. Structures and Classes

struct (all public by default)

```
struct Point {
    double x{};
    double y{};
};
Point p{3.0, 4.0};            // aggregate initialization
Point q{.x = 3.0, .y = 4.0}; // designated initializers (C++20)
auto [x, y] = p;              // structured bindings (C++17)
```

class (all private by default)

```
class Stock {
    std::string name{};
    double price{};
public:
    Stock(const std::string & name, double price)
        : name(name), price(price) {}   // member initializer list

    std::string get_name() const { return name; }  // const member function
    void set_price(double p) { price = p; }
    ~Stock() = default;
};
```

Concept	Notes
`public` / `private` / `protected`	access specifiers
member initializer list	`: name(n), price(p) {}` – preferred over body assignment
`const` member function	can be called on `const` objects; promises not to modify state
`explicit` constructor	prevents implicit conversions from single-parameter constructors
aggregate initialization	works for structs with all public members and no constructors
`static` member	shared by all instances; accessed via `Class::member`
`friend` function	non-member that can access private members
`bool operator==(const T &) const = default`	compiler-generated equality (C++20)
`auto operator<=>(const T &) const = default`	generates all 6 comparisons (C++20)

enum class

```
enum class Color { red, green, blue };
Color c = Color::red;
```

Scoped, no implicit conversion to `int`. Use `static_cast<int>(c)` to convert.

operator« overload

```
friend std::ostream & operator<<(std::ostream & os, const Point & p) {
    return os << "(" << p.x << ", " << p.y << ")";
}
```

Define as `friend` inside the class to access private members.

Traps: providing any constructor removes the implicit default constructor. Calling a non-const member function on a `const` object is a compile error. Assigning in the constructor body initializes members twice. Unscoped `enum` values leak into the enclosing scope.

Idioms: prefer member initializer lists; mark getters `const`; mark single-parameter constructors `explicit`; use `.` for direct access, `->` through a pointer. Prefer `enum class` over `enum`. Use `= default` for `operator==` when memberwise comparison is correct.

13. unique_ptr and Move

`#include <memory>`; `#include <utility>` for `std::move`

std::unique_ptr

Signature	Description
`auto p = std::make_unique<T>(args...)`	create on heap
`p->member`	access member
`*p`	dereference
`p.get()`	`T *` – raw pointer without releasing ownership
`if (p)`	check not nullptr
`p.reset()`	release and set to nullptr
`p.release()`	`T *` – relinquish ownership (caller must delete)
`auto q = std::move(p)`	transfer ownership (`p` becomes nullptr)

Cannot be copied. Only one `unique_ptr` owns the object at a time.

Move Semantics

- `std::move(obj)` casts to an rvalue reference, transferring ownership
- moved-from object is in a valid but unspecified state

Traps: `std::move` on a `const` object does nothing. `return std::move(local);` pessimizes by preventing NRVO – just write `return local;`. A class containing a `unique_ptr` member cannot be copied. `release()` gives you a raw pointer you must manually `delete`.

Idioms: always use `std::make_unique` – never raw `new`/`delete`.

14. Exceptions and Expectations

`#include <exception>`; `#include <stdexcept>`; `#include <expected>` for `std::expected`

Throwing and Catching

```
try {
    throw std::invalid_argument("negative value");
}
catch (const std::invalid_argument & ex) {
    std::println("{}", ex.what());
}
catch (const std::exception & ex) {
    std::println("error: {}", ex.what());
}
```

Common Exception Types

Type	Header	Typical Use
`exception`	`<exception>`	base type; generic catch-all
`invalid_argument`	`<stdexcept>`	bad parameter values
`runtime_error`	`<stdexcept>`	errors detected at runtime (e.g. file I/O)
`out_of_range`	`<stdexcept>`	index/value out of bounds

Type	Header	Typical Use
`bad_expected_access`	`<expected>`	`.value()` on error `expected`
`bad_optional_access`	`<optional>`	`.value()` on empty `optional`

All types above are in `std::`.

Custom exceptions: derive from `std::exception` and override `what()` if needed.

noexcept

Mark functions that will never throw: `void f() noexcept;`. Enables compiler optimizations and signals intent to callers.

std::expected (C++23)

Returns a value or an error without exceptions.

Signature	Description
`std::expected<T, E>`	holds `T` on success or `E` on error
`return value;`	return a success value
`return std::unexpected{err};`	return an error
`result.has_value()` / `if (result)`	`bool` – check for success
`result.value()`	`T` – access value (throws if error)
`result.error()`	`E` – access error

Traps: catch handlers are tried in order – put specific types (`invalid_argument`) before general types (`exception`), or the specific handler is never reached. An unhandled exception terminates the program. Calling `.value()` on an error `expected` throws `bad_expected_access`. You can throw any type (even `int`), but doing so is unconventional – always throw types derived from `std::exception`.

Idioms: always catch by `const &`. Wrap the body of `main` in try/catch to prevent exceptions from leaking to the user. Don't use exceptions for flow control. Reserve exceptions for truly exceptional conditions – for expected errors (e.g. bad user input), prefer `std::expected` or return a `bool`. Check conditions before throwing when possible (e.g. `std::filesystem::exists()` before opening a file).