C++ Quick Reference Sheet

Topic	Note / Best Practice
Value vs Reference	Pass by reference to avoid copying unless you need a copy.
Pointers vs References	Use references for aliasing, pointers for optional/nullable semantics.
`auto`	Deduces type; use `typeid()` to confirm.
Const Syntax	`const int*` (value const), `int* const` (pointer const), `const int* const` (both).
Smart Pointers	`unique_ptr` = sole ownership, `shared_ptr` = shared ownership.
Rule of 0/3/5	Use Rule of 3/5 if managing resources; else Rule of 0 + RAII.
Virtual & Override	`virtual` in base, `override` in derived — enables runtime dispatch.
Slicing & Polymorphism	Avoid storing derived objects by value — use refs/pointers.
`remove_if` + erase	`remove_if` moves, `erase` removes. Always pair them.
Containers (vector/list/map)	`vector` = fast access, `list` = insert/delete, `map` = sorted, `unordered_map` = faster but unordered.
`emplace` vs `insert`	`emplace` constructs in-place, avoids extra copies.
`explicit` keyword	Prevents implicit conversions on single-arg constructors.
Iterators & Ranges	`begin()` / `end()` standard; use `for (auto& x : c)` for clarity.
Exception Safety	Destructors run in reverse. Prefer RAII. Use `noexcept` carefully.
`= default` / `= delete`	`= default` for boilerplate, `= delete` to block copies.
Lambdas	`[=]` = capture by value, `[&]` = by ref. Use `->` for return type.
`std::transform`	Transforms a container in-place using a lambda.
Unit Testing	`assert()` for simple checks. No built-in test suite.
Logging	Use `cerr` or custom logger with timestamps.

Topic	Note / Best Practice
`pair` / `tuple`	`pair` = 2 values, `tuple` = more. Use `.first`, `.second`, `get<>`.
`typeid().name()`	Returns stringified type name — good with `auto`.
Stack vs Heap	`int x = 5;` (stack), `new int(5);` (heap). Prefer RAII.
Undefined Behavior	Common: bad iterators, double delete, out-of-bounds.
`std::sort` + lambda	`std::sort(v.begin(), v.end(), [](a,b){ return a < b; });`
`noexcept`	Declares no exceptions — helps correctness & optimizations.
`std::optional`	Wraps a value that might be missing — safer than raw pointers.
`std::filesystem`	Use `path`, `exists()`, `is_directory()`, `directory_iterator`.
`std::chrono`	Use `steady_clock::now()`, `duration_cast<>`, `milliseconds` for timing.
`std::regex`	Regex-based matching with `regex_match`, `regex_search`, `regex_replace`.
`std::array` vs `vector`	`array<T,N>` = fixed-size stack, `vector<T>` = dynamic heap.
`std::stack` / `queue`	`stack = LIFO`, `queue = FIFO`; use `.push()`, `.pop()`, `.top()`.
Iterator invalidation	`erase()` invalidates iterators; capture returned one to continue.
`std::for_each`	Applies lambda/function to each element — alternative to loops.
`decltype`	Yields declared type of an expression — complements `auto`.
`constexpr`	Compile-time constant — better than `#define` for typed constants.