Topic	Note / Best Practice
Compile	g++ <filename> -o <output_name> Only cpp files require compiling. Use -c flag in front of filename to compile to object file. Call again on object files without -c to link into an executable
Header files	#ifndef PROJECT_HEADER_NAME #define PROJECT_HEADER_NAME inline functions, function definitions etc, no namespaces #endif
Value vs Reference	Pass by reference to avoid copying unless you need a copy
auto	Deduces type; use typeid(<var>) to confirm and compare. For debugging, print type to console using typeid(var).name()
decltype	Yields declared type of an expression — complements auto
constexpr	Compile-time constant — better than #define for typed constants
Type Promotion	When types differ (usually in arithmetic), smaller, less precise types are promoted int + double → result is double; float + long → result is float; auto z = 'A' → promoted to int;
size() and sizeof()	size() → number of elements in a container sizeof() → size of type or object in bytes
Stack vs Heap	int x = 5; (stack), new int(5); (heap). Prefer RAII
Pointers vs References	Use references for aliasing, pointers for optional/nullable semantics
Const Syntax	const int* (value const), int* const (pointer const), const int* const (both)
Smart Pointers	#include <memory> std::unique_ptr = sole ownership, shared_ptr = shared ownership. Use std::make_<type> to create, std::move(<ptr>) for moving unique
Iterators & Ranges	begin()/end() standard; use for (auto& x : c) for clarity
Iterator invalidation	erase() invalidates iterators; capture returned one to continue.
Containers (vector/list/map)	vector = fast access, list = fast insert/delete, no indexing, map = sorted, unordered_map = faster but unordered
pair / tuple	pair = 2 values, tuple = more. Use .first, .second, get<>
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()
std::sort + lambda	std::sort(v.begin(), v.end(), [](a,b){ return a < b; });
emplace vs insert	emplace constructs in-place, avoids extra copies.

Topic	Note / Best Practice
std::transform	Applies a function to a container in-place
std::for_each	Applies lambda/function to each element — alternative to loops
Mathematics	#include <cmath> - common mathematical operations, #include <numeric> - reduction/aggregation operations
Rule of 0/3/5	Use Rule of 3/5 if managing resources; else Rule of 0 + RAII
Inheritance	class <Derived_class_name> : <access_specifier> <Parent_class_name> { // code };
Virtual & Override	virtual in base, override in derived — enables runtime dispatch
Slicing & Polymorphism	Avoid storing derived objects by value — use refs/pointers
explicit keyword	Prevents implicit conversions on single-arg constructors
Exception Safety	Destructors run in reverse. Prefer RAII (resource lifetime = object lifetime, constructor allocates, destructor releases). Use noexcept carefully
Error handling	Throw by value, catch by const reference
= default / = delete	= default for boilerplate, = delete to block copies
noexcept	Declares no exceptions — helps correctness & optimisations
Lambdas	[=] = capture by value, [&] = by ref. Use -> for return type
Random	#include <cstdlib> Use std::srand(<int>) to set seed (use time(0) for different seed), std::rand() generates rand, std::rand() % 10 generates rand between 0-9 (+1 for 1-10)
std::chrono	#include <chrono> Use steady_clock::now(), duration_cast<>, milliseconds for timing
Files	#include <fstream> ifstream for reading, ofstream for writing → <stream> var("path", mode) #include <string> getline(<fileStreamVar, StoreVar>) usually with while var.close()
std::filesystem	Use path, exists(), is_directory(), directory_iterator
Unit Testing	assert() for simple checks. No built-in test suite.
Logging	Use cerr or custom logger with timestamps.
std::optional	Wraps a value that might be missing — safer than raw pointers.
std::regex	Regex-based matching with regex_match, regex_search, regex_replace.