Modern C++, Your Conscience, and You

	Good 👼	EVIL 😈	Why?

Prefer *nullptr*	int* p = nullptr;	int* p = NULL;
		int* p = 0;
			Type safe, clear, unambiguous, correct.

Use *auto*	auto item = map.cbegin();	std::map< std::string, std::vector< double >>::const_iterator item = map.begin();	Life is short, then you die. Use *auto* to keep more time to yourself.
			
			That said, there is a legitimate philosophical tension between hiding the type with *auto* vs exposing the type explicitly. For things like this example (e.g. iterators), *auto* is a pure win. But for more common types it is sometimes better to make the type explicit to avoid unexpected mistakes. For example, this might be a bad use of *auto*:
			
			auto string = “Definitely a std::string.”; // Better: char* string = …;
			auto definitelyAFloat = 3.141; // Better: double definitelyAFloat…;

Use *auto&*	std::vector< int > v = { 1, 2, 3, 4, 5 };
	for(auto& x : v) { x *= x; }
		std::vector< int > v = { 1, 2, 3, 4, 5 };
		for(auto x : v) { x *= x; }
			Sometimes it’s simply more efficient to pass a reference. In this example, not mere simplicity, but *correctness* is involved due to the mutability of a reference. Typically, if you need to use an *auto&* you **need** to use an *auto&*. *const auto&* is common too.

Enjoy decltype	std::map< std::string, std::map< std::pair< double, std::string >, std::vector< std::string >>> firstMap;
	
	decltype(firstMap) secondMap;
		std::map< std::string, std::map< std::pair< double, std::string >, std::vector< std::string >>> firstMap;
		
		std::map< std::string, std::map< std::pair< double, std::string >, std::vector< std::string >>> secondMap;
			Often shorter. Sometimes indispensable (as with templates where you really don’t know the type of something but still need to refer to the type).

Enjoy lambdas	const bool smart = std::all_of(v.begin(), v.end(), [](decltype(v):: const_reference x) {
	return x.isSmart();
	}
		bool smart = true;
		for(const auto& x : v) {
		if(!x.isSmart()) {
		smart = false;
		break;
		}
		}
			Oh my goodness the power.

Enjoy first-class functions	int product(int x, int y, int z);
	std::function< int(int, int, int) > firstClassFunction;
	firstClassFunction = &product;
	assert(6 == firstClassFunction(1, 2, 3));
	auto productWith5 = std::bind(&product(5, std::placeholders::_1, std::placeholders::_2));
	assert(30 == productWith5(2, 3));
		I got nothing.	Can be confusing. But oh, oh so very powerful in sooo many places. Transformative.

Use const everywhere you can	const double PI = 3.14159265;
	const char* const string = “changeless”;
	class Foo {
	double twoPi() const { return PI * 2.0; }
	};
		double PI = 3.14159265;
		char* string = “changeless”;
		class Foo {
		double twoPi() { return PI * 2.0; }
		};
			const encodes semantic constraints, shrinking the graph complexity of your code, thus preventing some bugs and making others easier to find. Bonus points for using *constexpr* in cases like this example.

Pass by reference	void foo(const std::string& bar);	void foo(std::string bar);	O(n) memory and space usage is pure, unadulterated waste when O(1) is freely available. For objects larger than 8 bytes, pass by const reference rather than by value.
Avoid macros like the disease they are	inline int square(int x) { return x * x; } const std::string API_URL = “https://api.cvnt.net”;	#define square(x) ((x) * (x)) #define API_URL “https://api.cvnt.net"	Avoid macros except where they’re absolutely necessary. For the adventurous, research *constexpr*.

Avoid raw pointers like the disease they are	void foo(int& p);
	OR
	void foo(std::unique_ptr< int >&& p);
	OR
	void foo(std::shared_ptr< int > p);
		void foo(int* p);	Aggressively avoid raw pointers in modern C++. Prefer references to pointers wherever possible. Use unique, shared, and weak pointers in place of (almost) all other raw pointers. *foo** is an anti-pattern and should generally only appear when forced by external APIs.

Prefer *std::string* to *char**	const std::string API_URL = "https://api.cvnt.net";	const char* const API_URL = “https://api.cvnt.net";	Consistent use of std::string has marginal cost and avoids raw pointers and string conversion funny business.
Enjoy the new for loop syntax	for(auto x : v) { … }	for(auto iter = v.begin(); iter != v.end(); ++iter) { auto x = *iter; … }	Shorter, clearer, and less error-prone. But sometimes you need iterators or indexes, so when that happens, use them instead.

Carefully use move semantics	class Foo {
	std::vector< std::string > strings;
	void assign(std::vector< std::string >&&
	newStrings) {
	strings = std::move(newStrings); // O(1) move
	}
	};
		class Foo {
		std::vector< std::string > strings;
		void assign(const std::vector< std::string >&
		newStrings) {
		strings = newStrings; // O(n) copy
		}
		};
			Massive memory and time gains where you can get them. Takes education to do it properly though.

Use *override*	class Base {
	virtual void foo();
	};
	
	class Derived {
	virtual void foo() override;
	};
		class Base {
		virtual void foo();
		};
		
		class Derived {
		virtual void foo();
		};
			Helps avoid common errors: expected overrides that aren’t really (due to change in base class for example) and unintended overrides. Thanks C#!

Use *<chrono>*	using clock = std::chrono::high_resolution_clock;
	
	const auto start = clock::now();
	codeToBeTimed();
	const auto end = clock::now();
	const auto duration = end - start;
	std::cout << “Took “ <<
	std::chrono::duration_cast<std::chrono::milliseconds>(duration).count() << “ms.\n”;
		// Windows-only
		LARGE_INTEGER freq, start;
		QueryPerformanceFrequency(&freq);
		QueryPerformanceCounter(&start);
		codeToBeTimed();
		LARGE_INTEGER end;
		QueryPerformanceCounter(&end);
		const duration = end.QuadPart - start.QuadPart;
		std::cout << “Took “ << static_cast< int >(1000.0 * (duration / static_cast< double >(freq.QuadPart)) << “ms.\n”;
			Coherent, convenient cross-platform time library that even includes high frequency counter support.

Enjoy initializer lists	std::vector< int > v = { 1, 2, 3, 4, 5 };	std::vector< int > v;
		{
		int i = 0;
		std::generate_n(std::back_inserter(v), 5, [&]()
		{ return ++i; }
		}
			Much simpler for setting up objects and containers with known values.

Enjoy uniform initialization	std::string name{ “Billy” };	std::string name = “Billy”;	I’m not sold on this yet actually, but sometimes it’s a bit clearer, and very occasionally (very rarely, in fact) it’s absolutely necessary. Watch out for e.g. the difference between `std::vector< int > v( 5 );` and `std::vector< int > v{ 5 };`. They do completely different things.
Use raw string literals for long strings	const std::string message = R”EOF(This is a so-called “multiline” string. And note how quotes “ don’t break the string.)EOF”;	const std::string message = “This is a\n” “so-called \“multiline\”\n” “string. And note how\n” “quotes \“ don’t break\n” “the string.”;	Convenient.

Include “bare” standard headers	#include <cmath>
	#include <cassert>
		#include <math.h>
		#include <assert.h>
			The newer headers deal in namespaces; the old dump everything in the global namespace.

Prefer automatic cleanup	{
	std::unique_ptr< int > p{ new int };
	std::mutex mutex;
	std::lock_guard< std::mutex > guard(mutex);
	} // Automatic cleanup
		{
		int* p = new int;
		std::mutex mutex;
		mutex.lock();
		} // Memory leak. Dangling lock.
			Automatic necessaries are generally better than manual ones. Note that `std::auto_ptr` is now obsolete and to be avoided.

Prefer ++i to i++	for(auto iter = v.begin(); iter != v.end(); ++i) {}	for(auto iter = v.begin(); iter != v.end(); i++) {}	Believe it or not, *i++* is often more than twice as expensive as *++i*, depending on the type of i.

Use <sstream> and <iomanip> for string processing	std::istringstream in(string);
	int i, j, k;
	in >> i >> j >> k;
	
	std::ostringstream out;
	out << i << j << std::hex << k;
		int i, j, k;
		sscanf(string.str(), “%d %d %d”, &i, &j, &k);
		char buffer[GOD_PLEASE_LET_THIS_BE_LARGE_ENOUGH];
		sprintf(buffer, “%d%d%x”, &i, &j, &k);
			Vastly safer. Typically more readable and direct. Powerful error handling.

Use anonymous namespaces	namespace {
	int g_global = 0;
	}
		static int g_global = 0;	When used inside a cpp file, things inside an anonymous namespace are visible only within the current file. This is not for headers. Avoids “static” keyword ambiguity. Nicer for lumping file-private things together.