**What is Code Review**

A **code review** is a process where someone other than the author(s) of a piece of code examines that code. Code reviews are methodical assessments of code designed to identify bugs, increase code quality, and help developers learn the source code.

Code reviews act as quality assurance of the code base. Software developers should be encouraged to have their code reviewed as soon as they’ve completed coding to get a second opinion on the solution and implementation. The reviewer can also act as a second step in identifying bugs, logic problems, or uncovered edge cases.

**C++ coding rules**

1. **Minimize global and shared data**

Sharing causes contention: Avoid shared data, especially global data. Shared data increases coupling, which reduces maintainability and often performance.

1. **Use explicit RAII and smart pointers**

C++’s “resource acquisition is initialization” (RAII) idiom is the power tool for correct resource handling. RAII allows the compiler to provide strong and automated guarantees that in other languages require fragile hand-coded idioms. When allocating a raw resource, immediately pass it to an owning object.

1. **Use const proactively**

const is your friend: immutable values are easier to understand, track, and reason about, so prefer constants over variables wherever it is sensible. Make const your default choice when you define a value.

1. **Avoid macros**

Using macros can reduce the readability of your code. When you use them, you're basically creating a set of nonstandard language features.

1. **Avoid magic numbers**

Programming isn’t magic, so don’t incant it: Avoid spelling literal constants like 42 or 3.14159 in code. They are not self-explanatory and complicate maintenance by adding a hard-to-detect form of duplication.

1. **Declare variables as locally as possible**

Избегайте "раздувания" областей видимости. Переменных должно быть как можно меньше, а время их жизни - как можно короче.

1. **Always initialize variables**

Uninitialized variables are a common source of bugs in C and C++ programs. initialize variables upon definition.

1. **Preserve natural semantics for overloaded operators**

Programmers hate surprises: Overload operators only for good reason, and preserve natural semantics; if that’s difficult, you might be misusing operator overloading.

1. **Avoid overloading &&, ||, or ,(comma)**

The built-in &&, ||, and , (comma) enjoy special treatment from the compiler. If you overload them, they become ordinary functions with very different semantics and this is a sure way to introduce subtle bugs and fragilities. Don’t overload these operators naively.

1. **Prefer minimal classes to monolithic classes**

Small classes are easier to write, get right, test, and use. They are also more likely to be usable in a variety of situations.

1. **Prefer composition to inheritance**

Inheritance is the second-tightest coupling relationship in C++, second only to friendship. Tight coupling is undesirable and should be avoided where possible. Therefore, prefer composition to inheritance unless you know that the latter truly benefits your design.

1. **Avoid inheriting from classes that were not designed to be base classes**

Classes meant to be used standalone obey a different blueprint than base classes. Using a standalone class as a base is a serious design error and should be avoided. To add behavior, prefer to add nonmember functions instead of member functions. To add state, prefer composition instead of inheritance.

1. **Don’t give away your internals**

Don’t volunteer too much: Avoid returning handles to internal data managed by your class, so clients won’t uncontrollably modify state that your object thinks it owns.

1. **Define and initialize member variables in the same order**

Member variables are always initialized in the order they are declared in the class definition; the order in which you write them in the constructor initialization list is ignored.

1. **Avoid calling virtual functions in constructors and destructors**

Inside constructors and destructors, they don’t. Worse, any direct or indirect call to an unimplemented pure virtual function from a constructor or destructor results in undefined behavior.

1. **Design and write error-safe code**

In each function, give the strongest safety guarantee that won’t penalize callers who don’t need it. Always give at least the basic guarantee. Ensure that errors always leave your program in a valid state. Beware of invariant-destroying errors (including but not limited to leaks), which are just plain bugs.

1. **Prefer to use exceptions to report errors**

Prefer using exceptions over error codes to report errors. Use status codes (e.g., return codes, errno) for errors when exceptions cannot be used, and for conditions that are not errors.

1. **Throw by value, catch by reference**

Throw exceptions by value (not pointer) and catch them by reference (usually to const). This is the combination that meshes best with exception semantics.

1. **Prefer algorithm calls to handwritten loops**

For very simple loops, handwritten loops can be the simplest and most efficient solution. But writing algorithm calls instead of handwritten loops can be more expressive and maintainable, less error-prone, and as efficient.

1. **Avoid type switching; prefer polymorphism**

Avoid switching on the type of an object to customize behavior. Use templates and virtual functions to let types (not their calling code) decide their behavior.

1. **Don’t use C-style casts**

C-style casts have different (and often dangerous) semantics depending on context, all disguised behind a single syntax. Replacing C-style casts with C++-style casts helps guard against unexpected errors.

**Useful links**

1. **H. Sutter, A. Alexandrescu “C++ Coding Standards. 101 Rules, Guidelines, and Best Pracitces”**
2. [**CppCoreGuidelines/CppCoreGuidelines.md at master · isocpp/CppCoreGuidelines (github.com)**](https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md)
3. [**Google C++ Style Guide**](https://google.github.io/styleguide/cppguide.html)