C++ Coding Guidelines

# What to Do

1. Use Consistent Naming Conventions  
- Follow a clear and consistent naming convention (e.g., camelCase or snake\_case).  
- Use meaningful names for variables, functions, and classes.  
- Classes: PascalCase, variables and functions: camelCase.  
  
2. Keep Functions Small and Focused  
- Functions should do one thing and do it well. If a function does more than one thing, split it into smaller, focused functions.  
  
3. Use Smart Pointers Where Appropriate  
- Use std::unique\_ptr and std::shared\_ptr instead of raw pointers when ownership semantics need to be clear.  
- Avoid std::auto\_ptr (it’s deprecated in C++11 and removed in C++17).  
  
4. Prefer const and constexpr for Constants  
- Use const for variables that shouldn't change.  
- Use constexpr for compile-time constants to allow the compiler to optimize.  
  
5. Use Type Inference (auto) Where It Makes Code Cleaner  
- Use auto for local variable types when the type is obvious from the right-hand side, reducing redundancy and improving readability.  
- Be careful not to overuse auto in cases where the type is unclear or ambiguous.  
  
6. Prefer Range-Based For Loops  
- Use range-based for loops when iterating over collections or arrays:  
 for (auto& item : container) {  
 // process item  
 }  
  
7. Use nullptr Instead of NULL  
- nullptr is type-safe and the preferred way to represent null pointers.  
  
8. Use std::vector and std::string Instead of C Arrays  
- These standard containers manage memory automatically and are safer and more flexible than C-style arrays.  
  
9. Optimize for Readability, Not Just Performance  
- Prioritize clean, readable code over optimization unless performance is critical.  
- Use comments sparingly, but do use them to explain why something is done, especially in non-obvious parts of code.  
  
10. Leverage the Standard Library  
- Use the C++ Standard Library (STL) containers and algorithms when possible to avoid reinventing the wheel.  
  
11. Always Initialize Variables  
- Uninitialized variables can lead to undefined behavior. Always initialize variables before use.  
  
12. Use enum class for Strongly Typed Enumerations  
- Use enum class to avoid the problems of traditional enums (like implicit conversion to integers).  
 enum class Color { Red, Green, Blue };  
  
13. Document Your Code  
- Use documentation comments (e.g., /\*\* ... \*/ or ///) for functions, classes, and complex sections of code.  
  
14. Check for Resource Leaks  
- Always ensure that resources (like memory, file handles, etc.) are released properly. Use RAII (Resource Acquisition Is Initialization) to manage resource lifetime automatically.

# What Not to Do

1. Avoid Using Raw Pointers Unless Necessary  
- Raw pointers are error-prone and lead to memory management issues like memory leaks and dangling pointers.  
- Use smart pointers or references instead.  
  
2. Don’t Use using namespace std;  
- This can cause naming conflicts, especially in larger projects. Instead, prefer to specify the namespace explicitly or use std:: with the appropriate scope.  
  
3. Avoid Excessive Use of Macros  
- Macros are error-prone, difficult to debug, and can lead to unpredictable results. Prefer const, constexpr, or inline functions.  
  
4. Don’t Ignore Compiler Warnings  
- Always pay attention to compiler warnings. They usually highlight potential issues that could lead to bugs.  
  
5. Avoid Excessive `#include` Directives  
- Only include the necessary headers and avoid unnecessary inclusion of entire libraries. This reduces compilation time and minimizes dependencies.  
  
6. Don’t Use C-Style Arrays for Dynamic Memory  
- Prefer std::vector or std::array instead of C-style arrays to avoid manual memory management and bounds checking issues.  
  
7. Avoid Mixing Data Types in Containers  
- When using containers, ensure that all elements are of the same type to avoid confusion and potential errors.  
  
8. Don’t Overuse Global Variables  
- Global variables make the code harder to understand and maintain. They also introduce potential threading issues and debugging challenges.  
  
9. Avoid Complex Ternary Operators  
- While ternary operators are concise, they can make code harder to read when they become too complex. Use regular if statements for clarity when logic is complicated.  
  
10. Don’t Use goto  
- The goto statement can create tangled and difficult-to-follow control flow. Prefer structured control flow like loops and conditionals.  
  
11. Don’t Ignore Exceptions  
- Always handle exceptions, especially in cases where failure is possible. Uncaught exceptions lead to undefined behavior.  
  
12. Avoid Copying Large Objects  
- Prefer references or pointers, especially for large objects, to avoid the overhead of copying.  
  
13. Don’t Use Unnecessary Virtual Functions  
- Virtual functions introduce overhead due to dynamic dispatch. Only use virtual functions when polymorphism is required.  
  
14. Don’t Write Non-Portable Code  
- Avoid assuming specific platform or compiler behavior. Write code that is portable and follows the C++ standard.  
  
15. Avoid Tight Coupling Between Classes  
- Strongly coupled classes are hard to maintain and test. Strive for loose coupling using interfaces and abstraction.





