**Software Development in Modern C++**

**INTRODUCTION**

C++ is a multi-paradigm, imperative, procedural, functional, object-oriented, generic, and modular language invented in early 1980s and further developed by Bjarne Stroustrup.Since 1991 standardization of C++ is supported by the ANSI International Organization for Standardization (ISO). With performance and efficiency in mind, C++ extends and is compatible with the C programming language, while to incorporate the object-oriented and abstraction mechanisms it draws from Simula. This hybrid approach has proven extremely useful over the years, especially in such domains as systems programming, embedded systems, resource constrained platforms, large computing and simulation libraries, machine learning & artificial intelligence (ML/AI) and many others.

C++ is a powerful and versatile programming language that has evolved significantly over the years. With the release of C++11 and subsequent versions, a plethora of new features and best practices have been introduced, enabling developers to write more efficient, expressive, and maintainable code. C++11 and subsequent versions have introduced numerous enhancements to the standard library. These additions include new containers (e.g., unordered\_set, unordered\_map), algorithms (e.g., std::all\_of, std::any\_of), and utility classes (e.g., std::chrono, std::regex). Using these features not only simplifies code but also improves performance and code correctness.

**BEST PRACTICES**

Let’s explore some of the modern C++ best practices and features that every developer should be practicing.

1. **Exception Handling**

**1.a.** Use asserts to check for conditions that should always be true or always be false. Use exceptions to check for errors that might occur, for example, errors in input validation on parameters of public functions.

**1.b**. Throw exceptions by value, catch them by reference. Don't catch what you can't handle.

**1.d.** Use standard library exception types when they apply. Derive custom exception types from the exception class hierarchy.

1. **Smart pointers:**

C++11 has provided a great functionality of smart pointer. While dealing with dynamic programming in code, it is a good practice to use smart pointers and let the application take care of deallocation.

1. **Use Standard Library**

C++ 11 has provided good amount of libraries and try to use those as much as possible (of course as per requirement) rather going with c-style coding to prevent latency (myth).

C++ library containers like map,string are useful in terms of performance and maintainability.

1. Use initializer lists which is useful in terms of performance and readability of code.
2. Use enum class instead using plain enum.
3. There are really great static analysis tools which are useful to test our code.
4. Write tests to validate the code against every corner cases and it automates while future development and testing regression.

**BAD PRACTICES**

1. DO NOT ignore warnings shown by your compiler. Yes, there are certain warnings which are compiler dependent.
2. **Macros:**

Code in macro attracts bug e.g a space in macro function may fool with with result. A lot of macros, due to the specifics of their design, generate multiple false positives of static code analyzers. I can safely say that most false positives when checking C and C++ code are related to macros.Macros in code, False positive in static analysis of code, difficult to debug.

1. **Abort or terminating function:**

While developing libraries, we don’t know what application code is going to use that. and function calls like terminate, abort, exit may behave unexpected specially in case of embedded software development. Guidelines like MISRA, and AUTOSAR prohibits using such functions.

1. **Exception Handling :**

**4.c.** Don't use exception specifications, which are deprecated in C++11.

**4.e.** Don't allow exceptions to escape from destructors or memory-deallocation functions.