**Hello, world of concurrency in C++**

# 1.3 Concurrency and multithreading in C++

Since C++11 Standard, multithreading is supported

## 1.3.1 History of multithreading in C++

The 1998 C++ Standard doesn’t acknowledge the existence of threads. Not only that, but the memory model isn’t formally defined, so you can’t write multithreaded applications without compiler-specific extensions to the 1998 C++ Standard.

Compiler vendors are free to add extensions to the language, and the prevalence of C APIs for multithreading—such as those in the POSIX C standard and the Microsoft Windows API—has led many C++ compiler vendors to support multithreading with various platform-specific extensions.

C++ programmers have looked to their class libraries to provide object-oriented multithreading facilities. Application frameworks, such as MFC, and general-purpose C++ libraries, such as Boost and ACE, have accumulated sets of C++ classes that wrap the underlying platform-specific APIs and provide higher-level facilities for multithreading that simplify tasks.

## 1.3.2 Concurrency support in the C++11 standard

Since C++11 Standard, Not only is there a thread aware memory model, but the C++ Standard Library was extended to include classes for managing threads, protecting shared data, synchronizing operations between threads, and low-level atomic operations.

Boost Thread Library was used as the primary model on which the new library is based.

As the standard has evolved, this has been a two-way flow, and the Boost Thread Library has itself changed to match the C++ Standard in many respects.

## 1.3.3 More support for concurrency and parallelism in C++14 and C++17

C++14: new mutex type for protecting shared data, shared\_timed\_mutex and shared\_lock

C++17: adds considerably more: a full suite of parallel algorithms, shared\_mutex, scoped\_lock

Cache size access

* + hardware\_destructive\_interference\_size
  + hardware\_constructive\_interference\_size

Support for atomic operations directly in C++ enables programmers to write efficient code with defined semantics without the need for platform-specific assembly language

Optimizer can be written to take into account the semantics of the operations, enabling better optimization of the program as a whole

## 1.3.4 Efficiency in the C++ Thread Library

It’s important to understand the implementation costs associated with using any high-level facilities, compared to using the underlying low-level facilities directly. This cost is the abstraction penalty.

Standard C++ library has therefore been designed to allow for efficient implementation (with a low abstraction penalty) on most major platforms.

In those rare cases where the C++ Standard Library doesn’t provide the performance or behavior required, it might be necessary to use platform-specific facilities.

## 1.3.5 Platform-specific facilities

Thread Library may offer a native\_handle() member function that allows the underlying implementation to be directly manipulated using a platform-specific API. By its nature, any operations performed using native\_handle() are entirely platform dependent.

# References

Book | Anthony Williams - C++ Concurrency in Action