Jon Kalb Presents a Scott Meyers Training Course

Effective Modern C++

Truly understanding Modern C++ requires more than just familiarizing yourself with `auto` type declarations, move semantics, lambda expressions, concurrency support, etc. You also need to learn how to use those features *effectively* so that your software is correct, efficient, maintainable, and portable. That's where this seminar comes in. It describes how to write truly great software using C++11 and later C++ versions—i.e. using *modern C++*.

This course is based on Scott's best-selling *Effective Modern C++*.

Scott Meyers originally created this course based on his book *Effective Modern C++*. Jon has updated the material for C++17 and C++20 where directly relevant).

Course Highlights

Participants will gain:

- Familiarity with how best practices in "old" C++ programming (i.e., C++98) change for modern C++.
- Awareness of the often-subtle performance implications of modern C++ features and techniques.
- An understanding of the relationships among std::move, std::forward, rvalue references, and universal references.
- Techniques for writing clear, correct, *effective* lambda expressions.
- Insights into the pros and cons of braced initialization, noexcept specifications, perfect forwarding, and smart pointer make functions.

Who Should Attend

Systems designers, programmers, and technical managers involved in the design, implementation, and maintenance of libraries and applications using C++11 and C++14. Participants should be familiar with the fundamental concepts introduced by C++11 (e.g., move semantics, multithreading, lambda expressions, smart pointers, etc.), but expertise is not required. (One way to acquire the appropriate background is through Scott's more introductory course, *An Overview of the New C++ (C++11/14.17)*).

Format

Lecture and question/answer. There are no hands-on exercises.

Length

Four full days (six to seven lecture hours per day).

Detailed Topic Outline

- Preliminaries
  - Distinguish lvalues and rvalues.
  - Understand type deduction.
    - (`auto`-related) template type deduction
    - `auto` type deduction
    - Observing deduced types
    - `decltype` type deduction
    - Function return type deduction
  - Understand `std::move` and `std::forward`.
- Moving to Modern C++

  - Prefer `auto` to explicit types when declaring objects.
  - Remember that `auto` + `{ expr }` ⇒ `std::initializer_list`.
  - Distinguish `()` and `{}` when creating objects.
  - Prefer `nullptr` to `0` and `NULL`.
  - Prefer scoped enums to unscoped enums.
  - Prefer deleted functions to private undefined ones.
  - Declare overriding functions `override`.
  - Declare functions `noexcept` if they won't emit exceptions.
  - Use `constexpr` whenever possible.
  - Make `const` member functions thread-safe.
- Smart Pointers
  - Use `std::unique_ptr` for exclusive-ownership resource management.
  - Use `std::shared_ptr` for shared-ownership resource management.
  - Prefer `std::make_unique` and `std::make_shared` to direct use of `new`.
- Rvalue References, Move Semantics, and Perfect Forwarding
  - Distinguish universal references from rvalue references.
  - Pass and return rvalue references via `std::move`, universal references via `std::forward`.
  - Avoid overloading on universal references.
  - Understand reference collapsing.
  - Assume that move operations are not present, not cheap, and not used.
  - Familiarize yourself with perfect forwarding failure cases.
- Lambda Expressions
  - Avoid default capture modes.
  - Use init capture to move objects into closures.
  - Prefer lambdas to `std::bind`.
- The Concurrency API
  - Make `std::threads` unjoinable on all paths.

    - Use `std::launch::async` with `std::async` if asynchronicity is essential.
    - Be aware of varying thread handle destructor behavior.
    - Create tasks, not threads.
    - Consider `void` futures for one-shot event communication.
- Miscellaneous
    - Consider emplacement instead of insertion.
- Resources for Further Information

For more information on this course, contact Jon Kalb at jon@cpp.training.