Migrating a large codebase to C++ 11

Arnaud Desitter
29 June 2016

Agenda

- Motivations
- Strategy
- Compilers, tools and techniques
- Refactoring:
  - `nullptr`
  - `override`
  - `auto`
  - `std::unique_ptr`
  - move semantic
- Conclusions

Motivations

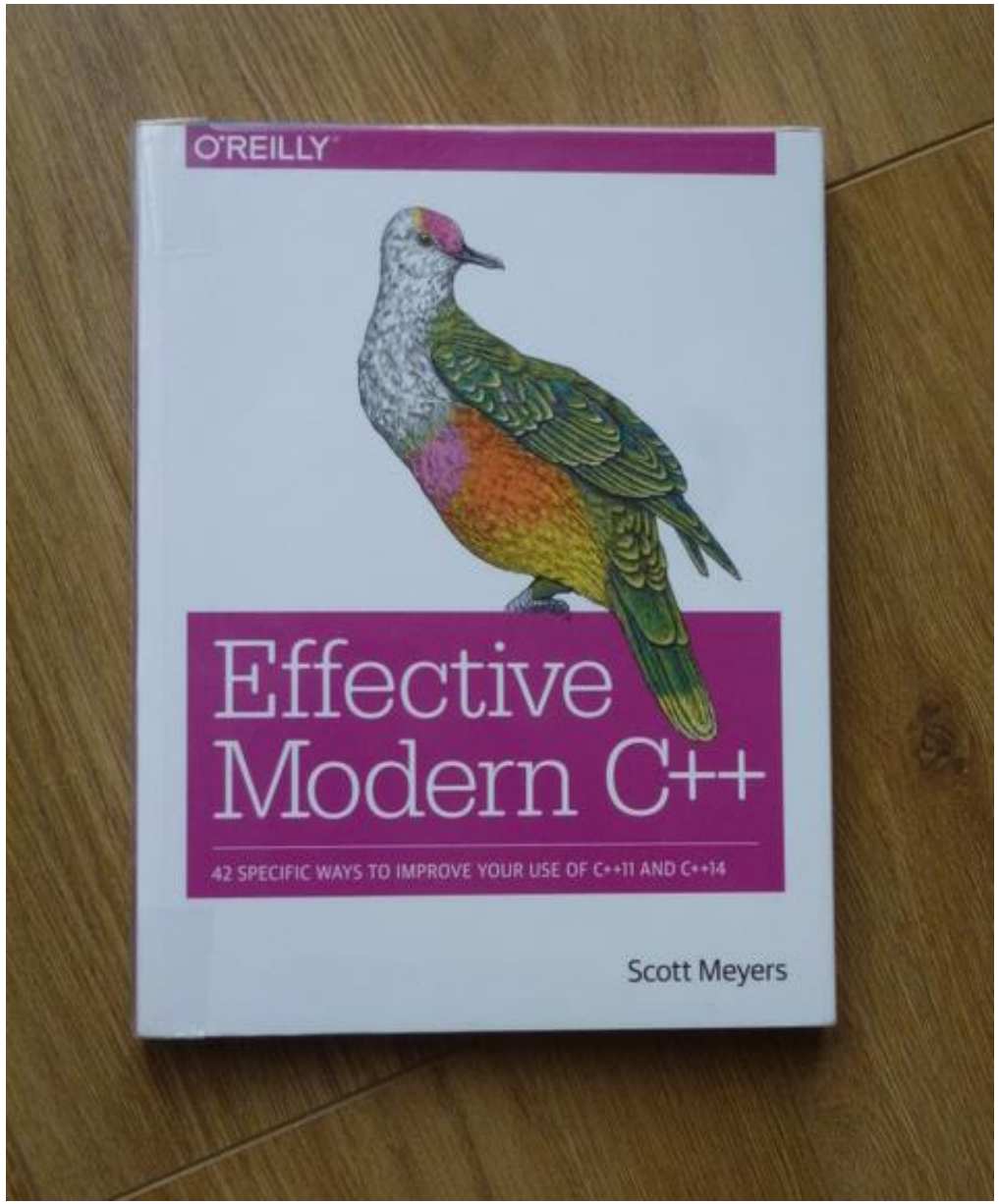
- Improve code readability
- Improve code reliability
- Improve code performance
- Retain/attract staff

Strategy

1. Identify a code transformation that achieves improved **readability** or **reliability** or **performance**.

2. Perform changes on entire codebase.

3. Enforce new code standard
   - using a compiler option,
   - using some measurements.


O'REILLY
Effective Modern C++
42 SPECIFIC WAYS TO IMPROVE YOUR USE OF C++11 AND C++14
Scott Meyers

Compilers

- Microsoft Visual Studio®
  - Major new versions break binary compatibility.
  - **No compiler option needed to enable C++11 or C++14.**
  - Assumes VS 2012 or later. VS 2013 and VS 2015 are better choices.
- GCC
  - Binary compatibility promised across versions.
  - **Need to use "`-std=c++11`" or "`-std=c++14`".**
  - A new ABI appears with GCC 5.
  - Assumes GCC 4.8.x or later.
- Clang
  - Same as GCC.

Tools and techniques

- “`clang-tidy`”
  - Part of Clang 3.8.
  - Supersedes “`clang-modernize`”.
  - My experience:
    - Clang built from source.
    - Easiest way is to use CMake to emit a compilation database.
    - Used successfully to apply specific transformations on codebases I work on.
- Compiler options
  - Warnings can be used to drive transforms and enforce code standards.
  - Make sure to exclude third-party headers (GCC’s “`-isystem`”, CMake’s `INCLUDE_DIRECTORIES(SYSTEM)`).
  - Examples: GCC’s “`-Wsuggest-override`”, “`-Wdeprecated`”, etc.

`nullptr`

- Improves readability.
- Supported by old compilers.
- Can be used with non C++11 code base (“`#define cxx11_nullptr NULL`”).
- EMC++ item 8.

- Performed by clang-tidy’s check “`modernize-use-nullptr`”.
  Works flawlessly. Highly recommended.

- GCC’s “`-Wzero-as-null-pointer-constant`”
  - Sadly, this warns about problems in third-party libraries templates (e.g. Boost) so cannot be used easily to enforce code standard.

`override`

- Improves readability and reliability. Allows safe refactoring.
- Supported by old-ish compilers.
- Can be used with non C++11 code base as it does not change semantics (“`#define cxx11_override`”).
- EMC++ item 12.

- Performed by clang-tidy’s check “`modernize-use-override`”.
  Works flawlessly. Highly recommended.

- Can be enforced with GCC 5’s “`-Werror=suggest-override`”.
  Used successfully in our codebases.

`auto`

- Improves readability **when used tastefully**.
  - A source of conflicts between developers.
  - EMC++ item 5.

- Consensus: useful for iterators. Range-base loops are even more readable.

- Performed by clang-tidy's check "`modernize-use-auto`".
  Works well. Core auditing recommended.

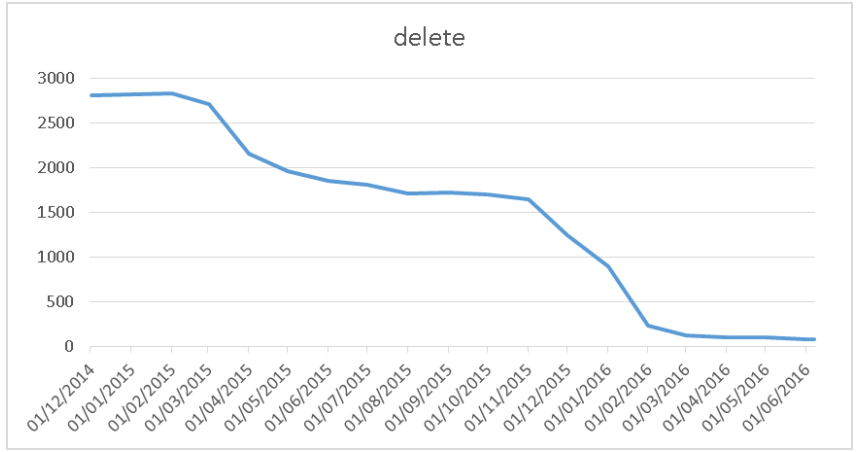
- Enforcement: look for instances of "`::iterator`" and "`::const_iterator`".

std::unique_ptr

- Enormously improves readability and reliability.
  - Highly recommended.
  - EMC++ item 18.
- Transformations:
  - `std::auto_ptr` -> `std::unique_ptr`
  - `boost::scoped_ptr` -> `const std::unique_ptr`
  - `boost::scoped_array` -> `const std::unique_ptr<[]>`
- Performed by clang-tidy's check "modernize-replace-auto-ptr".
  I did not use it as I performed gradual manual replacements.
- Enforcement:
  - GCC's "`-Wdeprecated`" for std::auto_ptr
  - Look for instances of "`boost::scoped_ptr`", etc.

std::unique_ptr

- Manual memory management -> `std::unique_ptr`
  - Manual, time-consuming code transformation but well worth it.
  - Can be done be looking for raw “`new`” and “`delete`”.
  - Special case: moving pimpl idioms to `std::unique_ptr` (EMC++ item 22).
  - Can be enforced by looking for instances of “`delete`”.



Move semantics

- Improves performance and readability.
- EMC++ items 23, 25.
- Transformations: no magic recipe, manual work.
  - Moving memory to `std::unique_ptr` helps to detect which classes can be made movable.
  - A trick: passing large objects by rvalue reference.

    e.g. `X::X(std::vector<std::vector<double>>&& data)`
    Pay for copy before the call. Very often, large objects can be moved.

Summary

- C++11 can be used to make codebase more readable, reliable and performant.

- Transformations are better done for the entire codebase and then implement a check to enforce the new code standard.

- Tools such as “clang-tidy” can be used to automate some transformations. Compiler options and searches make it possible to enforce new code standards.