Migrating a large codebase to C++ 14: further adventures

Arnaud Desitter
18 July 2017

Agenda

- Recap on last year’s talk.
- What’s new?
- A series of tips.

Recap on last year’s talk

- Use recent compilers
- Select transformations, apply on whole code base, enforce new coding standard
  - `nullptr`
  - `override`
  - `auto`
  - `std::unique_ptr`
- Use either clang-tidy or compiler options (e.g. gcc’s “`-Werror=suggest-override`”)

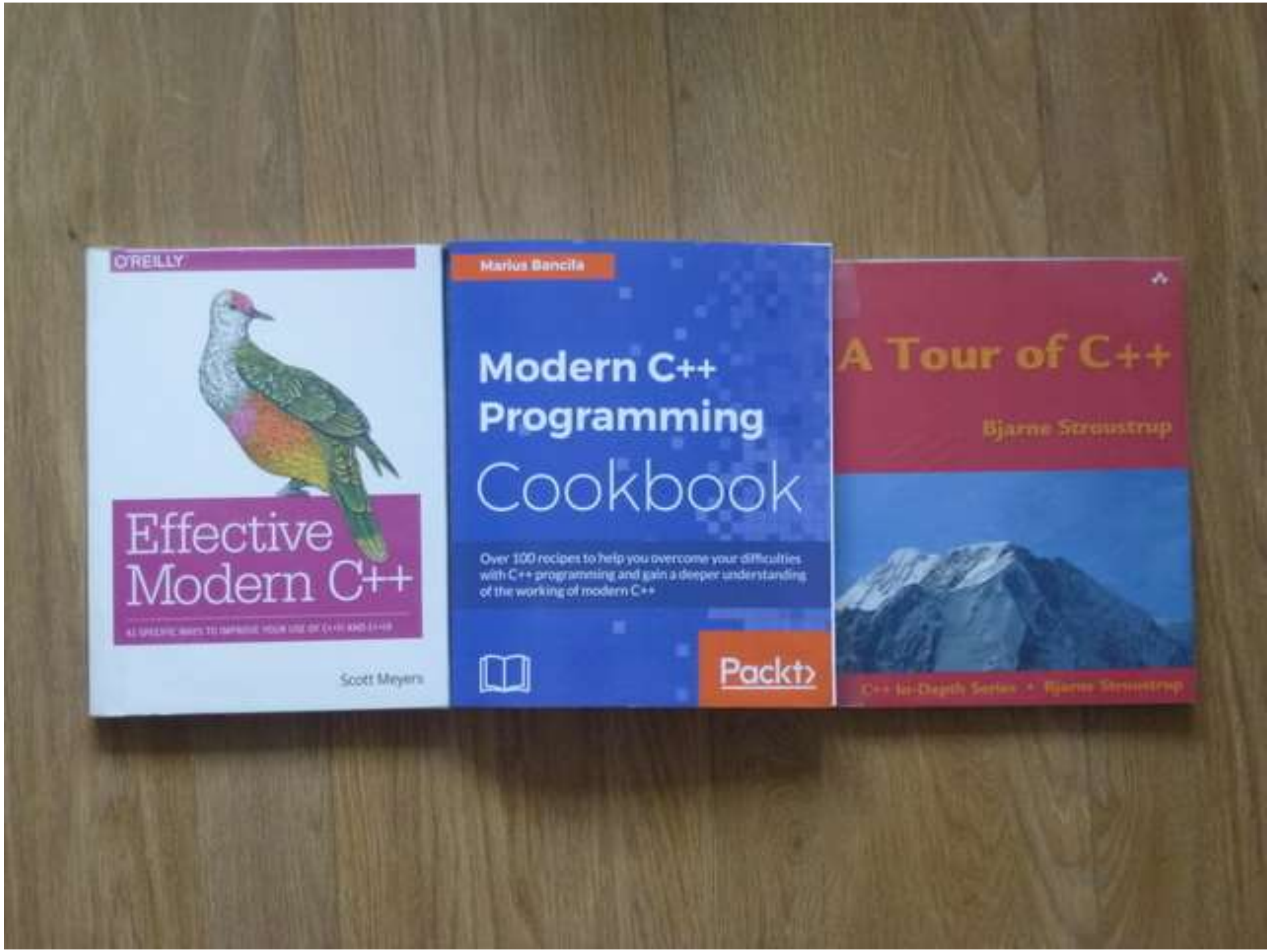
What’s new?

- C++11/14 is now more widely adopted. C++17 is coming.
- Tooling has improved. Release frequency has increased.
  - GCC, clang and Visual Studio® have a major version per year.
- Younger developers expect to use latest technology – which is great.

Tip: use good references

- Beware of out-of-date information.
- Experience is becoming available but there is still much controversy around.

1. “C++ core guidelines”
2. Many good talks online
   A favourite: CppCon 2016: Jason Turner “Practical Performance Practices"
3. Some good books…


O'REILLY
Effective Modern C++
42 SPECIFIC WAYS TO IMPROVE YOUR USE OF C++11 AND C++14
Scott Meyers
Marius Bancila
Modern C++ Programming Cookbook
Over 100 recipes to help you overcome your difficulties with C++ programming and gain a deeper understanding of the working of modern C++
Packt›
A Tour of C++
Bjarne Stroustrup
C++ In-Depth Series • Bjarne Stroustrup

Tip: use recent compilers

- Fighting old compilers may be necessary but ultimately pointless.
  - The best open-source libraries do not work with old compilers (pybin11, json parsers, trompeloeil, …).

- Improved warnings
  - GCC 5: “-Werror=suggest-override”
  - GCC 6: “-Werror=misleading-indentation” (implied by -Wall -Werror)
  - GCC 7: “-Werror=implicit-fallthrough”

- Support of C++17
  - std::string_view, std::optional, std::any, …

Tip: use Address Sanitizer (“ASAN”)

- Supported by clang and GCC.
- Checks produced at compile time.
- Can mix checked and unchecked code.

- Memory checking.
- Memory leak detection.
- No uninitialized variables detection.
- Good instrumentation of stack and global variables.

- Very fast (slowdown ~x4)

Valgrind™/ ASAN

- Valgrind
  - (+) does not need a special build
  - (+) detects uninitialized variables
  - (-) very slow

- ASAN
  - (-) needs a special build
  - (-) no detection of uninitialized variables
  - (+) fast

ASAN: more tips

- Run your unit tests with ASAN enabled.

- Be aware of environment variable “ASAN_OPTIONS”

- Detect “static initialisation fiasco”

  ```
  ASAN_OPTIONS=“check_initialization_order=1:strict_init_order=1”
  ```

  This has helped us uncover mysterious long standing issues.

Tip: use clang-tidy

- This is now the best available tool for modernising C++ code.
- It is free.
- It works well.
- Everybody is using it!

- Caveat: your code must build with clang.

clang-tidy: deployment

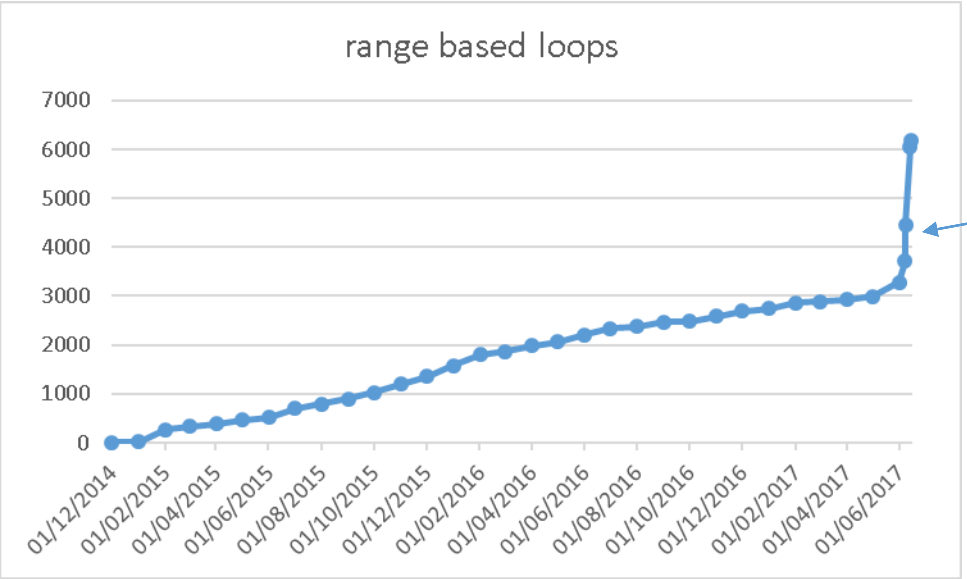
```
# create a json compilation database
cmake … -DCMAKE_EXPORT_COMPILE_COMMANDS=ON …

# run where the compilation database is
run-clang-tidy.py -p . \
 '-checks=-*,modernize-use-override' \
 '-header-filter=.*' \
 -j 32 \
 -fix
```

clang-tidy

- Some transforms are totally reliable and some need to be audited.

- Incredibly productive



clang-tidy's "modernize-loop-convert" was able to translate 3000 loops to their range-based form.

Tip: use heaptrack

- A heap memory profiler on Linux®

- Non intrusive:

```
heaptrack <your application and its parameters>
```

- Able to pinpoint “temporary allocations”

- Excellent GUI
  - Flame chart

Tip: start using C++17

- std::string_view, std::optional, std::any implementation available at

  https://github.com/tcbrindle/cpp17_headers

- A lot of “static initialisation fiasco” are related to strings.

  “std::string_view” is usually the solution.

- A lot of temporary allocations are related to strings

  “std::string_view” is usually the solution.

Tips

- Use good references
- Use recent compilers
- Use Address Sanitizer
- Use clang-tidy
- Use heaptrack
- Use C++17