The C++ Core Guidelines Project

Bjarne Stroustrup
Morgan Stanley, Columbia University
www.stroustrup.com
The big question

• “What is good modern C++?”
  • Many people want to write “Modern C++”

• What would you like your code to look like in 5 years time?
  • “Just like what I write today” is a poor answer

• The C++ Core Guidelines project
  • https://github.com/isocpp/CppCoreGuidelines
  • Produce a useful answer
    • Implies tool support and enforcement
  • Enable many people to use that answer
    • For most programmers, not just language experts
  • Please help
C++ Core Guidelines

• We offer complete type- and resource-safety
  • No memory corruption
  • No resource leaks
  • No garbage collector (because there is no garbage to collect)
  • No runtime overheads (Except where you need range checks)
  • No new limits on expressibility
  • ISO C++ (no language extensions required)
  • Simpler code
  • Tool enforced

• “C++ on steroids”
  • Not some neutered subset

Caveat: work in progress
Work in progress

• General approach
  • Guidelines
  • Library
  • Static analysis

• Not all production ready
  • Some experimental
  • Some conjectures

• Many parts in use
  • Not Science Fiction
Why not just “fix” C++?

• C++ is too big and complicated
  • Obviously
  • With many features dating back to the 1970s and 1980s
• Everybody wants “just two more features”
  • And not the same two features
• Don’t break my code!!!
  • Nobody wants their code broken, however ugly
  • There are hundreds of billions of lines of C++ code “out there”
  • There are millions of C++ programmers

• Stability/compatibility is a feature
  • We can’t simplify C++, but we can simplify the use of C++
C++ use

- About 4.5M C++ developers
- 2007-17: increase of about 100,000 developers/year
- www.stroustrup.com/applications.html
Why open source?

• Some things are suspect unless done in public
• We want everybody to benefit
• We want everybody to be able to contribute
  • Initial contributors: Morgan Stanley, Microsoft, Red Hat, Facebook, CERN
  • 230 contributors so far
Taste

• A crowd doesn’t have taste
• People with taste don’t have the same tastes

• How to maintain coherence, integrity of design?
  • Articulate design principles
  • Have a stable team of gatekeepers
Guidelines: High-level rules

• Provide a conceptual framework
  • Primarily for humans
• Many can’t be checked completely or consistently

  • P.1: Express ideas directly in code
  • P.2: Write in ISO Standard C++
  • P.3: Express intent
  • P.4: Ideally, a program should be statically type safe
  • P.5: Prefer compile-time checking to run-time checking
  • P.6: What cannot be checked at compile time should be checkable at run time
  • P.7: Catch run-time errors early
  • P.8: Don’t leak any resource
  • P.9: Don’t waste time or space
Guidelines: Lower-level rules

• Provide enforcement
  • Some complete
  • Some heuristics
  • Often easy to check “mechanically”

• Primarily for tools
  • To allow specific feedback to programmer

• Help to unify style
  • R.1: Manage resources automatically using resource handles and RAI
  • R.2: In interfaces, use raw pointers to denote individual objects (only)
  • R.3: A raw pointer (a T*) is non-owning
  • R.4: A raw reference (a T&) is non-owning
  • R.5: Prefer scoped objects, don't heap-allocate unnecessarily
  • R.6: Avoid non-const global variables

• Not minimal or orthogonal
Static analyzer (currently integrated)
GSL – Guidelines support Library

- Minimal, to be absorbed into ISO C++
- `not_null, owner, Expects, Ensures, ...`
- `span`
  - Non-owning potentially run-time checked reference to a continuous sequence
    - Implemented as a pointer, integer pair
      ```cpp
      int a[100];
      span s {a}; // note: template argument deduction
      for (auto x : s) // note: no range error, not nullptr check
          cout << x << '\n';
      ```

Overview

• Maintain static type safety
  • Avoid cast and un-tagged unions

• Be precise about ownership
  • Don’t litter
  • Use ownership abstractions

• Eliminate dangling pointers

• Make general resource management implicit
  • Hide every explicit delete/destroy/close/release
  • “lots of explicit annotations” doesn’t scale

• Static guarantees (run-time is too late)

• Test for `nullptr` and range
  • Minimize run-time checking
  • Use checked library types