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](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++
• About 4.5M C++ developers
• 2007-17: increase of about 100,000 developers/year
• www.stroustrup.com/applications.html
Coding guidelines

• We need guidelines for writing good modern C++
  • Static type safe
  • No resource leaks
  • No dangling pointers
  • No range errors
  • No nullptr references
  • No misuse of unions
  • No casts
  • No bloat
  • No messy error-prone low-level code
  • No known inefficiencies
  • Good use of the standard library
  • ...

How?
We all hate coding rules*†

• Rules are (usually)
  • Written to prevent misuse by poor programmers
    • “don’t do this and don’t do that”
  • Written by people with weak experience with C++
    • At the start of an organization’s use of C++

• Rules (usually) focus on
  • “layout and naming”
  • Restrictions on language feature use
  • Not on programming principles

• Rules (usually) are full of bad advice
  • Write “pseudo-Java” (as some people thought was cool in 1990s)
  • Write “C with Classes” (as we did in 1986)
  • Write C (as we did in 1978)
  • …

*Usual caveats
†and thanks
Coding guidelines

• Let’s build a good set!
  • Comprehensive
  • Browsable
  • Supported by tools (from many sources)
  • Suitable for gradual adoption

• For modern C++
  • Compatibility and legacy code be damned! (initially)

• Prescriptive
  • Not punitive

• Teachable
  • Rationales and examples

• Flexible
  • Adaptable to many communities and tasks

• Non-proprietary
  • But assembled with taste and responsiveness

• We aim to offer guidance
  • What is good modern C++?
  • Confused, backwards-looking teaching is a big problem
Current (Partial) Solutions

• These are old problems and old solutions
  • 40+ years
• Manual resource management doesn’t scale
• Smart pointers add complexity and cost
• Garbage collection is at best a partial solution
  • Doesn’t handle non-memory solutions ("finalizers are evil")
  • Is expensive at run time
  • Is non-local (systems are often distributed)
  • Introduces non-predictability
• Static analysis doesn’t scale
  • Gives false positives (warning of a construct that does not lead to an error)
  • Doesn’t handle dynamic linking and other dynamic phenomena
  • Is expensive at compile time
Our solution: A cocktail of techniques

- Not a single neat miracle cure
  - Rules (from the “Core C++ Guidelines”)
    - Statically enforced
  - Libraries (STL, GSL)
    - So that we don’t have to directly use the messy parts of C++
  - Reliance on the type system
    - The compiler is your friend
  - Static analysis
    - To extend the type system

- None of those techniques is sufficient by itself
- Enforces basic ISO C++ language rules
- Not just for C++
  - But the “cocktail” relies on much of C++
Subset of superset

• Simple sub-setting doesn’t work
  • We need the low-level/tricky/close-to-the-hardware/error-prone/expert-only features
    • For implementing higher-level facilities efficiently
    • Many low-level features can be used well
  • We need the standard library

• Extend language with a few abstractions
  • Use the STL
  • Add a small library (the GSL)
    • No new language features
    • Messy/dangerous/low-level features can be used to implement the GSL
  • Then subset

• What we want is “C++ on steroids”
  • Simple, safe, flexible, and fast
  • Not a neutered subset
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 RAlI
  • 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';
      ```
Core Rules

• Some people will not be able to apply all rules
  • At least initially
  • Gradual adoption will be very common

• Many people will need additional rules
  • For specific needs

• We initially focus on the core rules
  • The ones we hope that everyone eventually could benefit from

• The core of the core
  • No leaks
  • No dangling pointers
  • No type violations through pointers
No resource leaks

• We know how
  • Root every object in a scope
    • `vector<T>`
    • `string`
    • `ifstream`
    • `unique_ptr<T>`
    • `shared_ptr<T>`
  • RAII
    • “No naked` new”
    • “No naked` delete”
Dangling pointers – the problem

• One nasty variant of the problem

```cpp
void f(X* p)
{
  // ...
  delete p;  // looks innocent enough
}

void g()
{
  X* q = new X;  // looks innocent enough
  f(q);
  // ... do a lot of work here ...
  q->use();  // Ouch! Read/scramble random memory
}
Dangling pointers

- We **must** eliminate dangling pointers
  - Or type safety is compromised
  - Or memory safety is compromised
  - Or resource safety is compromised

- Eliminated by a combination of rules
  - Distinguish owners from non-owners
  - Assume raw pointers to be non-owners
  - Catch all attempts for a pointer to “escape” into a scope enclosing its owner’s scope
    - **return**, **throw**, out-parameters, long-lived containers, ...
  - Something that holds an owner is an owner
    - E.g. `vector<owner<int*>>`, `owner<int*>[]`, ...
Owners and pointers

- Every object has one owner
- An object can have many pointers to it
- No pointer can outlive the scope of the owner it points to
- An owner is responsible for owners in its object

*For an object on the free store the owner is a pointer*
*For an object on the stack the owner itself*
*For a static object the owner is itself*
How do we represent ownership?

• Low-level: mark owning pointers **owner**
  • An **owner** must be **deleted** or passed to another **owner**
  • A non-**owner** may not be **deleted**

• High-level: Use an ownership abstraction
  • Low-level owner annotations don’t scale
  • Use them only for
    • C-style pointer interfaces
    • In ownership abstraction implementations

• Note
  • I talk about pointers
  • What I say applies to anything that refers to an object
    • References, Containers of pointers, Smart pointers, ...
GSL: owner<T>

• How do we implement ownership abstractions?
  
  template<SemiRegular T>
  class vector {
    owner<T*> elem; // the anchors the allocated memory
    T* space;      // just a position indicator
    T* end;        // just a position indicator
    // ...
  };

• owner<T*> is just an alias for T*
  template<typename T> using owner = T;
How to avoid/catch dangling pointers

• Classify pointers according to ownership

  vector<int*>  // returning non-owner
  f(int* p)       // return p would be OK
  {
    int x = 4;       // return &x would be bad: local
    int* q = new int{7};  // return q would be bad: owner
    vector<int*> res = {p, &x, q};  // Bad: { unknown, pointer to local, owner }
    return res;
  }

• Don’t mix different ownerships in an array

• Don’t let different return statements of a function mix ownership
Dangling pointer summary

• Simple:
  • We never let a “pointer” point to an out-of-scope object
• It’s not just pointers
  • All ways of “escaping”
    • return, throw, place in long-lived container, ...
  • Same for containers of pointers
    • E.g. vector<int*>, unique_ptr<int>, iterators, built-in arrays, ...
  • Same for references
Other problems

• Other ways of misusing pointers
  • Range errors: use std::span<T>
  • nullptr dereferencing: use gsl::not_null<T>

• Wasteful ways of addressing pointer problems
  • Misuse of smart pointers

• Other ways of breaking the type system (beyond the scope of this talk)
  • Unions: use std::variant
  • Casts: don’t except for hardware quantities (e.g., device registers)

• “Just test everywhere at run time” is not an acceptable answer
  • Hygiene rules
  • Static analysis
  • Run-time checks
• In: Introduction
• P: Philosophy
• I: Interfaces
• F: Functions
• C: Classes and class hierarchies
• Enum: Enumerations
• R: Resource management
• ES: Expressions and statements
• Per: Performance
• CP: Concurrency and parallelism
• E: Error handling
• Con: Constants and immutability
• T: Templates and generic programming
• CPL: C-style programming
• SF: Source files
• SL: The Standard Library

Supporting sections
• A: Architectural ideas
• NR: Non-Rules and myths
• RF: References
• Pro: Profiles
• GSL: Guidelines support library
• NL: Naming and layout rules
• FAQ: Answers to frequently asked questions
• Appendix A: Libraries
• Appendix B: Modernizing code
• Appendix C: Discussion
• Appendix D: Supporting tools
• Glossary
• To-do: Unclassified proto-rules
Expression rules

- **ES.40**: Avoid complicated expressions
- **ES.41**: If in doubt about operator precedence, parenthesize
- **ES.42**: Keep use of pointers simple and straightforward
- **ES.43**: Avoid expressions with undefined order of evaluation
- **ES.44**: Don't depend on order of evaluation of function arguments
- **ES.45**: Avoid "magic constants"; use symbolic constants
- **ES.46**: Avoid narrowing conversions
- **ES.47**: Use `nullptr` rather than 0 or NULL
- **ES.48**: Avoid casts
- **ES.49**: If you must use a cast, use a named cast
- **ES.50**: Don't cast away const
- **ES.55**: Avoid the need for range checking
- **ES.56**: Write `std::move()` only when you need to explicitly move an object to another scope
- **ES.60**: Avoid new and delete outside resource management functions
- **ES.61**: Delete arrays using `delete[]` and non-arrays using `delete`
- **ES.62**: Don't compare pointers into different arrays
- **ES.63**: Don't slice
- **ES.64**: Use the T{e} notation for construction
- **ES.65**: Don't dereference an invalid pointer
Arithmetic rules

- **ES.100**: Don't mix signed and unsigned arithmetic
- **ES.101**: Use unsigned types for bit manipulation
- **ES.102**: Use signed types for arithmetic
- **ES.103**: Don't overflow
- **ES.104**: Don't underflow
- **ES.105**: Don't divide by zero
- **ES.106**: Don't try to avoid negative values by using unsigned
- **ES.107**: Don't use unsigned for subscripts, prefer `gsl::index`
Parameter passing semantic rules:

- **F.22**: Use `T*` or `owner<T*>` to designate a single object
- **F.23**: Use a `not_null<T>` to indicate that "null" is not a valid value
- **F.24**: Use a `span<T>` or a `span_p<T>` to designate a half-open sequence
- **F.25**: Use a `zstring` or a `not_null<zstring>` to designate a C-style string
- **F.26**: Use a `unique_ptr<T>` to transfer ownership where a pointer is needed
- **F.27**: Use a `shared_ptr<T>` to share ownership

Value return semantic rules:

- **F.42**: Return a `T*` to indicate a position (only)
- **F.43**: Never (directly or indirectly) return a pointer or a reference to a local object
- **F.44**: Return a `T&` when copy is undesirable and "returning no object" isn't needed
- **F.45**: Don't return a `T&&`
- **F.46**: `int` is the return type for `main()`
- **F.47**: Return `T&` from assignment operators
- **F.48**: Don't return `std::move(local)`
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