

Writing good C++14

Bjarne Stroustrup

Morgan Stanley, Columbia University

www.stroustrup.com



The problem and the opportunity

- We have a great modern language
 - C++11 (good)
 - -> C++14 (better)
 - -> C++17 (much better still, I hope)
 - Technical specifications
 - Shipping
 - in wide-spread production work
 - and more facilities well in the works
 - C++1*
 - is easier to write and maintain
 - runs faster
 - can express more than older C++
 - with less code
 - **Many** people want to write "Modern C++"
 - "What is good modern C++?"



The problem and the opportunity

- Many people
 - Use C++ in archaic or foreign styles
 - Get lost in details
 - Are obsessed with language-technical details

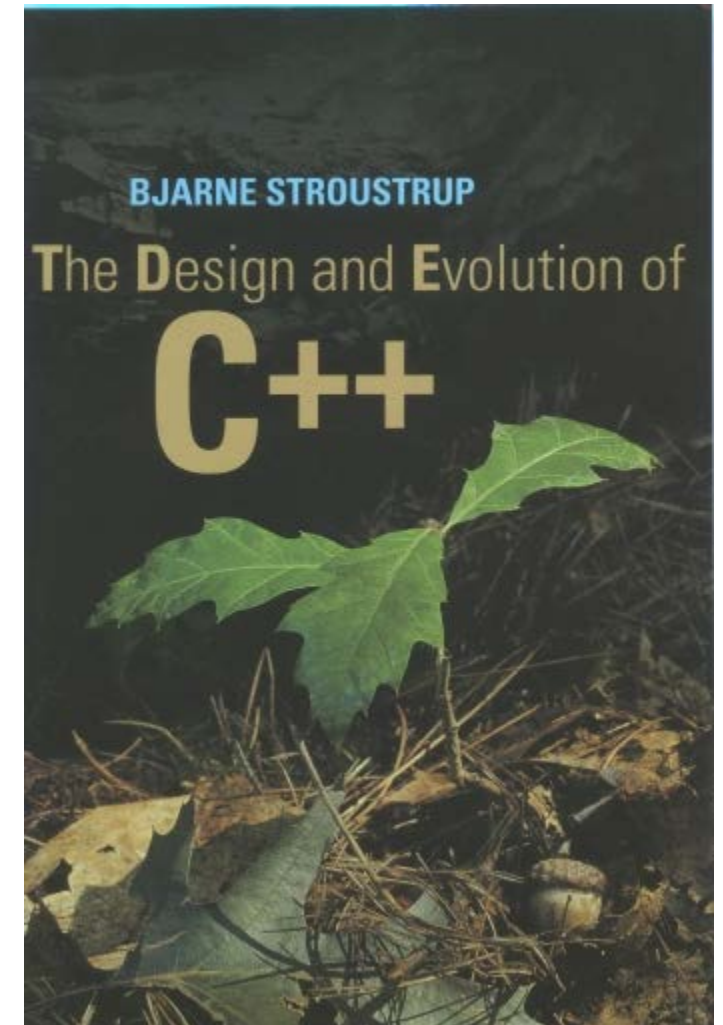
Doctor, doctor, it hurts when I do X!!!
So don't do X



- “Within C++ is a smaller, simpler, safer language struggling to get out”
 - Code can be simpler
 - as efficient as ever
 - as expressive as ever

A smaller, simpler C++

- Let's get it out
 - **Now!**
 - Without inventing a new language
 - 100% compatibility – compile with current compilers
- Coding guidelines
 - Supported by a “guidelines support library” (GSL)
 - Supported by analysis tools
- Don't sacrifice
 - Generality
 - Performance
 - Simplicity
 - Portability across platforms



A smaller, simpler C++

- I think **we** can do it
 - I can't do it alone
 - No individual can
 - No single company can
- Please help!



Initial work (still incomplete)

- I describe significant initial work
 - Microsoft (Herb Sutter and friends)
 - Morgan Stanley (Bjarne Stroustrup and friends)
 - CERN (Axel Naumann and friends)
- Available
 - Core guidelines (now)
 - Guidelines support library (now; Microsoft, GCC, Clang)
 - Analysis tool (Microsoft in October; ports later (November?))
 - MIT License
- Related CppCon talks
 - Herb Sutter: ***Writing Good C++14 By Default*** (Tuesday)
 - Gabriel Dos Reis: ***Modules*** (Tuesday)
 - Gabriel Dos Reis: ***Contracts*** (Wednesday)
 - Neil MacIntosh: ***Static analysis*** (Wednesday)
 - Neil MacIntosh: ***array_view, string_view, etc.*** (Wednesday)

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”
 - Language features
 - Restrictions
 - Not on programming principles
- Rules (usually) are full of bad advice
 - Write “pseudo-Java” (as some people thought was cool in 1994)
 - Write “C with Classes” (as we did in 1986)
 - Write C (as we did in 1978)
 - ...

*Usual caveats

†and thanks

Coding rules*

- Are outdated
 - Become a drag of their users
- Are specialized
 - but used outside their intended domain
- Are not understood by their users
 - Enforced by dictate: Do this or else!
 - Require detailed language-lawyer knowledge to follow
- Are not well supported by tools
 - Platform dependencies
 - Compiler dependencies
 - Expensive
- Are too simple or too complicated
 - For real-world use
 - Naming and layout rules are useful, but not all we need
 - Restrictions can be essential, but we need guidance

*Usual caveats



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
- Flexible
 - Adaptable to **many** communities and tasks
- Non-proprietary
 - But assembled with taste and responsiveness
- Teachable
 - Rationales and examples

High-level rules

- Provide a conceptual framework
- 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*



Lower-level rules

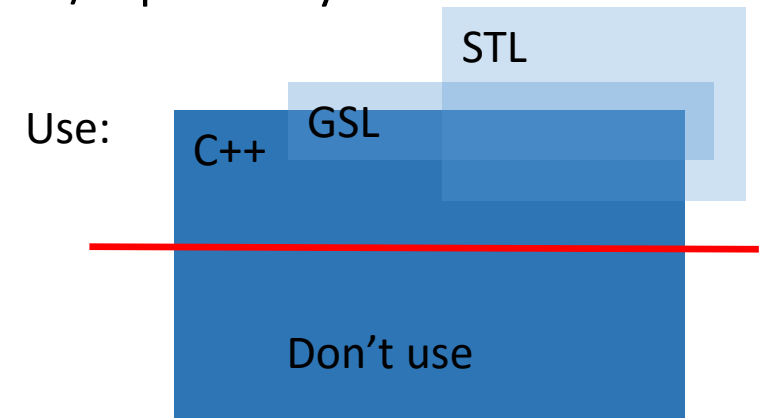
- Provide enforcement
- Often easy to check “mechanically”
- Unify style
 - *F.16: Use T^* or **owner** $\langle T^* \rangle$ to designate a single object*
 - *C.49: Prefer initialization to assignment in constructors*
 - *ES.20: Always initialize an object*
- Checking
 - Some complete
 - Some heuristics
 - Many rely on static analysis
 - Some beyond our current tools
- Not minimal or orthogonal

The structure of a rule

- The rule itself - e.g., ***no naked `new`***
- **Reference number** - e.g., **C.7** (the 7th rule related to classes).
- **Reason** (rationale) - because programmers find it hard to follow rules they don't understand
- **Example** - because rules are hard to understand in the abstract; can be positive or negative
- **Alternative** - for "don't do this" rules
- **Exception** - we prefer simple general rules. However, many rules apply widely, but not universally
- **Enforcement** - ideas about how the rule might be checked "mechanically"
- **See also** - references to related rules and/or further discussion (in this document or elsewhere)
- **Note** (comments) - something that needs saying that doesn't fit the other classifications
- **Discussion** - references to more extensive rationale and/or examples placed outside the main lists of rules

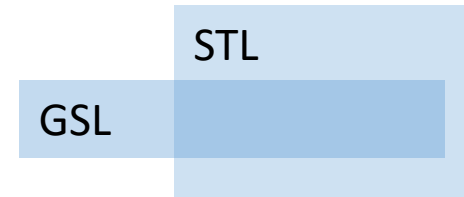
Subset of superset

- 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



Some rules rely on libraries

- The ISO C++ standard library
 - E.g., **vector<T>** and **unique_ptr<T>**
- The Guideline Support Library
 - E.g., **array_view<T>** and **not_null<T>**
- Some rules using the GSL
 - *l.11: Never transfer ownership by a raw pointer (T^*)*
 - Use an ownership pointer (e.g. **unique_ptr<T>**) or **owner<T*>**
 - *l.12: Declare a pointer that may not be the **nullptr** as **not_null***
 - E.g., **not_null<int*>**
 - *l.13 Do not pass an array as a single pointer*
 - Use a handle type, e.g., **vector<T>** or **array_view<T>**



Double our productivity without loss of performance

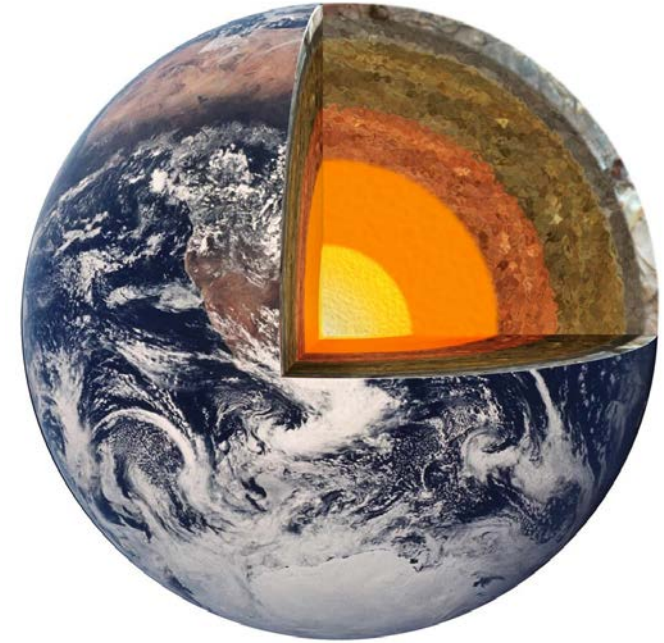
- “Imitate experienced programmers”
- Eliminate whole classes of errors
 - Fewer crashes and security violations
- Simplify
 - Simplicity aids maintenance
 - Consistent style speeds up learning
 - Guide people away from obscure corners and exotic technique
 - Emphasis on avoiding waste improves performance
 - Separate rules for exceptional needs

Have you gone mad? (no)

- We attack the most common and the most serious sources of errors
 - I hate debugging
- We eliminate whole classes of errors
 - Eliminate resource leaks
 - Without loss of performance
 - Eliminate dangling pointers
 - Without loss of performance
 - Eliminate out-of-range access
 - With minimal cost
- Tool support is essential
 - Static analysis
 - Support library (tiny)
 - Reinforce the type system

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

- Root every object in a scope
 - `vector<T>`
 - `string`
 - `ifstream`
 - `unique_ptr<T>`
 - `shared_ptr<T>`
- Mark owning `T*` pointers `owner<T*>`
 - In resource manager classes
 - In code that cannot be changed (e.g. for ABI reasons)



Dangling pointers

- Ensure that no pointer outlives the object it points to
- 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*>[], ...`



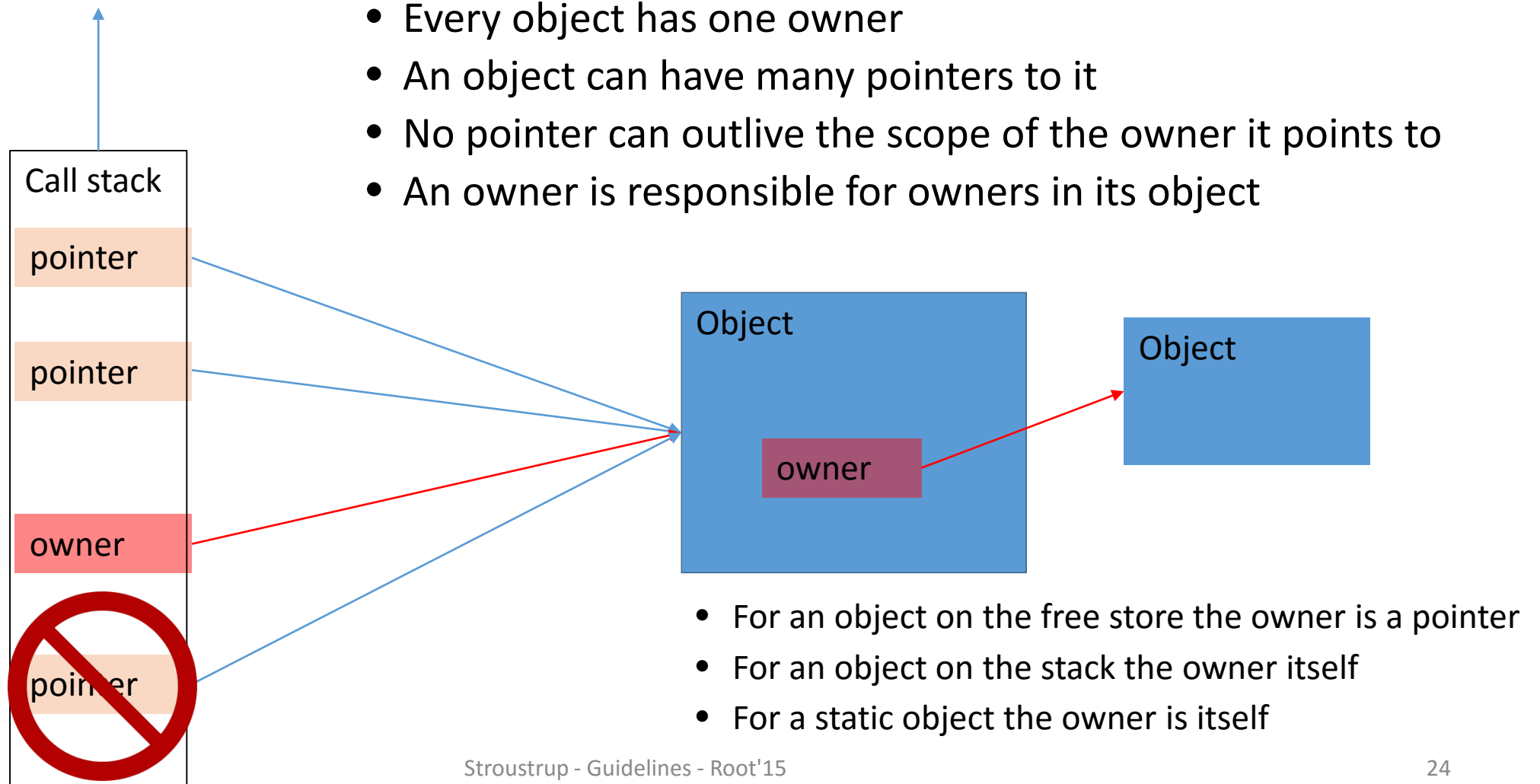
Dangling pointers

- Ensure that no pointer outlives the object it points to

```
void f(X* p) {  
    // ...  
    delete p;           // bad: delete non-owner  
}  
  
void g()  
{  
    X* q = new X;       // bad: assign object to non-owner  
    f(q);  
    // ... do a lot of work here ...  
    q.use(); // Ouch!  
}
```



Owners and pointers



How to avoid/catch dangling pointers

- Rules (giving pointer safety):
 - Don't transfer to pointer to a local to where it could be accessed by a caller
 - A pointer passed as an argument can be passed back as a result
 - A pointer obtained from new can be passed back as a result as an owner

```
int* f(int* p)
{
    int x = 4;
    return &x;           // No! would point to destroyed stack frame
    return new int{7};    // OK (sort of: doesn't dangle, but returns an owner as an int*)
    return p;            // OK: came from caller
}
```


How to avoid/catch dangling pointers

- 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
- Never let a “pointer” point to an out-of-scope object

How to avoid/catch dangling pointers

- Classify pointers according to ownership

```
vector<int*> f(int* p)  
{  
    int x = 4;  
    int* q = new int{7};  
    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

GSL: owner<T>

- How do we implement those 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;**
 - an annotation to help static analysis tools
 - A low-level ownership indicator

GSL: owner<T>

- How about code we cannot change?

```
void f(owner<int*> p, int* q)
{
    // ...
    delete p;           // necessary
    delete q;           // bad: not an owner
}
```

- **owner<T*>** is just an alias for **T***
 - E.g., **template<typename X> using owner = X;**
 - an annotation to help static analysis tools
 - we have ABI compatibility
 - You must **delete** an owner or pass it along to another owner
 - **new** returns an owner

Other problems

- Other ways of misusing pointers
 - Range errors: **array_view<T>**
 - **nullptr** dereferencing: **not_null<T>**
- Other ways of breaking the type system (beyond the scope of this talk)
 - Unions
 - Casts
- Wasteful ways of addressing pointer problems
 - Misuse of smart pointers
- “Just test everywhere at run time” is ***not*** an acceptable answer
 - Hygiene rules
 - Static analysis
 - Run-time checks

GSL - array_view<T>

- Common style

```
void f(int* p, int n)           // what is n? (How would a tool know?)
{
    p[7] = 9;                   // OK?
    for (int i=0; i<n; ++i) p[i] = 7; // OK?
}
```

- Better

```
void f(array_view<int> a)
{
    a[7] = 9;                   // OK? Checkable against a.size()
    for (int x : a) a = 7;      // OK
}
```

GSL - array_view<T>

- Common style

```
void f(int* p, int n);  
int a[100];  
// ...  
f(a,100);  
f(a,1000);    // likely disaster
```

- “Make simple things simple”

- Simpler than “old style”
- Shorter
- At least as fast
- Sometimes using the GSL
- Sometimes using the STL

- Better

```
void f(array_view<int> a)  
int a[100];  
// ...  
f(array_view<int>{a});  
f(a);  
f(a,1000);    // easily checkable
```

nullptr problems

- Mixing **nullptr** and pointers to objects
 - Causes confusion
 - Requires (systematic) checking

- Caller

```
void f(char*);
```

```
f(nullptr);           // OK?
```

- Implementer

```
void f(char* p)
```

```
{
```

```
    if (p==nullptr)    // necessary?
```

```
    // ...
```

```
}
```

- Can you trust the documentation?
- Compilers don't read manuals, or comments
- Complexity, errors, and/or run-time cost

GSL - not_null<T>

- Caller

```
void f(not_null<char*>);
```

```
f(nullptr);    // Obvious error: caught by static analysis
```

```
char* p = nullptr;
```

```
f(p);          // Constructor for not_null can catch the error
```

- Implementer

```
void f(not_null<char*> p)
```

```
{
```

```
    // if (p==nullptr) // not necessary
```

```
    // ...
```

```
}
```

GSL - `not_null<T>`

- **`not_null<T>`**
 - A simple, small class
 - **`not_null<T*>`** is **`T*`** except that it cannot hold **`nullptr`**
 - Can be used as input to analyzers
 - Minimize run-time checking
 - Checking can be “debug only”
 - For any **`T`** that can be compared to **`nullptr`**
 - E.g. **`not_null<array_view<T>>`**

(Mis)uses of smart pointers

- Consider
 - `void f(T*);` *// use; no ownership transfer or sharing*
 - `void f(unique_ptr<T>);` *// transfer unique ownership and use*
 - `void f(shared_ptr<T*>);` *// share ownership and use*
- Taking a raw pointer (`T*`)
 - Is familiar
 - Is simple, general, and common
 - Is cheaper than passing a smart pointer (usually)
 - Doesn't lead to dangling pointers
 - Doesn't lead to replicated versions of a function for different shared pointers
- In terms of tradeoffs with smart pointers, other simple “object designators” are equivalent to `T*`
 - iterators, references, **`array_view`**, etc.

(Mis)uses of smart pointers

- Don't use ownership pointers unless you change ownership

```
void f(X*);                // just uses X; no ownership transfer or sharing – good
void g(shared_ptr<X>);      // just uses X – bad
unique_ptr<X> h(unique_ptr<X>); // just uses X – bad (give pointer back to prevent destruction)
```

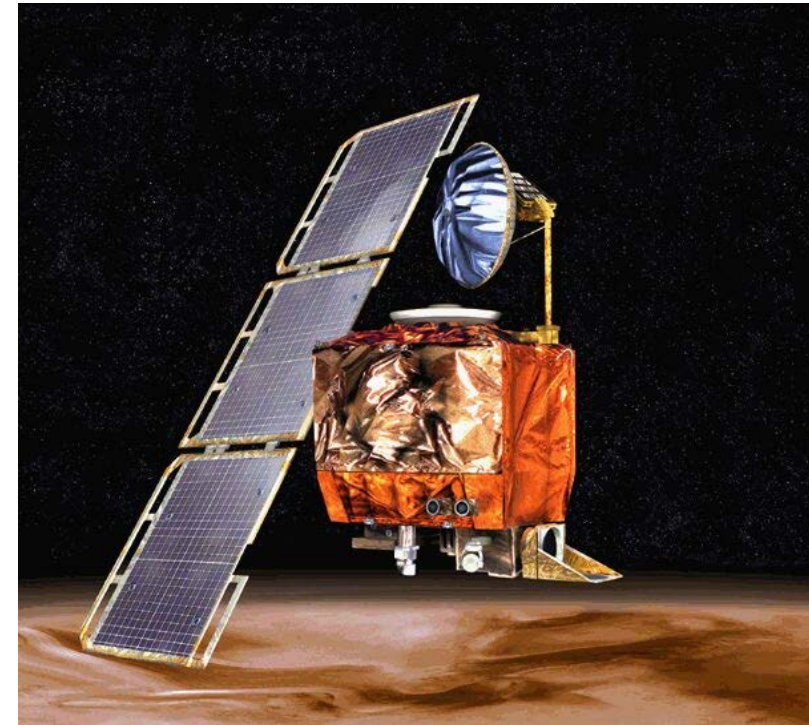
```
void use()
{
    auto p = make_shared<X>{};
    f(p.get());                // extract raw pointer (note: pointers do not dangle)
    g(p);                      // mess with use count (probably a mistake)
    auto q = h(make_unique<X>(p.get())); // (probably a mistake)
                                // extract raw pointer, then wrap it and copy
    q.release();               // prevent destruction
}
```

Rules, standards, and libraries

- Could the rules be enforced by the compiler?
 - Some could, but we want to use the rules *now*
 - Many could not
 - Rules will change over time
 - Compilers have to be more careful about false positives
 - Compilers cannot ban legal code
- Could the GSL be part of the standard?
 - Maybe, but we want to use it *now*
 - The GSL is tiny and written in portable C++11
 - The GSL does not depend on other libraries
 - The GSL is similar to, but not identical to **boost::** and **experimental::** components
 - So they may become standard
- We rely on the standard library

Too many rules

- For
 - Novices, experts, infrastructure, ordinary large applications, low-latency, high-reliability, security targets, hard-real time
- You can't remember all of those rules!
- You don't need all of those rules
- You couldn't learn all of those rules before writing code
- You'd hate to even look through all of those rules
- The rule set must be extensible
 - you'll never know them all
- The tools know the rules
 - And will point you to the relevant ones



Rule classification

- P: Philosophy
- I: Interfaces
- F: Functions
- C: Classes and class hierarchies
- Enum: Enumerations
- ES: Expressions and statements
- E: Error handling
- R: Resource management
- T: Templates and generic programming
- CP: Concurrency
- The Standard library
- SF: Source files
- CPL: C-style programming
- GSL: Guideline support library

Supporting sections

- NL: Naming and layout
- PER: Performance
- N: Non-Rules and myths
- RF: References
- Appendix A: Libraries
- Appendix B: Modernizing code
- Appendix C: Discussion
- To-do: Unclassified proto-rules

We are not unambitious

- Type and resource safety
 - No leaks
 - No dangling pointers
 - No bad accesses
 - No range errors
 - No use of uninitialized objects
 - No misuse of
 - Casts
 - Unions
- We think we can do it
 - At scale
 - 4+ million C++ Programmers, N billion lines of code
 - Zero-overhead principle



We aim to change the way we write code

- That means ***you***
- What would you like your code to look like in 5 years?
 - Once we know, we can aim to achieve that
 - Modernizing a large code base is not easy
 - The answer is not “just like my code today”
 - Think “gradual adoption” (except for brand-new code)
- Not everybody will agree what the code should look like
 - Not all code should look the same
 - We think there can be a common core
 - We need discussion, feedback, and a variety of tools
- Help wanted!
 - Rules, tools, reviews, comments
 - editors



Current status

- Available
 - About 350 Rules (<https://github.com/isocpp/CppCoreGuidelines>)
 - GSL for Clang, GCC, and Microsoft (<https://github.com/microsoft/gsl>)
 - First tools: October for Microsoft; ports later (November?)
 - MIT License
- We need help
 - Review of rules
 - More examples and refinements for existing rules
 - Specialized rule sets
 - For particular application areas, projects, ...
 - For concurrency
 - For libraries
 - ...
- Continuous development
 - “forever”

The basic C++ model is complete

- We
 - Eliminate dangling pointers
 - Eliminate resource leaks
 - Check for range errors (optionally and cheaply)
 - Check for **nullptr** (optionally and cheaply)
 - Have concepts
- Why not a new C++-like language?
 - Competing with C++ is hard
 - Most attempts fail, C++ constantly improves
 - It would take 10 years (at least)
 - And we would still have lots of C and C++
 - A new C++-like language might damage the C++ community
 - Dilute support, divert resources, distract



Questions

- P: Philosophy
- I: Interfaces
- F: Functions
- C: Classes and class hierarchies
- Enum: Enumerations
- ES: Expressions and statements
- E: Error handling
- R: Resource management
- T: Templates and generic programming
- CP: Concurrency
- The Standard library
- SF: Source files
- CPL: C-style programming
- GSL: Guideline support library

Supporting sections

- NL: Naming and layout
- PER: Performance
- N: Non-Rules and myths
- RF: References
- Appendix A: Libraries
- Appendix B: Modernizing code
- Appendix C: Discussion
- To-do: Unclassified proto-rules



Coding guidelines

- Boost Library Requirements and Guidelines
- Bloomberg: BDE C++ Coding
- Facebook: ???
- GCC Coding Conventions
- Google C++ Style Guide
- JSF++: JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS
- Mozilla Portability Guide.
- Geosoft.no: C++ Programming Style Guidelines
- Possibility.com: C++ Coding Standard
- SEI CERT: Secure C++ Coding Standard
- High Integrity C++ Coding Standard
- llvm.org/docs/CodingStandards.html

Non-aims

- Create “the one true C++ subset”
 - There can be no such marvel
 - Core guidelines + guidelines for specific needs
- Making a totally flexible set of rules to please everybody
 - Our rules are **not** value neutral
 - Total freedom is chaos
 - We want “modern C++”
 - **not** “everything anyone ever thought was cool and/or necessary”
- Turning C++ into Java, Haskell, C, or whatever
 - “If you want Smalltalk you know where to find it”
- What we want is “**C++ on steroids**”
 - Simple, safe, flexible, and fast
 - Not a neutered subset

Philosophy

- Attack hard problems
 - Resources, interfaces, bounds, ...
- Be prescriptive
 - “don’t do that” is not very helpful
- Give rationale
 - “because I say so” is not very helpful
- Offer machine-checkable rules
 - Machines are systematic, fast, and don’t get bored
- Don’t limit generality
 - For most of us most of the time
- Don’t compromise performance
 - Of course
- Subset of superset
 - Don’t fiddle with subtle language rules