No littering!

Bjarne Stroustrup
Morgan Stanley, Columbia University
www.stroustrup.com

Work in progress

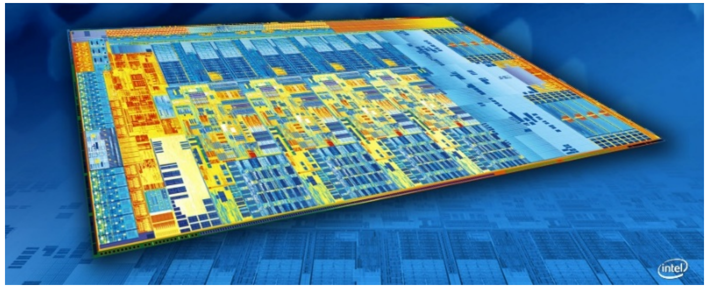
- Not all production ready
  - Some experimental
  - Some conjectures
- Many parts in use
- Not Science Fiction

Morgan Stanley

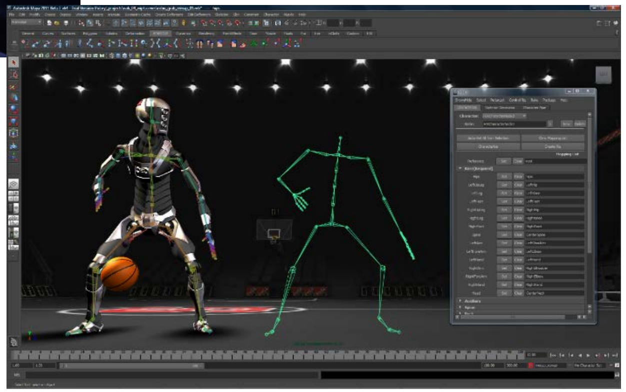
Executive summary

- We now 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
- Support
  - C++ Core Guidelines: https://github.com/isocpp/CppCoreGuidelines
  - GSL: https://github.com/microsoft/gsl
  - Static analysis/enforcement: In Microsoft Visual Studio, a bit in Clang tidy
- "C++ on steroids"
  - Not some neutered subset

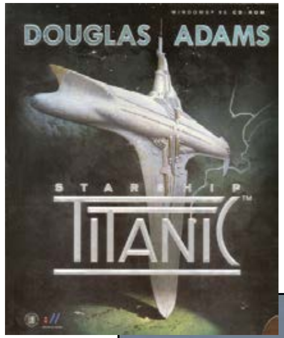
Caveat: work in progress

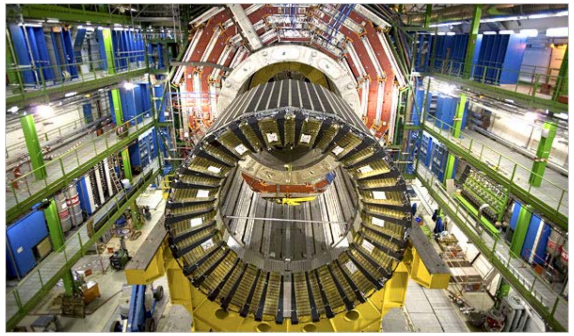


C++ use



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





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
- 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

P: Philosophical rules

- 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 resources
- P.9: Don't waste time or space
- P.10: Prefer immutable data to mutable data
- P.11: Encapsulate messy constructs, rather than spreading through the code
- P.12: Use supporting tools as appropriate
- P.13: Use support libraries as appropriate

Morgan Stanley

Resource management rule summary:

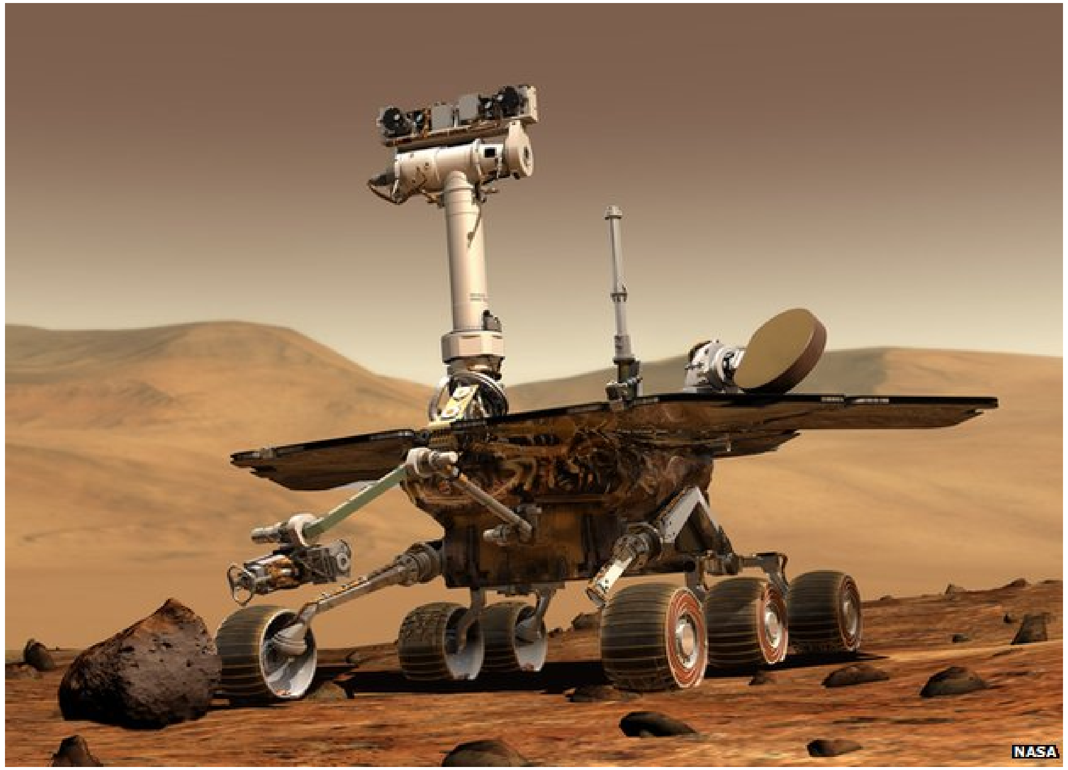
- R.1: Manage resources automatically using resource handles and RAII
- R.2: In interfaces, use raw pointers to denote individual objects (only)
- R.3: A raw pointer (a T*) is non-owning
- R.4: A raw reference (a T&) is non-owning
- R.5: Prefer scoped objects, don't heap-allocate unnecessarily
- R.6: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

Morgan Stanley

ES: Expressions and Statements

- General rules:
  - ES.1: Prefer the standard library to other libraries and to "handcrafted code"
  - ES.2: Prefer suitable abstractions to direct use of language features
- Declaration rules:
  - ES.5: Keep scopes small
  - ES.6: Declare names in for-statement initializers and conditions to limit scope
  - ES.7: Keep common and local names short, and keep uncommon and nonlocal names longer
  - ES.8: Avoid similar-looking names
  - ES.9: Avoid ALL_CAPS names
  - ES.10: Declare one name (only) per declaration
  - ES.11: Use auto to avoid redundant repetition of type names
  - ES.12: Do not reuse names in nested scopes
  - ES.20: Always initialize an object
  - ES.21: Don't introduce a variable (or constant) before you need to use it
  - …

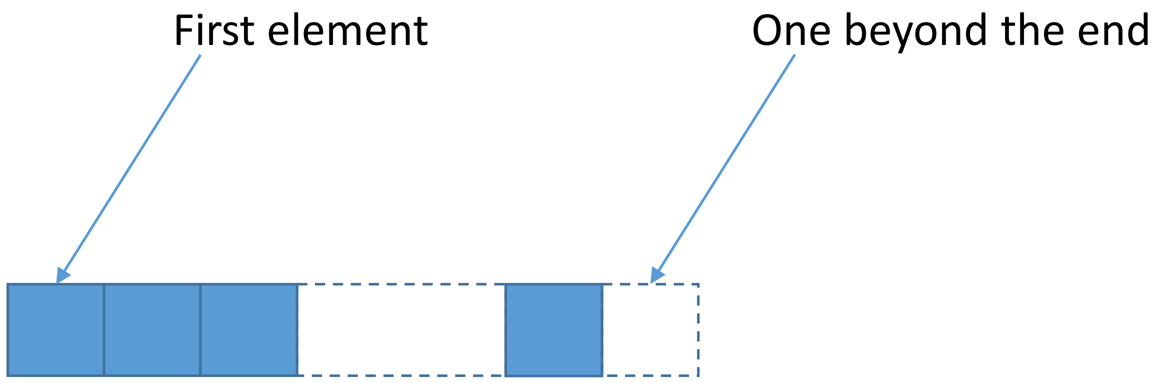
Overview



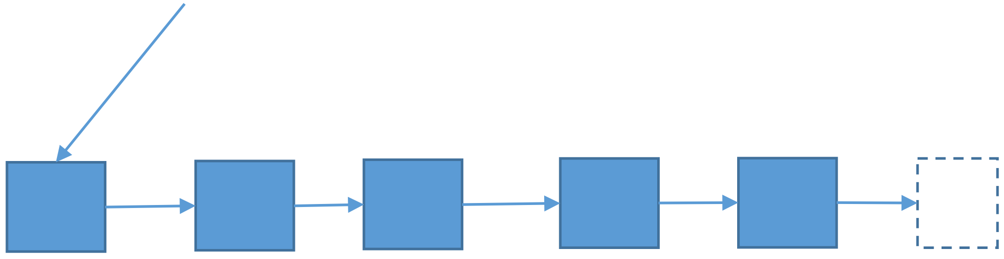
- Pointer problems
  - Memory corruption
  - Resource leaks
  - Expensive run-time support
  - Complicated code
- The solution
  - Eliminate dangling pointers
  - Eliminate resource leaks
  - Library support for range checking (**span**) and **nullptr** checking
  - And then deal with casts and unions (**variant**)

I like pointers!

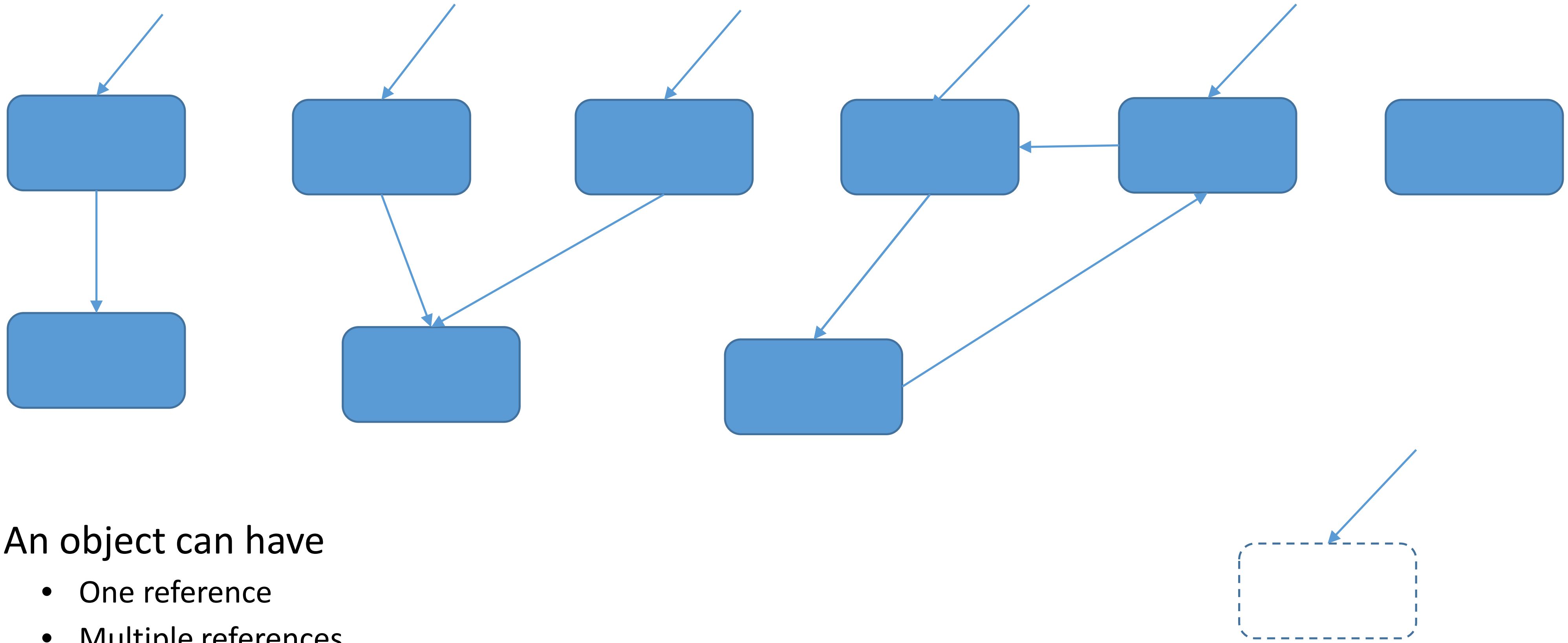
- Pointers are what the hardware offers
  - Machine addresses
  - For good reasons
    - They are simple
    - They are general
    - They are fast
    - They are compact



- C's memory model has served us really well for decades
  - Sequences of objects
- But pointers are not "respectable"
  - Dangerous, low-level, not mathematical, …
  - There is a huge ABP crowd

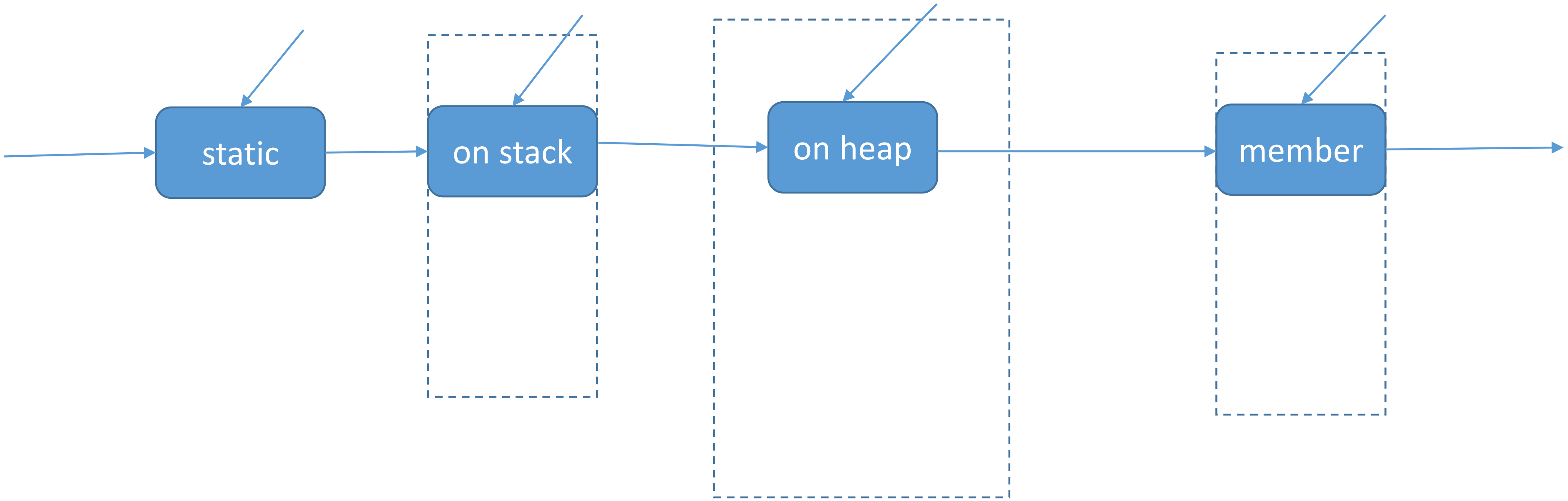


Lifetime can be messy



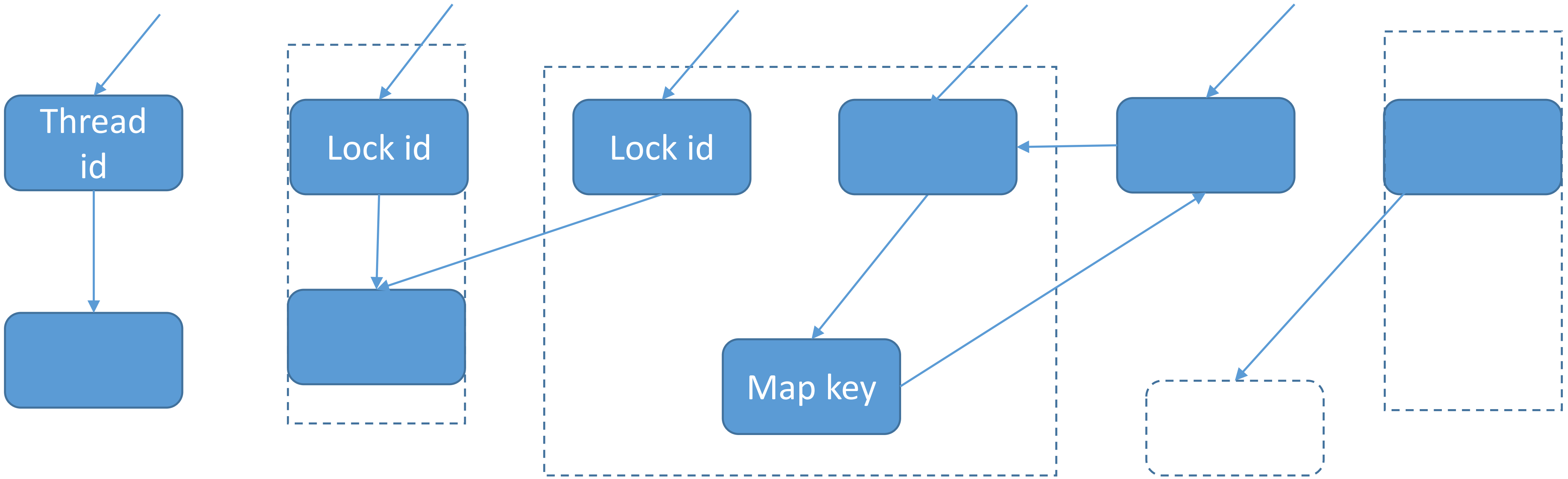
- An object can have
  - One reference
  - Multiple references
  - Circular references
  - No references (leaked)
  - Reference after deletion (dangling pointer)

Ownership can be messy



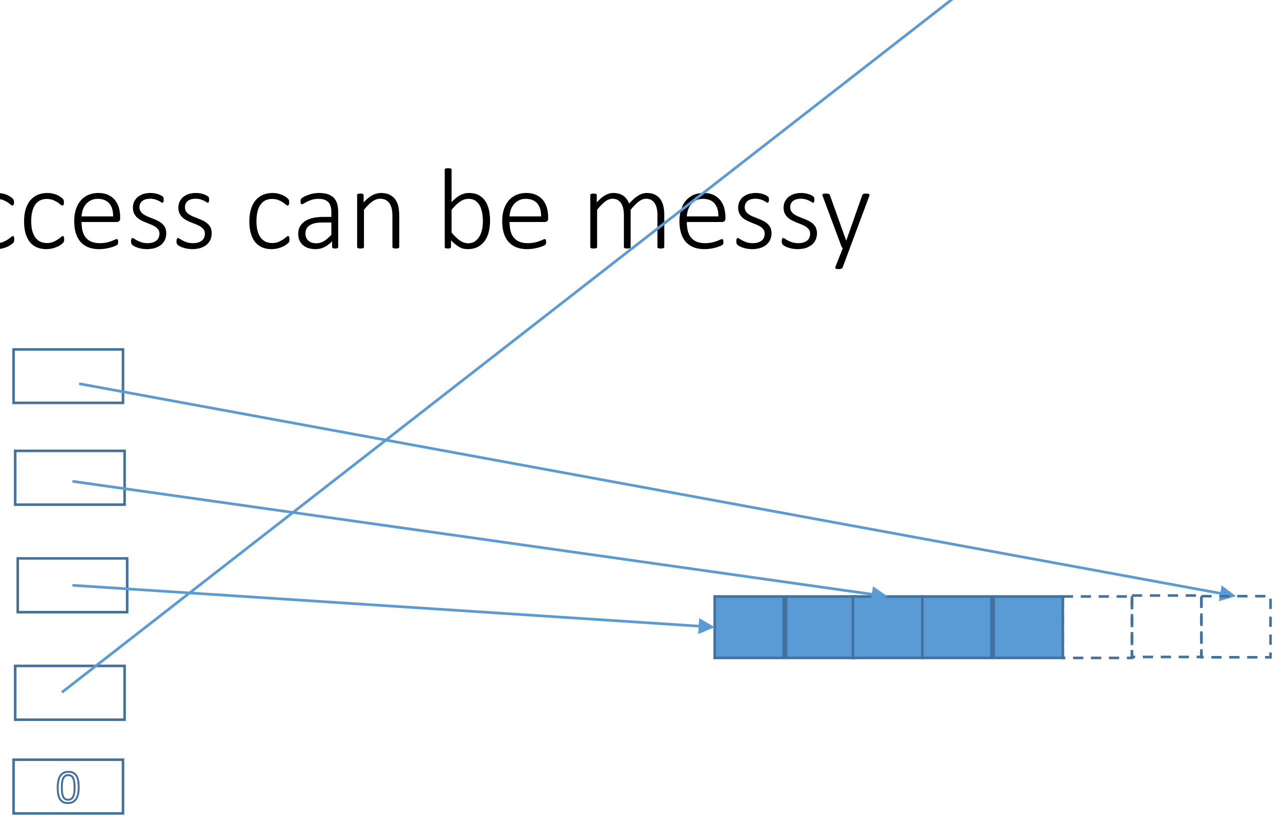
- An object can be
  - on stack (automatically freed)
  - on free store (must be freed)
  - in static store (must never be freed)
  - in another object

Resource management can be messy



- Objects are not just memory
- Sometimes, significant cleanup is needed
  - File handles
  - Thread handles
  - Locks
  - …

Access can be messy



- Pointers can
  - point outside an object (range error)
  - be a **nullptr** (useful, but don't dereference)
  - be unititialized (bad idea)
  - Point to memory formerly used by an object that has been deleted

Eliminate all leaks and all memory corruption

- Every object is constructed before use
  - Once only
- Every fully constructed object is destroyed
  - Once (and only once)
  - Every object allocated by **new** must be **delete**d (once and only once)
  - No scoped object must be **delete**d (it is implicitly destroyed)
- No access through a pointer that does not point to an object
  - Read or write
  - Off the end of an object (out of range)
  - To **delete**d object
  - To “random” place in memory (e.g., uninitialized pointer)
  - Through **nullptr** (originally: “there is no object at address zero”)
  - That has gone out of scope

Current (Partial) Solutions

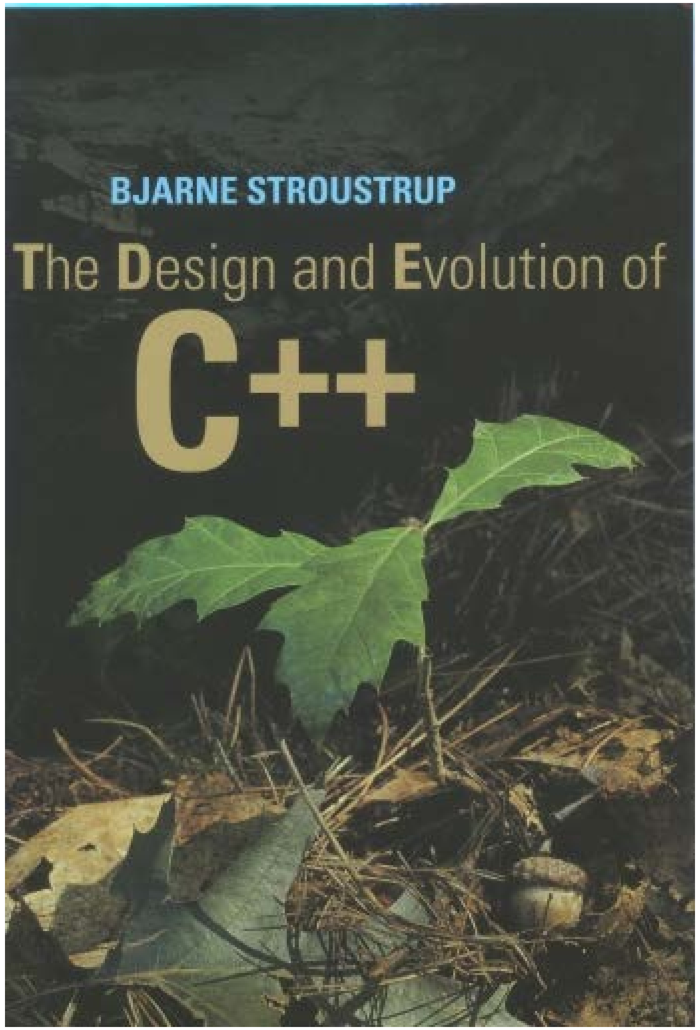
- Ban or seriously restrict pointers
  - Add indirections everywhere
  - Add checking everywhere
- Manual memory management
  - Combined with manual non-memory resource management
- Garbage collectors
  - Plus manual non-memory resource management
- Static analysis
  - To supplement manual memory management
- "Smart" pointers
  - Starting with counted pointers
- Functional Programming
  - Eliminate pointers

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

Constraints on the solution

- I want it ***now***
  - I don’t want to invent a new language
  - I don’t want to wait for a new standard
- I want it guaranteed
  - “Be careful” isn’t good enough
- Don’t sacrifice
  - Generality
  - Performance
  - Simplicity
  - Portability



A solution

- Be precise about ownership
  - Don’t litter
  - Offer static guarantee of release/destruction
- Eliminate dangling pointers
  - Static guarantee (run-time is too late)
- Make general resource management implicit
  - Hide every explicit delete/destroy/close/release
  - “lots of explicit annotations” doesn’t scale
    - becomes a source of bugs
- Test for **nullptr** and range
  - Minimize run-time checking
  - Use checked library types
- Avoid other problems with pointers
  - Avoid cast and un-tagged unions

No resource leaks

- We know how
  - Root every object in a scope
    - **vector<T>**
    - **string**
    - **ifstream**
    - **unique_ptr<T>**
    - **shared_ptr<T>**
    - **lock_guard<T>**
  - RAII
    - “No naked **new**”
    - “No naked **delete**”
  - Constructor/destructor
    - “since 1979, and still the best”

Dangling pointers

- One nasty variant of the problem

```
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
    - Non-owner: **T***
    - Primitive: **gsl::owner<T*>**
    - **Best: vector<T>, unique_ptr<T>, …**
    - Something that holds an owner is an owner
    - Don't forget **malloc()**, etc.
  - Catch every attempt for a pointer to "escape" into a scope enclosing its owner's scope
    - **return**, **throw**, out-parameters, long-lived containers, …

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();		// we never get here
}
```

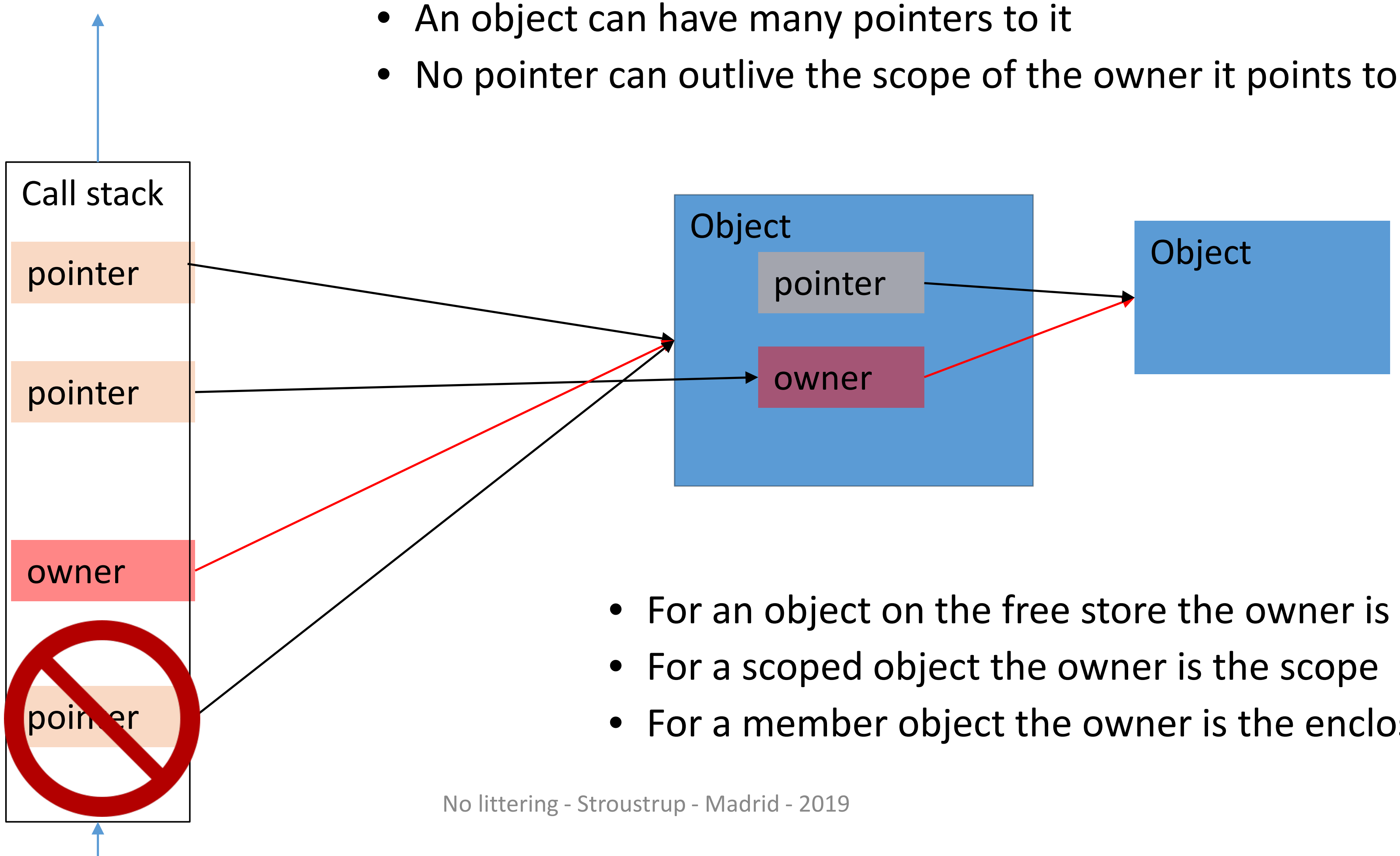
How to avoid/catch dangling pointers

- Rules (giving pointer safety):
  - Basic rule: no pointer must outlive the object it points to
  - Practical rules
    - 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
      - Essential for real-world pointer use
    - A pointer obtained from new can be passed back
      - But we have to remember to eventually delete it

```
int*  f(int* p)
{
      int x = 4;
      return &x;                // No! would point to destroyed stack frame
      return new int{7};        // OK: doesn't dangle, but we must "remember" to delete
      return p;                 // OK: came from caller
}
```

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



- For an object on the free store the owner is a pointer
- For a scoped object the owner is the scope
- For a member object the owner is the enclosing object

How do we manage ownership?

- High-level: Use an ownership abstraction
  - Simple and preferred
    - E.g., **unique_ptr**, **shared_ptr**, **vector**, and **map**
- Low-level: mark owning pointers **owner**
  - An **owner** must be **delete**d or passed to another **owner**
  - A non-**owner** may not be **delete**d
  - This is essential in places but does not scale
  - Applies to both pointers and references

"...And that, in simple terms, is what's wrong with your software design."

How do we represent ownership?

- Mark an owning **T***: **gsl::owner<T*>**
  - Initial idea (2005 and before)
    - Yet another kind of "smart pointer"
    - **owner<T*>** would hold a **T*** and an "owner bit"
    - Costly: bit manipulation
    - Not ABI compatible
    - Not C compatible
    - Finds errors too late (at run time)
  - So **gsl::owner**
    - Is a handle for static analysis
    - Is documentation
    - Is not a type with it's own operations
    - Incurs no run-time cost (time or space)
    - Is ABI compatible
    - **template<typename T> using owner = T;**

GSL is our Guidelines Support Library

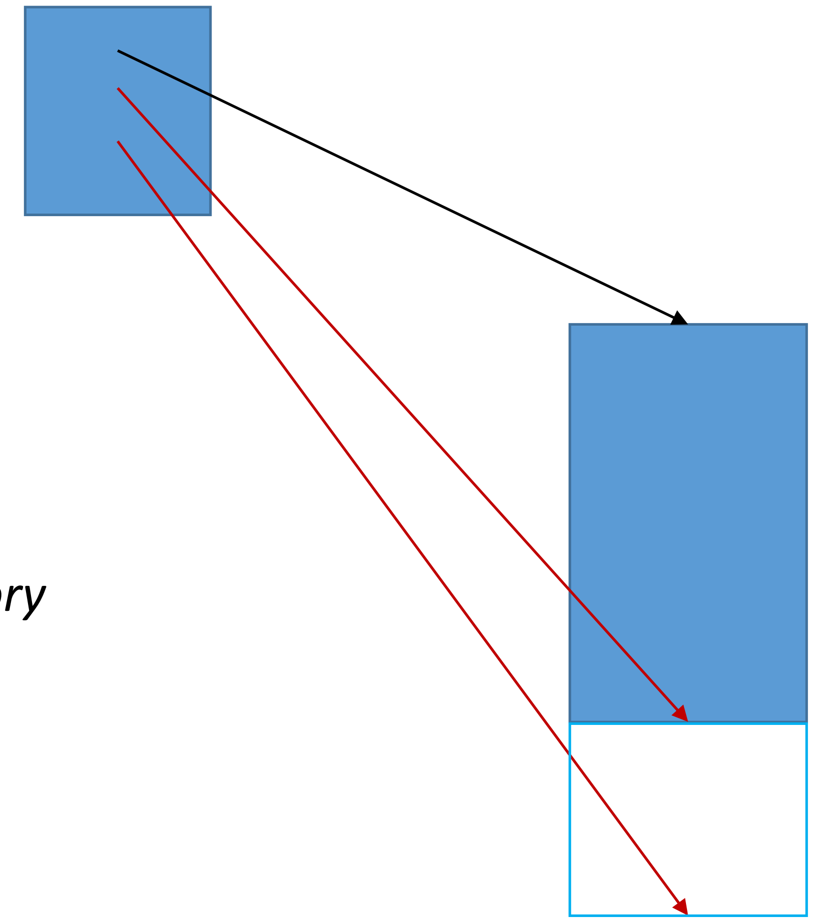
How do we manage ownership?

- **owner** is intended to simplify static analysis
  - Necessary inside ownership abstractions
  - **owner**s in application code is a sign of a problem
    - Usually, C-style interfaces
  - "Lots of annotations" doesn't scale
    - Becomes a source of errors
- **owner<T*>** is just an alias for **T***
  - **template<T> using owner = T;**

GSL: owner<T>

- How do we implement ownership abstractions?

```
template<SemiRegular T>
class vector {
public:
        // …
private:
        owner<T*> elem;        // the anchors the allocated memory
        T* space;              // just a position indicator
        T* end;                // just a position indicator
        // …
};
```



GSL: owner<T>

- How about code we cannot change?
  - ABI stability

```
void foo(owner<int*>);           // foo requires an owner

void f(owner<int*> p, int* q, owner<int*> p2, int* q2)
{
    foo(p);                      // OK: transfer ownership
    foo(q);                      // bad: q is not an owner
    delete p2;                   // necessary
    delete q2;                   // bad: not an owner
}
```

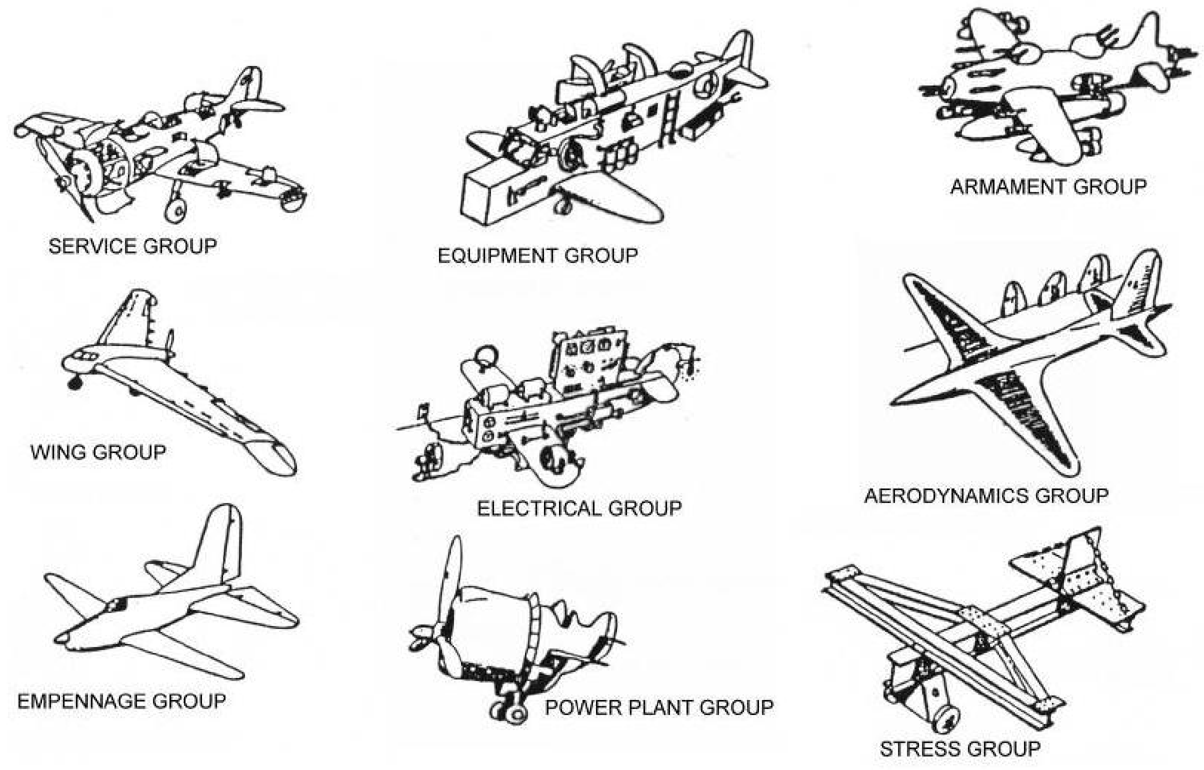
- A static analysis tool can tell us where our code mishandles ownership

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++

Details (aka engineering)

- "Invention is 1% inspiration and 99% perspiration" – T. Edison
- The simple lifetime and ownership model needs to be enforced by many dozens of detailed checks
  - Be comprehensive
  - Minimize false positives
  - Scale to industrial programs
    - Fast analysis is essential – local analysis only
  - Allow for gradual adoption
  - Provide coherent toolsets for all platforms



"Static" is not quite as flexible as "dynamic"

- Classify pointers according to lifetime

```
int glob = 666;

vector<int*> f(int* p)                         // ignore ownership for now
{
        int x = 4;
        int* q = new int{7};
        vector<int*> res = {p, &x, q, &glob};  // potentially bad: mix lifetimes
        return res;                            // Bad: return { unknown, &local, free store, &global }
}
```

- Don't mix different lifetimes in an array (overly conservative?)
  - If you must, encapsulate
- Don't let return statements mix lifetimes

"Static" is not quite as flexible as "dynamic"

- Classify pointers according to ownership

```
int glob = 666;

vector<int*> f(int* p)
{
        int x = 4;
        owner<int*> q = new int{7};
        vector<int*> res = {p, &x, q, &glob};          // potentially bad: mix ownership
        return res;                                    // Bad: return {unknown, &local, &owner, &global}
}
```

- Don't mix different ownerships in an array
  - If you must, encapsulate
- Don't let different return statements mix ownership

Ownership and pointers

- Owners are a tree
  - Except for **shared_ptr**: a DAG
  - Simple
  - efficient
  - Minimal resource retention
  - No ownership cycles
- Owners can be invalidated
  - Catch simple cases at compile time
  - Use **shared_ptr** and/or **nullptr** checks for not-so-simple cases
- Pointers
  - can only refer to live objects
    - To objects with a live owner
    - To objects "back or to the same level" in a stack
  - can have cycles

Research problem:
How do you represent
a safe, general, and efficient
Graph?

Concurrency

- Use scopes and **shared_ptr** to keep threads alive as needed
- A tread is a container (of pointers)
  - The usual rules for containers of pointers apply
  - **std::tread**
    - May or may not outlive its scope
      - Bad
      - we must conservatively assume that it lives forever
  - **gsl::joining_thread**
    - Joins
      - so it is a local container
    - **stl:jthread**?
- CP.26: Don't **detach()** a thread
  - If you do, you lose lifetime information

Owner invalidation

- Some cases are simple

```
void f()
{
	auto p = new int{7};
	delete p;		// invalidate p
	*p = 9;			// bad: must be prevented
}
```

- Such examples can be handled by static analysis
  - Avoid “naked new” and “naked delete”

Owner invalidation

- Some cases are less simple

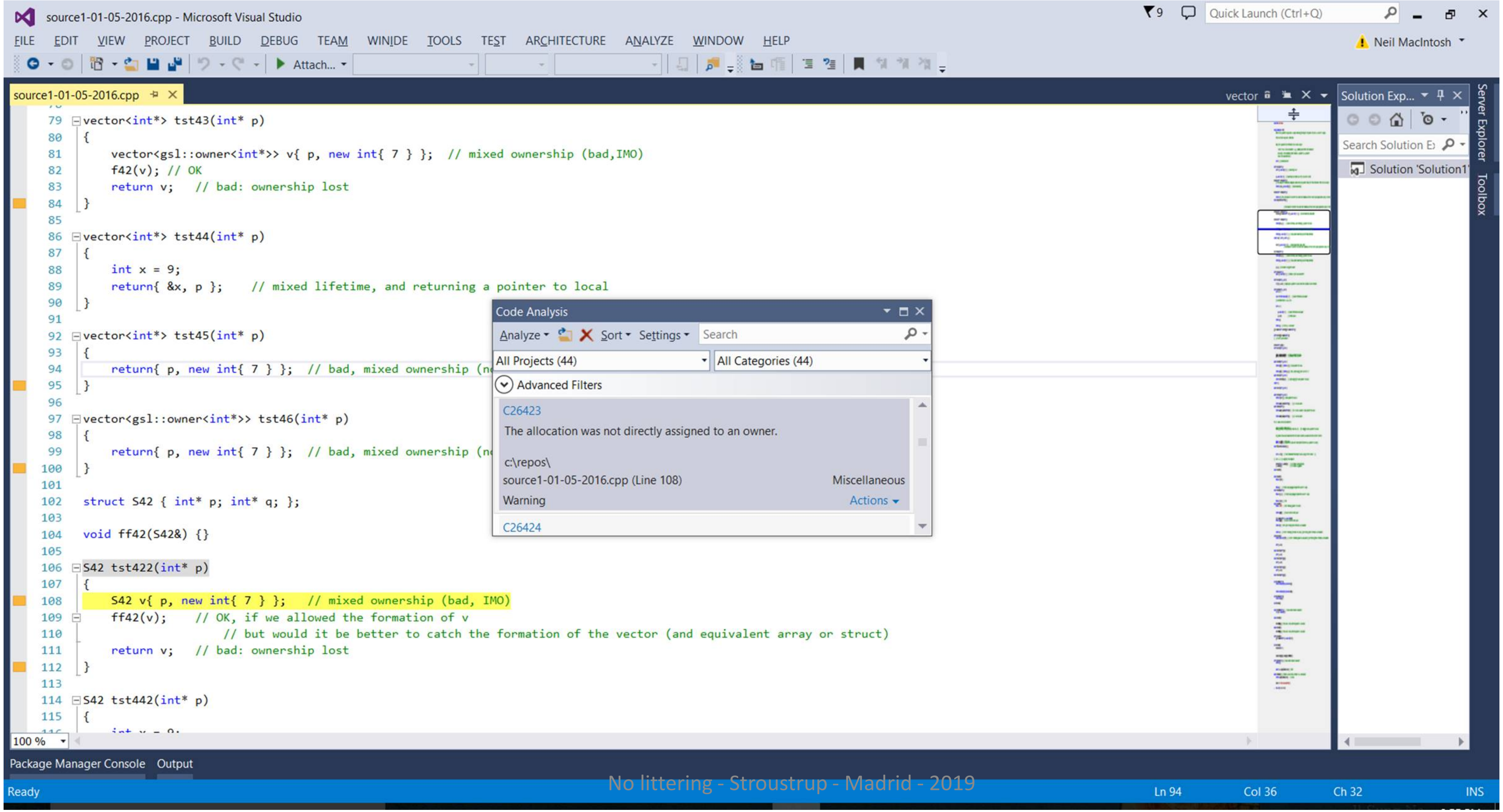
```
void g(int* q) { *q = 9; }          // looks innocent

void f()
{
        vecor<int> v {7};
        gsl::joining_thread(g,&v[0]);
        v.push_back(11);            // invalidates q
        // …
}
```

- Such examples can be handled by static analysis
  - Avoid unscoped threads
  - In an emergency, use **shared_ptr** to defeat “false positives”

Why not "just use smart pointers"?

- Complexity and (sometimes) cost
  - E.g., different versions of functions for different kinds of pointers
- Use only when you need to manipulate ownership
  - **unique_ptr** for unique ownership
    - guard against exceptions
    - Return pointer-to-base in OOP
  - **shared_ptr** for shared ownership
    - For cases where you can't identify a single owner
    - ***Not*** for guarding against exceptions
    - ***Not*** for returning objects from the free store
    - More expensive that raw pointers – use counts
    - Can led to need for **weak_ptr**s
    - Can lead to "GC delays"
- Remember
  - Local variables (e.g., handles)
  - Move semantics

Static analysis (integrated)



Dangling pointer summary

- Simple:
    - Never let a “pointer” escape to where it can refer to its object after that object is destroyed
- It’s not just pointers
    - All ways of “escaping”
        - **return**, **throw**, place in long-lived container, threads, …
    - Same for containers of pointers
        - E.g. **vector<int*>**, **unique_ptr<int>**, **thread**s, iterators, built-in arrays, …
    - Same for references
- We need a formal paper/proof
- We need to demonstrate scaling
    - 1M line code bases (some examples done)

Other problems

- Other ways of breaking the type system
  - Unions: use **std::variant**
  - Casts: don't use them outside abstractions
  - …
- Other ways of misusing pointers
  - Range errors: use **gsl::span<T>**
  - **nullptr** dereferencing: use **gsl::not_null<T>**
- Wasteful ways of addressing pointer problems
  - Misuse of smart pointers
- …
- "Just test everywhere at run time" is ***not*** an acceptable answer
  - We want comprehensive guidelines

gsl::span<T>

- Common interface style
  - major source of bugs

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

gsl::span<T>

- Common style

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

- Better

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

- "Make simple things simple"
  - Simpler than "old style"
  - Shorter
  - At least as fast

nullptr problems

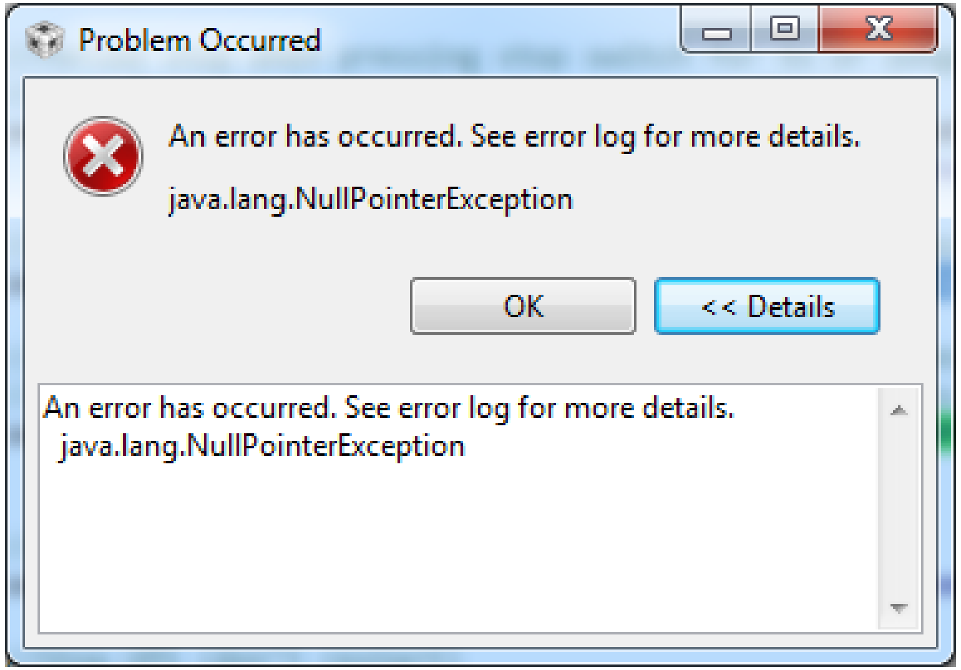
- Mixing **nullptr** and pointers to objects
  - Causes confusion
  - Requires (systematic) checking
- Caller

```
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 be 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
    - Should it be an alias like **owner**?
  - **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**

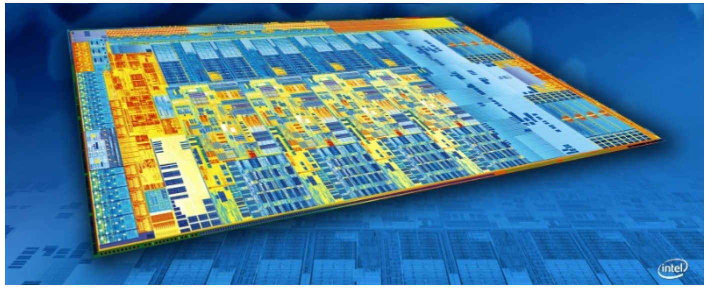
To summarize

- Type and resource safety:
  - RAII (scoped objects with constructors and destructors)
  - No dangling pointers
  - No leaks (track ownership pointers)
  - Eliminate range errors
  - Eliminate **nullptr** dereference
- That done, we attack other sources of problems
  - Logic errors
  - Performance bugs
  - Maintenance hazards
  - Verbosity
  - …

We are not unambitious (rough seas ahead)

- 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

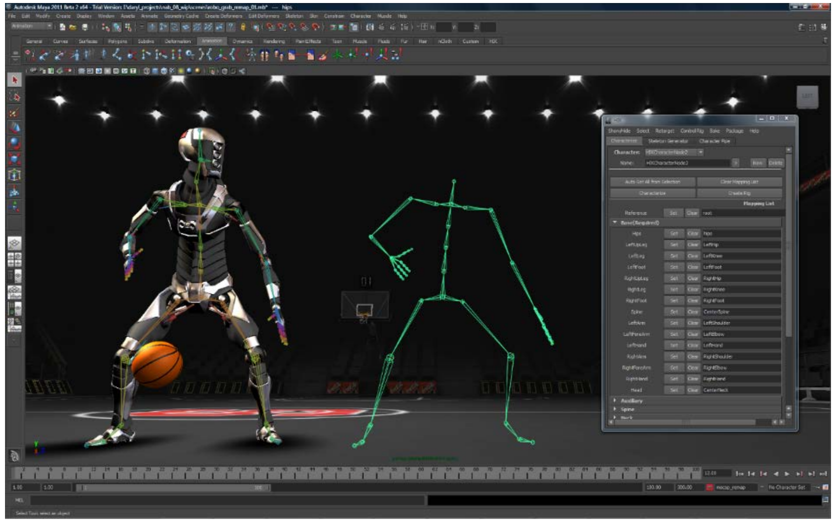
Questions?



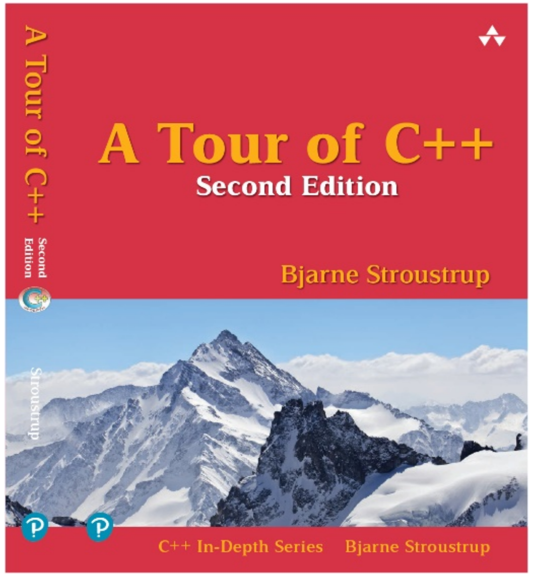
- Type- and Resource-safe C++
  - No garbage collector (because there is no garbage to collect)
  - No runtime overheads (Except necessary range checks)
  - No new limits on expressibility
  - ISO C++
  - Simpler code

Current state: the game is changing dramatically

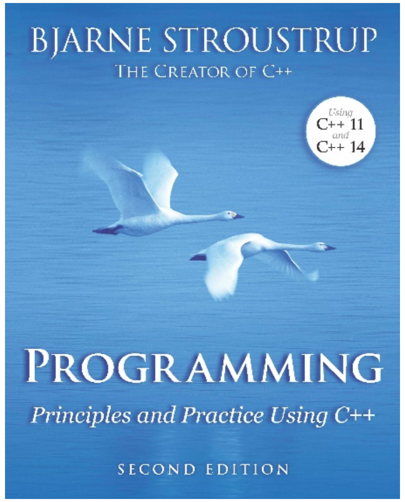
- Papers
  - B. Stroustrup, H. Sutter, G. Dos Reis: A brief introduction to C++'s model for type- and resource-safety.
  - H. Sutter and N. MacIntosh: Preventing Leaks and Dangling
  - T. Ramananandro, G. Dos Reis, X Leroy: A Mechanized Semantics for C++ Object Construction and Destruction, with Applications to Resource Management
- Code (MIT license)
  - https://github.com/isocpp/CppCoreGuidelines
  - https://github.com/microsoft/gsl
  - Static analysis: experimental versions available (Microsoft)
- Videos
  - B. Stroustrup: : Writing Good C++ 14
  - H. Sutter: Writing good C++ 14 By Default
  - G. Dos Reis: Contracts for Dependable C++
  - N. MacIntosh: Static analysis and C++: more than lint
  - N. MacIntosh: A few good types: Evolving array_view and string_view for safe C++ code

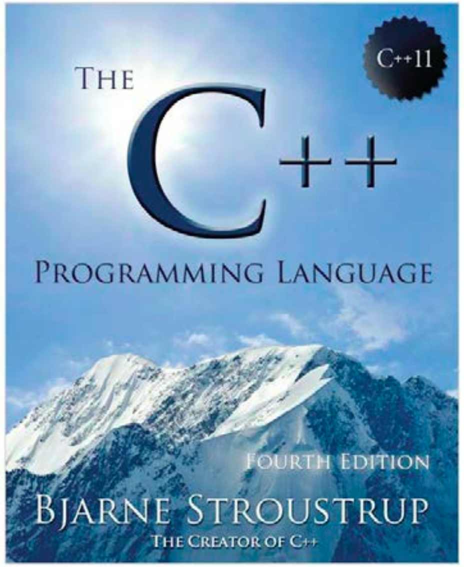


C++ Information



- The C++ Foundation: www.isocpp.org
  - Standards information, articles, user-group information
- Bjarne Stroustrup: www.stroustrup.com
  - Publication list, C++ libraries, FAQs, etc.
  - *A Tour of C++ (2nd edition)*: All of C++ in 240 pages
  - *The C++ Programming Language (4th edition)*: All of C++ in 1,300 pages
  - *Programming: Principles and Practice using C++ (2nd edition)*: A textbook
- The ISO C++ Standards Committee: www.open-std.org/jtc1/sc22/wg21/
  - All committee documents (incl. proposals)





Videos

- Cppcon: https://www.youtube.com/user/CppCon 2014, 2015
- Going Native: http://channel9.msdn.com/Events/GoingNative/ 2012, 2013

Guidelines: https://github.com/isocpp/CppCoreGuidelines