No Littering!

Bjarne Stroustrup
Morgan Stanley
Columbia University, Texas A&M University
www.stroustrup.com
The big question

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

• Guidelines project
  • https://github.com/isocpp/CppCoreGuidelines
  • Produce a useful answer
  • Enable many people to use that answer
    • For most programmers, not just language experts
  • Please help!
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 and nullptr checking
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
  • 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` (don’t dereference)
  • be unitialized (don’t dereference)
No littering, no leaks, no corruption

- Every object is constructed before use
  - Once only
  - initialized

- Every fully constructed object is destroyed
  - Once only
  - Every object allocated by `new` must be `deleted`
  - No scoped object must be `deleted` (it is implicitly destroyed)

- No access through a pointer that is not pointing to an object
  - Read or write
  - Off the end of an object (out of range)
  - To `deleted` object
  - To “random” place in memory (e.g., uninitialized pointer)
  - Through `nullptr` (originally: “there is no object at address zero”)
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
  - Is non-local (systems are often distributed)
  - Introduces non-predictability
- Static analysis doesn’t scale
  - False positives
  - Dynamic linking and other dynamic phenomena
A solution

- Be precise about ownership
  - Don’t litter
  - Static guarantee

- Eliminate dangling pointers
  - Static guarantee

- Make general resource management implicit
  - Hide every explicit delete/destroy/close/release

- Test for `nullptr` and range
  - Do minimal run-time checking

- There are other problems with C++ pointers
  - Dealt with by other rules
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

• Part of C++ Core Coding guidelines
  • Supported by a “guidelines support library” (GSL)
  • Supported by analysis tools
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`”
  • Constructor/destructor
    • “since 1979, and still the best”
Dangling pointers – the worst problem

• One nasty variant of the problem

```c
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 every attempt 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

• 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
Dangling pointers

• Ensure that no pointer outlives the object it points to

```cpp
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(); // Make sure we never get here
}
```
How do we represent ownership?

• High-level: Use an ownership abstraction
  • This is simple and preferred
  • E.g., `unique_ptr`, `shared_ptr`, `vector`, and `map`

• Low-level: mark owning pointers **owner**
  • An **owner** must be deleted or passed to another **owner**
  • A non-**owner** may not be deleted
  • This is essential in places but does not scale
  • Applies to both pointers and references

• **owner** is intended to simplify static analysis
  • **owners** in application code is a sign of a problem
    • Usually, C-style interfaces
  • “Lots of annotations” doesn’t scale
    • Becomes a source of errors
How do we represent ownership?

- Mark an owning `T*`: `owner<T*>`
  - Initial idea
    - 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
- So our 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: 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
   };

• owner<T*> is just an alias for T*
GSL: owner<T>

• How about code we cannot change?
  • ABI stability

```c
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
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
    • Essentially to extend the type system

• Each of those techniques are insufficient by itself
• Not just for C++
  • But the “cocktail” relies on much of C++
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

```cpp
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, should return an owner<int*>  
    return p;  // OK: came from caller
}
```
How to avoid/catch dangling pointers

• Classify pointers according to ownership
  
  ```cpp
  vector<int*> f(int* p)
  {
    int x = 4;
    owner<int*> q = new int{7};
    vector<int*> res = {p, &x, q};  // Bad: { unknown, pointer to local, owner }
    return res;
  }
  ```

• Here
  
  • 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

• Concurrency
  • Keep threads alive with scoping or shared_ptr
  • Apply the usual rules for a thread’s stack
  • Threat another thread as just another object (it is)
Other problems

• Other ways of breaking the type system
  • Unions: use variant
  • Casts: don’t use them
  • …

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

```cpp
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

```cpp
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

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

• Better

```c
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 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
    - 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
  • ...

No littering - Stroustrup - TAMU - March’16
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
  • https://github.com/isocpp/CppCoreGuidelines
  • https://github.com/microsoft/gsl
  • Static analysis: coming soon (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](http://www.isocpp.org)
  - Standards information, articles, user-group information
- Bjarne Stroustrup: [www.stroustrup.com](http://www.stroustrup.com)
  - Publication list, C++ libraries, FAQs, etc.
  - *A Tour of C++*: All of C++ in 180 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/](http://www.open-std.org/jtc1/sc22/wg21/)
  - All committee documents (incl. proposals)

Videos
- Cppcon: [https://www.youtube.com/user/CppCon 2014](https://www.youtube.com/user/CppCon 2014), [2015](https://www.youtube.com/user/CppCon2015)

Guidelines: [https://github.com/isocpp/CppCoreGuidelines](https://github.com/isocpp/CppCoreGuidelines)
(Mis)uses of smart pointers

• “Smart pointers” are popular
  • To represent ownership
  • To avoid dangling pointers

• “Smart pointers” are overused
  • Can be expensive
    • E.g., `shared_ptr`
  • Can mess up interfaces for otherwise simple functions
    • E.g., `unique_ptr` and `shared_ptr`
  • Often, we don’t need a pointer
    • Scoped objects
    • We need pointers
      • For OO interfaces
      • When we need to change the object referred to

But ordinary pointers don’t dangle any more
(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 (uncommon style)
  • `void f(shared_ptr<T*>);`  // share ownership and use (implies cost)

• 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 (now!)
  • 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, `span`, etc.
Rules, standards, and libraries

• Could the rules be enforced by the compiler?
  • Some could, but we want to use the rules now
    • Some compiler support would be very nice; let’s talk
  • 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
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
The basic C++ model is now complete

• C++ (using the guidelines) is type safe and resource safe
  • Which other language can claim that?
  • 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
To do / being done

• Implementations
  • Static analysis: Microsoft (coming soon), Clang (starting), GCC (?)
  • Support library (GSL):
• Technical specification
• Coding Guidelines
• Popular explanations
• Academic explanations
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; Windows, Linux, Mac)
  • Analysis tool (Microsoft shipping in November; ports later (November?))
  • MIT License

• Related CppCon talks (available on video)
  • Herb Sutter: Writing Good C++14 By Default (focused on safety)
  • Gabriel Dos Reis: Modules (fast compilation, no macro problems)
  • Gabriel Dos Reis: Contracts (for documentation, run-time checking, and safety)
  • Neil MacIntosh: Static analysis (no dangling pointers, no leaks)
  • Neil MacIntosh: span, string_span, etc. (GSL)