Preface

• These are the slides I (Bjarne Stroustrup) presented to the Safety Study Group (SG23) and the Evolution Working Group (EWG) at the February 2023 C++ Standard Committee (ISO SC22/WG21) meeting in Issaquah, Washington State, USA.

• The purpose of my talks was to build a consensus for a direction to allow dramatically improved safety for C++ programs without damaging performance, flexibility, or compatibility where needed. The resulting vote was 47 for and 2 against.

• Please note that this presents a direction/strategy, rather than a completed product. However, it is based on significant previous work; see the references. More experiments and more documentation are in the works.

• A safety profile is a set of guarantees enforced by an implementation. A profile presents the programmer with a set of rules and library components that together delivers the desired guarantees.
Safety Profiles:
Type-and-resource Safe programming in ISO Standard C++

Bjarne Stroustrup
Columbia University
www.stroustrup.com

Gabriel Dos Reis
Microsoft
Abstract – Safety Profiles

• **Type-and-resource Safe programming in ISO Standard C++**
  - You can write C++ with no violations of the type system, no resource leaks, no memory corruption, no garbage collector, no limitation of expressiveness or performance degradation compared to well-written modern C++.
  - We must develop ways of guaranteeing that where guarantees make sense.
  - This can be achieved – and guaranteed – by the applying the strategy from the C++ Core Guidelines: coding rules, simple supporting libraries (mostly the ISC C++ standard library), and enforcement through static analysis.
  - Doing this well requires some standardization and some standardized support: Safety Profiles.
  - Often, this can be done with code that’s dramatically simpler than older C++ (and C) code.
  - Examples: RAII, pointer safety, span, range checking, nullptr, initialization, invalidation, casting and variants.
A cause for concern (not panic)

• The overarching software community across the private sector, academia, and the U.S. Government have begun initiatives to drive the culture of software development towards utilizing memory safe languages.

• ...

• NSA advises organizations to consider making a strategic shift from programming languages that provide little or no inherent memory protection, such as C/C++, to a memory safe language when possible. Some examples of memory safe languages are C#, Go, Java, Ruby™, and Swift®.
  • NSA: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2739r0.pdf
To contrast (not a cause for complacency)

• **February Headline: C++ still unstoppable**
  
  Last month, C++ won the TIOBE programming language of the year award for 2022. C++ is continuing its success in 2023 so far. Its current year-over-year increase is 5.93%. This is far ahead of all other programming languages, of which the most popular ones only gain about 1%.

<table>
<thead>
<tr>
<th>Feb 2023</th>
<th>Feb 2022</th>
<th>Change</th>
<th>Programming Language</th>
<th>Ratings</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>Python</td>
<td>15.48%</td>
<td>+0.16%</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td>C</td>
<td>15.39%</td>
<td>+1.31%</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>↑</td>
<td>C++</td>
<td>13.94%</td>
<td>+5.93%</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>↓</td>
<td>Java</td>
<td>13.21%</td>
<td>+1.07%</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td>C#</td>
<td>6.38%</td>
<td>+1.01%</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
<td>Visual Basic</td>
<td>4.14%</td>
<td>-1.09%</td>
</tr>
</tbody>
</table>

• But what does Tiobe measure?
• But this implies that what we do matters to billions of people – for good and bad
We must address the “safety” issue

- There is a real, serious problem for many uses and users
  - Incl. diversion of resources to other languages
  - Incl. discouraging people from learning C++

- Massive improvements are possible in many areas

- C++ has a massive image problem (“C/C++”)  
  - And it is getting worse

- Governments and large corporations can coerce

- Ignoring the safety issues would hurt large sections of the C++ community and undermine much of the other work we are doing to improve C++.
  - So would focusing exclusively on safety
References

- B. Stroustrup: A call to action: Think seriously about “safety”; then do something sensible about it. P2739R0 2022-12-6.
- B. Stroustrup and G. Dos Reis: Design Alternatives for Type-and-Resource Safe C++. P2687R0. 2022-20-15
- B. Stroustrup: Type-and-resource safety in modern C++. P2410r0. 2021-07-12.
- B. Stroustrup: Writing Good C++14 CppCon 2015.
- The C++ Core Guidelines
- The Core Guidelines Support Library (GSL)
- A Microsoft guide to using the Core Guidelines static analyzer in Visual Studio.
- T. Ramananandro, G. Dos Reis, and X. Leroy: A mechanized semantics for C++ object construction and destruction, with applications to resource management. ACM/SIGPLAN Notices 2012/01/18.
Complete type-and-resource safety

• Is an ideal (aim) of C++
  • From very early on (1979)
  • Also for C: “C is a strongly-typed, weakly checked language” a – DMR
  • “Being careful” doesn’t scale

• Requires judicious programming techniques
  • Supported by libraries
  • Enforced by language rules and static analysis
  • The basic model for achieving that can be found in A brief introduction to C++'s model for type- and resource-safety (2015) and Type-and-resource safety in modern C++ (2021).

• Does not imply limitations of what can be expressed or run-time overhead
  • Compared to traditional C and C++ programming techniques
How?

- Every object is accessed according to the type with which it was defined (type safety)
- Every object is properly constructed and destroyed (resource safety)
- Every pointer either points to a valid object or is the `nullptr` (memory safety)
- Every reference through a pointer is not through the `nullptr` (often a run-time check)
- Every access through a subscripted pointer is in-range (often a run-time check)

- That is
  - just what C++ requires (also C)
  - what most programmers have tried to ensure since the dawn of time

- The enforcement rules are more deduced than invented

Enforcement rules are mutually dependent. Don’t judge individual rules in isolation
Many notions of safety

- **Logic errors**: perfectly legal constructs that don’t reflect the programmer’s intent, such as using `<` where `a <=` or `a >` was intended.
- **Resource leaks**: failing to delete resources (e.g., memory, file handles, and locks) potentially leading to the program grinding to a halt because of lack of available resources.
- **Concurrency errors**: failing to correctly take current activities into account leading to (typically) obscure problems (such as data races and deadlocks).
- **Memory corruption**: for example, through the result of a range error or by accessing and memory through a pointer to an object that no longer exists thereby changing a different object.
- **Type errors**: for example, using the result of an inappropriate cast or accessing a union through a member different from the one through which it was written.
- **Overflows and unanticipated conversions**: For example, an unanticipated wraparound of an unsigned integer loop variable or a narrowing conversion.
- **Timing errors**: for example, delivering a result in 1.2ms to a device supposedly responding to an external event in 1ms.
- **Termination errors**: a library that terminates in case of “unanticipated conditions” being part of a program that is not allowed to unconditionally terminate.
Constraints on a solution

- C++ must serve wide variety of users/areas
  - One size doesn’t fit all
  - C++ is (also) a systems programming language – we can’t “outsource” dangerous operations to some other language

- We can’t just break billions of lines of existing code
  - Even if we wanted to - major users would insist on compatibility (probably compatibility by default)

- We can’t just “upgrade” millions of developers
  - And teaching material, courses, videos, books, articles

- If you want a shiny new language, please go ahead
  - But it won’t be C++ or the job of WG21

- But we **must** improve
Strategy: Safety Profiles

• We can succeed only if we have a strategy/framework
  • A framework for “details” to fit into
  • Ad hoc, independent “patches” won’t add up to a coherent, complete solution ("Safety")
    • Even if those “patches” can be immensely useful

• We – WG21 – must be seen to work towards a coherent solution
  • A complete solution will take significant time
  • Until then then, we must be able to point to steady progress
  • Until then then, we must deliver partial solutions
Strategy: Safety Profiles

- Our approach is “a cocktail of techniques” not a single neat miracle cure

- Static analysis
  - to verify that no unsafe code is executed.

- Coding rules
  - to simplify the code to make industrial-scale static analysis feasible.

- Libraries
  - to make such simplified code reasonably easy to write
  - to guarantee run-time checks where needed.
Strategy: Safety Profiles

• This is a strategy
  • Not a finished product
  • Based on significant previous work
    • The C++ Core guidelines (on GitHub)
    • Static checkers
    • Library design
    • and more
Is this strategy “too novel”?  

• “People are afraid of new things.
  You should have taken an existing product and put a clock in it.”
  – Homer Simpson

• Parts have been tried each individual approach many times before
  • Succeeded for specific tasks
    • E.g., smart pointers, libraries, static analyzers
  • Failed as general solutions
    • Static analysis – doesn’t scale to complete safety
    • Guidelines/rules – aren’t followed without enforcement
    • Foundation libraries – doesn’t give full access to the machine and system
    • Language subsetting – the most dangerous language features are essential (e.g., subscripting of pointers)

• A combined approach is necessary
  • Similar to Ada’s safety profiles: https://docs.adacore.com/gnathie_ug-docs/html/gnathie_ug/gnathie_ug/the_predefined_profiles.html#the-predefined-profiles
The C++ Core Guidelines

• For any reasonable definition of safe
  • We cannot accept arbitrarily complex code while maintaining conventional good performance.
  • Legal != provably safe

• Use a carefully crafted set of programming techniques
  • supported by library facilities
  • enforced by static analysis.

• Available on GitHub
  • https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

• Many rules checked by the Visual Studio static analyzer and other checkers (Clang, Clion)

• Safety Profiles must go beyond the CG
  • We need some standardization of what’s to be checked
  • We need some annotations in the code to specify what is to be checked

Caveat: The Core Guidelines does a good, but not complete, job

This talk discusses potential improvements to gain guarantees
C++ Core Guidelines

• You can write type-and-resource-safe C++
  • No leaks
  • No memory corruption
  • No garbage collector
  • No limitation of expressibility
  • No performance degradation
  • ISO C++
  • Tool enforced (eventually)
    • some checking in Visual Studio, Clang tidy, Clion, ...
    • A job for the tools SG? For the new Safety SG? Both!
  • For guaranteed type safety, we need range checking and nullptr checking

Safety Profiles go beyond the CG, e.g., safety requirements in code (slide 24)
- Narrowing conversions and overflow
  - E.g., signed and unsigned mess-ups
- Data races and deadlocks
- Logic errors
  - Including error-prone constructs; e.g., misspelling, missing breaks, and overly complex code
- Performance bugs
  - E.g., copying of large objects, allocations in time-critical code
- Some of this is covered by the Core Guidelines and in existing checkers
  - Rarely systematically
  - Guarantees require systematic checking
Fundamental ideas

• Core Guidelines
  • **P.1: Express ideas directly in code**
  • **P.9: Don't waste time or space**
  • **P.11: Encapsulate messy constructs, rather than spreading through the code**

• Safety Profiles: Beyond (current) Core Guidelines
  • If (local) static analysis cannot prove a construct safe, it’s banned
  • Annotations and run-time checks to enforce guarantees

• Rules should help, not hinder
  • No non-essential restrictions on coding style

---

Don’t destroy maintainability by lowering the abstraction level to a subset of C

Suggestions and help most welcome
High-level rules – “Philosophy”

• 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
  • 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
Lower-level rules

• Provide enforcement
  • Many rely on static analysis
  • Some beyond our current tools
  • Often easy to check “mechanically”

• Primarily for tools (static analysis)
  • To allow specific feedback to programmer

• Help to unify style

• Not minimal or orthogonal

  • **F.16**: Use `T*` or `owner<T*>` to designate a single object
  • **C.49**: Prefer initialization to assignment in constructors
  • **ES.20**: Always initialize an object
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)
    - Messy/dangerous/low-level features can be used to implement the GSL
  - For complete memory safety, enforce range-checking
  - Then subset
- What we want is "C++ on steroids"
  - Simple, safe, flexible, and fast

Use:
- C++
- GSL
- STL

Don’t use

No change of meaning: The resulting code is ISO C++
Some rules rely on libraries

- The ISO C++ standard library
  - E.g., `vector<T>` and `unique_ptr<T>`
- The Guideline Support Library
  - E.g., `span<T>` and `not_null<T>`

- Some rules using the GSL and STL
  - I.11: Never transfer ownership by a raw pointer (`T*`)
    - Use an ownership pointer (e.g., `unique_ptr<T>`) or `owner<T*>`
  - I.12: Declare a pointer that may not be the `nullptr` as `not_null`
    - E.g., `not_null<int*>`
  - I.13 Do not pass an array as a single pointer
    - Use a handle type, e.g., `vector<T>` or `span<T>`

Ideally, absorb the GSL functionality into the standard
Static analysis

• If local static analysis cannot prove a construct safe, it’s banned

• To scale, static analysis must be local
  • Constructors and destructors must be considered together

• We need rules to simplify to allow local analysis
  • It is easy to write messy code that cannot be statically determined to be safe
  • Classify: safe, not safe, not sure
  • Reject if not sure ("call a human")

See also Sunny Chatterjee’s CppCon’21 talk on static analysis of C++
Profile controls

• The (current) Core Guidelines are controlled using compiler/build options

• Some users and some organizations insist on annotations in the code
  • E.g., “This code type-and-resource safe.”
    • Must be enforced
  • E.g., “this is unverified, trusted code”
    • Something like that is in every “safe” language

• Maybe also compiler/build options

• How do we handle programs composed out of fragments with different requirements?
  • Very difficult problem
  • Unavoidable (in any language)
  • See P2687R0 (Stroustrup and Dos Reis 2022)
Examples

• Type safety
  • Initialization, construction, destruction

• Pointer safety
  • Every pointer points to a valid object or is the nullptr

• Ownership
  • No littering

• Invalidation
  • Aliases

• Run-time checks
  • Span, not_null, not_end

• Memory pools
  • A tricky set of problems

• Concurrency rules

For a much more detailed paper: P2687R0 and in particular P2687R1 (in the works)
Every object is accessed according to the type with which it was defined

- **ES.20: Always initialize an object**
  - To a meaningful value
  - Just “zero out all objects” isn’t enough

- Prevent
  - Unsafe casting
    - Restrict casting to converting untyped data (bytes) into typed objects
    - `dynamic_cast` is safe and accepted
  - Unsafe uses of `unions`
    - Use alternatives, e.g., `variant`
  - Unsafe use of pointers
    - E.g., subscripting
    - Use alternatives, e.g., `span`
Always initialize an object

```cpp
int f(int x)
{
    int y; // Not OK: uninitialized
    if (x) y = g(x);
    return y;
}
```

```cpp
Message read(int n) // we need buffers for low-level input: std::byte
{
    [[uninitialized]] std::byte buf[n]; // uninitialized buffer
    return fill_message(buf, n); // fill and convert to correct type
    // (had better check the value of n)
}
```

Yes, we could be more clever, but simplicity is valuable

Yes, we could close all loopholes, but for real-world problems we can’t be purists
Every object is properly constructed and destroyed

- **P.8: Don't leak any resources**
- Resources – Entities that must be acquired and later released
  - represented as objects with destructors doing the release
  - often with constructors that do the acquisition as part of establishing an invariant (RAII)
  - This scope-based resource management ensures predictability and minimizes resource retention
- The language guarantees that destructors are invoked
  - Except for objects pointed to only by static variables
- Using copy elision or move operations, objects can be safely moved between scopes.
  - Moved objects will be destroyed in their new scope or moved further
  - The CG insist that a moved-from object be assignable
- Prevent the creation of uninitialized objects
  - Buffers of uninitialized **unsigned chars** are acceptable
Every pointer either points to an object or is the nullptr:

- Aka “no dangling pointers”

- When I say “pointer” I mean anything that refers to an object
  - References
  - Containers of pointers
  - Smart pointers
  - Lambda captures of pointers
  - ...

Turning simple logical rules into detailed enforceable rules is a lot of hard work.
Dangling pointers – the worst problem

• One nasty variant of the problem

```cpp
void f(X* p)
{
    // ...
    delete p; // looks innocent enough (not OK)
}

void g()
{
    X* q = new X; // looks innocent enough (not OK)
    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
  • memory safety is compromised
  • resource safety is compromised

• Eliminated by a combination of rules
  • Distinguish owners from non-owners
    • Annotation `gsl::owner<int*>`
    • Something that holds an owner is an owner
    • Don’t forget `malloc()`, etc.
  • 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, lambda captures, long-lived containers, ...
Dangling pointer rules

• A pointer can be returned from a scope *iff*
  • It was passed into the scope (e.g., as an argument or retrieved from an object external to the scope)
  • It points to an object external to the scope (e.g., it was initialized by `new`)

• If static analysis cannot prove that, the pointer cannot be returned
  • This implies limitations to the complexity of the flow of control leading to the return of a pointer value

*Ownership and invalidation rules guarantee that pointers points to an object or are the nullptr

A problem, given multiple analyzers:
How clever should an analyzer required to be?
Example: Pointer to deleted object

```c
int* f()
{
    int* p = new int {7};
    int* q = p;
    delete p;
    *q = 9;       // not OK: detected by local static analysis
    return q;     // not OK: returning pointer to deleted object
}
```

We need to address aliasing in general

Static analysis must involve flow analysis
Example: Escaping pointers

```c
int glob = 9;
int* glob2 = &glob;  // OK: global pointer to global

int* confused(int i, int* arg)
{
    int loc = 0;

    switch (i) {
    case 1:  return &loc;  // not OK: pointer to local
    case 2:  return new int{7}; // OK: pointer to free-store object (but ownership problem)
    case 3:  return &glob;  // OK: pointer to global
    case 4:  return arg;    // OK: returning what we received as an argument
    case 5:  return glob2;  // OK: returning what someone stored globally
    }
}
```
ES.65: Don't dereference an invalid pointer

- A pointer can be made not dereferenceable in several ways:
  - Uninitialized
    - Forget to initialize a pointer
  - Dangling pointer
    - Point to an object after it has gone out of scope (e.g., return a pointer to a local variable from a function)
    - Retain a pointer to a deleted object.
    - Violate the type system by placing a value that does not refer to an object into a pointer (e.g., a cast, misuse of union, or range error)
  - Invalidation
    - Retain a pointer to an object that has been deleted or moved (so that the object pointed to have been deleted or now hold a value that is logically different from the one expected).
  - Be the `nullptr`.
  - Point to one-past-the-end of a sequence (such a pointer may not be dereferenced).
Ownership

- An owner is something responsible for invoking a destructor
  - A scope
  - An object
- Something holding an owner is an owner
  - Container (vector, map, array, pointer to pointer, ...)
- `operator new` returns an owner
- Ownership annotation
  - `template<typename T> using owner = T;`
  - Used by static analysis
  - Useful in code reviews
  - Doesn’t affect ABI
Prefer ownership abstractions

• Such as
  • `vector`, `map`, `unique_ptr`, `fstream`, `jthread`, ...

• **owner** annotations is for
  • Implementation of ownership abstractions
    • E.g., `vector`, `map`, `unique_ptr`, `fstream`, `jthread`, ...
  • Avoiding ABI breaks
    • E.g., C-style functions with pointers

• **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
Low-level ownership rules

• To keep static analysis local, use `gsl::owner` annotations
  • A pointer returned by `new` is an `owner` and must be `deleted`
    • unless stored in static storage to ensure that it lives “forever”
  • Only a pointer known to be an `owner` can be `deleted`
    • a pointer passed into a scope as an `owner` must be `deleted` in that scope or passed to another scope as an `owner`.
  • A pointer passed to another scope as an `owner` and not passed back as an `owner` is invalidated
    • cannot be used again in its original scope (since it will have been `deleted`).
Example: Ownership

```c
void f(int* pp)
{
    // ...
    delete pp; // Not OK: can’t delete non-owner
    // ...
}

void use();
{
    int* p = new int{99}; // Not OK: assigns owner to non-owner
    // ...
    f(p);
    // ...
    *p = 7; // dangling pointer; but we’ll never get here
}
Example: Ownership

```cpp
void f(owner<int*> pp)
{
    // ...
    delete pp;               // OK: f() must delete owner (or pass it along)
    // ...
}

void use();
{
    owner<int*> p = new int{99}; // OK: assigns owner to owner
    // ...
    f(p);                        // OK: pass owner along
    // ...
    *p = 7;                      // dangling pointer; but we’ll never get here
}
```
Example: ownership abstraction implementation

• How do we implement ownership abstractions?
  
  template<semiregular T>
  class vector {
  public:
    // …
  private:
    owner<T*> elem;  // this anchors the allocated memory
    T* space;      // just a position indicator
    T* end;        // just a position indicator
    // …
  }
  // …

• owner<T*> is just an alias for T*
Invalidation

- Any operation that may reallocate the elements of a container invalidates all operations on it
- Deleting a container invalidates all operations on it
  - See “ownership”

- Container: anything that holds a pointer
  - Classes with pointer members
  - Lambdas (they are classes, and remember capture-by-reference)
  - Pointers to pointers
  - References to pointers
  - Arrays of pointers
  - `unique_ptr` and `shared_ptr`
  - Threads with pointer arguments

```cpp
int x = 7;
int* p = &x;
int** pp = &p;
cout << **pp;  // 7
p = nullptr;
cout << **pp;  // not OK
```
Example: Invalidation

```cpp
void f(vector<int>& vi)
{
    vi.push_back(9); // may relocate vi’s elements
}

void g()
{
    vector<int> vi { 1,2 }; // point to first element of vi
    auto p = vi.begin();
    f(vi);
    *p = 7; // not OK, may appear to work correctly
}
```
Invalidation

• Currently
  • Any non-\texttt{const} member function invalidates (see [Sutter’19])
  • Any function with a non-\texttt{const} pointer argument invalidates
  • Note: Con.2: By default, make member functions \texttt{const}

• Suggested (not implemented)
  • The current rule is simple and safe, but overly conservative
  • Some important functions are not non-const yet don’t invalidate
    • E.g., \texttt{vector::swap()} and \texttt{vector::operator[]()}
  • Mark those \texttt{[[not_invalidating]]}
    • \texttt{[[not_invalidating]]} is a testable optimization of a safe default
  • ES.2x: Don’t use pointers to pointers or references to pointers
Example: Invalidation

• Aliasing problems are subtle
  • Best left to tools (compilers, static analyzers)

• Consider
  • `vec.insert(vec.begin(), vec.front());`  // OK, guaranteed by standard
    // (may have to copy *vec.front())
  • `vec.insert(vec.begin(), {4,5,6});`  // OK, add a range
  • `vec.insert(vec.begin(), vec.begin(), vec.end());`  // likely disaster: for some allowed implementations
    // allocate more space for elements
    // copy old elements
    // delete old allocation
    // copy elements from old allocation
    // caught by CG invalidation check
  • `lst.insert(lst.begin(), lst.begin(), lst.end());`  // OK (lists don’t relocate elements)
Library support

- **gsl::dynarray**
  - Like `vector`, but no resizing
  - No invalidation

- **std::unique_ptr** (and `std::shared_ptr`)
  - Use static analysis to prevent `get()` or introduce `gsl::unique_ptr`
  - Prevent unnecessary aliasing

- In general, we may need restricted versions of library facilities to simplify static analysis
Low-level code

• C++ is extensively used for low-level manipulation of memory and other system resources
  • Making C++ safe by eliminating all direct access to “raw” memory is not an option
  • Languages that ban such unsafe access, typically have ways of allowing unsafe code
    or delegate such manipulation to code written in C or C++.

• Currently
  • Selective use of static analysis
    • E.g., CG “profiles”

• Suggestion (unimplemented)
  • Annotate necessarily messy code [[unverified]]
    • E.g., fundamental data structures, concurrency primitives, etc.
    • Possibly using “profiles” [[unverified lifetime]]
  • Discourage use of [[unverified]]
    • It will be overused
Some run-time checks are unavoidable

• Access that depend on values not known until run time
  • `nullptr`
    • Use `gsl::not_null`
  • Range errors
    • Use `gsl::span`
  • One-past-the-end pointers

Run-time checks are allowed by the standard, but needs to be enforced for safety guarantees
Example: range checks

• Use raw pointers only for pointing
  • **F.22: Use T* or owner<T*> to designate a single object**

• Then what?
  • **ES.71: Prefer a range-for-statement to a for-statement when there is a choice**
  • **F.24: Use a span<T> or a span_p<T> to designate a half-open sequence**
  • **R.14: Avoid [] parameters, prefer span**
  • **SL.con.1: Prefer using STL array or vector instead of a C array**

• For example:
  ```cpp
  void f(int* p, span<int> s)
  {
    p[7] = 9; // not OK
    s[7] = 9; // OK (might throw)
    for (int x : s) x=f(x); // better: no runtime check
  }
  ```

Note: using span is simpler than (pointer,count) argument pairs
Example: `nullptr` problems

- Mixing `nullptr` and pointers to objects
  - Causes confusion
  - Requires (systematic) checking
- Caller
  ```c
  void f(char*);
  f(nullptr);  // OK?
  ```
- Implementer
  ```c
  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
Example: not_null<T> use

• not_null in interfaces
  void f(not_null<char*> p)
  {
    if (p==nullptr) *p = 'c';    // OK (but redundant – warn)
    *p = 'c';                    // OK
    // ...
  }

  void user(char* q)
  {
    f(nullptr);                  // not OK: detected or throws
    f(q);                        // OK: might throw
    if (q) f(q);                 // OK: won’t throw
  }

Stroustrup - Safety - Issaquah 2023
Example: not_null<T> use

• Not using not_null implies that tests are required
  
  ```c++
  void f(char* p)
  {
    *p = 7; // not OK
    if (p!=nullptr) *p = 7 // OK
    // ...
  }

  void user(char* q)
  {
    f(nullptr); // OK: f() is supposed to check
    f(q); // OK: q might be nullptr but f() is supposed to check
    if (q) f(q); // OK: redundant check
  }
  ```

  ```c++
  ```
One-past-the-end pointers

• Can be formed, but not dereferenced
  
  vector<int> v;  // fill v
  auto p = find(v,42);  // p becomes v.end()
  *p = 9;  // disaster
  if (p!=v.end()) *p=9;  // allowed

• Hard for static analyzers
  • Exactly what pointers are one-past-the-end – not just std

• Suggestion (not implemented)
  • Introduce gsl::not_end(p,v) overloaded on p (pointer) and v (container)
    • to help analyzers and human readers
  *p = 9;  // not OK
  if (not_end(p,v)) *p = 9;  // OK
Memory pools

• Not all memory is directly managed by `new` and deleted by `delete`
  • E.g., `malloc()/free()`
• We must handle user-managed “memory pools”
  • Problem: there is no standard memory pool abstraction
  • `<memory_resource>` is not yet widely used
• Alternative strategies (for using pointers to members of a pool)
  • Disallow members to be deleted or relocated
    • Requires `[[not_invalidating]]` annotations unless all pointers to elements are `const`
  • Disallow pointers to members to escape
  • Invalidate all pointers to elements if a potentially deleting or relocating operation is invoked
    • `std::vector` is an example
    • Must be communicated to the static analyzer: non-`const` and `[[not_invalidating]]`
Example of memory pools: Graphs

• Consider a general graph:

```c++
struct Tree_node {
    Value val;
    unique_ptr<Tree_node> left;
    unique_ptr<Tree_node> right;
};

struct Tree {
    unique_ptr<Tree_node> head;
    // ...
};
```

• Not OK: can lead to loops, implying resource leaks
  • Conservative strategy: reject `Tree_node` because ownership loops and leaks are possible
  • `Shared_ptr` would not solve this

• Extracting `Tree_node*`s from `unique_ptr<Tree_node>`s would cause a lot of invalidation
Example of memory pools: Graphs

• One solution: separate ownership from access

    struct Tree_node2 {
        Value val;
        Tree_node2* left;
        Tree_node2* right;
    };

    struct Tree2 {
        vector<unique_ptr<Tree_node2>> nodes;
        Tree_node2* head;
        // ...
    };

• Accessor loops are acceptable
• Ownership loops are not OK (and detectable)
Concurreny

• The Core Guideline rules are incomplete (but still helpful)
  
  • CP.20: Use RAII, never plain lock()/unlock()
  • CP.21: Use lock() or scoped_lock to acquire multiple mutexes
  • CP.22: Never call unknown code while holding a lock (e.g., a callback)
  • CP.23: Think of a jthread as a scoped container
  • CP.24: Think of a thread as a global container (implies invalidation checks against aliasing)
  • CP.25: Prefer jthread over thread
  • CP.26: Don't detach() a thread

  • std::jthread is a “joining thread”, obeying RAII

*For more suggested CG concurrency rules see Michael Wong’s CppCon’21 MISRA C++ talk*
Why not enforcement *exclusively* through language rules?

- **Stability/compatibility**
  - Billions of lines of code
- **Different domains have different definition of “safety”**
  - Basic type-and-resource safety should be common
- **Gradual adoption**
  - Essential
  - Many of the Core Guidelines checks are in use “at scale”

- **Most desirable**
  - Platform-independent static analyzer
  - Uniform adoption of the basic type-and-resource safety rules
  - Compiler and build options for invoking the static analyzer
Many notions of safety

• **Logic errors**: perfectly legal constructs that don’t reflect the programmer’s intent, such as using `<` where `a <=` or `a >` was intended.

• **Resource leaks**: failing to delete resources (e.g., memory, file handles, and locks) potentially leading to the program grinding to a halt because of lack of available resources.

• **Concurrency errors**: failing to correctly take current activities into account leading to (typically) obscure problems (such as data races and deadlocks).

• **Memory corruption**: for example, through the result of a range error or by accessing and memory through a pointer to an object that no longer exists thereby changing a different object.

• **Type errors**: for example, using the result of an inappropriate cast or accessing a union through a member different from the one through which it was written.

• **Overflows and unanticipated conversions**: For example, an unanticipated wraparound of an unsigned integer loop variable or a narrowing conversion.

• **Timing errors**: for example, delivering a result in 1.2ms to a device supposedly responding to an external event in 1ms.

• **Termination errors**: a library that terminates in case of “unanticipated conditions” being part of a program that is not allowed to unconditionally terminate.
Why Safety Profiles?

- Arbitrary C or C++ code is too complex for static analysis
  - Halting problem
  - Dynamic linking
  - Cost of global analysis
- Arbitrary C or C++ forces us to deal with too low an abstraction level
  - Ends up chasing complexities in messy old-style code
  - Backwards looking
- We care about performance as well as type-and-resource safety
- Eventually much higher productivity
Why Safety Profiles?

• We need a coherent set of rules
  • Not just a lot of unrelated tests

• Profile: a coherent sets of rules yielding a guarantee
  • Current: bounds, type, memory
  • Must be visible in code
    • To indicate intent
    • To trigger analysis
Strategy: Safety Profiles

- Our approach is “a cocktail of techniques” not a single neat miracle cure

- Static analysis
  - to verify that no unsafe code is executed.

- Coding rules
  - to simplify the code to make industrial-scale static analysis feasible.

- Libraries
  - to make such simplified code reasonably easy to write
  - to guarantee run-time checks where needed.