C++ now

2025

Safer C++ at Scale with Static Analysis

Yitzhak Mandelbaum

Memory Safety Now!

- 2024 White House memo
- Secure by Design at Google (2024 report)
- A lot more … !

- Rust Language — a true alternative
- Detection and mitigation is not enough.
  - Constant game of catch up.

C++ Static Safety

- Goal: drag C++ incrementally towards safety
- No illusions, but we can improve things!
- Compromise: rule out bug classes, don’t chase bugs
    - the only way to win is not to play the game

Safety: The Ideal

“Well-typed programs can’t go wrong”
—Robin Milner, 1978

Compilation = Freedom from *Undefined Behavior*.

Memory Safety

Spatial safety
Temporal safety
Type safety
Initialization safety
Data-race safety

Nullptr Safety
Integer Overflow Safety
more!

out-of-bounds data access.

use-after-free,
use-after-return,
double free, ...

use of uninitialized memory.

dereference of nullptr.

C++ at Google Scale

- Hundreds of millions of C++ LoC
- Tens of millions of pointer declarations
- Millions of files
- Tens of thousands of C++ developers

Nullptr Safety

C++ pointers conflate “value required” and “value optional”.

Worse, you can’t distinguish — no specification for “this pointer may (not) be null”.

Compare with `int` vs `std::optional<int>`, which are clearly distinct.

Outages

Large percentage of crashes in Google C++ binaries caused by nullptr dereferences.

Cognitive Overhead

Developers need to figure out the category of every pointer they interact with.

Impedance Mismatch

… with memory-safe languages like
Rust, Swift and Carbon.

Solution: ~~Change~~ Extend the language...

... with pointer annotations — API contracts with well-defined meaning.

Which annotations?

?

Which annotations?

Just Nonnull! (because the default semantics are Nullable)

Just Nullable! (because I don’t want false positives)

Both! 3 states: Nullable, Nonnull and legacy.

Nonnull: null is invalid.
Nullable: null is valid.
Unknown: contract not specified — treated optimistically.

Clang	Abseil
`_Nonnull`	`absl_nonnull`
`_Nullable`	`absl_nullable`
`_Null_unspecified`	`absl_nullability_unknown`

```
int foo(int *_Nonnull p);

int *_Nullable bar(int q);

int *zab();
```

Legacy pointers implicitly Unknown.

Enforcement

```
int f(int *_Nonnull p) {
  return *p + 1;
}

int f(int *_Nullable p) {
  return *p + 1; // ERROR
}
```

```
int f(int *_Nullable p) {
  if (p != nullptr)
    return *p + 1;
  return 0;
}
```

Conservative Extension

- Contract, not a guarantee.
- No effect on compilation.
- No UB from contract violation (like passing Nullable to Nonnull).
    - only from actual violation (like dereferencing a nullptr).

Gradual Analysis

```
void g(int*);   // Unknown

void f(int *_Nullable p) {
  g(p);   // Assume Nullable
          // is permitted.
}
```

```
int g(int *_Nonnull);

void f(int *p) {  // Unknown
  g(p);   // Assume p is
          // Nonnull.
}
```

Gradual Program Analysis for Null Pointers (S. Estep et al.)

Gradual Analysis (sort of)

```
void g(int*);   // Unknown

void f(int *_Nullable p) {
  g(p);   // Assume Nullable
          // is permitted.
}
```

```
int g(int *_Nonnull);

void f(int *p) {  // Unknown
  g(p);   // Assume p is
          // Nonnull.
}
```

Dynamic check?

Function-by-function Analysis

```
int f(int *_Nullable p) {
  if (p == nullptr)
    return 0;
  ...
  foo(*p + 1);
  ...;
}
```

```
bool isNull(int *_Nullable p) {
  return p == nullptr;
}

int f(int *_Nullable p) {
  if (isNull(p)) return 0;

  foo(*p + 1);   // ERROR

  ...;
}
```

A Good Start

- Function-scope + Gradual = Strong foundation for incremental deployment
- But: how do we get from 0 to 100?

Inference

Challenge: automatically infer the contracts on pointer types …

… given only the source code.

Suggestions?

- Dereference
- Parameter-binding
- Assignment
- Boolean test (esp. if-condition)
- All this + more

Inference Example

```
char *get(int i) {
  if (i > 10)
    return doSomething(i);
  return nullptr;
}

int f(int *p) {
  return *p + 1;
}
```

```
char *_Nullable get(int i) {
  if (i > 10)
    return doSomething(i);
  return nullptr;
}

int f(int *_Nonnull p) {
  return *p + 1;
}
```

Foundation for C++ Nullability at Scale

1. Conservative language extension
2. Gradual, modular analysis
3. Automated annotation
4. Very high precision

Reducing the Noise

Annotations add clarity, but clutter the code.

Imbalance, in practice:

- Google code: :1 Nonnull : Nullable
- Third-party code: :1 Nonnull : Nullable

Toggle default meaning of T* from Unknown to Nonnull.

clang: region-based, Abseil: whole file.

Reducing the Noise

Annotations add clarity, but clutter the code.

Imbalance, in practice:

- Google code: 6:1 Nonnull : Nullable
- Third-party code: 4:1 Nonnull : Nullable

Toggle default meaning of T* from Unknown to Nonnull.

clang: region-based, Abseil: whole file.

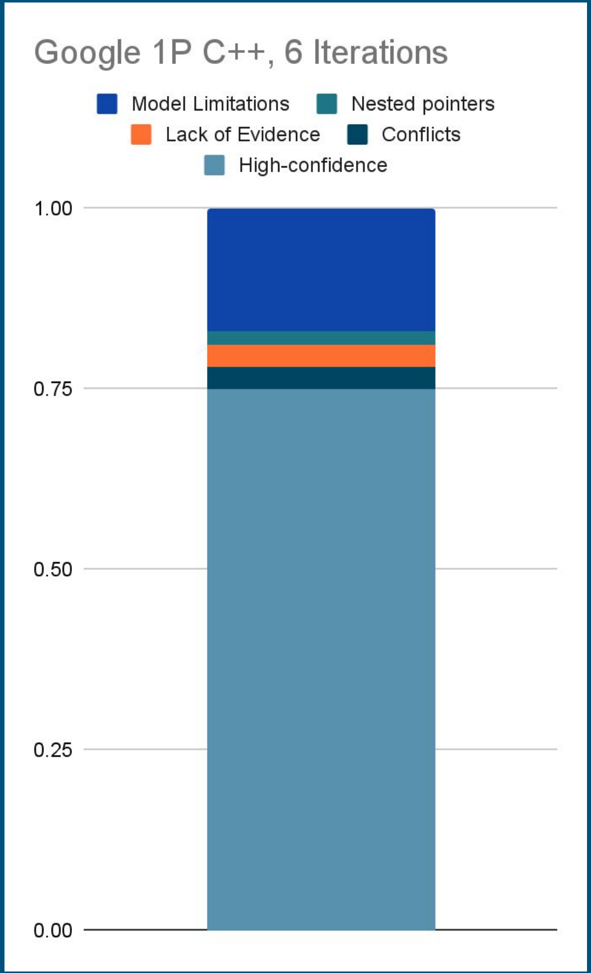
Analysis Status

- Implemented as a ClangTidy check:
  - Check released in Crubit OSS repository.
  - Built on Clang's Dataflow Framework.
- Enabled for all of our first-party C++ code in March 2024.
  - Only a small number of common false-positive and false-negative patterns.
  - On average, about 1% of diagnostics are flagged "not useful".
  - Many bug reports are user-error — more education and documentation needed.

Inference Status

- 75% high-confidence
- 3% conflicts
- 3% lack of evidence
- 19% — model limitations, of which nested pointers are 2-3%.

With “guesses”, est. >= 95% correct



Lessons Learned

- Naive approaches go further than expected.
  - Analysis: “Soundiness”, intra-procedural, simple SAT solver.
  - Inference: brute-force heuristics.
- Syntactic models are hard. Start simple.
- Hard-code common patterns
  - Example: “stable” functions.
- UX is crucial!

```
void f(Obj &o) {
  if (o.get() != nullptr)
    foo(*o.get());
}
```

Questions?

Spatial Safety

Can we solve* spatial safety for C++?

*modulo other safeties

Spatial safety: challenges

- Container indexing
- Pointer arithmetic
- Iterators (bounds and invalidation)
- C-style strings (null-terminated)
- Third-party libraries

We'll focus on the first 2

Safe Buffers

- Proposed by Apple in [Clang RFC] in 2022
  - Google started [investigating adoption] in 2024.
- Use (size-bearing) bounds-checked containers.
  - Harden primitive arrays.
  - Harden all core container types.
- Replace pointer arithmetic/indexing with use of Safe Buffer abstractions.

Challenge: distinguish buffer pointers

- When is a pointer not a pointer? When it’s a buffer.
- Reminder: tens of millions of pointers to consider.
- What’s so hard?

Source

```
void bar(int *p);
…
int arr[10];
bar(arr);
```

Use

```
void foo(int *p) {
  …
  int x = p[i];
  …
}
```

The Catch

Call graph

```
void bar(int *p, int j) {
  if (j > 10)
    foo(p);
  else
    zab(p, j);
}
```

?

Mixed use

```
void zab(int *p, int j) {
  *p = j;
}
```

External code

```
extern "C"
void zab(int *p, int j);
```

Solution

- Triple predicate:
  - can hold buffers,
  - can be used as buffers, (optimization)
  - not disqualified by loss of precision.
- Compute three subsets of the graph and intersect them.
  - “can and should replace”

Solution

- Inductive, transitive relation, computed iteratively.
- Summarize: build pointer-flow graph for each TU.
- Solve: combine per-TU graphs and solve across codebase.

Solution

- Cluster: to scale, also need to compute lists of files that must be changed together.
- Edit: translate identified pointers to appropriate Safe Buffer types.
  - Also: uses (e.g. assignments to non-buffer pointers) need coercions.
  - Or, annotate as “indexable” for Bounds Safety.

Anticipated Coverage:
70-80% of SB warnings,
50+% of GWP-ASAN OOB bugs.

Complications

- Ownership
  - views or containers?
  - need ownership detection.
- Unique pointers: `std::unique_ptr<T[]>`
  - clearly identified as buffer, but no size
  - `UniqueArray` — API to pair unique pointer with size.

Status

- Enabled bounds-checking for the C++ standard library, across workloads.
    - Resulted in average 0.3% performance impact.
- Completed automated migrations of some `new[]` to safe alternatives.
    - Local RAII
- Nearing completion of tooling for `unique_ptr<T[]>` migration.
- Piloting deployment of primitive-array hardening.
- Implementation start for spanification tooling.

Iterators, C-strings and 3rd-party code remain to be addressed.

Initialization Safety

The Perils of Uninitialized Memory

```
void maybeLaunch() {
  bool launch;
  // buggy logic that fails to
  // initalize `launch`.
  ...
  if (launch) launchRocket();
}
```

Launch depends on previous value of stack.

The Perils of Uninitialized Memory

Leaks

potentially compromising ASLR.

Data Corruption

including bad pointers.

Hangs

from uninitialized time delay.

Initialize ALL THE THINGS

But, initialization isn't free.

Zero-init isn't always a good fit.

Approach

- Compiler initializes all stack variables.
  - Rely on Dead-Store Elimination (DSE) to optimize
- Compiler (vs code rewriting)
  - covers more code with less churn
  - but, needs to be conservative, because no user review
- Focus on stack
  - Performance: lower cost to redundant stores.
  - Security: stack most easily exploited.

Improvements to Dead-Store Elimination

1. Enable inter-procedural DSE by annotating callees
2. Modify ThinLTO to surface DSE annotations so they're visible at link time.

Benefits accrue to all code!

Status

- Available in Clang with `-ftrivial-auto-var-init`
    - supports zero and (fixed) pattern init.
- Pattern-init enabled in Google's (unoptimized) test builds.
- DSE improvements integrated into Clang.
- Early estimates: < 1% overhead in optimized builds.

Temporal Safety

Temporal Safety

- Temporal Safety = lifetimes + validity
- Harder to exploit
- Harder to solve
  - Static solution seems to require extensive changes to the codebase

Identify pragmatic subsets!

Lifetimes

Lifetime attributes:

- `lifetime_bound`, `lifetime_capture_by(X)` implemented in Clang.
- Next: RFC to extend to function scope.

Rust-style lifetime syntax for interop:

- 2022 Clang RFC (not implemented).
- Planned for 2025.

Closing

Lessons for Scale

- Solve for the codebase.
- Incremental solutions are essential.
- Very low false-positive rate.
- Build relationships.

Towards Safer C++

- Safety requires substantial investments!
  - system development
  - codebase updates
- But, automation works and can drastically reduce the cost.
- Safety pays dividends:
  - improves the existing code
  - leads to better interop
- Focus on eliminating bug classes, rather than chasing bugs.
- Safety is not binary — “safer” is valuable and (incrementally) achievable.

Acknowledgements

Samira Bazuzi Bakon
Martin Brænne
Chandler Carruth
Wontae Choi
Yu Hao
Dmytro Hrybenko
Shreya Jain
Florian Mayer
Sam McCall
Utkarsh Saxena
Max Shavrick
Venkatesh Srinivasan
Juan Vazquez
Jan Voung
Kinuko Yasuda

And many more!

The End