Lifetime Safety in C++:

Past, Present and Future

GABOR HORVATH

Gabor.Horvath@microsoft.com

Welcome to CppCon 2023!

Come by our booth and join #visual_studio channel on CppCon Discord **https://aka.ms/cppcon/discord**

- Meet the Microsoft C++ team
- Ask any questions
- Discuss the latest announcements

Take our survey

Win prizes

https://aka.ms/cppcon/lifetime

2012

2014

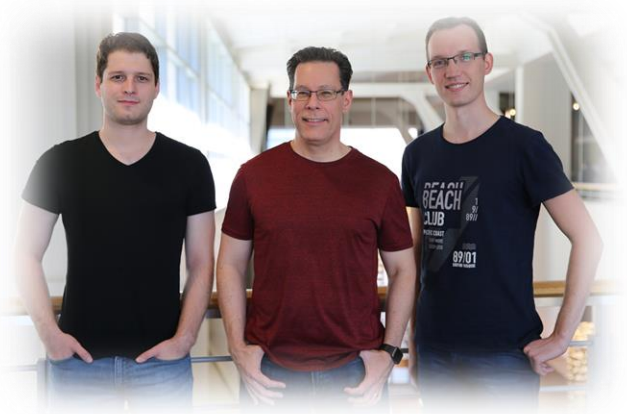
2015

2016

2017

2019/2020

2019



2018

2020

2021

2022

Now

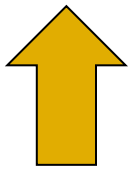
Gabor.Horvath@microsoft.com

Approaches to safety

C++ is getting safer

The lifetime safety toolbox

What comes next?




Picture in picture
2023
Safety & Security:
The Future of C++
JF Bastien
C++ now


Picture in picture
2023
Safety & Security:
The Future
JF Bastien
C++ now


Picture in picture
2023
All the Safeties
Sean Parent
C++ now


Picture in picture
2023
Safety & Security:
The Future
JF Bastien
C++ now
Picture in picture
2023
All the
Sean Parent
C++ now
Picture in picture
2023
Coding for Safety, Security, and Sustainability
Panel Discussion
Bob Steagall, JF Bastien, Chandler Carruth, Daisy Hollman, Lisa Lippincott, Sean Parent, and Herb Sutter
C++ now


Picture in picture
2023
Safety & Security:
The Future
JF Bastien
C++ now
Picture in picture
2023
All the
Sean Parer
C++ now
Picture in picture
2023
Coding fo
and Susta
Panel Discus
Bob Steagall,
Daisy Hollma
and Herb Sut
C++ now
Picture in picture
2023
Safety-First
Understanding How to Develop
Safety-critical Software
Andreas Weis
C++ now

Picture in picture

2023

Safety & Securit
The Future

JF Bastien

C++ now

National Security Agency | Cybersecurity Information Sheet

Software Memory Safety

Executive summary

Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in Chrome®. [2] Malicious cyber actors can exploit these vulnerabilities for remote code execution or other adverse effects, which can often compromise a device and be the first step in large-scale network intrusions.

Commonly used languages, such as C and C++, provide a lot of freedom and flexibility in memory management while relying heavily on the programmer to perform the needed checks on memory references. Simple mistakes can lead to exploitable memory-based vulnerabilities. Software analysis tools can detect many instances of memory management issues and operating environment options can also provide some protection, but inherent protections offered by memory safe software languages can prevent or mitigate most memory management issues. NSA recommends using a memory safe language when possible. While the use of added protections to non-memory safe languages and the use of memory safe languages do not provide absolute protection against exploitable memory issues, they do provide considerable protection. Therefore, 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. [3] [4] [5]

2023

How to Develop
oftware

Picture in picture

2023

C++ now

Picture in picture

2023

Safety & Securit
The Future

C++

National Security Agency | Cybersecurity Information Sheet

Software Memory Safety

protection against exploitable memory issues, they do provide considerable protection. Therefore, 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. [3] [4] [5]

NIST

Search NIST

Menu

Information Technology Laboratory / Software and Systems Division

SOFTWARE QUALITY GROUP

Safer Languages

[SAMATE HOME | INTRO TO SAMATE | SARD | SATE | BUGS FRAMEWORK | PUBLICATIONS | TOOL SURVEY | RESOURCES]

Safety or quality cannot be "tested into" programs. It must be designed in from the start. Choosing to implement with a safer or more secure language or language subset can entirely avoid whole classes of weaknesses.

23

C++ now

Safety & Securit
The Future

NIST

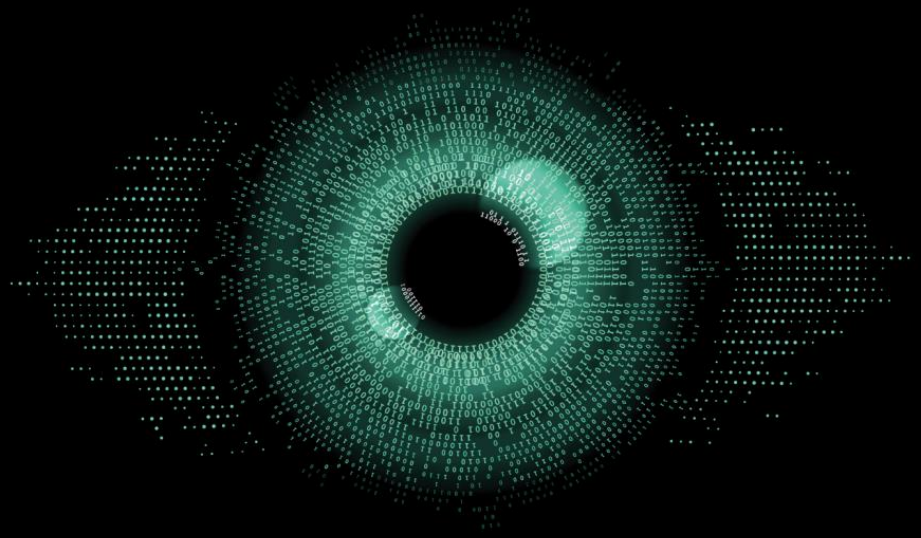
Information Technology Laboratory / Sof

SOFTWARE QUALITY G

Safer Language

[SAMATE HOME

Safety or quality cannot be "tested into" pr
entirely avoid whole classes of weaknesses



Future of Memory Safety

Challenges and Recommendations

YAEL GRAUER
JANUARY 2023

CR Consumer Reports | Security Planner

arch NIST

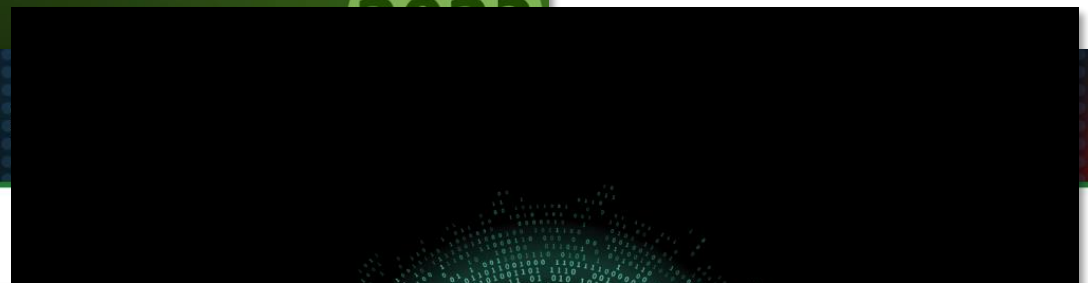
Menu

L SURVEY | RESOURCES]

r or more secure language or language subset can

C++ now

Safety & Securit
The Future



CYBERSECURITY & INFRASTRUCTURE SECURITY AGENCY

AMERICA'S CYBER DEFENSE AGENCY

Search

Topics | Spotlight | Resources & Tools | News & Events | Careers | About

Home / News & Events / News

BLOG

The Urgent Need for Memory Safety in Software Products

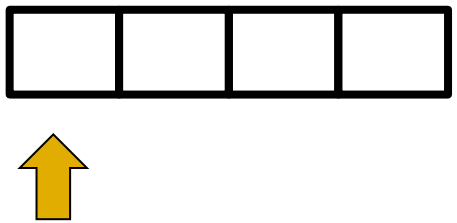
CR Consumer Reports | Security Planner

C++ now

Memory Safety

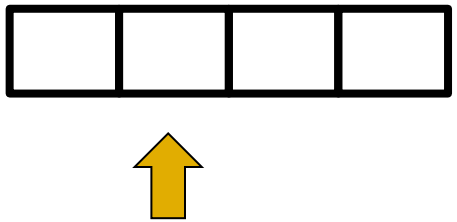
- Microsoft: 70 percent of all security bugs are memory safety issues | ZDNET
- Memory safety (chromium.org)
- Implications of Rewriting a Browser Component in Rust - Mozilla Hacks - the Web developer blog
- Google Online Security Blog: Memory Safe Languages in Android 13 (googleblog.com)

Spatial safety



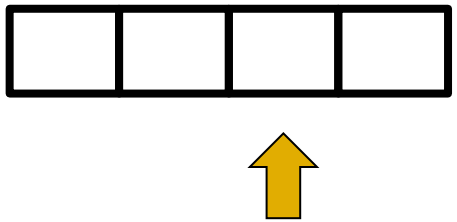
Temporal safety

Spatial safety



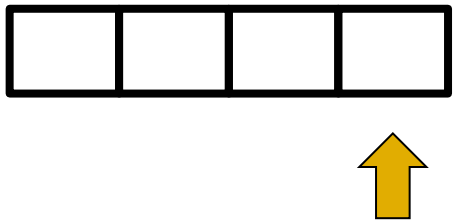
Temporal safety

Spatial safety



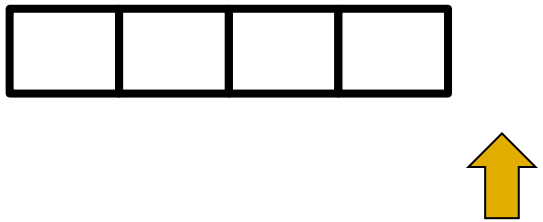
Temporal safety

Spatial safety



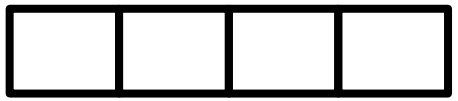
Temporal safety

Spatial safety



Temporal safety

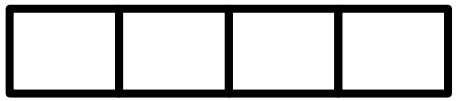
Spatial safety



- BufferCheck (soon), SAL
- ASAN, GWP-ASAN, HWASAN + Fuzzing
- Bounds-checked data structures
- Checked C, Deputy
- -fbounds-safety, buffer hardening

Temporal safety

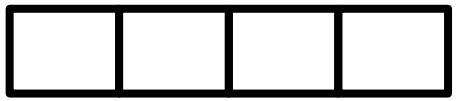
Spatial safety



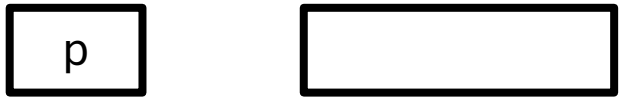
- BufferCheck (soon), SAL
- ASAN, GWP-ASAN, HWASAN + Fuzzing
- Bounds-checked data structures
- Checked C, Deputy
- -fbounds-safety, buffer hardening

Temporal safety

Spatial safety

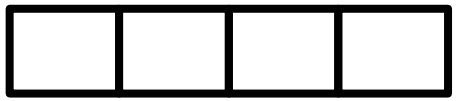


Temporal safety

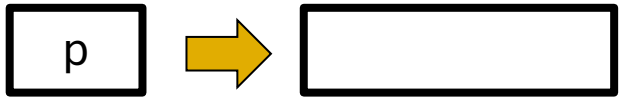


- BufferCheck (soon), SAL
- ASAN, GWP-ASAN, HWASAN + Fuzzing
- Bounds-checked data structures
- Checked C, Deputy
- -fbounds-safety, buffer hardening

Spatial safety

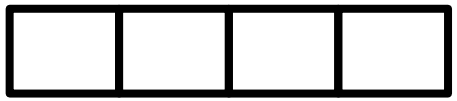


Temporal safety

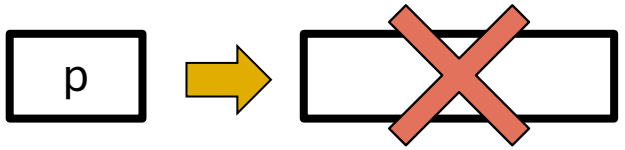


- BufferCheck (soon), SAL
- ASAN, GWP-ASAN, HWASAN + Fuzzing
- Bounds-checked data structures
- Checked C, Deputy
- -fbounds-safety, buffer hardening

Spatial safety

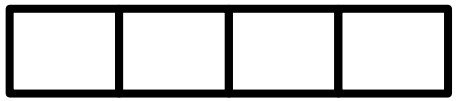


Temporal safety



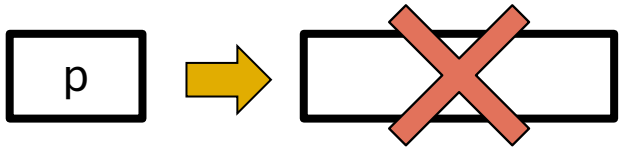
- BufferCheck (soon), SAL
- ASAN, GWP-ASAN, HWASAN + Fuzzing
- Bounds-checked data structures
- Checked C, Deputy
- -fbounds-safety, buffer hardening

Spatial safety



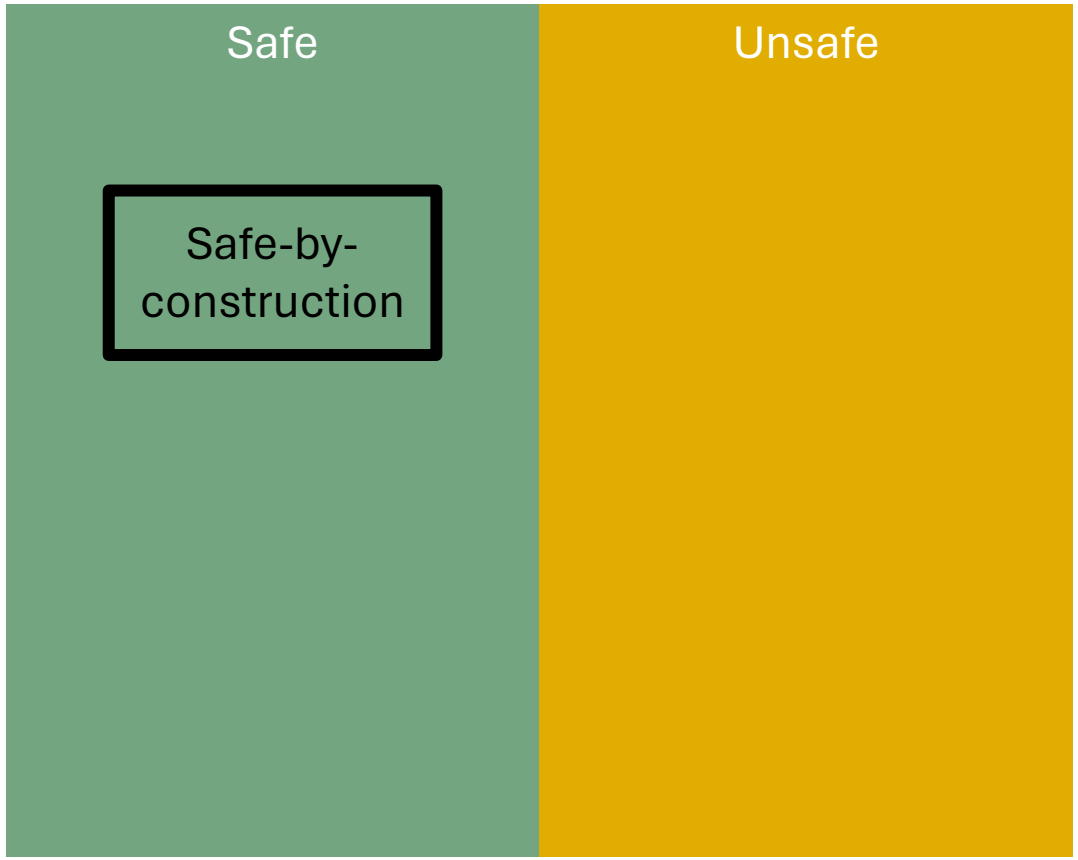
- BufferCheck (soon), SAL
- ASAN, GWP-ASAN, HWASAN + Fuzzing
- Bounds-checked data structures
- Checked C, Deputy
- -fbounds-safety, buffer hardening

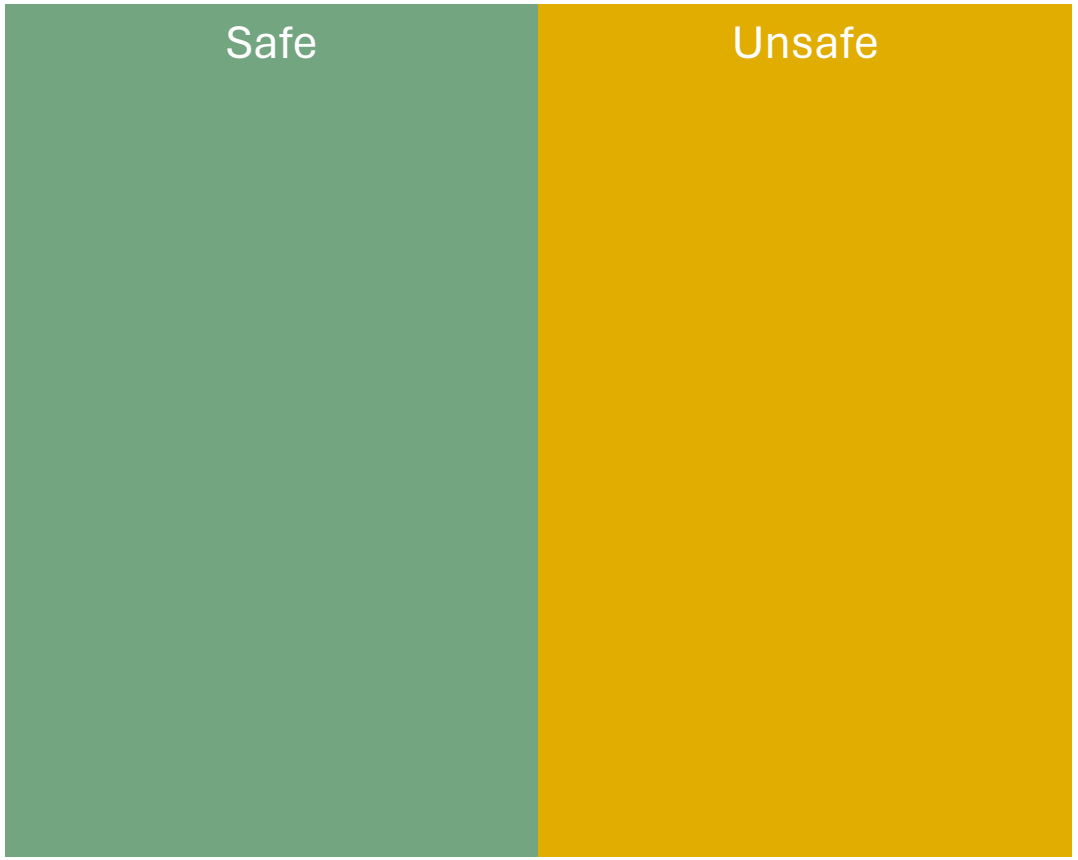
Temporal safety



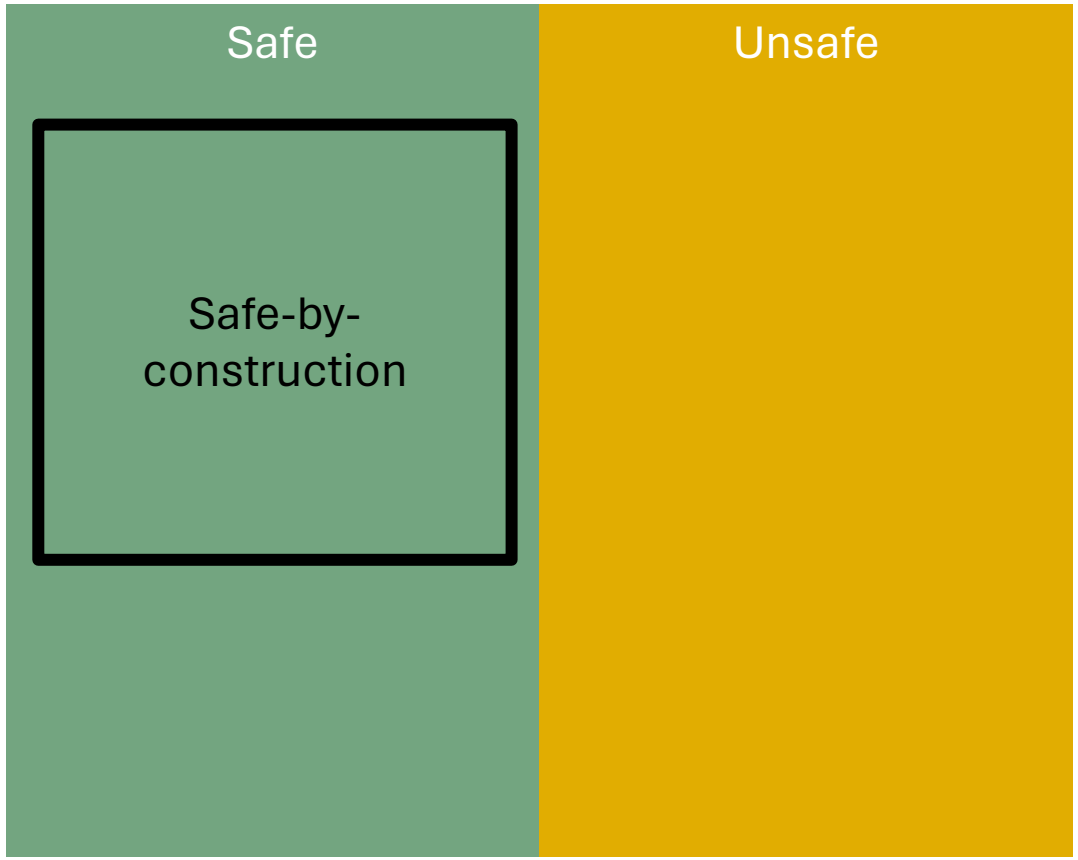
- ASAN, GWP-ASAN, HWASAN + Fuzzing
- Lifetime Safety Profile, Crubit
- C26815, C26816
- Cyclone, P2771
- -Wdangling-gsl
- -Wdangling-reference

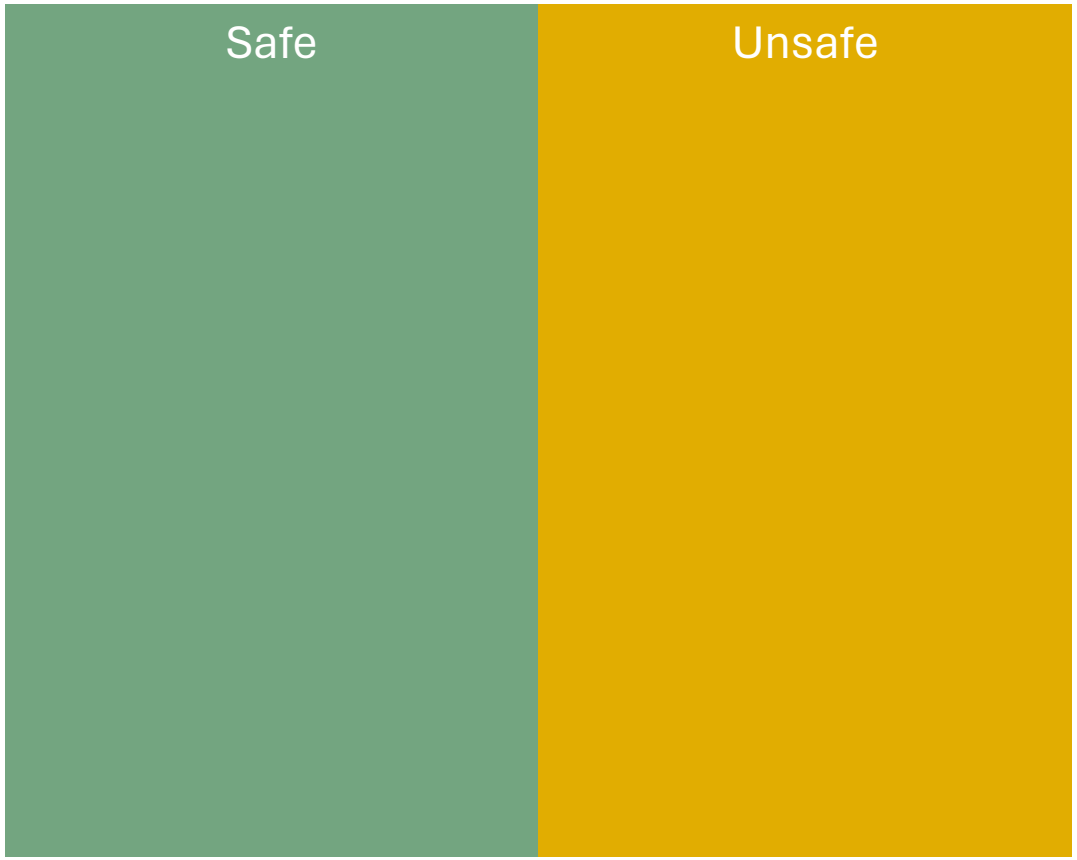
Approaches to safety



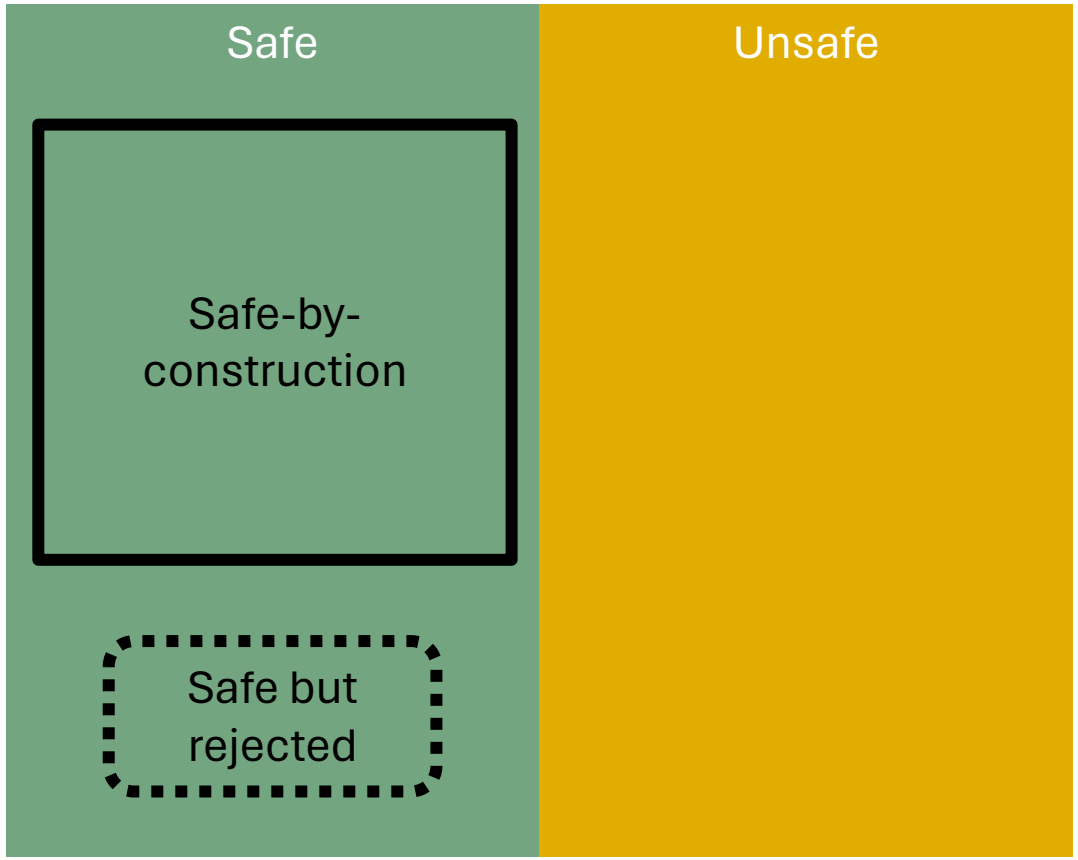


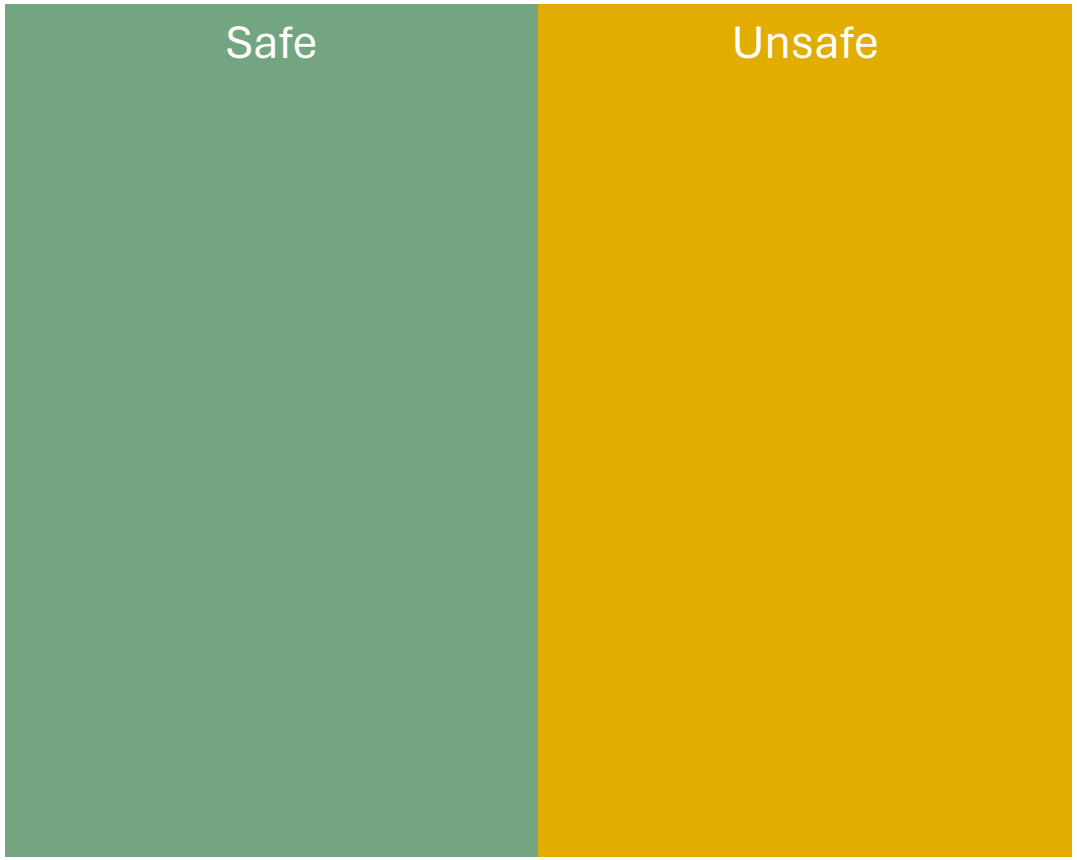
Approaches to safety



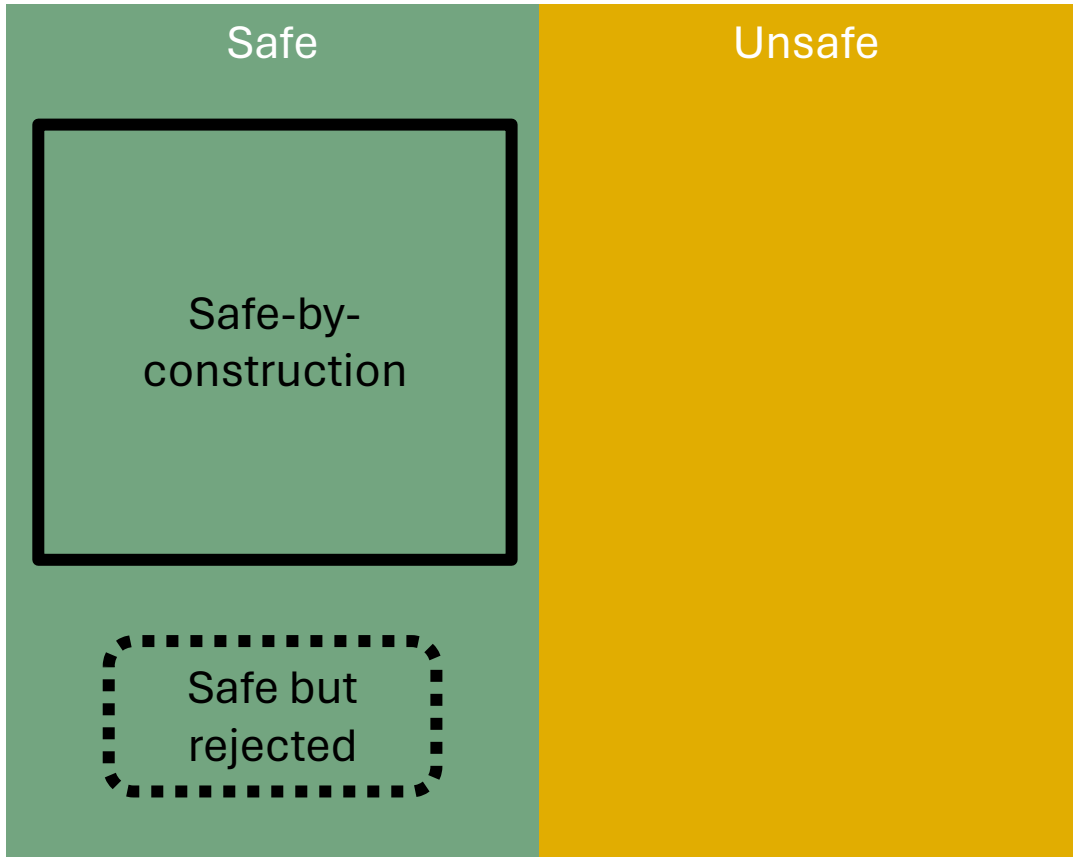


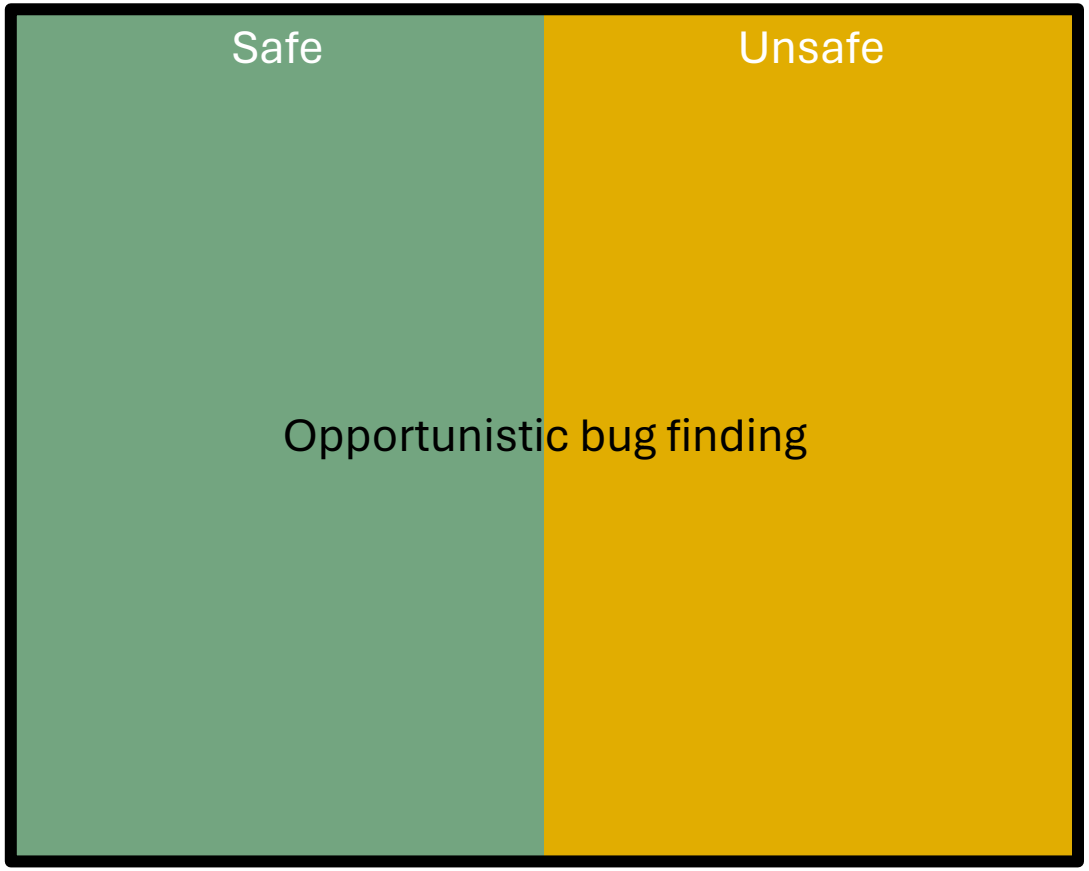
Approaches to safety



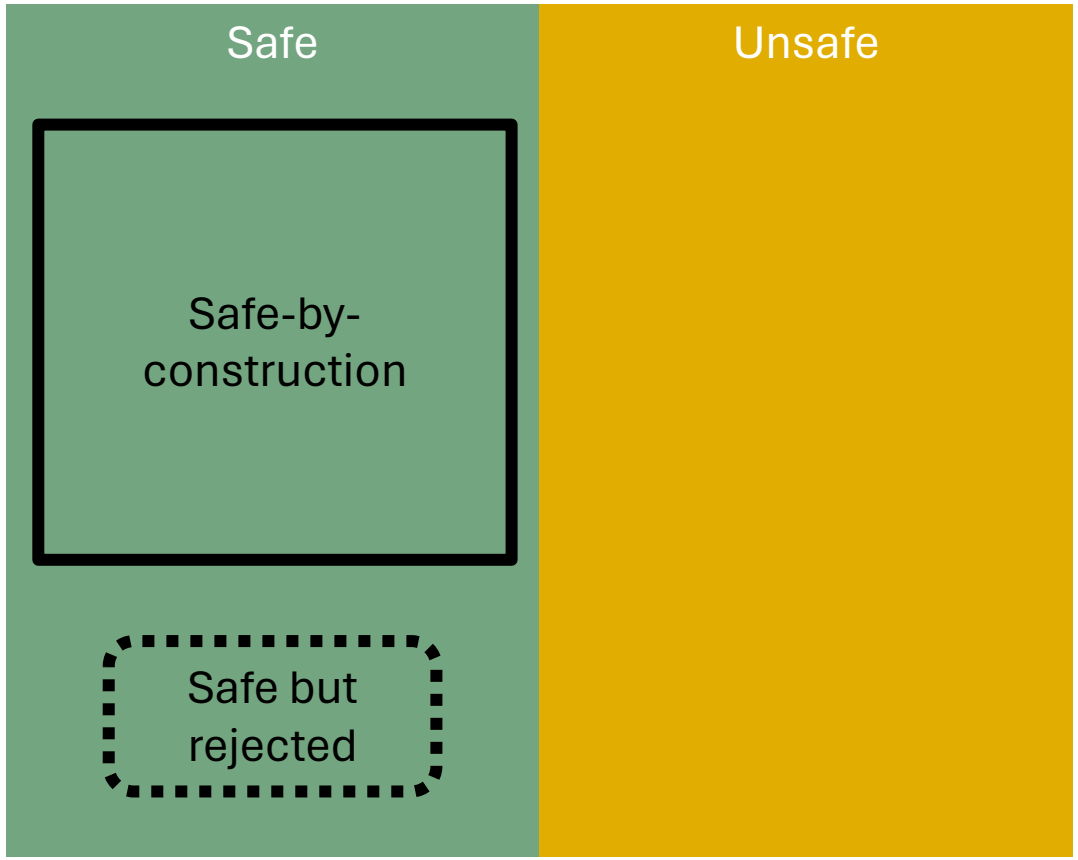


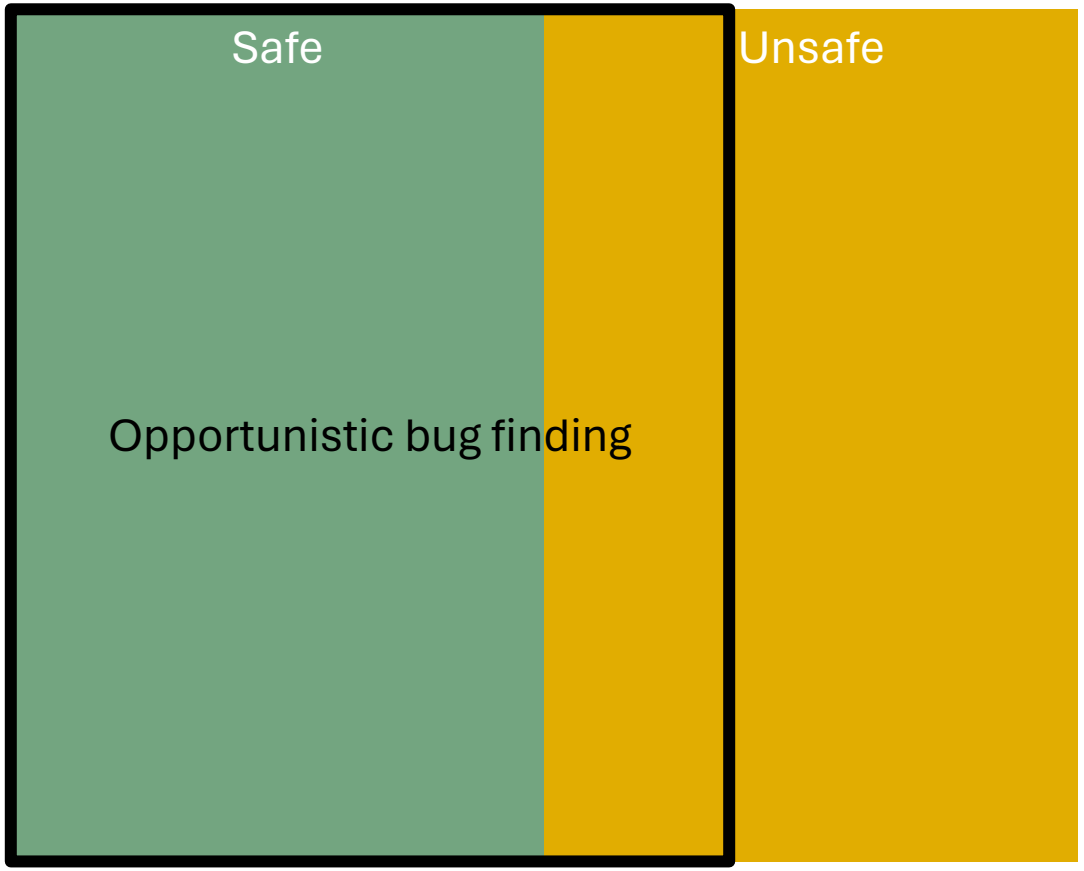
Approaches to safety



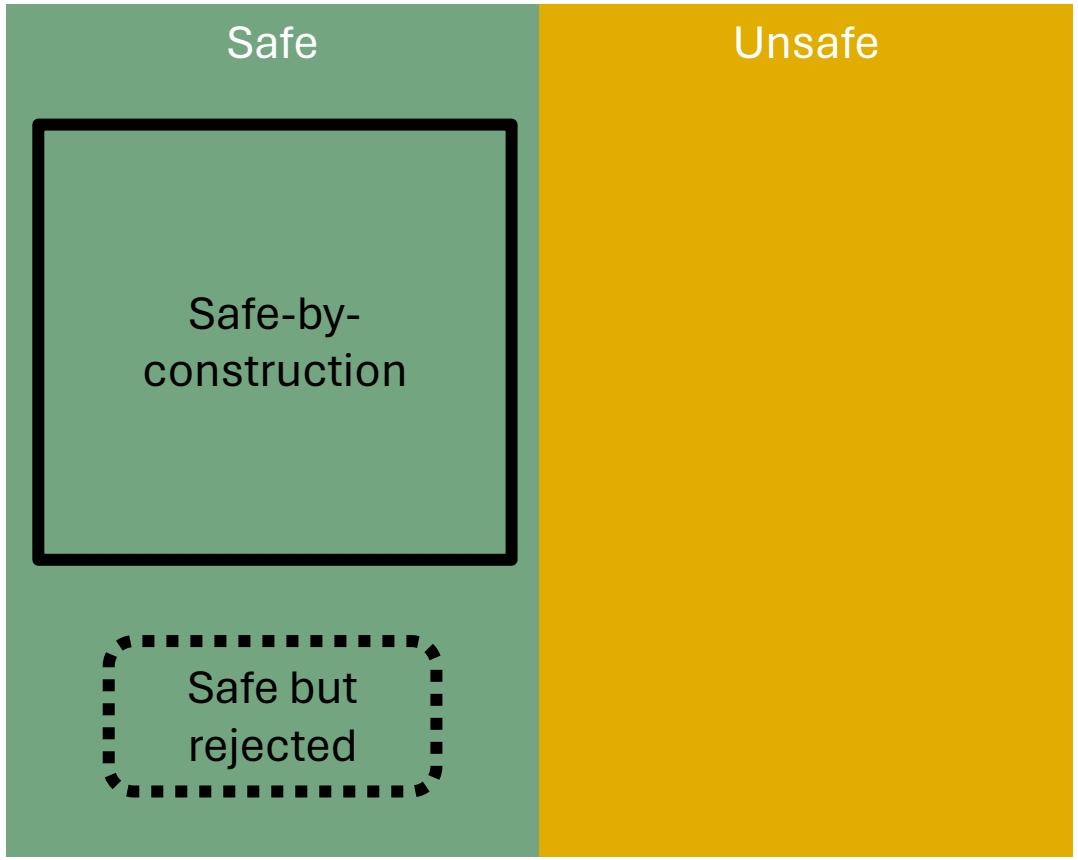


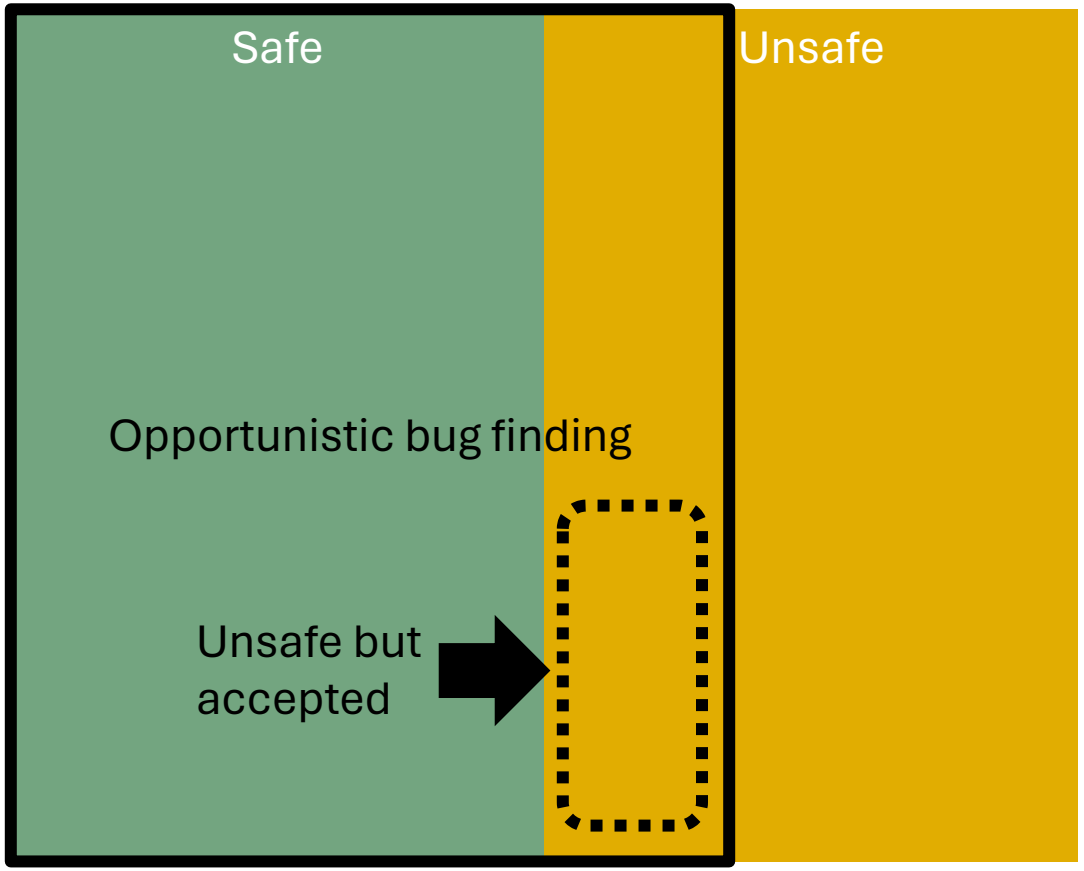
Approaches to safety



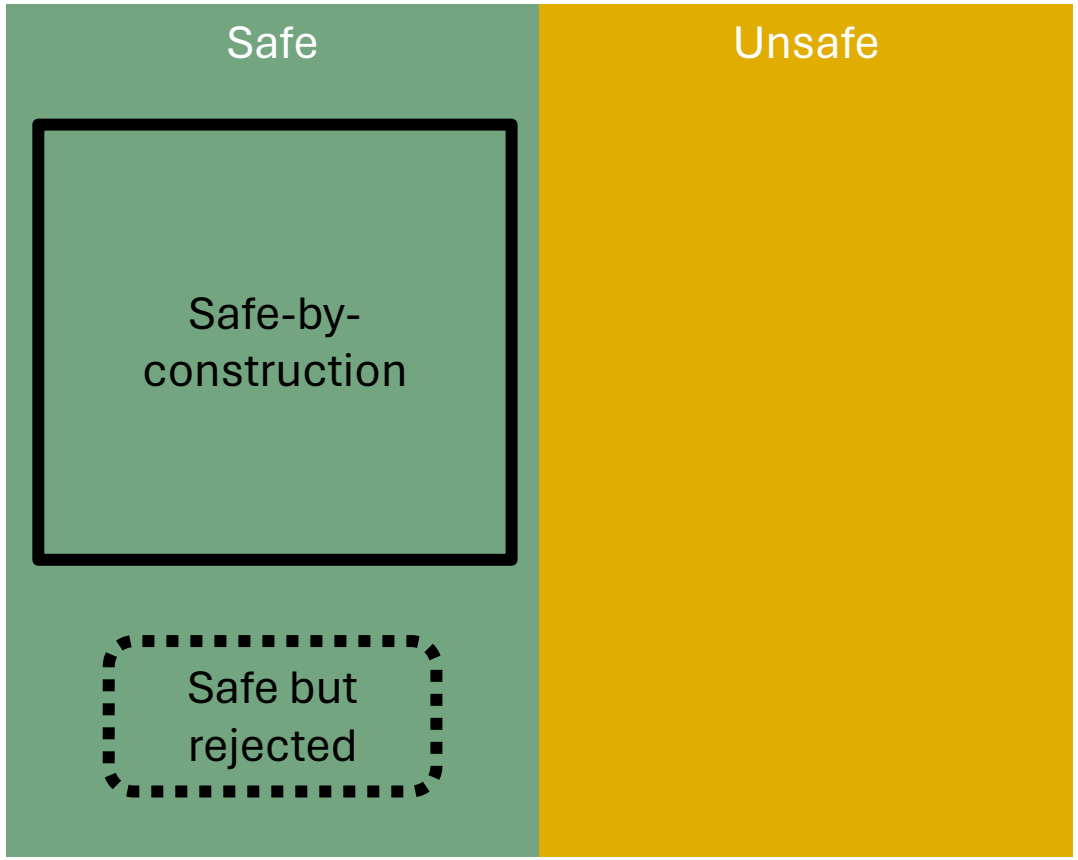


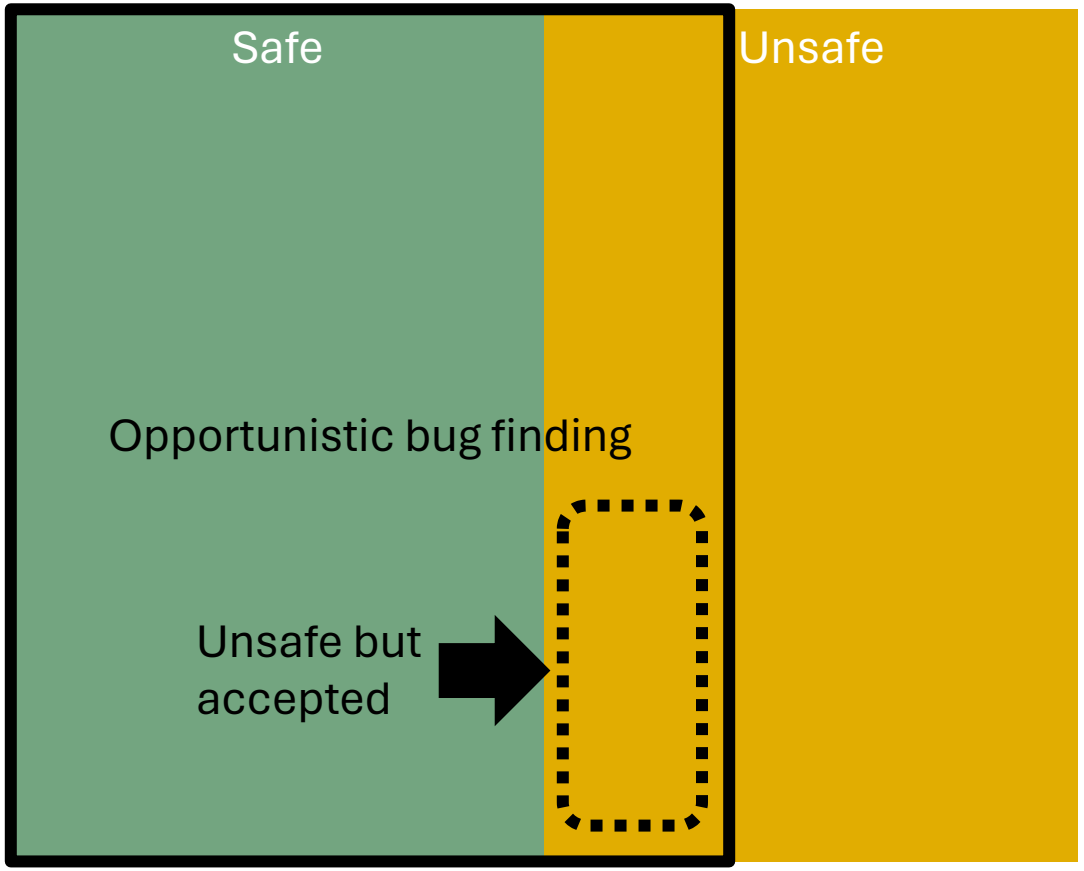
Approaches to safety





Approaches to safety





Rice’s theorem

Safe-by-construction

- Guarantees
  - Eliminates classes of bugs
- Comes with escape hatches
  - Type system + unsafe casts
  - Borrow checker + raw pointers
- Might need rearchitecting
- Can make code less malleable

Opportunistic bug-finding

- Applicable to all code
- Easier to identify false positives
- Small/localized fixes
- Easier to roll out incrementally

It needs different APIs to work well:

To exit full screen, swipe down from top of screen or press F11

C++

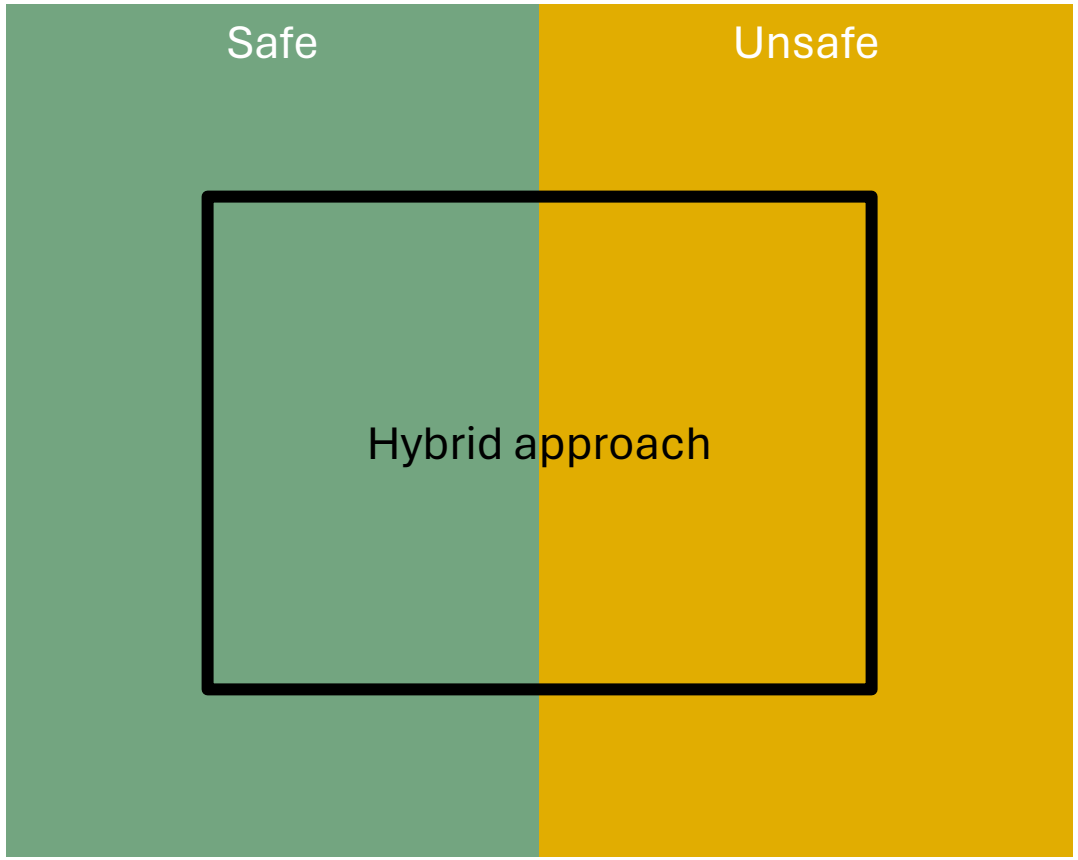
```cpp
void swap_span(std::span<int> a,
               std::span<int> b) {
  for (size_t i = 0;
       i < a.size();
       i += 1) {
    std::swap(a[i], b[i]);
  }
}

int main() {
   std::vector<int> v = {1, 2, 3, 4, 5, 6};

   swap_span(
       std::span(v).subspan(0, 3),
       std::span(v).subspan(3, 3)
   );
}
```

Rust

```rust
fn swap_span(a: &mut [i32],
             b: &mut [i32]) {


  for i in 0..a.len() {
    std::mem::swap(&mut a[i], &mut b[i])
  }
}

pub fn main() {
  let mut v = vec![1, 2, 3, 4, 5, 6];

  // Mutable borrows `v` once, but produces
  // two independent mutable spans.
  let (first, second) = v.split_at_mut(3);

  swap_span(first, second);
}
```

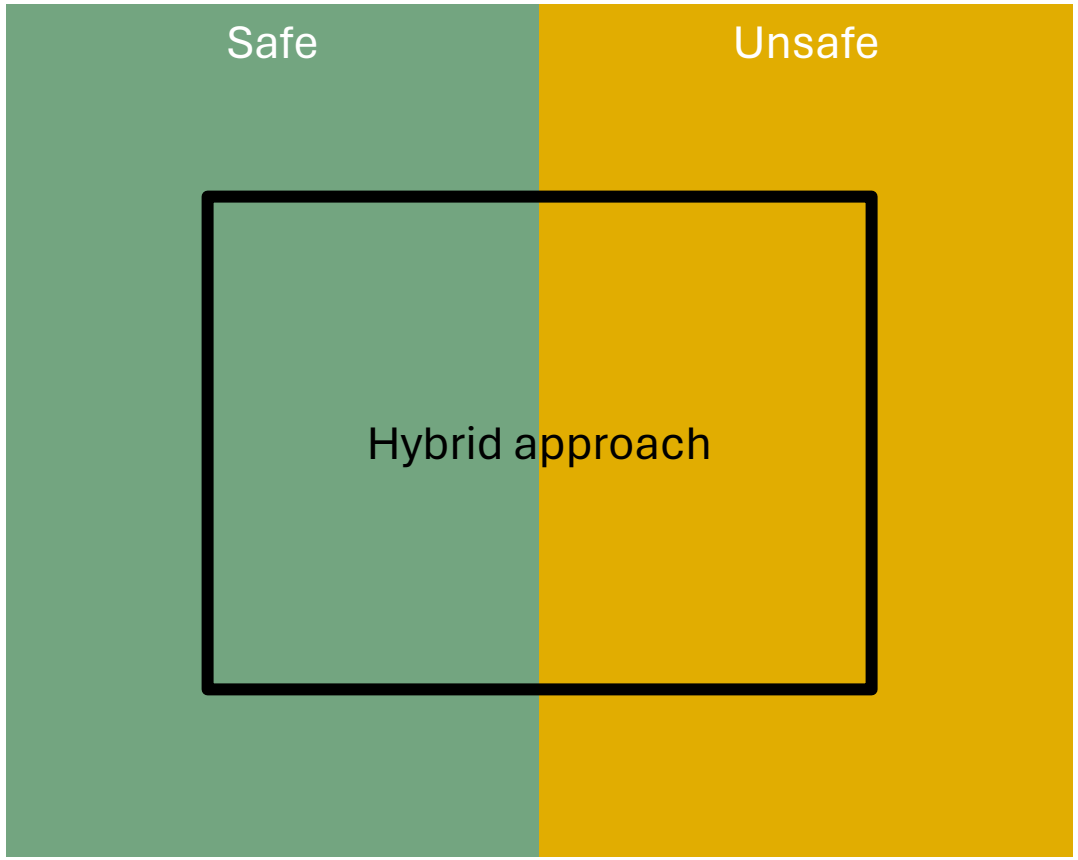
Carbon Language Successor Strategy: From C++ Interop to Memory Safety - Chandler Carruth - CppNow 23

Approaches to safety



- Suggest safer constructs
- Find bugs

Approaches to safety

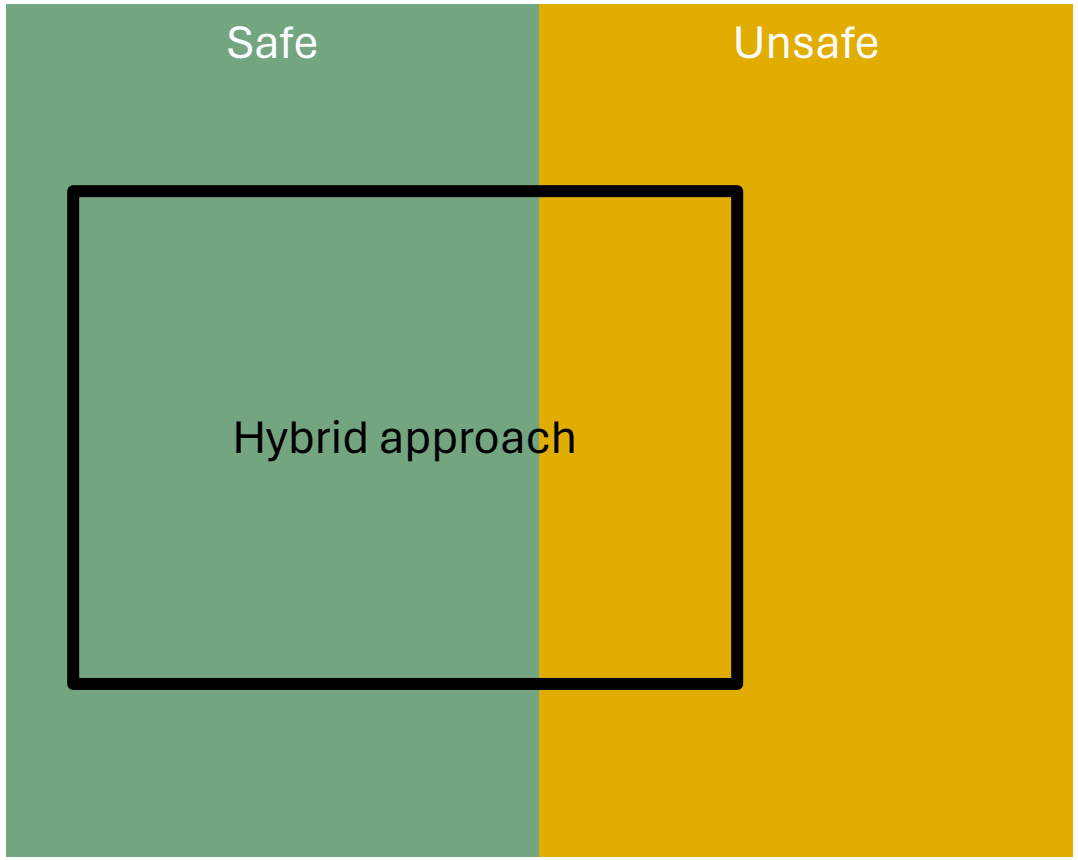


- Suggest safer constructs
- Find bugs

- Suggest use of RAII
- Find bad locking patterns

- Choose your own adventure

Approaches to safety



- Suggest safer constructs
- Find bugs

- Suggest use of RAII
- Find bad locking patterns

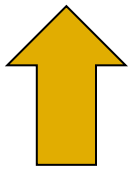
- Choose your own adventure

Approaches to safety

C++ is getting safer

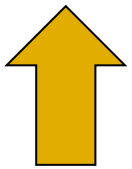
The lifetime safety toolbox

What comes next?



Approaches to safety

C++ is getting safer



The lifetime safety toolbox

What comes next?

C++ is getting safer: constexpr/consteval!

```
consteval int f()
{
    int *p = nullptr;
    {
        int i = 1729;
        p = &i;
    }
    return *p;
}

constexpr int x = f();
```

C++ is getting safer: constexpr/consteval!

```
consteval int f()
{
    int *p = nullptr;
    {
        int i = 1729;
        p = &i;
    }
    return *p;
}

constexpr int x = f();
```

```
<source>(11): error C2131: expression did not evaluate to a constant
<source>(8): note: failure was caused by a read of a variable outside its lifetime
<source>(8): note: see usage of 'i'
```

C++ is getting safer: constexpr/consteval!

```
consteval int f()
{
    int *p = nullptr;
    {
        int i = 1729;
        p = &i;
    }
    return *p;
}

constexpr int x = f();
```

<source>(11): error C2131: expression did not evaluate to a constant
<source>(8): note: failure was caused by a read of a variable outside its lifetime
<source>(8): note: see usage of 'i'

Caught by all major compilers!

C++ is getting safer: constexpr/consteval!

```
consteval int f()
{
    int *p = nullptr;
    {
        int i = 1729;
        p = &i;
    }
    return *p;
}

constexpr int x = f();
```

```
<source>(11): error C2131: expression did not evaluate to a constant
<source>(8): note: failure was caused by a read of a variable outside its lifetime
<source>(8): note: see usage of 'i'
```

Caught by all major compilers!

Constexpr all the things?

C++ is getting safer: P2718!

```
string readInput();

for(char c: readInput()) {
    if (c != ' ')
        cout << c;
}
```

C++ is getting safer: P2718!

```
string readInput();

for(char c: readInput()) {
    if (c != ' ')
        cout << c;
}
```

```
auto&& __range = readInput();
auto __begin = __range.begin();
auto __end = __range.end();
for(; __begin != __end; __begin++) {
    char c = *__begin;
    [...]
}
```

C++ is getting safer: P2718!

```
string readInput();

for(char c: readInput()) {
    if (c != ' ')
        cout << c;
}
```

```
auto&& __range = readInput();
auto __begin = __range.begin();
auto __end = __range.end();
for(; __begin != __end; __begin++) {
    char c = *__begin;
    [...]
}
```

C++ is getting safer: P2718!

```
string readInput();

for(char c: readInput()) {
    if (c != ' ')
        cout << c;
}
```

```
auto&& __range = readInput();
auto __begin = __range.begin();
auto __end = __range.end();
for(; __begin != __end; __begin++) {
    char c = *__begin;
    [...]
}
```

Lifetime analysis for everyone - Gábor Horváth & Matthias Gehre - CppCon 2019

C++ is getting safer: P2718!

```
string readInput();

for(char c: readInput()) {
    if (c != ' ')
        cout << c;
}
```

```
auto&& __range = readInput();
auto __begin = __range.begin();
auto __end = __range.end();
for(; __begin != __end; __begin++) {
    char c = *__begin;
    [...]
}
```

```
optional<string> mayReadInput();

for(char c: mayReadInput().value()) {
    if (c != ' ')
        cout << c;
}
```

Lifetime analysis for everyone - Gábor Horváth & Matthias Gehre - CppCon 2019

C++ is getting safer: P2718!

```
string readInput();

for(char c: readInput()) {
    if (c != ' ')
        cout << c;
}
```

```
auto&& __range = readInput();
auto __begin = __range.begin();
auto __end = __range.end();
for(; __begin != __end; __begin++) {
    char c = *__begin;
    [...]
}
```

```
optional<string> mayReadInput();

for(char c: mayReadInput().value()) {
    if (c != ' ')
        cout << c;
}
```

```
string& optional<string>::value();

auto&& __range = mayReadInput().value();
[...]
```

Lifetime analysis for everyone - Gábor Horváth & Matthias Gehre - CppCon 2019

C++ is getting safer: P2718!

```
string readInput();

for(char c: readInput()) {
    if (c != ' ')
        cout << c;
}
```

```
auto&& __range = readInput();
auto __begin = __range.begin();
auto __end = __range.end();
for(; __begin != __end; __begin++) {
    char c = *__begin;
    [...]
}
```

```
optional<string> mayReadInput();

for(char c: mayReadInput().value()) {
    if (c != ' ')
        cout << c;
}
```

```
string& optional<string>::value();

auto&& __range = mayReadInput().value();
[...]
```

Lifetime analysis for everyone - Gábor Horváth & Matthias Gehre - CppCon 2019

C++ is getting safer: P2718!

```
string readInput();

for(char c: readInput()) {
    if (c != ' ')
        cout << c;
}
```

```
auto&& __range = readInput();
auto __begin = __range.begin();
auto __end = __range.end();
for(; __begin != __end; __begin++) {
    char c = *__begin;
    [...]
}
```

```
optional<string> mayReadInput();

for(char c: mayReadInput().value()) {
    if (c != ' ')
        cout << c;
}
```

```
string& optional<string>::value();

auto&& __range = mayReadInput().value();
[...]
```

Available in C++23!

Lifetime analysis for everyone - Gábor Horváth & Matthias Gehre - CppCon 2019

C++ is getting safer: P2255!

```
std::tuple<const std::string&> x("hello");
```

C++ is getting safer: P2255!

```
std::tuple<const std::string&> x("hello");
```

```
reference_constructs_from_temporary
reference_converts_from_temporary
```

C++ is getting safer: P2255!

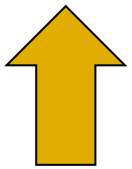
```cpp
std::tuple<const std::string&> x("hello");
```

```
reference_constructs_from_temporary
reference_converts_from_temporary
```

Available in C++23!

Approaches to safety

C++ is getting safer



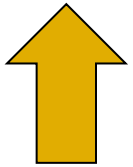
The lifetime safety toolbox

What comes next?

Approaches to safety

C++ is getting safer

The lifetime safety toolbox



What comes next?

Statement-local lifetime analysis

```
void prettyPrint(const string& thing) {
    string_view pretty = thing.empty() ? "<empty>" : thing;
    cout << pretty << '\n';
}
```

Statement-local lifetime analysis

```
void prettyPrint(const string& thing) {
    string_view pretty = thing.empty() ? "<empty>" : thing;
    cout << pretty << '\n';
}
```

Statement-local lifetime analysis

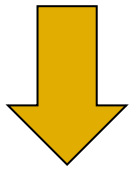
```
string_view pretty = thing.empty() ? "<empty>"   : thing;
```

Statement-local lifetime analysis

```
string_view pretty = thing.empty() ? "<empty>"   : thing;
                                     const char[6]   string
```

Statement-local lifetime analysis

```
string_view pretty = thing.empty() ? "<empty>"   : thing;
```

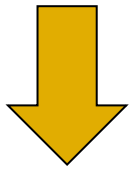


```
                                     const char[6]    string
```

Statement-local lifetime analysis

```
string_view pretty = thing.empty() ? "<empty>"   : thing;
```

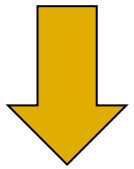
`const char[6]` `string`



```
if (thing.empty())
    pretty = string_view(string("<empty>"));
else
    pretty = string_view(string(thing));
```

Statement-local lifetime analysis

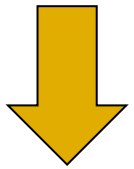
```
string_view pretty = thing.empty() ? "<empty>"   : thing;
                                     const char[6]   string
```



```
if (thing.empty())
    pretty = string_view(string("<empty>"));
else
    pretty = string_view(string(thing));
```

Statement-local lifetime analysis

```
string_view pretty = thing.empty() ? "<empty>"   : thing;
                                     const char[6]   string
```



```
if (thing.empty())
    pretty = string_view(string("<empty>"));
else
    pretty = string_view(string(thing));
```

C26815, C26816 in MSVC, -Wdangling-gsl in Clang, -Wdangling-reference in GCC

C26815 & C26816, -Wdangling-gsl

```
int &f() {
    std::stack<int> s;
    return s.top(); // warn
}

void g() {
    int &&r = *std::optional<int>();    // warn
    int &&r2 = *std::optional<int>(5);  // warn
    int &r3 = std::vector<int>().at(3); // warn
}

void h() {
    std::basic_string_view<char> sv;
    takeStringView(sv = std::basic_string<char>()); // warn
}
```

C26815 & C26816, -Wdangling-gsl

```
int &f() {
    std::stack<int> s;
    return s.top(); // warn
}

void g() {
    int &&r = *std::optional<int>();    // warn
    int &&r2 = *std::optional<int>(5);  // warn
    int &r3 = std::vector<int>().at(3); // warn
}

void h() {
    std::basic_string_view<char> sv;
    takeStringView(sv = std::basic_string<char>()); // warn
}
```

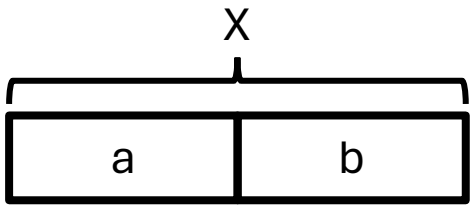
On-by-default in VS Background Code Analysis!

Lifetimebound annotation (P0936)

```
struct X { int a, b; };
const int& f(const X& x                          ) {
    return x.b;
}

const int& r = f(X());
```

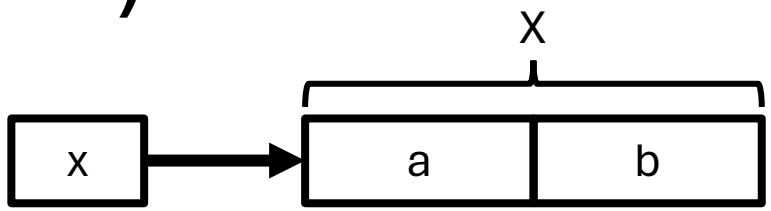
Lifetimebound annotation (P0936)

```
struct X { int a, b; };
const int& f(const X& x                    ) {
    return x.b;
}

const int& r = f(X());
```



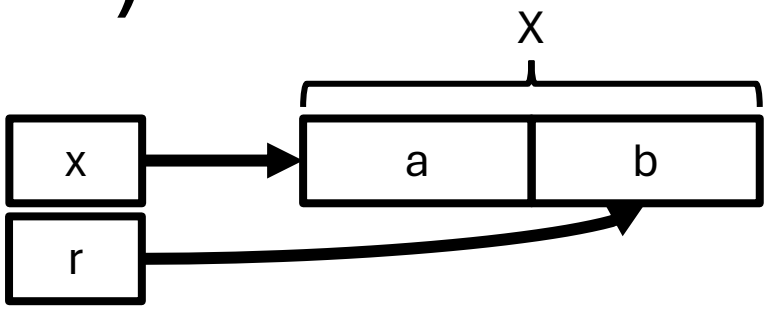
Lifetimebound annotation (P0936)

```
struct X { int a, b; };
const int& f(const X& x                    ) {
    return x.b;
}

const int& r = f(X());
```



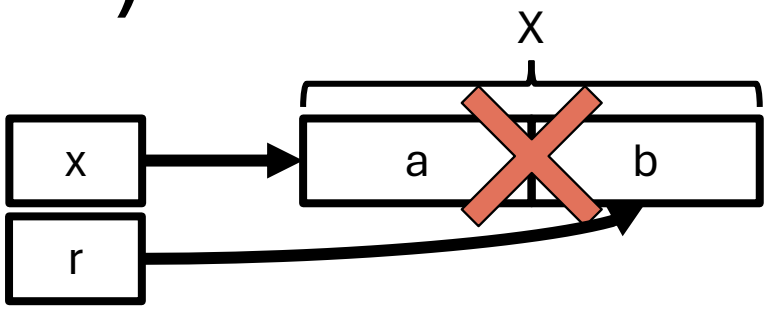
Lifetimebound annotation (P0936)

```
struct X { int a, b; };
const int& f(const X& x                    ) {
    return x.b;
}

const int& r = f(X());
```



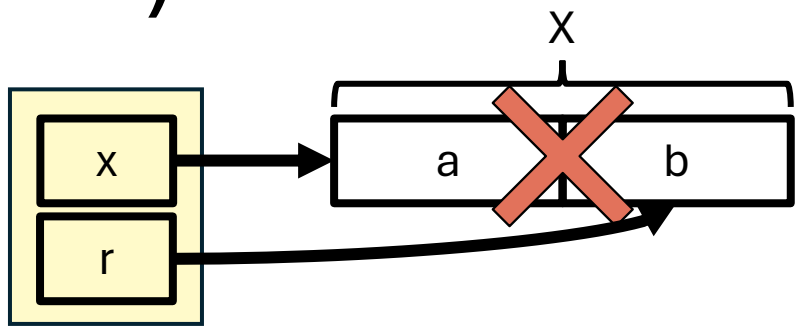
Lifetimebound annotation (P0936)

```
struct X { int a, b; };
const int& f(const X& x                    ) {
    return x.b;
}

const int& r = f(X());
```



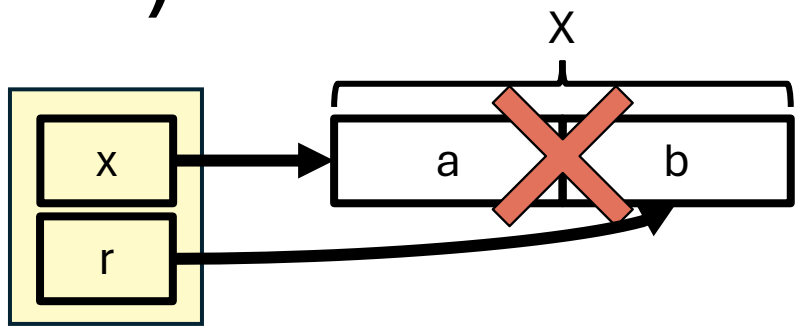
Lifetimebound annotation (P0936)

```
struct X { int a, b; };
const int& f(const X& x                    ) {
    return x.b;
}

const int& r = f(X());
```



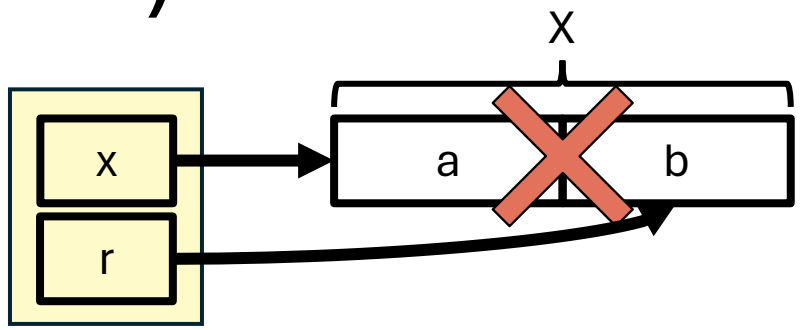
Lifetimebound annotation (P0936)

```
struct X { int a, b; };
const int& f(const X& x [[msvc::lifetimebound]] ) {
    return x.b;
}

const int& r = f(X());   // warn
```

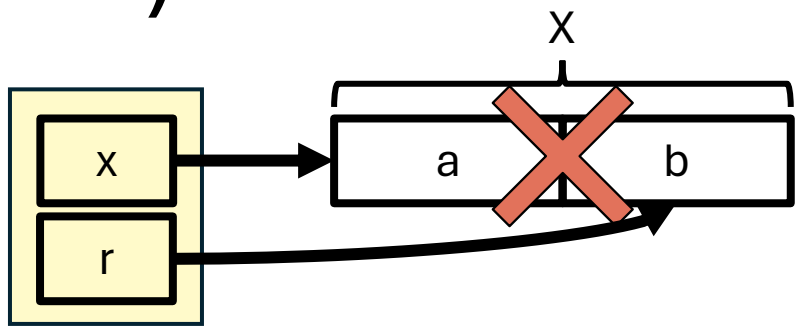


Lifetimebound annotation (P0936)

```
struct X { int a, b; };
const int& f(const X& x [[msvc::lifetimebound]] ) {
    return x.b;
}

const int& r = f(X());   // warn

const char* f(std::string_view x [[msvc::lifetimebound]]);
```

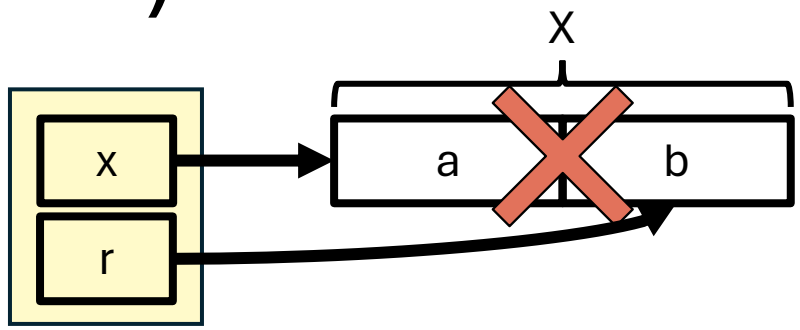


Lifetimebound annotation (P0936)




```
struct X { int a, b; };
const int& f(const X& x [[msvc::lifetimebound]] ) {
    return x.b;
}

const int& r = f(X());   // warn

const char* f(std::string_view x [[msvc::lifetimebound]]);

const int& passthrough(const int& param [[msvc::lifetimebound]]);
const int& min(const int& param1 [[msvc::lifetimebound]],
               const int& param2 [[msvc::lifetimebound]]);

void deeply_nested() {
    int i = 5;
    const auto& a = passthrough(min(passthrough(i), passthrough(5)));
    const auto& b = passthrough(min(passthrough(5), passthrough(i)));
    const auto& c = passthrough(min(passthrough(i), passthrough(i)));
}
```

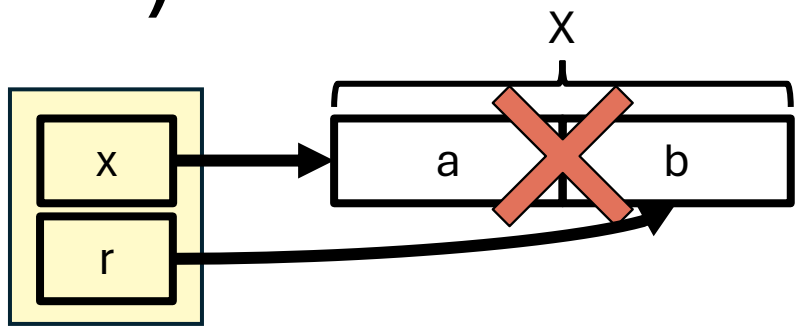
Lifetimebound annotation (P0936)




```
struct X { int a, b; };
const int& f(const X& x [[msvc::lifetimebound]] ) {
    return x.b;
}

const int& r = f(X());   // warn

const char* f(std::string_view x [[msvc::lifetimebound]]);

const int& passthrough(const int& param [[msvc::lifetimebound]]);
const int& min(const int& param1 [[msvc::lifetimebound]],
               const int& param2 [[msvc::lifetimebound]]);

void deeply_nested() {
    int i = 5;
    const auto& a = passthrough(min(passthrough(i), passthrough(5))); // warn
    const auto& b = passthrough(min(passthrough(5), passthrough(i))); // warn
    const auto& c = passthrough(min(passthrough(i), passthrough(i)));
}
```

Lifetimebound annotation (P0936)

```
struct X { int a, b; };
const int& f(const X& x [[msvc::lifetimebound]] ) {
    return x.b;
}

const int& r = f(X());   // warn

const char* f(std::string_view x [[msvc::lifetimebound]]);

const int& passthrough(const int& param [[msvc::lifetimebound]]);
const int& min(const int& param1 [[msvc::lifetimebound]],
               const int& param2 [[msvc::lifetimebound]]);

void deeply_nested() {
    int i = 5;
    const auto& a = passthrough(min(passthrough(i), passthrough(5)));
    const auto& b = passthrough(min(passthrough(5), passthrough(i)));
    const auto& c = passthrough(min(passthrough(i), passthrough(i)));
}
```



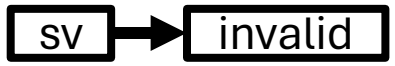
Available in Clang and MSVC

Flow-sensitive lifetime analysis

```
char hello(bool b) {
    string_view sv;
    if (b)
        sv = "Hello CppCon!";
    ...
    if (cond())
        return sv[0];
    ...
}
```

Flow-sensitive lifetime analysis

```
char hello(bool b) {
    string_view sv;
    if (b)
        sv = "Hello CppCon!";
    ...
    if (cond())
        return sv[0];
    ...
}
```

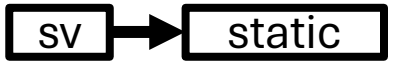


Flow-sensitive lifetime analysis

```
char hello(bool b) {
    string_view sv;
    if (b)
        sv = "Hello CppCon!";
    ...
    if (cond())
        return sv[0];
    ...
}
```

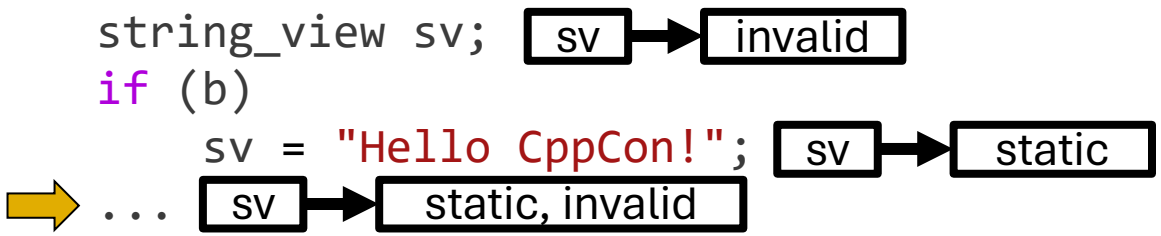
Flow-sensitive lifetime analysis

```
char hello(bool b) {
    string_view sv;
    if (b)
        sv = "Hello CppCon!";
    ...
    if (cond())
        return sv[0];
    ...
}
```



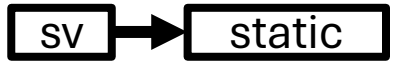
Flow-sensitive lifetime analysis

```
char hello(bool b) {
    string_view sv;
    if (b)
        sv = "Hello CppCon!";
    ...
    if (cond())
        return sv[0];
    ...
}
```



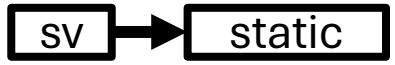
Flow-sensitive lifetime analysis

```
char hello(bool b) {
    string_view sv;
    if (b)
        sv = "Hello CppCon!";
    ...
    if (cond())
        return sv[0];
    ...
}
```



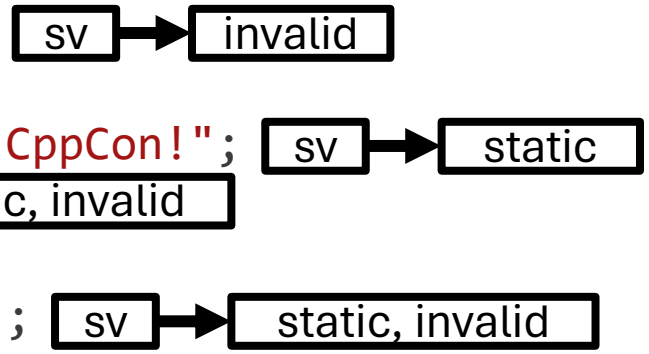
Flow-sensitive lifetime analysis

```
char hello(bool b) {
    string_view sv;
    if (b)
        sv = "Hello CppCon!";
    ...
    if (cond())
    ➡ return sv[0];
    ...
}
```



Flow-sensitive lifetime analysis

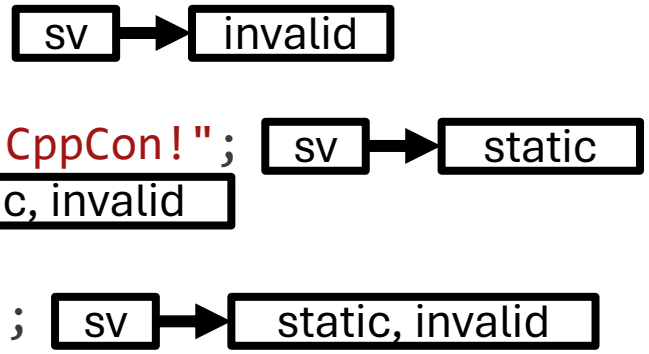
```
char hello(bool b) {
    string_view sv;
    if (b)
        sv = "Hello CppCon!";
    ...
    if (cond())
        return sv[0];
    ...
}
```



Optimist	Pessimist
It can be fine, don't warn!	It can go wrong, warn!
Better at pointing out real problems	Better at avoiding disaster

Flow-sensitive lifetime analysis

```
char hello(bool b) {
    string_view sv;
    if (b)
        sv = "Hello CppCon!";
    ...
    if (cond())
        return sv[0];
    ...
}
```



Optimist	Pessimist
It can be fine, don't warn!	It can go wrong, warn!
Better at pointing out real problems	Better at avoiding disaster

[High-confidence Lifetime Checks in Visual Studio version 17.5 Preview 2 - C++ Team Blog]

Could the optimist agree with the pessimist?

Could the optimist agree with the pessimist?

```
int* g(int* a, int* b);

void h() {
    int x;
    int* p = g(&x, nullptr);
    *p = 42;
}
```

Could the optimist agree with the pessimist?

```
int* g(int* a, int* b);

void h() {
    int x;
    int* p = g(&x, nullptr);
    *p = 42;
}
```

Could the optimist agree with the pessimist?

```
int* g(int* a, int* b);

void h() {
    int x;
    int* p = g(&x, nullptr);
    *p = 42;
}
```

Could the optimist agree with the pessimist?

```
int* g(int* a, int* b)
    [[post: lifetime(Return, {a})]];
void h() {
     int x;
     int* p = g(&x, nullptr);
     *p = 42;
}
```

Could the optimist agree with the pessimist?

```
int* g(int* a, int* b)
    [[post: lifetime(Return, {a})]];
void h() {
     int x;
     int* p = g(&x, nullptr);
     *p = 42;
}
```

```
void g(          int** q);
void f() {
     int* p;
     g(&p);
     *p = 42;
}
```

Could the optimist agree with the pessimist?

```
int* g(int* a, int* b)
    [[post: lifetime(Return, {a})]];
void h() {
    int x;
    int* p = g(&x, nullptr);
    *p = 42;
}
```

```
void g(          int** q);
void f() {
    int* p;
    g(&p);
    *p = 42;
}
```

Could the optimist agree with the pessimist?

```
int* g(int* a, int* b)
    [[post: lifetime(Return, {a})]];
void h() {
     int x;
     int* p = g(&x, nullptr);
     *p = 42;
}
```

```
void g(          int** q);
void f() {
     int* p;
     g(&p);
     *p = 42;
}
```

Could the optimist agree with the pessimist?

```
int* g(int* a, int* b)
    [[post: lifetime(Return, {a})]];
void h() {
     int x;
     int* p = g(&x, nullptr);
     *p = 42;
}
```

```
void g( _Outptr_ int** q);
void f() {
     int* p;
     g(&p);
     *p = 42;
}
```

Could the optimist agree with the pessimist?

```
int* g(int* a, int* b)
    [[post: lifetime(Return, {a})]];
void h() {
     int x;
     int* p = g(&x, nullptr);
     *p = 42;
}
```

```
void g( _Outptr_ int** q);
void f() {
     int* p;
     g(&p);
     *p = 42;
}
```

```
int& f(std::vector<int>& v) {
     int& before_last = v.back();
     v.push_back(42);
     return before_last;
}
```

Could the optimist agree with the pessimist?

```
int* g(int* a, int* b)
    [[post: lifetime(Return, {a})]];
void h() {
     int x;
     int* p = g(&x, nullptr);
     *p = 42;
}
```

```
void g( _Outptr_ int** q);
void f() {
     int* p;
     g(&p);
     *p = 42;
}
```

```
int& f(std::vector<int>& v) {
     int& before_last = v.back();
     v.push_back(42);
     return before_last;
}
```

Could the optimist agree with the pessimist?

```
int* g(int* a, int* b)
    [[post: lifetime(Return, {a})]];
void h() {
     int x;
     int* p = g(&x, nullptr);
     *p = 42;
}
```

```
void g( _Outptr_ int** q);
void f() {
     int* p;
     g(&p);
     *p = 42;
}
```

```
int& f(std::vector<int>& v) {
     int& before_last = v.back();
     v.push_back(42);
     return before_last;
}
```

Other lifetime checks

- Dereferencing empty optional
  - Dereferencing nullptr vs std::optional::value
  - C26829, C26830, C26859, C26860
  - bugprone-unchecked-optional-access
- Use after move
  - C26800
  - bugprone-use-after-move
  - cplusplus.Move

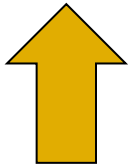
Other lifetime checks

- Dereferencing empty optional
  - Dereferencing nullptr vs std::optional::value
  - C26829, C26830, C26859, C26860
  - bugprone-unchecked-optional-access
- Use after move
  - C26800
  - bugprone-use-after-move
  - cplusplus.Move

Approaches to safety

C++ is getting safer

The lifetime safety toolbox



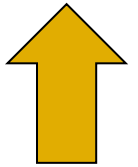
What comes next?

Approaches to safety

C++ is getting safer

The lifetime safety toolbox

What comes next?



Short-term investigations at Microsoft

- Closing the gap between statement-local lifetime checks
- Move high-confidence lifetime warnings out of experimental
- Extend lifetimebound support to flow-sensitive checks
- Compare Crubit and the Core Guidelines' Lifetime Safety Profile
- Herb plans to revise the Lifetime Safety Profile as part of Cpp2

Future of C++?

Future of C++?

C++ Core Guidelines' Lifetime Safety Profile

Future of C++?

P2771:
Thomas Neumann's
Dependency Annotations

C++ Core
Guidelines' Lifetime
Safety Profile

Future of C++?

P2771:
Thomas Neumann's
Dependency Annotations

C++ Core
Guidelines' Lifetime
Safety Profile

Crubit: Adopting
Rust's Type System

Future of C++?

P2771:
Thomas Neumann's
Dependency Annotations

C++ Core
Guidelines' Lifetime
Safety Profile

Crubit: Adopting
Rust's Type System

Hylo (formerly Val):
Mutable Value Semantics

Future of C++?

P2771:
Thomas Neumann's
Dependency Annotations

Vale:
Generational References +
Simplified borrowing

C++ Core
Guidelines' Lifetime
Safety Profile

Crubit: Adopting
Rust's Type System

Hylo (formerly Val):
Mutable Value Semantics

Future of C++?

P2771:
Thomas Neumann's
Dependency Annotations

Vale:
Generational References +
Simplified borrowing

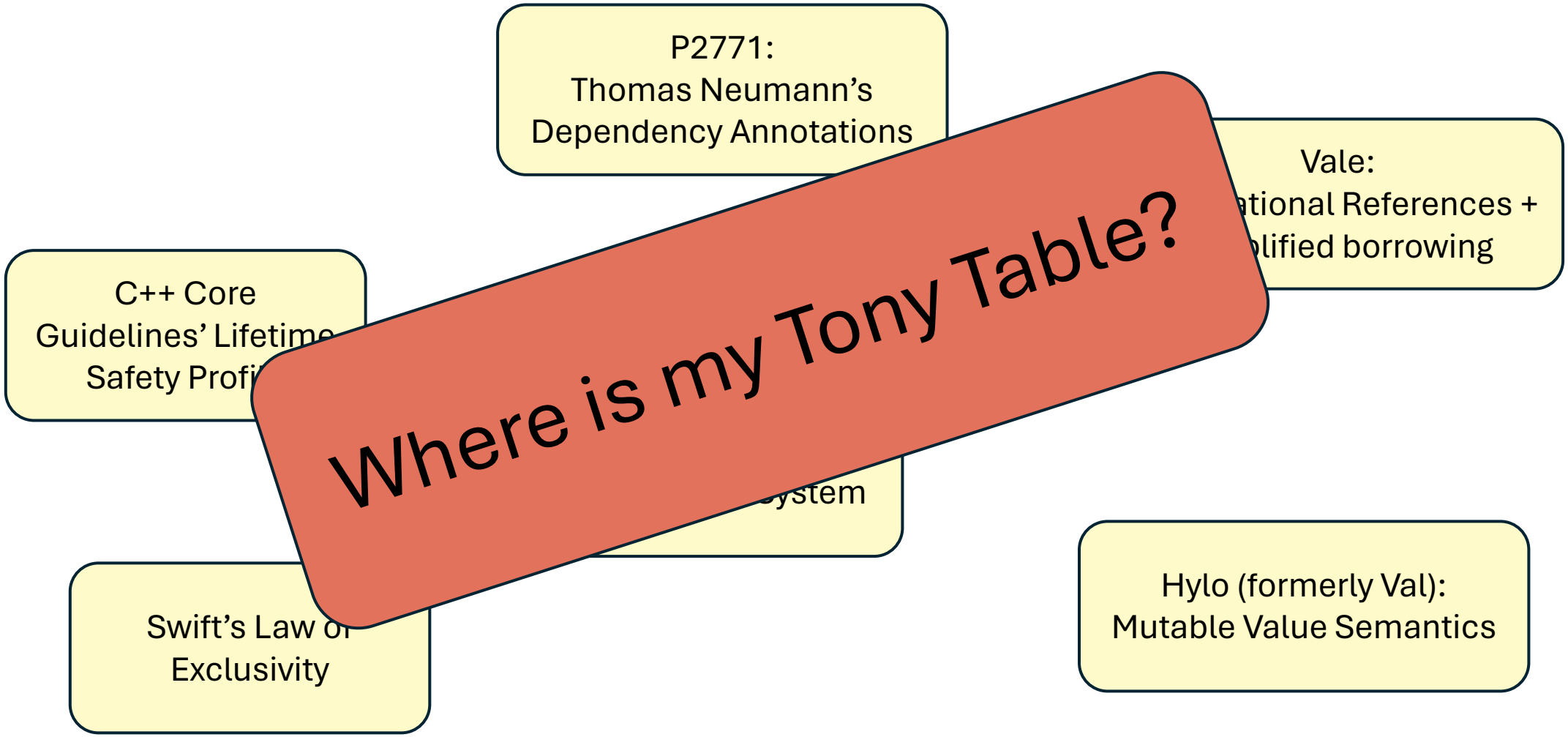
C++ Core
Guidelines' Lifetime
Safety Profile

Crubit: Adopting
Rust's Type System

Swift's Law of
Exclusivity

Hylo (formerly Val):
Mutable Value Semantics

Future of C++?


P2771:
Thomas Neumann's
Dependency Annotations
Vale:
ational References +
olified borrowing
C++ Core
Guidelines' Lifetime
Safety Profi
ystem
Where is my Tony Table?
Swift's Law of
Exclusivity
Hylo (formerly Val):
Mutable Value Semantics

Building a safe subset from scratch?

Building a safe subset from scratch?

Type Systems for Memory Safety (borretti.me)

Building a safe subset from scratch?

Type Systems for Memory Safety (borretti.me)

```
let num = 5;

let r1 = &num as *const i32;
let r2 = &num;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", r2);
}
```

Building a safe subset from scratch?

Type Systems for Memory Safety (borretti.me)

```
let num = 5;

let r1 = &num as *const i32;
let r2 = &num;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", r2);
}
```

Building a safe subset from scratch?

Type Systems for Memory Safety (borretti.me)

```
let num = 5;

let r1 = &num as *const i32;
let r2 = &num;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", r2);
}
```

Building a safe subset from scratch?

Type Systems for Memory Safety (borretti.me)

```
let num = 5;

let r1 = &num as *const i32;
let r2 = &num;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", r2);
}
```

gsl::span

Building a safe subset from scratch?

Type Systems for Memory Safety (borretti.me)

```
let num = 5;

let r1 = &num as *const i32;
let r2 = &num;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", r2);
}
```

gsl::span

- Add a way to define affine/linear types
  - Almost like destructive moves
  - Templates are a challenge

Building a safe subset from scratch?

Type Systems for Memory Safety (borretti.me)

```
let num = 5;

let r1 = &num as *const i32;
let r2 = &num;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", r2);
}
```

gsl::span

- Add a way to define affine/linear types
  - Almost like destructive moves
  - Templates are a challenge
- Add a safe reference with simplified borrowing rules

Building a safe subset from scratch?

Type Systems for Memory Safety (borretti.me)

```
let num = 5;

let r1 = &num as *const i32;
let r2 = &num;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", r2);
}
```

gsl::span

- Add a way to define affine/linear types
  - Almost like destructive moves
  - Templates are a challenge
- Add a safe reference with simplified borrowing rules
- Gradually make safe references more expressive

Building a safe subset from scratch?

Type Systems for Memory Safety (borretti.me)

```
let num = 5;

let r1 = &num as *const i32;
let r2 = &num;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", r2);
}
```

gsl::span

- Add a way to define affine/linear types
  - Almost like destructive moves
  - Templates are a challenge
- Add a safe reference with simplified borrowing rules
- Gradually make safe references more expressive
- No profiles/unsafe markers needed

Building a safe subset from scratch?

Type Systems for Memory Safety (borretti.me)

```rust
let num = 5;

let r1 = &num as *const i32;
let r2 = &num;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", r2);
}
```

gsl::span

- Add a way to define affine/linear types
  - Almost like destructive moves
  - Templates are a challenge
- Add a safe reference with simplified borrowing rules
- Gradually make safe references more expressive
- No profiles/unsafe markers needed

Or just write new code in Rust and focus on interop and hardening old code?

Conclusions

- C++ is getting safer
- Wide variety of solutions
  - Safer language spec
  - Dynamic analysis
  - Static analysis
    - Pessimistic
    - Optimistic
- We need you!
  - Adopt the tools, report the bugs
  - Experiment with ideas, share the experience
  - Participate in SG23

Enjoy the rest of the conference!

Come by our booth and join #visual_studio channel on CppCon Discord **https://aka.ms/cppcon/discord**

- Meet the Microsoft C++ team
- Ask any questions
- Discuss the latest announcements

Take our survey

Win prizes

https://aka.ms/cppcon/lifetime

Our sessions

Monday 2nd

- **Informal Birds of a Feather for Cpp2/cppfront –** Herb Sutter

Tuesday 3rd

- **What's New in Visual Studio –** David Li & Mryam Girmay

Thursday 5th

- **Cooperative C++ Evolution: Towards a Typescript for C++** – Herb Sutter (Keynote)
- **How Visual Studio Code Can Help You Develop More Efficiently in C++** – Alexandra Kemper & Sinem Akinci
- **Regular, Revisited** – Victor Ciura

Friday 6th

- **Getting Started with C++ –** Michael Price