

+ 25

Building Secure C++ Applications

A Practical End-to-End Approach

CHANDRANATH BHATTACHARYYA
& BHARAT KUMAR



20
25



Introduction

Chandranath Bhattacharyya

Principal Software Engineer

Microsoft Edge Browser **Consumer Team** since 2018



Bharat Kumar

Principal Software Engineer

Microsoft Edge Browser **Security Team** since 2018



Quick Note on Questions

3



We kindly ask that you hold questions until the end of the presentation.

If your question relates to a specific slide, please note its number (top right corner) so we can refer to it during the Q&A.

Why this talk?

C++ is a powerful language, but it's also **memory-unsafe** and has a **wide range of undefined behaviors**.

We can't simply rewrite the massive amount of existing C++ code into a memory-safe language overnight.

Microsoft Edge is built on Chromium, which is predominantly C++.

Fortunately, we can make significant progress without a full rewrite.

A practical approach to C++ Safety

[C++ safety, in context - Herb Sutter: 2024-03-11](#)

Four key safety categories:

- **Bounds:** *Preventing out-of-bounds access to arrays and memory.*
- **Lifetime:** *Ensuring objects are used only while they are valid.*
- **Initialization:** *Guaranteeing variables are properly initialized before use.*
- **Type:** *Preventing incorrect type conversions and operations.*

A 98% reduction across those four categories is achievable in new/updated C++ code, and partially in existing code.

Our approach has the same spirit. There is no silver bullet. We chip away at the problem to make to incrementally better, including moving some portions to memory safe languages like Rust.

Our Approach in Microsoft Edge

We'll walk through how the Edge team tackles safety across Core safety categories:

- Bounds
- Lifetime
- Initialization
- Type

And touch upon our effort in the following categories:

- Thread safety
- Definition safety
- Reducing other undefined behaviors

This talk focuses on **safety** - minimize exploitable behavior and undefined states.

Safe code \neq correct code

We'll also touch on some strategies we use to improve overall code correctness.


Safety Across the Development Lifecycle

We'll explore safety practices across each phase:

- Design
- Implementation
- Code Review
- Continuous Integration (CI) Pipeline
- After Release

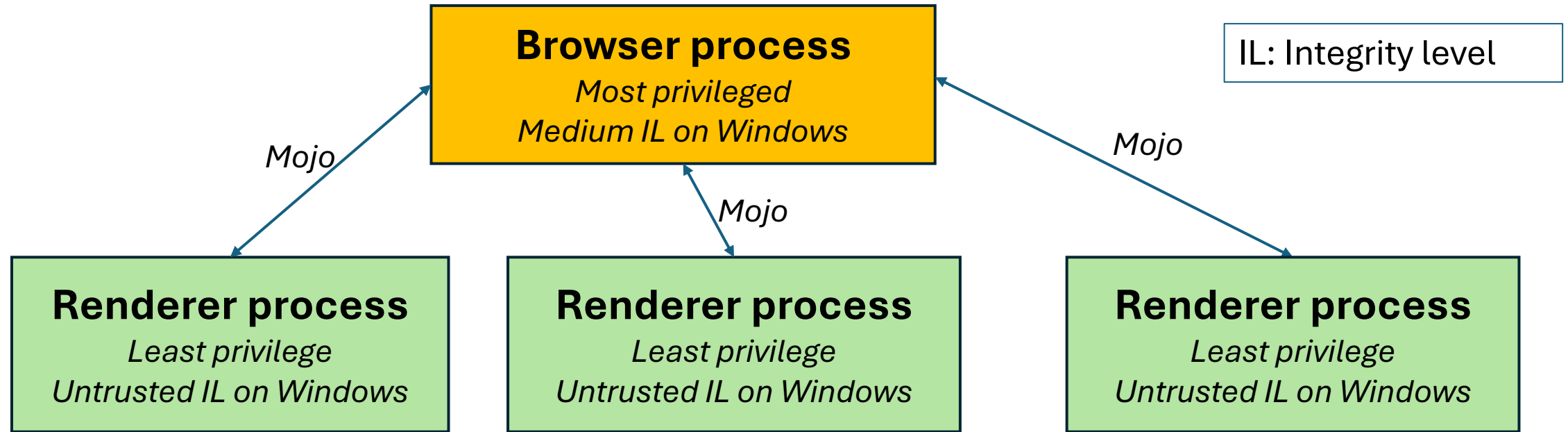
Many of the techniques we'll discuss were developed by the **incredible open-source community at Chromium**. We're excited to share these with you.

Our C++ Environment

- Our project uses **C++20** with the **Clang compiler** and **libc++** standard library.
-  **Exception handling is disabled.**
- Our discussions will focus on the behavior of this specific toolchain, not on other compilers like GCC or MSVC.

Design Phase

Chromium: Process Architecture: High level

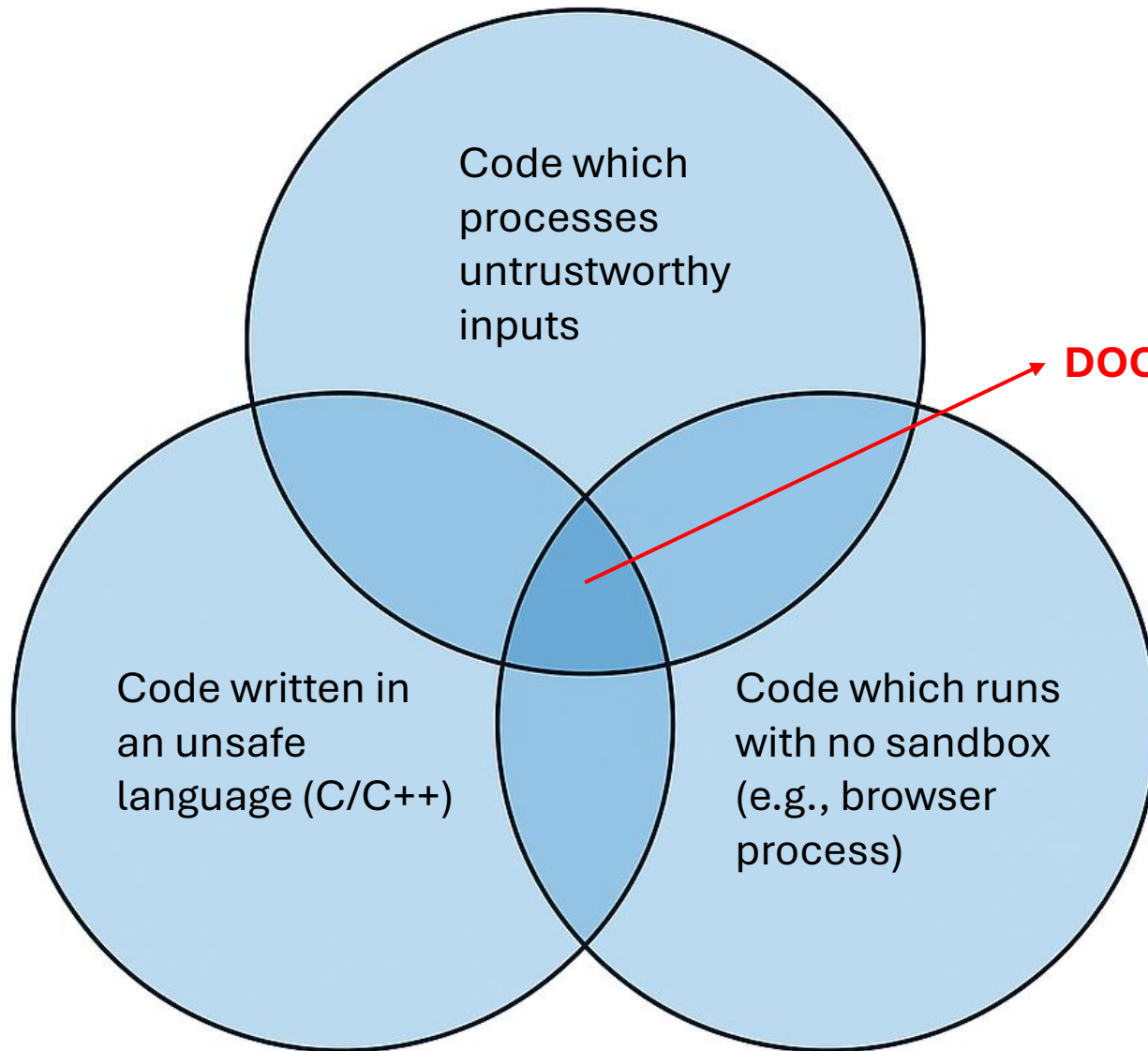


Mojo: The Mojo system API provides a small suite of low-level IPC primitives: message pipes, data pipes, and shared buffers.

Renderer processes are always assumed to be compromised

Use mojo messages to broker (browser process) for any privileged operations

Chromium Rule of 2



Chromium's [rule of 2](#),
Pick no more than 2 of:

- untrustworthy inputs
- unsafe implementation language
- high privilege.

- JSON parsing in C++ code in browser process ✗
- JSON parsing in C++ code in sandboxed utility process ✓
- JSON parsing in RUST in browser process ✓





Security review process

- Security champions along with security team review the design of every feature.
- Are we introducing any new entry points for untrusted data?
 - If yes, do we need a new process to handle this data?
- Are there any new Inter-Process-Communication / Mojo communications required?
- Do we need additional fuzzing coverage?

Bounds Safety

libc++ with hardening
-Wunsafe-buffer-usage

libc++ with hardening

- libc++ offers four hardening modes to catch violations of library preconditions. Each mode balances runtime performance and safety differently:
 - **Unchecked mode / none:**  All checks disabled. Fastest, but no protection against undefined behavior.
 - **Fast mode:**  Enables essential, low-cost security checks. Recommended for most production builds.
 - **Extensive mode:**  Includes fast mode's checks + additional non-critical ones with moderate overhead. Best for production builds requiring broader coverage.
 - **Debug mode:**  Enables all checks (including internal and heuristic ones). Significant overhead. Use only for testing, CI, or local development—not in production.

libc++ with hardening

Compiler options for hardening:

- `-D_LIBCPP_HARDENING_MODE=_LIBCPP_HARDENING_MODE_NONE`
- `-D_LIBCPP_HARDENING_MODE=_LIBCPP_HARDENING_MODE_FAST`
- `-D_LIBCPP_HARDENING_MODE=_LIBCPP_HARDENING_MODE_EXTENSIVE`
- `-D_LIBCPP_HARDENING_MODE=_LIBCPP_HARDENING_MODE_DEBUG`


Category name	fast	extensive	debug
valid-element-access	✓	✓	✓
valid-input-range	✓	✓	✓
non-null	✗	✓	✓
non-overlapping-ranges	✗	✓	✓
valid-deallocation	✗	✓	✓
valid-external-api-call	✗	✓	✓
compatible-allocator	✗	✓	✓
argument-within-domain	✗	✓	✓
pedantic	✗	✓	✓
semantic-requirement	✗	✗	✓
internal	✗	✗	✓
uncategorized	✗	✓	✓

We use EXTENSIVE in our project.

libc++ hardening handles “more” than just “bounds safety”.

C++ Standard Library hardening

C++26: Add supports for [standard library hardening](#).


Keynote: Three Cool Things in C++

think-cell


C++26: Standard library hardening

C++26 “stdlib hardening” adds bounds checking for common std:: type operations:

Sequence containers (e.g., vector, array)	operator[], front, back, pop_front, pop_back
string family	operator[], front, back, pop_back
string_view family	operator[], front, back, remove_prefix, remove_suffix
span	operator[], front, back, first, last, subspan, constructors
mdspan	operator[], constructors
bitset	operator[]
valarray	operator[]

And null/empty checking for:

optional	operator->, operator*
expected	operator->, operator*, error



Herb Sutter

8

cpponseas.uk

2025

libc++ offers more comprehensive hardening than what's currently proposed for C++26.

libc++ with hardening: Configuring failure

```
[[noreturn]] inline _LIBCPP_HIDE_FROM_ABI void __libc++_hardening_failure() {  
    __builtin_trap();  
}  
  
#define _LIBCPP_ASSERTION_HANDLER(message) \  
    ((void)message, __libc++_hardening_failure())
```

__builtin_trap causes the program to stop its execution abnormally.

It triggers an immediate CPU trap (e.g., **ud2: Undefined Instruction** on x86) causing an immediate and unconditional crash.

This ensures the program halts immediately without giving an attacker a chance to manipulate the program's state.

When you see "*traps*" in later slides, we are referring to this non-exploitable crash caused by `__builtin_trap()`.

libc++ hardening: valid-element-access

Checks that any attempts to access a container element, whether through the container object or through an iterator, are valid and do not attempt to go out of bounds or otherwise access a non-existent element.

```
void ProblemVector(std::vector<int> v) {
    // Will fire when called on empty vector.
    const auto f = v.front();
    std::cout << f << '\n';

    // Will fire when called on empty vector.
    const auto b = v.back();
    std::cout << b << '\n';

    // Will fire when called on empty vector.
    v.pop_back();

    v.push_back(1);
    v.push_back(2);
    // Will fire when called with index out of range.
    std::cout << v[2] << '\n';

    // Will fire when called with end().
    auto it = v.erase(v.end());
    std::cout << (it != v.end()) << '\n';
}

int main() { ProblemVector({}); }
```

vector:

- *operator[]*
- *front(), back(), pop_back()* – empty vector
- *erase()* – called with *end()*

Each of these calls will “trap” for these scenarios.

libc++ hardening: valid-element-access

```
void ProblemString(std::string s) {
    // Will fire when called on empty string.
    const auto f = s.front();
    std::cout << "s.front(): " << f << '\n';

    // Will fire when called on empty string.
    const auto b = s.back();
    std::cout << "s.back(): " << b << '\n';

    // Will fire when called on empty string.
    s.pop_back();
    s += "ab";

    // Will fire when called with index out of range.
    std::cout << "s[3]: " << s[3] << '\n';

    // Will fire when called with end().
    auto it = s.erase(s.end());
    std::cout << "(it != s.end()): " << (it != s.end()) << '\n';
}

int main() {
    ProblemString({});
}
```

string:

- *operator[]*
- *front(), back(), pop_back()* – empty vector
- *erase()* – called with *end()*

Each of these calls will “trap” for these scenarios.

Clang: **-std=c++20 -stdlib=libc++ -O3**: No crash.

```
s.front():
s.back():
s[3]:
(it != s.end()): 0
```

libc++ hardening: valid-element-access

```
void ProblemOptional(std::optional<int> o) {
    // Will flag when called with empty optional.
    std::cout << "*o: " << *o << '\n';
    // Will flag when called with empty optional.
    std::cout << "o->operator(): " << (void*)(o.operator->()) << '\n';
}

int main() {
    ProblemOptional({});
}
```

optional

- operator->(), operator*() – Called on empty

Each of these calls will “trap” for these scenarios.

std::optional is “not” a container, but it falls under the ambit of valid-element-access category.

Clang: **-std=c++20 -stdlib=libc++ -O3**: No crash.

```
*o: 0
o->operator(): 0x7ffd14d77ac8
```

std::expected is “not” a container, but it falls under the ambit of valid-element-access category.

expected:

- operator->, operator*: When expected has no “success” object.
- error() : When expected has no “error” object.

libc++ hardening: valid-element-access

Other common containers which are covered are:

- `std::string`
- `std::string_view`
- `std::span`
- `std::array`
- `std::list`
- `std::deque`

libc++ hardening: valid-element-access

```
void ProblemMinMax() {
    std::initializer_list<int> l{};
    // Will flag for empty range being passed.
    const auto [min, max] = std::ranges::minmax(l);
    std::cout << "l: min: " << min << ", max: " << max << '\n';
}

void ProblemMin() {
    std::initializer_list<int> l{};
    // Will flag for empty range being passed.
    const auto min = std::ranges::min(l);
    std::cout << "min: " << min << '\n';
}

void ProblemMax() {
    std::initializer_list<int> l{};
    // Will flag for empty range being passed.
    const auto max = std::ranges::max(l);
    std::cout << "max: " << max << '\n';
}

int main() {
    ProblemMinMax();
    ProblemMin();
    ProblemMax();
}
```

algorithm:

- min, max, minmax: empty range.

Each of these calls will “trap” for these scenarios.

libc++ hardening: valid-input-range

Checks that ranges (whether expressed as an iterator pair, an iterator and a sentinel, an iterator and a count, or a `std::range`) given as input to library functions are valid: - the sentinel is reachable from the begin iterator

```
void ProblemVector(std::vector<int> v) {
    // Will flag when called with first > last.
    v.erase(v.end(), v.begin());
}

int main() {
    std::vector<int> v;
    v.push_back(10);
    ProblemVector(v);
}
```

vector:

- erase: When called with first > last.

It will “trap” for this scenario.

Clang: `-std=c++20 -stdlib=libc++ -O3`: No crash.

```
void ProblemString(std::string s) {
    // Will flag when called with first > last.
    s.erase(s.end(), s.begin());
}

int main() {
    std::string s("hello");
    ProblemString(s);
}
```

string:

- erase: When called with first > last.

It will “trap” for this scenario.

Clang: `-std=c++20 -stdlib=libc++ -O3`: No crash.

libc++ hardening: non-null

Checks that the **pointer being dereferenced is not null**. On most modern platforms, the zero address does not refer to an actual location in memory, so **a null pointer dereference would not compromise the memory security** of a program (however, it is still undefined behavior that can result in **strange errors due to compiler optimizations**).

```
void ProblemStringView(const char* p) {  
    // Will flag if p is nullptr.  
    std::string_view s{p};  
    std::cout << s << '\n';  
}  
  
void ProblemStringView2(const char* p, size_t n) {  
    // Will flag if p is nullptr and n != 0.  
    std::string_view s{p, n};  
    std::cout << s << '\n';  
}  
  
int main() {  
    std::string_view sv;  
    ProblemStringView2(sv.data(), 1u);  
}
```

string_view:

- string_view(const char*) : If called with nullptr at runtime.
- string_view(const char*, size_t len) : If called with nullptr at runtime and len != 0.

It will “trap” for these scenarios.

libc++ hardening: non-null


```
void ProblemString(const char* p) {  
    // Will flag if p is nullptr.  
    std::string s{p};  
    std::cout << s << '\n';  
}  
  
void ProblemString2(const char* p, size_t n) {  
    // Will flag if p is nullptr and n != 0.  
    std::string s{p, n};  
    std::cout << s << '\n';  
}  
  
int main() {  
    std::string_view sv;  
    ProblemString(sv.data());  
    ProblemString2(sv.data(), 1u);  
}
```

string

- string(const char*) : If called with nullptr at runtime.
- string(const char*, size_t len) : If called with nullptr at runtime and len != 0.

It will “trap” for these scenarios.

Is more bounds safety necessary?



```
void Print(const int* arr, size_t len)
{
    for (size_t i = 0; i <= len; ++i) {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}

int main() {
    const int arr[]{1, 2, 3, 4, 5};
    Print(arr, std::size(arr));
}
```

1 2 3 4 5 392233686

This is undefined behavior.


This code may crash at runtime with sanitized builds.

Since production builds are non-sanitized, this undefined behavior may lead to exploitation.

-Wunsafe-buffer-usage attempts to convert such code constructs to compile time errors.

It is a compile time flag.

-Wunsafe-buffer-usage




```
void Print(const int* arr, size_t len)
{
    for (size_t i = 0; i <= len; ++i) {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}

int main() {
    const int arr[]{1, 2, 3, 4, 5};
    Print(arr, std::size(arr));
}
```

1	2	3	4	5	392233686
---	---	---	---	---	-----------

-Wunsafe-buffer-usage



```
void Print(const int* arr, size_t len) {
    for (size_t i = 0; i <= len; ++i) {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}

int main() {
    const int arr[]{1, 2, 3, 4, 5};
    Print(arr, std::size(arr));
}
```

```
error: unsafe buffer access [-Werror, -Wunsafe-buffer-usage]
    std::cout << arr[i] << ' ';
                  ^~~
```

`i <= len`: The exact error location is ***not pointed out***.

Coding patterns which “may cause” problems are flagged at compile time.

-Wunsafe-buffer-usage

Coding patterns which “may cause” problems are flagged at compile time.

```
int main() {
    const int arr[] = {1, 2, 3};
    for (int i = 0; i < std::size(arr); ++i) {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}
```



error: unsafe buffer access [-Werror, -Wunsafe-buffer-usage]
std::cout << arr[i] << ' ';
 ^{^~~}

This code is fine. But it is the pattern which is pointed out.

[LLVM safe buffers documentation](#) contains more information.


Some points from the above link:

- Buffer operations should never be performed over raw pointers. Warning is emitted for:
 - Array indexing with [].
 - Pointer arithmetic.
 - Bounds-unsafe standard C functions such as memcpy.
- All buffers need to be encapsulated into safe containers and view types:
 - Containers: std::array, std::vector, std::string.
 - Views: std::span, std::string_view.

-Wunsafe-buffer-usage


Buffer operations should never be performed over raw pointers. Warning is emitted for:

- Array indexing with [].
- Pointer arithmetic.




```
int main() {
    const int arr[] = {1, 2, 3};
    for (int i = 0; i < std::size(arr); ++i) {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}
```

```
error: unsafe buffer access [-Werror, -Wunsafe-buffer-usage]
      std::cout << arr[i] << ' ';
                  ^~~
```



```
int main() {
    const char str[] = "Hello";
    const char* pstr = str;
    while (*pstr) {
        std::cout << *pstr++;
    }
    std::cout << '\n';
}
```


```
error: unsafe pointer arithmetic [-Werror, -Wunsafe-buffer-usage]
      std::cout << *pstr++;
                  ^~~~
```




```
int main() {
    const int arr[] = {1, 2, 3};
    for (const auto i : arr) {
        std::cout << i << ' ';
    }
    std::cout << '\n';
}
```

Moving to range based for loop fixes the issue.

Better to convert to array.




```
int main() {
    static constexpr std::array kArr{1, 2, 3};
    for (const auto i : kArr) {
        std::cout << i << ' ';
    }
    std::cout << '\n';
}
```



```
int main() {
    static constexpr std::string_view kStr{"Hello"};
    std::cout << kStr << '\n';
}
```


-Wunsafe-buffer-usage



```
void Print(const int* arr, size_t len) {
    for (size_t i = 0; i <= len; ++i) {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}

int main() {
    const int arr[]{1, 2, 3, 4, 5};
    Print(arr, std::size(arr));
}
```

```
error: unsafe buffer access [-Werror,-Wunsafe-buffer-usage]
    std::cout << arr[i] << ' ';
                  ^~~
```



```
void Print(std::span<const int> sp) {
    for (const auto elem : sp) {
        std::cout << elem << ' ';
    }
    std::cout << '\n';
}

int main() {
    const int arr[]{1, 2, 3, 4, 5};
    Print(arr);
}
```

std::span can be used instead of {pointer, size}

With `_LIBCPP_HARDENING_MODE=_LIBCPP_HARDENING_MODE_EXTENSIVE`, out of bound indexing attempts for `std::span` will cause a non-exploitable crash.

-Wunsafe-buffer-usage

All buffers need to be encapsulated into safe containers and view types:

- Containers: `std::array`, `std::vector`, `std::string`.
- Views: `std::span`, `std::string_view`

```
const int arr[] = {1, 2, 3};

int main() {
    const int local_arr[] = {1, 2, 3};
    // Use `arr` and `local_arr`.
    return std::size(arr) + std::size(local_arr);
}
```



```
constexpr auto kArr = std::to_array({1, 2, 3});

int main() {
    static constexpr auto kLocalArr = std::to_array({1, 2, 3});
    // Use `kArr` and `kLocalArr`.
    return std::size(kArr) + std::size(kLocalArr);
}
```

```
const char str[] = "Global string";

int main() {
    const char local_str[] = "Local string";
    // Use `str` and `local_str`.
    return std::size(str) + std::size(local_str);
}
```



```
constexpr std::string_view kStr{"Global string"};

int main() {
    static constexpr std::string_view kLocalStr{"Local string"};
    // Use `kStr` and `kLocalStr`.
    return std::size(kStr) + std::size(kLocalStr);
}
```

```
template <typename T>
void Foo(const T* arr, size_t len);
```



```
template <typename T>
void Foo(std::span<const T> arr);
```


```
void Foo(const char* str, size_t len);
```



```
void Foo(std::string_view);
```


With `_LIBCPP_HARDENING_MODE=_LIBCPP_HARDENING_MODE_EXTENSIVE`, out of bound indexing attempts for the containers will cause a non-exploitable crash.

What if we cannot fix an “unsafe buffer” problem?



```
// Callback from a C-library.  
int CCallbackFunc(int* arr, int len) {  
    int result = 0;  
    for (int i = 0; i < len; ++i) {  
        result += arr[i];  
    }  
    return result;  
}
```

```
error: unsafe buffer access [-Werror,-Wunsafe-buffer-usage]  
    result += arr[i];  
              ^~~
```




```
int CCallbackFunc(int* arr, int len) {  
    int result = 0;  
    for (int i = 0; i < len; ++i) {  
#pragma clang unsafe_buffer_usage begin  
        result += arr[i];  
#pragma clang unsafe_buffer_usage end  
    }  
    return result;  
}
```

```
#pragma clang unsafe_buffer_usage begin  
// Unsafe code can be wrapped around with  
// the pragma.  
#pragma clang unsafe_buffer_usage end
```

std::span constructor example and how to handle it

```
void Print(std::span<const int> sp) {
    for (const auto elem : sp) {
        std::cout << elem << ' ';
    }
    std::cout << '\n';
}
```




```
// Callback from a C-library.
void UnsafeCallback(const int* arr, size_t n) {
    Print(std::span{arr, n});
}
```

error: the two-parameter std::span construction is unsafe as it can introduce mismatch between buffer size and the bound information [-Werror,-Wunsafe-buffer-usage-in-container]

```
Print(std::span{arr, n});
      ^
```

```
template <typename T>
auto MakeSpan(const T* arr, size_t n) {
#pragma clang unsafe_buffer_usage begin
    return std::span{arr, n};
#pragma clang unsafe_buffer_usage end
}
```



```
// Callback from a C-library.
void UnsafeCallback(const int* arr, size_t n) {
    Print(MakeSpan(arr, n));
}
```

Ensuring that functions using unsafe buffers are flagged

```
// Callback from a C-library.
void UnsafeCallback(const int* arr, size_t n) {
    Print(MakeSpan(arr, n));
}
```

```
int main() {
    const int arr[]{1, 2, 3, 4, 5};
    UnsafeCallback(arr, std::size(arr) + 2);
}
```

1 2 3 4 5 1336145702 1336145702

It will be good to flag all the calls to *UnsafeCallback* to ensure extra safety is taken while calling them.

```
[[clang::unsafe_buffer_usage]] void UnsafeCallback(
    const int* arr, size_t n) {
    Print(MakeSpan(arr, n));
}
```

[[clang::unsafe_buffer_usage]] can be used to "tag" an unsafe function.

```
int main() {
    const int arr[]{1, 2, 3, 4, 5};
    UnsafeCallback(arr, std::size(arr) + 2);
}
```

error: function introduces unsafe buffer manipulation [-Werror, -Wunsafe-buffer-usage]
UnsafeCallback(arr, std::size(arr) + 2);
^~~~~~

```
int main() {
    const int arr[]{1, 2, 3, 4, 5};
    #pragma clang unsafe_buffer_usage begin
        UnsafeCallback(arr, std::size(arr));
    #pragma clang unsafe_buffer_usage end
}
```

The additional warning increases the chances of “detecting” such issues in the coding and code review process.

-Wunsafe-buffer-usage: C-Style functions

Unsafe buffer usage is also flagged for C-style functions like the following (*non-exhaustive list*):

- memcpy
- memset
- memmove
- strcpy
- strlen
- fprintf
- printf
- sprintf
- snprintf
- vsnprintf
- sscanf


The checker is smart enough to ***not always*** flag:

```
[[maybe_unused]] const auto l = strlen("hello"); // no warn  
printf("%s%d", "hello", *p); // no warn
```




Some example tests can be found in LLVM [codebase](#).

-Wunsafe-buffer-usage: memcpy



```
int main() {
    const char str[] = "Hello!!";
    char buffer[2 * std::size(str)]{};
    memcpy(buffer, str, std::size(str));
    std::cout << buffer << '\n';
}
```

*error: function 'memcpy' is unsafe [-Werror,-Wunsafe-buffer-usage-in-libc-call]
memcpy(buffer, str, std::size(str));
^~~~~~*



```
int main() {
    const char str[] = "Hello!!";
    char buffer[2 * std::size(str)]{};
    std::span buffer_as_span(buffer);
    std::span str_as_span(str);
    std::ranges::copy(str_as_span, buffer_as_span.begin());
    std::cout << buffer;
}
```

```
main:
    push    rbx
    sub     rsp, 16
    xorps   xmm0, xmm0
    movaps  xmmword ptr [rsp], xmm0
    movabs  rax, 9325436675581256
    mov     qword ptr [rsp], rax
    // More code for std::cout
```

```
.L.str:
    .asciz  "Hello!!"
```

9325436675581256 (decimal) = 0x21216F6C6C6548 (hexadecimal)

H -> 0x48
e -> 0x65
l -> 0x6C
l -> 0x6C
o -> 0x6F
! -> 0x21
! -> 0x21

-Wunsafe-buffer-usage: memcpy

```
int main() {
    const char str[] = "Hello!!";
    char buffer[2 * std::size(str)]{};
    std::span buffer_as_span(buffer);
    std::span str_as_span(str);
    std::ranges::copy(str_as_span, buffer_as_span.begin());
    std::cout << buffer;
}
```



This has no checks for span size!

Since the compiler can see through it, it generates the same code:

```
main:
    push    rbx
    sub     rsp, 16
    xorps   xmm0, xmm0
    movaps  xmmword ptr [rsp], xmm0
    movabs  rax, 9325436675581256
    mov     qword ptr [rsp], rax
    // More code for std::cout
```

```
template <typename T, std::size_t DestExtent,
          std::size_t SrcExtent>
void MemCpySpan(std::span<T, DestExtent> destination,
                std::span<const T, SrcExtent> source) {
    ASSERT(destination.size() >= source.size());
    std::ranges::copy(source, destination.begin());
}

int main() {
    const char str[] = "Hello!!";
    char buffer[2 * std::size(str)]{};
    MemCpySpan(std::span{buffer}, std::span{str});
    std::cout << buffer;
}
```

Example with just clang + x86:

```
#define ASSERT(condition) \
do { \
    if (!(condition)) [[unlikely]] { \
        asm volatile("ud2"); \
    } \
} while (0)
```

-Wunsafe-buffer-usage: memcpy

```
template <typename T, std::size_t DestExtent, std::size_t SrcExtent>
void MemCpySpan(std::span<T, DestExtent> destination,
               std::span<const T, SrcExtent> source) {
    ASSERT(destination.size() >= source.size());
    std::ranges::copy(source, destination.begin());
}
```

```
void CopyVector(std::vector<int>& dest, const std::vector<int>& src) {
    MemCpySpan(std::span{dest}, std::span{src});
}
```

```
CopyVector(std::__1::vector<int, std::__1::allocator<int>>&, std::__1::vector<int, std::__1::allocator<int>> const&):
```

```
    mov     rax, rsi
    mov     rcx, rdi
    mov     rdi, qword ptr [rdi]
    mov     rcx, qword ptr [rcx + 8]
    sub     rcx, rdi
    mov     rsi, qword ptr [rsi]
    mov     rax, qword ptr [rax + 8]
    mov     rdx, rax
    sub     rdx, rsi
    cmp     rcx, rdx
    jb      .LBB0_3
    cmp     rax, rsi
    jne     memmove@PLT
    ret
```

```
.LBB0_3:
```

```
    ud2
    jmp     memmove@PLT
```


-Unsafe-buffer-usage: memcpy

```
int main() {
    const char str[] = "Hello!!";
    char buffer[2 * std::size(str)]{};
    std::span buffer_as_span(buffer);
    std::span str_as_span(str);
    std::ranges::copy(str_as_span,
                      buffer_as_span.begin());
    std::cout << buffer;
}
```

In Chromium code **base::span** is an alternate for **std::span** which is **hardened** and has extra functions to handle **memcpy**.



```
int main() {
    const char str[] = "Hello!!";
    char buffer[2 * std::size(str)]{};
    base::span{buffer}.first<std::size(str)>().copy_from(base::span{str});
    std::cout << buffer << '\n';
}
```

```
template <typename T, std::size_t DestExtent, std::size_t SrcExtent>
void MemCpySpan(base::span<T, DestExtent> destination,
               base::span<const T, SrcExtent> source) {
    destination.first(source.size()).copy_from(source);
}
```

base::span::copy_from will trap at runtime for source / destination span size mismatch.

```
int main() {
    const char str[] = "Hello!!";
    char buffer[2 * std::size(str)]{};
    MemCpySpan(base::span{buffer}, base::span{str});
    std::cout << buffer << '\n';
}
```

-Wunsafe-buffer-usage: memset

```
int arr[4]{};
// Attempt to set each element in array 1.
memset(arr, 1, sizeof(arr));
```



error: function 'memset' is unsafe [-Werror, -Wunsafe-buffer-usage-in-libc-call]
 memset(arr, 1, sizeof(arr));
 ^~~~~~

```
void Print(std::span<int, 4> sp) {
    std::println("{} {} {} {}", sp[0], sp[1], sp[2], sp[3]);
}
```

```
Print(arr);
```

```
16843009 16843009 16843009 16843009
```

```
int main() {
    int arr[4]{};
    // Attempt to set each element in array 1.
    std::ranges::fill(arr, 1);
    Print(arr);
}
```



```
1 1 1 1
```

-Wunsafe-buffer-usage: statistics

Chromium has reduced unsafe buffer usage by over 90% in their codebase.

Edge-only code:

- Unsafe buffer usage (non-C library functions): Got down to 0 in 3 months.
- Unsafe C library function usage: > 75% fixed in 2 months.

Bounds safety: Edge code state

- With libc++ hardening adoption and `-Wunsafe-buffer-usage` implementation, we believe are in a good state.
- It also helps that it is part of the toolchain, so makes it harder to “miss” issues.
- We also used ASAN in CI pipelines to catch more such issues from tests.

Quick note on ASAN

ASAN: Address Sanitizer

- It is a fast memory error detector which can detect:
 - Out-of-bounds accesses to heap, stack and globals.
 - Use-after-free.
 - Double-free, invalid free.

Lifetime Safety

Use raw pointers carefully

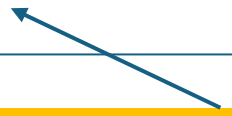
```
// another_class.h
struct AnotherClass {
    // Other stuff.
    int Foo();
};
```

```
// some_class.h
class SomeClass {
public:
    SomeClass(AnotherClass* a) : a_(a) {}
    void DoSomething();

    // Non copyable and movable.
    SomeClass& operator=(SomeClass&&) = delete;

private:
    AnotherClass* a_ = nullptr;
};
```

```
// some_class.cc
void SomeClass::DoSomething() {
    a_->Foo();
}
```



Without careful lifetime considerations this call can run into Use-After-free.

Try to see whether it can be converted to `unique_ptr`, so that the ownership model is clear.

```
class SomeClass {
public:
    SomeClass(std::unique_ptr<AnotherClass> a) : a_(std::move(a)) {}
    void DoSomething();
    // Non copyable and movable.
    SomeClass& operator=(SomeClass&&) = delete;

private:
    std::unique_ptr<AnotherClass> a_;
};
```

`unique_ptr` solves this issue.

However, it may not be possible in some cases to convert to `unique_ptr`.

Chromium raw_ptr (MiraclePtr)

```
// some_class.h
class SomeClass {
public:
    SomeClass(AnotherClass* a) : a_(a) {}
    void DoSomething();
private:
    AnotherClass* a_ = nullptr;
};
```

```
// some_class.cc
void SomeClass::DoSomething() {
    a_->Foo();
}
```

In cases where we “cannot” move to unique_ptr, use-after-free can lead to exploits.

Chromium has [raw_ptr](#) (also known as MiraclePtr) which can be used to significantly reduce the chances of exploit.

```
// some_class.h
class SomeClass {
public:
    SomeClass(AnotherClass* a) : a_(a) {}
    void DoSomething();
private:
    raw_ptr<AnotherClass> a_ = nullptr;
};
```

```
// some_class.cc
void SomeClass::DoSomething() {
    a_->Foo();
}
```

Now, this Use-after-free has ***significantly lesser chances*** of being exploited *under certain conditions*.

Chromium raw_ptr (MiraclePtr)

```
// some_class.h
class SomeClass {
public:
    SomeClass(AnotherClass* a) : a_(a) {}
    void DoSomething();
private:
    base::raw_ptr<AnotherClass> a_ = nullptr;
};
```

```
// some_class.cc
void SomeClass::DoSomething() {
    a_->Foo();
}
```

Now, this Use-after-free has ***significantly lesser chances*** of being exploited *under certain conditions*.

raw_ptr depends on PartitionAlloc allocator in Chromium.

Any memory allocated and freed using PartitionAlloc is poisoned with 0xEF pattern.

*Note that the sheer act of dereferencing a dangling pointer won't crash, **but poisoning increases chances that a subsequent usage of read memory will crash** (particularly if the read poison is interpreted as a pointer and dereferenced thereafter), thus giving us a chance to investigate and fix. Having said that, we want to emphasize that dereferencing a **dangling pointer remains an Undefined Behavior**.*

Dependence on PartitionAlloc also means that it cannot help with literals, stack allocated memory, TLS, etc.

Chromium raw_ptr (MiraclePtr)

The Chromium C++ Style Guide asks to use `raw_ptr<T>` for class and struct fields in place of a raw C++ pointer `T*` *whenever possible*, except in Renderer-only code.

```
// some_class.h
class SomeClass {
public:
    SomeClass(AnotherClass& a) : a_(a) {}
    void DoSomething();
private:
    base::raw_ref<AnotherClass> a_;
};
```

There is a similar **raw_ref** class which can handle *dangling references* to pointers allocated using PartitionAlloc.

Chromium raw_ptr (MiraclePtr)

raw_ptr makes code less exploitable.

Does not make code correct.

Correct ownership model and deterministic coding approaches are preferred.

Use raw pointers carefully

```
// some_class.h
class SomeClass {
public:
    SomeClass(AnotherClass* a) : a_(a) {}
    void DoSomething();

    // Non copyable and movable.
    SomeClass& operator=(SomeClass&&) = delete;

private:
    AnotherClass* a_ = nullptr;
};
```

```
// some_class.cc
void SomeClass::DoSomething() {
    a_->Foo();
}
```



If **unique_ptr** is not possible, consider using **weak_ptr**.

```
// some_class.h
class SomeClass {
public:
    SomeClass(std::shared_ptr<AnotherClass> a) : a_(a) {}
    void DoSomething();

    // Non copyable and movable.
    SomeClass& operator=(SomeClass&&) = delete;

private:
    std::weak_ptr<AnotherClass> a_;
};
```

```
// some_class.cc
void SomeClass::DoSomething() {
    if (auto p = a_.lock()) {
        p->Foo();
    }
}
```

Use raw pointers carefully

```
// some_class.h
class SomeClass {
public:
    SomeClass(AnotherClass* a) : a_(a) {}
    void DoSomething();
private:
    AnotherClass* a_ = nullptr;
};
```

```
// some_class.cc
void SomeClass::DoSomething() {
    a_->Foo();
}
```

If shared_ptr / weak_ptr is not possible, consider **storing an id** and **querying later** from **another object with guaranteed longer lifetime**.

```
class AnotherClassManager {
public:
    AnotherClass* Create();
    AnotherClass* GetWithId(int i) const;

    static AnotherClassManager& GetInstance();
    // Other implementation stuff.
};
```

```
// another_class.h
struct AnotherClass {
    AnotherClass(int id);
    // Other stuff.
    int Foo();
    int GetId() const;
    // Other stuff.
};
```

```
// some_class.h
class SomeClass {
public:
    SomeClass(AnotherClass* a)
        : id_(a->GetId()) {}

    void DoSomething();
private:
    int id_ = 0;
};
```

```
// some_class.cc
void SomeClass::DoSomething() {
    auto* p =
        AnotherClassManager::GetInstance().GetWithId(id_);
    if (p) {
        p->Foo();
    }
}
```

Using shared_ptr carefully

```
// Assume this class is used as shared_ptr.
struct MyClass {
    MyClass() { std::cout << "MyClass()\n"; }
    ~MyClass() { std::cout << "~MyClass()\n"; }

    void Foo(std::function<void()> fn) {
        std::cout << "MyClass::Foo before fn\n";
        fn();
        std::cout << "MyClass::Foo after fn\n";
    }
};
```



```
struct MyClass : public std::enable_shared_from_this<MyClass> {
    MyClass() { std::cout << "MyClass()\n"; }
    ~MyClass() { std::cout << "~MyClass()\n"; }

    void Foo(std::function<void()> fn) {
        std::cout << "MyClass::Foo before fn\n";
        auto strong_this = shared_from_this();
        fn();
        std::cout << "MyClass::Foo after fn\n";
    }
};
```

```
MyClass()
MyClass::Foo before fn
~MyClass()
MyClass::Foo after fn
```

```
MyClass()
MyClass::Foo before fn
MyClass::Foo after fn
~MyClass()
```

We may need to “increase” the reference count of a class before calling functions which may delete that object from which it got called.

Async call of member function

```
using FuncType = std::function<void(const std::string&)>;
```

```
void ReadFile(const std::string& name,
             FuncType callback);
```

```
struct MyClass {
    MyClass() { std::cout << "MyClass()\n"; }
    ~MyClass() { std::cout << "~MyClass()\n"; }

    void StartReadingFile(const std::string& name) {
        ReadFile(name, [this](const std::string& content) {
            Callback(content);
        });
    }
private:
    void Callback(const std::string& content) {
        std::cout << "MyClass::Callback: content: "
                  << content << '\n';
    }
};
```

Be careful with callbacks being registered for class member functions to be called later.

Consider handling possible use-after-free scenarios using weak_ptr or un-registration approaches.

```
MyClass()
~MyClass()
MyClass::Callback: content: Hello world!!
```

The callback can be called after the object has been destroyed.

```
struct MyClass :
    public std::enable_shared_from_this<MyClass> {
    MyClass() { std::cout << "MyClass()\n"; }
    ~MyClass() { std::cout << "~MyClass()\n"; }

    void StartReadingFile(const std::string& name) {
        ReadFile(name, [wp = std::weak_ptr<MyClass>{
            shared_from_this()}](
            const std::string& content) {
            if (auto p = wp.lock()) {
                p->Callback(content);
            }
        });
    }
private:
    void Callback(const std::string& content) {
        std::cout << "MyClass::Callback: content: "
                  << content << '\n';
    }
};
```

```
MyClass()
~MyClass()
```

Chromium codebase rules

- `std::shared_ptr`, `std::function` are banned.
- Use replacements
 - `base::RefCounted`, `base::RefCountedThreadSafe`.
 - `base::OnceCallback`, `base::RepeatingCallback`.

Async call of member function: Chromium

```
using FuncType = base::OnceCallback<void(const std::string&)>;

void ReadFile(const std::string& name, FuncType callback);
```

```
struct MyClass {
    MyClass() { std::cout << "MyClass()\n"; }
    ~MyClass() { std::cout << "~MyClass()\n"; }

    void StartReadingFile(const std::string& name) {
        ReadFile(name,
            base::BindOnce(&MyClass::Callback, this));
    }

private:
    void Callback(const std::string& content) {
        std::cout << "MyClass::Callback: content: "
            << content << '\n';
    }
};
```



This code “does not” compile.



```
struct MyClass {
    MyClass() { std::cout << "MyClass()\n"; }
    ~MyClass() { std::cout << "~MyClass()\n"; }

    void StartReadingFile(const std::string& name) {
        ReadFile(name, base::BindOnce(&MyClass::Callback,
            base::Unretained(this)));
    }

private:
    void Callback(const std::string& content) {
        std::cout << "MyClass::Callback: content: "
            << content << '\n';
    }
};
```

The developer needs to explicitly use **base::Unretained** to get it to compile.

base::Unretained causes **BindOnce** to store the pointer internally as **raw_ptr**, so the code becomes “safer” from **use-after-free** exploits.

Async call of member function: Chromium

```
using FuncType = base::OnceCallback<void(const std::string&);>;
void ReadFile(const std::string& name, FuncType callback);
```

base::Unretained is frowned upon during use.

```
struct MyClass {
    MyClass() { std::cout << "MyClass()\n"; }
    ~MyClass() { std::cout << "~MyClass()\n"; }

    void StartReadingFile(const std::string& name) {
        ReadFile(name,
            base::BindOnce(&MyClass::Callback,
                weak_factory_.GetWeakPtr()));
    }


private:
    void Callback(const std::string& content) {
        std::cout << "MyClass::Callback: content: "
            << content << '\n';
    }

    base::WeakPtrFactory<MyClass> weak_factory_{this};
};
```

Code to capture “this” pointer also does not compile.

Chromium also supports a weak-ptr infrastructure which also handles this scenario.

It is also possible to solve this with base::RefCounted.



```
struct MyClass {
    MyClass() { std::cout << "MyClass()\n"; }
    ~MyClass() { std::cout << "~MyClass()\n"; }

    void StartReadingFile(const std::string& name) {
        ReadFile(name, base::BindOnce([this](
            const std::string& context) {
                Callback(context);
            })));
    }
}
```

Use attributes like `[[clang::lifetimebound]]`

```
std::string_view GetOrDefault(std::string_view str,
                             std::string_view key) {
    if (str.starts_with(key)) {
        return str.substr(key.size());
    }
    return std::string_view{};
}
```

```
const auto v1 = GetOrDefault("hello|world", "hello|");
std::cout << v1 << '\n';
```

world

```
std::string_view GetOrDefault(std::string_view str [[clang::lifetimebound]],
                             std::string_view key) {
    if (str.starts_with(key)) {
        return str.substr(key.size());
    }
    return std::string_view{};
}
```

```
const auto v2 = GetOrDefault(GetStr(), "hello|");
std::cout << v2 << '\n';
```

```
std::string GetStr() {
    return "hello|world|and a longer string";
}
```

```
const auto v2 = GetOrDefault(GetStr(), "hello|");
std::cout << v2 << '\n';
```

c??Q29??a longer string

string returned by **GetStr()** is destroyed here.

v2 here refers to destroyed memory.

error: object backing the pointer will be destroyed at the end of the full-expression [-Werror,-Wdangling-gsl]
const auto v2 = GetOrDefault(GetStr(), "hello|");
~~~~~

Use attributes like `[[clang::lifetimebound]]`

```
class MyClass {
public:
    MyClass(std::string str) : str_(std::move(str)) {}
    const std::string& str() const { return str_; }

private:
    std::string str_;
};
```

```
int main() {
    MyClass m("hello");
    std::cout << m.str() << '\n';
    const auto& s = MyClass{"world"}.str();
    std::cout << s << '\n';
}
```

hello
world

GCC does produce error with `-Werror=dangling-pointer`

"s" here refers to destroyed memory.

string returned by `str()` is destroyed here.

```
class MyClass {
public:
    MyClass(std::string str) : str_(std::move(str)) {}
    const std::string& str() const [[clang::lifetimebound]] {
        return str_;
    }
private:
    std::string str_;
};
```

error: temporary bound to local reference 's' will be destroyed at the end of the full-expression [-Werror,-Wdangling]


```
const auto& s = MyClass{"world"}.str();
               ^~~~~~
```

An alternate way to fix this is to use overloading.

```
class MyClass {
public:
    MyClass(std::string str) : str_(std::move(str)) {}
    const std::string& str() const& { return str_; }
    std::string str() && { return std::move(str_); }

private:
    std::string str_;
};
```

Clang -Wdangling: Enabled by default



```
int main() {  
    const char* pc = std::string{"hello"}.c_str();  
    std::cout << pc << '\n';  
}
```

error: object backing the pointer will be destroyed at the end of the full-expression [-Werror,-Wdangling-gsl]

```
    const char* pc = std::string{"hello"}.c_str();  
    ^~~~~~
```

Currently works for **only** for some STL types.

Lifetime safety: Edge status

- `raw_ptr` – helps with safety
 - Code review phase.
 - CI pipeline.
- `base::OnceCallback`, `base::Unretained`, `base::WeakPtr`:
 - Partially enforced with tooling.
 - Code review phase.
- `[[clang::lifetimebound]]` – Only code review.
- Coding strategy – Code review / implementation review.
- `-Wdangling` – enabled by default.
- Clang-tidy check.
- ASAN / MSAN pipelines help figure out more issue from tests in CI pipeline.
- GWPAsan – Helps after release.

Quick note on MSAN

MSAN: Memory Sanitizer

- It is an uninitialized memory use detector which can detect:
 - Uninitialized value was used in a conditional branch.
 - Uninitialized pointer was used for memory accesses.
 - Uninitialized value was passed or returned from a function call.
 - Use-after-destruction.

Initialization Safety

Class: Initialize members at point of declaration

```
struct Size {  
    Size() {  
        width = 0;  
        height = 0;  
    }  
  
    int width;  
    int height;  
};
```



```
struct Size {  
    int width = 0;  
    int height = 0;  
};
```


Initializing at the point of declaration, when possible, reduces the possibility of mistakes with missing out on initialization.

```
struct Size {  
    Size() {  
        width = 0;  
        // height = 0;  
    }  
  
    int width;  
    int height;  
};
```

Clang-tidy check [cppcoreguidelines-pro-type-member-init](#) can be used to catch scenarios.


```
warning: constructor does not initialize these fields: height [cppcoreguidelines-pro-type-member-init]  
    Size() {  
    ^  
        width = 0;  
        // height = 0;  
    }  
  
    int width;  
    int height;
```

Class: Initialize in the member initialization list.



```
class MyClass {
public:
    MyClass() {
        member_b_ = member_a_ + 1;
        member_a_ = 10;
    }

private:
    // Declared first, but initialized second.
    int member_a_ = 0;
    // Declared second, but initialized first.
    int member_b_ = 0;
};
```





```
class MyClass {
public:
    MyClass() : member_b_(member_a_ + 1), member_a_(10) {}

private:
    // Declared first, but initialized second in the list.
    int member_a_ = 0;
    // Declared second, but initialized first in the list.
    int member_b_ = 0;
};
```

```
error: field 'member_b_' will be initialized after field 'member_a_' [-Werror,-Wreorder-ctor]
    MyClass() : member_b_(member_a_ + 1), member_a_(10) {}
               ^~~~~~
               member_a_(10)
               member_b_(member_a_ + 1)
```

-Wreorder-ctor (-Wall)



```
class MyClass {
public:
    MyClass() : member_a_(10), member_b_(member_a_ + 1) {}

    // To remove `-Wunused-private-field` error.
    int member_b() const { return member_b_; }

private:
    // Declared first, but initialized second in the list.
    int member_a_ = 0;
    // Declared second, but initialized first in the list.
    int member_b_ = 0;
};
```


Initialize variables

```
int Foo() {
    return 21;
}
```

```
int main() {
    int a;
    a = Foo();
    return a;
}
```

clang-tidy: [cppcoreguidelines-init-variables](#):

```
warning: variable 'a' is not initialized [cppcoreguidelines-init-variables]
      int a;
      ^
```



```
int main() {
    const int a;
    a = Foo();
    return a;
}
```

Making variable “const” can help compiler detect such issues.

Clang-tidy also has other checks for uninitialized variables in various scenarios:

```
core-undefinedbinaryoperatorresult
core-uninitialized-arraysubscript
core-uninitialized-assign
core-uninitialized-branch
core-uninitialized-capturedblockvariable
core-uninitialized-newarraysize
core-uninitialized-undefreturn
```

[P2795](#) (Erroneous behaviour for uninitialized reads) from C++26 removes the undefined behavior with uninitialized variables.

Initialization safety: Edge status:

- Coding guidelines.
- Code review.
- Warning flags and clang-tidy checks.
- MSAN pipelines help figure out more issue from tests in CI pipeline.

Reducing Type confusion

Here's an attempt to *type pun* `int` to `float`.

This is correct in C, but not in C++.



Use `std::bit_cast` instead of `reinterpret_cast` when applicable.

Avoid reinterpret_cast when possible

```
std::optional<float> GetFloat(char* p, size_t n) {
    if (n < sizeof(float)) {
        return std::nullopt;
    }
    return *reinterpret_cast<float*>(p);
}
```



```
std::optional<float> GetFloatCorrect(char* p, size_t n) {
    if (n < sizeof(float)) {
        return std::nullopt;
    }
    float f;
    memcpy(&f, p, sizeof(float));
    return f;
}
```

GetFloat function can run into **SIGBUS** on some platforms:

- Some platforms (e.g. ARM) have very strict memory alignment requirements
- On Intel, these alignment constraints exist for SIMD code

GCC's **-Wcast-align=strict** flags this issue.

memcpy solves this problem.

But it runs into unsafe buffer usage errors. So, consider using some memcpy wrappers to reduce direct memcpy usage in multiple places in your codebase. Chromium has such wrappers.

clang-tidy checks: [cppcoreguidelines-pro-type-reinterpret-cast](#), [cppcoreguidelines-pro-type-static-cast-downcast](#) flag “all” reinterpret_cast / static_cast uses including valid ones.

These flag “all” reinterpret_cast / static_cast uses **including valid ones**.

Clang built with **-fsanitize=address** generates runtime errors for such unaligned access.

Convert union to variant

```

struct Wrapper {
    bool is_string_;
    union U {
        std::string s;
        std::vector<int> v;
    };
    U() {}
    ~U() {} // destruction managed manually
};

void SetString(const std::string& str) {
    new (&u.s) std::string(str);
    is_string_ = true;
}

void SetVector(std::vector<int> vv) {
    new (&u.v) std::vector<int>(std::move(vv));
    is_string_ = false;
}

~Wrapper() {
    if (is_string_) {
        u.s.~basic_string();
    } else {
        u.v.~vector();
    }
}
};

```

Wrapper w;

Can crash at runtime.

Constructor does not initialize correctly.

Doesn't call previous type destructors.

Destructor is now working incorrectly.

```

struct Wrapper {
    std::variant<std::monostate,
                std::string, std::vector<int>> u;

    void SetString(const std::string& str) { u = str; }
    void SetVector(std::vector<int> vv) { u = std::move(vv); }
};

```


Control flow integrity

Control flow integrity checks help find invalid cast issues.

```
class Base {  
public:  
    virtual ~Base() = default;  
    virtual void Foo() { std::cout << "Base::Foo()\n"; }  
};
```

```
class Derived : public Base {  
public:  
    void Foo() override { std::cout << "Derived::Foo()\n"; }  
};
```

```
int main() {  
    Base* b = new Base();  
    // Invalid cast.  
    Derived* d = static_cast<Derived*>(b);  
  
    d->Foo();  
    delete b;  
}
```

*runtime error: control flow integrity check for
type 'Derived' failed during base-to-derived
cast (vtable address 0x00...0)
0x00..0: note: vtable is of type 'Base'*

Reducing type confusion: Edge status

- Coding guidelines.
- Code review.
- Control flow integrity checks in CI pipeline.

Thread Safety

Use Sequences

In our code base we prefer to use “sequences” to provide thread safety without needing any actual locking.

Sequences are Chromium abstractions on top of system threads.

Chromium has its own internal libraries for supporting:

- Different sequences.
- Creating your own sequence.
- Posting to sequence.
- Posting to a single thread, UI thread, IO thread.

The library takes care of synchronization when tasks is being posted to a sequence.

Tasks posted to the same sequence will run in sequential order.

Each task can run in a different physical thread, but the library ensures that a task on start can see the side effects of the previous task from the same sequence.

A very useful utility is to post a task to “any” sequence and get back the result in the same thread from which it was posted.

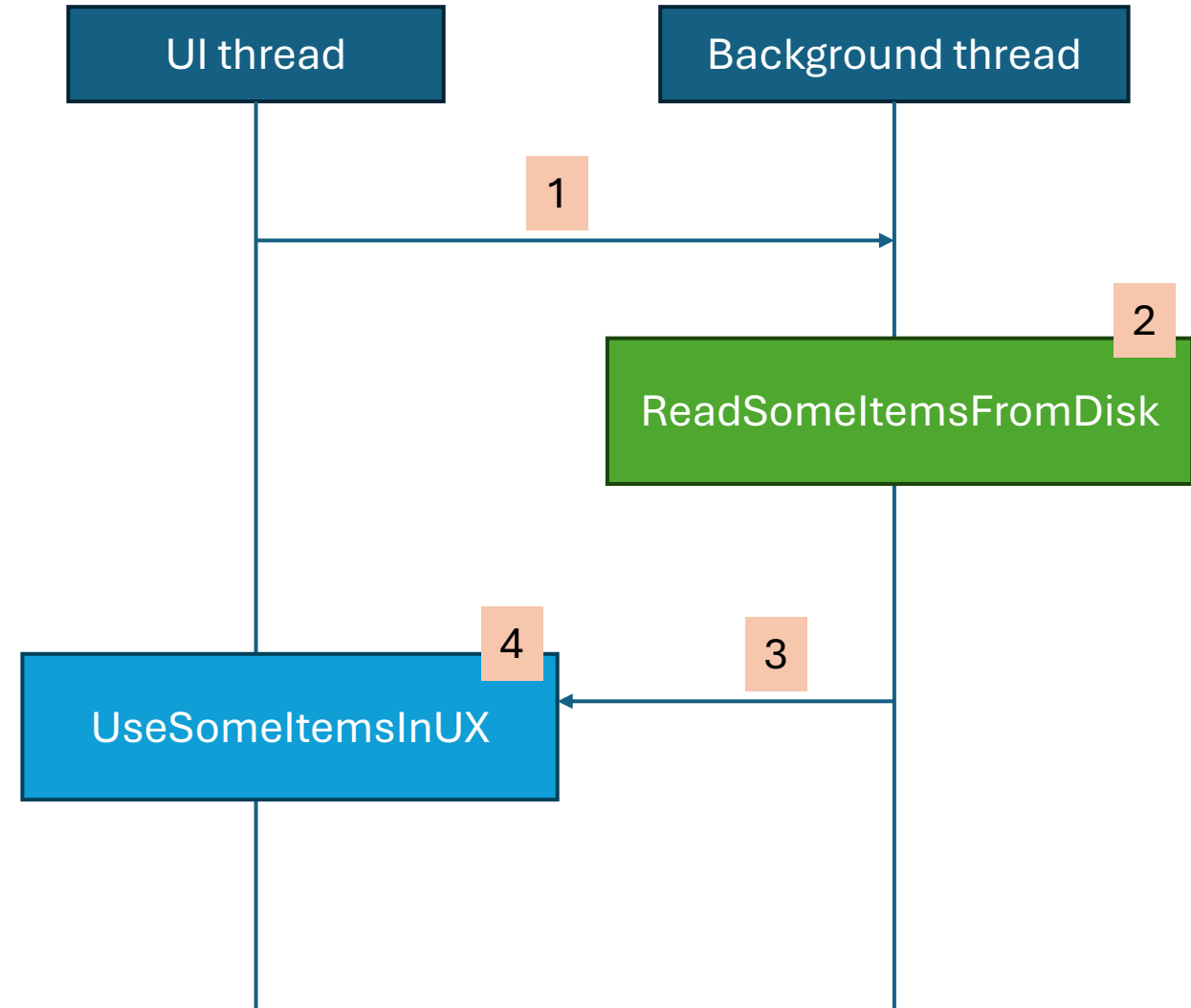
Use Sequences

A very useful utility is to post a task to “any” sequence and get back the result in the same thread from which it was posted.

```
std::vector<std::string> ReadSomeItemsFromDisk(
    const std::string& keyword) {
    if (keyword.empty()) {
        return {};
    }
    // Use keyword to read items from disk.
    return {"item1", "item2", "item3"};
}
```

```
void UseSomeItemsInUX(
    const std::vector<std::string>& item) {
    for (const auto& s : item) {
        // Use string in UX.
    }
}
```

```
base::ThreadPool::PostTaskAndReplyWithResult(
    FROM_HERE,
    {base::MayBlock()},
    base::BindOnce(&ReadSomeItemsFromDisk, "keyword"),
    base::BindOnce(&UseSomeItemsInUX));
```



Improving thread safety: Edge status:

- Coding guidelines.
- Code review.
- Thread sanitizer (TSan) runs in CI pipeline.
 - It is a data race detector.

Definition Safety

Handling ODR violations:

```
// header.h  
constexpr double kPi = 3.141592653589793;
```

Just “constexpr” variable in header can cause ODR violations.

```
// header.h  
inline constexpr double kPi = 3.141592653589793;
```

Use inline keyword to header-only definitions to ensure that one definition rule is not violated.

ODR violation: Ensure preprocessor flags are used consistently.

```
// my_class.h
class MyClass {
public:
    MyClass(std::string s);

    const std::string& str() const { return str_; }
#ifdef SPECIAL_SAUCE
    void SetSpecial(uint64_t s);
#endif // SPECIAL_SAUCE

private:
#ifdef SPECIAL_SAUCE
    uint64_t special_number_ = 0;
#endif // SPECIAL_SAUCE
    std::string str_;
};
```

We found out such a scenario in our code using Address Sanitizer (ASAN) runs for tests.

```
// my_class.cc
#define SPECIAL_SAUCE

#include "my_class.h"

MyClass::MyClass(std::string s) : str_(std::move(s)) {
    SetSpecial(std::hash<std::string>{}(str_));
}

void MyClass::SetSpecial(size_t s) {
    special_number_ = s;
}
```

```
// main.cc
#include <iostream>

#include "my_class.h"

int main() {
    MyClass m("hello world hi there!");
    std::cout << m.str() << '\n';
}
```

Program terminated with signal: SIGSEGV

ODR violation: Ensure preprocessor flags are used consistently.

```
// my_class.h
class MyClass {
public:
    MyClass(std::string s);

    const std::string& str() const { return str_; }
#ifdef SPECIAL_SAUCE
    void SetSpecial(uint64_t s);
#endif // SPECIAL_SAUCE

private:
#ifdef SPECIAL_SAUCE
    uint64_t special_number_ = 0;
#endif // SPECIAL_SAUCE
    std::string str_;
};
```

```
// main.cc
int main() {
    MyClass m("hello world hi there!");
    std::cout << m.str() << '\n';
}
```

```
// my_class.cc
MyClass::MyClass(std::string s) : str_(std::move(s)) {
    SetSpecial(std::hash<std::string>{}(str_));
}
```

main.cc: MyClass

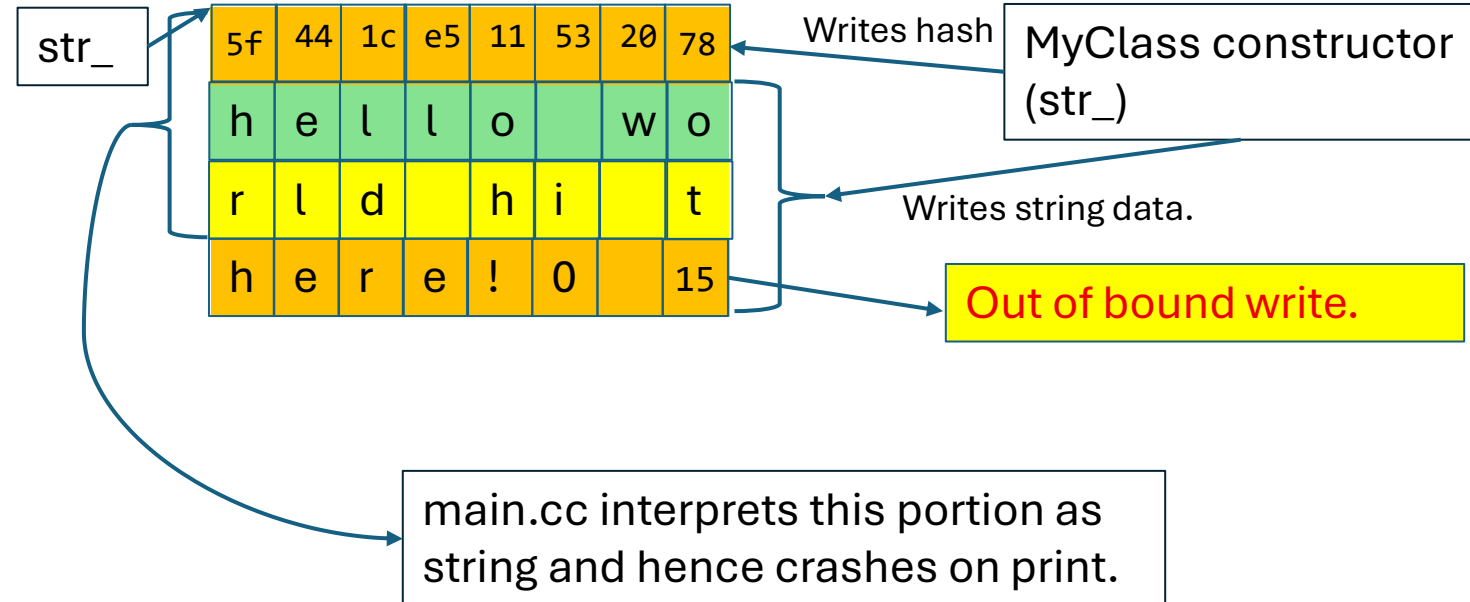
std::string str

my_class.cc: MyClass

uint64_t special_number_

std::string str

Considering std::string is 24 bytes.



Handling ODR violations:

```
// source.cc  
// Only used in source.cc  
class SomeLocalClass {  
    // Stuff.  
};  
  
int some_value = 0;  
  
void Foo() {  
    // Do something.  
}
```



```
// source.cc  
namespace {  
  
class SomeLocalClass {  
    // Stuff.  
};  
  
int some_value = 0;  
  
void Foo() {  
    // Do something.  
}  
  
} // namespace
```


Add source-only classes / variables / functions to unnamed namespace to get internal linkage.

Improving Definition safety: Edge status:

- Coding guidelines.
- Code review.
- ASAN runs in CI pipeline.

Reduce Undefined Behavior

Use constexpr, constinit, const variables




```
#include <limits.h>
#include <iostream>

void SignedIntegerOverflow() {
    int a = INT_MAX;
    int b = a + 1; // Undefined behavior: signed integer overflow
    std::cout << "SignedIntegerOverflow: a = " << a << ", b = " << b << '\n';
}

int main() {
    SignedIntegerOverflow();
}
```

SignedIntegerOverflow: a = 2147483647, b = -2147483648




```
void SignedIntegerOverflow() {
    constexpr int a = INT_MAX;
    constexpr int b = a + 1; // Undefined behavior: signed integer overflow
    std::cout << "SignedIntegerOverflow: a = " << a << ", b = " << b << '\n';
}
```

error: constexpr variable 'b' must be initialized by a constant expression

constexpr int b = a + 1; // Undefined behavior: signed integer overflow

note: value 2147483648 is outside the range of representable values of type 'int'

constexpr int b = a + 1; // Undefined behavior: signed integer overflow



```
void SignedIntegerOverflow() {
    const int a = INT_MAX;
    const int b = a + 1; // Undefined behavior: signed integer overflow
    std::cout << "SignedIntegerOverflow: a = " << a << ", b = " << b << '\n';
}
```


error: overflow in expression; result is -2'147'483'648 with type 'int' [-Werror, -Winteger-overflow]

const int b = a + 1; // Undefined behavior: signed integer overflow

-Winteger-overflow (-Wall)

Consider making variables which can be calculated at compile time **constexpr**. That will help catch undefined behavior.

Use constexpr, constinit, const variables




```
void DivisionByZero() {
    int a = 10;
    int b = 0;
    const int c = a / b;
    std::cout << "DivisionByZero: c = " << c << '\n';
}

void ModuloByZero() {
    int a = 10;
    int b = 0;
    const int c = a % b;
    std::cout << "ModuloByZero: c = " << c << '\n';
}

void ShiftOperations() {
    int a = 1;
    int b = -1;
    const int c = a << b;
    std::cout << "ShiftOperations: c = " << c << '\n';
}
```

These warnings are enabled by default.



```
void DivisionByZero() {
    const int a = 10;
    const int b = 0;
    const int c = a / b;
    std::cout << "DivisionByZero: c = " << c << '\n';
}

void ModuloByZero() {
    const int a = 10;
    const int b = 0;
    const int c = a % b;
    std::cout << "ModuloByZero: c = " << c << '\n';
}

void ShiftOperations() {
    const int a = 1;
    const int b = -1;
    const int c = a << b;
    std::cout << "ShiftOperations: c = " << c << '\n';
}
```

error: division by zero is undefined [-Werror,-Wdivision-by-zero]

```
const int c = a / b;
             ^ ~
```

error: remainder by zero is undefined [-Werror,-Wdivision-by-zero]

```
const int c = a % b;
             ^ ~
```

error: shift count is negative [-Werror,-Wshift-count-negative]

```
const int c = a << b;
             ^ ~
```

Reduce undefined behavior

Consider using constexpr / consteval functions when applicable to help detect undefined behavior.

We saw an example before with type punning being caught.

```
union U {  
    int n;  
    float f;  
};
```

```
consteval float TestUnion1(int n) {  
    U u{.n = n};  
    return u.f;  
}
```

```
int main() {  
    constexpr float f = TestUnion1(10);  
    std::cout << f << '\n';  
}
```



Shafik Yaghmour's article: [Exploring Undefined Behavior Using Cpp20's constexpr](#) contains many more examples.

Undefined behavior can cause unwanted optimizations.

Consider this example from [2011 llvm blog](#):

```
#include <stdio.h>

static void (*FP)() = 0;
static void impl() {
    printf("hello\n");
}

void set() {
    FP = impl;
}

void call() {
    FP();
}

int main() {
    call();
}
```

When built is Clang with -O2

```
set():
    ret
call():
    lea    rdi, [rip + .Lstr]
    jmp    puts@PLT
main:
    push   rax
    lea    rdi, [rip + .Lstr]
    call   puts@PLT
    xor    eax, eax
    pop    rcx
    ret
.Lstr:
    .asciz "hello"
```

hello

It is allowed to do this because **calling a null pointer is undefined**, which permits it to assume that **set() must be called** before **call()**.

When built with **-fno-delete-null-pointer-checks**:

Program terminated with signal: SIGSEGV

```
set():
    lea    rax, [rip + impl()]
    mov    qword ptr [rip + FP], rax
    ret
impl():
    lea    rdi, [rip + .Lstr]
    jmp    puts@PLT
call():
    jmp    qword ptr [rip + FP]
main:
    push   rax
    call   qword ptr [rip + FP]
    xor    eax, eax
    pop    rcx
    ret
.Lstr:
    .asciz "hello"
```

Undefined behavior can cause optimizations.


From [documentation](#):

-fdelete-null-pointer-checks, -fno-delete-null-pointer-checks

*When enabled, **treat null pointer dereference, creation of a reference to null**, or passing a null pointer to a function parameter annotated with the **“nonnull” attribute as undefined behavior**. (And, thus the optimizer may assume that any pointer used in such a way must not have been null and **optimize away the branches accordingly**.) **On by default**.*

Our code base has **-fno-delete-null-pointer-checks** on which removes this undefined behavior.

Missing return in function



```
int Foo(int i) {  
    if (i == 0) {  
        return i + 1;  
    }  
    if (i % 2 == 0) {  
        return i + 2;  
    }  
    if (i % 3 == 0) {  
        return i + 3;  
    }  
}
```

```
error: non-void function does not return a value in all control paths [-Werror,-Wreturn-type]  
    }  
    ^
```

-Wreturn-type is a default warning in Clang.

From [stmt.return](#):

Flowing off the end of a constructor, a destructor, or a non-coroutine function with a cv void return type is equivalent to a return with no operand. Otherwise, flowing off the end of a function that is neither main ([\[basic.start.main\]](#)) nor a coroutine ([\[dcl.fct.def.coroutine\]](#)) results in undefined behavior.

Use sanitizers to detect undefined behavior at runtime

Clang's [Undefined Behavior Sanitizer](#) has many checks:

```
-fsanitize=alignment
-fsanitize=bool
-fsanitize=builtin
-fsanitize=bounds
-fsanitize=enum
-fsanitize=float-cast-overflow
-fsanitize=float-divide-by-zero
-fsanitize=function
-fsanitize=implicit-unsigned-integer-
truncation
-fsanitize=implicit-signed-integer-truncation
-fsanitize=implicit-integer-sign-change
-fsanitize=integer-divide-by-zero
-fsanitize=implicit-bitfield-conversion
-fsanitize=nonnull-attribute
-fsanitize=null
-fsanitize=nullability-arg
-fsanitize=nullability-assign
```

```
-fsanitize=nullability-return
-fsanitize=objc-cast
-fsanitize=object-size
-fsanitize=pointer-overflow
-fsanitize=return
-fsanitize=returns-nonnull-attribute
-fsanitize=shift
-fsanitize=unsigned-shift-base
-fsanitize=signed-integer-overflow
-fsanitize=unreachable
-fsanitize=unsigned-integer-overflow
-fsanitize=vla-bound
-fsanitize=vptr
```

Reducing undefined behavior: Edge status

- Coding guidelines.
- Code review.
- Clang-tidy checks.
- Undefined behavior sanitizer runs in CI pipeline.

Writer Fuzzer

Write Fuzzers

- Using libfuzzers
 - Coverage-guided fuzzing engine
 - Easy Crash reproduction
 - Integration with Sanitizers
 - Uses FuzzedDataProvider (from LLVM)

```
int Process(int a, int b) {  
    return a * b;  
}  
  
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {  
    FuzzedDataProvider provider(data, size);  
    int a = provider.ConsumeIntegral();  
    int b = provider.ConsumeIntegral();  
    Process(a, b);  
    return 0;  
}
```

Fuzzer - advanced

- Advanced fuzzers for browsers
 - **Domato**
 - **Type:** Generative DOM fuzzer.
 - **Usage:** Generates HTML/CSS/JS samples using grammars to fuzz browser DOM engines.
 - **Strengths:** Grammar-based generation.
 - **Fuzzilli**
 - **Type:** Coverage-guided JavaScript engine fuzzer.
 - **Usage:** Targets engines like V8 using an intermediate representation (FuzzIL).
 - **Strengths:**
 - Efficient for JIT and type confusion bugs.
 - Uses REPRL for fast execution.

Fuzzer - statistics

- Running fuzzers with Address Sanitizer.
- Last year, found
 - ~700 bugs
 - 200+ security bugs
 - Excluding External/duplicates.

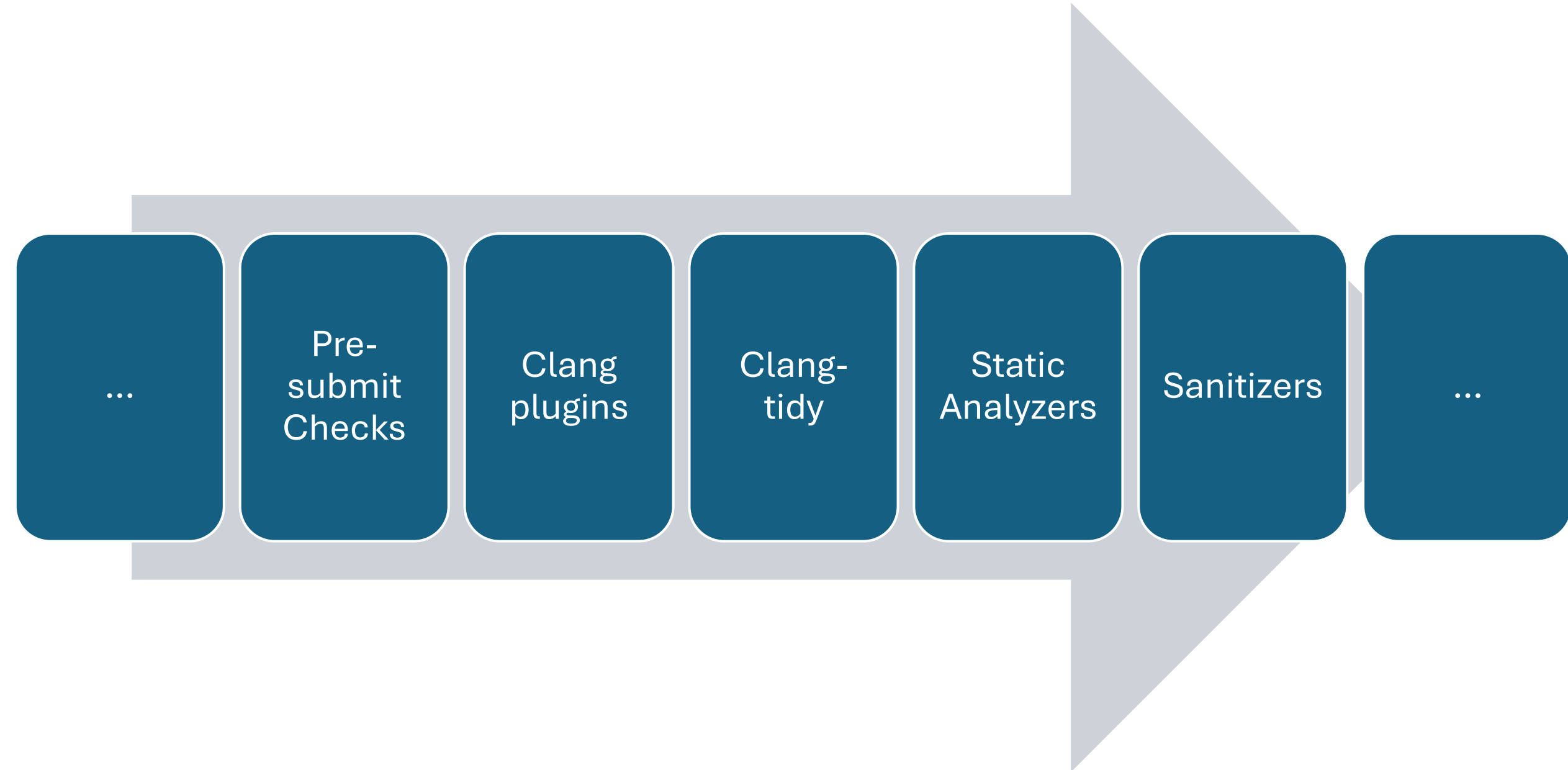
Code review / CI pipeline / After
release

Code review

- Reviewers run against a checklist of possible issues.
- Using pre-existing rules for bot comments.
- Using AI to assist with reviews.
- Add a security reviewer to code review.

CI Pipeline

100



CI pipeline: Presubmit checks

- Catch usage of program-specific unsafe APIs.
 - E.g., no use allowed for something like `chrome.send` or `memcpy`.
 - No use allowed for certain unsecure crypto library methods.
- Ensure review of the security critical files.
- Scan for passwords/secret-keys in the diff.

CI pipeline: Clang plugins

- Extra clang plugins for additional checks
 - `clang_plugin("find_bad_constructs")`
 - Missing "virtual" keywords on methods that should be virtual.
 - Classes that derive from `base::RefCounted` / `base::RefCountedThreadSafe` should have protected or private destructors.
 - `clang_plugin("raw_ptr_check")`
 - Ensures class data members declared as raw pointers (T^*) are replaced with `raw_ptr<T>`.
 - Encourages the use of `raw_ref<T>` for non-nullable references instead of raw references ($T\&$).
 - Flags unsafe casts between `raw_ptr<T>` and raw pointers that could bypass safety mechanisms, such as `reinterpret_cast` or `static_cast`.
 - Disallows `raw_ptr<T>` where T is a stack-allocated type, since such usage is unsafe.

CI pipeline: Clang-tidy checks

Checks: '-*,

bugprone-argument-comment,

bugprone-assert-side-effect,

bugprone-bool-pointer-implicit-conversion,

bugprone-dangling-handle,

bugprone-forward-declaration-namespace,

bugprone-inaccurate-erase,

bugprone-redundant-branch-condition,

bugprone-string-constructor,

bugprone-string-integer-assignment,

bugprone-suspicious-memset-usage,

bugprone-suspicious-realloc-usage,

bugprone-terminating-continue,

bugprone-undefined-memory-manipulation,

bugprone-unique-ptr-array-mismatch,

bugprone-unused-raii,

bugprone-use-after-move,

bugprone-virtual-near-miss,

...

CI pipeline: Run static analyzers

- Using Weggli queries
 - Lightweight compared to CodeQL
 - Uses clang's AST
 - Useful for finding security bugs

```
const char* getName() {  
    std::string name = "example";  
    return name.c_str(); // ❌ Dangling pointer - 'name' is destroyed at return  
}
```

weggli queries:

```
'const char* $func(...) { std::string $s = ...; return $s.c_str(); }'  
'const char* $func(...) { std::string $s = ...; return $s.data(); }'
```


CI pipeline: Sanitizers

- We run ASAN, MSAN, UBSAN pipelines for the tests periodically.
- These help in catching memory issues.

```
int main() {
    int* ptr = new int(42);
    delete ptr;
    // Use-after-free
    std::cout << *ptr << '\n'; // ✗
    return 0;
}
```

ASan Output

```
==12345==ERROR: AddressSanitizer: heap-use-after-free on
address 0x602000000010
READ of size 4 at 0x602000000010 thread T0
#0 0x... in main use_after_free.cpp:6
```

```
int main() {
    int x;
    // Use of uninitialized value
    std::cout << x << '\n'; // ✗
    return 0;
}
```

MSan Output

```
==12345==WARNING: MemorySanitizer: use-of-uninitialized-value
#0 0x... in main uninit.cpp:5
```


```
int main() {
    int a = INT_MAX; // Max value signed int
    int b = a + 1;    // UB: signed integer overflow
    std::cout << "b = " << b << '\n';
    return 0;
}
```

UBSan Output

```
==12345==WARNING: UndefinedBehaviorSanitizer: signed integer
overflow ubsan_example.cpp:7:13: runtime error: signed integer
overflow: 2147483647 + 1 cannot be represented in type 'int'
```

After Release

Using GWPAsan

- **Sampling-based memory error detector** designed to catch **use-after-free**, **heap buffer overflows**, and **double frees** in production environments with minimal overhead
-  **How It Works**
 - **Sampling Allocator**: Randomly selects a small fraction of heap allocations to be placed in a **guarded pool**.
 - **Guard Pages**: Surrounds selected allocations with **inaccessible memory pages** to catch overflows and underflows.
 - **Delayed Free**: Keeps freed allocations in quarantine to detect use-after-free.
 - **Crash on Violation**: Any illegal access triggers a **segmentation fault**, enabling crash reporting and debugging.
- **Stats:**
 - ~1200 bugs reported in last year.
 - Fixed ~140 security bugs

Conclusion

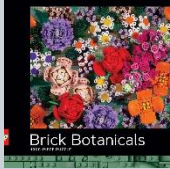




Final takeaways:

- Building secure C++ applications requires a multi-layered approach—there is no silver bullet.
- Incremental improvements are possible:
 - By adopting safer coding practices.
 - Leveraging modern toolchains (C++20, Clang, libc++ with hardening, -Wunsafe-buffer-usage, clang-tidy checks).
 - Using compile-time and runtime checks.
 - Using sanitizers and fuzzers.
- Focus on core safety categories: bounds, lifetime, initialization, and type safety. And on thread, and definition safety.
- Integrate security reviews, static analysis, and sanitizers throughout the development lifecycle—from design to release.

Additional notes

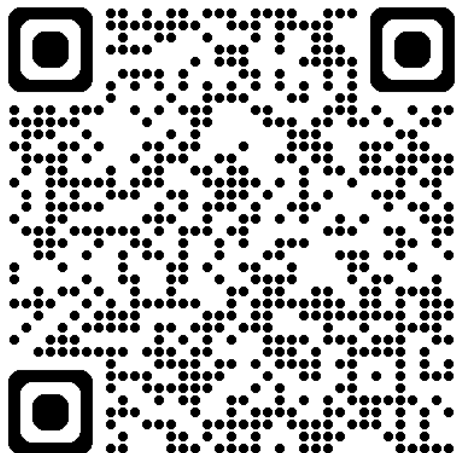
- Most of the code snippets used in the slides and additional ones are present in [github](#).
- Some more categories and code snippets are present in the appendix.
- We hope this talk gave you something interesting to use in your code.
- Big thank you to:
 - Wonderful Chromium developer team and our awesome Edge development team colleagues.
 - Kathleen Baker / Prithvi Okade for helping review the content for this presentation.
 - Wonderful C++ community for the books, blogs, videos, questions and answers.

Microsoft @ CppCon

Monday, Sept 15	Tuesday, Sept 16	Wednesday, Sept 17	Thursday, Sept 18	Friday, Sept 19
Building Secure Applications: A Practical End-to-End Approach Chandranath Bhattacharyya & Bharat Kumar 16:45 – 17:45	What's New for Visual Studio Code: Cmake Improvements and GitHub Copilot Agents Alexandra Kemper 09:00 – 10:00	LLMs in the Trenches: Boosting System Programming with AI Ion Todirel 14:00 – 15:00	MSVC C++ Dynamic Debugging: How We Enabled Full Debuggability of Optimized Code Eric Brumer 14:00 – 14:30	Reflection-based JSON in C++ at Gigabytes per Second Daniel Lemire & Francisco Geiman Thiesen 09:00 – 10:00
	What's new in Visual Studio for C++ Developers in 2025 Augustin Popa & David Li 14:00 – 15:00	C++ Performance Tips: Cutting Down on Unnecessary Objects Kathleen Baker & Prithvi Okade 15:15 – 16:15	It's Dangerous to Go Alone: A Game Developer Tutorial Michael Price 16:45 – 17:45	Duck-Tape Chronicles: Rust/C++ Interop Victor Ciura 13:30 – 14:30
	Back to Basics: Code Review Chandranath Bhattacharyya & Kathleen Baker 14:00 – 15:00	Connecting C++ Tools to AI Agents Using the Model Context Protocol Ben McMorran 15:50 – 16:20	Take our survey, win prizes https://aka.ms/cppcon/secure     	
		Welcome to v1.0 of the meta::[[verse]]! Inbal Levi 16:45 – 17:45		

Questions

Github link for code snippets:



Take our survey, win prizes
<https://aka.ms/cppcon/secure>



Appendix

Bounds Safety

libc++ hardening: valid-element-access

```
void ProblemExpected(std::expected<int, int> e) {
    // Will flag when called with "error" object.
    std::cout << "*e: " << *e << '\n';
    // Will flag when called with "error" object.
    std::cout << "e->operator(): " << (void*)e.operator->() << '\n';
}

void ProblemUnexpected(std::expected<int, int> e) {
    // Will flag when called with "success" object.
    std::cout << "e.error(): " << e.error() << '\n';
}

int main() {
    ProblemExpected(std::unexpected{1});
    ProblemUnexpected({});
}
```

expected:

- operator->, operator*: No “expected” object.
- error() : No “error” object.

Each of these calls will “trap” for these scenarios.

std::expected is “not” a container, but it falls under the ambit of valid-element-access category.

Clang: `-std=c++23 -stdlib=libc++ -O3`: No crash.

```
*e: 1
e->operator(): 0x7fffc626f4b8
e.error(): 0
```

libc++ hardening: valid-element-access

```
int main() {  
    const auto in = {1, 2, 3, 4, 5, 6};  
    std::vector<int> out;  
    auto gen = std::mt19937{std::random_device{}}();  
    // Passing negative sample size will flag.  
    std::ranges::sample(in, std::back_inserter(out), -1, gen);  
}
```

algorithm:

- sample – If $n < 0$ (n: number of samples to take)

Each of these calls will “trap” for these scenarios.

Algorithms `sort`, `partition`, `nth_element` also trap in some cases where the comparator does not provide strict weak ordering.

libc++ hardening: compatible-allocator

```
#include <new>
#include <set>

// A custom stateful allocator.
// It has an ID to differentiate instances.
template <class T>
class MyAllocator final {
public:
    using value_type = T;

    explicit MyAllocator(int id) : id_(id) {}
    template <class U>
    MyAllocator(const MyAllocator<U>& other) : id_(other.id_) {}

    T* allocate(size_t n) {
        return static_cast<T*>(::operator new(n * sizeof(T)));
    }
    void deallocate(T* p, size_t /*n*/) { ::operator delete(p); }
    friend bool operator==(const MyAllocator& a, const MyAllocator& b) {
        return a.id_ == b.id_;
    }

private:
    const int id_;
    template <class>
    friend class MyAllocator;
};
```

```
int main() {
    // Create two sets with different MyAllocator instances (id 1 and id 2).
    // The allocators are considered incompatible because their IDs differ.
    std::set<int, std::less<int>, MyAllocator<int>> set1{MyAllocator<int>(1)};
    std::set<int, std::less<int>, MyAllocator<int>> set2{MyAllocator<int>(2)};

    set1.insert(42);

    // Extract a node from the first set.
    // This node retains the allocator information from set1.
    auto node = set1.extract(42);

    // Attempt to insert the node into the second set.
    // Since the node's allocator (id 1) is not compatible with
    // set2's allocator (id 2), this will trigger the assertion.
    set2.insert(std::move(node));
}
```

Will crash here.

```
int main() {
    std::vector<int, MyAllocator<int>> vec1{1u, 2, MyAllocator<int>(1)};
    std::vector<int, MyAllocator<int>> vec2{1u, 3, MyAllocator<int>(2)};
    // Will crash here.
    vec1.swap(vec2);
}
```

libc++ hardening: non-overlapping-ranges

For functions that take several ranges as arguments, checks that those ranges do not overlap.

std::char_traits:

- `copy(char* s1, const char* s2, size_type n)`: *Fires when s1 and s2 ranges overlap.*

```
int main() {  
    std::string s("hello");  
    std::string_view sv(s);  
    std::string_view sub = sv.substr(1);  
    // Will flag if sub overlaps with s.  
    std::char_traits<char>::copy(s.data(), sub.data(), sub.size());  
    std::cout << s << '\n';  
}
```

It will “trap” for these scenarios.

Clang: `-std=c++20 -stdlib=libc++ -O3`: No crash.

`elloo`

libc++ hardening: argument-within-domain

Checks that the given argument is within the domain of valid arguments for the function.

Violating this typically produces an incorrect result or puts an object into an invalid state.

This category is for assertions violating which doesn't cause any immediate issues in the library – whatever the consequences are, they will happen in the user code.

`std::clamp (const T& v, const T& lo, const T& hi):` *Fires if $lo > hi$*

```
int main() {  
    // Will flag when hi < lo.  
    const auto v = std::clamp(10, 2, 1);  
    std::cout << v << '\n';  
}
```

It will “trap” for these scenarios.

Clang: `-std=c++20 -stdlib=libc++ -O3`: No crash.

1

libc++ hardening: pedantic

Checks preconditions that are imposed by the C++ standard, but violating which ***happens to be benign*** in libc++.

```
int main() {
    std::vector<int> v;
    std::ranges::make_heap(v);
    // Will flag when called with empty range.
    std::ranges::pop_heap(v);
}
```

algorithm:

- pop_heap: empty range.

```
int main() {
    std::cmatch match;
    // All these calls will flag since match is not yet ready.
    std::cout << "Length: " << match.length() << '\n';
    std::cout << "Position: " << match.position() << '\n';
    std::cout << "str: " << match.str() << '\n';
    std::cout << "match[0]: " << match[0] << '\n';
    std::cout << "prefix: " << match.prefix() << '\n';
    std::cout << "suffix: " << match.suffix() << '\n';
}
```

std::cmatch:


- length, position, str, operator[], prefix, suffix:
 - When called if match is not-ready.

Benign and will print the following using libc++ without
_LIBCPP_HARDENING_MODE_EXTENSIVE

```
Length: 0
Position: 0
str:
match[0]:
prefix:
suffix:
```



-Wunsafe-buffer-usage

More transformation examples:



```
int main() {  
    int* arr = new int[10]{};  
    for (int i = 0; i < 10; ++i) {  
        arr[i] = i;  
    }  
    delete[] arr;  
}
```

error: unsafe buffer access [-Werror, -Wunsafe-buffer-usage]
arr[i] = i;
^~~



```
int main() {  
    auto arr = std::make_unique<int[]>(10);  
    for (int i = 0; i < 10; ++i) {  
        arr[i] = i;  
    }  
}
```

Lifetime Safety

Use raw pointers carefully

```
class Base {  
public:  
    virtual ~Base() = default;  
    int GetId() const { return id_; }  
    virtual void Foo() = 0;  
  
protected:  
    Base(int id) : id_(id) {}  
    const int id_;  
};
```

```
void Foo(std::function<void()> fn) {  
    fn();  
}
```

```
class Derived1 final : public Base {  
public:  
    Derived1(int i) : Base(i) {}  
    void Foo() final { std::cout << "Derived1::Foo()\n"; }  
};  
  
class Derived2 final : public Base {  
public:  
    Derived2(int i) : Base(i) {}  
    void Foo() final { std::cout << "Derived2::Foo()\n"; }  
};
```

```
int main() {  
    std::unique_ptr<Base> b = std::make_unique<Derived1>(1);  
    Foo([ptr = b.get()] { ptr->Foo(); });  
}
```

Derived1::Foo()

Use raw pointers carefully

```
class Base {
public:
    virtual ~Base() = default;
    int GetId() const { return id_; }
    virtual void Foo() = 0;

protected:
    Base(int id) : id_(id) {}
    const int id_;
};
```

```
class Derived1 final : public Base {
public:
    Derived1(int i) : Base(i) {}
    void Foo() final { std::cout << "Derived1::Foo()\n"; }
};

class Derived2 final : public Base {
public:
    Derived2(int i) : Base(i) {}
    void Foo() final { std::cout << "Derived2::Foo()\n"; }
};
```

```
class ObjectManager {
public:
    Base* CreateDerived1() { return Create<Derived1>(); }
    Base* CreateDerived2() { return Create<Derived2>(); }

private:
    template <typename T>
    Base* Create() {
        auto p = std::make_unique<T>(++counter_);
        auto* ptr = p.get();
        m_.emplace(ptr->GetId(), std::move(p));
        return ptr;
    }

    int counter_ = 0;
    std::map<int, std::unique_ptr<Base>> m_;
};
```

```
class SomeManager {
public:
    void AddHandler(Base* p) {
        fns_.push_back([p] { p->Foo(); });
    }
    void RunFunctions() {
        for (auto&& f : fns_) {
            f();
        }
    }
private:
    std::vector<std::function<void()>> fns_;
};
```

```
int main() {
    ObjectManager o;
    SomeManager s;
    s.AddHandler(o.CreateDerived1());
    s.AddHandler(o.CreateDerived2());
    s.RunFunctions();
}
```

```
Derived1::Foo()
Derived2::Foo()
```

Use raw pointers carefully

```
class ObjectManager {  
public:  
    Base* CreateDerived1() { return Create<Derived1>(); }  
    Base* CreateDerived2() { return Create<Derived2>(); }  
private:  
    template <typename T>  
    Base* Create() {  
        auto p = std::make_unique<T>(++counter_);  
        auto* ptr = p.get();  
        m_.emplace(ptr->GetId(), std::move(p));  
        return ptr;  
    }  
  
    int counter_ = 0;  
    std::map<int, std::unique_ptr<Base>> m_;  
};
```

```
class SomeManager {  
public:  
    void AddHandler(Base* p) {  
        fns_.push_back([p] { p->Foo(); });  
    }  
    void RunFunctions() {  
        for (auto&& f : fns_) {  
            f();  
        }  
    }  
private:  
    std::vector<std::function<void()>> fns_;  
};
```

```
int main() {  
    ObjectManager o;  
    SomeManager s;  
    s.AddHandler(o.CreateDerived1());  
    s.AddHandler(o.CreateDerived2());  
    s.RunFunctions();  
}
```

Use raw pointers carefully

```
class ObjectManager {
public:
    Base* CreateDerived1() { return Create<Derived1>(); }
    Base* CreateDerived2() { return Create<Derived2>(); }

    void RemoveFirst() { m_.erase(m_.begin()); }

private:
    template <typename T>
    Base* Create() {
        auto p = std::make_unique<T>(++counter_);
        auto* ptr = p.get();
        m_.emplace(ptr->GetId(), std::move(p));
        return ptr;
    }

    int counter_ = 0;
    std::map<int, std::unique_ptr<Base>> m_;
};
```

```
class SomeManager {
public:
    void AddHandler(Base* p) {
        fns_.push_back([p] { p->Foo(); });
    }
    void RunFunctions() {
        for (auto&& f : fns_) {
            f();
        }
    }
private:
    std::vector<std::function<void()>> fns_;
};
```

```
int main() {
    ObjectManager o;
    SomeManager s;
    s.AddHandler(o.CreateDerived1());
    s.AddHandler(o.CreateDerived2());
    o.RemoveFirst();
    s.RunFunctions();
}
```

Program terminated with signal: SIGSEGV

Virtual function call attempted on deleted object.

Any time a raw pointer is stored for “future” use, we need to be very careful about using it.

Use raw pointers carefully

```
class ObjectManager {
public:
    Base* CreateDerived1() { return Create<Derived1>(); }
    Base* CreateDerived2() { return Create<Derived2>(); }
    Base* GetObject(int i) const {
        auto it = m_.find(i);
        if (it == m_.end()) {
            return nullptr;
        }
        return it->second.get();
    }
    void RemoveFirst() { m_.erase(m_.begin()); }

private:
    template <typename T>
    Base* Create() {
        auto p = std::make_unique<T>(++counter_);
        auto* ptr = p.get();
        m_.emplace(ptr->GetId(), std::move(p));
        return ptr;
    }

    int counter_ = 0;
    std::map<int, std::unique_ptr<Base>> m_;
};
```

```
class SomeManager {
public:
    SomeManager(ObjectManager* o) : o_(o) {}
    void AddHandler(Base* p) {
        fns_.push_back([o = o_, id = p->GetId()] {
            auto* p = o->GetObject(id);
            if (p) {
                p->Foo();
            }
        });
    }

    void RunFunctions() {
        for (auto&& f : fns_) {
            f();
        }
    }

private:
    ObjectManager* o_;
    std::vector<std::function<void()>> fns_;
};
```

```
int main() {
    ObjectManager o;
    SomeManager s(&o);
    s.AddHandler(o.CreateDerived1());
    s.AddHandler(o.CreateDerived2());
    o.RemoveFirst();
    s.RunFunctions();
}
```

```
Derived1::Foo()
```

Instead of storing the raw pointer, store an “id” using which it can be retrieved back later.

Use raw pointers carefully

```
class ObjectManager {
public:
    std::weak_ptr<Base> CreateDerived1() {
        return Create<Derived1>();
    }
    std::weak_ptr<Base> CreateDerived2() {
        return Create<Derived2>();
    }
    void RemoveFirst() { vec_.erase(vec_.begin()); }

private:
    template <typename T>
    std::weak_ptr<Base> Create() {
        auto p = std::make_shared<T>(++counter_);
        vec_.push_back(p);
        return p;
    }

    int counter_ = 0;
    std::vector<std::shared_ptr<Base>> vec_;
};
```

```
class SomeManager {
public:
    void AddHandler(std::weak_ptr<Base> p) {
        fns_.push_back([wp = p] {
            if (auto p = wp.lock()) {
                p->Foo();
            }
        });
    }

    void RunFunctions() {
        for (auto&& f : fns_) {
            f();
        }
    }

private:
    std::vector<std::function<void()>> fns_;
};
```

```
int main() {
    ObjectManager o;
    SomeManager s;
    s.AddHandler(o.CreateDerived1());
    s.AddHandler(o.CreateDerived2());
    o.RemoveFirst();
    s.RunFunctions();
}
```

```
Derived2::Foo()
```

shared_ptr and weak_ptr can be another possible solution.

Using shared_ptr carefully

```
struct MyClass {
    MyClass() { std::cout << "MyClass()\n"; }
    ~MyClass() { std::cout << "~MyClass()\n"; } ← 6

    void Foo(std::function<void()> fn) {
        std::cout << "MyClass::Foo before fn\n";
        fn(); ← 3
        std::cout << "MyClass::Foo after fn\n"; ← 7
    }
};
```

```
int main() {
    ObjectManager o;
    const auto id = o.AddObject();
    o.CallFooOnObject(id, [id, po = &o]() { ← 1
        po->Remove(id); ← 4
    });
}
```

```
MyClass()
MyClass::Foo before fn
~MyClass()
MyClass::Foo after fn
```

```
struct ObjectManager {
public:
    int AddObject() {
        const auto id = ++counter_;
        m_.emplace(id, std::make_shared<MyClass>());
        return id;
    }

    void Remove(int id) { m_.erase(id); } ← 5

    void CallFooOnObject(int id, std::function<void()> fn) {
        auto it = m_.find(id);
        if (it != m_.end()) {
            it->second->Foo(fn); ← 2
        }
    }

private:
    int counter_ = 0;
    std::map<int, std::shared_ptr<MyClass>> m_;
};
```

Using shared_ptr carefully

```
struct MyClass :
    public std::enable_shared_from_this<MyClass> {
    MyClass() { std::cout << "MyClass()\n"; }
    ~MyClass() { std::cout << "~MyClass()\n"; } ← 7

    void Foo(std::function<void()> fn) {
        std::cout << "MyClass::Foo before fn\n";
        auto strong_this = shared_from_this();
        fn(); ← 3
        std::cout << "MyClass::Foo after fn\n"; ← 6
    }
};
```

```
int main() {
    ObjectManager o;
    const auto id = o.AddObject();
    o.CallFooOnObject(id, [id, po = &o]() { ← 1
        po->Remove(id); ← 4
    });
}
```

```
struct ObjectManager {
public:
    int AddObject() {
        const auto id = ++counter_;
        m_.emplace(id, std::make_shared<MyClass>());
        return id;
    }

    void Remove(int id) { m_.erase(id); } ← 5

    void CallFooOnObject(int id, std::function<void()> fn) {
        auto it = m_.find(id);
        if (it != m_.end()) {
            it->second->Foo(fn); ← 2
        }
    }

private:
    int counter_ = 0;
    std::map<int, std::shared_ptr<MyClass>> m_;
};
```

```
MyClass()
MyClass::Foo before fn
MyClass::Foo after fn
~MyClass()
```

Use attributes like `[[clang::lifetimebound]]`

```
class MyClass {
public:
    MyClass(const std::string& s) : s_(s) {}
    void Print() const { std::cout << s_ << '\n'; }

private:
    std::string_view s_;
};
```

```
std::string GetStr() {
    return "hello, how are you!";
}

int main() {
    MyClass m{GetStr()};
    m.Print();
}
```

4Q?k??Q?

```
class MyClass {
public:
    MyClass(const std::string& s [[clang::lifetimebound]]) : s_(s) {}
    void Print() const { std::cout << s_ << '\n'; }

private:
    std::string_view s_;
};
```

error: temporary whose address is used as value of local variable 'm1' will be destroyed at the end of the full-expression [-Werror,-Wdangling]

```
MyClass m1{GetStr()};
          ^~~~~~
```

Use attributes like `[[clang::lifetimebound]]`

```
std::string_view GetOrDefault(std::string_view str,
                             std::string_view key) {
    if (str.starts_with(key)) {
        return str.substr(key.size());
    }
    return std::string_view{};
}
```

```
std::string GetStr() {
    return "hello|world|and a longer string";
}
```

```
int main() {
    for (auto c : GetOrDefault(GetStr(), "hello|")) {
        std::cout << c;
    }
    std::cout << '\n';
}
```

C++20: `k??0F}` a longer string

C++23: `world|and a longer string`

[P2644](#) fixed this lifetime issue for range based for loop in C++23.

```
std::string_view GetOrDefault(std::string_view str [[clang::lifetimebound]],
                             std::string_view key);
```

C++23:

```
int main() {
    for (auto c : GetOrDefault(GetStr(), "hello|")) {
        std::cout << c;
    }
    std::cout << '\n';
}
```



error: object backing the pointer will be destroyed at the end of the full-expression [-Werror,-Wdangling-gsl]
for (auto c : GetOrDefault(GetStr(), "hello|")) {
~~~~~

This attribute “does not” consider C++23 changes.

Temporary binding



```
struct X {  
    X(int) {}  
};  
  
struct S2 {  
    const X &x;  
    S2(int i) : x(i) {}  
};
```

error: reference member 'x' binds to a temporary object whose lifetime would be shorter than the lifetime of the constructed object

```
    S2(int i) : x(i) {}  
                ^
```

note: reference member declared here

```
    const X &x;  
        ^
```

This construct causes “error” even without any flags in clang


With GCC `-Wextra` flag is necessary to start flagging this construct.

Clang-tidy checks

```
struct S {
    int v;
    constexpr S(int v) : v(v) {}
};

constexpr const S& fn(const S& a) {
    return a;
}
```

```
int main() {
    [[maybe_unused]] const S& s = fn(S{1});
    return s.v;
}
```



With clang-tidy `-checks=bugprone-return-const-ref-from-parameter`


warning: returning a constant reference parameter may cause use-after-free when the parameter is constructed from a temporary
[bugprone-return-const-ref-from-parameter]
 return a;
 ^

```
int main() {
    constexpr auto i = fn(S{1}).v;
    return i;
}
```

There is “no” undefined behavior here, but the same clang-tidy warning shows up.

So, this warning is not related to usage, but just the pattern in code.

Clang -Wdangling: Enabled by default




```
int main() {
    const char* pc = std::string{"hello"}.c_str();
    std::cout << pc << '\n';
}
```

error: object backing the pointer will be destroyed at the end of the full-expression [-Werror, -Wdangling-gsl]

```
const char* pc = std::string{"hello"}.c_str();
^~~~~~
```

```
struct String {
    const char* c_str() const { return str_.c_str(); }
    std::string str_{"Hello"};
};
```



```
int main() {
    const char* pc = String{}.c_str();
    std::cout << pc << '\n';
}
```

Currently works for **only** for some STL types.

Clang -Wdangling: Enabled by default

Currently works for **only** for some STL types.

```
struct String {
    const char* c_str() const { return str_; }
    const char* str_ = "Hello";
};

struct StringView {
    StringView(const String&) {}
};
```



```
struct [[gsl::Owner]] String {
    const char* c_str() const { return str_; }
    const char* str_ = "Hello";
};

struct [[gsl::Pointer]] StringView {
    StringView(const String&) {}
};
```

```
int main() {
    [[maybe_unused]] StringView sv = String{};
}
```




```
int main() {
    [[maybe_unused]] StringView sv = String{};
}
```



error: object backing the pointer will be destroyed at the end of the full-expression [-Werror,-Wdangling-gsl]
 [[maybe_unused]] StringView sv = String{};
 ^~~~~~


Reduce undefined behavior

Use constexpr, constinit, const variables



```
void IntegerDivisionOverflow() {
    const int a = INT_MIN;
    const int b = -1;
    const int c = a / b;
    std::cout << "IntegerDivisionOverflow: c = "
                << c << '\n';
}

void NaNPropagation() {
    const double a = NAN;
    const double b = a + 1;
    std::cout << "NaNPropagation: b = " << b << '\n';
}
```



```
void IntegerDivisionOverflow() {
    const int a = INT_MIN;
    const int b = -1;
    constexpr int c = a / b;
    std::cout << "IntegerDivisionOverflow: c = "
                << c << '\n';
}

void NaNPropagation() {
    const double a = NAN;
    constexpr double b = a + 1;
    std::cout << "NaNPropagation: b = " << b << '\n';
}
```

```
error: constexpr variable 'c' must be initialized by a constant expression
constexpr int c = a / b;
                  ^ ~~~~~
note: value 2147483648 is outside the range of representable values of type 'int'
constexpr int c = a / b;
                  ^
error: constexpr variable 'b' must be initialized by a constant expression
constexpr double b = a + 1;
                  ^ ~~~~~
note: read of non-constexpr variable 'a' is not allowed in a constant expression
constexpr double b = a + 1;
                  ^
note: declared here
const double a = NAN;
              ^
```

Maintain consistent state

Handle multiple std::move for a single variable.

```
bool IsInteresting(const std::string& s) {  
    return s.length() > 3u;  
}
```

```
std::vector<std::string> GetStringVec(std::string s) {  
    std::vector<std::string> vec;  
    vec.push_back(std::move(s));  
    if (IsInteresting(s)) {  
        vec.push_back(std::move(s));  
    }  
    return vec;  
}
```

The first move means that this code is incorrect.



Be careful to ensure that we don't “move” from same variable multiples times.

```
std::vector<std::string> GetStringVec(std::string s) {  
    std::vector<std::string> vec;  
    if (IsInteresting(s)) {  
        vec.push_back(s);  
        vec.push_back(std::move(s));  
    } else {  
        vec.push_back(std::move(s));  
    }  
    return vec;  
}
```

Handle multiple std::move for a single variable.

```
bool IsInteresting(const std::string& s) {  
    return s.length() > 3u;  
}
```

```
std::vector<std::string> GetStringVec(std::string s) {  
    std::vector<std::string> vec;  
    vec.push_back(std::move(s));  
    if (IsInteresting(s)) {  
        vec.push_back(std::move(s));  
    }  
    return vec;  
}
```

When clang-tidy check is used: --**checks=**bugprone-use-after-move

```
warning: 's' used after it was moved [bugprone-use-after-move]  
    if (IsInteresting(s)) {  
    ^  
note: move occurred here  
    vec.push_back(std::move(s));  
    ^
```

Don't use std::move / std::forward for callbacks in a loop.¹⁴²

```
template <typename... Args>
void CallAll(const std::vector<std::function<void(Args...)>>& funcs,
             Args&&... args) {
    for (const auto& func : funcs) {
        func(std::forward<Args>(args)...);
    }
}
```

When clang-tidy check is used: `--checks=bugprone-use-after-move`

```
warning: 'args' used after it was forwarded [bugprone-use-after-move]
    func(std::forward<Args>(args)...);
      ^
note: forward occurred here
    func(std::forward<Args>(args)...);
      ^
note: the use happens in a later loop iteration than the forward
    func(std::forward<Args>(args)...);
      ^
```

Don't use std::move / std::forward for callbacks in a loop. ¹⁴³

```
template <typename... Args>
void CallAll(const std::vector<std::function<void(Args...)>>& funcs,
            Args&&... args) {
    for (const auto& func : funcs) {
        func(std::forward<Args>(args)...);
    }
}
```

```
int main() {
    std::vector<std::function<void(std::string)>> funcs{Func1, Func2};
    CallAll(funcs, std::string{"hello world"});
}
```

```
void Func1(std::string s) {
    printf("Func1(%s)\n", s.c_str());
}

void Func2(std::string s) {
    printf("Func2(%s)\n", s.c_str());
}
```

```
Func1(hello world)
Func2()
```

In calling **Func1**, **std::string::move** was called so **Func2** is printing empty string.

```
template <typename... Args>
void CallAll(const std::vector<std::function<void(Args...)>>& funcs,
            Args&&... args) {
    for (const auto& func : funcs) {
        func(args...);
    }
}
```

```
Func1(hello world)
Func2(hello world)
```

This version makes copies of strings.

If we are calling multiple functions using variadic arguments, then consider *removing* std::forward.

Don't use std::move / std::forward for callbacks in a loop.¹⁴⁴

```
template <typename... Args>
void CallAll(const std::vector<std::function<void(const Args&...)>>& funcs,
            Args&&... args) {
    for (const auto& func : funcs) {
        func(args...);
    }
}
```

```
int main() {
    std::vector<std::function<void(const std::string&)>> funcs{Func1, Func2};
    CallAll(funcs, std::string{"hello world"});
}
```

```
void Func1(const std::string& s) {
    printf("Func1(%s)\n", s.c_str());
}

void Func2(const std::string& s) {
    printf("Func2(%s)\n", s.c_str());
}
```

```
Func1(hello world)
Func2(hello world)
```

This approach removes the copies.

Maintain consistent state

- Add invariant checks.
- Use checks that guarantee program termination for failures instead of letting program get into inconsistent state.

```
enum class MyEnum { kA, kB };
```

```
// Reading back some data from disk.  
std::string_view ToString(int v) {  
    const auto e = static_cast<MyEnum>(v);  
    switch (e) {  
        case MyEnum::kA:  
            return "A";  
        case MyEnum::kB:  
            return "B";  
    }  
}
```

Incorrect data can cause undefined state.

```
        case MyEnum::kB:  
            return "B";  
    }  
    return std::string_view{};  
}
```



```
// Reading back some data from disk.  
std::string_view ToString(int v) {  
    const auto e = static_cast<MyEnum>(v);  
    switch (e) {  
        case MyEnum::kA:  
            return "A";  
        case MyEnum::kB:  
            return "B";  
        case MyEnum::kC:  
            return "C";  
    }  
    NOTREACHED_NORETURN();  
}
```

Fail-fast in case we get “inconsistent” data.

Maintain consistent state

- Move away from checks which fire only in debug build.

```
void MyFoo(MyClass* pc) {  
    // Check only in debug build.  
    DCHECK(pc && pc->IsValid());  
    pc->Foo();  
}
```



```
void MyFoo(MyClass* pc) {  
    // Fail-fast if invariant does not hold.  
    CHECK(pc && pc->IsValid());  
    pc->Foo();  
}
```

Maintaining consistent state: Edge status

- Coding guidelines.
- Code review.
- Clang-tidy checks

Arithmetic Safety

Use STL functions when possible.

```
void CheckMid(int a, int b) {  
    const auto naive_mid = (a + b) / 2; // UB if a + b overflows  
  
    // Safe midpoint (C++20)  
    const auto safe_mid = std::midpoint(a, b);  
  
    std::cout << "a = " << a << '\n';  
    std::cout << "b = " << b << '\n';  
    std::cout << "naive_mid = " << naive_mid  
                << " (undefined behavior possible)\n";  
    std::cout << "std::midpoint = " << safe_mid << " (well-defined)\n";  
}
```

```
int main() {  
    const auto a = std::numeric_limits<int>::max();  
    const auto b = std::numeric_limits<int>::max();  
    CheckMid(a, b);  
}
```

```
a = 2147483647  
b = 2147483647  
naive_mid = -1 (undefined behavior possible)  
std::midpoint = 2147483647 (well-defined)
```

Use standard library functions like midpoint, lerp instead of own implementations.

Sometimes algorithms need to change

```
void ProblemSum() {  
    float sum = 0;  
    for (size_t i = 0; i < 1e8; ++i) {  
        sum += 1.f;  
    }  
    std::cout << "ProblemSum: " << sum << '\n';  
}
```

ProblemSum: 1.67772e+07

This issue is caused by limited precision of floating point.



Compensated (Kahan) summation

```
void KahanSum() {  
    float sum = 0.0f;  
    float c = 0.0f; // A running compensation for lost low-order bits.  
    const float term = 1.0f;  
    for (size_t i = 0; i < 1e8; ++i) {  
        float y = term - c; // y = next term - compensation.  
        float t = sum + y; // t = sum + y  
        c = (t - sum) - y; // c = (t - sum) - y, this is the lost part  
        sum = t; // sum becomes the new sum.  
    }  
    std::cout << "Kahan summation: " << sum << '\n';  
}
```

Kahan summation: 1e+08

Use saturated arithmetic (C++26).

- In **saturated arithmetic**, the result is clamped to the nearest boundary rather than overflow:
 - If the result is **greater than the max**, it becomes the **max**.
 - If the result is **less than the min**, it becomes the **min**.

```
#include <iostream>
#include <saturation>
#include <limits>

int main() {
    std::saturated<int> a = std::numeric_limits<int>::max();
    std::saturated<int> b = 1;
    auto result = a + b; // Saturates to INT_MAX
    std::cout << static_cast<int>(result) << std::endl; // Prints INT_MAX
}
```

Use checked math functions.

- Checked math functions return a `std::expected<T, std::errc>` type:
 - T is the result type (e.g., `int`).
 - If the operation is successful, the result is stored.
 - If it fails (e.g., due to overflow), an error code is returned.

```
#include <checked_math>

int main() {
    int a = std::numeric_limits::max();
    int b = 1;
    auto result = std::checked_add(a, b);
    if (result) {
        std::cout << "Sum: " << *result << std::endl;
    } else {
        std::cout << "Overflow occurred!" << std::endl;
    }
    return 0;
}
```

Available Checked Math Functions

From `<checked_math>`:

- `std::checked_add(a, b)`
- `std::checked_sub(a, b)`
- `std::checked_mul(a, b)`
- `std::checked_div(a, b)`
- `std::checked_mod(a, b)`
- `std::checked_neg(a)`

Handle division by zero.

- Use `checked_div` for C++ 26.
- No protection with saturated arithmetic (C++ 26).
- Always check before division

```
#include <checked_math>
#include <expected>
#include <iostream>

int main(){
    int a=10,b=0;
    auto result = std::checked_div(a,b);
    if (result){
        std::cout<<"Result:"<<* result<<"\n";
    }
    else {
        std::cout<<"Divisionbyzero!\n";
    }
}
```

```
int a = 10, b = 0;
if (b != 0) {
    int result = a / b;
    std::cout << "Result: " << result << "\n";
} else {
    std::cout << "Division by zero!\n";
}
```

Chromium implementations

Chromium uses its own implementations for clamped/checked arithmetic

- [base/numerics/clamped_math.h](#)
 - contains the ClampedNumeric template class and helper functions for performing fast, clamped (i.e. non-sticky saturating) arithmetic operations and conversions.
- [base/numerics/checked_math.h](#)
 - contains the CheckedNumeric template class and helper functions for performing arithmetic and conversion operations that detect errors and boundary conditions (e.g. overflow, truncation, etc.).
- [base/numerics/safe_conversions.h](#)
 - contains the StrictNumeric template class and a collection of custom casting templates and helper functions for safely converting between a range of numeric types.
- [Documented here.](#)

SafeInt library

- [dcleblanc/SafeInt](#): SafeInt is a class library for C++ that manages integer overflows.

Improving arithmetic safety: Edge status

- Coding guidelines.
- Code review.
- Sanitizers.