C++ now

Security in C++

Hardening Techniques from the Trenches

Louis Dionne

2024

Who Am I Quoting?

Experts have identified a few programming languages that both **lack traits associated with memory safety** and also have high proliferation across critical systems, such as **C and C++**.

Choosing to use memory safe programming languages at the outset, as recommended by the Cybersecurity and Infrastructure Security Agency's (CISA) Open-Source Software Security Roadmap is one example of developing software in a secure-by-design manner. *

* https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

How Important Is Security?

- Cybercrime is estimated to cost 10 trillion by 2025
- Spyware like Pegasus targeted journalists and high-profile activists
- The WannaCry ransomware affected over 300k computers in 150 countries, cost over $4B
  - Affected critical systems like hospitals

C++ Is Memory Unsafe

Memory Unsafety Accounts For ~70% Of High Severity Security Issues *

* https://www.chromium.org/Home/chromium-security/memory-safety/

We Can Do Better

We Must Do Better

Partly an Attitude Problem

C++ has generally adopted an expert-friendly attitude:

- If the user makes a mistake, it’s their fault
- Performance at all costs
- Complicated APIs in the name of flexibility

The Mindset Is Changing

- More general awareness about the problem
- Creation of SG23 (Safety and Security Study Group)
- Most of WG21 understands this is a make or break
- However, still few concrete improvements

We’re Engineers, Let’s Solve Problems

Agenda

Overview of Memory Safety

Library Undefined Behavior

Standard Library Hardening

Typed Memory Operations

Conclusions

Types of Memory Safety

- **Spatial memory safety**
- **Temporal memory safety**
- Type safety
- Guaranteed initialization
- Thread safety

Spatial memory safety

- Each memory allocation has a given size (or bounds)
- Accessing memory out of bounds is called an out-of-bounds (OOB) access

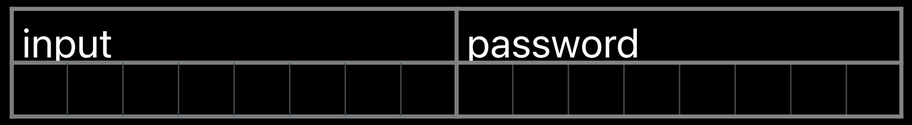
Example

```cpp
int main() {
  char input[8];
  char password[8];

  std::ifstream("/etc/password") >> password;

  std::cout << "Enter password: ";
  std::cin >> input;

  if (std::strncmp(password, input, 8) == 0)
    std::cout << "Access granted";
  else
    std::cout << "Access denied";
}
```




```
$ authenticate

> securitysecurity
```

Inspired from https://snyk.io/blog/buffer-overflow-attacks-in-c/

Example

```cpp
int main() {
  char input[8];
  char password[8];

  std::ifstream("/etc/password") >> password;

  std::cout << "Enter password: ";
  std::cin >> input;

  if (std::strncmp(password, input, 8) == 0)
    std::cout << "Access granted";
  else
    std::cout << "Access denied";
}
```

input								password							
								r	e	a	l	p	a	s	s

```
$ authenticate

> securitysecurity
```

Inspired from https://snyk.io/blog/buffer-overflow-attacks-in-c/

Example

```
int main() {
  char input[8];
  char password[8];

  std::ifstream("/etc/password") >> password;

  std::cout << "Enter password: ";
  std::cin >> input;

  if (std::strncmp(password, input, 8) == 0)
    std::cout << "Access granted";
  else
    std::cout << "Access denied";
}
```

input								password							
s	e	c	u	r	i	t	y	r	e	a	l	p	a	s	s

```
$ authenticate

> securitysecurity
```

Inspired from https://snyk.io/blog/buffer-overflow-attacks-in-c/

Example

```c
int main() {
  char input[8];
  char password[8];

  std::ifstream("/etc/password") >> password;

  std::cout << "Enter password: ";
  std::cin >> input;

  if (std::strncmp(password, input, 8) == 0)
    std::cout << "Access granted";
  else
    std::cout << "Access denied";
}
```

```
$ authenticate

> securitysecurity
```

input								password							
s	e	c	u	r	i	t	y	s	e	c	u	r	i	t	y

Inspired from https://snyk.io/blog/buffer-overflow-attacks-in-c/

Example

```
int main() {
  char input[8];
  char password[8];

  std::ifstream("/etc/password") >> password;

  std::cout << "Enter password: ";
  std::cin >> input;

  if (std::strncmp(password, input, 8) == 0)
    std::cout << "Access granted";
  else
    std::cout << "Access denied";
}
```

✅

```
$ authenticate

> securitysecurity
```

input								password							
s	e	c	u	r	i	t	y	s	e	c	u	r	i	t	y

🚨

Inspired from https://snyk.io/blog/buffer-overflow-attacks-in-c/

Now Imagine This

```
using Authentifier = bool (*)(Account const&, Password const&);

struct Account {
  ...
  int transit_number[5];
  int account_number[8];
  Authentifier identification_method = &two_factor_auth;
  ...
};

void teller(FILE* socket) {
  while (true) {
    char const* data = read_socket(socket); // untrusted

    Account account;
    parse_transit_number(data, &account.transit_number);
    parse_account_number(data, &account.account_number);
    auto authenticate = account.identification_method;
    if (authenticate(account, password)) {
      // we're in!
    }
    free(data);
  }
}
```

Now Imagine This

```
using Authentifier = bool (*)(Account const&, Password const&);

struct Account {
  ...
  int transit_number[5];
  int account_number[8];
  Authentifier identification_method = &two_factor_auth;
  ...
};

void teller(FILE* socket) {
  while (true) {
    char const* data = read_socket(socket); // untrusted

    Account account;
    parse_transit_number(data, &account.transit_number);
    parse_account_number(data, &account.account_number);
    auto authenticate = account.identification_method;
    if (authenticate(account, password)) {
      // we're in!
    }
    free(data);
  }
}
```

Now Imagine This

```
using Authentifier = bool (*)(Account const&, Password const&);

struct Account {
  ...
  int transit_number[5];
  int account_number[8];
  Authentifier identification_method = &two_factor_auth;
  ...
};

void teller(FILE* socket) {
  while (true) {
    char const* data = read_socket(socket); // untrusted

    Account account;
    parse_transit_number(data, &account.transit_number);
    parse_account_number(data, &account.account_number);
    auto authenticate = account.identification_method;
    if (authenticate(account, password)) {
      // we're in!
    }
    free(data);
  }
}
```

Now Imagine This

```
using Authentifier = bool (*)(Account const&, Password const&);

struct Account {
  ...
  int transit_number[5];
  int account_number[8];
  Authentifier identification_method = &two_factor_auth;
  ...
};

void teller(FILE* socket) {
  while (true) {
    char const* data = read_socket(socket); // untrusted

    Account account;
    parse_transit_number(data, &account.transit_number);
    parse_account_number(data, &account.account_number);
    auto authenticate = account.identification_method;
    if (authenticate(account, password)) {
      // we're in!
    }
    free(data);
  }
}
```

Now Imagine This

```
using Authentifier = bool (*)(Account const&, Password const&);

struct Account {
  ...
  int transit_number[5];
  int account_number[8];
  Authentifier identification_method = &two_factor_auth;
  ...
};

void teller(FILE* socket) {
  while (true) {
    char const* data = read_socket(socket); // untrusted

    Account account;
    parse_transit_number(data, &account.transit_number);
    parse_account_number(data, &account.account_number);
    auto authenticate = account.identification_method;
    if (authenticate(account, password)) {
      // we're in!
    }
    free(data);
  }
}
```

Now Imagine This

```
using Authentifier = bool (*)(Account const&, Password const&);

struct Account {
  ...
  int transit_number[5];
  int account_number[8];
  Authentifier identification_method = &two_factor_auth;
  ...
};

void teller(FILE* socket) {
  while (true) {
    char const* data = read_socket(socket); // untrusted

    Account account;
    parse_transit_number(data, &account.transit_number);
    parse_account_number(data, &account.account_number);
    auto authenticate = account.identification_method;
    if (authenticate(account, password)) {
      // we're in!
    }
    free(data);
  }
}
```

Temporal Memory Safety

- All memory accesses to an object should occur during the lifetime of the object's allocation
- Access to the object outside of this window is called a use-after-free

Example

```
void teller(FILE* socket) {
  std::vector<std::future<void>> transactions;
  while (true) {
    char const* data = read_socket(socket);
    transactions.push_back(std::async([=] {
      Account account;
      parse_transit_number(data, &account.transit_number);
      parse_account_number(data, &account.account_number);
      auto authenticate = account.identification_method;
      if (authenticate(account, password)) {
        // we're in!
      }
    }));
    free(data);
  }
}
```

Example

```
void teller(FILE* socket) {
  std::vector<std::future<void>> transactions;
  while (true) {
    char const* data = read_socket(socket);
    transactions.push_back(std::async([=] {
      Account account;
      parse_transit_number(data, &account.transit_number);
      parse_account_number(data, &account.account_number);
      auto authenticate = account.identification_method;
      if (authenticate(account, password)) {
        // we're in!
      }
    }));
    free(data);
  }
}
```

Example

```
void teller(FILE* socket) {
  std::vector<std::future<void>> transactions;
  while (true) {
    char const* data = read_socket(socket);
    transactions.push_back(std::async([=] {
      Account account;
      parse_transit_number(data, &account.transit_number);
      parse_account_number(data, &account.account_number);
      auto authenticate = account.identification_method;
      if (authenticate(account, password)) {
        // we're in!
      }
    }));
    free(data);
  }
}
```

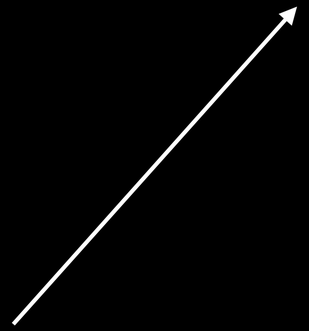
Example

```
void teller(FILE* socket) {
  std::vector<std::future<void>> transactions;
  while (true) {
    char const* data = read_socket(socket);
    transactions.push_back(std::async([=] {
      Account account;
      parse_transit_number(data, &account.transit_number);
      parse_account_number(data, &account.account_number);
      auto authenticate = account.identification_method;
      if (authenticate(account, password)) {
        // we're in!
      }
    }));
    free(data);
  }
}
```

Example

```
void teller(FILE* socket) {
  std::vector<std::future<void>> transactions;
  while (true) {
    char const* data = read_socket(socket);
    transactions.push_back(std::async([=] {
      Account account;
      parse_transit_number(data, &account.transit_number);
      parse_account_number(data, &account.account_number);
      auto authenticate = account.identification_method;
      if (authenticate(account, password)) {
        // we're in!
      }
    }));
    free(data);
  }
}
```

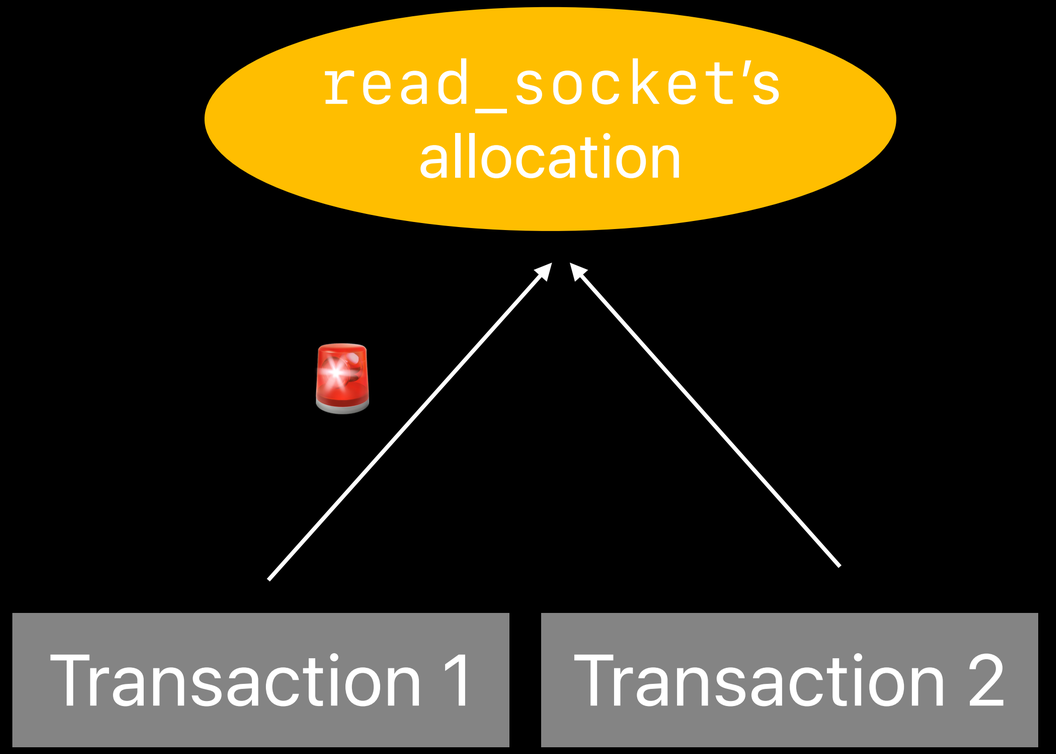
`read_socket`'s allocation



Transaction 1

Example

```
void teller(FILE* socket) {
  std::vector<std::future<void>> transactions;
  while (true) {
    char const* data = read_socket(socket);
    transactions.push_back(std::async([=] {
      Account account;
      parse_transit_number(data, &account.transit_number);
      parse_account_number(data, &account.account_number);
      auto authenticate = account.identification_method;
      if (authenticate(account, password)) {
        // we're in!
      }
    }));
    free(data);
  }
}
```

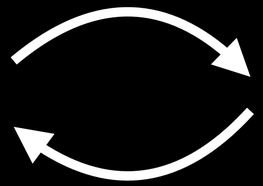


Type Safety

- A memory allocation is used to represent an object of a particular type
- Interpreting it as an object of a different type is called a type confusion

Most Temporal Memory Issues Involve “Type Confusion”

```
struct timespec {
  time_t tv_sec;
  long tv_nsec;
};
```



```
struct iovec {
  char* iov_base;
  size_t iov_len;
};
```

Guaranteed Initialization

- When memory is allocated, it contains garbage
- Using that undefined content can lead to information disclosure
- Can also be exploited if the attacker controls the “garbage”

Thread Safety

- Multi-threaded programs do concurrent accesses to memory
- If accesses are not properly synchronized, they can lead to data races
- Those can be composed (typically non-deterministically) with other bugs to create exploits
- Techniques exist to make non-deterministic exploits fairly reliable

What About C++?

It turns out that most safety issues are technically Undefined Behavior

Agenda

Overview of Memory Safety

Library Undefined Behavior

Standard Library Hardening

Typed Memory Operations

Conclusions

Library Undefined Behavior

UB is what happens when the Standard doesn’t guarantee anything

For the library, it is generally even stated explicitly:

24 Containers library **[containers]**

24.7 Views **[views]**

24.7.2 Contiguous access **[views.contiguous]**

24.7.2.2 Class template `span` **[views.span]**

24.7.2.2.6 Element access **[span.elem]**

```
constexpr reference operator[](size_type idx) const;
```

1 *Preconditions*: `idx < size()` is `true`.

2 *Effects*: Equivalent to: `return *(data() + idx);`

```
constexpr reference front() const;
```

3 *Preconditions*: `empty()` is `false`.

4 *Effects*: Equivalent to: `return *data();`

```
constexpr reference back() const;
```

5 *Preconditions*: `empty()` is `false`.

6 *Effects*: Equivalent to: `return *(data() + (size() - 1));`

```
constexpr pointer data() const noexcept;
```

7 *Effects*: Equivalent to: `return data_;`

From https://eel.is/c++draft/span.elem

Undefined Behavior Is a Specification Tool

- Creates a contract with the programmer
- Allows writing simpler APIs that make sense

  (No need to define results for meaningless inputs)
- Gives freedom to the implementation

Implementations Can Take Advantage of It

Valid strategies:

- Don’t do anything (execute code by assuming preconditions)
- Trap if a precondition is violated
- Log a message, keep executing
- … basically anything else you can think of

Agenda

Overview of Memory Safety

Library Undefined Behavior

Standard Library Hardening

Typed Memory Operations

Conclusions

Standard Library Hardening

- Turn select UB into guaranteed traps
- Provide *hardening* modes with high-level semantics
- Allow users to select hardening mode that’s right for them
- Allow vendors to select the default mode

This is not a “debugging” feature

You should ship that way!

Libc++ Hardening Modes

- **none** — do not compromise *any* performance
- **fast** — security-critical low-overhead checks only
- **extensive** — low-overhead checks
- **debug** — all checks

Inside the Library

- Checks are grouped in a few large categories
- Categories represent the nature of a check

Valid Element Access Checks

Checks that an attempt to access a container element is valid

`std::optional` is considered a container

Example:

```
template <class _Tp, class _Allocator>
const_reference vector<_Tp, _Allocator>::operator[](size_type __n) const noexcept {
  _LIBCPP_ASSERT_VALID_ELEMENT_ACCESS(__n < size(), "vector[] index out of bounds");
  return this->__begin_[__n];
}
```

Valid Element Access Checks

Checks that an attempt to access a container element is valid

`std::optional` is considered a container

Example:

```
template <class _Tp, class _Allocator>
const_reference vector<_Tp, _Allocator>::operator[](size_type __n) const noexcept {
  _LIBCPP_ASSERT_VALID_ELEMENT_ACCESS(__n < size(), "vector[] index out of bounds");
  return this->__begin_[__n];
}
```

Valid Input Range Checks

Checks that a range given as an input is valid:

- End iterator is reachable from the begin iterator
- Both iterators refer to the same container

Example:

```
template <class _CharT, class _Traits, class _Allocator>
iterator basic_string<_CharT, _Traits, _Allocator>::erase(const_iterator __first, const_iterator __last) {
  _LIBCPP_ASSERT_VALID_INPUT_RANGE(__first <= __last, "string::erase(first, last) called with invalid range");
  iterator __b = begin();
  size_type __r = static_cast<size_type>(__first - __b);
  erase(__r, static_cast<size_type>(__last - __first));
  return __b + static_cast<difference_type>(__r);
}
```

Valid Input Range Checks

Checks that a range given as an input is valid:

- End iterator is reachable from the begin iterator
- Both iterators refer to the same container

Example:

```
template <class _CharT, class _Traits, class _Allocator>
iterator basic_string<_CharT, _Traits, _Allocator>::erase(const_iterator __first, const_iterator __last) {
  _LIBCPP_ASSERT_VALID_INPUT_RANGE(__first <= __last, "string::erase(first, last) called with invalid range");
  iterator __b = begin();
  size_type __r = static_cast<size_type>(__first - __b);
  erase(__r, static_cast<size_type>(__last - __first));
  return __b + static_cast<difference_type>(__r);
}
```

Internal Checks

Internal preconditions that do not relate to user code

A failure here is a bug in libc++, not a bug in the user code

Example:

```
template <class _NodePtr>
_NodePtr __tree_min(_NodePtr __x) noexcept {
  _LIBCPP_ASSERT_INTERNAL(__x != nullptr, "Root node shouldn't be null");
  while (__x->__left_ != nullptr)
    __x = __x->__left_;
  return __x;
}
```

Internal Checks

Internal preconditions that do not relate to user code

A failure here is a bug in libc++, not a bug in the user code

Example:

```
template <class _NodePtr>
_NodePtr __tree_min(_NodePtr __x) noexcept {
  _LIBCPP_ASSERT_INTERNAL(__x != nullptr, "Root node shouldn't be null");
  while (__x->__left_ != nullptr)
    __x = __x->__left_;
  return __x;
}
```

High Level Modes are Collections of Categories

valid-input-range				
valid-element-access				
non-null-argument				
non-overlapping-ranges				
compatible-allocator				
valid-comparator				
internal				

High Level Modes are Collections of Categories

	None			
valid-input-range	❌			
valid-element-access	❌			
non-null-argument	❌			
non-overlapping-ranges	❌			
compatible-allocator	❌			
valid-comparator	❌			
internal	❌			

High Level Modes are Collections of Categories

	None	Fast		
valid-input-range	❌	✅		
valid-element-access	❌	✅		
non-null-argument	❌	❌		
non-overlapping-ranges	❌	❌		
compatible-allocator	❌	❌		
valid-comparator	❌	❌		
internal	❌	❌		

High Level Modes are Collections of Categories

	None	Fast	Extensive	
valid-input-range	❌	✅	✅	
valid-element-access	❌	✅	✅	
non-null-argument	❌	❌	✅	
non-overlapping-ranges	❌	❌	✅	
compatible-allocator	❌	❌	✅	
valid-comparator	❌	❌	❌	
internal	❌	❌	❌	

High Level Modes are Collections of Categories

	None	Fast	Extensive	Debug
valid-input-range	❌	✅	✅	✅
valid-element-access	❌	✅	✅	✅
non-null-argument	❌	❌	✅	✅
non-overlapping-ranges	❌	❌	✅	✅
compatible-allocator	❌	❌	✅	✅
valid-comparator	❌	❌	❌	✅
internal	❌	❌	❌	✅

Selecting the Hardening Mode

- Define this macro: `-D_LIBCPP_HARDENING_MODE=<mode>`
- Valid modes are:
  - `_LIBCPP_HARDENING_MODE_NONE`
  - `_LIBCPP_HARDENING_MODE_FAST`
  - `_LIBCPP_HARDENING_MODE_EXTENSIVE`
  - `_LIBCPP_HARDENING_MODE_DEBUG`
- Hardening mode can be selected in each TU

Usage Example

```
int main(int argc, char** argv) {
  std::vector<std::string> args(argv + 1, argv + argc);

  int total = 0;
  for (int i = 0; i != argc; ++i) {
    total += std::atoi(args[i].c_str());
  }
  std::cout << total << std::endl;
}
```

Usage Example

```
int main(int argc, char** argv) {
  std::vector<std::string> args(argv + 1, argv + argc);

  int total = 0;
  for (int i = 0; i != argc; ++i) {
    total += std::atoi(args[i].c_str());
  }
  std::cout << total << std::endl;
}
```

Usage Example

```
int main(int argc, char** argv) {
  std::vector<std::string> args(argv + 1, argv + argc);

  int total = 0;
  for (int i = 0; i != argc; ++i) {
    total += std::atoi(args[i].c_str());
  }
  std::cout << total << std::endl;
}
```

Usage Example

```
int main(int argc, char** argv) {
  std::vector<std::string> args(argv + 1, argv + argc);

  int total = 0;
  for (int i = 0; i != argc; ++i) {
    total += std::atoi(args[i].c_str());
  }
  std::cout << total << std::endl;
}
```

Usage Example

```
$ clang++ test.cpp -D_LIBCPP_HARDENING_MODE=_LIBCPP_HARDENING_MODE_DEBUG

$ ./a.out 1 2 3

[...]/usr/include/c++/v1/vector:1537: assertion __n < size() failed: vector[] index out of bounds

zsh: abort      ./a.out 1 2 3
```

Usage Example

```
$ clang++ test.cpp -D_LIBCPP_HARDENING_MODE=_LIBCPP_HARDENING_MODE_FAST

$ ./a.out 1 2 3

zsh: trace trap      ./a.out 1 2 3
```

Verbosity of Handler

Tradeoff between code size/optimizations and user experience

- fast ➡️ trap
- extensive ➡️ trap
- debug ➡️ abort verbosely

ABI Considerations

Some useful checks require changing the ABI:

```
std::span<int> span(ptr, 3);
auto b = span.begin();
b += 999;
int value = *b; // can we trap here?
```

ABI Selection Is Orthogonal to Hardening

- ABI is a property of the platform
- Vendors can select the desired ABI
- It doesn’t make sense for users to control that
- Huge simplification: this prevents having to deal with ABI-related concerns as part of hardening

Example: Bounded Iterators

Library is configured with `_LIBCPP_ABI_BOUNDED_ITERATORS` (by vendor)

```
template <class _Tp, size_t _Extent>
class span {
public:
  using element_type           = _Tp;
  using value_type             = remove_cv_t<_Tp>;
  using size_type              = size_t;
  // ...

#ifdef _LIBCPP_ABI_BOUNDED_ITERATORS
  using iterator               = __bounded_iter<pointer>;
#else
  using iterator               = pointer;
#endif
  // ...
};
```

Example: Bounded Iterators

Library is configured with `_LIBCPP_ABI_BOUNDED_ITERATORS` (by vendor)

```
template <class _Tp, size_t _Extent>
class span {
public:
  using element_type           = _Tp;
  using value_type             = remove_cv_t<_Tp>;
  using size_type              = size_t;
  // ...

#ifdef _LIBCPP_ABI_BOUNDED_ITERATORS
  using iterator               = __bounded_iter<pointer>;
#else
  using iterator               = pointer;
#endif
  // ...
};
```

Example: Bounded Iterators

Iterators now have enough information for bounds checking:

```
std::span<int> span(ptr, 3);
auto b = span.begin();
b += 999;
int value = *b; // trap!
```

If hardening mode is *none*, there is **still** no trap

Deployment Experience

Very positive experience so far:

- WebKit is using hardened libc++
- Several other Apple adopters
- Other adoption: Chrome and Google Cloud Network Virtualization *
- Lots of anecdotal evidence of bugs being found
- Already some known in-the-wild security issues that hardening would have prevented or alleviated

* https://bughunters.google.com/blog/6368559657254912/llvm-s-rfc-c-buffer-hardening-at-google

Deployment Experience

However, adoption can require some work:

- Adoption is easy for modern C++ code bases
- Harder for code bases that don’t use the Standard Library
- Adoption of any new feature can introduce bugs if not careful
- Non-zero performance cost

⚠️ Latent bugs may result in new crashes

A typical example

```
std::array<T, 30> array = {...};
std::sort(&array[0], &array[30]);
```

Diagnostics to Push Adoption

Clang's –Wunsafe–buffer–usage

```
int& f(int* p, int n) {
  return p[n];
}
```

```
$ clang++ foo.cpp –Wunsafe–buffer–usage
<stdin>:2:10: warning: unsafe buffer access [–Wunsafe–buffer–usage]
    2 |   return p[n];
      |          ^
1 warning generated.
```

Standardization Path

Enter Contracts

- Observe that we're only enforcing existing preconditions
- Contracts provide a framework for expressing them

```
template <class _Tp, class _Allocator>
const_reference vector<_Tp, _Allocator>::operator[](size_type __n) const noexcept
  pre [[clang::diag("vector[] index out of bounds")]] (__n < size())
{
  return this->__begin_[__n];
}
```

P2900 Contracts MVP

P2900 is based on different *contract semantics*:

	verbose	failures trap
ignore	❌	❌
observe	✅	❌
since Tokyo P3191	❌	✅
enforce	✅	✅

Why We Want Contracts In C++26

- Allows formally expressing library hardening
- Allows formalizing things like sanitizers
  - After all they just check various types of language UB
- Generally useful feature for users to express assertions

Why We Want Contracts In C++26

A “safe” mode for ISO C++ that safety and security critical industries can build on top of

Agenda

Overview of Memory Safety

Library Undefined Behavior

Standard Library Hardening

Typed Memory Operations

Conclusions

A Clever Observation

Most temporal memory safety exploits require some type confusion

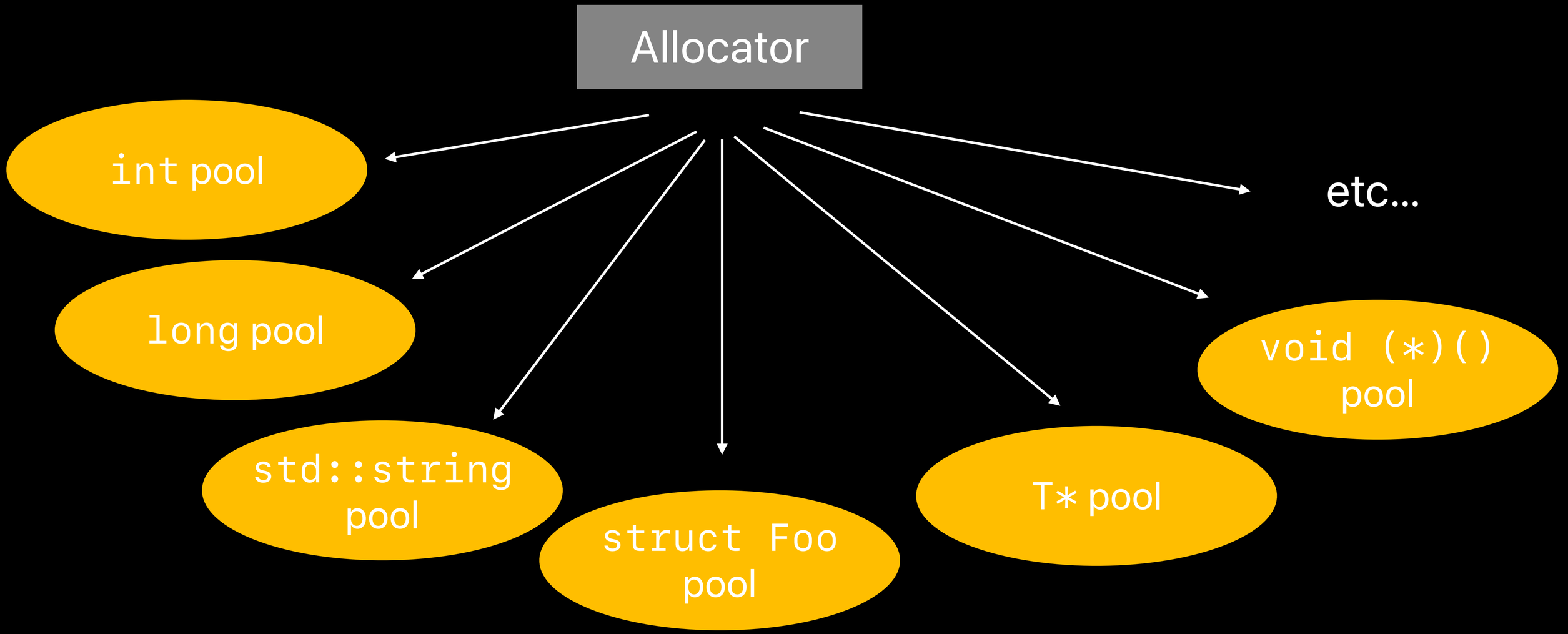
If memory is never reused for a different type, confusions are impossible

➡️ Segregate allocations by type!

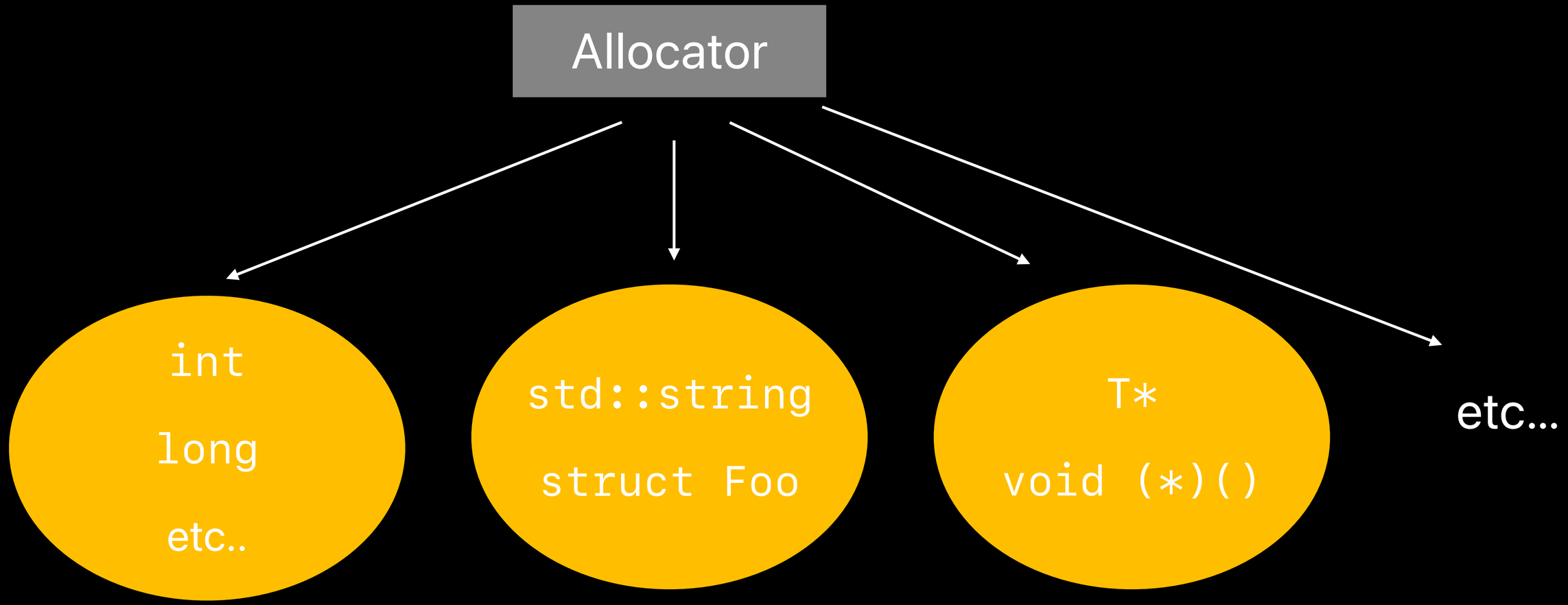
This was introduced in the Darwin Kernel a few years ago *

* https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety

A Naive Typed Memory Allocator



A Performance / Security Tradeoff



The Most Important Property

Data must not alias pointers

Making Exploits Harder

Randomize buckets on boot!

How Effective Is Type Segregation?

In the Darwin Kernel, data suggests that the vast majority of dynamic allocation lifetime bugs are not exploitable anymore

Type Segregation For General C++

```
struct Foo {
  // ...
};

std::unique_ptr<Foo> f() {
  return new Foo{args...}; // GOAL: should come from the Foo pool
}
```

The Usual `operator new` Rewriting

User writes

```
std::unique_ptr<Foo> f() {
  return new Foo{args...};
}
```

Compiler rewrites

```
std::unique_ptr<Foo> f() {
  Foo* __alloc = operator new(sizeof(Foo));

  new (__alloc) Foo{args...};
  return __alloc;
}
```

The Problem

There is no type information

```
void* operator new(std::size_t);
void* operator new(std::size_t, const std::nothrow_t&) noexcept;
void* operator new(std::size_t, std::align_val_t);
void* operator new(std::size_t, std::align_val_t, const std::nothrow_t&) noexcept;
```

Thankfully, We Can Modify the Standard Library!

Let’s add type information

```
enum class __type_descriptor_t : unsigned long long;

void* operator new(std::size_t, std::__type_descriptor_t);
void* operator new(std::size_t, const std::nothrow_t&, std::__type_descriptor_t) noexcept;
```

And the Compiler Too!

User writes

```
std::unique_ptr<Foo> f() {
  return new Foo{args...};
}
```

Compiler rewrites

```
std::unique_ptr<Foo> f() {
  Foo* __alloc = operator new(sizeof(Foo));

  new (__alloc) Foo{args...};
  return __alloc;
}
```

And the Compiler Too!

User writes

```
std::unique_ptr<Foo> f() {
  return new Foo{args...};
}
```

Compiler rewrites

```
std::unique_ptr<Foo> f() {
  Foo* __alloc = operator new(sizeof(Foo),
                              __builtin_type_descriptor(Foo));
  new (__alloc) Foo{args...};
  return __alloc;
}
```

Then We Forward Type Information to the System Allocator

```
void* operator new(std::size_t size, std::__type_descriptor_t desc) {
  if (size == 0)
    size = 1;

  void* p;
  while ((p = malloc_type_malloc(size, static_cast<malloc_type_id_t>(desc))) == nullptr) {
    // ...
  }
  if (p == nullptr)
    throw std::bad_alloc();
  return p;
}
```

What About User-Defined `operator new`?

Isn’t That an ABI Break?

It Would Be!

It Would Be!

But we’re clever

It Would Be!

But we’re clever

```
void* operator new(std::size_t size, std::__type_descriptor_t desc) {
  if (std::__is_function_overridden(static_cast<void* (*)(std::size_t)>(&operator new))) {
    return static_cast<void* (*)(std::size_t)>(&operator new)(size);
  } else {
    // ... the typed implementation ...
  }
}
```

The Worst Code I've Written?

```
__attribute__((__section__("__TEXT,__lcxx_override")))
void* operator new(size_t __size) noexcept {
  // ...
}
```

```
template <class _Ret, class... _Args>
bool __is_function_overridden(_Ret (*__fptr)(_Args...)) noexcept {
  extern char __segstart __asm("section$start$__TEXT$__lcxx_override");
  extern char __segend __asm("section$end$__TEXT$__lcxx_override");

  uintptr_t __start = (uintptr_t)(&__segstart);
  uintptr_t __end   = (uintptr_t)(&__segend);
  uintptr_t __ptr   = (uintptr_t)(__fptr);

  return __ptr < __start || __ptr > __end;
}
```

The Worst Code I've Written?

```
__attribute__((__section__("__TEXT,__lcxx_override")))
void* operator new(size_t __size) noexcept {
  // ...
}
```

```
template <class _Ret, class... _Args>
bool __is_function_overridden(_Ret (*__fptr)(_Args...)) noexcept {
  extern char __segstart __asm("section$start$__TEXT$__lcxx_override");
  extern char __segend __asm("section$end$__TEXT$__lcxx_override");

  uintptr_t __start = (uintptr_t)(&__segstart);
  uintptr_t __end   = (uintptr_t)(&__segend);
  uintptr_t __ptr   = (uintptr_t)(__fptr);

  return __ptr < __start || __ptr > __end;
}
```

The Worst Code I've Written?

```
__attribute__((__section__("__TEXT,__lcxx_override")))
void* operator new(size_t __size) noexcept {
  // ...
}
```

```
template <class _Ret, class... _Args>
bool __is_function_overridden(_Ret (*__fptr)(_Args...)) noexcept {
  extern char __segstart __asm("section$start$__TEXT$__lcxx_override");
  extern char __segend __asm("section$end$__TEXT$__lcxx_override");

  uintptr_t __start = (uintptr_t)(&__segstart);
  uintptr_t __end   = (uintptr_t)(&__segend);
  uintptr_t __ptr   = (uintptr_t)(__fptr);

  return __ptr < __start || __ptr > __end;
}
```

The Worst Code I've Written?

```
__attribute__((__section__("__TEXT,__lcxx_override")))
void* operator new(size_t __size) noexcept {
  // ...
}
```

```
template <class _Ret, class... _Args>
bool __is_function_overridden(_Ret (*__fptr)(_Args...)) noexcept {
  extern char __segstart __asm("section$start$__TEXT$__lcxx_override");
  extern char __segend __asm("section$end$__TEXT$__lcxx_override");

  uintptr_t __start = (uintptr_t)(&__segstart);
  uintptr_t __end   = (uintptr_t)(&__segend);
  uintptr_t __ptr   = (uintptr_t)(__fptr);

  return __ptr < __start || __ptr > __end;
}
```

The Worst Code I've Written?

```
__attribute__((__section__("__TEXT,__lcxx_override")))
void* operator new(size_t __size) noexcept {
  // ...
}
```

```
template <class _Ret, class... _Args>
bool __is_function_overridden(_Ret (*__fptr)(_Args...)) noexcept {
  extern char __segstart __asm("section$start$__TEXT$__lcxx_override");
  extern char __segend __asm("section$end$__TEXT$__lcxx_override");

  uintptr_t __start = (uintptr_t)(&__segstart);
  uintptr_t __end   = (uintptr_t)(&__segend);
  uintptr_t __ptr   = (uintptr_t)(__fptr);

  return __ptr < __start || __ptr > __end;
}
```

Deployment Experience

- TMO adopted in the Darwin Kernel (XNU)
- Typed `operator new` adopted in Darwin user space system libraries
- Extremely effective
- Essentially no adoption cost

Deployment Experience

- Not a silver bullet (not all allocations are funnelled through `new`)
- Effectiveness relies on QOI of the system allocator, which is a performance tradeoff

Standardization Path

Upcoming Paper

P2719: Type-aware allocation and deallocation functions

Before

```
// user writes:
new (args...) T(...)


// compiler checks (in order):
T::operator new(sizeof(T), args...)
::operator new(sizeof(T), args...)
```

After

Upcoming Paper

P2719: Type-aware allocation and deallocation functions

Before

```
// user writes:
new (args...) T(...)


// compiler checks (in order):
T::operator new(sizeof(T), args...)
::operator new(sizeof(T), args...)
```

After

```
// user writes:
new (args...) T(...)


// compiler checks (in order):
T::operator new(sizeof(T), args...)
operator new(sizeof(T), type_identity<T>{}, args...)
::operator new(sizeof(T), args...)
```

Users Could Now Write

```
struct Druid : Character { };
struct Paladin : Character { };
struct Sorceress : Character { };

template <std::derived_from<Character> T>
void* operator new(std::size_t size, std::type_identity<T>) {
  // ... some special allocation scheme for these types ...
}
```

A Conforming Extension Under the As-If Rule

```
template <class _Tp>
  __attribute__((__overload_priority__(-1)))
void* operator new(std::size_t __size, std::type_identity<_Tp>) {
  std::__type_descriptor_t __descriptor = __builtin_type_descriptor(_Tp);
  // ... typed operator new implementation ...
}
```

Conclusions

- Standard Library hardening tackles (mostly) spatial memory safety
  - Requires adoption to be effective
  - Great for bug finding and production “hardening”
  - Contracts provide a great framework for it
  - Go try it out!

Conclusions

- TMO makes temporal memory issues harder to exploit
  - Adoption is almost 100% unintrusive
  - Does not fix any actual bugs, but makes them difficult to exploit
  - We propose a standardization path with other benefits

Conclusions

- There's **a huge amount** of existing C++ code
  - A lot of it is unsafe by everyone's standard
  - We need to do something about that
  - Ease of adoption is a necessity

Conclusions

- **Better** safety and security is achievable in C++
- We must look for simple and high impact changes, not perfection
- We encourage more WG21 work on immediate solutions
- Pragmatically consider the greater good, not only C++'s interests

Thank You!