Security in C++ Hardening techniques from the trenches

Louis Dionne and Konstantin Varlamov

15-11-2024

How Important Is Security?

- Financial threat: *WannaCry* (ransomware) affected over 300k computers in 150 countries, cost over $4B
- Infrastructure threat: *Stuxnet* and *Triton* targeted power stations
- Physical threat: Spyware like *Pegasus* targeted journalists and high-profile activists

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

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

C++ Is Memory Unsafe

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++**.

[...]

The highest leverage method to reduce memory safety vulnerabilities is to secure one of the building blocks of cyberspace: the programming language.*

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

There are memory safe alternatives to C++

But migrating is not always an option

C++ Can Do Better

C++ 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

The Mindset Is Changing

- More general awareness about the problem
- Creation of SG23 (Safety and Security Study Group)
- Most of WG21 understands the urgency
- However, still few concrete solutions

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

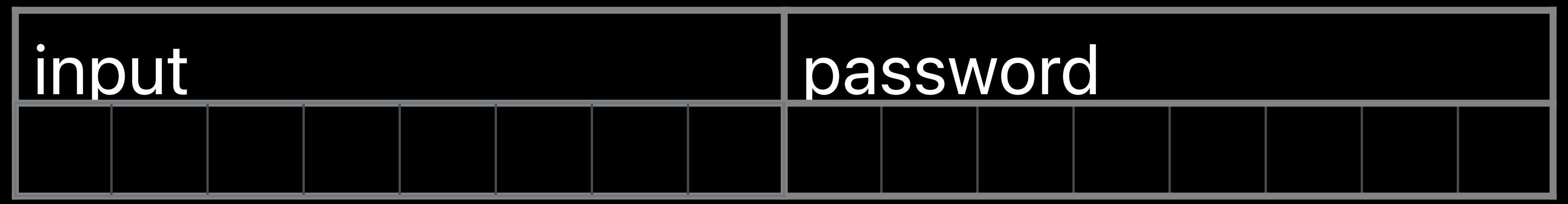
Spatial memory safety

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

Example

```
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
```



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
```

input								password							
								r	e	a	l	p	a	s	s

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";
}
```

```
$ authenticate

> securitysecurity
```

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

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

🚨

```
$ 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";
}
```

```
$ 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/

Temporal Memory Safety

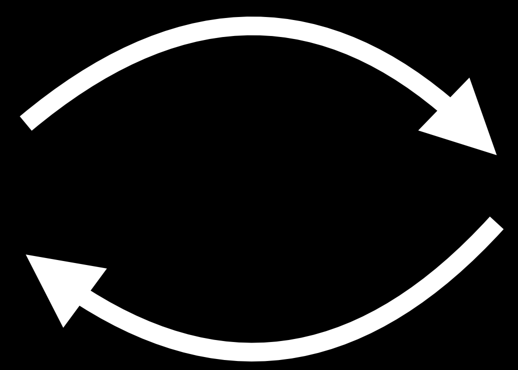
- All memory accesses to an object should occur during the lifetime of the object's allocation
- Access to the object outside of this window is called a use-after-free

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"

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



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

Tying this back to ISO C++

Most safety issues fall under Undefined Behavior in the Standard

Agenda

Overview of Memory Safety

Library Undefined Behavior

Standard Library Hardening

Typed Memory Operations

Conclusions

Undefined Behavior

In the Standard, all memory safety bugs fall under undefined behavior.

Undefined Behavior

In the Standard, all memory safety bugs fall under undefined behavior.

3.63 **undefined behavior** **[defns.undefined]**

behavior for which this document imposes no requirements

[*Note 1*: Undefined behavior may be expected when this document omits any explicit definition of behavior or when a program uses an incorrect construct or invalid data. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a *diagnostic message* ([defns.diagnostic])), to terminating a translation or execution (with the issuance of a diagnostic message). Many incorrect program constructs do not engender undefined behavior; they are required to be diagnosed. Evaluation of a constant expression ([expr.const]) never exhibits behavior explicitly specified as undefined in [intro] through [cpp]. — *end note*]

Undefined behavior != undefined behavior

Undefined behavior != undefined behavior

- Language-level UB

Undefined behavior != undefined behavior

- Language-level UB

  The *compiler* is free to do anything

Undefined behavior != undefined behavior

- Language-level UB

  The *compiler* is free to do anything

- Library-level UB

Undefined behavior != undefined behavior

- Language-level UB

  The *compiler* is free to do anything

- Library-level UB

  The *standard library* is free to do anything

Library Undefined Behavior

Library Undefined Behavior

- Usually explicitly stated

Library Undefined Behavior

- Usually explicitly stated

23.7.2.2.6 Element access

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

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

2 *Returns*: `*(data() + idx)`.

3 *Throws*: Nothing.

Library Undefined Behavior

- Usually explicitly stated
- Bounded

23.7.2.2.6 Element access

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

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

2 *Returns*: `*(data() + idx)`.

3 *Throws*: Nothing.

Library Undefined Behavior

- Usually explicitly stated
- Bounded
- Always due to user input

23.7.2.2.6 Element access

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

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

2 *Returns*: `*(data() + idx)`.

3 *Throws*: Nothing.

Library Undefined Behavior

- Usually explicitly stated
- Bounded
- Always due to user input
- **Easier** to check

23.7.2.2.6 Element access

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

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

2 *Returns*: `*(data() + idx)`.

3 *Throws*: Nothing.

Library Undefined Behavior

- Usually explicitly stated
- Bounded
- Always due to user input
- **Easier** to check

  (compared to language)

23.7.2.2.6 Element access

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

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

2 *Returns*: `*(data() + idx)`.

3 *Throws*: Nothing.

Two axes for classifying undefined behavior

Two axes for classifying undefined behavior

- Severity: from benign to security-critical

Two axes for classifying undefined behavior

- Severity: from benign to security-critical
- Difficulty of validating: from trivial to impossible

Two axes for classifying undefined behavior

- Severity: from benign to security-critical
- Difficulty of validating: from trivial to impossible

3 For all algorithms that take `Compare`, there is a version that uses `operator<` instead. That is, `comp(*i, *j) != false` defaults to `*i < *j != false`. For algorithms other than those described in [alg.binary.search], `comp` shall induce a strict weak ordering on the values.

4 The term *strict* refers to the requirement of an irreflexive relation (`!comp(x, x)` for all `x`), and the term *weak* to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define `equiv(a, b)` as `!comp(a, b) && !comp(b, a)`, then the requirements are that `comp` and `equiv` both be transitive relations:

Two axes for classifying undefined behavior

- Severity: from benign to **security-critical**
- Difficulty of validating: from **trivial** to impossible

Defining undefined behavior

UB means the implementation can do *anything*.

"Anything" doesn't have to be harmful or useless!

We can turn UB into useful implementation-defined behavior.

3.63 **undefined behavior** **[defns.undefined]**

behavior for which this document imposes no requirements

[*Note 1*: Undefined behavior may be expected when this document omits any explicit definition of behavior or when a program uses an incorrect construct or invalid data. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a *diagnostic message* ([defns.diagnostic])), to terminating a translation or execution (with the issuance of a diagnostic message). Many incorrect program constructs do not engender undefined behavior; they are required to be diagnosed. Evaluation of a constant expression ([expr.const]) never exhibits behavior explicitly specified as undefined in [intro] through [cpp]. — *end note*]

Agenda

Overview of Memory Safety

Library Undefined Behavior

Standard Library Hardening

Typed Memory Operations

Conclusions

Standard Library Hardening

Standard Library Hardening

- Turn select UB into guaranteed traps

Standard Library Hardening

- Turn select UB into guaranteed traps
- Provide hardening *modes* with high-level semantics

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

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

Libc++ Hardening Modes

Libc++ Hardening Modes

- **none** — do not compromise *any* performance

Libc++ Hardening Modes

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

Libc++ Hardening Modes

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

Libc++ Hardening Modes

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

Usage example

```
std::string get(int index) {
  std::vector<std::string> data = {"foo", "bar", "baz"};

  if (index < std::ssize(data))
    return data[index];
  return "<not found>";
}

int main(int argc, char** argv) {
  if (argc != 2) return -1;
  int index = std::stoi(argv[1]);

  std::cout << get(index) << '\n';
}
```

Usage example

```
std::string get(int index) {
  std::vector<std::string> data = {"foo", "bar", "baz"};

  if (index < std::ssize(data))
    return data[index];
  return "<not found>";
}

int main(int argc, char** argv) {
  if (argc != 2) return -1;
  int index = std::stoi(argv[1]);

  std::cout << get(index) << '\n';
}
```

Usage example

```cpp
std::string get(int index) {
  std::vector<std::string> data = {"foo", "bar", "baz"};

  if (index < std::ssize(data))
    return data[index];
  return "<not found>";
}

int main(int argc, char** argv) {
  if (argc != 2) return -1;
  int index = std::stoi(argv[1]);

  std::cout << get(index) << '\n';
}
```

Usage example

```
std::string get(int index) {
  std::vector<std::string> data = {"foo", "bar", "baz"};

  if (index < std::ssize(data))
    return data[index];
  return "<not found>";
}

int main(int argc, char** argv) {
  if (argc != 2) return -1;
  int index = std::stoi(argv[1]);

  std::cout << get(index) << '\n';
}
```

Usage example

```
std::string get(int index) {
  std::vector<std::string> data = {"foo", "bar", "baz"};

  if (index < std::ssize(data))
    return data[index];
  return "<not found>";
}

int main(int argc, char** argv) {
  if (argc != 2) return -1;
  int index = std::stoi(argv[1]);

  std::cout << get(index) << '\n';
}
```

Usage example

```
std::string get(int index) {
  std::vector<std::string> data = {"foo", "bar", "baz"};

  if (index < std::ssize(data))
    return data[index];
  return "<not found>";
}

int main(int argc, char** argv) {
  if (argc != 2) return -1;
  int index = std::stoi(argv[1]);

  std::cout << get(index) << '\n';
}
```

Usage example

```
std::string get(int index) {
  std::vector<std::string> data = {"foo", "bar", "baz"};

  if (index < std::ssize(data))
    return data[index];
  return "<not found>";
}

int main(int argc, char** argv) {
  if (argc != 2) return -1;
  int index = std::stoi(argv[1]);

  std::cout << get(index) << '\n';
}
```

Usage example

```
std::string get(int index) {
  std::vector<std::string> data = {"foo", "bar", "baz"};

  if (index < std::ssize(data))
    return data[index];
  return "<not found>";
}

int main(int argc, char** argv) {
  if (argc != 2) return -1;
  int index = std::stoi(argv[1]);

  std::cout << get(index) << '\n';
}
```

Usage example

```cpp
std::string get(int index) {
  std::vector<std::string> data = {"foo", "bar", "baz"};

  if (index < std::ssize(data))
    return data[index];
  return "<not found>";
}

int main(int argc, char** argv) {
  if (argc != 2) return -1;
  int index = std::stoi(argv[1]);

  std::cout << get(index) << '\n';
}
```

Usage example

Usage example

```
$ clang++ -std=c++23 -g main.cc && ./a.out 1
bar
```

Usage example

```
$ clang++ -std=c++23 -g main.cc && ./a.out 1
bar

$ clang++ -std=c++23 -g main.cc && ./a.out -1
```

Usage example

```
$ clang++ -std=c++23 -g main.cc && ./a.out 1
bar

$ clang++ -std=c++23 -g main.cc && ./a.out -1


$ clang++ -std=c++23 -g main.cc \
           -D_LIBCPP_HARDENING_MODE=_LIBCPP_HARDENING_MODE_FAST \
         && ./a.out -1
[1]    16295 trace trap  ./a.out -1
```

Usage example: attaching a debugger

Usage example: attaching a debugger

```
$ lldb a.out
(lldb) target create "a.out"
Current executable set to '/Users/varconst/demo/a.out' (arm64).
```

Usage example: attaching a debugger

```
$ lldb a.out
(lldb) target create "a.out"
Current executable set to '/Users/varconst/demo/a.out' (arm64).

(lldb) run -1
Process 16434 launched: '/Users/varconst/demo/a.out' (arm64)
Process 16434 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = Runtime Error:
/Library/Developer/CommandLineTools/SDKs/MacOSX15.2.sdk/usr/include/c++/v1/
vector:1394: assertion __n < size() failed: vector[] index out of bounds
```

Usage example: attaching a debugger

```
$ lldb a.out
(lldb) target create "a.out"
Current executable set to '/Users/varconst/demo/a.out' (arm64).

(lldb) run -1
Process 16434 launched: '/Users/varconst/demo/a.out' (arm64)
Process 16434 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = Runtime Error:
/Library/Developer/CommandLineTools/SDKs/MacOSX15.2.sdk/usr/include/c++/v1/
vector:1394: assertion __n < size() failed: vector[] index out of bounds

    frame #1: 0x0000000100000808 a.out`std::__1::vector<std::__1::basic_str
ing<char, std::__1::char_traits<char>, std::__1::allocator<char>>, std::__1
::allocator<std::__1::basic_string<char, std::__1::char_traits<char>, std::
__1::allocator<char>>>>::operator[](this=0x000000016fdfeb38 size=3, __n=184
46744073709551615) at vector:1393:3
   1390 template <class _Tp, class _Allocator>
   1391 constexpr inline typename vector<_Tp, _Allocator>::reference
   1392 vector<_Tp, _Allocator>::operator[](size_type __n) noexcept {
-> 1393   _LIBCPP_ASSERT_VALID_ELEMENT_ACCESS(__n < size(),
   1394         "vector[] index out of bounds");
   1395   return this->__begin_[__n];
   1396 }
Target 0: (a.out) stopped.
(lldb)
```

This is not a “debugging” feature

You should ship this way!

There is no "one-size-fits-all" approach

There is no "one-size-fits-all" approach

- Different projects make different tradeoffs between safety and performance
  - This can be true even for separate parts of the same project

There is no "one-size-fits-all" approach

- Different projects make different tradeoffs between safety and performance
  - This can be true even for separate parts of the same project
- 80/20 principle: "80"% of CVEs are caused by "20"% of *types* of issues (memory safety)
  - We can focus on the few most critical checks

There is no "one-size-fits-all" approach

- Different projects make different tradeoffs between safety and performance
  - This can be true even for separate parts of the same project
- 80/20 principle: "80"% of CVEs are caused by "20"% of *types* of issues (memory safety)
  - We can focus on the few most critical checks
- Wide adoption is critical, more important than perfect coverage
  - Better have 80% of programs catching 20% of issues than vice versa

Inside the Library

Inside the Library

- Checks are grouped into a few large categories

Inside the Library

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

Inside the Library

- Checks are grouped into a few large categories
- Categories represent the nature of a check
  - Valid element access

Inside the Library

- Checks are grouped into a few large categories
- Categories represent the nature of a check
    - Valid element access
    - Valid input range

Inside the Library

- Checks are grouped into a few large categories
- Categories represent the nature of a check
  - Valid element access
  - Valid input range
  - Non-null pointer

Inside the Library

- Checks are grouped into a few large categories
- Categories represent the nature of a check
  - Valid element access
  - Valid input range
  - Non-null pointer
  - And so on...

Inside the Library

- Checks are grouped into a few large categories
- Categories represent the nature of a check
  - Valid element access
  - Valid input range
  - Non-null pointer
  - And so on...
- Categories are internal – users only see modes

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

Non-overlapping Ranges Checks

Checks that two ranges given to an algorithm do not overlap

Example:

```
static inline constexpr char_type*
copy(char_type* __s1, const char_type* __s2, size_t __n) noexcept {
  _LIBCPP_ASSERT_NON_OVERLAPPING_RANGES(
      !std::__is_pointer_in_range(__s1, __s1 + __n, __s2),
      "char_traits::copy: source and destination ranges overlap");
  std::memmove(__s1, __s2, __element_count(__n));
  return __s1;
}
```

Non-overlapping Ranges Checks

Checks that two ranges given to an algorithm do not overlap

Example:

```
static inline constexpr char_type*
copy(char_type* __s1, const char_type* __s2, size_t __n) noexcept {
  _LIBCPP_ASSERT_NON_OVERLAPPING_RANGES(
      !std::__is_pointer_in_range(__s1, __s1 + __n, __s2),
      "char_traits::copy: source and destination ranges overlap");
  std::memmove(__s1, __s2, __element_count(__n));
  return __s1;
}
```

High Level Modes are Collections of Categories

valid-input-range				
valid-element-access				
non-null-argument				
non-overlapping-ranges				
valid-deallocation				
semantic-requirement				
internal				

High Level Modes are Collections of Categories

	None			
valid-input-range	✗			
valid-element-access	✗			
non-null-argument	✗			
non-overlapping-ranges	✗			
valid-deallocation	✗			
semantic-requirement	✗			
internal	✗			

High Level Modes are Collections of Categories

	None	Fast		
valid-input-range	❌	✅		
valid-element-access	❌	✅		
non-null-argument	❌	❌		
non-overlapping-ranges	❌	❌		
valid-deallocation	❌	❌		
semantic-requirement	❌	❌		
internal	❌	❌		

High Level Modes are Collections of Categories

	None	Fast	Extensive
valid-input-range	❌	✅	✅
valid-element-access	❌	✅	✅
non-null-argument	❌	❌	✅
non-overlapping-ranges	❌	❌	✅
valid-deallocation	❌	❌	✅
semantic-requirement	❌	❌	❌
internal	❌	❌	❌

High Level Modes are Collections of Categories

	None	Fast	Extensive	Debug
valid-input-range	❌	✅	✅	✅
valid-element-access	❌	✅	✅	✅
non-null-argument	❌	❌	✅	✅
non-overlapping-ranges	❌	❌	✅	✅
valid-deallocation	❌	❌	✅	✅
semantic-requirement	❌	❌	❌	✅
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

Failed checks lead to termination

- The program reliably terminates in all modes
- Production and debug modes use different termination methods

  Tradeoff between performance 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
- 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 the 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 the 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

Sometimes, an ABI change is not necessary

Inside a `unique_ptr<T[]>`, we can get the size from the array cookie

```
template <class _Tp, class _Dp>
class unique_ptr<_Tp[], _Dp> {
  // ...
  template <class _Deleter,
            class _Tp,
            __enable_if_t<__is_default_deleter<_Deleter>::value &&
            __has_array_cookie<_Tp>::value, int> = 0>
  constexpr bool __in_bounds(_Tp* __ptr, size_t __index) const {
    size_t __cookie = std::__get_array_cookie(__ptr);
    return __index < __cookie;
  }
```

Sometimes, an ABI change is not necessary

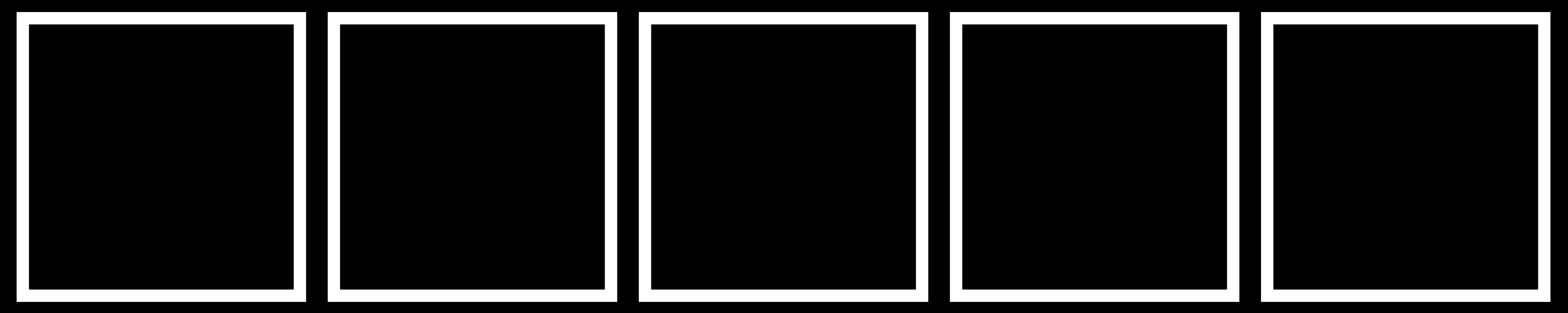
Inside a `unique_ptr<T[]>`, we can get the size from the array cookie

```
template <class _Tp, class _Dp>
class unique_ptr<_Tp[], _Dp> {
  // ...
  template <class _Deleter,
            class _Tp,
            __enable_if_t<__is_default_deleter<_Deleter>::value &&
            __has_array_cookie<_Tp>::value, int> = 0>
  constexpr bool __in_bounds(_Tp* __ptr, size_t __index) const {
    size_t __cookie = std::__get_array_cookie(__ptr);
    return __index < __cookie;
  }
```

Sometimes, an ABI change is not necessary

Inside a `unique_ptr<T[]>`, we can get the size from the array cookie

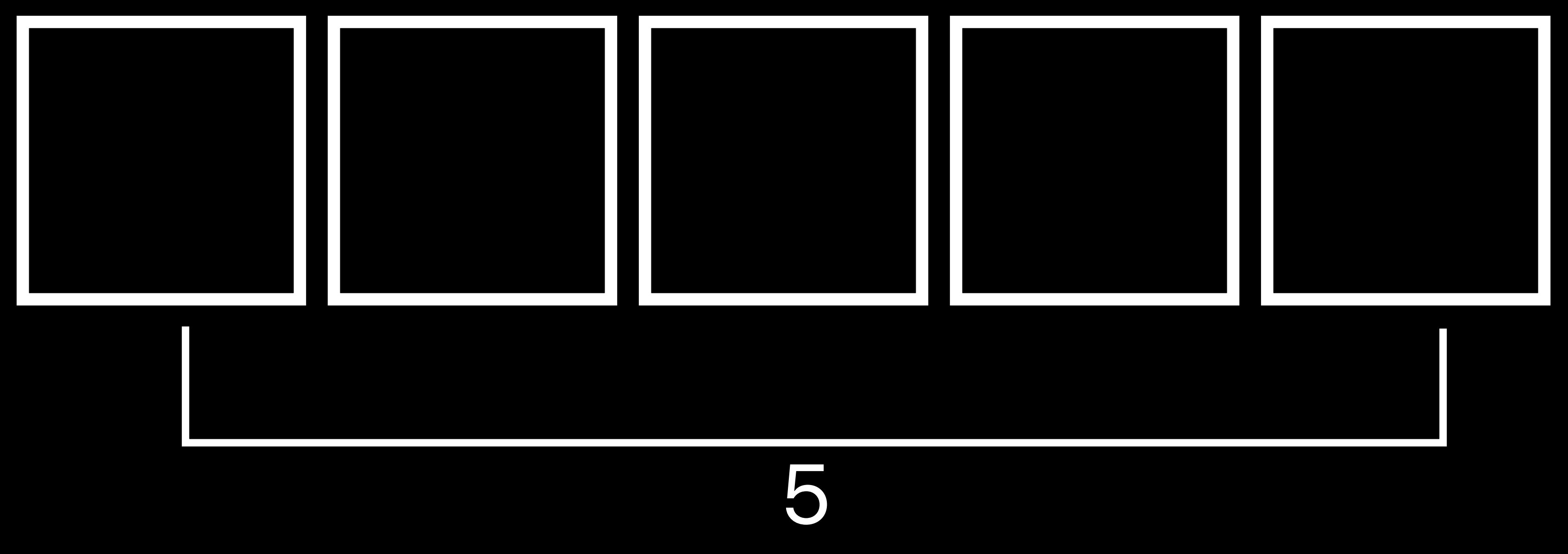
```
template <class _Tp, class _Dp>
class unique_ptr<_Tp[], _Dp> {
  // ...
  template <class _Deleter,
            class _Tp,
            __enable_if_t<__is_default_deleter<_Deleter>::value &&
            __has_array_cookie<_Tp>::value, int> = 0>
  constexpr bool __in_bounds(_Tp* __ptr, size_t __index) const {
    size_t __cookie = std::__get_array_cookie(__ptr);
    return __index < __cookie;
  }
```



Sometimes, an ABI change is not necessary

Inside a `unique_ptr<T[]>`, we can get the size from the array cookie

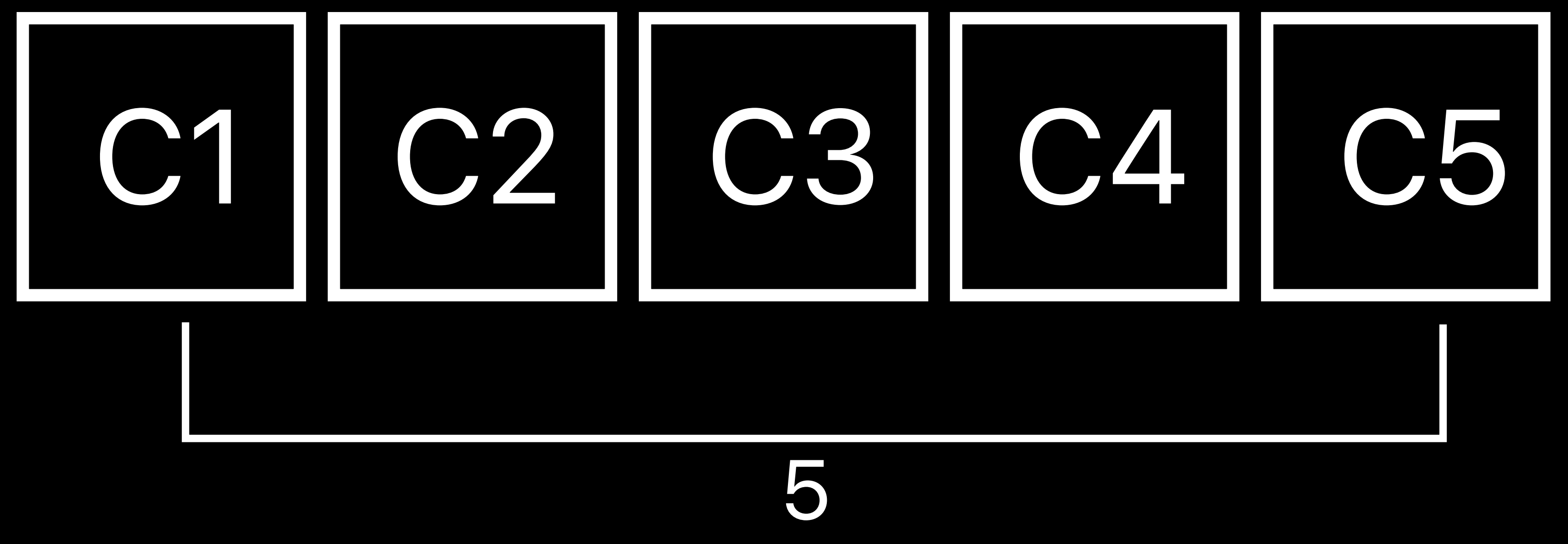
```
template <class _Tp, class _Dp>
class unique_ptr<_Tp[], _Dp> {
  // ...
  template <class _Deleter,
            class _Tp,
            __enable_if_t<__is_default_deleter<_Deleter>::value &&
            __has_array_cookie<_Tp>::value, int> = 0>
  constexpr bool __in_bounds(_Tp* __ptr, size_t __index) const {
    size_t __cookie = std::__get_array_cookie(__ptr);
    return __index < __cookie;
  }
```



Sometimes, an ABI change is not necessary

Inside a `unique_ptr<T[]>`, we can get the size from the array cookie

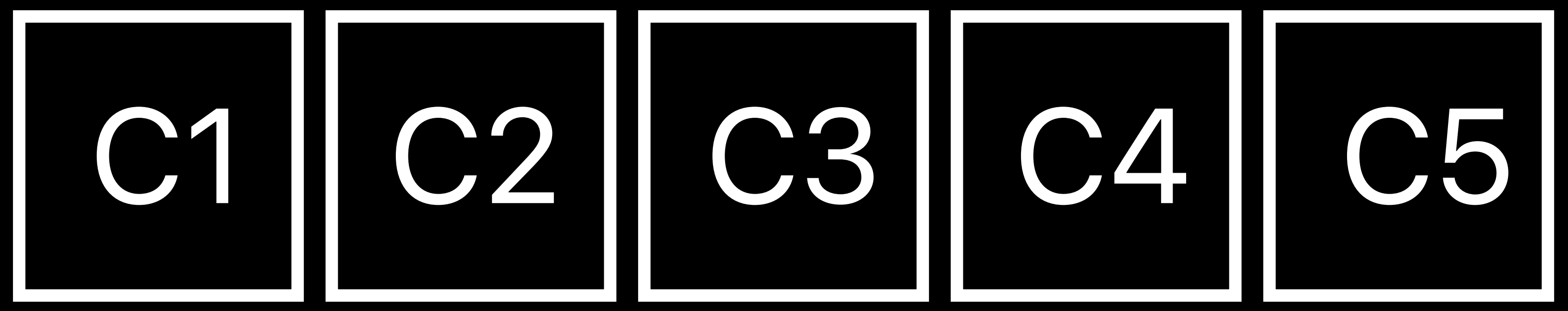
```
template <class _Tp, class _Dp>
class unique_ptr<_Tp[], _Dp> {
  // ...
  template <class _Deleter,
            class _Tp,
            __enable_if_t<__is_default_deleter<_Deleter>::value &&
            __has_array_cookie<_Tp>::value, int> = 0>
  constexpr bool __in_bounds(_Tp* __ptr, size_t __index) const {
    size_t __cookie = std::__get_array_cookie(__ptr);
    return __index < __cookie;
  }
```



Sometimes, an ABI change is not necessary

Inside a unique_ptr<T[ ]>, we can get the size from the array cookie

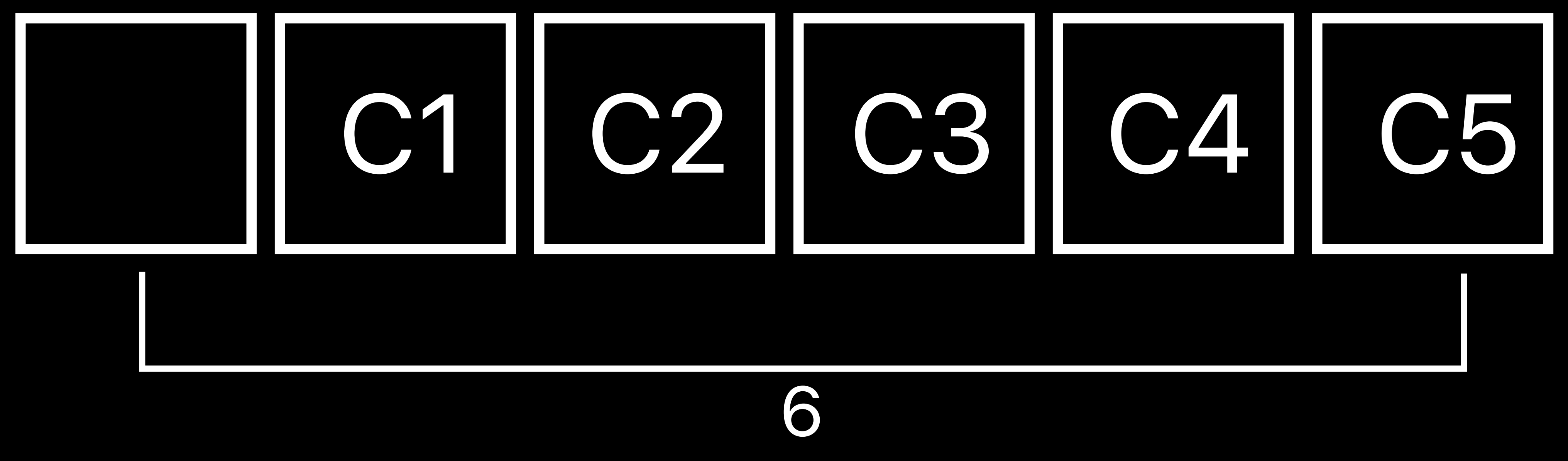
```
template <class _Tp, class _Dp>
class unique_ptr<_Tp[], _Dp> {
  // ...
  template <class _Deleter,
            class _Tp,
            __enable_if_t<__is_default_deleter<_Deleter>::value &&
            __has_array_cookie<_Tp>::value, int> = 0>
  constexpr bool __in_bounds(_Tp* __ptr, size_t __index) const {
    size_t __cookie = std::__get_array_cookie(__ptr);
    return __index < __cookie;
  }
```



Sometimes, an ABI change is not necessary

Inside a `unique_ptr<T[]>`, we can get the size from the array cookie

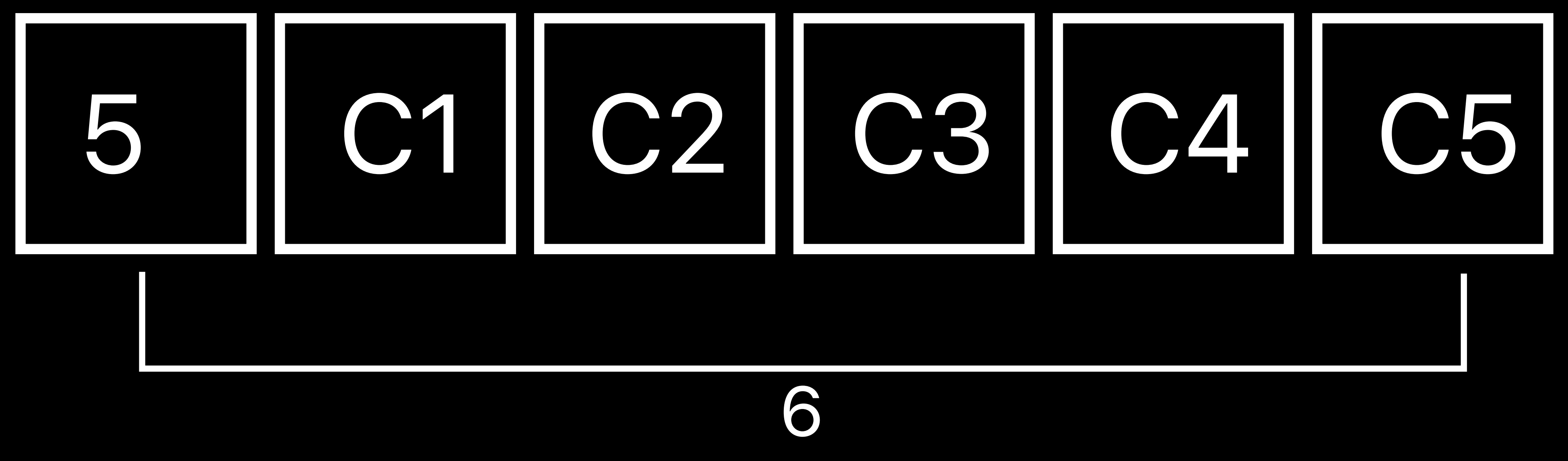
```
template <class _Tp, class _Dp>
class unique_ptr<_Tp[], _Dp> {
  // ...
  template <class _Deleter,
            class _Tp,
            __enable_if_t<__is_default_deleter<_Deleter>::value &&
            __has_array_cookie<_Tp>::value, int> = 0>
  constexpr bool __in_bounds(_Tp* __ptr, size_t __index) const {
    size_t __cookie = std::__get_array_cookie(__ptr);
    return __index < __cookie;
  }
```



Sometimes, an ABI change is not necessary

Inside a `unique_ptr<T[]>`, we can get the size from the array cookie

```
template <class _Tp, class _Dp>
class unique_ptr<_Tp[], _Dp> {
  // ...
  template <class _Deleter,
            class _Tp,
            __enable_if_t<__is_default_deleter<_Deleter>::value &&
            __has_array_cookie<_Tp>::value, int> = 0>
  constexpr bool __in_bounds(_Tp* __ptr, size_t __index) const {
    size_t __cookie = std::__get_array_cookie(__ptr);
    return __index < __cookie;
  }
```



Deployment Experience

Very positive experience so far:

- We have several projects at Apple that use hardening, including WebKit and Darwin kernel (XNU)
- Other adoption: Chrome and Google Andromeda*
- 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

Standardization Path

P3471 "Standard library hardening"

- Mark some existing library preconditions as *hardened*
- Provide a single hardened mode that checks hardened preconditions

https://wg21.link/P3471

P3471 "Standard library hardening"

- Mark some existing library preconditions as *hardened*
- Provide a single hardened mode that checks hardened preconditions

(24.7.2.2.6) Element access 24.7.2.2.6 [span.elem]

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

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

https://wg21.link/P3471

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

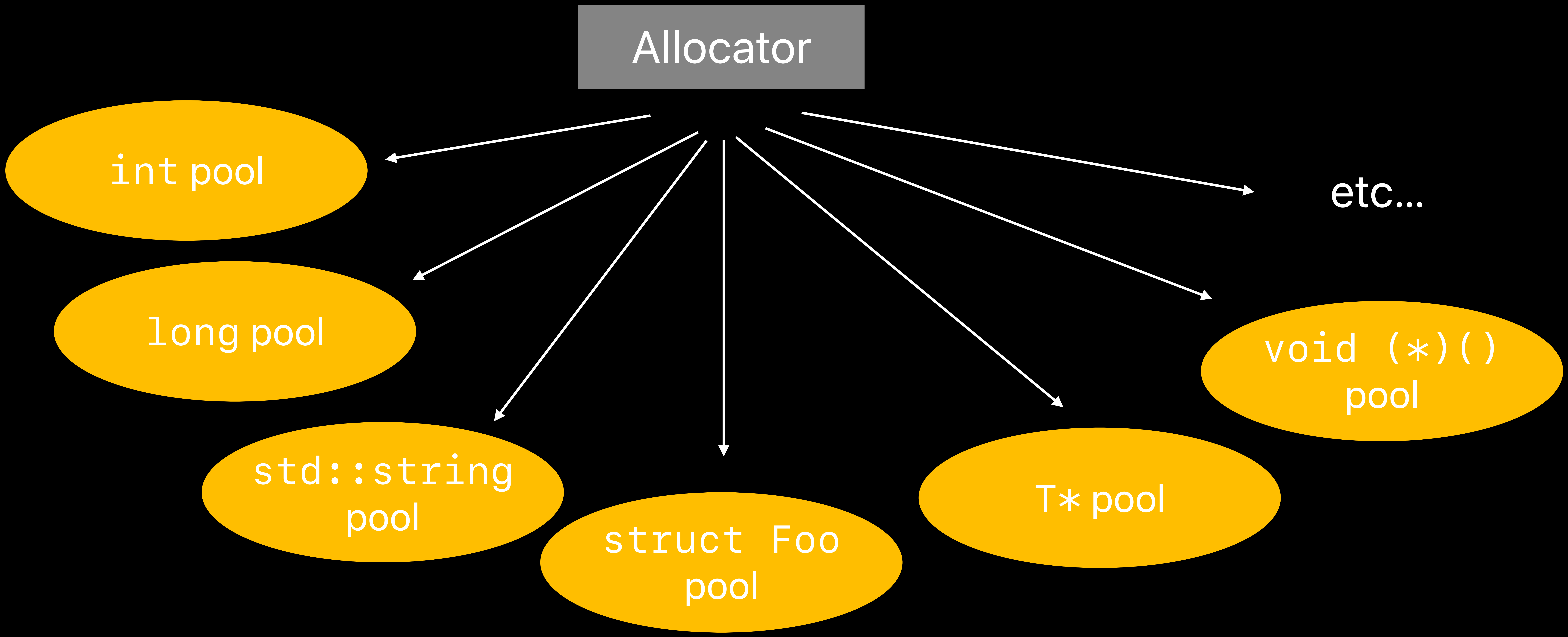
If memory is never reused for a different type, confusions are impossible

➡️ Isolate allocations by type!

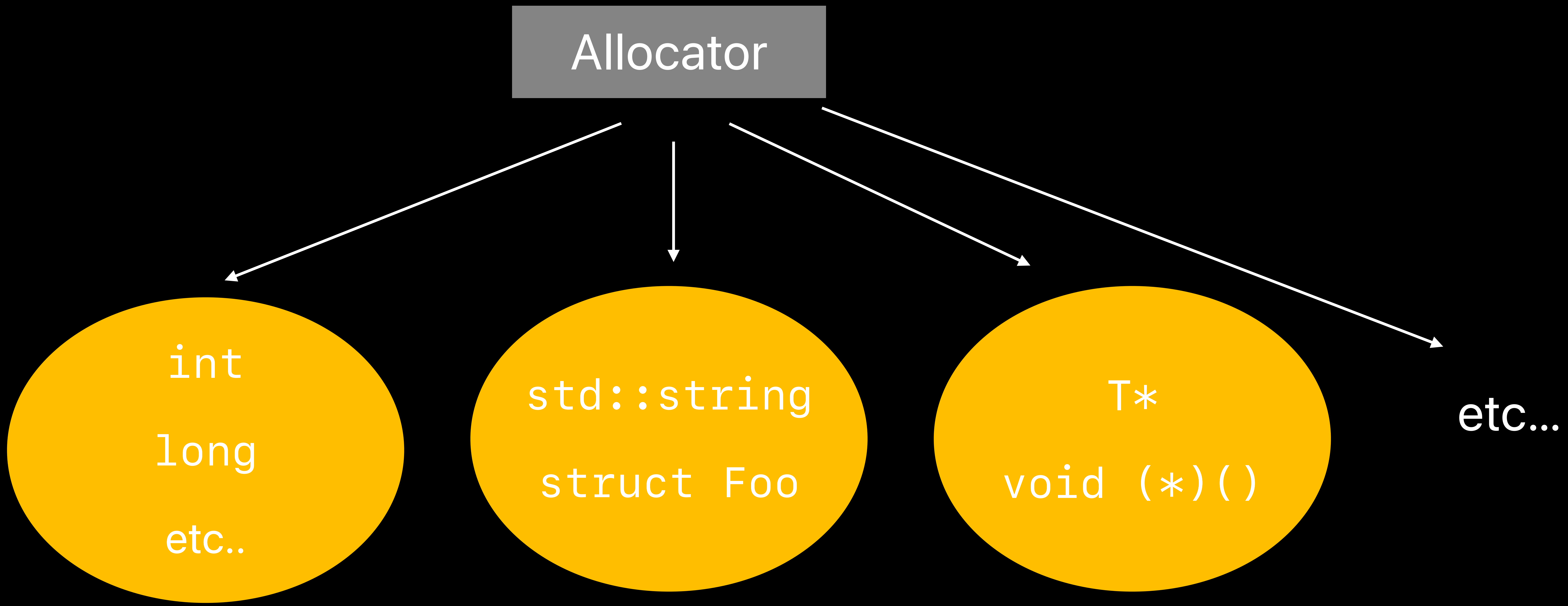
This was utilized in the Darwin Kernel a few years ago *

* 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

How Effective Is Type Isolation?

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

Type Isolation 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;
}
```

Deployment Experience

- 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

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(type_identity<T>{}, sizeof(T), args...)
T::operator new(sizeof(T), args...)
::operator new(type_identity<T>{}, sizeof(T), args...)
::operator new(sizeof(T), args...)
```

https://wg21.link/P2719

Users Could Now Write

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

template <std::derived_from<Character> T>
void* operator new(std::type_identity<T>, std::size_t size) {
  // ... 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::type_identity<_Tp>, std::size_t __size) {
  std::__type_descriptor_t __descriptor = __builtin_type_descriptor(_Tp);
  // ... typed operator new implementation ...
}
```

Conclusions

- Standard Library hardening tackles (mostly) spatial memory safety
  - May require adoption to be effective
  - Great for bug finding and production “hardening”
  - We would like ISO C++ to make this a portable guarantee
  - Go try it out!

Conclusions

- TMO makes temporal memory issues harder to exploit
  - Adoption is almost 100% non-intrusive
  - 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!