

Curiously Recurring C++ Bugs

at Facebook

Louis Brandy
Facebook

C++ @ fb

- Second most used language at Facebook
- The infrastructure and services stack powered primarily by C++
- By cpu-cycle, C++ binaries dominate
- Cross-platform code.

my team

- Responsible for the C++ tooling and core libraries
- C++ onboarding, teaching
- Tools: compilers, sanitizers, CI, build, etc.
- Core libraries: folly, "common", refactors
- Frequently involved in nasty bugs, escalations.

Agenda

- Bugs.
- Some history, some examples, some consequences.
- Mitigation (welcome to the church of ASAN).
- All the tools at our disposal.
- warning: gcc/clang bias from me.

Two audiences

- Audience #1. Show you a bug you've never seen before.
 - Save you from a really, really bad day.
- Audience #2. Show you a bug you've seen a hundred times
 - that you never, ever make yourself anymore...
 - How can we fix it?

some philosophical gibberish

- Modern C++ is an idiom-based language.
- "We" explore, reason, argue, and eventually identify idioms
- New people keep coming.
- Spreading new best practices? Language? Tools? Slides?
- us vs them

Learning C++

- a talk about education
- and the bugs that live in "the gap"
- where expectations \neq reality

Bug #1

```
std::vector::operator[]
```


Bug #1

```
std::vector::operator[]  
    array[]  
    *ptr
```

vector::operator[]

```
return vec[27];
```

- Crashing the Site 101.
- Eternal problem.
- ... and for our purposes, generally uninteresting.

vector::operator[]

- Buffer overflows, unchecked ranges, etc.
- Bad pointer dereference.
- But tools?

mitigation

- Static analysis. Complicated, hard, expensive.
- Improved abstractions (range-based operations).
- Dynamic analysis.
 - Bounds checking
 - Address sanitizer (+fuzzing!)
 - `-fsanitize=address`

Bug #2

greatest C++ newbie bug

Bug #2

```
std::map::operator[]
```

std::map::operator[]

```
map<string, int> m;  
m["hey"] = 12;  
cout << m["hye"];
```

Weird, but useful

```
map<char, int> occ;  
for (char c : str) {  
    occ[c]++;  
}
```


Won't compile

const correctness

```
void Widget::configure(  
    const map<string, string>& settings) {  
    m_timeout = settings["timeout"];  
    m_size = settings["max_items"];  
}
```

Nooooooo!

```
void Widget::configure(  
    map<string, string>& settings) {  
    m_timeout = settings["timeout"];  
    m_size = settings["max_items"];  
}
```

"error passing const"

```
error: passing 'const std::map<std::string<char>,  
                                std::string<char> >  
as 'this' argument discards qualifiers
```

Twice

```
Widget::Widget(  
    const map<string, int>& settings) :  
        m_settings (settings) {  
  
  
  
  
  
  
  
  
  
}
```

Twice

```
Widget::Widget(  
    const map<string, int>& settings) :  
    m_settings (settings) {  
std::cout << "initializing with ... \n"  
    << "timeout: "  
    << m_settings["timeout"];  
    << std::endl;  
}
```

Helpfully set timeout to 0

```
Widget::Widget(  
    const map<string, string>& settings) :  
    m_settings (settings) {  
    std::cout << "initializing with ... \n"  
        << "timeout: "  
        << m_settings["timeout"];  
        << std::endl;  
    }
```

mitigation

- none
- const correctness?
- ban it?



Bug #3

`mapGetDefault()`

some haskell!

```
findWithDefault def k m  
  = case lookup k m of  
      Nothing -> def  
      Just x   -> x
```

folly::get_default

```
template <class Map, typename Key>
typename Map::mapped_type get_default(
    const Map& map,
    const Key& key,
    const typename Map::mapped_type& dflt ) {
    auto pos = map.find(key);
    return (pos != map.end() ? pos->second : dflt);
}
```

string-version

```
string get_default(  
    const map<string,string>& map,  
    const string& key,  
    const string& dflt) {  
  
    auto pos = map.find(key);  
    return (pos != map.end() ?  
            pos->second : dflt);  
}
```

fixed!

```
const string& get_default(  
    const map<string,string>& map,  
    const string& key,  
    const string& dflt) {  
  
    auto pos = map.find(key);  
    return (pos != map.end() ?  
            pos->second : dflt);  
}
```

Sorry, this page isn't available

The link you followed may be broken, or the page may have been removed.



People love temporary
default values.
(understandably)

```
auto& v = get_default(m, k, "127.0.0.1");
```

Broader class of problems

- "smuggling" a reference to a temporary through a function.
- One of many, many examples.
- C++ standardization: smart references make this worse?

mitigation

- Address sanitizer will catch this!
- Maybe with extra flags. Depends on clang version. I think.
- `-fsanitize-address-use-after-scope`

folly's fix (sorta)

```
template <class Map, typename Key>  
const typename Map::mapped_type& get_ref_default(  
    const Map& map,  
    const Key& key,  
    typename Map::mapped_type&& dflt) = delete;
```

Bug #4

a success story

volatile

volatile lint warning

'volatile' does not make your code thread-safe. If multiple threads are sharing data, use `std::atomic` or locks. In addition, 'volatile' may force the compiler to generate worse code than it could otherwise.

we had so many arguments

std::atomics

- ... have been wildly successful.
- Solved a very hard, social problem.
 - expectations == reality?
- ... maybe too successful. That's a longer story.

Bug #5

the worst question in all of C++

Is

Is shared_ptr

Is `shared_ptr` thread-safe?



is shared_ptr|



is shared_ptr **thread safe**

is_shared_ptr

is shared_ptr **slow**

is shared_ptr **null**

Google Search

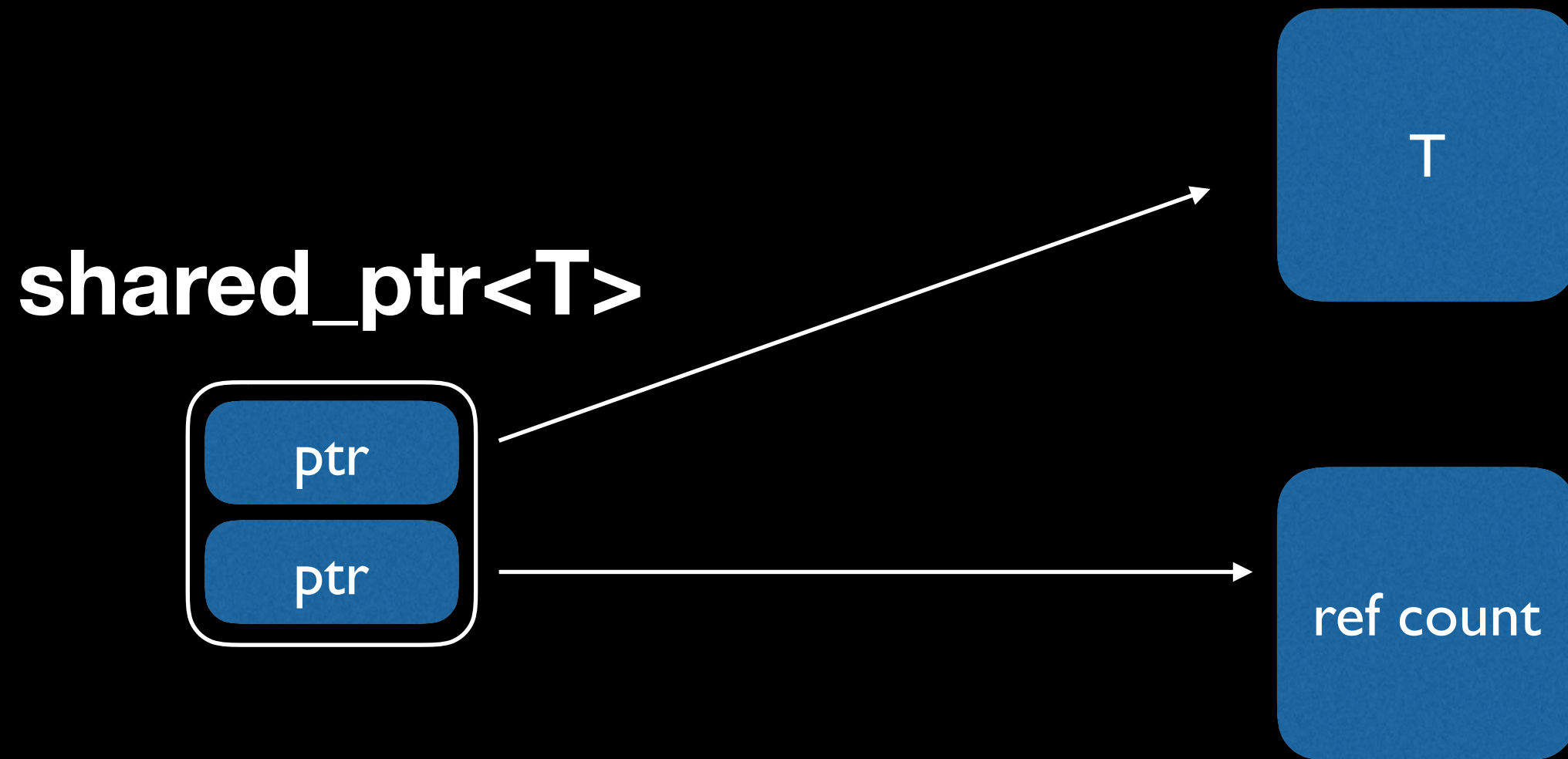
I'm Feeling Lucky

Report inappropriate predictions

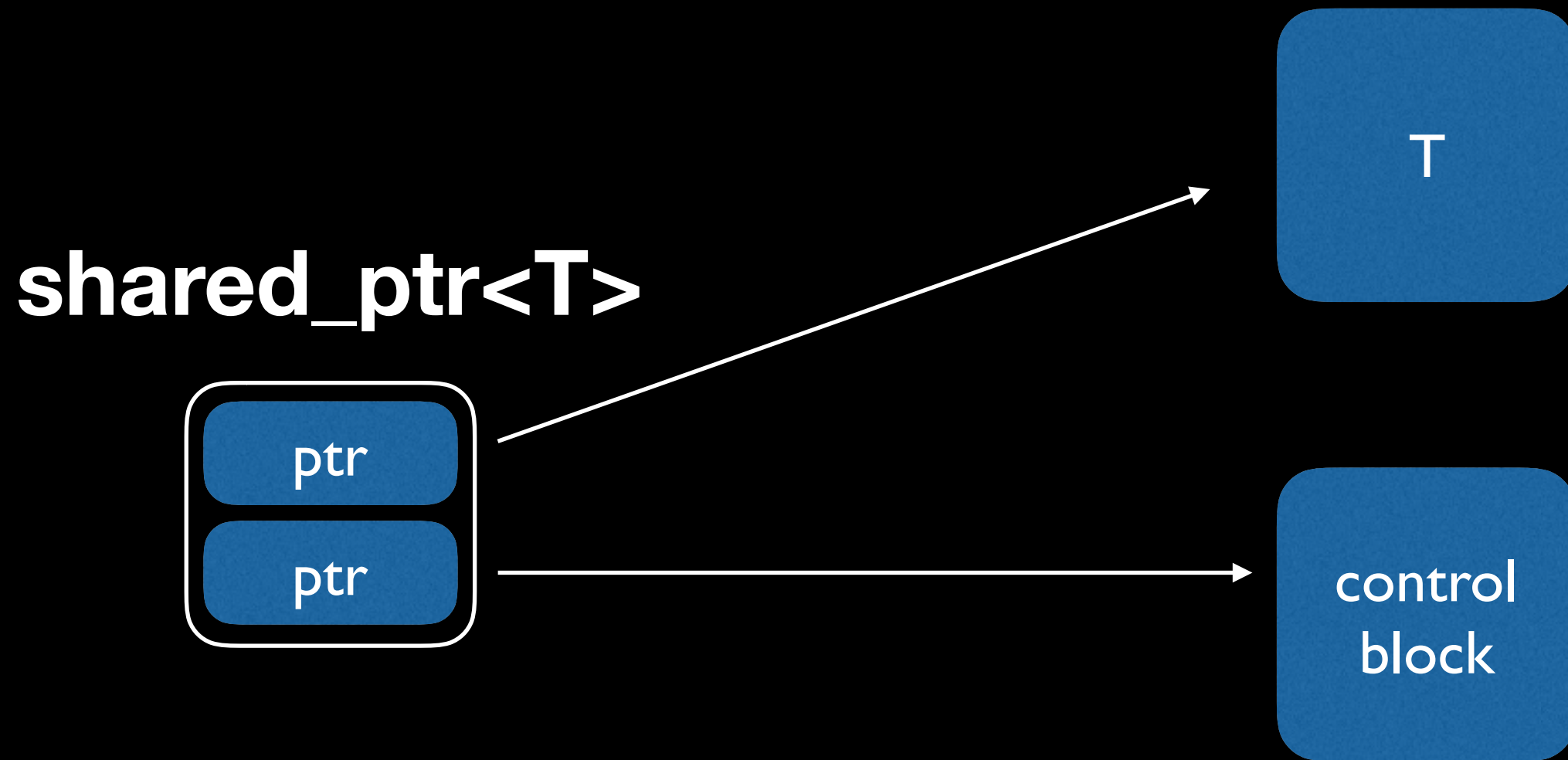
Correct answer

- Is `shared_ptr` thread-safe?
- If you need to ask...
- Let's go with "No!"

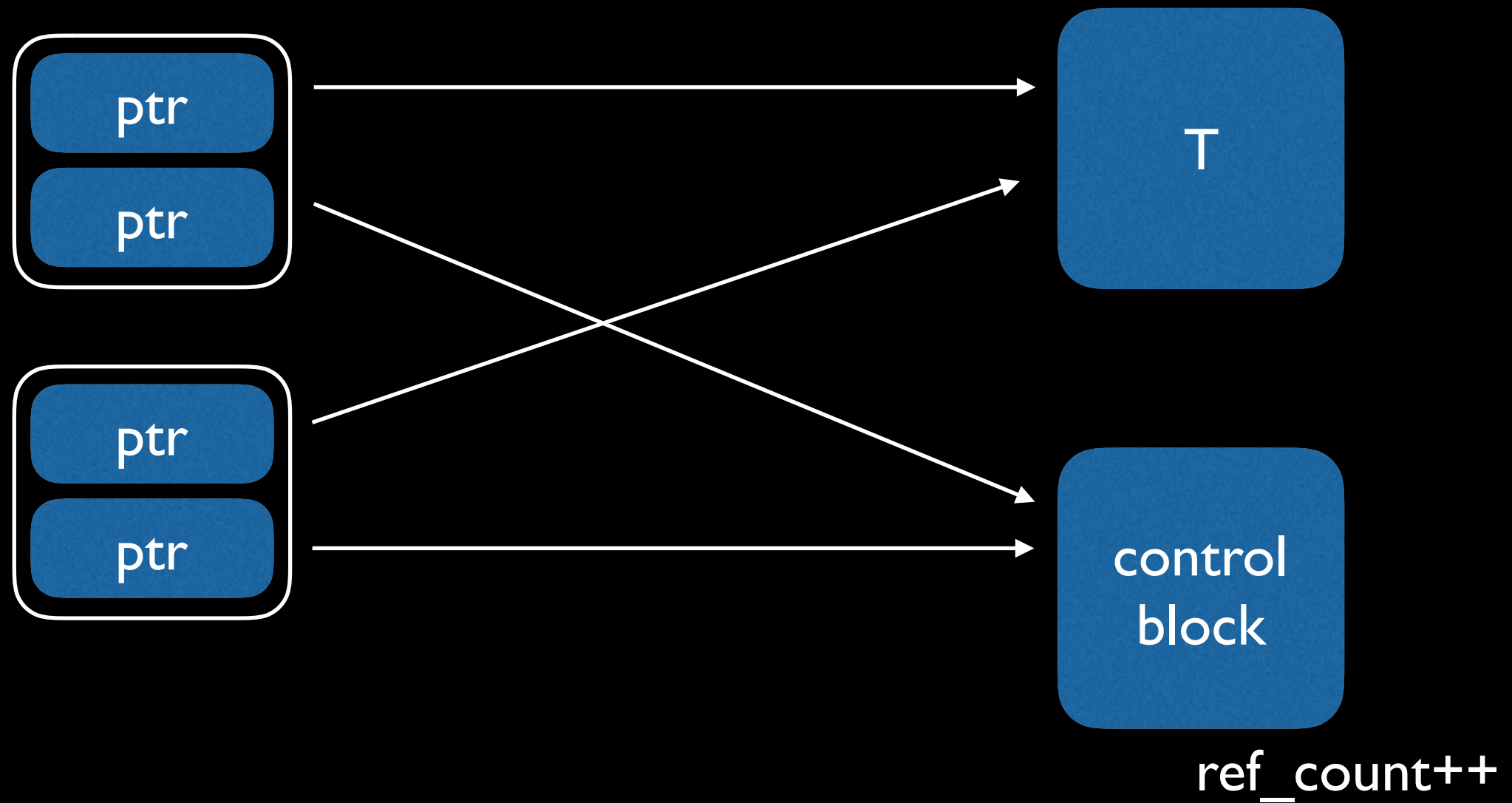
a naive mental model



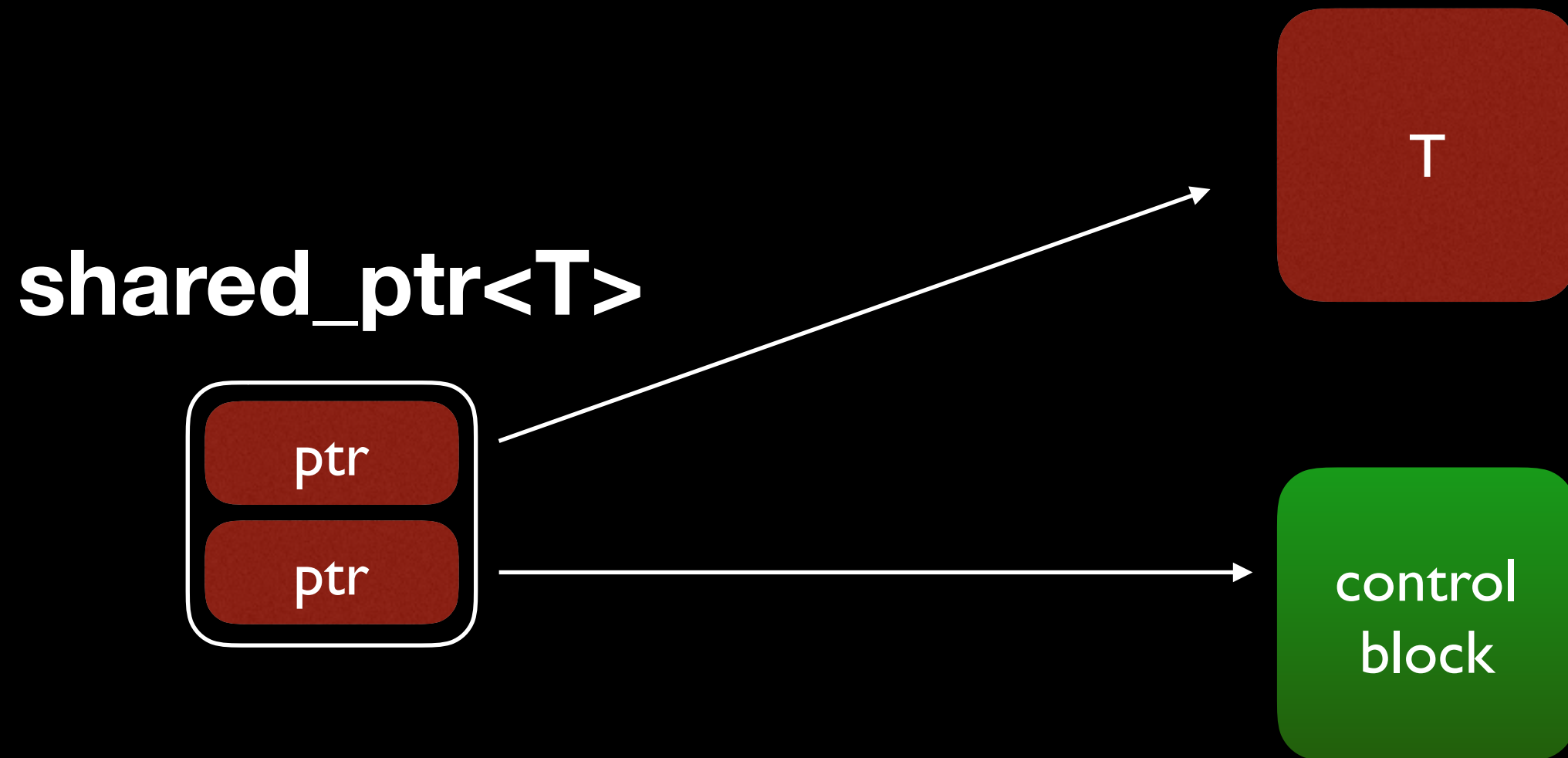
thread safety



copying

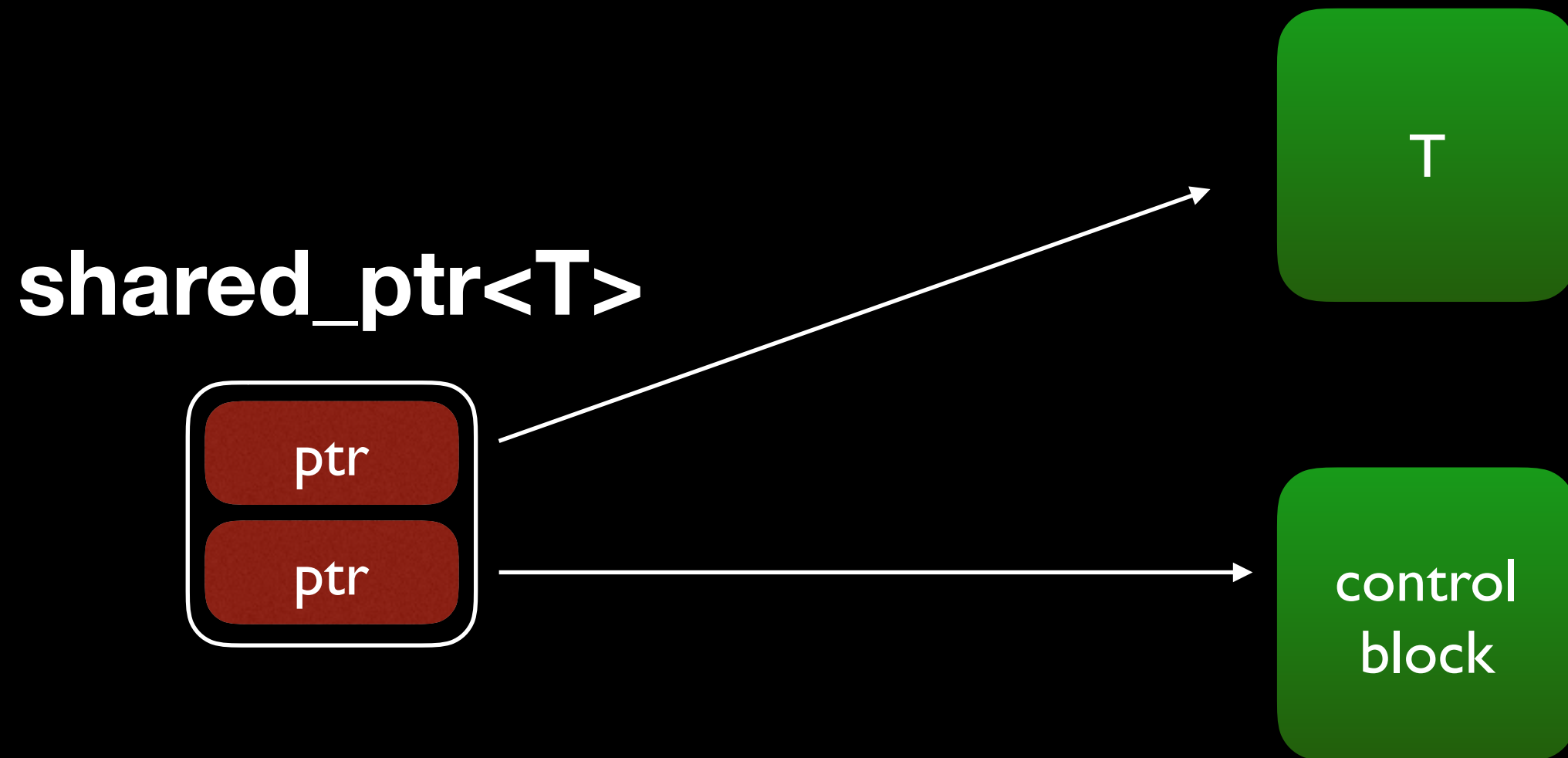


thread safety



- Thread safe
- NOT thread safe

Our intuition



Expected



Not so expected

buggy RCU

- Read-Copy-Update
- One thread is writing (updating) to some one-true-shared_ptr
- Many threads are reading (copying) the shared_ptr.
- Use a lock on the shared_ptr at the center.

mitigation

- Sanitizers will catch this **IF** you reliably exercise it.
- Thread sanitizer.
- Address Sanitizer often too, though incidentally.
- Library!
 - C++11: `atomic<shared_ptr>`
 - Concurrency TS: `atomic_shared_ptr`
 - Maybe back? `atomic<shared_ptr>`

related

```
auto& ref = *returns_a_shared_ptr();  
ref.boom();
```

Sanitizers will catch this too.

Bug #6

code review

does this compile?

```
#include <string>
```

```
void f() {  
    std::string(foo);  
}
```

does this compile?

```
#include <string>

void f() {
    std::string(foo);
}
```

✓ YES

Same.

```
std::string(foo);
```

```
std::string foo;
```


x86-64 gcc 7.1 (Editor #1, Compiler #1) x

x86-64 gcc 7.1

-std=c++11

11010

.LX0:

.text

//

\s+

Intel

A



1 f():

2 push rbp

3 mov rbp, rsp

4 sub rsp, 32

5 lea rax, [rbp-32]

6 mov rdi, rax

7 call std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::basic_string()

8 lea rax, [rbp-32]

9 mov rdi, rax

10 call std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::~~basic_string()

11 nop

12 leave

13 ret

Ok, but so what?

```
std::string(foo);
```

```
std::string foo;
```

See the problem?

```
void Obj::update() noexcept {  
  
}
```

See the problem?

```
void Obj::update() noexcept {  
    unique_lock<mutex>(m_mutex);  
    do_the_mutation();  
}
```

Fixed

```
void Obj::update() noexcept {  
    unique_lock<mutex> g(m_mutex);  
    do_the_mutation();  
}
```

Do a code search.

You'll find em.

Do a code search.

While prepping these slides...

I found a bug.

```
std::unique_lock<std::mutex>(mtx);
```

```
126 std::chrono::steady_clock::time_point Reader::getLastDataTime() {  
127     std::unique_lock<std::mutex>(mtx);
```

Louis Brandy (ldbrandy) commented

Line 127 Previous · Next · Like · Reply

just fyi this is a bug. It needs to have a name.

```
std::unique_lock<std::mutex> **lock**(mtx);
```


I found two bugs.

```
unique_lock<mutex>(m_);
```

```
11 shared_ptr<Client> ChromosomeJob::getClient(  
12     const std::string& clusterName) {  
13     unique_lock<mutex>(m_);
```

Louis Brandy (ldbrandy) commented

[Line 13](#) [Previous](#) · [Next](#) · [Like](#) · [Reply](#)

just fyi, this is a bug... the local variable needs a name....

```
unique_lock<mutex> lock(m_);
```

the worst C++ quiz
does it compile?

Remember?

```
#include <string>

void f() {
    std::string(foo);
}
```

✓ YES

does this compile?

```
#include <string>

void f() {
    std::string(foo);
    std::string("wow");
}
```

✓ YES

does this compile?

```
#include <string>

void f() {
    std::string(foo);
    std::string(foo);
}
```

X NO

does this compile?

```
#include <string>

void f() {
    std::string(foo);
    std::string{foo};
}
```

✓ YES

does this compile?

```
#include <string>

void f() {
    std::string(foo);
    {std::string(foo);}
}
```

✓ YES

Some important notes

```
unique_lock<mutex>(m_mutex);
```

- RAI-like types are especially affected because you won't otherwise "use" them.
- Only works because of the default constructor.
 - `std::unique_lock` is affected.
 - `std::lock_guard` is not.
- RAI types + default constructors -> **DANGER**.

Mitigation

- This is a shadowing bug. `-Wshadow` finds this.
- `-Wshadow` tends to be noisy and isn't used often.
- `-Wshadow-compatible` and `-Wshadow-compatible-local` do **not** find these.

```
<source>: In function 'void f()':  
10 : <source>:10:41: warning: declaration of 'm_mutex' shadows a global declaration [-Wshadow]  
    std::unique_lock<std::mutex>(m_mutex);  
                                ^  
5 : <source>:5:12: note: shadowed declaration is here  
    std::mutex m_mutex;  
    ~~~~~  
Compiler exited with result code 0
```

Easily detectable

- ... but not widely detected.
- Our clang based linter finds these, now.
- Warn on extraneous parenthesis in declarations?
- Perhaps a `[[nodiscard]]` attribute for constructors?

Some conclusions

- C++ is hard.
- Tools are our best teaching weapons.
- Invest in a good, extendable linter (e.g. clang-tidy)
- ASAN is life.

Questions?