Destructor Case Studies

Best Practices for Safe and Efficient Teardown

Pete Isensee
Facebook Reality Labs

Case Study: End Brace

```
    }
```

Destructor Case Studies

Best Practices for Safe and Efficient Teardown

Pete Isensee
Facebook Reality Labs

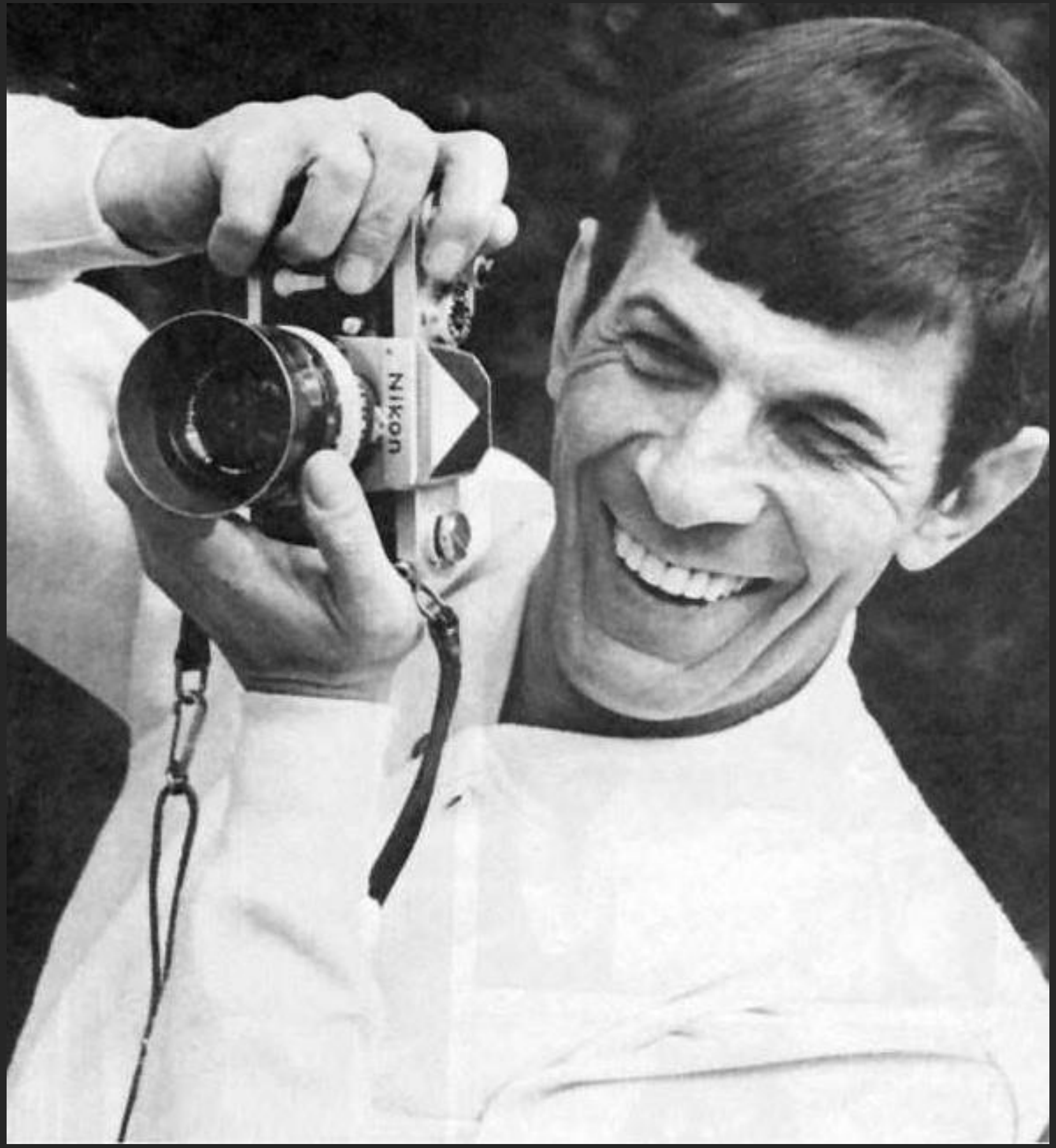
Slides and Code

Presentation

https://tinyurl.com/y3ehsaxt

Source Code

https://godbolt.org/z/OUJp7F



Baseline

C++17 Standard

C++ Core Guidelines

Standard libraries

- Visual Studio 2017
- GCC 9.2
- Clang libc++ 8.0.1

INTERNATIONAL STANDARD

ISO/IEC 14882

Fifth edition
2017-12

C++ Destructors Defined

One

Deterministic

Automatic

Symmetric

Special

Member

Function

With

- No name
- No parameters
- No return type

Designed to

- Give last rites
  before object death

When Destructors Are Invoked

Scenario	Destructor called	Notes
Named automatic	Scope exit	Called at }
Statics and globals	Program exit	Reverse order of construction
Thread locals	Thread exit	Reverse order of construction
Free store	delete expression	Prior to memory being freed
Array elements	From last element to first	Reverse order of construction
STL container elements	Container destroyed	Unspecified order
Temporary	End of expression in which created	Unless bound to ref/named obj
Exception thrown	Stack unwinding	Reverse order of construction
Explicit dtor	t.~T() or p->~T();	Rare
exit()	For global & static objects only	Plus atexit functions; no locals
abort()	No; immediate app exit	No auto, global, or static dtors

Case Study: No Dtor Declared

```
// std::pair
template <typename T1, typename T2>
struct pair {
  T1 a;
  T2 b;
  pair(): a(), b() {}
  pair(const pair&) = default;
  pair(pair&&) = default;
  pair(const T1& x, const T2& y) : a(x), b(y) {}
  // ... Destructor not specified
};
```

Implicit Destructors

Not specified by programmer

Public and inline

Non-throwing unless base or members throw

Implicitly declared as defaulted

```
// As if you wrote:
~pair() noexcept = default;
```

Implicit dtor appropriate for most objects

Recommendation

Avoid specifying dtors whenever possible

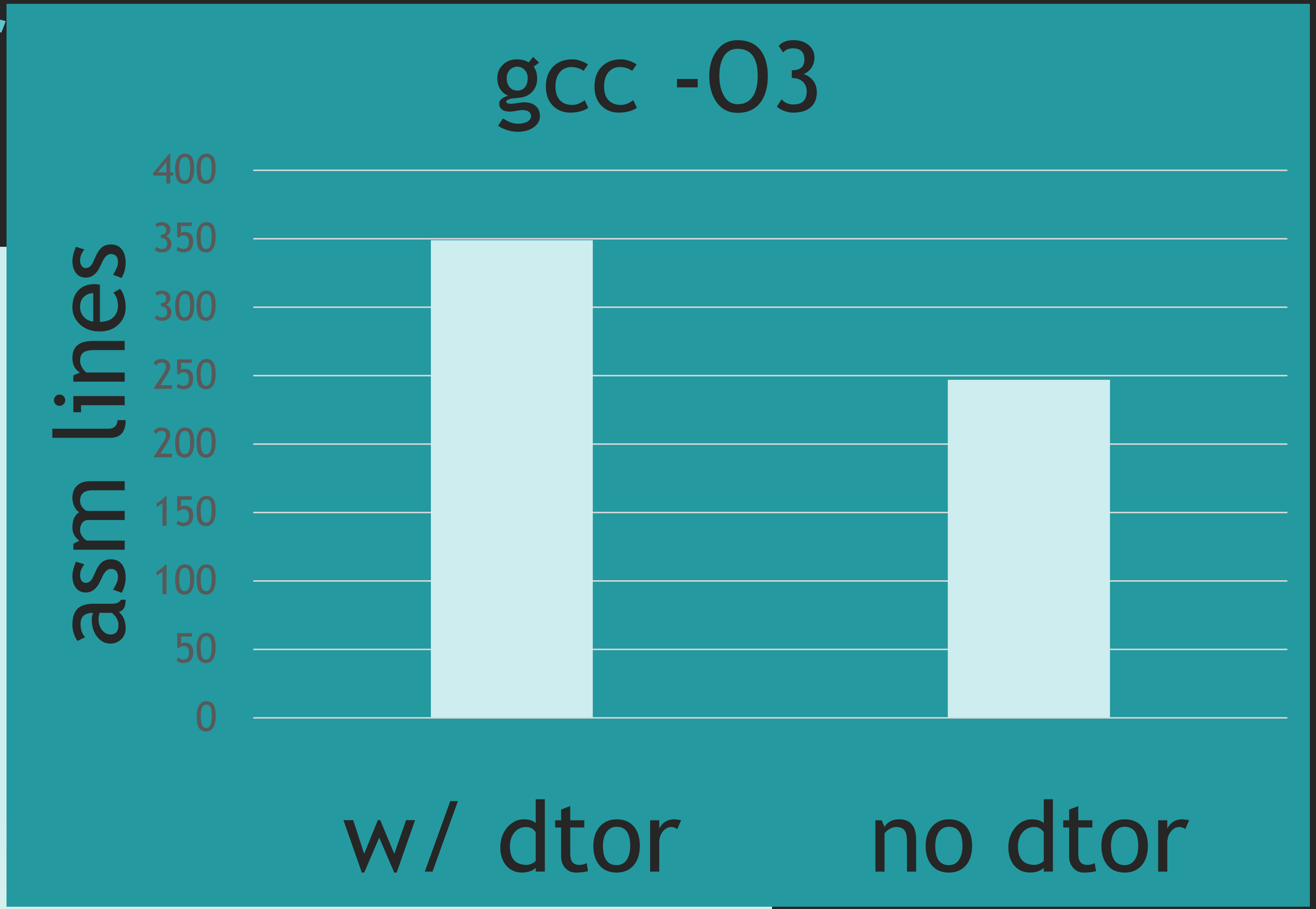
- See *Rule of Zero*

Only declare dtors for classes that require them

- Clearly conveys intent
- See *Rule of Five* and *Rule of All or Nothing*

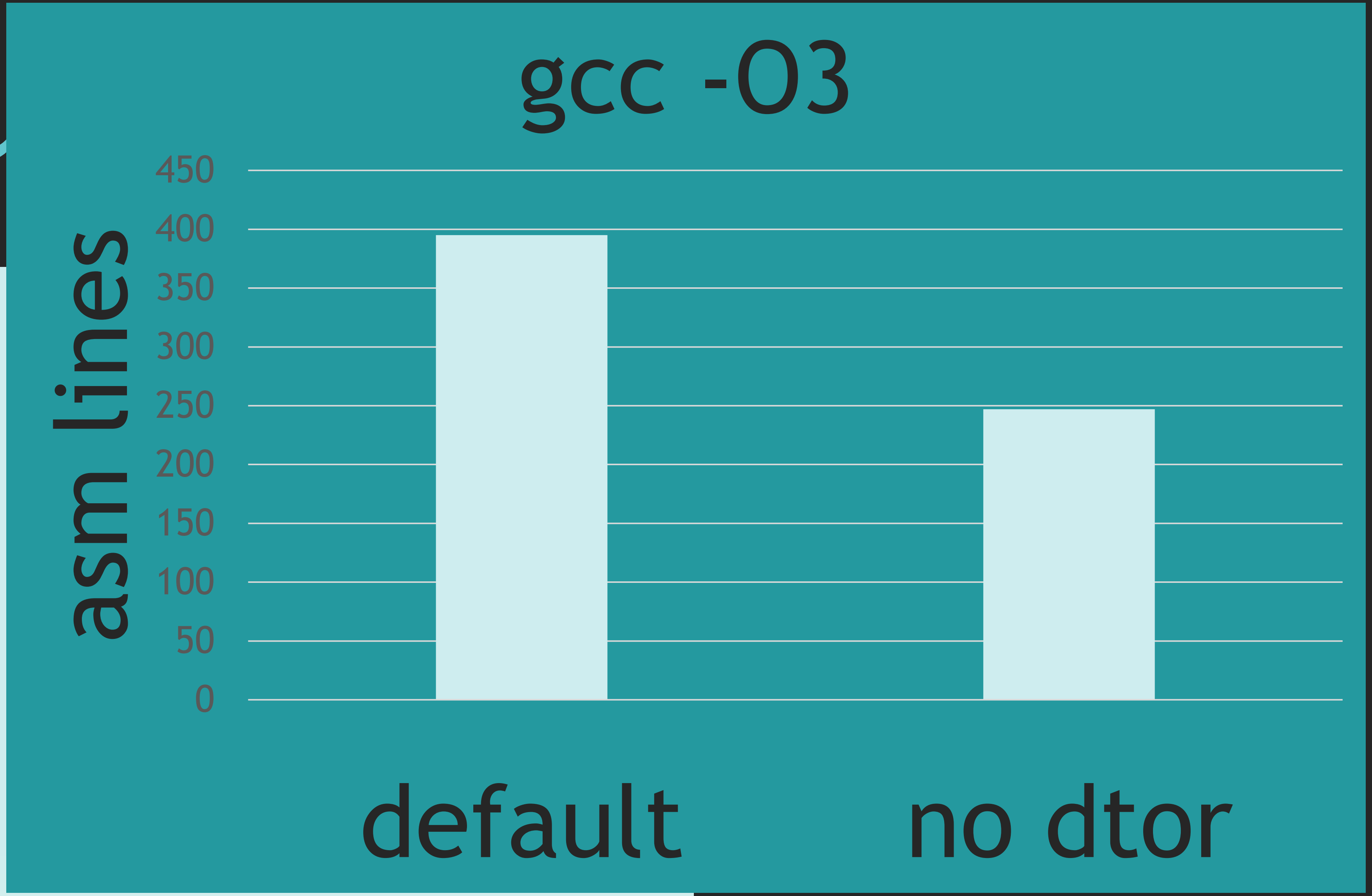
Case Study: Perf

```
class Tribble {
  std::string name;
  int ID;
public:
  ~Tribble();
};

int main() {
  std::vector<Tribble> v;
  v.emplace_back(); v.emplace_back();
}
```



Case Study: Defau

```
class Tribble {
  std::string name;
  int ID;
public:
  ~Tribble() = default;
};

int main() {
  std::vector<Tribble> v;
  v.emplace_back(); v.emplace_back();
}
```



Strong Recommendation

The best destructor is no destructor

Embrace implicit dtors

Only declare dtors when they are required

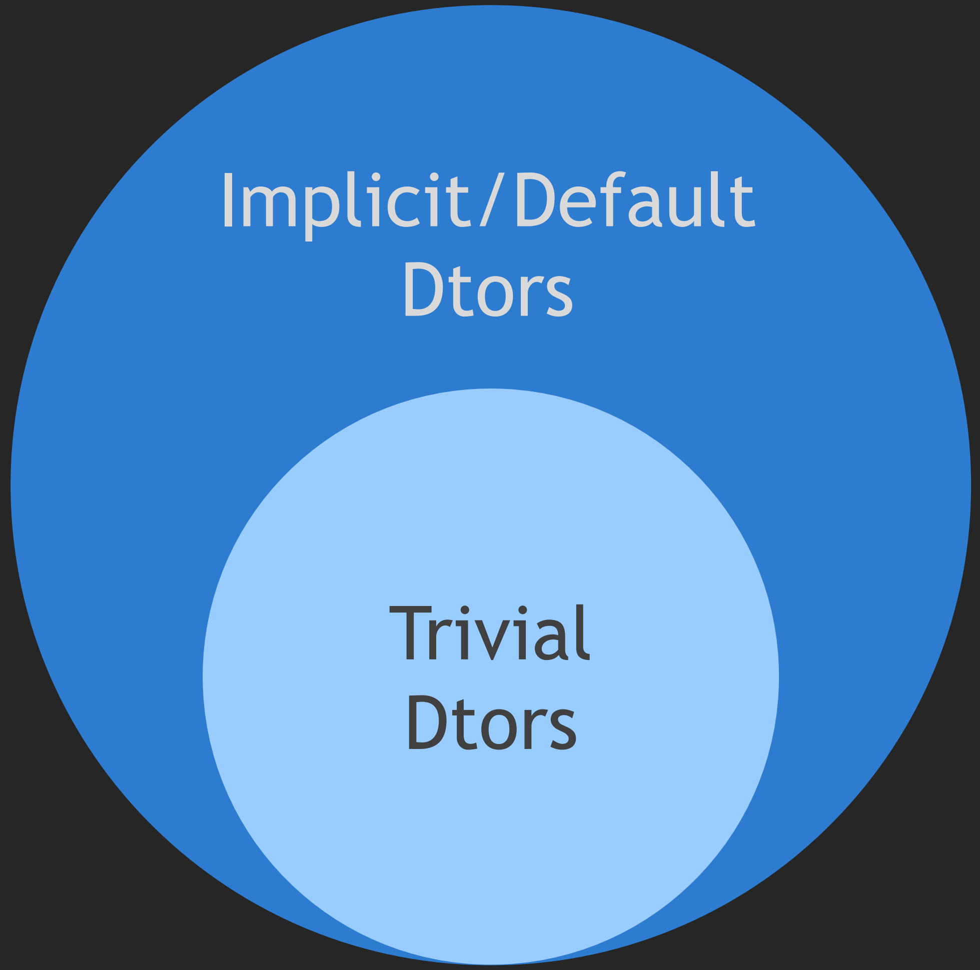
```
class Tribble {
  std::string name;
  int ID;
};
```

Case Study: Trivial Dtors

```
// std::bitset
template <size_t Bits>
class bitset {
  enum { Words = /* math on Bits and CHAR_BIT */ };
  unsigned long long array[Words]; // array of POD
public:
  constexpr bitset() noexcept;
  constexpr bitset(unsigned long long) noexcept;

  // no destructor declared
};
```

Trivial Destructors

- Requirements
  - Implicit (not declared) or defaulted (=default)
  - Not virtual
  - Base classes have trivial dtors
  - Non-static members have trivial dtors
- Trivial destructors do nothing
  - Compiler can optimize away!



Case Study: Extra Work

```
~WarpCore() {
  if (dilithiumChamber != nullptr)
    delete dilithiumChamber;
  dilithiumChamber = nullptr;
  matterAntimatterReactor.clear();
  magneticField.reset();
  plasmaConduitCount = 0;
}
```

Avoid Redundant/Unnecessary Work

`delete/free` handle nullptr/NULL internally

Avoid zeroing member pointers, handles, PODs

Let member data clean up after itself

```
~WarpCore() {
  delete dilithiumChamber;
}
```

Case Study: Public Funcs in Dtors

```
~WarpCore() {
  Shutdown(); // Is this OK?
}
void Shutdown() {
  delete dilithiumChamber;
  dilithiumChamber = nullptr;
  matterAntimatterReactor.clear();
  magneticField.reset();
  plasmaConduitCount = 0;
}
void Startup() { /* … */ }
```

Avoid Calling Public Funcs in Dtors

Public functions must maintain class invariants

Destructors don’t need to maintain invariants

Avoid the overhead of unnecessary functions

```
~WarpCore() {
  delete dilithiumChamber;
}
```

Case Study: Raw Resource

```
class Phaser {
  HANDLE phaserEvent;
  // Other data
public:
  ~Phaser() {
    if (phaserEvent)
      CloseHandle(phaserEvent);
    // Other cleanup code
  }
};
```

Resource Wrapper

```
struct ScopedHandle {
  HANDLE h;
  ScopedHandle(): h(INVALID_HANDLE_VALUE) {}
  ScopedHandle(HANDLE handle): h(handle) {}
  operator HANDLE() { return h; }
  ~ScopedHandle() {
    if (h != INVALID_HANDLE_VALUE)
      CloseHandle(h);
  }
};
```

Wrap Raw Resources

```
class Phaser {
  ScopedHandle phaserEvent;
  // Other data
public:
  ~Phaser() {
    // Other cleanup code
  }
};
```

Takeaway: Put any resource that needs to be released in its own object (RAII)

Case Study: Raw Pointers

```
class Uhura {
  X* x; Y* y;
public:
  Uhura() : x(new X), y(new Y) { } // Alert: leaky
  ~Uhura() { delete x; delete y; }
};
```

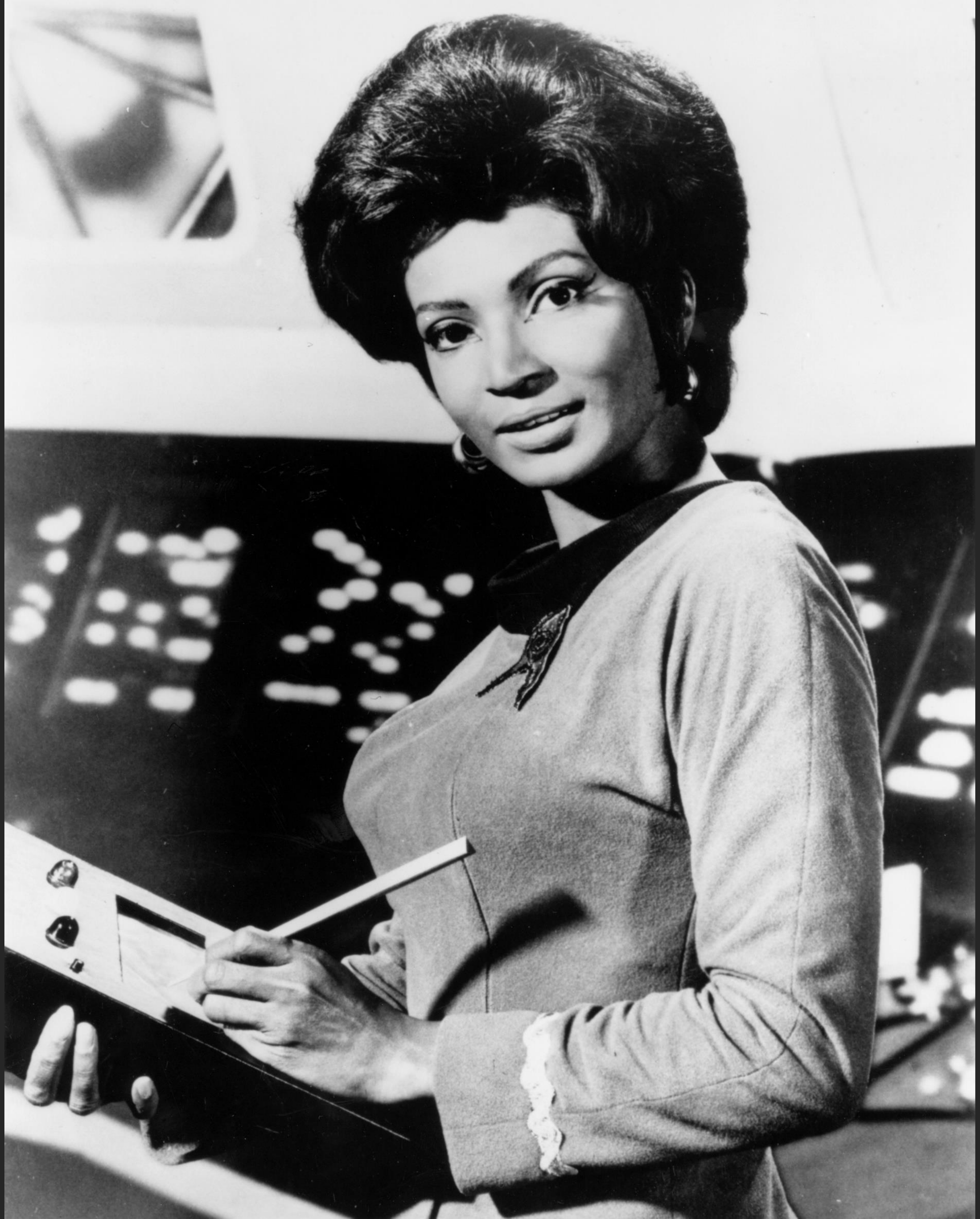
Dtors only called for fully constructed objs

If ctor throws, object not fully constructed

Wrap Raw Pointers

```
class Uhura {
  std::unique_ptr<X> x;
  std::unique_ptr<Y> y;
public:
  Uhura(): x(new X), y(new Y) { }
};
```

Takeaway: Store only a single raw resource (pointer, handle, lock, etc.) in a class



Case Study: Raw Pointers, Part II

```
class Chekov {
  std::vector<Wessel*> serviceRecord;
public:
  ~Chekov() {
    for (auto* p : serviceRecord)
      delete p;
  }
};
```

Wrap Raw Pointers, Part II

```
class Chekov {
  std::vector<std::unique_ptr<Wessel>> serviceRecord;
public:
  // dtor no longer necessary
};
```

Takeaway: Don’t store owned pointers in containers

Case Study: Threads

```
class Scotty {
  std::vector<std::thread> pool;
public:
  ~Scotty() { // necessary?
    for (auto& t : pool) {
      if (t.joinable())
        t.join();
    }
  }
};
```

Prefer Joining Threads

```
class Scotty {
  std::vector<gsl::joining_thread> pool;
public:
  // no dtor necessary
};
```

Joining Threads

```
class joining_thread : public std::thread {
public:
  ~joining_thread() {
    if (joinable())
      join();
  }
  void detach() = delete;
};
```

Prefer joining_thread (or jthread C++20) to thread

Related: don’t detach a thread

Case Study: Virtual Dtors

```
// std::memory_resource
class memory_resource {
public:
  virtual ~memory_resource() {}
  void* allocate(size_t bytes, size_t alignment);
  void deallocate(void* p, size_t bytes, size_t
                  alignment);
private:
  virtual void* do_allocate(/* as above */) = 0;
  virtual void do_deallocate(/* as above */) = 0;
};
```

Virtual Destructors

Guarantee that derived classes get cleaned up

If delete on a Base* could ever point to a Derived*

Rule of thumb: if virtual functions in class

- Destructor should be virtual
- Destructor should be public

Idiom exception: mixins (e.g. old unary_function)

Case Study: Spock

```
class Human : Ego, public virtual Id {};
class Vulcan: Katra, Kolinahr {};

class Spock : Human, Vulcan {
  Tricorder tricorder;
  Phaser phaser;
};

{ Spock s; } // Order of destruction?
```

Order of Destruction

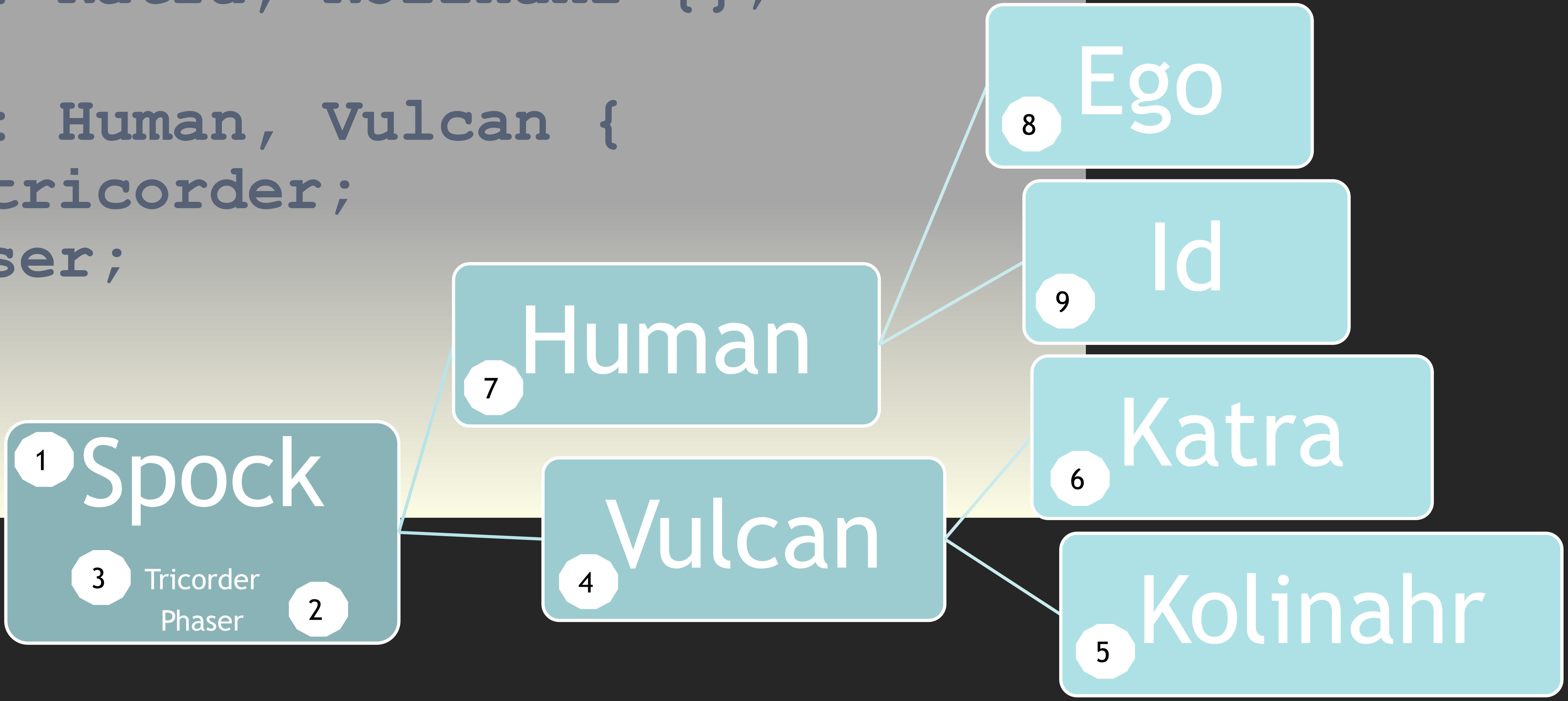
Rule of Thumb: reverse order of construction

Specifically:

1. Destructor body
2. Data members in reverse order of declaration
3. Direct non-virtual base classes in reverse order
4. Virtual base classes in reverse order

Destruction Order Example

```
class Human : Ego, public virtual Id {};
class Vulcan: Katra, Kolinahr {};

class Spock : Human, Vulcan {
  Tricorder tricorder;
  Phaser phaser;
};

{ Spock s; }
```



Case Study: Virtual Funcs in Dtors

```
class HelmsPerson {
public:
   virtual ~HelmsPerson() { Release(); }
private:
   virtual void Release() = 0; // pure virtual
};
class Sulu : public HelmsPerson { … };
```

Takeaway: don’t call virtual functions from destructors (or constructors)

Case Study: Ignoring Exceptions

```
Teleporter::~Teleporter() {
  try {
    Stop();
    pads.reset();
    TeleporterManager::Destroy();
  }
  catch (...) {
  }
}
```

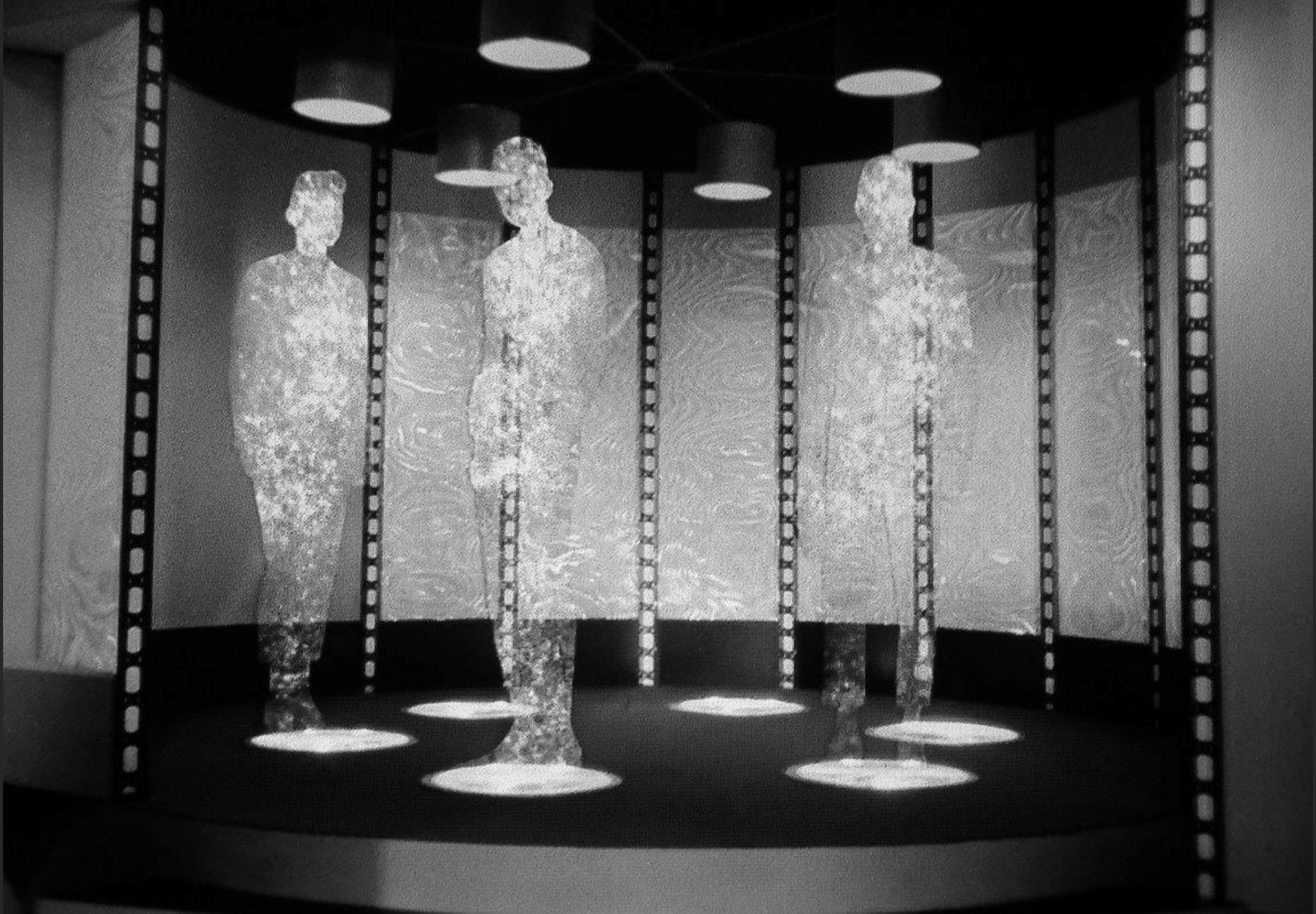
Destructors Should Never Throw

- Reasoning
  - Dtors invoked when exception thrown, stack unwound
  - If another exception is thrown: `terminate()`!
- Never allow an exception to exit a dtor
- Core Guideline: a destructor may not fail
- Try/catch(...) should still be rare

Indicate Dtor Doesn’t Throw

```
Teleporter::~Teleporter() noexcept {
  try {
    Stop();
    pads.reset();
    TeleporterManager::Destroy();
  }
  catch (...) {
  }
}
```

CoreGuidelines best practice



Case Study: Custom Mem Objects

```
class SpecialKirk {
  Kirk* k;
public:
  SpecialKirk() {
    void* raw = myAlloc(sizeof(Kirk));
    k = new (raw) Kirk; // placement new
  }
  ~SpecialKirk() noexcept {
    k->~Kirk(); // explicit destructor
    myFree(k);
  }
};
```

Explicit Destructors

- Destructors can be called directly
- Very powerful for custom memory scenarios
- Example uses
    - Paired w/ placement new
    - std::vector
    - Custom allocators

Custom Allocators

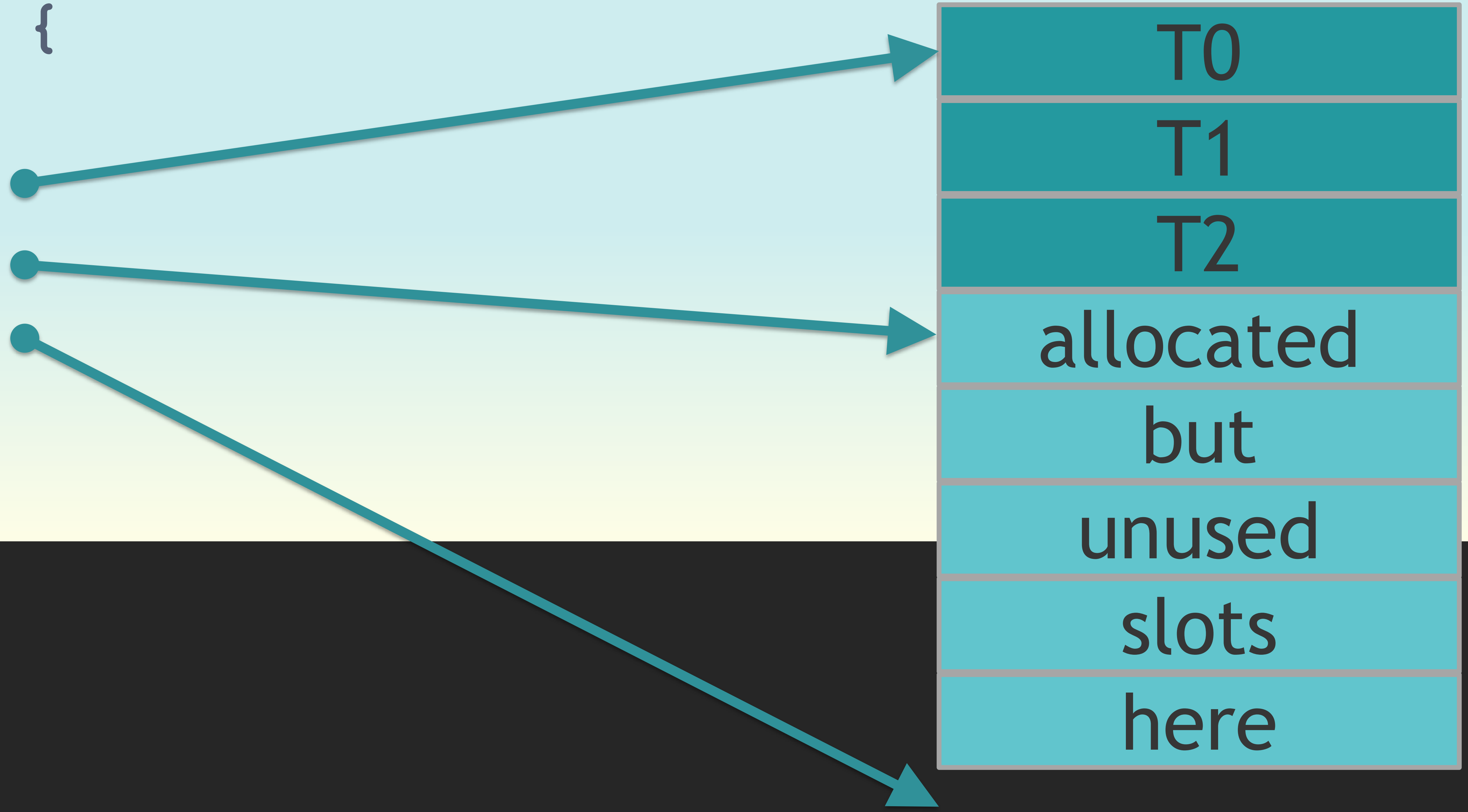
```
template <typename T>
struct MyAllocator : public std::allocator<T> {
  T* allocate(size_t n) {
    auto* raw = myAlloc(n);
    if (raw == nullptr)
      throw std::bad_alloc();
    return static_cast<T*>(raw);
  }
  void deallocate(T* raw, size_t) noexcept {
    myFree(raw);
  }
};
```

Custom Allocator Usage

```
class SpecialKirk {
  Kirk* k;
  MyAllocator<Kirk> a;
public:
  SpecialKirk() {
    auto* raw = a.allocate(sizeof(Kirk));
    k = new (raw) Kirk;
  }
  ~SpecialKirk() {
    k->~Kirk();
    a.deallocate(k, sizeof(Kirk));
  }
```

Vector Internals

```
template <typename T, typename A = allocator<T>>
class vector {
private:
  T* first;
  T* last;
  T* end;
  A  al;
};
```



T0

T1

T2

allocated

but

unused

slots

here

Case Study: Vector Dtor

```
~vector() {
  if (first != nullptr) {
    for (auto* p = first; p != last; ++p) {
      p->~T(); // run dtor on each element
    }
    a.deallocate(first, capacity());
  }
}
```

Side Trip: Destructor Traits

```
#include <type_traits>

class Gorn {
  std::string name;
  int armorClass;
};

static_assert( is_destructible_v< Gorn >);
static_assert( is_nothrow_destructible_v< Gorn >);
static_assert(!is_trivially_destructible_v< Gorn >);
static_assert(!has_virtual_destructor_v< Gorn >);
```

Case Study: Vector Dtor

```
~vector() {
  if (first != nullptr) {
    for (auto* p = first; p != last; ++p) {
      p->~T(); // run dtor on each element
    }
    a.deallocate(first, capacity());
  }
}
```

Fast Vector Destructor

```
~vector() {
  if (first != nullptr) {
    if constexpr (!is_trivially_destructible_v<T>) {
      for (auto* p = first; p != last; ++p) {
        p->~T(); // destroy each element
      }
    }
    a.deallocate(first, capacity());
  }
}
```

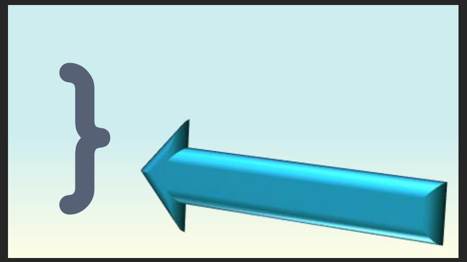
Destructor Faves

```
// no destructor!

= default; // but beware

= delete;

{ assert(…); } noexcept

{ Log(…); } noexcept

{ chkInvariants();} noexc

{ delete p; } noexcept

{ InterlockedDecr();} noe
```

```
{ try { maythrow(); }
  catch(…){ } } noexcept

{ closesocket(…); } noexc

{ free(p); } noexcept

{ SetEvent(…); } noexcept

{ lock_guard<mutex> l(m);
  /*modify shared data*/
} noexcept

{ SecureZeroMemory(p,sz);}
```

Performance

- Destructors are called a LOT
- They’re invisible in code
- Recommendations
  - Streamline common dtors
  - The best dtor is default/empty
  - Inlining may be useful
  - Measure/profile, update, rinse, repeat



Lots o’ destruction here

References

C++17 Standard http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf

Core Guidelines https://github.com/isocpp/cppcoreguidelines

Destructors https://en.cppreference.com/w/cpp/language/destructor

Recommended Practices

- Follow the Principal of Minimalization
  - Best dtor is no zero; avoid specifying whenever possible
  - Only declare dtors when they are required
  - Calling public functions in dtors is a red flag; avoid
  - Avoid unnecessary/redundant work in dtors
- RAII is your friend
  - Wrap raw resources in a class
  - Don’t own more than a single raw resource
  - Don’t store owned pointers in containers

Recommended Practices

Make dtor virtual iff delete Base* could be Derived*

Don’t call virtual functions from a dtor (or ctor)

Don’t let exceptions escape dtors; dtors must not fail

Use explicit dtors cautiously, paired with placement new

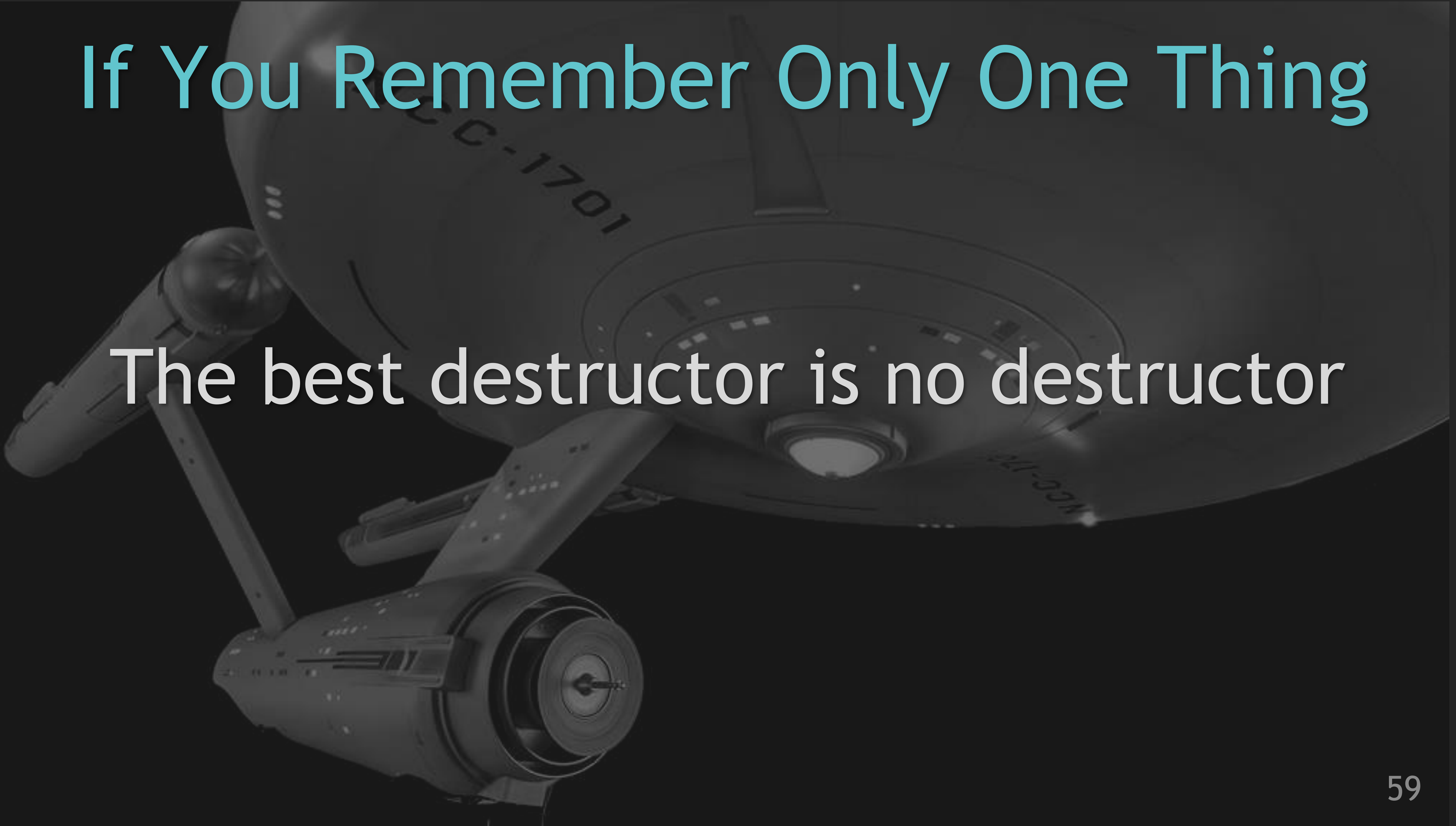
Destructor traits allow important optimizations

Destructors: a great place to check invariants

Optimize common destructors

If You Remember Only One Thing

The best destructor is no destructor



Thanks!

WHEN YOU WORK IN TECH
AND A FAMILY MEMBER SAYS "I HAVE A QUESTION"

Slides and Code

Presentation

https://tinyurl.com/y3ehsaxt

Source Code

https://godbolt.org/z/OUJp7F