std::shared_ptr<T>

(NOT SO) SMART HAMMER FOR EVERY POINTY NAIL

Mateusz Pusz
November 15, 2016

WHY DO WE USE C++?

WHY DO WE USE C++?

Because we love it! It is COOL and the MOST developer-friendly programming language ever!!!

WHY DO WE USE C++?

Because we love it! It is COOL and the MOST developer-friendly programming language ever!!!

Yeah, really???

WHY DO WE REALLY USE C++?

WHY DO WE REALLY USE C++?

C++ language, if used correctly, provides hard to beat performance. C++ developers are in control of what is being used and how they use it.

WHY DO WE REALLY USE C++?

C++ language, if used correctly, provides hard to beat performance. C++ developers are in control of what is being used and how they use it.

C++ motto:
You don't pay for what you don't use.

C++ STANDARD LIBRARY SMART POINTERS

```
template<class T, class Deleter = std::default_delete<T>>
class unique_ptr;
```

C++ STANDARD LIBRARY SMART POINTERS

```cpp
template<class T, class Deleter = std::default_delete<T>>
class unique_ptr;
```

```cpp
template<class T>
class shared_ptr;
```

C++ STANDARD LIBRARY SMART POINTERS

```
template<class T, class Deleter = std::default_delete<T>>
class unique_ptr;
```

```
template<class T>
class shared_ptr;
```

> C++ is the best language for garbage collection principally because it creates less garbage.
>
> *-- Bjarne Stroustrup*

RAII (RESOURCE ACQUISITION IS INITIALIZATION)

- Deterministic *release* of any kind *of resource* (not only memory) *at well-defined time*
- The destructor of local object is (nearly) always called when the object goes out of scope, be it normal program flow or exception

RAII (RESOURCE ACQUISITION IS INITIALIZATION)

- Deterministic *release* of any kind *of resource* (not only memory) *at well-defined time*
- The destructor of local object is (nearly) always called when the object goes out of scope, be it normal program flow or exception

```
class resource {
  // resource handle
public:
  resource(/* args */)
  {
    // obtain ownership of a resource and store the handle
  }
  ~resource()
  {
    // reclaim the resource
  }
};
```

`std::shared_ptr<T>`

- Smart pointer that retains **shared ownership** of an object through a pointer
- Several `shared_ptr` objects **may own the same object**
- The shared object is destroyed and its memory deallocated when the last remaining `shared_ptr` owning that object is either destroyed or assigned another pointer via `operator=` or `reset()`

`std::shared_ptr<T>`

- Smart pointer that retains **shared ownership** of an object through a pointer
- Several **`shared_ptr`** objects **may own the same object**
- The shared object is destroyed and its memory deallocated when the last remaining **`shared_ptr`** owning that object is either destroyed or assigned another pointer via **`operator=`** or **`reset()`**

Great tool! It even releases the developer from thinking about ownership design. Right?

RESOURCE OWNERSHIP

- Ownership of resources is core of software engineering
- Is *shared ownership* a golden bullet... or a tool for lazy developers/architects?

RESOURCE OWNERSHIP

- Ownership of resources is core of software engineering
- Is *shared ownership* a golden bullet... or a tool for lazy developers/architects?

> Do not design your code to use shared ownership without a very good reason.
>
> *-- Google C++ Style Guide*

PROBLEM STATEMENT

```
template<class T> class shared_ptr;
```

PROBLEM STATEMENT

```cpp
template<class T> class shared_ptr;
```

- Really easy tool to use
- Impressive list of features provided through a really simple interface

PROBLEM STATEMENT

```
template<class T> class shared_ptr;
```

- Really easy tool to use
- Impressive list of features provided through a really simple interface

Too often overused by C++ programmers.

SIMPLE USAGE EXAMPLE

```cpp
void foo()
{
  std::unique_ptr<int> ptr{new int{1}};
  // some code using 'ptr'
}
```

```cpp
void foo()
{
  std::shared_ptr<int> ptr{new int{1}};
  // some code using 'ptr'
}
```

What is the difference here?


OMG WHAT HAPPENED?

THESIS

The best way to rise awareness among C++ developers of performance issues caused by **`std::shared_ptr<T>`** is to make them code it by themselves.

OUR GOAL

- *Implement* the most important features of `std::shared_ptr<T>`
- *Understand* what happened in the presented assembly code
- *Learn* some Modern C++ programming techniques
  - How to code them?
  - What is their cost?

OUR GOAL

- *Implement* the most important features of `std::shared_ptr<T>`
- *Understand* what happened in the presented assembly code
- *Learn* some Modern C++ programming techniques
  - How to code them?
  - What is their cost?

There will be some simplifications in our code...


ARE YOU READY FOR
SOMETHING INTENSE?

LET'S TRY TO PROTOTYPE

```
template<typename T>
class shared_ptr;
```

std::shared_ptr<T> - INCORRECT APPROACH

```cpp
template<typename T>
class shared_ptr {
  T* ptr_      = nullptr;
  int counter_ = 0;
public:
  // interface follows...
};
```

```cpp
template<typename T>
class shared_ptr {
  T* ptr_                 = nullptr;
  static int counter_ = 0;
public:
  // interface follows...
};
```

std::shared_ptr<T> - INCORRECT APPROACH

```cpp
template<typename T>
class shared_ptr {
  T* ptr_       = nullptr;
  int counter_ = 0;
public:
  // interface follows...
};
```

```cpp
template<typename T>
class shared_ptr {
  T* ptr_               = nullptr;
  static int counter_ = 0;
public:
  // interface follows...
};
```

- Each object pointing to the same **ptr_** holds a copy of **counter_** that is not updated when other objects modify it

std::shared_ptr<T> - INCORRECT APPROACH

```
template<typename T>
class shared_ptr {
  T* ptr_       = nullptr;
  int counter_ = 0;
public:
  // interface follows...
};
```

- Each object pointing to the same **ptr_** holds a copy of **counter_** that is not updated when other objects modify it

```
template<typename T>
class shared_ptr {
  T* ptr_                = nullptr;
  static int counter_ = 0;
public:
  // interface follows...
};
```

- This time **counter_** gets updated but it is shared between all instances of **shared_ptr<T>** with the same type **T** even if **ptr_** differ

std::shared_ptr<T> - IDEALISTIC APPROACH

```
template<typename T>
class shared_ptr {
  T* ptr_       = nullptr;
  int* counter_ = nullptr;
public:
  // interface follows...
};
```

std::shared_ptr<T> - IDEALISTIC APPROACH

```
template<typename T>
class shared_ptr {
  T* ptr_       = nullptr;
  int* counter_ = nullptr;
public:
  // interface follows...
};
```

- Such an implementation could probably address 90% of all **`std::shared_ptr<T>`** use cases out there

`std::shared_ptr<T>` - IDEALISTIC APPROACH

```
template<typename T>
class shared_ptr {
  T* ptr_       = nullptr;
  int* counter_ = nullptr;
public:
  // interface follows...
};
```

- Such an implementation could probably address 90% of all **`std::shared_ptr<T>`** use cases out there
- Unfortunately it is not the case and the reality is much more complicated...

IS std::shared_ptr<T> THREAD-SAFE?

IS `std::shared_ptr<T>` THREAD-SAFE?

```cpp
template<typename Func>
void runThreads(std::size_t count, Func func)
{








}
```

Helper function to run **`func(id)`** in parallel by **`count`** number of threads.

IS `std::shared_ptr<T>` THREAD-SAFE?

```cpp
template<typename Func>
void runThreads(std::size_t count, Func func)
{
  std::vector<std::future<int>> threads{count};







}
```

Helper function to run **`func(id)`** in parallel by **`count`** number of threads.

IS std::shared_ptr<T> THREAD-SAFE?

```
template<typename Func>
void runThreads(std::size_t count, Func func)
{
  std::vector<std::future<int>> threads{count};
  for(auto& t : threads) {
    t = std::async(std::launch::async, func, count--);
  }



}
```

Helper function to run **`func(id)`** in parallel by **`count`** number of threads.

IS std::shared_ptr<T> THREAD-SAFE?

```
template<typename Func>
void runThreads(std::size_t count, Func func)
{
  std::vector<std::future<int>> threads{count};
  for(auto& t : threads) {
    t = std::async(std::launch::async, func, count--);
  }
  for(auto& t : threads) {
    std::cout << t.get() << "\n";
  }
}
```

Helper function to run **`func(id)`** in parallel by **`count`** number of threads.

IS std::shared_ptr<T> THREAD-SAFE?

```
template<typename Func>
void runThreads(std::size_t count, Func func);

void foo()
{
  auto ptr = std::make_shared<int>(0);
  runThreads(5, [&](int i)
  {
    *ptr = i;
    return *ptr;
  });
}
```

IS std::shared_ptr<T> THREAD-SAFE?

```
template<typename Func>
void runThreads(std::size_t count, Func func);

void foo()
{
  auto ptr = std::make_shared<int>(0);
  runThreads(5, [&](int i)
  {
    *ptr = i;
    return *ptr;
  });
}
```

NOT thread-safe!

IS std::shared_ptr<T> THREAD-SAFE?

```cpp
template<typename Func>
void runThreads(std::size_t count, Func func);

void foo()
{
  auto ptr = std::make_shared<int>(0);
  runThreads(5, [=](int i) mutable
  {
    ptr = std::make_shared<int>(i);
    return *ptr;
  });
}
```

IS std::shared_ptr<T> THREAD-SAFE?

```
template<typename Func>
void runThreads(std::size_t count, Func func);

void foo()
{
  auto ptr = std::make_shared<int>(0);
  runThreads(5, [=](int i) mutable
  {
    ptr = std::make_shared<int>(i);
    return *ptr;
  });
}
```

Thread-safe :-)

IS std::shared_ptr<T> THREAD-SAFE?

```
template<typename Func>
void runThreads(std::size_t count, Func func);

void foo()
{
  auto ptr = std::make_shared<int>(0);
  runThreads(5, [&](int i)
  {
    ptr = std::make_shared<int>(i);
    return *ptr;
  });
}
```

IS std::shared_ptr<T> THREAD-SAFE?

```
template<typename Func>
void runThreads(std::size_t count, Func func);

void foo()
{
  auto ptr = std::make_shared<int>(0);
  runThreads(5, [&](int i)
  {
    ptr = std::make_shared<int>(i);
    return *ptr;
  });
}
```

NOT thread-safe!

IS std::shared_ptr<T> THREAD-SAFE?

```cpp
template<typename Func>
void runThreads(std::size_t count, Func func);

void foo()
{
  auto ptr = std::make_shared<int>(0);
  runThreads(5, [&](int i)
  {
    std::atomic_store(&ptr, std::make_shared<int>(i));
    return *ptr;
  });
}
```

IS std::shared_ptr<T> THREAD-SAFE?

```
template<typename Func>
void runThreads(std::size_t count, Func func);

void foo()
{
  auto ptr = std::make_shared<int>(0);
  runThreads(5, [&](int i)
  {
    std::atomic_store(&ptr, std::make_shared<int>(i));
    return *ptr;
  });
}
```

Thread-safe :-)

IS `std::shared_ptr<T>` THREAD-SAFE?

- *All member functions* can be called by multiple threads *on different instances* of **`std::shared_ptr<T>`** *without additional synchronization* even if these instances share ownership of the same object

IS `std::shared_ptr<T>` THREAD-SAFE?

- *All member functions* can be called by multiple threads *on different instances* of **`std::shared_ptr<T>`** *without additional synchronization* even if these instances share ownership of the same object
- If multiple threads of execution *access the same* **`std::shared_ptr<T>`** *without synchronization* and any of those accesses *uses a non-const member function* then a **data race will occur**

IS `std::shared_ptr<T>` THREAD-SAFE?

- *All member functions* can be called by multiple threads *on different instances* of **`std::shared_ptr<T>`** *without additional synchronization* even if these instances share ownership of the same object
- If multiple threads of execution *access the same* **`std::shared_ptr<T>`** *without synchronization* and any of those accesses *uses a non-const member function* then a **data race will occur**
- The **`std::shared_ptr<T>`** overloads of atomic functions can be used to prevent the data race

IS `std::shared_ptr<T>` THREAD-SAFE?

- *All member functions* can be called by multiple threads *on different instances* of **`std::shared_ptr<T>`** *without additional synchronization* even if these instances share ownership of the same object
- If multiple threads of execution *access the same* **`std::shared_ptr<T>`** *without synchronization* and any of those accesses *uses a non-const member function* then a **data race will occur**
- The **`std::shared_ptr<T>`** overloads of atomic functions can be used to prevent the data race
- **`std::shared_ptr<T>`** does not provide any thread-safety guarantees to the *usage of owned object* and its usage *without additional synchronization* will lead to **data race**

IS `std::shared_ptr<T>` THREAD-SAFE?

- *All member functions* can be called by multiple threads *on different instances* of **`std::shared_ptr<T>`** *without additional synchronization* even if these instances share ownership of the same object
- If multiple threads of execution *access the same* **`std::shared_ptr<T>`** *without synchronization* and any of those accesses *uses a non-const member function* then a **data race will occur**
- The **`std::shared_ptr<T>`** overloads of atomic functions can be used to prevent the data race
- **`std::shared_ptr<T>`** does not provide any thread-safety guarantees to the *usage of owned object* and its usage *without additional synchronization* will lead to **data race**

Atomic operations are used even in single-threaded applications or in cases when sharing of pointers between the threads is not needed.

MANDATORY SYNCHRONIZATION

```cpp
template<typename T>
class shared_ptr {
  T* ptr_                  = nullptr;
  std::atomic_int* counter_ = nullptr;
public:
  // interface follows...
};
```

MANDATORY SYNCHRONIZATION

```
template<typename T>
class shared_ptr {
  T* ptr_                  = nullptr;
  std::atomic_int* counter_ = nullptr;
public:
  // interface follows...
};
```

Remember to not pass **`std::shared_ptr<T>`** by value if not needed.

IDEALISTIC IMPLEMENTATION

```
template<typename T, typename Counter>
class basic_shared_ptr {
  T* ptr_             = nullptr;
  Counter* counter_ = nullptr;
public:
  // interface follows...
};
```

IDEALISTIC IMPLEMENTATION

```
template<typename T, typename Counter>
class basic_shared_ptr {
  T* ptr_           = nullptr;
  Counter* counter_ = nullptr;
public:
  // interface follows...
};
```

```
template<typename T> using shared_ptr      = basic_shared_ptr<T, int>;
template<typename T> using safe_shared_ptr = basic_shared_ptr<T, std::atomic_int>:
```

IDEALISTIC IMPLEMENTATION

```cpp
template<typename T, typename Counter>
class basic_shared_ptr {
  T* ptr_             = nullptr;
  Counter* counter_ = nullptr;
public:
  // interface follows...
};
```

```cpp
template<typename T> using shared_ptr      = basic_shared_ptr<T, int>;
template<typename T> using safe_shared_ptr = basic_shared_ptr<T, std::atomic_int>:
```

```cpp
template<typename T> using byte_shared_ptr = basic_shared_ptr<T, std::int8_t>;
```

IDEALISTIC IMPLEMENTATION

```
template<typename T, typename Counter>
class basic_shared_ptr {
  T* ptr_              = nullptr;
  Counter* counter_ = nullptr;
public:
  // interface follows...
};
```

```
template<typename T> using shared_ptr      = basic_shared_ptr<T, int>;
template<typename T> using safe_shared_ptr = basic_shared_ptr<T, std::atomic_int>:
```

```
template<typename T> using byte_shared_ptr = basic_shared_ptr<T, std::int8_t>;
```

We are not even close to finish **`std::shared_ptr<T>`** prototyping yet...

`std::weak_ptr<T>`

- Smart pointer that holds a non-owning reference to an object that is managed by `std::shared_ptr<T>`
- It must be converted to `std::shared_ptr<T>` in order to access the referenced object

std::weak_ptr<T>

- Smart pointer that holds a non-owning reference to an object that is managed by `std::shared_ptr<T>`
- It must be converted to `std::shared_ptr<T>` in order to access the referenced object
- Is mostly used to
  - break circular references of `std::shared_ptr<T>`
  - implement object store and caching mechanisms

HERB SUTTER'S FAVORITE C++ 10-LINER

```
shared_ptr<widget> get_widget(int id) {
  static map<int, weak_ptr<widget>> cache;
  static mutex m;

  lock_guard<mutex> hold(m);
  auto sp = cache[id].lock();
  if (!sp) cache[id] = sp = load_widget(id);
  return sp;
}
```

HERB SUTTER'S FAVORITE C++ 10-LINER

```
shared_ptr<widget> get_widget(int id) {
  static map<int, weak_ptr<widget>> cache;
  static mutex m;

  lock_guard<mutex> hold(m);
  auto sp = cache[id].lock();
  if (!sp) cache[id] = sp = load_widget(id);
  return sp;
}
```

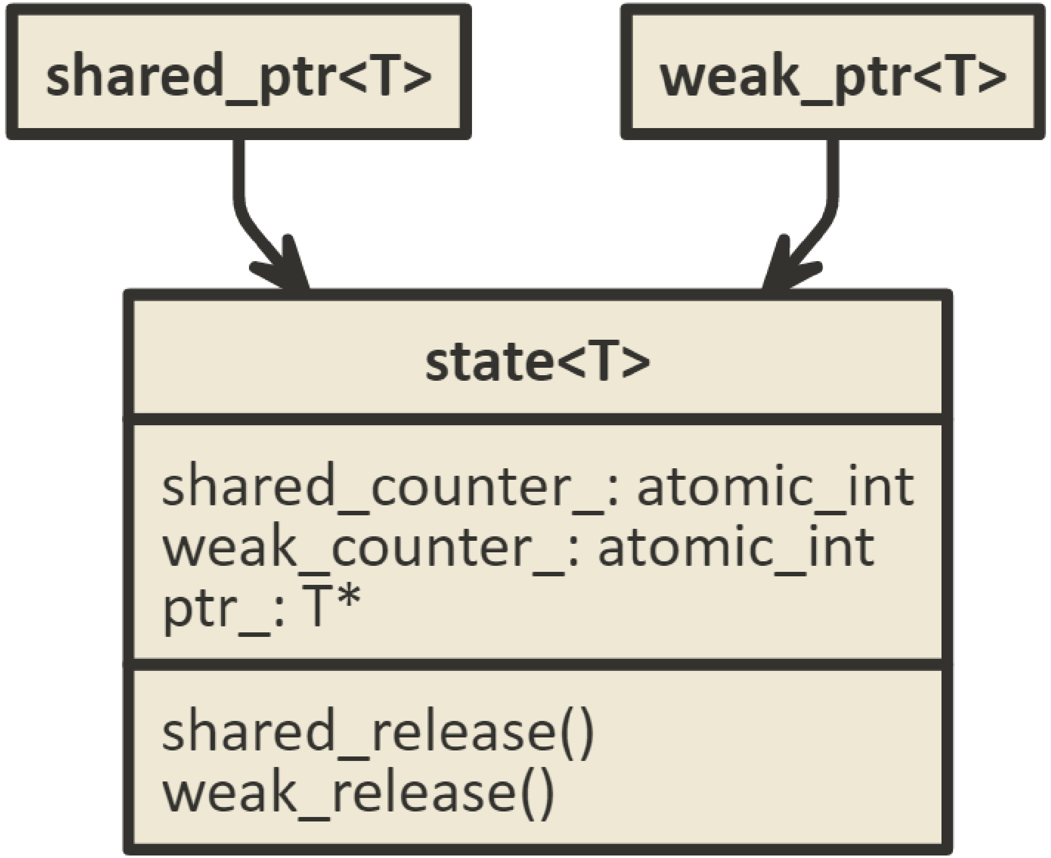
HERB SUTTER'S FAVORITE C++ 10-LINER

```cpp
shared_ptr<widget> get_widget(int id) {
  static map<int, weak_ptr<widget>> cache;
  static mutex m;

  lock_guard<mutex> hold(m);
  auto sp = cache[id].lock();
  if (!sp) cache[id] = sp = load_widget(id);
  return sp;
}
```

HERB SUTTER'S FAVORITE C++ 10-LINER

```
shared_ptr<widget> get_widget(int id) {
  static map<int, weak_ptr<widget>> cache;
  static mutex m;

  lock_guard<mutex> hold(m);
  auto sp = cache[id].lock();
  if (!sp) cache[id] = sp = load_widget(id);
  return sp;
}
```

`std::weak_ptr<T>` SUPPORT



std::weak_ptr<T> SUPPORT

```cpp
template<typename T>
class state {
  std::atomic_int shared_counter_{1};
  std::atomic_int weak_counter_{0};
  T* ptr_ = nullptr;
public:
  // interface follows...
};
```

std::weak_ptr<T> SUPPORT

```
template<typename T>
class state {
  std::atomic_int shared_counter_{1};
  std::atomic_int weak_counter_{0};
  T* ptr_ = nullptr;
public:
  // interface follows...
};
```

```
template<typename T>
class shared_ptr {
  state<T>* state_ = nullptr;
public:
  // interface follows...
};
```

```
template<typename T>
class weak_ptr {
  state<T>* state_ = nullptr;
public:
  // interface follows...
};
```

std::weak_ptr<T> SUPPORT

```
template<typename T>
class state {
  std::atomic_int shared_counter_{1};
  std::atomic_int weak_counter_{0};
  T* ptr_ = nullptr;
public:
  void shared_release() noexcept
  {
    if(--shared_counter_ == 0) {
      delete ptr_;
      if(weak_counter_ == 0)
        delete this;
    }
  }
  // ...
};
```

std::weak_ptr<T> SUPPORT

```
template<typename T>
class state {
  std::atomic_int shared_counter_{1};
  std::atomic_int weak_counter_{0};
  T* ptr_ = nullptr;
public:
  void weak_release() noexcept
  {
    if(--weak_counter_ == 0 && shared_counter_ == 0)
      delete this;
  }
  // ...
};
```

std::weak_ptr<T> SUPPORT

```
template<typename T>
class state {
  std::atomic_int shared_counter_{1};
  std::atomic_int weak_counter_{0};
  T* ptr_ = nullptr;
public:
  void weak_release() noexcept
  {
    if(--weak_counter_ == 0 && shared_counter_ == 0)
      delete this;
  }
  // ...
};
```

Oops, do you see an issue here?

std::weak_ptr<T> SUPPORT

```
template<typename T>
class state {
  std::atomic_int shared_counter_{1};
  std::atomic_int weak_counter_{0};
  T* ptr_ = nullptr;
public:
  void weak_release() noexcept
  {
    if(--weak_counter_ == 0 && shared_counter_ == 0)
      delete this;
  }
  // ...
};
```

Oops, do you see an issue here?

std::weak_ptr<T> SUPPORT

```
template<typename T>
class state {
  std::atomic_int shared_counter_{1};
  std::atomic_int weak_counter_{1};     // #weak + (#shared != 0)
  T* ptr_ = nullptr;
public:
  void weak_release() noexcept
  {
    if(--weak_counter_ == 0)
      delete this;
  }
  // ...
};
```

std::weak_ptr<T> SUPPORT

```
template<typename T>
class state {
  std::atomic_int shared_counter_{1};
  std::atomic_int weak_counter_{1};     // #weak + (#shared != 0)
  T* ptr_ = nullptr;
public:
  void shared_release() noexcept
  {
    if(--shared_counter_ == 0) {
      delete ptr_;
      if(--weak_counter_ == 0)
        delete this;
    }
  }
  // ...
};
```

WHAT ELSE CAN `std::shared_ptr<T>` HIDE?

WHAT ELSE CAN std::shared_ptr<T> HIDE?

Do you remember?

```
.L23:
        mov     rdx, QWORD PTR [rax]
        mov     rdi, rax
        mov     QWORD PTR [rsp+8], rax
        call    [QWORD PTR [rdx+16]]
```

DYNAMIC DISPATCH IN `std::shared_ptr<T>`

Why do we pay for virtual function calls every time we use **`std::shared_ptr<T>`**?

DYNAMIC DISPATCH IN `std::shared_ptr<T>`

Why do we pay for virtual function calls every time we use **`std::shared_ptr<T>`**?

Let's compare with **`std::unique_ptr<T>`** again...

std::unique_ptr<T> CONSTRUCTION

```
struct A { int data; };


void foo()
{
  std::unique_ptr<A> u1;
  std::unique_ptr<A> u2{new A};


}
```

std::unique_ptr<T> CONSTRUCTION

```
struct A { int data; };
my_deleter<A> deleter;

void foo()
{
  std::unique_ptr<A> u1;
  std::unique_ptr<A> u2{new A};
  std::unique_ptr<A, my_deleter<A>> u3{new A};
  std::unique_ptr<A, my_deleter<A>> u4{new A, deleter};
}
```

std::unique_ptr<T> CONSTRUCTION

```cpp
struct A { int data; };
my_deleter<A> deleter;

void foo()
{
  std::unique_ptr<A> u1;
  std::unique_ptr<A> u2{new A};
  std::unique_ptr<A, my_deleter<A>> u3{new A};
  std::unique_ptr<A, my_deleter<A>> u4{new A, deleter};
}
```

All information about deleter stored in a type.

std::shared_ptr<T> CONSTRUCTION

```
struct A { /* ... */ };



void foo()
{
  std::shared_ptr<A> s1;
  std::shared_ptr<A> s2{new A};


}
```

std::shared_ptr<T> CONSTRUCTION

```
struct A { /* ... */ };
my_deleter<A> deleter;


void foo()
{
  std::shared_ptr<A> s1;
  std::shared_ptr<A> s2{new A};
  std::shared_ptr<A> s3{new A, deleter};            // Why no info in the type?

}
```

`std::shared_ptr<T>` CONSTRUCTION

```cpp
struct A { /* ... */ };
my_deleter<A> deleter;
my_allocator<A> allocator;

void foo()
{
  std::shared_ptr<A> s1;
  std::shared_ptr<A> s2{new A};
  std::shared_ptr<A> s3{new A, deleter};            // Why no info in the type?
  std::shared_ptr<A> s4{new A, deleter, allocator}; // Where all those types are stored?
}
```

std::shared_ptr<T> CONSTRUCTION

```
struct A { /* ... */ };
my_deleter<A> deleter;
my_allocator<A> allocator;

void foo()
{
  std::shared_ptr<A> s1;
  std::shared_ptr<A> s2{new A};
  std::shared_ptr<A> s3{new A, deleter};            // Why no info in the type?
  std::shared_ptr<A> s4{new A, deleter, allocator}; // Where all those types are stored?
}
```

Type Erasure in action

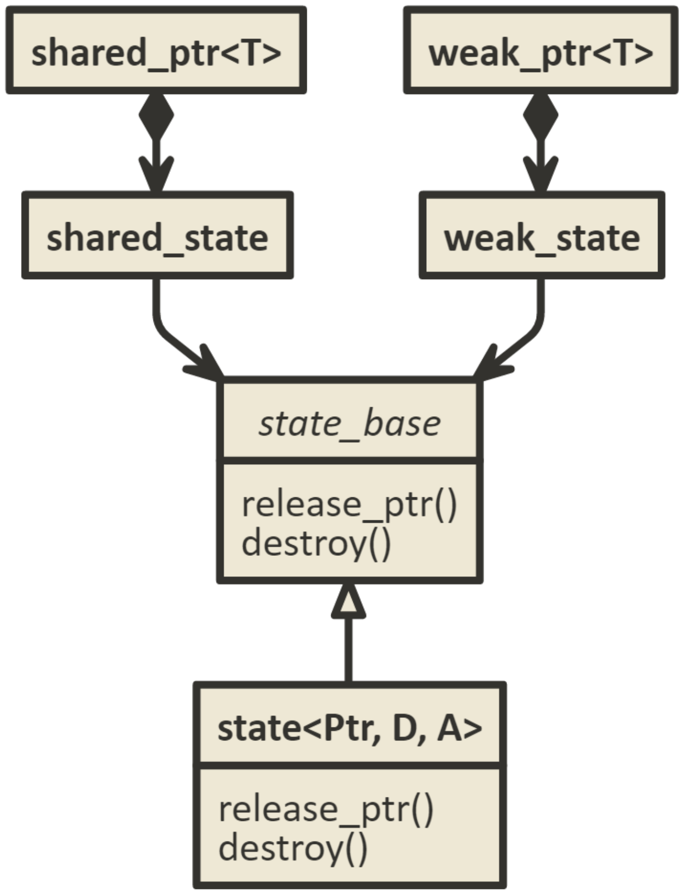
TYPE ERASURE

- Programming technique to represent a variety of concrete types through a single type-neutral interface

TYPE ERASURE

- Programming technique to represent a variety of concrete types through a single type-neutral interface
- In C++ achieved by
  - encapsulating a concrete implementation in a generic wrapper
  - providing virtual methods to access the concrete implementation via a generic interface

-- wikibooks.org



TYPE ERASURE

```
class state_base {
  // interface to handle reference counting and destruction of owned pointer
};
```

TYPE ERASURE

```
class state_base {
  // interface to handle reference counting and destruction of owned pointer
};
```

```
class shared_state {
  state_base* base_ = nullptr;
public:
  // interface follows...
};

template<typename T>
class shared_ptr {
  shared_state state_;
public:
  // interface follows...
};
```

TYPE ERASURE

```cpp
class state_base {
  // interface to handle reference counting and destruction of owned pointer
};
```

```cpp
class shared_state {
  state_base* base_ = nullptr;
public:
  // interface follows...
};

template<typename T>
class shared_ptr {
  shared_state state_;
public:
  // interface follows...
};
```

```cpp
class weak_state {
  state_base* base_ = nullptr;
public:
  // interface follows...
};

template<typename T>
class weak_ptr {
  weak_state state_;
public:
  // interface follows...
};
```

TYPE ERASURE

```
class state_base {
  std::atomic_int shared_counter_{1};
  std::atomic_int weak_counter_{1};          // #weak + (#shared != 0)




public:
  state_base() = default;






  void shared_release() noexcept;
  void weak_release() noexcept

  // ...
};
```

TYPE ERASURE

```
class state_base {
  std::atomic_int shared_counter_{1};
  std::atomic_int weak_counter_{1};          // #weak + (#shared != 0)

  virtual void release_ptr() noexcept = 0;  // releases owned pointer

public:
  state_base() = default;




  void shared_release() noexcept;
  void weak_release() noexcept

  // ...
};
```

TYPE ERASURE

```
class state_base {
  std::atomic_int shared_counter_{1};
  std::atomic_int weak_counter_{1};         // #weak + (#shared != 0)

  virtual void release_ptr() noexcept = 0;  // releases owned pointer

public:
  state_base() = default;
  virtual ~state_base() = default;




  void shared_release() noexcept;
  void weak_release() noexcept

  // ...
};
```

TYPE ERASURE

```
class state_base {
  std::atomic_int shared_counter_{1};
  std::atomic_int weak_counter_{1};          // #weak + (#shared != 0)

  virtual void release_ptr() noexcept = 0;  // releases owned pointer

public:
  state_base() = default;
  virtual ~state_base() = default;

  state_base(const state_base&) = delete;
  state_base& operator=(const state_base&) = delete;

  void shared_release() noexcept;
  void weak_release() noexcept

  // ...
};
```

TYPE ERASURE

```
class state_base { /* ... */ };

template<typename Ptr>
class state final : public state_base {


public:

};
```

TYPE ERASURE

```
class state_base { /* ... */ };

template<typename Ptr>
class state final : public state_base {
  Ptr ptr_;

public:
  explicit state(Ptr ptr) noexcept : ptr_{ptr} {}
};
```

TYPE ERASURE

```
class state_base { /* ... */ };

template<typename Ptr>
class state final : public state_base {
  Ptr ptr_;
  void release_ptr() noexcept override { delete ptr_; }
public:
  explicit state(Ptr ptr) noexcept : ptr_{ptr} {}
};
```

TYPE ERASURE

```
class state_base { /* ... */ };

template<typename Ptr>
class state final : public state_base { /* ... */ };

class shared_state {
  state_base* base_ = nullptr;
public:
  shared_state() = default;







};
```

TYPE ERASURE

```
class state_base { /* ... */ };

template<typename Ptr>
class state final : public state_base { /* ... */ };

class shared_state {
  state_base* base_ = nullptr;
public:
  shared_state() = default;

  template<typename Ptr>
  explicit shared_state(Ptr p) try : base_{new state<Ptr>{p}}
  {}
  catch(...) {
    delete p;
    throw;
  }
};
```

TYPE ERASURE

```
class state_base { /* ... */ };

template<typename Ptr>
class state final : public state_base { /* ... */ };

class shared_state {
  state_base* base_ = nullptr;
public:
  shared_state() = default;

  template<typename Ptr>
  explicit shared_state(Ptr p) try : base_{new state<Ptr>{p}}
  {}
  catch(...) {
    delete p;
    throw;
  }
};
```


HOLD TIGHT!
INTENSE CODING TIME!

TYPE ERASURE

Do you remember?

```cpp
struct A { int data; };
my_deleter<A> deleter;
my_allocator<A> allocator;

std::shared_ptr<A> ptr{new A, deleter, allocator};
```

TYPE ERASURE

Do you remember?

```
struct A { int data; };
my_deleter<A> deleter;
my_allocator<A> allocator;

std::shared_ptr<A> ptr{new A, deleter, allocator};
```

- So far we only erased the owned pointer type `T`

We have to handle Deleter and Allocator types too...

CUSTOM DELETER INTERFACE

```
template<typename T>
struct my_deleter {



  void operator()(T* ptr) const
  {
    delete ptr;
  }
};
```

- Deleter must be *FunctionObject* or *lvalue reference to a FunctionObject* or *lvalue reference to function*, callable with an argument of type **`std::unique_ptr<T, Deleter>::pointer`**

CUSTOM DELETER INTERFACE

```
template<typename T>
struct my_deleter {
  my_deleter() = default;
  template<class U> my_deleter(const my_deleter<U>&) noexcept {}

  void operator()(T* ptr) const
  {
    delete ptr;
  }
};
```

- Deleter must be *FunctionObject* or *lvalue reference to a FunctionObject* or *lvalue reference to function*, callable with an argument of type **`std::unique_ptr<T, Deleter>::pointer`**
- The converting constructor template makes possible the implicit conversion from **`std::unique_ptr<Derived>`** to **`std::unique_ptr<Base>`**

CUSTOM ALLOCATOR INTERFACE

```
template<typename T>
struct my_allocator {
  using value_type = T;









};
```

CUSTOM ALLOCATOR INTERFACE

```
template<typename T>
struct my_allocator {
  using value_type = T;





  T* allocate(std::size_t n)
  {
    return static_cast<T*>(::operator new(n * sizeof(T)));
  }





};
```

CUSTOM ALLOCATOR INTERFACE

```
template<typename T>
struct my_allocator {
  using value_type = T;





  T* allocate(std::size_t n)
  {
    return static_cast<T*>(::operator new(n * sizeof(T)));
  }
  void deallocate(T* p, std::size_t n)
  {
    ::operator delete(p, n);
  }
};
```

CUSTOM ALLOCATOR INTERFACE

```
template<typename T>
struct my_allocator {
  using value_type = T;

  my_allocator() = default;
  template<class U> my_allocator(const my_allocator<U>&) noexcept {}

  T* allocate(std::size_t n)
  {
    return static_cast<T*>(::operator new(n * sizeof(T)));
  }
  void deallocate(T* p, std::size_t n)
  {
    ::operator delete(p, n);
  }
};
```

ALLOCATOR TRAITS

The **`std::allocator_traits<Alloc>`** class template provides the standardized way to access various properties of allocators. It is also a great example of traits-like programming power.

ALLOCATOR TRAITS

The **`std::allocator_traits<Alloc>`** class template provides the standardized way to access various properties of allocators. It is also a great example of traits-like programming power.

```
template<class Alloc>
struct allocator_traits {













};
```

ALLOCATOR TRAITS

The **`std::allocator_traits<Alloc>`** class template provides the standardized way to access various properties of allocators. It is also a great example of traits-like programming power.

```
template<class Alloc>
struct allocator_traits {
  using value_type = Alloc::value_type;
  using pointer = // 'Alloc::pointer;' if present, otherwise 'value_type*;'









};
```

ALLOCATOR TRAITS

The **`std::allocator_traits<Alloc>`** class template provides the standardized way to access various properties of allocators. It is also a great example of traits-like programming power.

```
template<class Alloc>
struct allocator_traits {
  using value_type = Alloc::value_type;
  using pointer = // 'Alloc::pointer;' if present, otherwise 'value_type*;'




  static pointer allocate(Alloc& a, size_type n)              { a.allocate(n); }
  static void deallocate(Alloc& a, pointer p, size_type n) { a.deallocate(p, n); }





};
```

ALLOCATOR TRAITS

The **std::allocator_traits<Alloc>** class template provides the standardized way to access various properties of allocators. It is also a great example of traits-like programming power.

```
template<class Alloc>
struct allocator_traits {
  using value_type = Alloc::value_type;
  using pointer = // 'Alloc::pointer;' if present, otherwise 'value_type*;'
  template<class T>
  using rebind_alloc<T> = // 'Alloc::rebind<T>::other;' if present, otherwise
                          // 'Alloc<T, Args>;' if this Alloc is Alloc<U, Args>

  static pointer allocate(Alloc& a, size_type n)           { a.allocate(n); }
  static void deallocate(Alloc& a, pointer p, size_type n) { a.deallocate(p, n); }




};
```

ALLOCATOR TRAITS

The **std::allocator_traits<Alloc>** class template provides the standardized way to access various properties of allocators. It is also a great example of traits-like programming power.

```
template<class Alloc>
struct allocator_traits {
  using value_type = Alloc::value_type;
  using pointer = // 'Alloc::pointer;' if present, otherwise 'value_type*;'
  template<class T>
  using rebind_alloc<T> = // 'Alloc::rebind<T>::other;' if present, otherwise
                          // 'Alloc<T, Args>;' if this Alloc is Alloc<U, Args>

  static pointer allocate(Alloc& a, size_type n)              { a.allocate(n); }
  static void deallocate(Alloc& a, pointer p, size_type n) { a.deallocate(p, n); }
  template<class T, class... Args>
  static void construct(Alloc& a, T* p, Args&&... args);
  template<class T>
  static void destroy(Alloc& a, T* p);
  // and many more...
};
```

ALLOCATOR TRAITS

```
template<class Alloc>
struct allocator_traits {
  template<class T, class... Args>
  static void construct(Alloc& a, T* p, Args&&... args)
  {
    // 'a.construct(p, std::forward<Args>(args)...);'
    // or if not possible
    // '::new (static_cast<void*>(p)) T(std::forward<Args>(args)...);'
  }
};
```

ALLOCATOR TRAITS

```
template<class Alloc>
struct allocator_traits {
  template<class T, class... Args>
  static void construct(Alloc& a, T* p, Args&&... args)
  {
    // 'a.construct(p, std::forward<Args>(args)...);'
    // or if not possible
    // '::new (static_cast<void*>(p)) T(std::forward<Args>(args)...);'
  }

  template<class T>
  static void destroy(Alloc& a, T* p)
  {
    // 'a.destroy(p);'
    // or if not possible
    // 'p->~T();'
  }
};
```

TYPE ERASURE

```
class state_base { /* ... */ };

template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base {
public:


};
```

TYPE ERASURE

```
class state_base { /* ... */ };

template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base {
public:
  using allocator_type = typename std::allocator_traits<A>::template rebind_alloc<state>;
  // ...
};
```

TYPE ERASURE

```
class state_base { /* ... */ };

template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base { /* ... */ };

class shared_state {
public:
  template<typename Ptr>
  explicit shared_state(Ptr p);      // calls base_{new state<Ptr>{p}}




};
```

TYPE ERASURE

```
class state_base { /* ... */ };

template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base { /* ... */ };

class shared_state {
public:
  template<typename Ptr>
  explicit shared_state(Ptr p);       // calls base_{new state<Ptr>{p}}
  template<typename Ptr, typename D>
  shared_state(Ptr p, D&& d);         // calls base_{new state<Ptr, D>{p, d}}


};
```

TYPE ERASURE

```
class state_base { /* ... */ };

template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base { /* ... */ };

class shared_state {
public:
  template<typename Ptr>
  explicit shared_state(Ptr p);      // calls base_{new state<Ptr>{p}}
  template<typename Ptr, typename D>
  shared_state(Ptr p, D&& d);        // calls base_{new state<Ptr, D>{p, d}}
  template<typename Ptr, typename D, typename A>
  shared_state(Ptr p, D&& d, A&& a);
};
```

TYPE ERASURE

```
template<typename Ptr, typename D, typename A>
shared_state::shared_state(Ptr p, D&& d, A&& a) try
{





}
catch(...) {
  d(p);
  throw;
}
```

TYPE ERASURE

```
template<typename Ptr, typename D, typename A>
shared_state::shared_state(Ptr p, D&& d, A&& a) try
{
  using alloc_type = typename state<Ptr, D, A>::allocator_type;




}
catch(...) {
  d(p);
  throw;
}
```

TYPE ERASURE

```
template<typename Ptr, typename D, typename A>
shared_state::shared_state(Ptr p, D&& d, A&& a) try
{
  using alloc_type = typename state<Ptr, D, A>::allocator_type;
  alloc_type alloc{a};


}
catch(...) {
  d(p);
  throw;
}
```

TYPE ERASURE

```
template<typename Ptr, typename D, typename A>
shared_state::shared_state(Ptr p, D&& d, A&& a) try
{
  using alloc_type = typename state<Ptr, D, A>::allocator_type;
  alloc_type alloc{a};
  base_ = std::allocator_traits<alloc_type>::allocate(alloc, 1);

}
catch(...) {
  d(p);
  throw;
}
```

TYPE ERASURE

```
template<typename Ptr, typename D, typename A>
shared_state::shared_state(Ptr p, D&& d, A&& a) try
{
  using alloc_type = typename state<Ptr, D, A>::allocator_type;
  alloc_type alloc{a};
  base_ = std::allocator_traits<alloc_type>::allocate(alloc, 1);
  std::allocator_traits<alloc_type>::construct(alloc, base_, p, d, std::forward<A>(a));
}  // simplification: not exception-safe
catch(...) {
  d(p);
  throw;
}
```

TYPE ERASURE

```
class state_base {
  std::atomic_int shared_counter_{1};
  std::atomic_int weak_counter_{1};        // #weak + (#shared != 0)

  virtual void release_ptr() noexcept = 0; // releases owned pointer
  virtual void destroy() noexcept = 0;     // releases state

public:
  state_base() = default;
  virtual ~state_base() = default;

  state_base(const state_base&) = delete;
  state_base& operator=(const state_base&) = delete;

  void shared_release() noexcept;
  void weak_release() noexcept

  // ...
};
```

std::weak_ptr<T> SUPPORT RECAP

```
void state_base::shared_release() noexcept
{
  if(--shared_counter_ == 0) {
    delete ptr_;
    if(--weak_counter_ == 0) {
      delete this;
    }
  }
}

void state_base::weak_release() noexcept
{
  if(--weak_counter_ == 0) {
    delete this;
  }
}
```

TYPE ERASURE

```
void state_base::shared_release() noexcept
{
  if(--shared_counter_ == 0) {
    release_ptr();
    if(--weak_counter_ == 0) {
      destroy();
    }
  }
}

void state_base::weak_release() noexcept
{
  if(--weak_counter_ == 0) {
    destroy();
  }
}
```

TYPE ERASURE

```
template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base {
public:
  using allocator_type = typename std::allocator_traits<A>::template rebind_alloc<state>;
  // ...
private:
  Ptr ptr_;
  D& deleter();
  A& allocator();









};
```

TYPE ERASURE

```cpp
template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base {
public:
  using allocator_type = typename std::allocator_traits<A>::template rebind_alloc<state>;
  // ...
private:
  Ptr ptr_;
  D& deleter();
  A& allocator();
  void release_ptr() noexcept override { deleter()(ptr_); }





};
```

TYPE ERASURE

```
template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base {
public:
  using allocator_type = typename std::allocator_traits<A>::template rebind_alloc<state>;
  // ...
private:
  Ptr ptr_;
  D& deleter();
  A& allocator();
  void release_ptr() noexcept override { deleter()(ptr_); }
  void destroy() noexcept override {
    allocator_type alloc{allocator()};


  }
};
```

TYPE ERASURE

```
template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base {
public:
  using allocator_type = typename std::allocator_traits<A>::template rebind_alloc<state>;
  // ...
private:
  Ptr ptr_;
  D& deleter();
  A& allocator();
  void release_ptr() noexcept override { deleter()(ptr_); }
  void destroy() noexcept override {
    allocator_type alloc{allocator()};
    std::allocator_traits<allocator_type>::destroy(alloc, this);

  }
};
```

TYPE ERASURE

```
template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base {
public:
  using allocator_type = typename std::allocator_traits<A>::template rebind_alloc<state>;
  // ...
private:
  Ptr ptr_;
  D& deleter();
  A& allocator();
  void release_ptr() noexcept override { deleter()(ptr_); }
  void destroy() noexcept override {
    allocator_type alloc{allocator()};
    std::allocator_traits<allocator_type>::destroy(alloc, this);
    std::allocator_traits<allocator_type>::deallocate(alloc, this, 1);
  } // simplification: not exception-safe
};
```


INTENSE ENOUGH?

TYPE ERASURE

```
template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base {
public:
  using allocator_type = typename std::allocator_traits<A>::template rebind_alloc<state>;
  // ...
private:
  Ptr ptr_;
  D& deleter();
  A& allocator();
  void release_ptr() noexcept override;
  void destroy() noexcept override;
};
```

- So far we covered Allocator and Deleter interface
- We need to take care about their storage and implement getters...

EMPTY BASE OPTIMIZATION (EBO)

- The *size* of any object or member subobject is required by C++ standard to **be at least 1**
- True even if the type is an empty class type
- However, base class subobjects are not so constrained, and can be completely optimized out from the object layout

EMPTY BASE OPTIMIZATION (EBO)

- The *size* of any object or member subobject is required by C++ standard to **be at least 1**
- True even if the type is an empty class type
- However, base class subobjects are not so constrained, and can be completely optimized out from the object layout

EBO is commonly used by allocator-aware standard library classes (`std::vector`, `std::function`, `std::shared_ptr`, etc) to avoid occupying any additional storage for its allocator member if the allocator is stateless.

EMPTY BASE OPTIMIZATION

```cpp
template<typename T, int Idx, bool UseEbo = !std::is_final_v<T> && std::is_empty_v<T>>
struct ebo_helper;
```

EMPTY BASE OPTIMIZATION

```
template<typename T, int Idx, bool UseEbo = !std::is_final_v<T> && std::is_empty_v<T>>
struct ebo_helper;

template<typename T, int Idx>
struct ebo_helper<T, Idx, true> : private T {



};

template<typename T, int Idx>
struct ebo_helper<T, Idx, false> {



private:
  T t_;
};
```

EMPTY BASE OPTIMIZATION

```
template<typename T, int Idx, bool UseEbo = !std::is_final_v<T> && std::is_empty_v<T>>
struct ebo_helper;

template<typename T, int Idx>
struct ebo_helper<T, Idx, true> : private T {
  template<typename U>
  constexpr explicit ebo_helper(U&& t) : T{std::forward<U>(t)} {}

};

template<typename T, int Idx>
struct ebo_helper<T, Idx, false> {
  template<typename U>
  constexpr explicit ebo_helper(U&& t) : t_{std::forward<U>(t)} {}

private:
  T t_;
};
```

EMPTY BASE OPTIMIZATION

```
template<typename T, int Idx, bool UseEbo = !std::is_final_v<T> && std::is_empty_v<T>>
struct ebo_helper;

template<typename T, int Idx>
struct ebo_helper<T, Idx, true> : private T {
  template<typename U>
  constexpr explicit ebo_helper(U&& t) : T{std::forward<U>(t)} {}
  constexpr T& get() { return *this; }
};

template<typename T, int Idx>
struct ebo_helper<T, Idx, false> {
  template<typename U>
  constexpr explicit ebo_helper(U&& t) : t_{std::forward<U>(t)} {}
  constexpr T& get() { return t_; }
private:
  T t_;
};
```

EMPTY BASE OPTIMIZATION

```
template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base, private ebo_helper<D, 0>, private ebo_helper<A, 1> {


public:






private:



};
```

EMPTY BASE OPTIMIZATION

```
template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base, private ebo_helper<D, 0>, private ebo_helper<A, 1> {
  using DBase = ebo_helper<D, 0>;
  using ABase = ebo_helper<A, 1>;
public:







private:





};
```

EMPTY BASE OPTIMIZATION

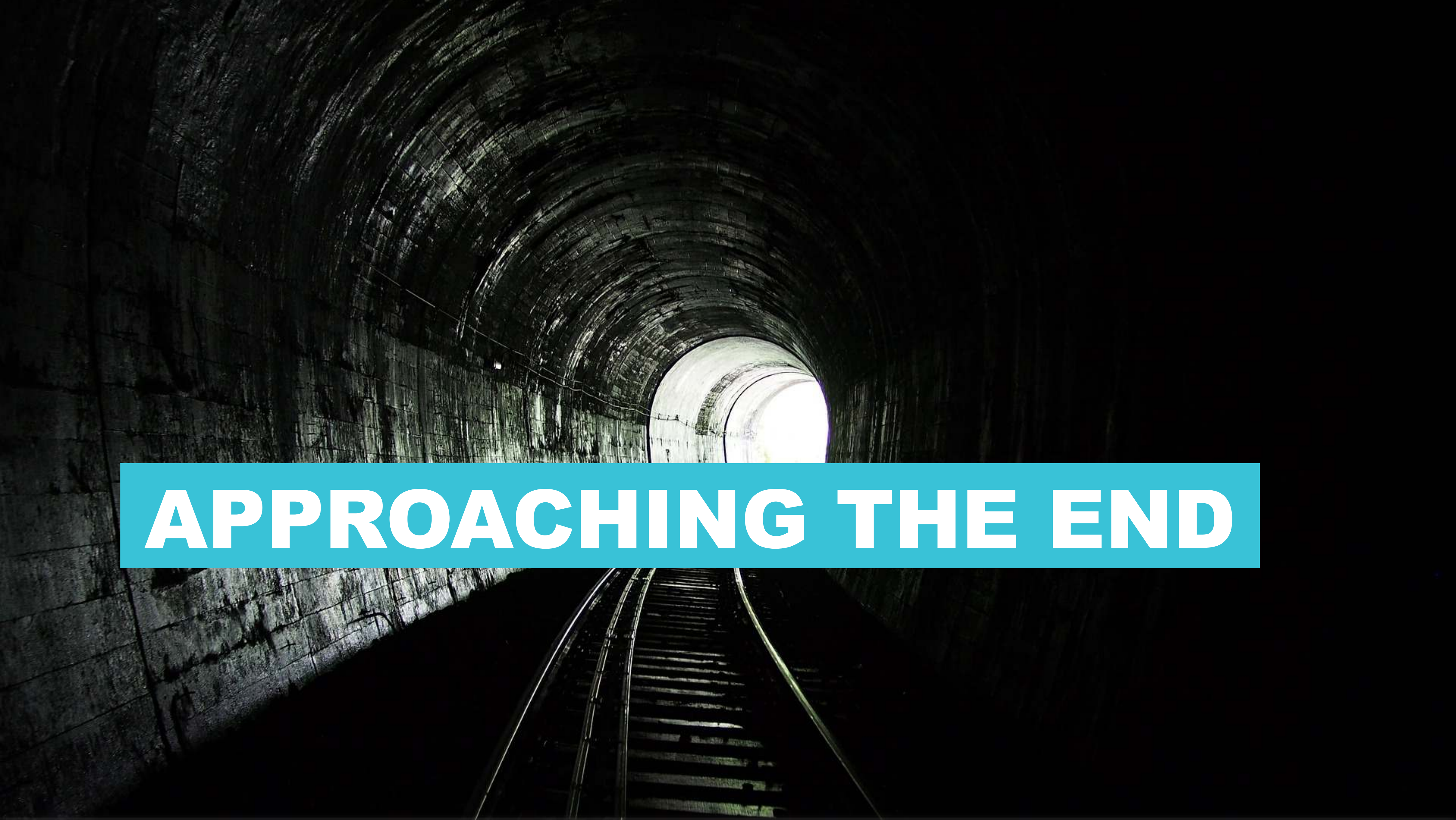
```
template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base, private ebo_helper<D, 0>, private ebo_helper<A, 1> {
  using DBase = ebo_helper<D, 0>;
  using ABase = ebo_helper<A, 1>;
public:



  template<typename DD, typename AA>
  state(Ptr ptr, DD&& d, AA&& a) noexcept :
    DBase{std::forward<DD>(d)}, ABase{std::forward<AA>(a)}, ptr_{ptr} {}
private:




};
```

EMPTY BASE OPTIMIZATION

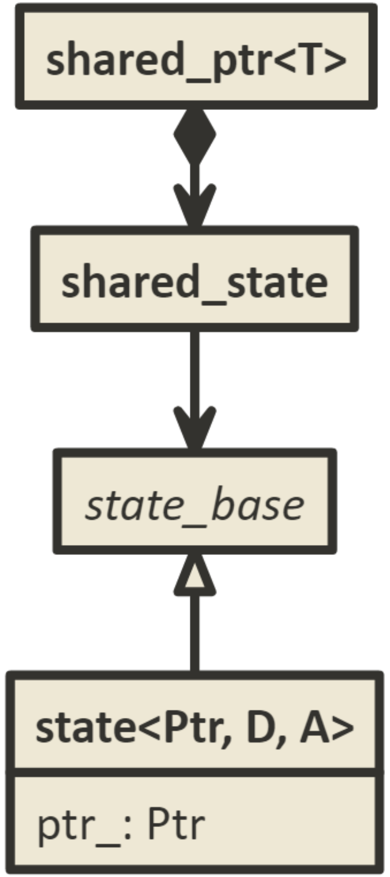
```
template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base, private ebo_helper<D, 0>, private ebo_helper<A, 1> {
  using DBase = ebo_helper<D, 0>;
  using ABase = ebo_helper<A, 1>;
public:
  explicit state(Ptr ptr) noexcept : state{ptr, D{}, A{}} {}
  template<typename DD>
  state(Ptr ptr, DD&& d) noexcept : state{ptr, std::forward<DD>(d), A{}} {}
  template<typename DD, typename AA>
  state(Ptr ptr, DD&& d, AA&& a) noexcept :
    DBase{std::forward<DD>(d)}, ABase{std::forward<AA>(a)}, ptr_{ptr} {}
private:




};
```

EMPTY BASE OPTIMIZATION

```cpp
template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base, private ebo_helper<D, 0>, private ebo_helper<A, 1> {
  using DBase = ebo_helper<D, 0>;
  using ABase = ebo_helper<A, 1>;
public:
  explicit state(Ptr ptr) noexcept : state{ptr, D{}, A{}} {}
  template<typename DD>
  state(Ptr ptr, DD&& d) noexcept : state{ptr, std::forward<DD>(d), A{}} {}
  template<typename DD, typename AA>
  state(Ptr ptr, DD&& d, AA&& a) noexcept :
    DBase{std::forward<DD>(d)}, ABase{std::forward<AA>(a)}, ptr_{ptr} {}
private:
  D& deleter() { return static_cast<DBase&>(*this).get(); }
  A& allocator() { return static_cast<ABase&>(*this).get(); }
  // ...
};
```

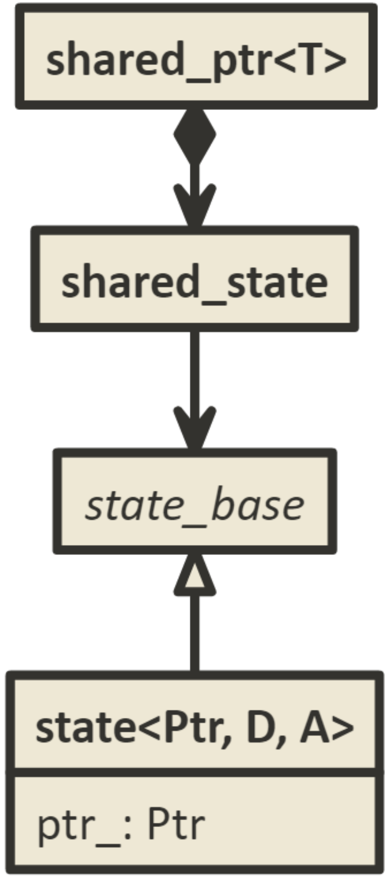

APPROACHING THE END

DESIGN OVERVIEW



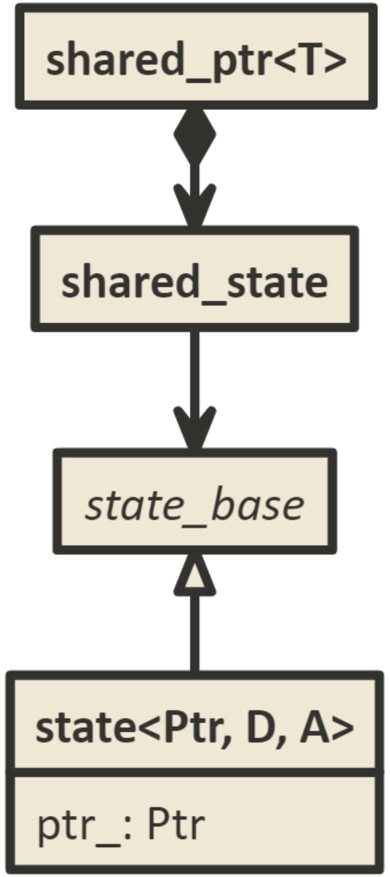
DESIGN OVERVIEW

- Expensive to get pointer instance



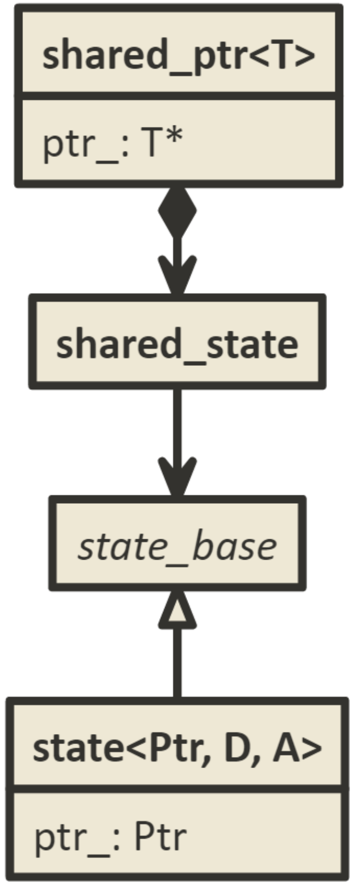
DESIGN OVERVIEW

- Expensive to get pointer instance
- Let's improve performance with the cost of additional memory footprint...



DESIGN OVERVIEW

- 2 pointers stored by each **`std::shared_ptr<T>`** instance
  - *owned pointer* in **`state<Ptr,D,A>`**
  - *stored pointer* in **`std::shared_ptr<T>`**
- **`std::shared_ptr<T>::get()`** returns stored pointer



OWNED AND STORED POINTERS

```cpp
class state_base { /* ... */ };

template<typename Ptr,
         typename D = std::default_delete<std::remove_pointer_t<Ptr>>,
         typename A = std::allocator<std::remove_pointer_t<Ptr>>>
class state final : public state_base {
  Ptr ptr_;     // owned pointer
  // ...
};
```

```cpp
class shared_state {
  state_base* base_ = nullptr;
  // ...
};
```

```cpp
template<typename T>
class shared_ptr {
  T* ptr_ = nullptr;     // stored pointer
  shared_state state_;
  // ...
};
```

ADVANTAGES OF HAVING 2 SEPARATE POINTERS

```cpp
struct A {
  virtual ~A() = default;
};
struct B : A {
  // ...
};

void foo()
{




}
```

```cpp
struct A {
  ~A() = default;
};
struct B : A {
  // ...
};

void foo()
{




}
```

ADVANTAGES OF HAVING 2 SEPARATE POINTERS

```cpp
struct A {
  virtual ~A() = default;
};
struct B : A {
  // ...
};

void foo()
{
  std::shared_ptr<B> s1{new B};      // OK
  std::shared_ptr<A> s2{new B};      // OK
  std::shared_ptr<A> s3{move(s1)};   // OK
}
```

```cpp
struct A {
  ~A() = default;
};
struct B : A {
  // ...
};

void foo()
{
  std::shared_ptr<B> s1{new B};      // OK
  std::shared_ptr<A> s2{new B};      // OK
  std::shared_ptr<A> s3{move(s1)};   // OK
}
```

ADVANTAGES OF HAVING 2 SEPARATE POINTERS

```cpp
struct A {
  virtual ~A() = default;
};
struct B : A {
  // ...
};

void foo()
{
  std::shared_ptr<B> s1{new B};      // OK
  std::shared_ptr<A> s2{new B};      // OK
  std::shared_ptr<A> s3{move(s1)};   // OK

  std::unique_ptr<B> u1{new B};      // OK
  std::unique_ptr<A> u2{new B};      // OK
  std::unique_ptr<A> u3{move(u1)};   // OK
}
```

```cpp
struct A {
  ~A() = default;
};
struct B : A {
  // ...
};

void foo()
{
  std::shared_ptr<B> s1{new B};      // OK
  std::shared_ptr<A> s2{new B};      // OK
  std::shared_ptr<A> s3{move(s1)};   // OK

  std::unique_ptr<B> u1{new B};      // OK
  std::unique_ptr<A> u2{new B};      // WRONG!
  std::unique_ptr<A> u3{move(u1)};   // WRONG!
}
```

WHAT IS THAT?

```
struct A { int data; };


void foo()
{
  std::shared_ptr<A> s1{new A};
  std::shared_ptr<int> s2{s1, &s1->data};


}
```

WHAT IS THAT?

```cpp
struct A { int data; };
void boo(std::shared_ptr<int> ptr);

void foo()
{
  std::shared_ptr<A> s1{new A};
  std::shared_ptr<int> s2{s1, &s1->data};
  s1.reset();
  boo(std::move(s2));
}
```

ALIASING CONSTRUCTOR

```
template<class T>
class shared_ptr {
public:
  template<class Y>
  shared_ptr(const shared_ptr<Y>& r, T *ptr);
  // ...
};
```

ALIASING CONSTRUCTOR

```
template<class T>
class shared_ptr {
public:
  template<class Y>
  shared_ptr(const shared_ptr<Y>& r, T *ptr);
  // ...
};
```

- Allows sharing ownership of one object but storing another one
- Stored pointer is not deleted by **`std::shared_ptr<T>`**

ALIASING CONSTRUCTOR

```
template<class T>
class shared_ptr {
public:
  template<class Y>
  shared_ptr(const shared_ptr<Y>& r, T *ptr);
  // ...
};
```

- Allows sharing ownership of one object but storing another one
- Stored pointer is not deleted by **`std::shared_ptr<T>`**

To avoid the possibility of dangling pointers care must be taken to make sure that the life time of the stored object is at least as long as that of the owned object.

SIDE EFFECTS

EMPTY STD::SHARED_PTR WITH NON-NULL STORED POINTER

```
int i;
std::shared_ptr<int> s(std::shared_ptr<int>{}, &i);
assert(s.use_count() == 0);
assert(s.get() == &i);
```

SIDE EFFECTS

EMPTY STD::SHARED_PTR WITH NON-NULL STORED POINTER

```
int i;
std::shared_ptr<int> s(std::shared_ptr<int>{}, &i);
assert(s.use_count() == 0);
assert(s.get() == &i);
```

NON-EMPTY STD::SHARED_PTR THAT STORES NULLPTR

```
std::shared_ptr<int> s1{new int};
std::shared_ptr<void> s2{s1, nullptr};
s1.reset();
assert(s2.use_count() == 1);
assert(s2.get() == nullptr);
```

std::make_shared<T>()

```
template<class T, class... Args>
shared_ptr<T> make_shared(Args&&... args);
```

std::make_shared<T>()

```
template<class T, class... Args>
shared_ptr<T> make_shared(Args&&... args);
```

```
template<class T, class A, class... Args>
shared_ptr<T> allocate_shared(const A& a, Args&&... args);
```

std::make_shared<T>()

```
template<class T, class... Args>
shared_ptr<T> make_shared(Args&&... args);
```

```
template<class T, class A, class... Args>
shared_ptr<T> allocate_shared(const A& a, Args&&... args);
```

- Constructs an object of type **`T`** and wraps it in a **`std::shared_ptr<T>`** using **`args`** as the parameter list for the constructor of **`T`**

std::make_shared<T>()

```
template<class T, class... Args>
shared_ptr<T> make_shared(Args&&... args);
```

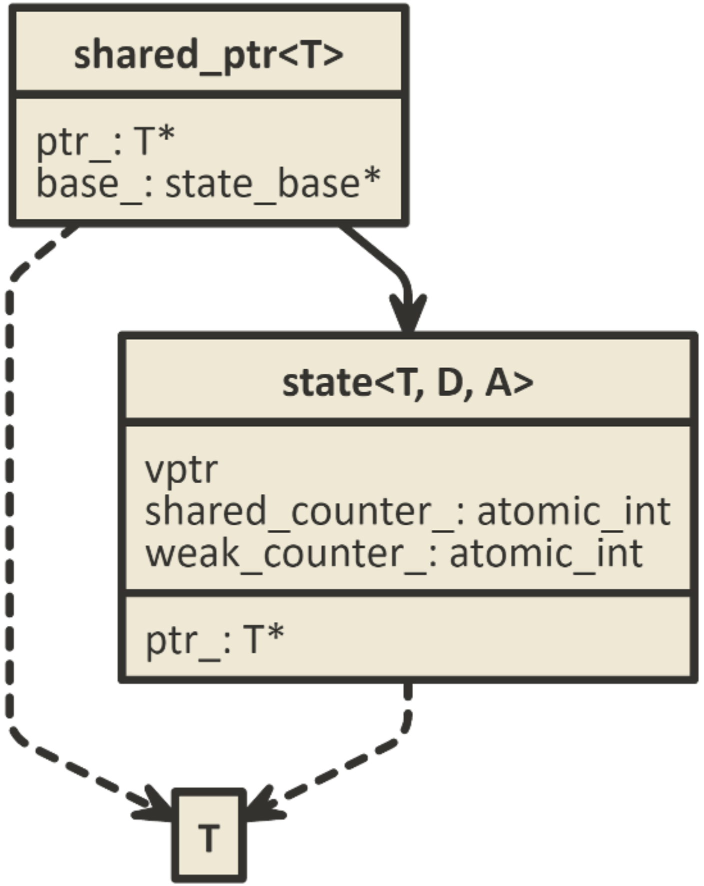
```
template<class T, class A, class... Args>
shared_ptr<T> allocate_shared(const A& a, Args&&... args);
```

- Constructs an object of type **`T`** and wraps it in a **`std::shared_ptr<T>`** using **`args`** as the parameter list for the constructor of **`T`**

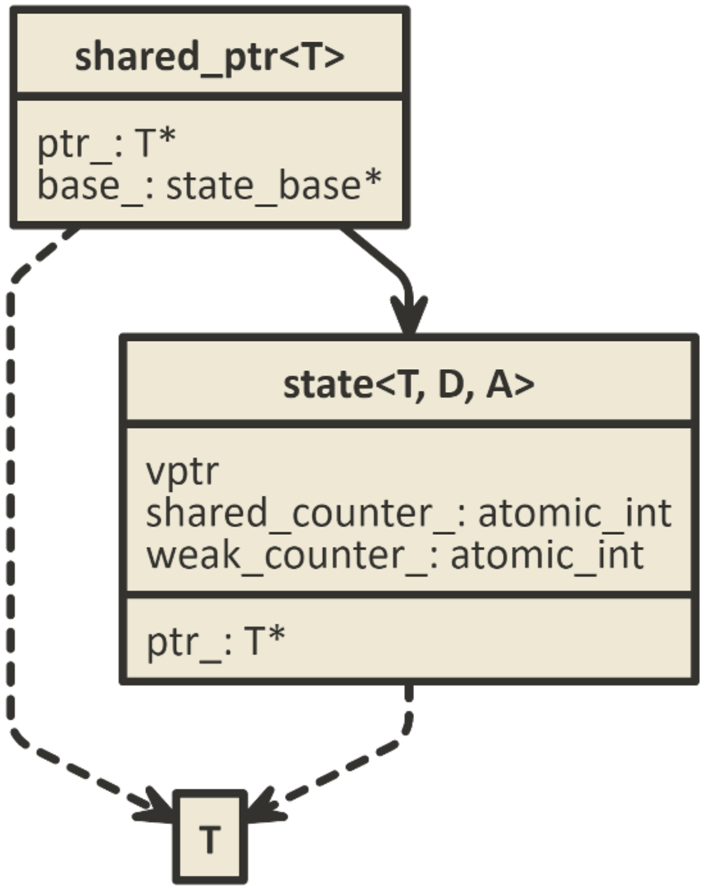
ISO C++ Standard 20.11.2.2.6 (6)

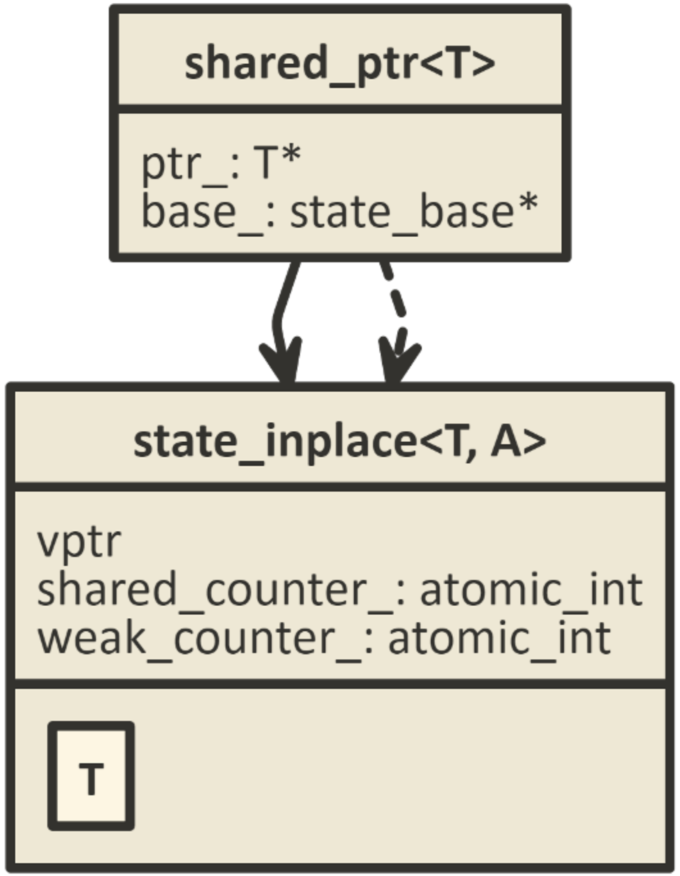
Remarks: Implementations should perform no more than one memory allocation. [Note: This provides efficiency equivalent to an intrusive smart pointer. —end note]

std::make_shared<T>()



std::make_shared<T>()





std::make_shared<T>()

```
class state_base { /* ... */ };

template<typename T, typename A>
class state_inplace final : public state_base, private ebo_helper<A, 0> {




public:





};
```

std::make_shared<T>()

```
class state_base { /* ... */ };

template<typename T, typename A>
class state_inplace final : public state_base, private ebo_helper<A, 0> {
  std::aligned_storage_t<sizeof(T), alignof(T)> buffer_; // owned object




public:




  T* ptr() { return reinterpret_cast<T*>(&buffer_); }
};
```

std::make_shared<T>()

```
class state_base { /* ... */ };

template<typename T, typename A>
class state_inplace final : public state_base, private ebo_helper<A, 0> {
  std::aligned_storage_t<sizeof(T), alignof(T)> buffer_; // owned object

  void release_ptr() noexcept override;                // destructs 'T' in 'ptr()'
  void destroy() noexcept override;                    // destroys '*this'
public:
  template<typename AA, typename... Args>
  state_inplace(const AA& a, Args&&... args) noexcept;   // constructs 'T' in 'ptr()'

  T* ptr() { return reinterpret_cast<T*>(&buffer_); }
};
```

std::make_shared<T>()

```
class state_base { /* ... */ };

template<typename T, typename A>
class state_inplace final : public state_base, private ebo_helper<A, 0> {
  std::aligned_storage_t<sizeof(T), alignof(T)> buffer_; // owned object

  void release_ptr() noexcept override;                 // destructs 'T' in 'ptr()'
  void destroy() noexcept override;                     // destroys '*this'
public:
  template<typename AA, typename... Args>
  state_inplace(const AA& a, Args&&... args) noexcept;  // constructs 'T' in 'ptr()'

  T* ptr() { return reinterpret_cast<T*>(&buffer_); }
};
```

No need to keep the pointer to the owned object (smaller size).

std::make_shared<T>()

```cpp
class state_base { /* ... */ };

template<typename T, typename A>
class state_inplace final : public state_base, private ebo_helper<A, 0> {
  std::aligned_storage_t<sizeof(T), alignof(T)> buffer_; // owned object

  void release_ptr() noexcept override;                 // destructs 'T' in 'ptr()'
  void destroy() noexcept override;                     // destroys '*this'
public:
  template<typename AA, typename... Args>
  state_inplace(const AA& a, Args&&... args) noexcept;   // constructs 'T' in 'ptr()'

  T* ptr() { return reinterpret_cast<T*>(&buffer_); }
};
```

No memory is released until all **shared_ptrs** and **weak_ptrs** go out of scope!


WE MADE IT!
PARALYMPIC GAMES
1065

USAGE ANTIPATTERNS

```
void print(std::shared_ptr<A> a)
{
  std::cout << a << "\n";
}
```

USAGE ANTIPATTERNS

```
void print(std::shared_ptr<A> a)
{
  std::cout << a << "\n";
}
```

Do not pass **`std::shared_ptr<T>`** by value if ownership passing is not required.

USAGE ANTIPATTERNS

```
int* i = new int{123};
int* j = i;
std::shared_ptr<int> p1{i}, p2{j};
```

USAGE ANTIPATTERNS

```
int* i = new int{123};
int* j = i;
std::shared_ptr<int> p1{i}, p2{j};
```

```
auto p1 = std::make_shared<int>(123);
std::shared_ptr<int> p2{p1->get()};
```

USAGE ANTIPATTERNS

```
int* i = new int{123};
int* j = i;
std::shared_ptr<int> p1{i}, p2{j};
```

```
auto p1 = std::make_shared<int>(123);
std::shared_ptr<int> p2{p1->get()};
```

Do not create many instances of **`std::shared_ptr<T>`** for the same allocated pointer.

USAGE ANTIPATTERNS

```
foo(std::shared_ptr<int>{new int{123}}, boo());
```

USAGE ANTIPATTERNS

```
foo(std::shared_ptr<int>{new int{123}}, boo());
```

Be careful when using **`operator new`** to initialize **`std::shared_ptr<T>`**. Prefer **`std::make_shared<T>()`** usage.

USAGE ANTIPATTERNS

```
auto p1 = std::make_shared<int>(123);
int* ptr = p1->get();
delete ptr;
```

USAGE ANTIPATTERNS

```
auto p1 = std::make_shared<int>(123);
int* ptr = p1->get();
delete ptr;
```

Do not use **`delete`** and **`new`** expressions explicitly in your code.

C++ MOTTO

You don't pay for what you don't use.

OUR FEATURE OVERHEAD DEFINITION

Any combination of excess computation time, memory, bandwidth, or other resources that are required to attain a particular goal when particular feature **is not used**.

QUIZ - WHICH FEATURES INTRODUCE OVERHEAD?

FEATURE	OVERHEAD
RAII	
Reference counting	
Thread safety	
Empty Base Optimization	
`std::weak_ptr<T>` support	
Type Erasure	
Custom Allocator/Deleter support	
Aliasing constructor	
`std::make_shared<T>()` support	

QUIZ - WHICH FEATURES INTRODUCE OVERHEAD?

FEATURE	OVERHEAD
RAII	No
Reference counting	
Thread safety	
Empty Base Optimization	
`std::weak_ptr<T>` support	
Type Erasure	
Custom Allocator/Deleter support	
Aliasing constructor	
`std::make_shared<T>()` support	

QUIZ - WHICH FEATURES INTRODUCE OVERHEAD?

FEATURE	OVERHEAD
RAII	No
Reference counting	Yes
Thread safety	
Empty Base Optimization	
`std::weak_ptr<T>` support	
Type Erasure	
Custom Allocator/Deleter support	
Aliasing constructor	
`std::make_shared<T>()` support	

QUIZ - WHICH FEATURES INTRODUCE OVERHEAD?

FEATURE	OVERHEAD
RAII	No
Reference counting	Yes
Thread safety	Yes
Empty Base Optimization	
`std::weak_ptr<T>` support	
Type Erasure	
Custom Allocator/Deleter support	
Aliasing constructor	
`std::make_shared<T>()` support	

QUIZ - WHICH FEATURES INTRODUCE OVERHEAD?

FEATURE	OVERHEAD
RAII	No
Reference counting	Yes
Thread safety	Yes
Empty Base Optimization	No
`std::weak_ptr<T>` support	
Type Erasure	
Custom Allocator/Deleter support	
Aliasing constructor	
`std::make_shared<T>()` support	

QUIZ - WHICH FEATURES INTRODUCE OVERHEAD?

FEATURE	OVERHEAD
RAII	No
Reference counting	Yes
Thread safety	Yes
Empty Base Optimization	No
`std::weak_ptr<T>` support	Yes
Type Erasure	
Custom Allocator/Deleter support	
Aliasing constructor	
`std::make_shared<T>()` support	

QUIZ - WHICH FEATURES INTRODUCE OVERHEAD?

FEATURE	OVERHEAD
RAII	No
Reference counting	Yes
Thread safety	Yes
Empty Base Optimization	No
`std::weak_ptr<T>` support	Yes
Type Erasure	Yes
Custom Allocator/Deleter support	
Aliasing constructor	
`std::make_shared<T>()` support	

QUIZ - WHICH FEATURES INTRODUCE OVERHEAD?

FEATURE	OVERHEAD
RAII	No
Reference counting	Yes
Thread safety	Yes
Empty Base Optimization	No
`std::weak_ptr<T>` support	Yes
Type Erasure	Yes
Custom Allocator/Deleter support	No
Aliasing constructor	
`std::make_shared<T>()` support	

QUIZ - WHICH FEATURES INTRODUCE OVERHEAD?

FEATURE	OVERHEAD
RAII	No
Reference counting	Yes
Thread safety	Yes
Empty Base Optimization	No
`std::weak_ptr<T>` support	Yes
Type Erasure	Yes
Custom Allocator/Deleter support	No
Aliasing constructor	Maybe
`std::make_shared<T>()` support	

QUIZ - WHICH FEATURES INTRODUCE OVERHEAD?

FEATURE	OVERHEAD
RAII	No
Reference counting	Yes
Thread safety	Yes
Empty Base Optimization	No
`std::weak_ptr<T>` support	Yes
Type Erasure	Yes
Custom Allocator/Deleter support	No
Aliasing constructor	Maybe
`std::make_shared<T>()` support	No

WHEN std::shared_ptr<T> IS THE OPTIMAL TOOL FOR YOU?

WHEN `std::shared_ptr<T>` IS THE OPTIMAL TOOL FOR YOU?

1. Shared ownership needed

WHEN `std::shared_ptr<T>` IS THE OPTIMAL TOOL FOR YOU?

1. Shared ownership needed
2. Multithreaded application that references **`std::shared_ptr<T>`** from several threads

WHEN `std::shared_ptr<T>` IS THE OPTIMAL TOOL FOR YOU?

1. Shared ownership needed
2. Multithreaded application that references **`std::shared_ptr<T>`** from several threads
3. **`std::weak_ptr<T>`** usage needed (e.g. to break cycles)

WHEN `std::shared_ptr<T>` IS THE OPTIMAL TOOL FOR YOU?

1. Shared ownership needed
2. Multithreaded application that references **`std::shared_ptr<T>`** from several threads
3. **`std::weak_ptr<T>`** usage needed (e.g. to break cycles)
4. Different allocators/deleters provided at runtime

WHEN `std::shared_ptr<T>` IS THE OPTIMAL TOOL FOR YOU?

1. Shared ownership needed
2. Multithreaded application that references **`std::shared_ptr<T>`** from several threads
3. **`std::weak_ptr<T>`** usage needed (e.g. to break cycles)
4. Different allocators/deleters provided at runtime
5. Aliasing constructor usage

WHEN `std::shared_ptr<T>` IS THE OPTIMAL TOOL FOR YOU?

1. Shared ownership needed
2. Multithreaded application that references **`std::shared_ptr<T>`** from several threads
3. **`std::weak_ptr<T>`** usage needed (e.g. to break cycles)
4. Different allocators/deleters provided at runtime
5. Aliasing constructor usage

If not all above points are met at the same time you probably pay for what you don't use.

QUOTES FROM THE C++ COMMUNITY

When in doubt, prefer **`unique_ptr`** by default.

-- Herb Sutter

QUOTES FROM THE C++ COMMUNITY

`shared_ptr` is best avoided in almost every circumstance when writing software of substantial size.

-- Sean Middleditch

QUOTES FROM THE C++ COMMUNITY

> A shared pointer is as good as a global variable.
>
> -- *Sean Parent*

QUOTES FROM THE C++ COMMUNITY

The Best Designed Library You Shouldn’t Use

-- Ahmed Charles

CAUTION

Programming

is addictive

(and too much fun)


BONUS SLIDES

ADDRESSING EXCEPTION SAFETY ISSUES

```
template<typename Ptr, typename D, typename A>
shared_state::shared_state(Ptr p, D&& d, A&& a) try
{
  using alloc_type = typename state<Ptr, D, A>::allocator_type;
  using alloc_traits = std::allocator_traits<alloc_type>;
  alloc_type alloc{a};
  base_ = alloc_traits::allocate(alloc, 1);
  alloc_traits::construct(alloc, base_, p, d, std::forward<A>(a));
}
catch(...) {
  d(p);
  throw;
}
```

Do you see an issue here?

RAII IN PRACTICE

```
template<typename A>
class alloc_guard {
public:









private:


};
```

RAII IN PRACTICE

```
template<typename A>
class alloc_guard {
public:
  using alloc_traits = std::allocator_traits<A>;
  using pointer = typename alloc_traits::pointer;









private:


};
```

RAII IN PRACTICE

```
template<typename A>
class alloc_guard {
public:
  using alloc_traits = std::allocator_traits<A>;
  using pointer = typename alloc_traits::pointer;

  explicit alloc_guard(A& alloc, pointer ptr) noexcept : alloc_{alloc}, ptr_{ptr} {}







private:
  A& alloc_;
  pointer ptr_;
};
```

RAII IN PRACTICE

```cpp
template<typename A>
class alloc_guard {
public:
  using alloc_traits = std::allocator_traits<A>;
  using pointer = typename alloc_traits::pointer;

  explicit alloc_guard(A& alloc, pointer ptr) noexcept : alloc_{alloc}, ptr_{ptr} {}
  ~alloc_guard() { if(ptr_) alloc_traits::deallocate(alloc_, ptr_, 1);  }




private:
  A& alloc_;
  pointer ptr_;
};
```

RAII IN PRACTICE

```
template<typename A>
class alloc_guard {
public:
  using alloc_traits = std::allocator_traits<A>;
  using pointer = typename alloc_traits::pointer;

  explicit alloc_guard(A& alloc, pointer ptr) noexcept : alloc_{alloc}, ptr_{ptr} {}
  ~alloc_guard() { if(ptr_) alloc_traits::deallocate(alloc_, ptr_, 1);  }

  alloc_guard(const alloc_guard&) = delete;
  alloc_guard& operator=(const alloc_guard&) = delete;


private:
  A& alloc_;
  pointer ptr_;
};
```

RAII IN PRACTICE

```cpp
template<typename A>
class alloc_guard {
public:
  using alloc_traits = std::allocator_traits<A>;
  using pointer = typename alloc_traits::pointer;

  explicit alloc_guard(A& alloc, pointer ptr) noexcept : alloc_{alloc}, ptr_{ptr} {}
  ~alloc_guard() { if(ptr_) alloc_traits::deallocate(alloc_, ptr_, 1);  }

  alloc_guard(const alloc_guard&) = delete;
  alloc_guard& operator=(const alloc_guard&) = delete;

  void release() { ptr_ = nullptr; }
private:
  A& alloc_;
  pointer ptr_;
};
```

TYPE ERASURE

```
template<typename Ptr, typename D, typename A>
shared_state::shared_state(Ptr p, D&& d, A&& a) try
{
  using alloc_type = typename state<Ptr, D, A>::allocator_type;
  using alloc_traits = std::allocator_traits<alloc_type>;
  alloc_type alloc{a};
  auto buffer = alloc_traits::allocate(alloc, 1);

  alloc_traits::construct(alloc, buffer, p, d, std::forward<A>(a));

  base_ = buffer;
}
catch(...) {
  d(p);
  throw;
}
```

TYPE ERASURE

```
template<typename Ptr, typename D, typename A>
shared_state::shared_state(Ptr p, D&& d, A&& a) try
{
  using alloc_type = typename state<Ptr, D, A>::allocator_type;
  using alloc_traits = std::allocator_traits<alloc_type>;
  alloc_type alloc{a};
  auto buffer = alloc_traits::allocate(alloc, 1);
  alloc_guard<alloc_type> guard{alloc, buffer};
  alloc_traits::construct(alloc, buffer, p, d, std::forward<A>(a));
  guard.release();
  base_ = buffer;
}
catch(...) {
  d(p);
  throw;
}
```

BACKUP

std::unique_ptr<T>

```
foo():
        sub     rsp, 8
        mov     edi, 4
        call    operator new(unsigned long)
        mov     esi, 4
        mov     DWORD PTR [rax], 1
        mov     rdi, rax
        add     rsp, 8
        jmp     operator delete(void*, unsigned long)
```

std::unique_ptr<T>

```
foo():
        sub     rsp, 8
        mov     edi, 4
        call    operator new(unsigned long)
        mov     esi, 4
        mov     DWORD PTR [rax], 1
        mov     rdi, rax
        add     rsp, 8
        jmp     operator delete(void*, unsigned long)
```

std::shared_ptr<T>

```
foo():
        push    rbx
        mov     edi, 4
        sub     rsp, 16
        call    operator new(unsigned long)
        mov     edi, 24
        mov     DWORD PTR [rax], 1
        mov     rbx, rax
        call    operator new(unsigned long)
        mov     edx, OFFSET FLAT:__gthrw___pthread_key_create(unsigned int*, void (*)(void*))
        mov     DWORD PTR [rax+8], 1
        mov     DWORD PTR [rax+12], 1
        test    rdx, rdx
        mov     QWORD PTR [rax], OFFSET FLAT:vtable for std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>+16
        mov     QWORD PTR [rax+16], rbx
        je      .L22
        lock sub        DWORD PTR [rax+8], 1
        je      .L23
.L6:
        add     rsp, 16
        pop     rbx
        ret
```

std::shared_ptr<T> - CONTINUE #1

```
.L22:
        mov     DWORD PTR [rax+8], 0
        mov     rdi, rax
        mov     QWORD PTR [rsp+8], rax
        call    std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>::_M_dispose()
        mov     rax, QWORD PTR [rsp+8]
        mov     edx, DWORD PTR [rax+12]
        lea     ecx, [rdx-1]
        mov     DWORD PTR [rax+12], ecx
.L14:
        cmp     edx, 1
        jne     .L6
        mov     rdx, QWORD PTR [rax]
        mov     rdi, rax
        mov     rdx, QWORD PTR [rdx+24]
        add     rsp, 16
        pop     rbx
        jmp     rdx
.L23:
        mov     rdx, QWORD PTR [rax]
        mov     rdi, rax
        mov     QWORD PTR [rsp+8], rax
        call    [QWORD PTR [rdx+16]]
```

std::shared_ptr<T> - CONTINUE #2

```
        mov     rax, QWORD PTR [rsp+8]
        mov     edx, -1
        lock xadd       DWORD PTR [rax+12], edx
        jmp     .L14
        mov     rdi, rax
        call    __cxa_begin_catch
        mov     esi, 4
        mov     rdi, rbx
        call    operator delete(void*, unsigned long)
        call    __cxa_rethrow
        mov     rbx, rax
        call    __cxa_end_catch
        mov     rdi, rbx
        call    _Unwind_Resume
```

std::shared_ptr<T> - CONTINUE #3

```
std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>::~_Sp_counted_ptr():
        rep ret
std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>::_M_get_deleter(std::type_info const&):
        xor     eax, eax
        ret
std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>::_M_dispose():
        mov     rdi, QWORD PTR [rdi+16]
        mov     esi, 4
        jmp     operator delete(void*, unsigned long)
std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>::~_Sp_counted_ptr():
        mov     esi, 24
        jmp     operator delete(void*, unsigned long)
std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>::_M_destroy():
        mov     esi, 24
        jmp     operator delete(void*, unsigned long)
```

std::shared_ptr<T> - CONTINUE #3

```
std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>::~_Sp_counted_ptr():
        rep ret
std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>::_M_get_deleter(std::type_info const&):
        xor     eax, eax
        ret
std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>::_M_dispose():
        mov     rdi, QWORD PTR [rdi+16]
        mov     esi, 4
        jmp     operator delete(void*, unsigned long)
std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>::~_Sp_counted_ptr():
        mov     esi, 24
        jmp     operator delete(void*, unsigned long)
std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>::_M_destroy():
        mov     esi, 24
        jmp     operator delete(void*, unsigned long)
```

```
typeinfo name for std::_Mutex_base<(__gnu_cxx::_Lock_policy)2>:
typeinfo for std::_Mutex_base<(__gnu_cxx::_Lock_policy)2>:
typeinfo name for std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>:
typeinfo for std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>:
typeinfo name for std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>:
typeinfo for std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>:
vtable for std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>:
```