Рецепты использования умных указателей

Алексей Малов

iSpring Solutions

alexey.malov@ispringsolutions.com

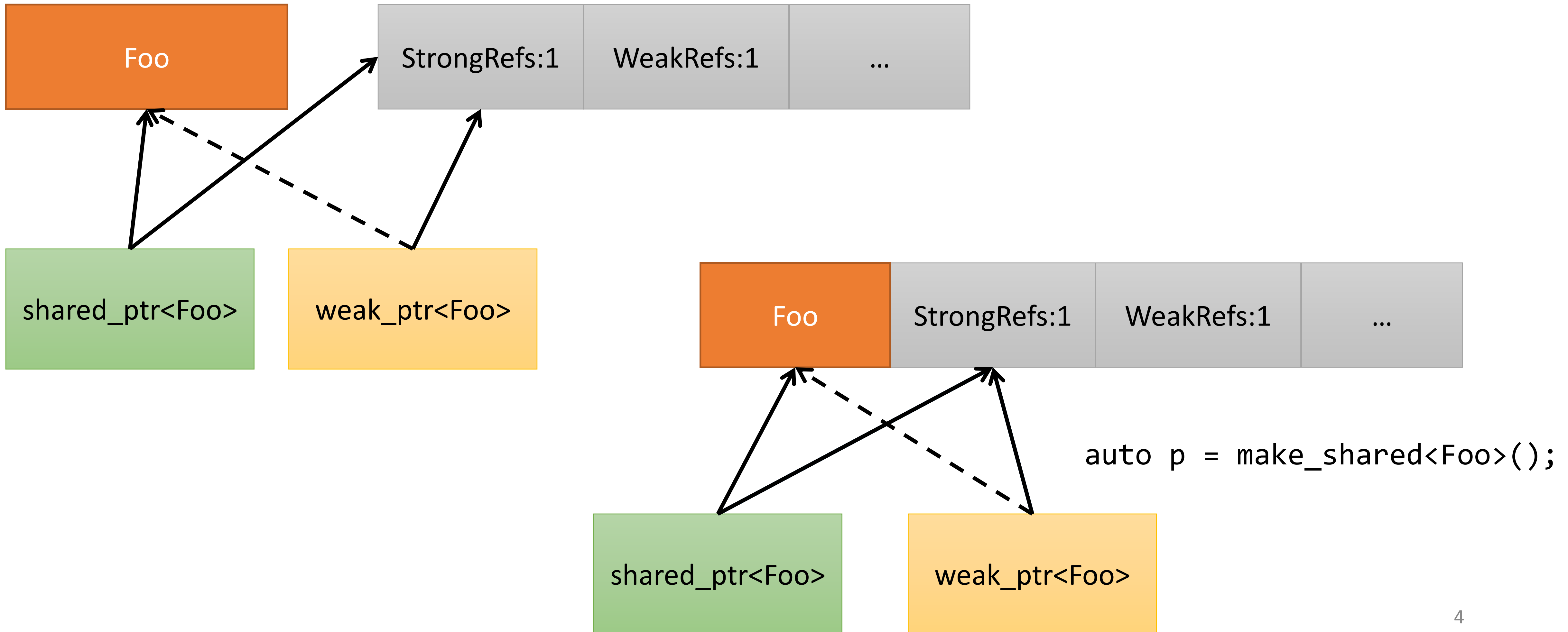
Содержание

- Краткий обзор стандартных умных указателей C++
- Copy on write из shared_ptr
- Построение иерархий владелец-подчинённый и решение связанных с этим проблем
- Идиомы strong self и weak self и их применение

Стандартные умные указатели

- ~~std::auto_ptr~~
- std::unique_ptr
  - Обеспечивает одиночное владение объектом
- std::shared_ptr
  - Совместное владение объектом
- std::weak_ptr
  - Наблюдение за lifetime объекта, которым владеют shared_ptr

Устройство shared_ptr и weak_ptr



enable_shared_from_this

- Наследование от std::enable_shared_from_this позволяет объекту безопасно создать shared_ptr/weak_ptr на самого себя
  - Методы shared_from_this(), weak_from_this()
- Нельзя вызывать вызов shared_from_this() и weak_from_this()
  - В конструкторе и деструкторе
  - У объекта, не обёрнутого в shared_ptr
- Если нужно вызывать в процессе инициализации объекта, используется двухэтапная инициализация
  - static create() + приватный конструктор

```cpp
struct Foo : enable_shared_from_this<Foo>
{
  static shared_ptr<Foo> Create(int arg) {
    shared_ptr<Foo> foo{new Foo(arg)};
    foo->Init();
    return foo;
  }
  …
private:
  void Init() {
    auto self = shared_from_this();
    …
  }
  explicit Foo(int arg);
};
```

2 выделения памяти

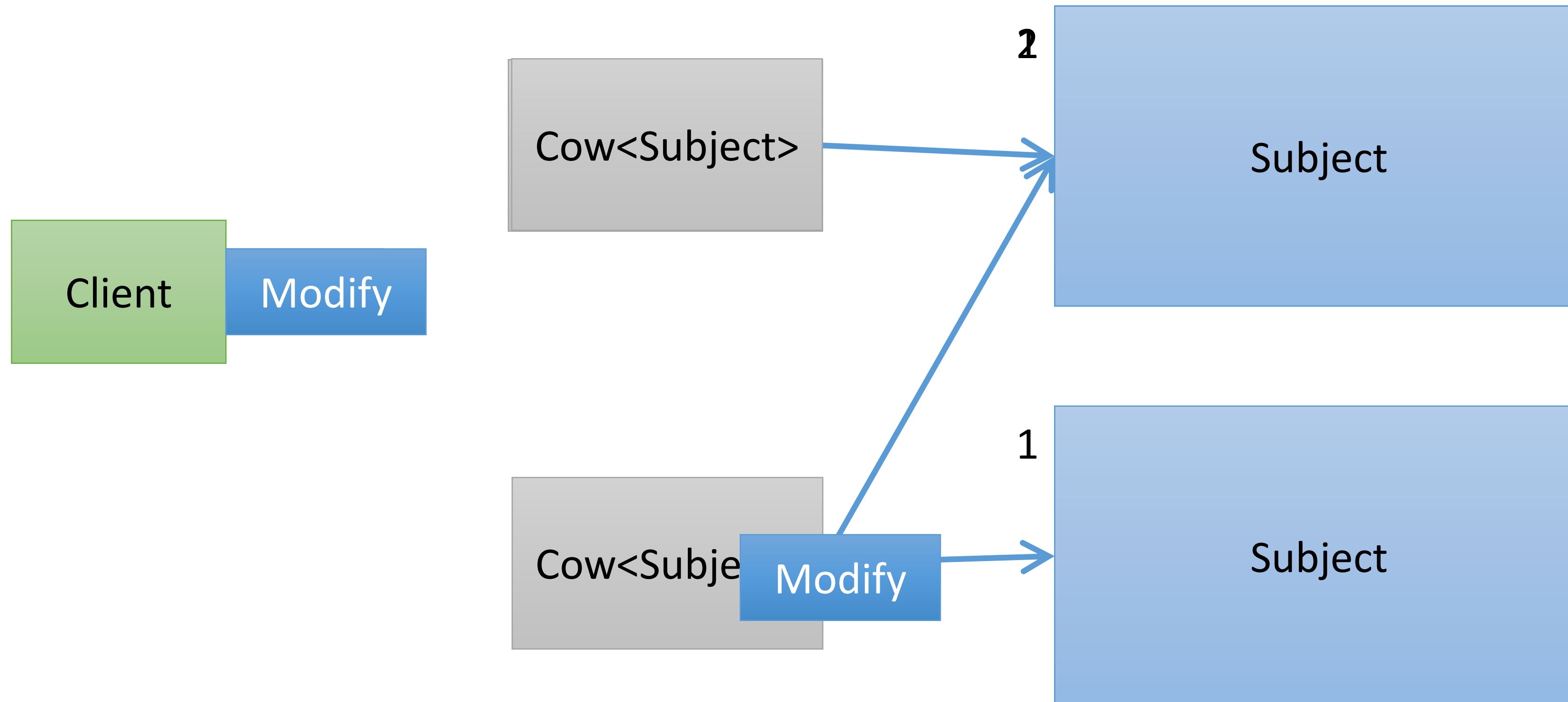
```cpp
struct Foo : enable_shared_from_this<Foo>
{
  static shared_ptr<Foo> Create(int arg) {
    struct Wrapper : Foo {
      Wrapper(int arg): Foo(arg) {}
    };
    auto foo = make_shared<Wrapper>(arg);
    foo->Init();
    return foo;
  }
  …
private:
  void Init() {
    auto self = shared_from_this();
    …
  }
  explicit Foo(int arg);
};
```

Foo не должен быть final

1 выделение памяти

Copy on Write

Copy on Write



```cpp
template <typename T> class Cow {
public:









private:
```

Разные объекты Cow совместно владеют одним объектом T

```cpp
    std::shared_ptr<T> m_shared;
};
```

```cpp
Cow<Subject> subj1;
auto subj2 = subj1;
```

```cpp
template <typename T> class Cow {
public:
  T const& operator*() const { return *m_shared; }

  T const* operator->() const { return m_shared.get(); }
```

```cpp
private:


  std::shared_ptr<T> m_shared;
};
```

```cpp
class Subject {
  int m_x = 0;
public:
  int GetValue() const;
  void SetValue(int x);
};


Cow<Subject> subj1;
std::cout << subj1->GetValue();
```

```
template <typename T> class Cow {
public:
  T const& operator*() const { return *m_shared; }

  T const* operator->() const { return m_shared.get(); }

  T& Write() {
    EnsureUnique();
    return *m_shared;
  }
private:
  void EnsureUnique() {
    if (m_shared.use_count() > 1)
      m_shared = CopyClass::Copy(*m_shared);
  }

  std::shared_ptr<T> m_shared;
};
```

При **совместном** доступе на запись создаём копию экземпляра

```
Cow<Subject> subj1;
subj1.Write().SetValue(1);
```

Варианты стратегий копирования

```
template <typename T> class Cow {
  template <typename U> struct CopyConstr {
    static auto Copy(U const& other) {
      return std::make_shared<U>(other);
    }
  };
  template <typename U> struct CloneConstr {
    static auto Copy(U const& other) {
      return other.Clone();
    }
  };


};
```

```
template <typename T> class Cow {
  template <typename U> struct CopyConstr {
    static auto Copy(U const& other) {
      return std::make_shared<U>(other);
    }
  };
  template <typename U> struct CloneConstr {
    static auto Copy(U const& other) {
      return other.Clone();
    }
  };
  using CopyClass =
    typename std::conditional_t<!std::is_abstract_v<T> &&
                                  std::is_copy_constructible_v<T>,
                                CopyConstr<T>,
                                CloneConstr<T>>;
  …
};
```

Клонирование полиморфных объектов

```
struct Shape {
  virtual ~Shape() = default;
  virtual shared_ptr<Shape> Clone() const = 0;
};

struct Circle : public Shape {
  Circle(double r): m_radius(r){}

  shared_ptr<Shape> Clone() const override {
    return make_shared<Circle>(*this);
  }

private:
  double m_radius;
};
```

```
Cow<Circle> circle(100.0);
Cow<Shape> shape1{ circle };
Cow<Shape> shape2{ shape1 };
```

```cpp
template <typename T> class Cow {
public:
  template <typename... Args, typename = std::enable_if_t<!std::is_abstract_v<T>>>
  Cow(Args&&... args): m_shared(std::make_shared<T>(std::forward<Args>(args)...)) { }

  template <typename U, typename Deleter>
  Cow(std::unique_ptr<U, Deleter> pUniqueObj): m_shared(std::move(pUniqueObj)) { }

  template <class U>  friend class Cow;
  template <typename U> Cow(Cow<U>& rhs): m_shared(rhs.m_shared) { }
  template <typename U> Cow(const Cow<U>& rhs): m_shared(rhs.m_shared) { }

  Cow(Cow&& rhs) = default;
  Cow(Cow const& rhs) = default;

  template <typename U> Cow& operator=(Cow<U>& rhs) {
    m_shared = rhs.m_shared;
    return *this;
  }

  Cow& operator=(Cow&& rhs) = default;
  Cow& operator=(Cow const& rhs) = default;
};
```

```cpp
Cow<vector<int>> vec(100000u, 42);
Cow<Base> base(make_unique<Derived>());
```

```
template <typename T> class Cow
{
public:
  struct WriteProxy {
    T* operator->() { return m_p; }
  private:
    friend class Cow;
    WriteProxy(T* p): m_p(p) {}
    WriteProxy(WriteProxy const&) = default;
    WriteProxy& operator=(WriteProxy const&) = delete;
    T* m_p;
  };



};
```

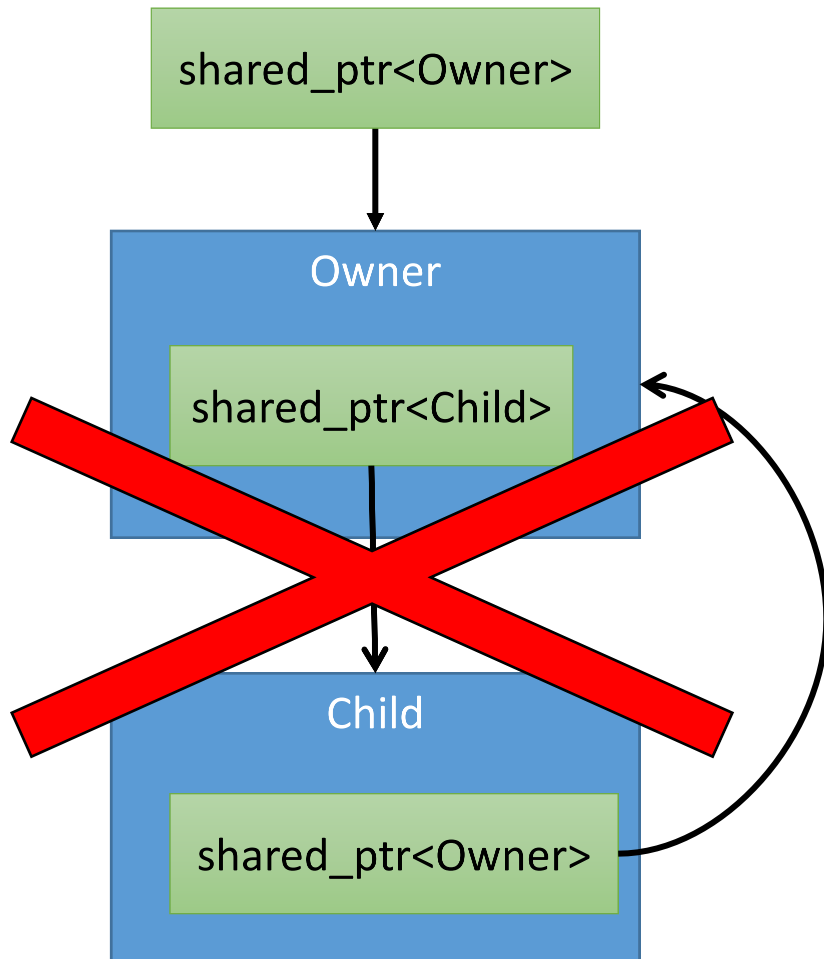
```
template <typename T> class Cow
{
public:
  struct WriteProxy {
    T* operator->() { return m_p; }
  private:
    friend class Cow;
    WriteProxy(WriteProxy const&) = default;
    WriteProxy& operator=(WriteProxy const&) = delete;
    WriteProxy(T* p): m_p(p) {}
    T* m_p;
  };

  WriteProxy operator--(int) {
    EnsureUnique();
    return { m_shared.get() };
  }
};
```

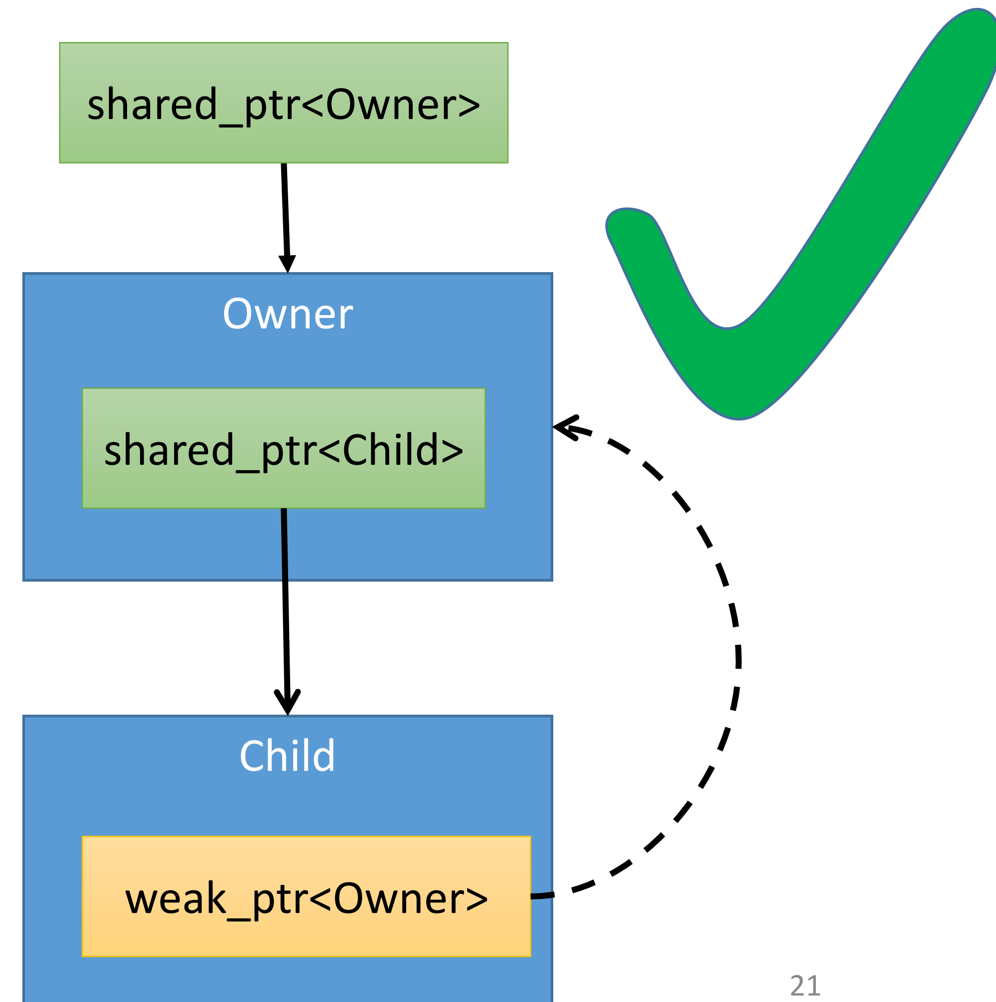
```
template <typename T> class Cow
{
public:
  struct WriteProxy {
    T* operator->() { return m_p; }
  private:
    friend class Cow;
    WriteProxy(WriteProxy const&) = default;
    WriteProxy& operator=(WriteProxy const&) = delete;
    WriteProxy(T* p): m_p(p) {}
    T* m_p;
  };

  WriteProxy operator--(int) {
    EnsureUnique();
    return { m_shared.get() };
  }
};
```

```
Cow<vector<int>> v;
cout << v->size(); // read
v--->push_back(42); // write
```

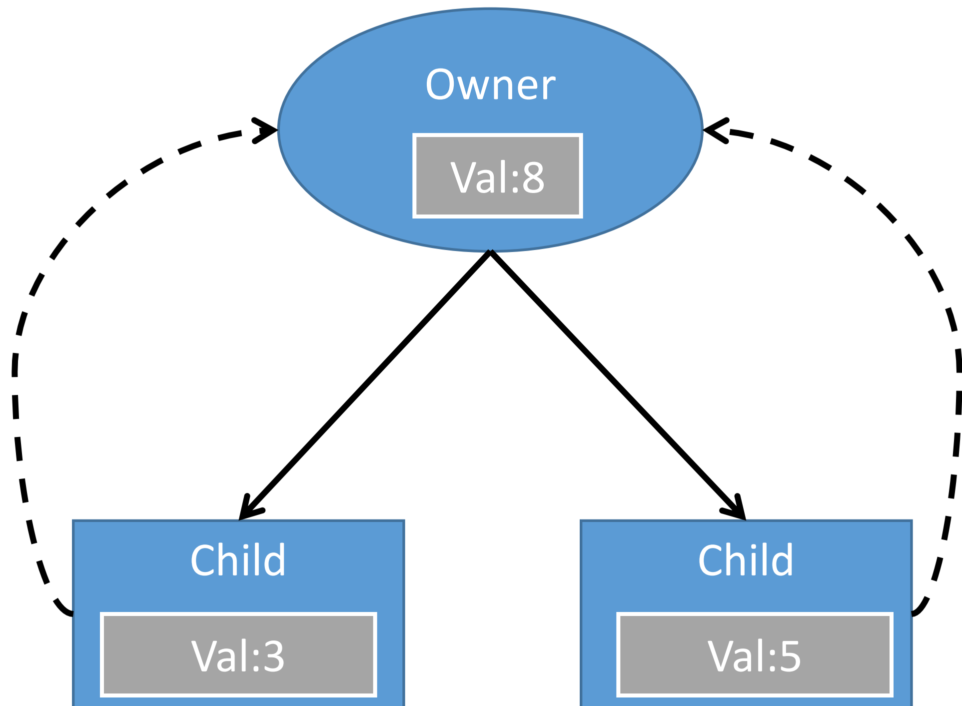
Совместное использование shared_ptr и weak_ptr

Построение иерархий «владелец-подчинённый»





```cpp
struct IOwner
{
  virtual ~IOwner() = default;
  virtual void Invalidate() = 0;
};
```




```cpp
struct Child
{
  Child(int value, weak_ptr<IOwner> owner)
    : m_owner(move(owner)), m_value(value)
  {}

  int GetValue() const {
    return m_value;
  }

  void SetValue(int value) {
    if (value != m_value) {
      m_value = value;
      if (auto owner = m_owner.lock()) {
        owner->Invalidate();
      }
    }
  }
private:
  weak_ptr<IOwner> m_owner;
  int m_value;
};
```

```cpp
struct Owner : IOwner, enable_shared_from_this<Owner> {
  shared_ptr<Child> AddChild(int val) {
    m_children.emplace_back(make_shared<Child>(val, weak_from_this()));
    m_value.reset();
    return m_children.back();
  }

  int GetValue() const {
    if (!m_value) {
      int sum = 0;
      for (auto&& child : m_children) {
        sum += child->GetValue();
      }
      m_value = sum;
    }
    return *m_value;
  }
private:
  void Invalidate() override {
    m_value.reset();
  }
  mutable optional<int> m_value;
  vector<shared_ptr<Child>> m_children;
};
```

```cpp
auto owner = make_shared<Owner>();
auto child1 = owner->AddChild(3);
auto child2 = owner->AddChild(2);

cout << "Old value: " << owner->GetValue()
     << "\n";

s2->SetValue(42);
cout << "New value:" << owner->GetValue()
     << "\n";

owner.reset()
s1->SetValue(10);
```

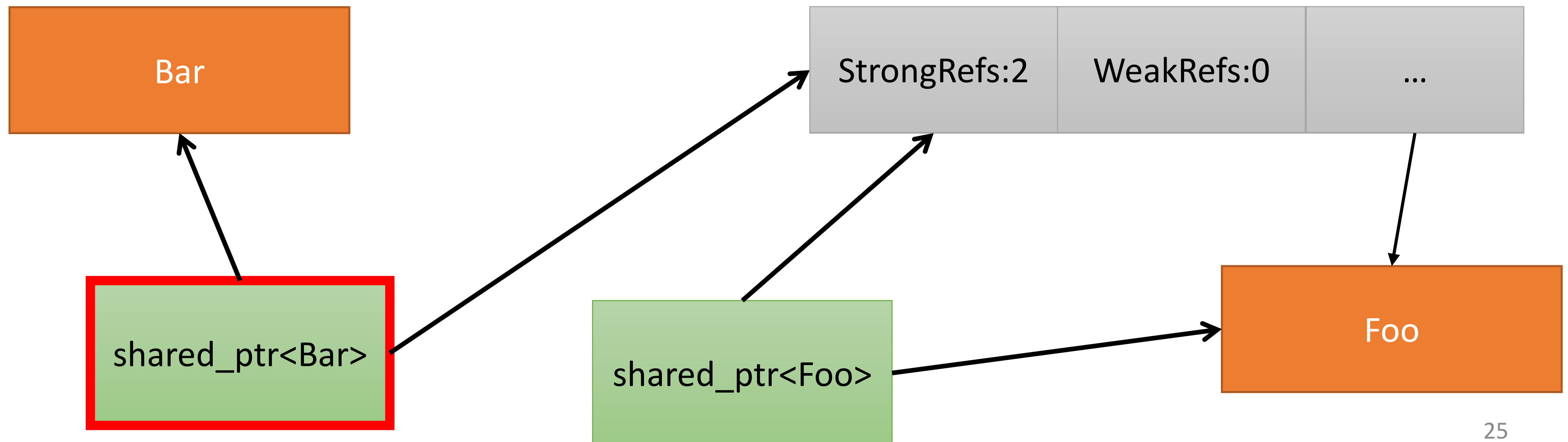
```
Old value: 5
New value: 45
```

Текущие проблемы

- В отсутствие сильных ссылок Владелец будет удалён
  - Клиенту останутся доступны ссылки на дочерние объекты
- Иногда это нежелательно
  - Был «человек», взяли его за «руку», а самого «человека» отпустили
  - У нас осталась только «рука»
- Простое решение: shared_ptr aliasing constructor

shared_ptr aliasing constructor

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



Невладеющий shared_ptr

```
template <typename T>
shared_ptr<T> MakeUnowningSP(T& value)
{
  return { shared_ptr<T>(), &value };
}

struct Bar{};
void Foo(shared_ptr<Bar> bar);
```

```
Bar bar;
Foo(MakeUnowningSP(bar));

auto sp = MakeUnowningSP(bar);
weak_ptr weakBar(sp);
assert(weakBar.expired());
```

UniversalPtr*

- Указатель, инкапсулирующий семантику владения ресурсом
  - No ownership
  - Unique/shared ownership
- Применимость – альтернатива передаче по ссылке или указателю
  - Объекту нужен доступ к ресурсу, но не нужны права владения
  - Lifetime объекта не превышает lifetime ресурса
  - Клиент может передать объекту ресурс вместе с правами владения
- Накладные расходы
  - Обертка над shared_ptr с соответствующими затратами на конструирование/копирование/присваивание

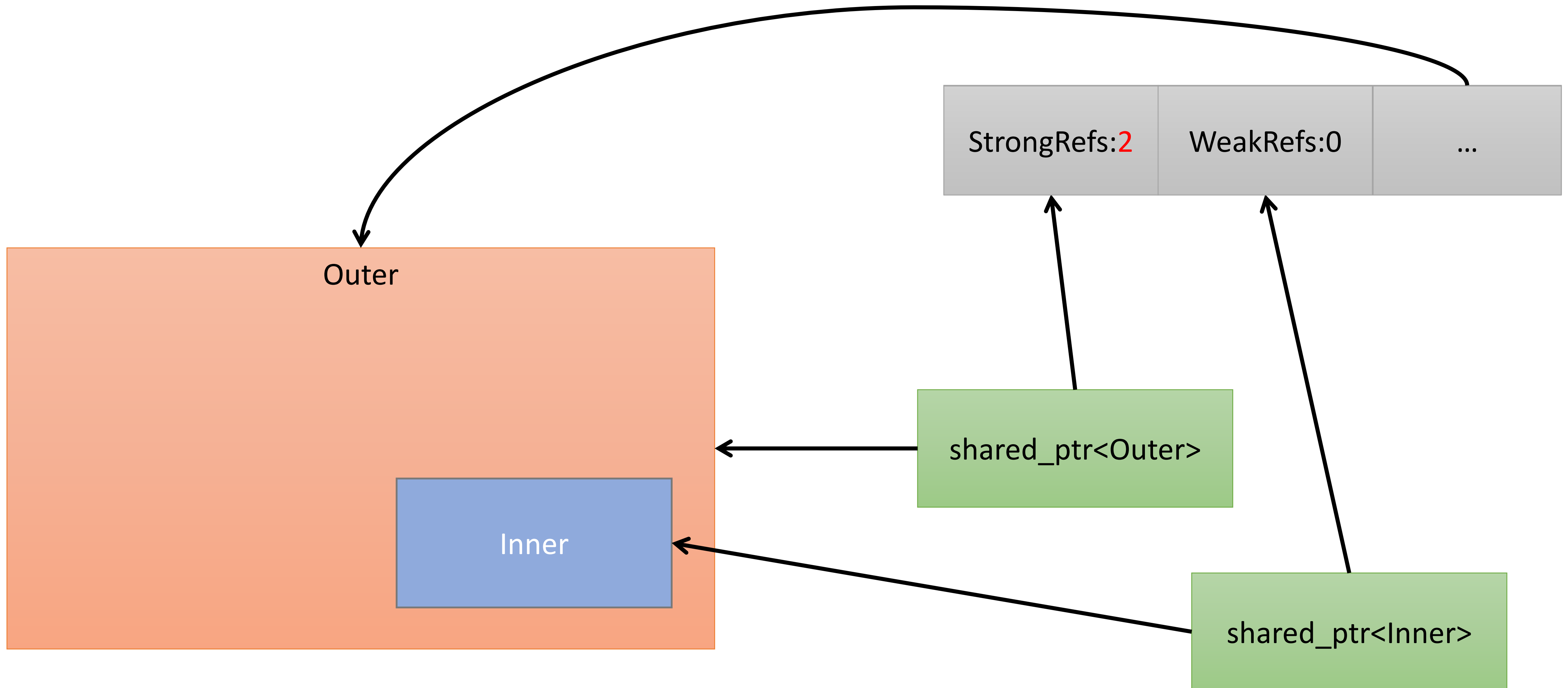
* Имя, наверное, можно придумать удачнее

```cpp
template <typename T> struct UniversalPtr {
  constexpr UniversalPtr() noexcept = default;
  constexpr UniversalPtr(nullptr_t) noexcept {}

  template <typename U> UniversalPtr(U& r) noexcept;
  template <typename U> UniversalPtr(U* p) noexcept;
  template <typename U, typename D> UniversalPtr(std::unique_ptr<U, D>&& p);
  template <typename U> UniversalPtr(const std::shared_ptr<U>& p) noexcept;

  explicit operator bool()const noexcept;
  T& operator*()const noexcept;
  T* operator->()const noexcept;
  T* get()const noexcept;
private:
  std::shared_ptr<T> m_ptr;
};
```

Полный фрагмент кода тут:

github.com/alexey-malov/cpprussia2019-samples/blob/master/sp_aliasing_constructor/UniversalPtr.h

Владеющая ссылка на вложенный объект

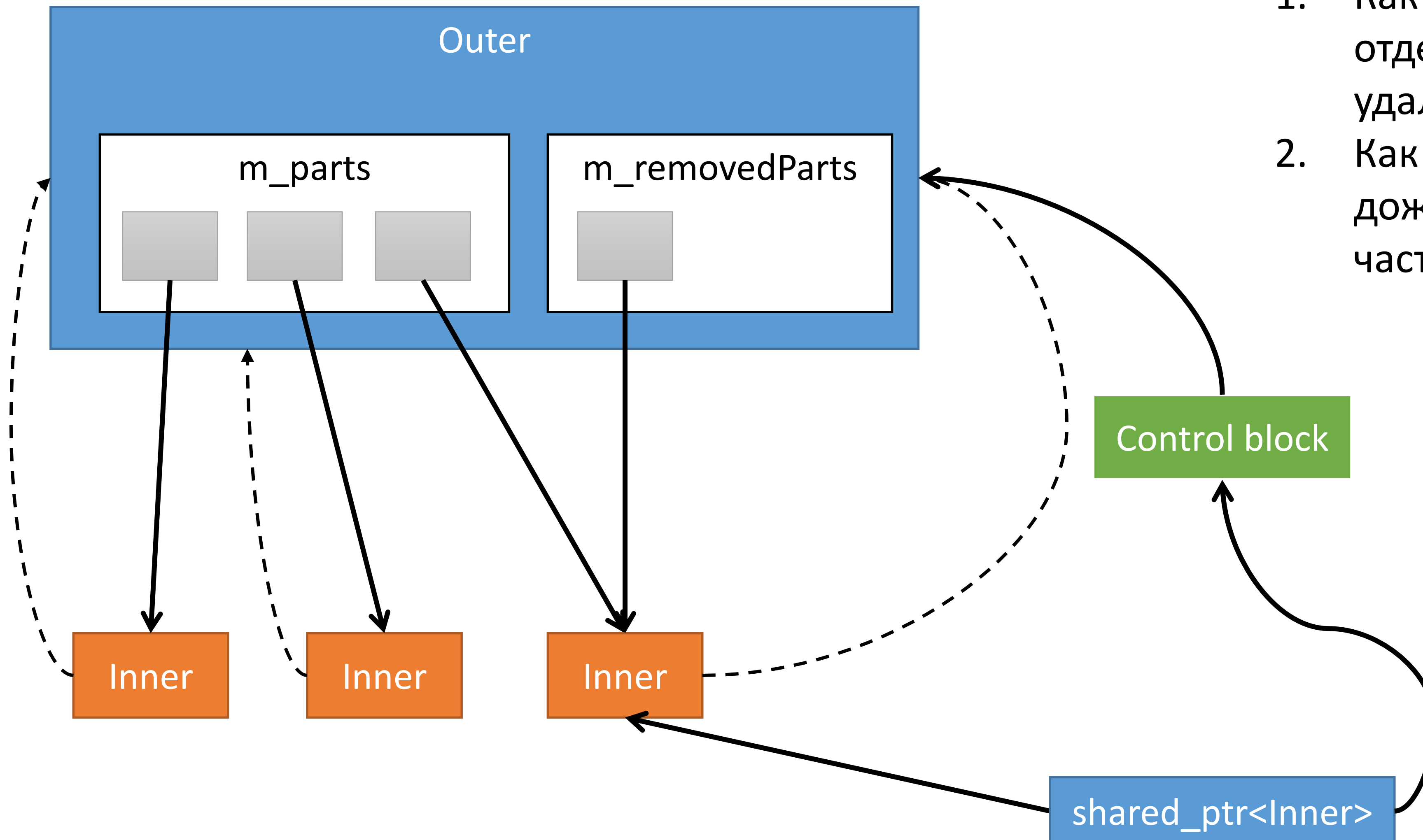


```
struct Inner
{
  friend struct Outer;
  Inner(string name, weak_ptr<Outer> outer)
    : m_name(move(name)), m_outer(move(outer))
  {}

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

  string_view GetName() const
  {
    return m_name;
  }

  shared_ptr<Outer> GetOuter() const
  {
    return m_outer.lock();
  }

private:
  string m_name;
  weak_ptr<Outer> m_outer;
};
```

```
struct Outer : enable_shared_from_this<Outer>
{
  Outer() = default;
  Outer(const Outer&) = delete;
  Outer& operator=(const Outer&) = delete;

  shared_ptr<Inner> CreateNewPart(string name) {
    m_parts.emplace_back(make_unique<Inner>(move(name), shared_from_this()));
    return GetPart(m_parts.size() - 1);
  }

  shared_ptr<Inner> GetPart(size_t index) const {
    return shared_ptr<Inner>(shared_from_this(), m_parts.at(index).get());
  }

  void RemovePart(size_t index) {
    m_removedParts.emplace_back(std::move(m_parts.at(index)));
    m_removedParts.back()->m_outer.reset();
    m_parts.erase(m_parts.begin() + index);
  }
private:
  vector<unique_ptr<Inner>> m_parts;
  vector<unique_ptr<Inner>> m_removedParts;
};
```

Нельзя просто взять и удалить часть объекта

Проблемы

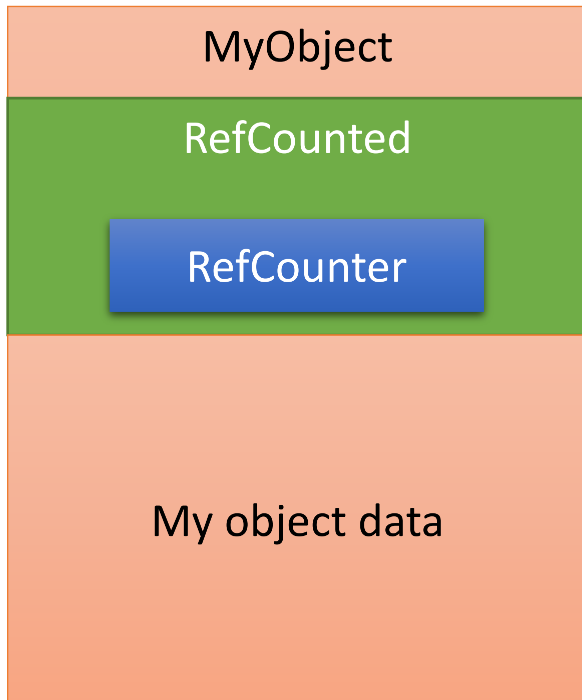


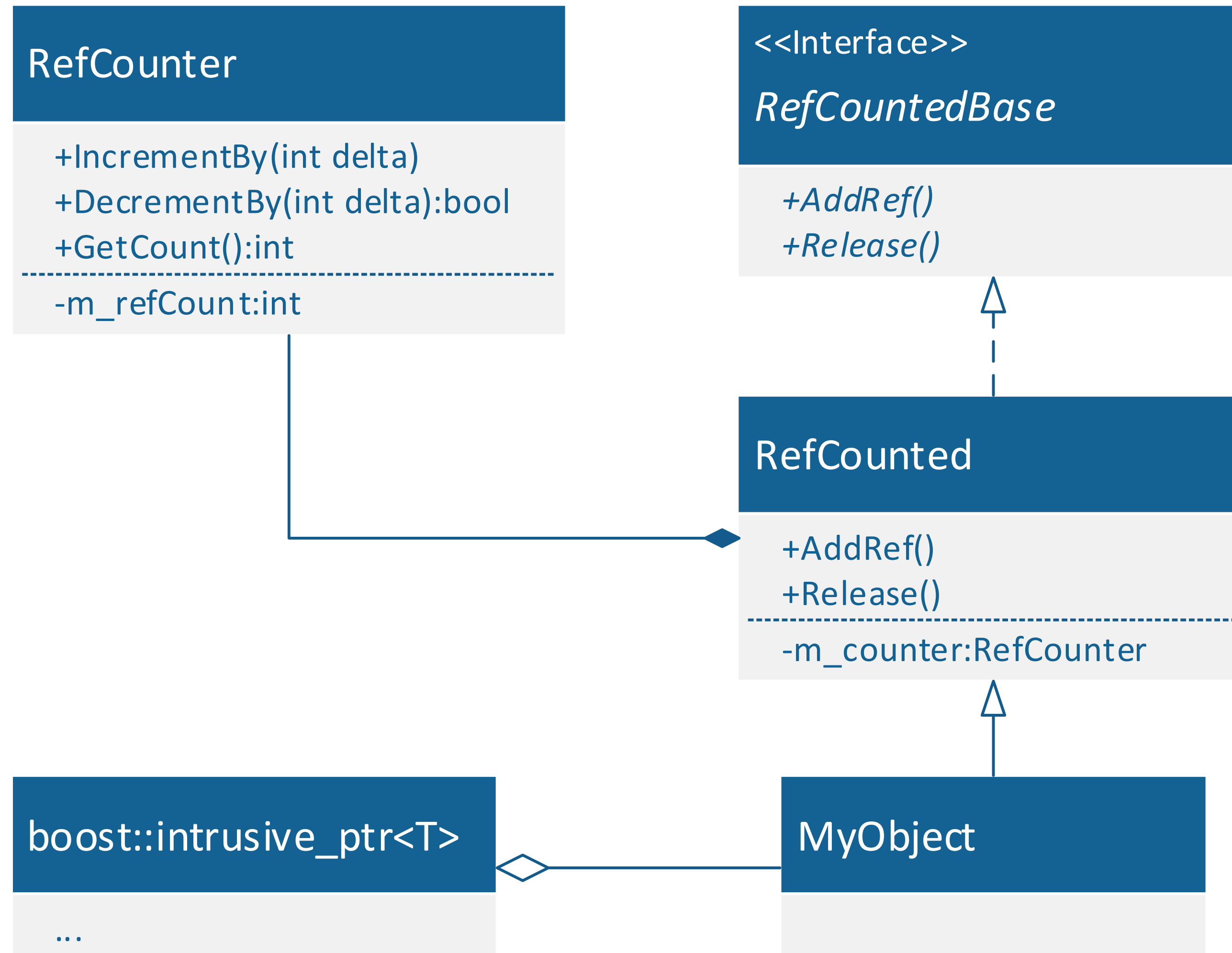
1. Как освободить память от отделённых частей, не дожидаясь удаления владеющего объекта?
2. Как удалить внешний объект, не дожидаясь удаления отцепленных частей?

При помощи shared_ptr и weak_ptr эта проблемы не разрешимы!

Но можно решить при помощи ~~garbage collector~~ intrusive ref counter

Простой класс с подсчётом ссылок





```cpp
class RefCounter final {
public:
  RefCounter() = default;
  RefCounter(const RefCounter&) = delete;
  RefCounter& operator=(const RefCounter&) = delete;

  void IncrementBy(int delta) noexcept {
    m_refCount += delta;
  }

  [[nodiscard]] bool DecrementBy(int delta) noexcept {
    if ((m_refCount -= delta) == 0) {
      return true;
    }
    return false;
  }

  int GetCount() const noexcept {
    return m_refCount;
  }

private:
  int m_refCount = 0;
};
```

```cpp
class RefCountedBase {
public:

  virtual void AddRef() const noexcept = 0;
  virtual void Release() const noexcept = 0;

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

protected:
  RefCountedBase() = default;

  virtual ~RefCountedBase() = default;
};

inline void intrusive_ptr_add_ref(const RefCountedBase* p) noexcept {
  p->AddRef();
}

inline void intrusive_ptr_release(const RefCountedBase* p) noexcept {
  p->Release();
}
```

```cpp
class RefCounted : public RefCountedBase {
public:

  void AddRef() const noexcept final {
    m_counter.IncrementBy(1);
  }

  void Release() const noexcept final {
    if (m_counter.DecrementBy(1)) {
      delete this;
    }
  }

private:
  mutable RefCounter m_counter;
};
```

```cpp
struct Foo : RefCounted {
  void DoSomething();
};

boost::intrusive_ptr<Foo> foo(new Foo());
foo->DoSomething();
```

Шило на мыло?

Владельцы и подобъекты

- У каждого объекта может быть 0 или 1 владельцев
  - Объект хранит указатель на владельца
- Владелец при своём удалении должен удалить свои подобъекты
  - Отделяемые объекты можно хранить по unique_ptr

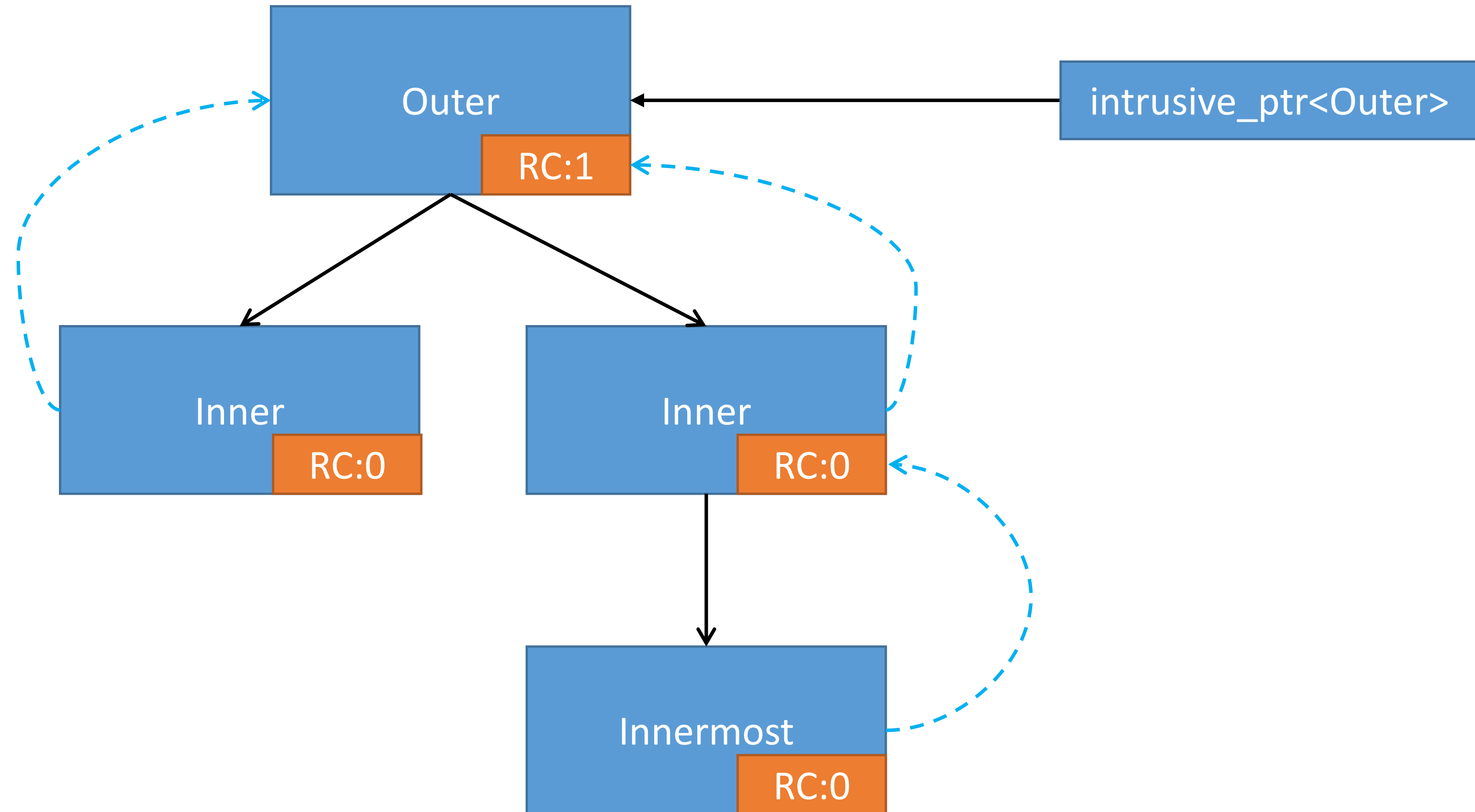
Подсчёт ссылок

- У каждого объекта есть встроенный счётчик ссылок
  - После создания объекта его счётчик ссылок равен 0
- Для управления счётчиком служат методы AddRef/Release
- При изменении счётчика ссылок на объект, изменяется счётчик ссылок на его владельца (при наличии)
  - **Инвариант**: счётчик ссылок объекта равен количеству ссылок на сам объект и на его подобъекты
- При обнулении счётчика ссылок объект, не имеющий владельца, удаляет сам себя

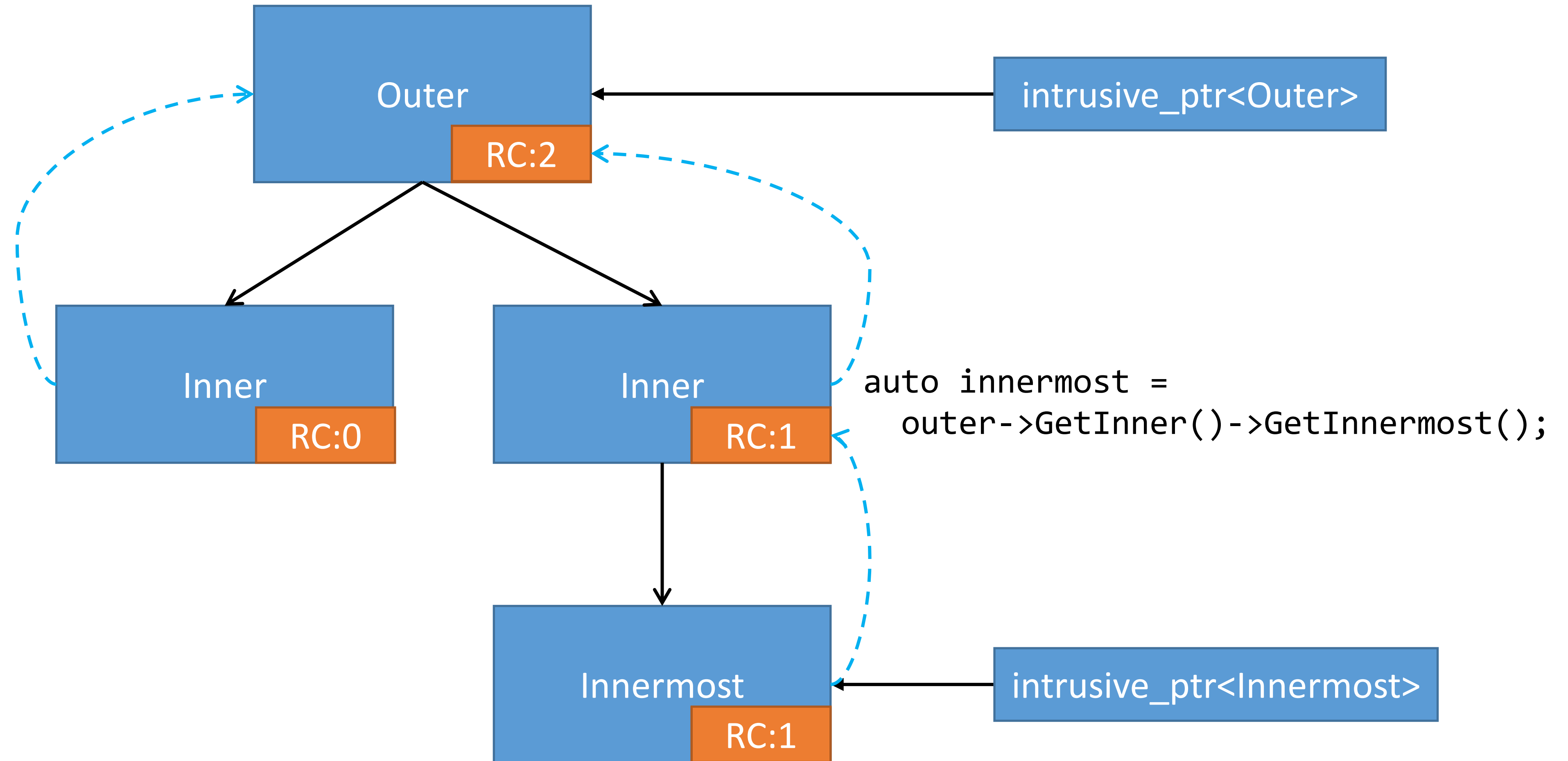
Присоединение и отделение подобъектов

- При присоединении к владельцу подобъект увеличивает счётчик ссылок своего владельца на количество собственных ссылок
- При своём отделении подобъект уменьшает счётчик ссылок своего владельца на количество собственных ссылок
  - Если счётчик ссылок на подобъект равен 0, он сам себя удаляет

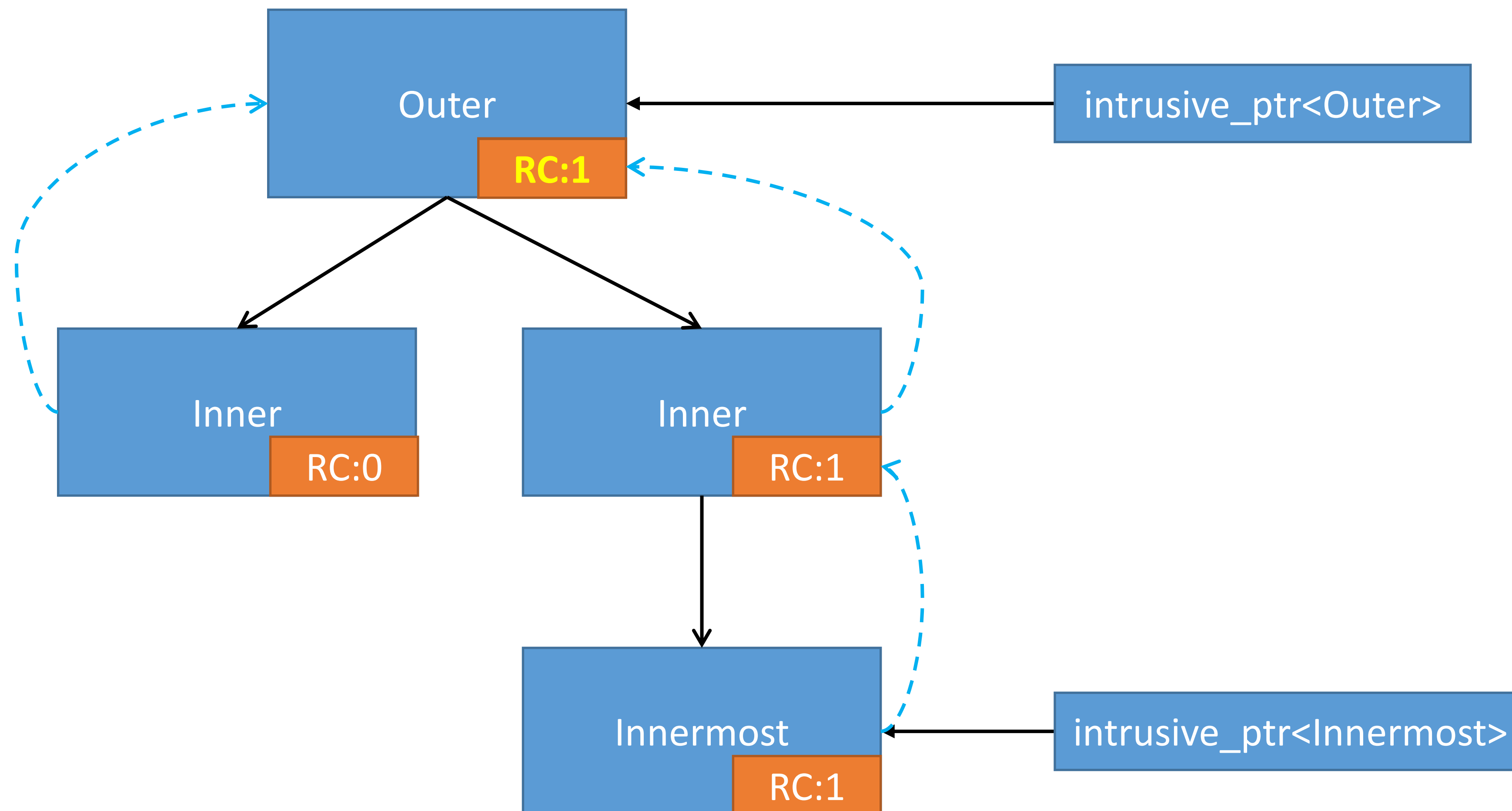
Начальное состояние



Добавление ссылки на часть структуры

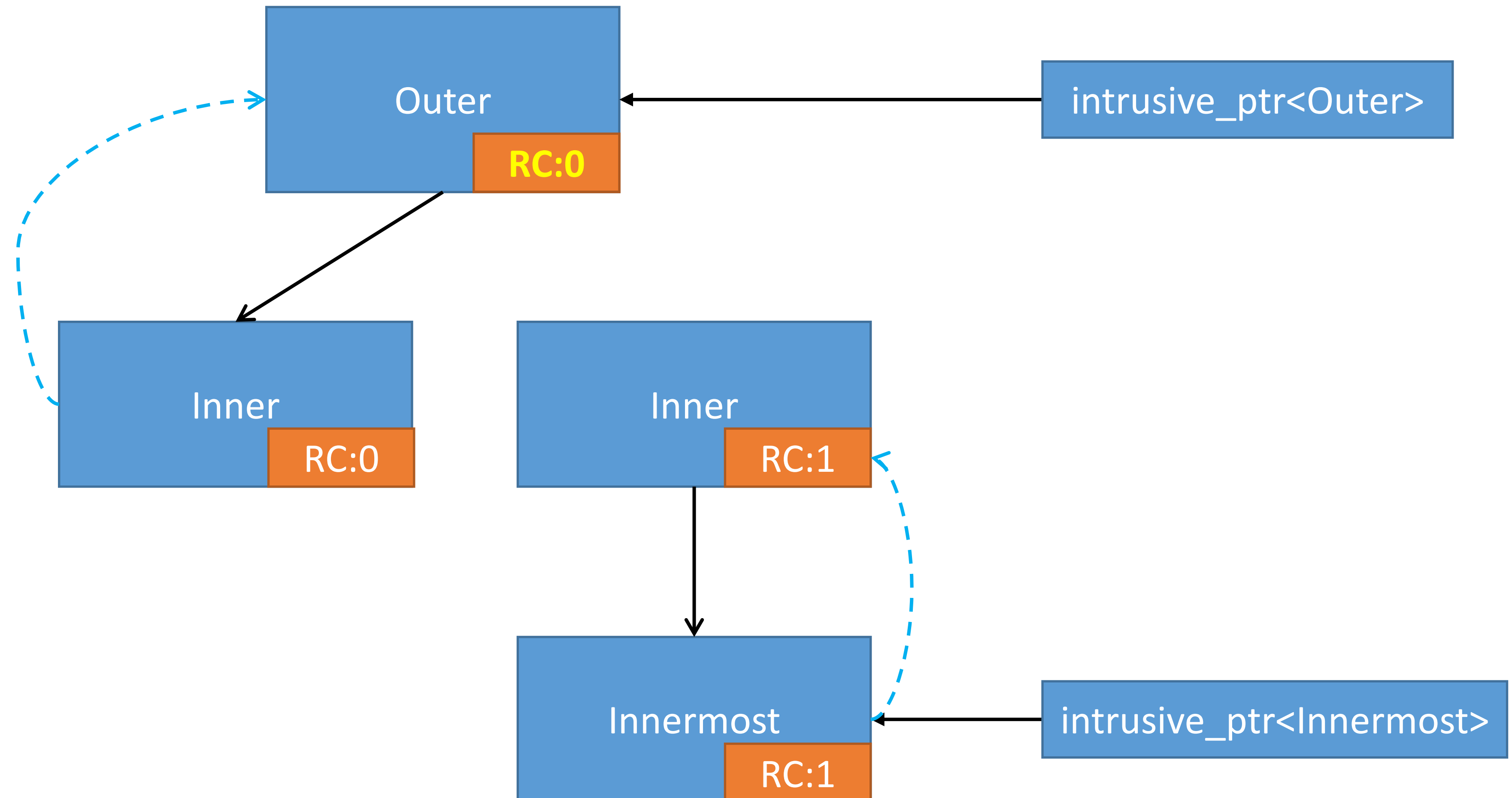


Отделение ссылки на внутреннюю часть

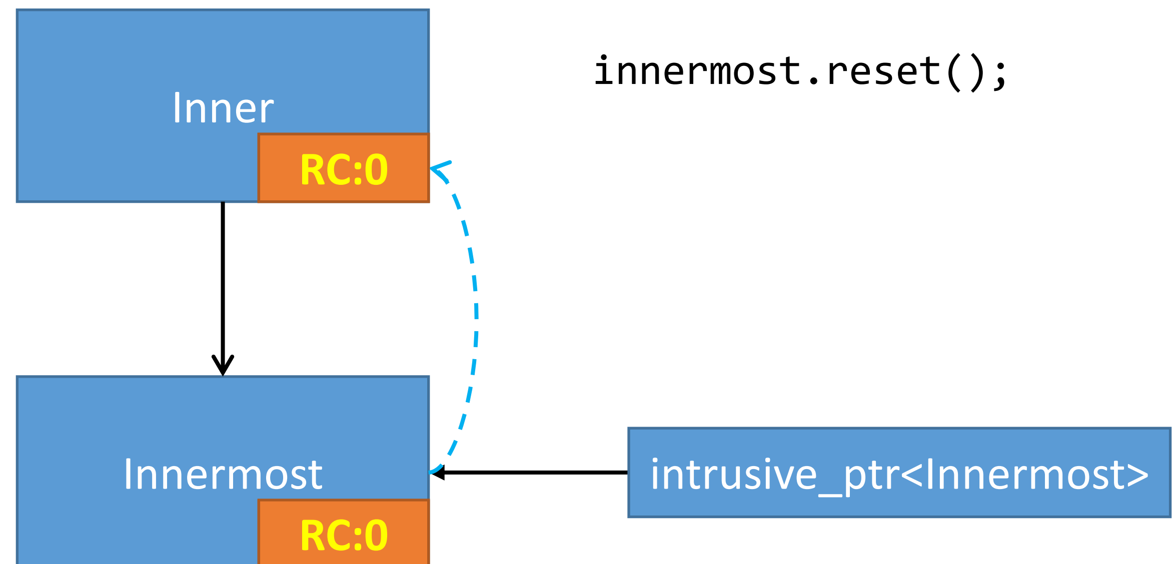


Освобождение ссылки на внешний объект

```
outer.reset();
```



Освобождение ссылки на внутренний объект




```
innermost.reset();
```

Увеличение счётчика ссылок

```
class DetachableRefCounted : public RefCountedBase {
public:
  void AddRef() const noexcept final {
    AddRefs(1);
  }
  …
private:
  void AddRefs(int refCount) const noexcept {
    m_counter.IncrementBy(refCount);
    AddRefsToOwner(refCount);
  }

  void AddRefsToOwner(int refCount) const noexcept {
    if (m_owner) {
      m_owner->AddRefs(refCount);
    }
  }
  …
  DetachableRefCounted* m_owner = nullptr;
  mutable RefCounter m_counter;
};
```

Уменьшение счётчика ссылок

```
class DetachableRefCounted : public RefCountedBase {
public:
  void Release() const noexcept final {
    ReleaseRefs(1);
  }
  …
private:
  void ReleaseRefs(int refCount) const noexcept {
    bool noSelfRefs = m_counter.DecrementBy(refCount);
    bool isSelfOwned = ReleaseOwnerRefs(refCount);
    if (noSelfRefs && isSelfOwned) {
      delete this;
    }
  }

  bool ReleaseOwnerRefs(int refCount) const noexcept {
    if (m_owner) {
      m_owner->ReleaseRefs(refCount);
      return false;
    }
    return true;
  }
  …
};
```

```
class RefCounter final {
  …
  bool DecrementBy(int delta) {
    if ((m_refCount -= delta) == 0) {
      return true;
    }
    return false;
  }
  …
};
```

Управление связью с владельцем

```
class DetachableRefCounted : public RefCountedBase {
public:
  void SetOwner(DetachableRefCounted& owner) {
    if (m_owner) {
      throw std::logic_error("already owned");
    }
    m_owner = &owner;
    m_owner->AddRefs(m_counter.GetCount());
  }

  void DetachFromOwner() noexcept {
    if (m_owner) {
      m_owner->ReleaseRefs(m_counter.GetCount());
      m_owner = nullptr;

      if (m_counter.GetCount() == 0) {
        delete this;
      }
    }
  }
  …
};
```

```cpp
template <typename T> class RefCountPtr final {
public:
  RefCountPtr(T* p = nullptr) noexcept : m_ptr(p) {
    if (m_ptr) m_ptr->AddRef();
  }

  RefCountPtr(const RefCountPtr& other) noexcept : m_ptr(other.m_ptr) {
    if (m_ptr) m_ptr->AddRef();
  }
  RefCountPtr(RefCountPtr&& other) noexcept: m_ptr(other.m_ptr) {
    other.m_ptr = nullptr;
  }

  ~RefCountPtr() noexcept {
    if (m_ptr) m_ptr->Release();
  }

  T* Get() const noexcept { return m_ptr; } // для любителей экстрима

  …
private:
  T* m_ptr;
};
```

```cpp
template <typename T> class RefCountPtr final {
  …
  RefCountPtr& operator=(const RefCountPtr& other) noexcept {
    if (m_ptr != other.m_ptr) {
      RefCountPtr(other).swap(*this);
    }
    return *this;
  }

  RefCountPtr& operator=(RefCountPtr&& other) noexcept {
    if (this != std::addressof(other)) {
      RefCountPtr(std::move(other)).swap(*this);
    }
    return *this;
  }

  void swap(RefCountPtr& other) noexcept {
    std::swap(m_ptr, other.m_ptr);
  }
  …
};
```

```cpp
template <typename T> class RefCountPtr final {
  …
  bool operator!() const noexcept { return m_ptr == nullptr; }

  bool operator==(const RefCountPtr& other) const noexcept {
    return m_ptr == other.m_ptr;
  }

  bool operator!=(const RefCountPtr& other) const noexcept {
    return m_ptr != other.m_ptr;
  }

  explicit operator bool() const noexcept { return m_ptr != nullptr; }

  void Reset() noexcept {
    if (m_ptr) {
      m_ptr->Release();
      m_ptr = nullptr;
    }
  }
  …
};
```

```
template <typename T> class RefCountPtr final {
  …
  NoAddRefRelease<T>& operator*() const noexcept {
    return static_cast<NoAddRefRelease<T>&>(*m_ptr);
  }

  NoAddRefRelease<T>* operator->() const noexcept {
    return static_cast<NoAddRefRelease<T>*>(m_ptr);
  }
  …
};
```

```
RefCountPtr<Foo> p(new Foo());
p->AddRef();
p->Release();
```

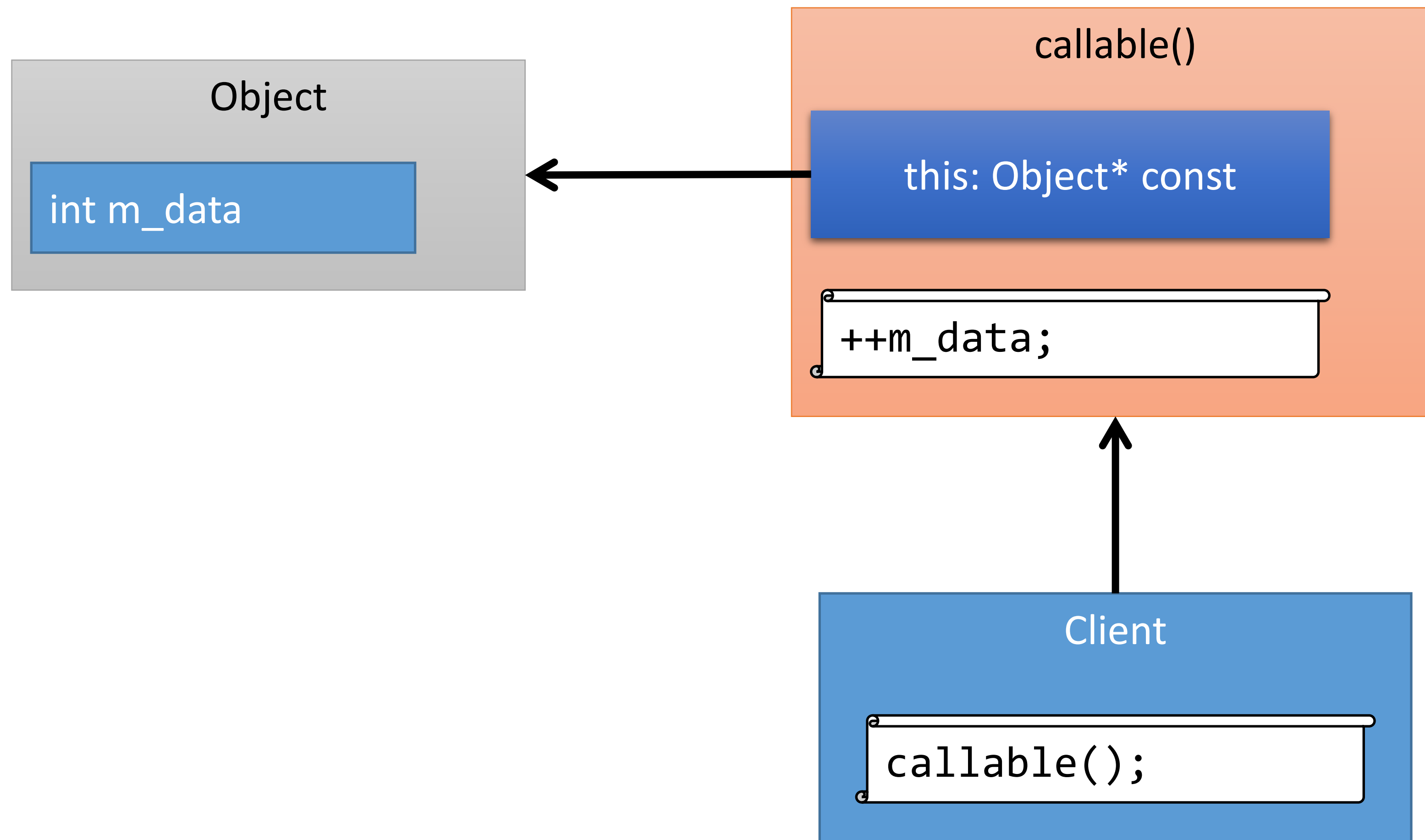
```
template <typename T> class NoAddRefRelease : public T {
  void AddRef() const noexcept override = 0;
  void Release() const noexcept override = 0;
public:
  void SetOwner(DetachableRefCounted& owner) = delete;
  void DetachFromOwner() noexcept = delete;
};
```

```cpp
struct Bar : DetachableRefCounted {
  void AddChild(Bar* obj) {
    m_children.emplace_back(obj);
    m_children.back()->SetOwner(*this);
  }

  RefCountPtr<Bar> GetChild(size_t index) const {
    return m_children.at(index).get();
  }

  void RemoveChild(size_t index) {
    auto it = m_children.begin() + index;
    it->release()->DetachFromOwner();
    m_children.erase(it);
  }
private:
  std::vector<std::unique_ptr<Bar>> m_children;
};
```

```cpp
RefCountPtr<Bar> parent(new Bar());
parent->AddChild(new Bar());
auto child = parent->GetChild(0);
parent->RemoveChild(0);
```

Вернемся вновь к shared/weak ptr

Проблема с захватом this в лямбда-выражениях

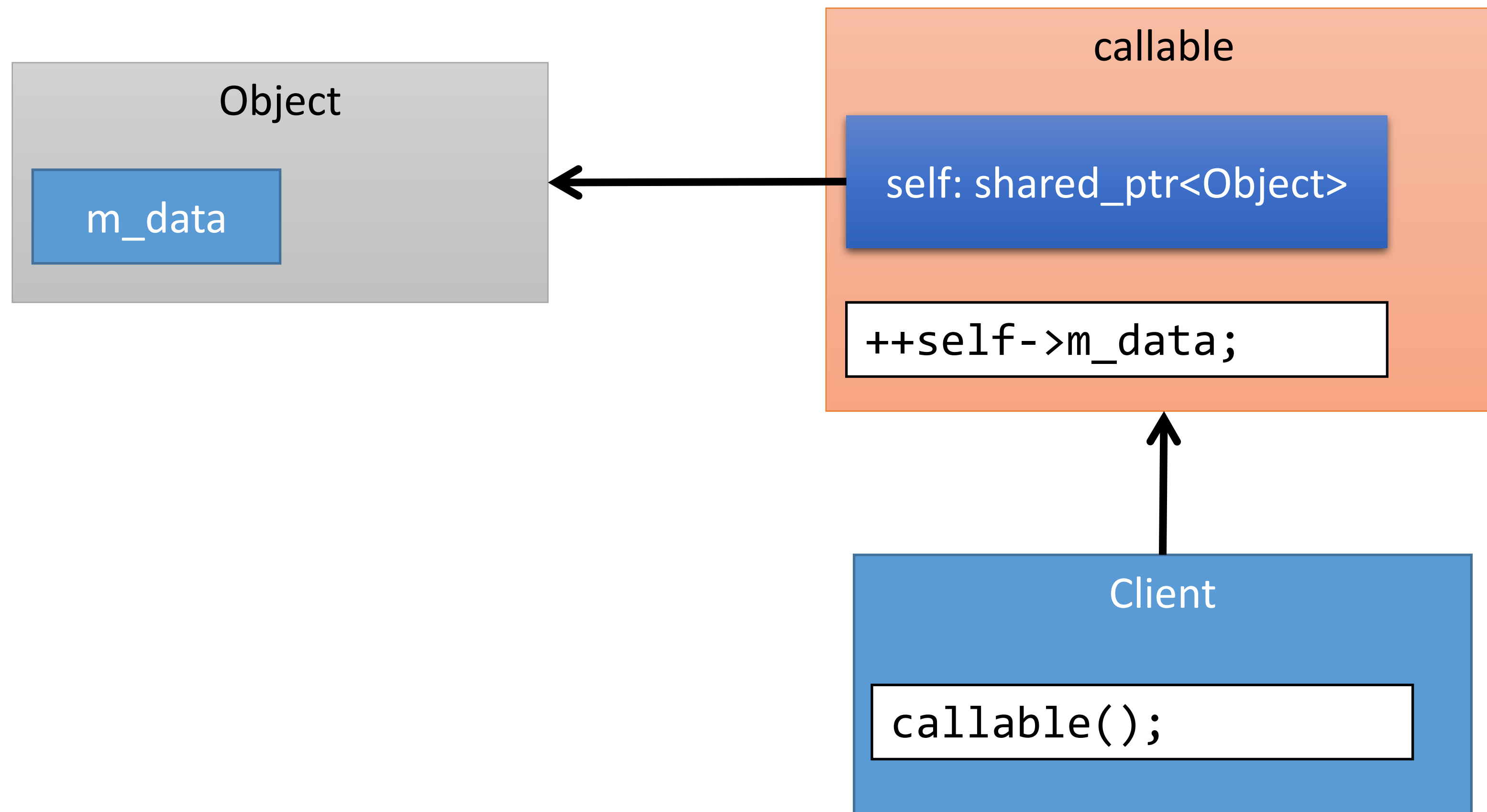


```cpp
class Foo
{
public:
  using Callback = function<void()>;

  ~Foo() {
    cout << "~Foo\n";
  }

  Callback GetCallback() {
    return [this] {
      ++m_data;
      cout << "Data:" << m_data << "\n";
    };
  }

private:
  int m_data = 0;
};
```

```cpp
void Test() {
  auto foo = make_shared<Foo>();
  auto cb = foo->GetCallback();

  cb();
  foo.reset();

  cb();
}
```

```
Data:1
~Foo
<undefined behavior>
```

Идиома Strong this (strong self)

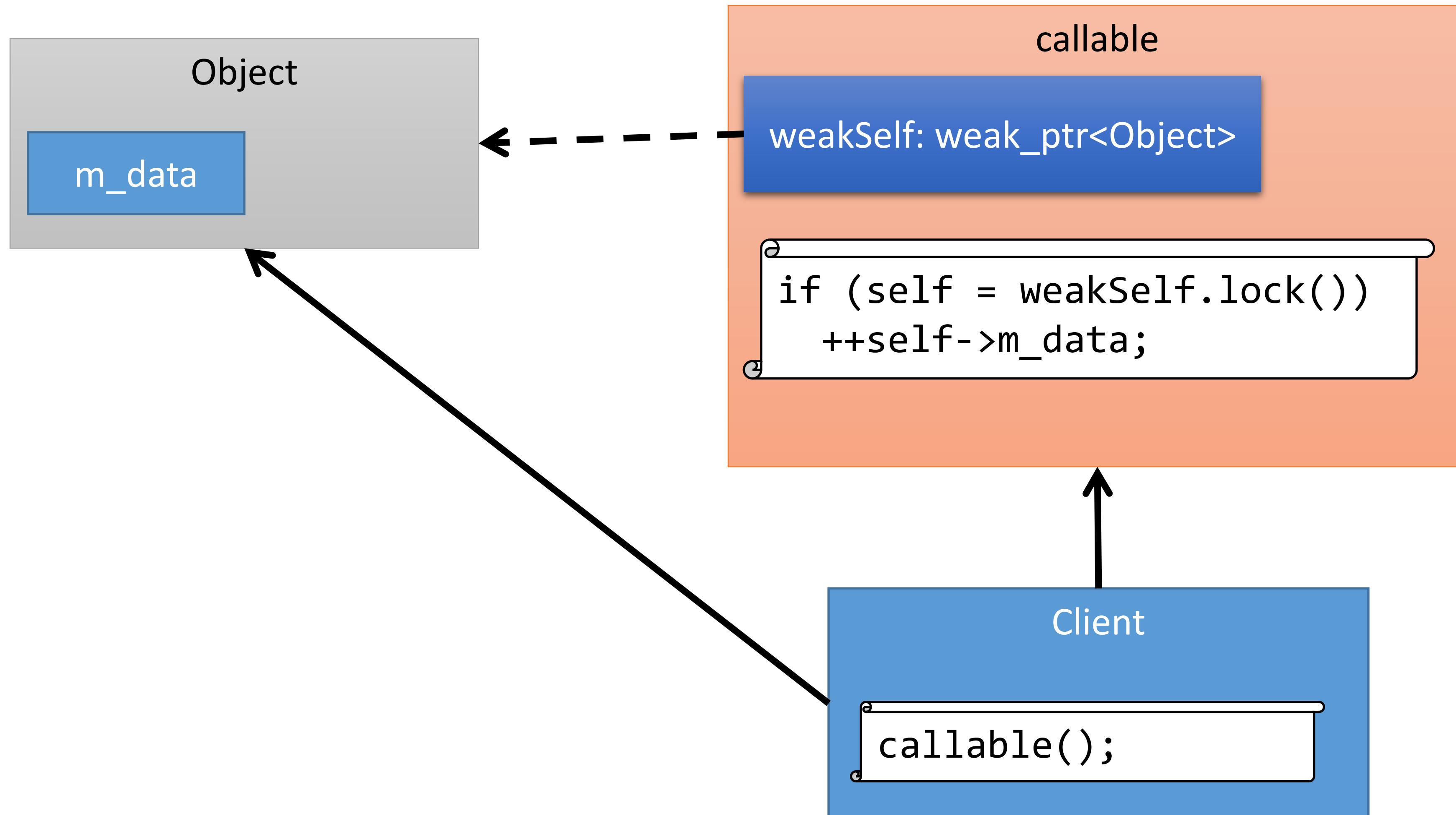


```cpp
class Foo : public enable_shared_from_this<Foo> {
public:
  using Callback = function<void()>;

  ~Foo() { cout << "~Foo\n"; }

  Callback GetCallback() {
    return [self = shared_from_this()] {
      ++self->m_data;
      cout << "Data:" << self->m_data << "\n";
    };
  }

private:
  int m_data = 0;
};
```

```cpp
void Test() {
  auto foo = make_shared<Foo>();
  weak_ptr weakFoo{ foo };
  auto cb = foo->GetCallback();

  cb();

  foo.reset();
  assert(!weakFoo.expired());
  cb();

  cb = {};
  assert(weakFoo.expired());
}
```

```
Data:1
Data:2
~Foo
```

Идиома Weak this (Weak self)



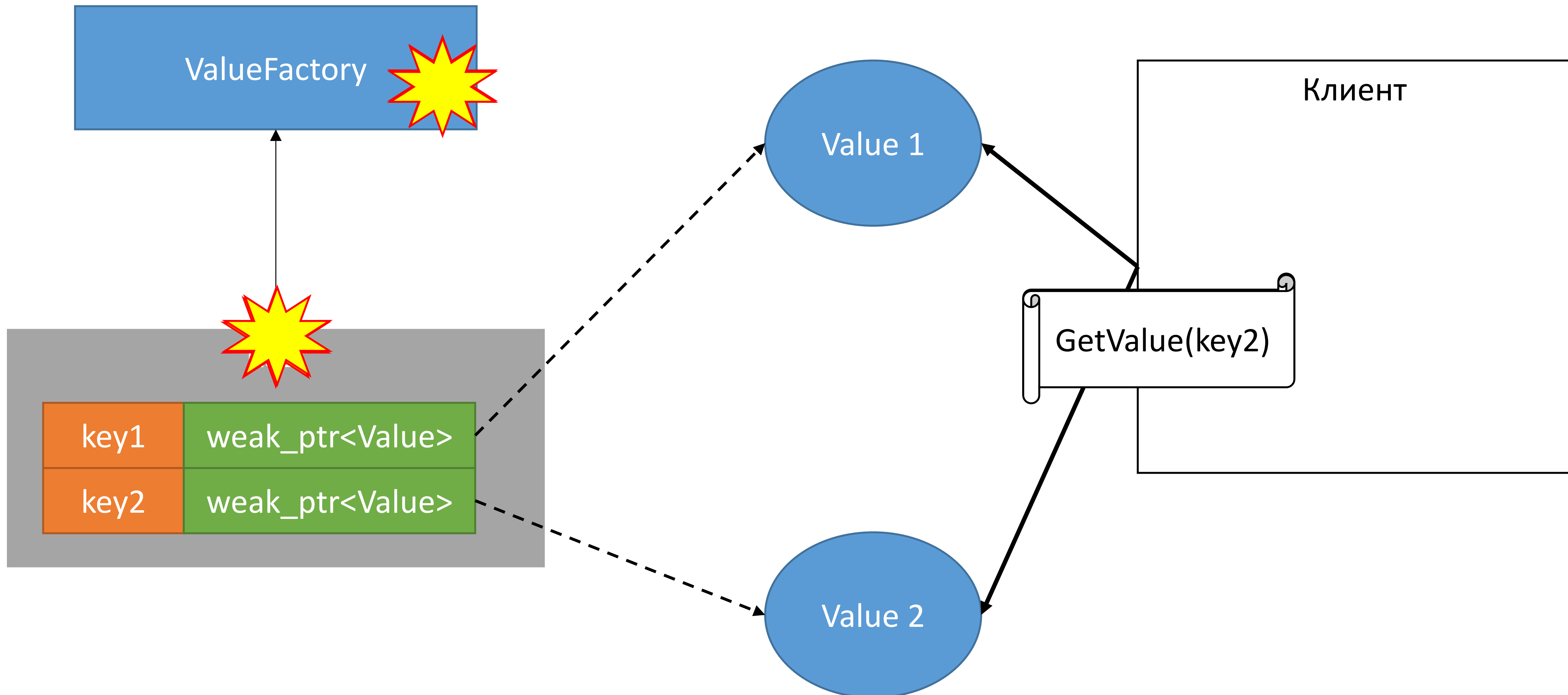
```cpp
class Foo : public enable_shared_from_this<Foo> {
public:
  using Callback = function<void()>;

  ~Foo() { cout << "~Foo\n"; }

  Callback GetCallback() {
    return [weakSelf = weak_from_this()] {
      cout << "entering callback\n";

      if (auto self = weakSelf.lock()) {
        cout << "  updating data\n";
        ++self->m_data;
      } else {
        cout << "  Foo was destroyed\n";
      }
    };
  }

private:
  int m_data = 0;
};
```

```cpp
void Test() {
  auto foo = make_shared<Foo>();
  auto cb = foo->GetCallback();

  cb();

  foo.reset();
  cb();
}
```

```
entering callback
  updating data
~Foo
entering callback
  Foo was destroyed
```

Пример – кэш

Структура кэша



shared_ptr custom deleter

```
struct Foo {};

shared_ptr<Foo> foo(new Foo(), [](Foo* p) {
  delete p;
  cout << "Foo has been deleted\n";
});
```

```cpp
template <class Key, class Val, class Hasher = hash<Key>, class KeyEq = equal_to<Key>>
class CacheT : public enable_shared_from_this<CacheT<Key, Val>> {
public:
  using MyType = CacheT<Key, Val, Hasher, KeyEq>;
  using ValuePtr = shared_ptr<Val>;
  using ValueWeakPtr = weak_ptr<Val>;
  using CacheCleaner = function<void()>;
  using ValueFactory = function<ValuePtr(const Key& key, CacheCleaner cleaner)>;

  CacheT(ValueFactory valueFactory)
    : m_valueFactory(move(valueFactory))
  {}

  …

private:
  mutable std::unordered_map<Key, ValueWeakPtr> m_items;
  ValueFactory m_valueFactory;
};
```

```cpp
template <class Key, class Val, class Hasher = hash<Key>, class KeyEq = equal_to<Key>>
class CacheT : public enable_shared_from_this<CacheT<Key, Val>> {
public:
  …
  ValuePtr GetValue(const Key& key) const {
    ValuePtr value;
    if (auto it = m_items.find(key); it != m_items.end()) {
      value = it->second.lock();
    }

    if (!value) {
      value = m_valueFactory(key, [key, weakSelf = MyType::weak_from_this()] {
        if (auto self = weakSelf.lock())
          self->m_items.erase(key);
      });
      m_items.emplace(key, value);
    }
    return value;
  }
};
```

```cpp
struct DataSource {};
using DataSourcePtr = shared_ptr<DataSource>;

struct Data {
  explicit Data(DataSourcePtr dataSrc): m_dataSrc(move(dataSrc)) {}
private:
  DataSourcePtr m_dataSrc;
};
using DataPtr = shared_ptr<Data>;

class DataCache : public CacheT<DataSourcePtr, Data> {
public:
  DataCache(): CacheT(DataCache::DataFactory) {}

private:
  static DataPtr DataFactory(const DataSourcePtr& key, CacheCleaner cleaner) {
    return DataPtr(new Data(key), [cleaner = move(cleaner)](Data* d) {
      cleaner();
      delete d;
    });
  }
};
```

DataCache в действии

```
auto cache = make_shared<DataCache>();
auto src = make_shared<DataSource>();

auto data = cache->GetValue(src);
```

Вопросы?

Спасибо за внимание!

Алексей Малов
alexey.malov@ispringsolutions.com