Лекция 1. Умные указатели

Умные указатели

- Почти те же указатели, только умнее
  - представляют собой RAII классы
  - часто поддерживают тот же интерфейс, что и обычные указатили: `op->`, `op*`, `op<` (например, чтобы положить в `std::set`)
  - сами управляют временем жизни объекта – вовремя вызывают деструкторы и освобождают память

Польза умных указателей

- Автоматическое освобождение памяти при удалении самого указателя
- Безопасность исключений

```
1.  void foo()
2.  {
3.      shared_ptr<my_class> ptr(new my_class("arg"));
4.      // or shorter definition:
5.      auto ptr = make_shared<my_class>("arg");
6.
7.      ptr->bar(); // if throws exception, nothing bad happened
8.  }
9.
10. void foo()
11. {
12.     my_class* ptr = new my_class(/*...*/);
13.     ptr->bar(); // oops, troubles in case of exception
14.     delete ptr; // common trouble is to forget delete
15. }
```

Популярные умные указатели

- `std::scoped_ptr`
- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`
- `boost::intrusive_ptr`

- Deprecated: `std::auto_ptr` (заменен `unique_ptr`)

scoped_ptr

- Удобен для хранения указателя на стеке или полем класса. Не позволяет копироваться.

```
 1. template<class T> struct scoped_ptr : noncopyable {
 2.
 3. public:
 4.     typedef T element_type;
 5.
 6.     explicit scoped_ptr(T * p = 0);
 7.     ~scoped_ptr();
 8.
 9.     void reset(T * p = 0);
10.
11.     T & operator *() const;
12.     T * operator->() const;
13.     T * get() const;
14.
15.     operator unspecified-bool-type() const;
16. };
17. //-------
18. scoped_ptr<int> p(new int(5));
```

Почему `explicit` конструктор?

```
1.  //what if scoped_ptr had implicit constructor
2.  void foo(scoped_ptr<my_class> ptr)
3.  {
4.      /*...*/
5.  }
6.
7.  auto p = new my_class(/*...*/);
8.  foo(p);       // epic fail, p is not valid after this call
9.
10. p->do_smth();// error
11. delete p;    // one more error
```

Возможности `scoped_ptr`

- Самый простой и быстрый
- Нельзя копировать и перемещать (move)
- Нельзя использовать в stl контейнерах
- Для массива: scoped_array
- При определении не требует полный тип, для инстанцировании - требует

Требование полноты типа

```
1   #include <boost/scoped_ptr.hpp>
2
3   // b.h
4   struct A;
5   struct B
6   {
7       boost::scoped_ptr<A> a;
8       // some declarations
9       // but no explicit destructor
10  };
11
12  // main.cpp
13  #include "b.h"
14
15  int main()
16  {
17     B b;
18     return 0;
19  }
```

- Такой код приводит к ошибке компиляции

checked_delete

```
1  // scoped_ptr.hpp
2  ~scoped_ptr() // never throws
3  {
4      boost::checked_delete(ptr);
5  }
6
7  // checked_delete.hpp
8  template<class T>
9  inline void checked_delete(T * x)
10 {
11     // intentionally complex - simplification causes regressions
12     typedef char type_must_be_complete[ sizeof(T)? 1: -1 ];
13     (void) sizeof(type_must_be_complete);
14     delete x;
15 }
```

- Либо стоит объявить полностью тип A в том же хидере после типа B
- Либо типу B необходимо добавить объявление деструктора (определение может быть в другом cpp-файле)

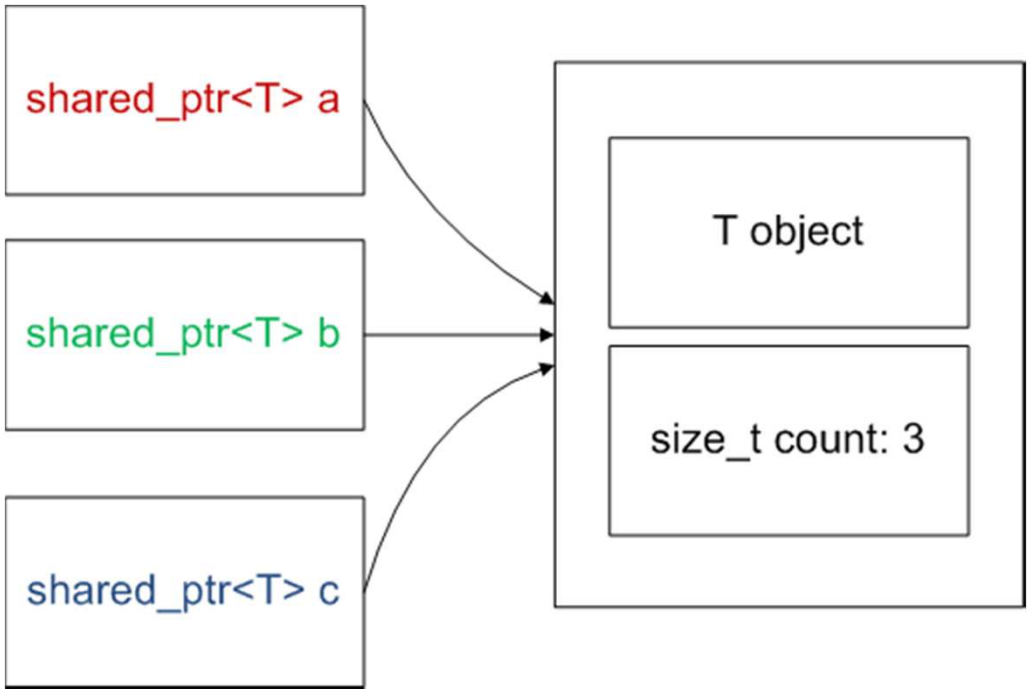
std::unique_ptr

- Владеет объектом эксклюзивно
- Нельзя копировать, но можно перемещать
- Заменяет `std::auto_ptr` (тот перемещал владение при копировании – требовал копирование от неконстантной ссылки)
- Удобно использовать при возврате из функции
- Есть функция `release()`

```
1  template<
2      class T,
3      class Deleter = std::default_delete<T>
4  > class unique_ptr;
5
6  template <
7      class T,
8      class Deleter
9  > class unique_ptr<T[], Deleter>;
```

shared_ptr

- Поддерживает общий счетчик ссылок на выделенный объект
- Удаляет объект только, когда последний из ссылающихся shared_ptr’ов удаляется или принимает указатель на другой объект



shared_ptr

- Наиболее используемый.
- Удобен для разделения владением.
- Можно возвращать из функций.
- *Можно передавать между модулями - запоминает правильную функцию удаления (из нужной библиотеки)

```
1.  template<class T> struct shared_ptr
2.  {
3.      /* more than scoped_ptr has */
4.      shared_ptr(shared_ptr const & r);
5.      template<class Y> shared_ptr(shared_ptr<Y> const & r);
6.
7.      shared_ptr(shared_ptr && r);
8.      template<class Y> shared_ptr(shared_ptr<Y> && r);
9.
10.     bool unique() const;
11.     long use_count() const;
12.     /*...*/
13. };
```

shared_ptr

- Можно класть в STL контейнеры (есть даже сравнение)
- Полный тип требует только не момент инициализации!

- Избегайте циклов (используйте weak_ptr)
- Не передавайте временные shared_ptr:

```
1  void foo(shared_ptr<A> a, int){/*...*/}
2  int bar() {/*may throw exception*/}
3
4  int main()
5  {
6      // dangerously
7      foo(shared_ptr<A>(new A), bar());
8  }
```

shared_ptr & casts

```
1   struct Base{};
2   struct Derived{};
3
4   shared_ptr<Base> der;
5
6   Derived* to_d = dynamic_cast<Derived*>(der.get());
7   shared_ptr<Derived> d_ptr(to_d); // logical error
8
9   // correct way
10  auto d_ptr = dynamic_pointer_cast<Derived>(der);
```

- Доступны все 4 вида преобразований

boost make_shared, allocate_shared

```
1   // usual way
2   shared_ptr<some_struct> ptr(new some_struct(a, b, c));
3
4   // better way
5   auto ptr = make_shared<some_struct>(a, b, c);
```

- Умный указатели изолируют не только операторы delete, но и new
- Выделяет память на счетчик одним блоком с объектом
- Для выделения со свои аллокатором используйте `allocate_shared`

boost weak_ptr

```
1   struct client;
2   struct server
3   {
4       //...
5       typedef shared_ptr<client> client_ptr;
6       vector<client_ptr> clients_;
7   };
8
9   struct client
10  {
11      // ...
12      weak_ptr<server> srv_;
13  }
14
15  //... in client member function
16  if(auto srv = srv_.lock())
17  {
18      srv->send(/*...*/)
19  }
```

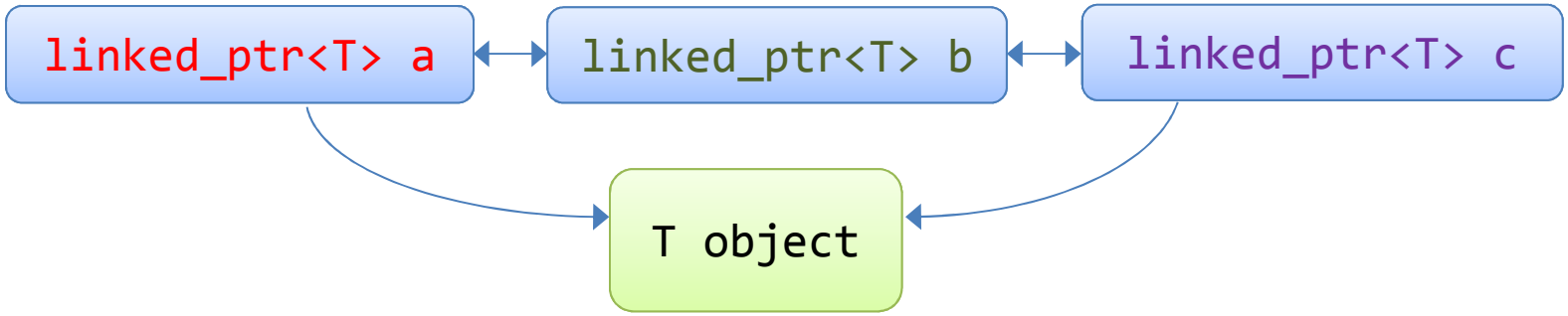
boots intrusive_ptr

- Хранит счетчик ссылок непосредственно в объекте

+ Нет дополнительных расходов на память
+ Можно передавать «сырой» указатель
+ Самый быстрый из умных указателей, разделяющих владение
- Требует вмешательство в класс
- Могут быть проблемы при построении иерархии

- Если неочевидно, что `intrusive_ptr` даст вам выигрыш, попробуйте сперва `shared_ptr`

linked_ptr



- Совместное владение объектом
- Не выделяет «лишней» памяти – быстрая инициализация, но медленное копирование
- Не фрагментирует память (32-bit)

shared_from_this*

```
1   struct client
2       : enable_shared_from_this
3   {
4       typedef shared_ptr<client> ptr_t;
5       ptr_t create(/*...*/) { return ptr_t(new client(/*...*/)); }
6
7   private:
8       void on_connected(service* srv)
9       {
10          srv->handle_read(bind(&client::on_read, shared_from_this(), _1));
11      }
12
13      void on_read(/*...*/){/*...*/}
14      //....
15  };
```

boost optional

- Очень похож на указатель, но хранит по значению в качестве своего поля. Вместе с флагом инициализации.

```
1   optional<double> try_get_value()
2   {
3       if (has_value)
4           return value;
5       else
6           return boost::none;
7   }
8   //-----------------------------
9   struct A
10  {
11      A(some_struct& s, double d);
12  }
13  //...
14  optional<A> a (A(s, 5.)); // makes copy
15  optional<A> b = in_place(ref(s), 5.);
```