Умные указатели

Александр Смаль

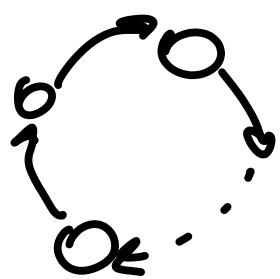
Академический университет
29 ноября 2013
Санкт-Петербург

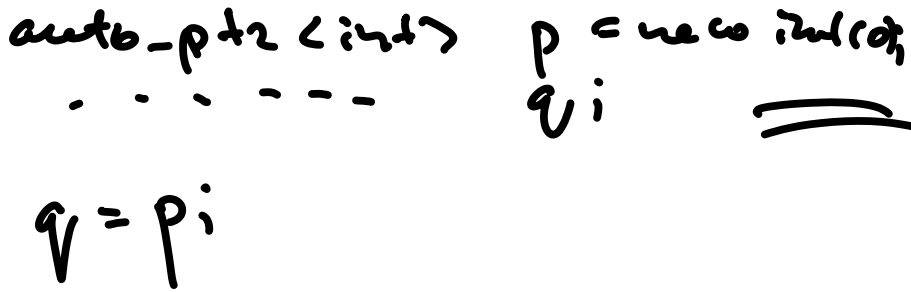
Умные указатели

1. Идиома RAII (Resource Acquisition Is Initialization): время жизни ресурса связанно с временем жизни объекта.
   - Получение ресурса в конструкторе.
   - Освобождение ресурса в деструкторе.
2. Основные области использования RAII:
   - для управления памятью,
   - для открытия файлов или устройств,
   - для мьютексов или критических секций.
3. Умные указатели — объекты, инкапсулирующие владение памятью. Синтаксически ведут себя так же, как и обычные указатели.

Основные стратегии

1. `scoped_ptr` — время жизни объекта ограничено временем жизни умного указателя.
2. `shared_ptr` — разделяемый объект, реализация с подсчётом ссылок.
3. `intrusive_ptr` — разделяемый объект, реализация самим внутри объекта.
4. `linked_ptr` — разделяемый объект, реализация списком указателей.
5. `auto_ptr`, `unique_ptr` — эксклюзивное владение объектом с передачей владения при присваивании.
6. `weak_ptr` — разделяемый объект, реализация с подсчётом ссылок, слабая ссылка (используется вместе с `shared_ptr`).





scoped_ptr

- Самый простой умный указатель: для хранения на стеке или в классе.
- Единственные владелец.
- Нельзя копировать и присваивать.
- Нельзя вернуть владение объектом.

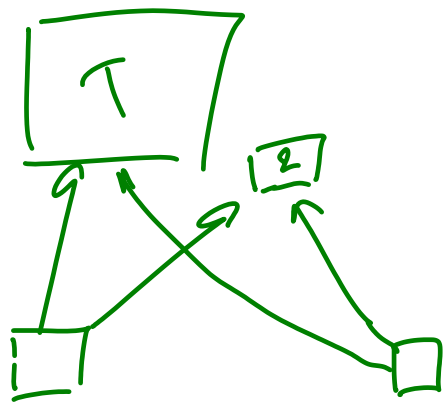
```
template<class T>
struct scoped_ptr {
    explicit scoped_ptr(T * p = 0) : p_(p) {}
    ~scoped_ptr(){ delete p_; }
    ...
    void reset(T * p = 0) { delete p_; p_ = p;}
    T *  get() const { return p_; }
private:
    scoped_ptr(scoped_ptr const&);
    scoped_ptr operator=(scoped_ptr const&);

    T   *   p_;
};
```

shared_ptr

- Для разделяемых объектов.
- Ведётся подсчёт ссылок, объект живёт до уничтожения последнего владельца.
- Нельзя вернуть владение объектом.

```
template<class T>
struct shared_ptr {
    explicit shared_ptr(T * p = 0) : p_(p), c_(0) {
        if (p_)
            c_ = new size_t(1);
    }
    shared_ptr(shared_ptr const& ptr) : p_(ptr.p_), c_(ptr.c_) {
        if(c_)
            ++*c_;
    }
    ~shared_ptr() {
        if (--*c_ == 0) {
            delete p_;
            delete c_;
        }
    }
    ...
private:
    T       *   p_;
    size_t  *   c_;
};
```



intrusive_ptr

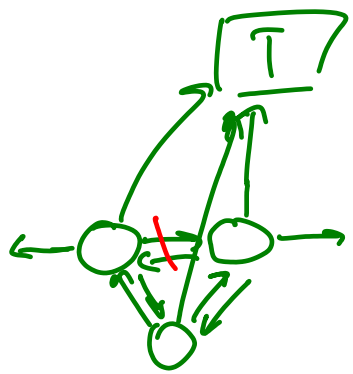
- Для разделяемых объектов.
- Объект самостоятельно управляет своим временем жизни.
- Нельзя вернуть владение объектом.

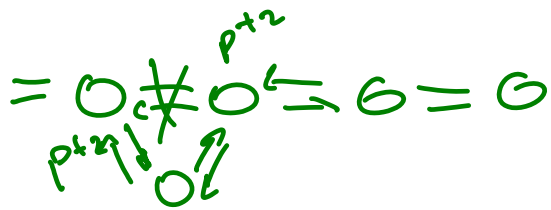
```
template<class T>
struct intrusive_ptr {
    explicit intrusive_ptr(T * p = 0) : p_(p) {
        if (p_)
            intrusive_addref(p_);
    }
    intrusive_ptr(intrusive_ptr const& ptr) : p_(ptr.p) {
        if (p_)
            intrusive_addref(p_);
    }
    ~intrusive_ptr() {
        if (p_) {
            intrusive_release(p_);
    }
    ...
private:
    T   *   p_;
};
```

linked_ptr

- Для разделяемых объектов.
- Указатели на один объект объединяются в список, исключает необходимость дополнительного выделения памяти.
- Нельзя вернуть владение объектом.

```
template<class T>
struct linked_ptr {
    explicit linked_ptr(T * p = 0) : p_(p), next_(0), prev_(0) {}
    linked_ptr(linked_ptr const& ptr) : p_(ptr.p) {
        if (p_)
            list_insert(ptr, *this);
    }
    ~linked_ptr() {
        if (next_ == prev_)
            delete p_;
        else
            list_remove(*this);
    }
    ...
private:
    mutable linked_ptr * next;
    mutable linked_ptr * prev;
    T    *    p_;
};
```





auto_ptr, unique_ptr

- Для передачи указателей в функции и возврата указателей из функций.
- Владение эксклюзивно и передаётся при присваивании.

```
template<class T>
struct auto_ptr {
    explicit auto_ptr(T * p = 0) : p_(p) {}
    ~auto_ptr(){ delete p_; }
    auto_ptr(auto_ptr & ptr) : p_(ptr.p_) {
        ptr.p_ = 0;
    }
    auto_ptr & operator=(auto_ptr & ptr) {
        delete p_;
        p_ = ptr.p_;
        ptr.p_ = 0;
        return *this;
    }
    T * release() { T * t = p_; p_ = 0; return t; }
    ...
private:
    T   *   p_;
};
```

sort

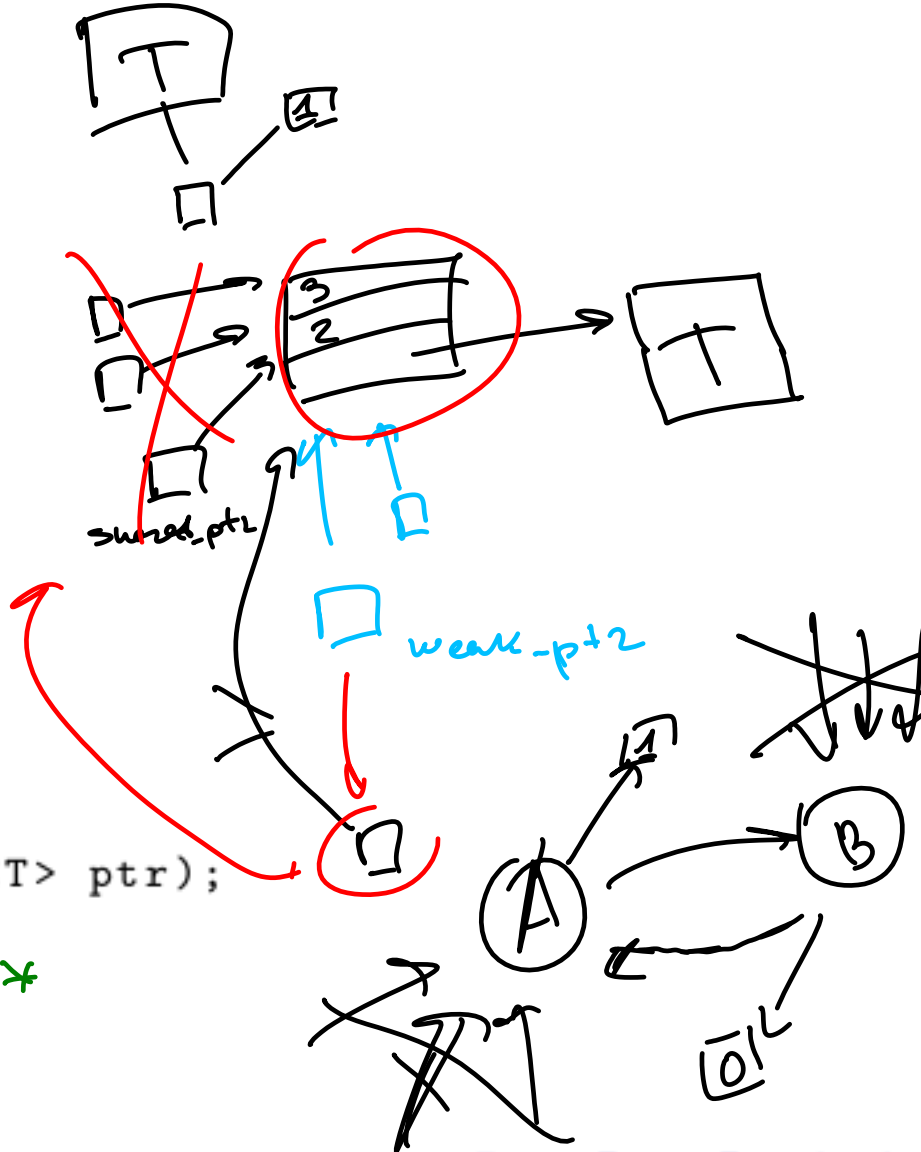
auto_ptr2 pivot = m[0];

weak_ptr

- Для использования вместе с shared_ptr.
- Задаёт слабую ссылку — для исключения циклических зависимостей.
- Не владеет объектом.

```
template<class T>
struct counter {
    size_t links;
    size_t weak_links;
    T * data_;
};
template<class T>
struct shared_ptr {
    ...
private:
    counter<T> * c_;
};

template<class T>
struct weak_ptr {
    explicit weak_ptr(shared_ptr<T> ptr);
    shared_ptr<T> lock();
    ...
private:
    counter * c_;
};
```



Заключение

- Умные указатели намного удобнее ручного управления памятью.
- Для локальных объектов — `scoped_ptr` или `scoped_array`.
- Для разделяемых объектов — `shared_ptr` или `shared_array`.
- Использовать `auto_ptr` нужно с большой осторожностью, т.к. у него нестандартная семантика присваивания.
- В сильносвязанных системах рассмотрите возможность использовать `weak_ptr`.
- Используйте `intrusive_ptr` для тех объектов, которые сами управляют своим временем жизни.
- Прочитайте документацию по `shared_ptr`.

boost.org