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

- 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.
- 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 && (--*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 {
    ...
    // Home assignment №3
    ...
private:
    linked_ptr * next;
    linked_ptr * prev;
    T    *    p_;
};
```

auto_ptr, unique_ptr

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

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

weak_ptr

- Для использования вместе с shared_ptr.
- Слабая ссылка 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 weak_ptr {
    explicit weak_ptr(shared_ptr<T> ptr);
    shared_ptr<T> lock();
    ...
private:
    counter<T> * 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`.

Safe bool: проблема

```
struct Testable {
    operator bool() const { return false; }
};

struct AnotherTestable {
    operator bool() const { return true; }
};

int main (void)
{
    Testable a;
    AnotherTestable b;
    if (a == b) { /* blah blah blah*/ }
    if (a < 0) { /* blah blah blah*/ }

    return 0;
}
```

Safe bool idiom

```
struct Testable {
    explicit Testable(bool b=true): ok_(b) {}

    operator bool_type() const {
      return ok_ ?
        & Testable::this_type_does_not_support_comparisons : 0;
    }
private:
    typedef void (Testable::*bool_type)() const;

    void this_type_does_not_support_comparisons() const {}

    bool ok_;
};

struct AnotherTestable  {...};.
```

Safe bool idiom (cont.)

```
template <typename T>
bool operator!=(const Testable& lhs, const T&) {
    lhs.this_type_does_not_support_comparisons();
    return false;
}
template <typename T>
bool operator==(const Testable& lhs, const T&) {
    lhs.this_type_does_not_support_comparisons();
    return false;
}
int main() {
    Testable t1;
    AnotherTestable t2;
    if (t1) {} // Works as expected
    if (t2 == t1) {} // Fails to compile
    if (t1 < 0) {} // Fails to compile
    return 0;
}
```