Лекция 12. Шаблоны. Умные указатели

Александр Смаль

CS центр
7 ноября 2017
Санкт-Петербург

Проблема “одинаковых классов”

```
struct ArrayInt {
explicit ArrayInt(size_t size)
    : data_(new int[size])
    , size_(size) {}

~ArrayInt() {delete [] data_;}

size_t size() const
{ return size_; }

int operator[](size_t i) const
{ return data_[i]; }

int & operator[](size_t i)
{ return data_[i]; }
...
private:
    int *   data_;
    size_t  size_;
};
```

```
struct ArrayFlt {
explicit ArrayFlt(size_t size)
    : data_(new float[size])
    , size_(size) {}

~ArrayFlt() {delete [] data_;}

size_t size() const
{ return size_; }

float operator[](size_t i) const
{ return data_[i]; }

float & operator[](size_t i)
{ return data_[i]; }
...
private:
    float  * data_;
    size_t   size_;
};
```

Решение в стиле C: макросы

```
#define DEFINE_ARRAY(Name, Type)\
struct Name {                   \
explicit Name(size_t size)      \
    : data_(new Type[size])     \
    , size_(size) {}            \
~Name() { delete [] data_; }    \
                                \
size_t size() const             \
{ return size_; }               \
                                \
Type operator[](size_t i) const \
{ return data_[i]; }            \
Type & operator[](size_t i)     \
{ return data_[i]; }            \
...                             \
private:                        \
    Type *  data_;              \
    size_t  size_;              \
}
```

```
DEFINE_ARRAY(ArrayInt, int);
DEFINE_ARRAY(ArrayFlt, float);

int main()
{
    ArrayInt ai(10);
    ArrayFlt af(20);
    ...
    return 0;
}
```

Решение в стиле C++: шаблоны классов

```
template <class Type>
struct Array {
explicit Array(size_t size)
    : data_(new Type[size])
    , size_(size) {}
~Array() { delete [] data_; }

size_t size() const
{ return size_; }

Type operator[](size_t i) const
{ return data_[i]; }
Type & operator[](size_t i)
{ return data_[i]; }
...
private:
    Type *  data_;
    size_t  size_;
};
```

```
int main()
{
    Array<int> ai(10);
    Array<float> af(20);
    ...
    return 0;
}
```

Шаблоны классов с несколькими параметрами

```
template <class Type,
          class SizeT = size_t,
          class CRet = Type>
struct Array {
explicit Array(SizeT size)
    : data_(new Type[size])
    , size_(size) {}
~Array() {delete [] data_;}

SizeT size() const {return size_;}
CRet operator[](SizeT i) const
{ return data_[i]; }
Type & operator[](SizeT i)
{ return data_[i]; }
...
private:
    Type *  data_;
    SizeT   size_;
};
```

```
void foo()
{
    Array<int> ai(10);
    Array<float> af(20);
    Array<Array<int>,
          size_t,
          Array<int> const&>
        da(30);
    ...
}

typedef Array<int> Ints;
typedef Array<Ints, size_t,
    Ints const &> IInts;

void bar()
{
    IInts da(30);
}
```

Шаблоны функций: возведение в квадрат

```
// C
int   squarei(int   x)   { return x * x; }
float squaref(float x)   { return x * x; }

// C++
int   square(int   x)  { return x * x; }
float square(float x)  { return x * x; }

// C++ + OOP
struct INumber {
    virtual  INumber * multiply(INumber * x) const = 0;
};
struct Int   : INumber { ... };
struct Float : INumber { ... };
INumber * square(INumber * x) { return x->multiply(x); }

// C++ + templates
template <typename Num>
Num square(Num x) { return x * x; }
```

Шаблоны функций: сортировка

```
// C
void qsort(void * base, size_t nitems, size_t size, /*function*/);

// C++
void sort(int    * p,   int     * q);
void sort(double * p,   double  * q);

// C++ + OOP
struct IComparable {
    virtual  int compare(IComparable * comp) const = 0;
    virtual ~IComparable() {}
};
void sort(IComparable ** p, IComparable ** q);

// C++ + templates
template <typename Type>
void sort(Type * p, Type * q);
```

NB: у шаблонных функций нет параметров по умолчанию.

Вывод аргументов (deduce)

```
template <typename Num>
Num square(Num n) { return n * n; }

template <typename Type>
void sort(Type * p, Type * q);

template <typename Type>
void sort(Array<Type> & ar);

void foo() {
    int a = square<int>(3);
    int b = square(a) + square(4); // square<int>(..)
    float * m = new float[10];
    sort(m, m + 10);     // sort<float>(m, m + 10)
    sort(m, &a);         // error: sort<float> vs. sort<int>
    Array<double> ad(100);
    sort(ad);            // sort<double>(ad)
}
```

Шаблоны методов

```
template <class Type>
struct Array {
    template<class Other>
    Array( Array<Other> const& other )
        : data_(new Type[other.size()])
        , size_(other.size()) {
        for(size_t i = 0; i != size_; ++i)
            data_[i] = other[i];
    }

    template<class Other>
    Array & operator=(Array<Other> const& other);
    ...
};

template<class Type>
template<class Other>
Array<Type> & Array<Type>::operator=(Array<Other> const& other)
{ ... return *this; }
```

Функции для вывода параметров

```
template<class First, class Second>
struct Pair {
    Pair(First const& first, Second const& second)
        : first(first), second(second) {}
    First first;
    Second second;
};

template<class First, class Second>
Pair<First, Second> makePair(First const& f, Second const& s) {
    return Pair<First, Second>(f, s);
}

void foo(Pair<int, double> const& p);

void bar() {
    foo(Pair<int, double>(3, 4.5));
    foo(makePair(3, 4.5));
}
```

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;
}
...
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;
}
```