

Шаблоны, часть 2

Александр Смаль

Академический университет
22 ноября 2013
Санкт-Петербург

Полная специализация шаблонов: классы

```
template<class T>
struct Array {
    explicit Array(size_t size)
        : size_(size)
        , data_(new T[size_])
    {}
    T&operator[](size_t i) const { return data_[i]; }
private:
    size_t size_;
    T * data_;
};
```

```
template<>
struct Array<bool> {
    explicit Array(size_t size)
        : size_(size / sizeof(int) / 8 + 1)
        , data_(new int[size_])
    {}
    bool&operator[](size_t i) const
    { return data_[i/sizeof(int)/8] & (1<<((i%(sizeof(int)*8)))); }
private:
    size_t size_;
    int * data_;
};
```

Array<bool> a(10);
a[2] = false;

Полная специализация шаблонов: функции

```
template<class T>
void swap(T & a, T & b)
{
    T tmp(a);
    a = b;
    b = tmp;
}
```

int a = 5,
int b = 6;
swap(a, b).

```
template<>
void swap<Database>(Database & a, Database & b)
{
    a.swap(b);
}
```

```
template<class T>
void swap(Array<T> & a, Array<T> & b)
{
    a.swap(b);
}
```

Специализация шаблонов функций и перегрузка

```
template<class T>  
void foo(T a, T b)  
{  
    printf("1\n");  
}
```

I

```
template<class T, class V>  
void foo(T a, V b)  
{  
    printf("2\n");  
}
```

II

```
template<>  
void foo<int, int>(int a, int b)  
{  
    printf("3\n");  
}
```

III

перезагрузка
до специализации

```
int main()  
{  
    foo(3, 4);  
    return 0;  
}
```

// foo(int, int);

Частичная специализация шаблонов

```
template<class T>
struct Array {
    explicit Array(size_t size)
        : size_(size)
        , data_(new T[size_])
    {}
    T & operator [] (size_t i) { return data_[i]; }
private:
    size_t size_;
    T * data_;
};
```

```
template<class T>
struct Array<T * > {
    explicit Array(size_t size)
        : size_(size)
        , data_(new T *[size_])
    {}
    T & operator [] (size_t i) { return *data_[i]; }
private:
    size_t size_;
    T ** data_;
};
```

template<class T
class U>
struct Array
<Pair<T, U>>

Нетиповые шаблонные параметры

```
template<class T, int N, int M>
struct Matrix
{
    T & operator()(size_t i, size_t j)
    { return data_[M * j + i]; }
private:
    T data_[N * M];
};
```

I

```
template<class T, int N, int M, int K>
Matrix<T, N, K> operator*(Matrix<T, N, M> const& a,
                          Matrix<T, M, K> const& b);
```

```
template<class T, template <class> class Container>
void print(Container<T> & c);
```

II

```
template<FILE * f>
struct Logger { ... };
```

III

Array<int> a;
print(a);
print<int, Array>(a);

Использование зависимых имён

```
template<class T>
struct Array {
    explicit Array(size_t size)
        : size_(size)
        , data_(new T [size_])
    {}
    typedef T value_type;
    T & operator[](size_t i) { return data_[i]; }
private:
    size_t size_;
    T * data_;
};

template<class Container>
bool contains(Container const& c,
    typename Container::value_type const& v);

int main()
{
    Array<int> a(10);
    contains(a, 5);
    return 0;
}
```

Handwritten annotations:

- A blue bracket on the left side of the `Array` struct definition.
- A blue arrow points from the `typedef T value_type;` line to the `typename` keyword in the `contains` function signature.
- Green arrows point from `Array<int>` to the `Container` parameter and from `int` to the `value_type` parameter in the `contains` function signature.
- A red circle highlights the `typename` keyword.
- A green underline is under the `Container::value_type` part of the `contains` signature.
- A pink question mark is placed below the `Container::value_type` part.
- Double green slashes are drawn under the `return 0;` line in `main`.

Использование функций для вывода параметров

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

```
int main()
{
    Pair<int, double> p;
    p = makePair(3, 4.5);
    return 0;
}
```

$T \quad t;$
 $f.c(\text{makePair}(t.c(), t.c()))$
 $// p = \underline{\text{Pair}(int, double)}(3, 4.5)$

Резюме про шаблоны

- 1 Компиляция происходит в точке первого использования — *инстанцирование шаблона*.
- 2 Компиляция шаблонов ленивая — компилируются только те методы, которые используются.
- 3 В точке инстанцирования шаблон должен быть полностью известен.
- 4 Шаблоны следует определять в заголовочных файлах. Большие классы следует разделять на два заголовочных файла: описание (`array.hpp`) и реализацию (`arrayimpl.hpp`).
- 5 Частичная специализация есть только у классов.
- 6 Специализация шаблонных функций не участвует в перегрузке.
- 7 Все шаблонные функции (свободные функции и методы) — `inline`.
- 8 Специализации функций — обычные функции.

NB: Не забывайте про `typedef` и `typename` =).

Вопрос для проверки

Вопрос: может ли шаблонный метод быть виртуальным?

```
A::foo(int)();  
A::foo(double)();  
B::foo(int)();  
B::foo(short)();
```

```
struct A {  
    template <class T> void foo();  
};  
struct B: A {  
    int, short  
};
```