Шаблоны

Спасибо CSCenter

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; }
```

Полная специализация шаблонов: классы

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

Полная специализация шаблонов: функции

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

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) {
    cout << "same types" << endl;
}

template<class T, class V>
void foo(T a, V b) {
    cout << "different types" << endl;
}

template<>
void foo<int, int>(int a, int b) {
    cout << "both parameters are int" << endl;
}

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

Частичная специализация шаблонов

```
template<class T>
struct Array {
    T & operator[](size_t i) { return data_[i]; }
    ...
};

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_;
};
```

Нетиповые шаблонные параметры

Параметрами шаблона могут быть типы, целочисленные значения, указатели/ссылки на значения с внешней линковкой и шаблоны.

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

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

// log - это глобальная переменная
template<ofstream & log>
struct FileLogger { ... };
```

Шаблонные параметры — шаблоны

```
// int -> string
string toString( int i );

// работает только с Array<>
Array<string> toStrings( Array<int> const& ar ) {
    Array<string> result(ar.size());
    for (size_t i = 0; i != ar.size(); ++i)
        result.get(i) = toString(ar.get(i));
    return result;
}

// от контейнера требуются: конструктор от size, методы size() и get()
template<template <class> class Container>
Container<string> toStrings( Container<int> const& c) {
    Container<string> result(c.size());
    for (size_t i = 0; i != c.size(); ++i)
        result.get(i) = toString(c.get(i));
    return result;
}
```

Использование зависимых имён

```
template<class T>
struct Array {
    typedef T value_type;
    ...
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;
}
```

Использование функций 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));
}
```

Компиляция шаблонов

- Шаблон независимо компилируется для каждого значения шаблонных параметров.
- Компиляция (*инстанциирование*) шаблона происходит в точке первого использования — *точке инстанциирования шаблона*.
- Компиляция шаблонов классов — ленивая, компилируются только те методы, которые используются.
- В точке инстанциирования шаблон должен быть полностью определён.
- Шаблоны следует определять в заголовочных файлах.
- Все шаблонные функции (свободные функции и методы) являются `inline`.
- В разных единицах трансляции инстанциирование происходит независимо.

Резюме про шаблоны

- Большие шаблонные классы следует разделять на два заголовочных файла: объявление (`array.hpp`) и определение (`array_impl.hpp`).
- Частичная специализация и шаблонные параметры по умолчанию есть только у шаблонов классов.
- Вывод шаблонных параметров есть только у шаблонов функций.
- Предпочтительно использовать перегрузку шаблонных функций вместо их полной специализации.
- Полная специализация функций — это обычные функции.
- Виртуальные методы, конструктор по умолчанию, конструктор копирования, оператор присваивания и деструктор не могут быть шаблонными.
- Используйте `typedef` для длинных шаблонных имён.