Лекция 2. Практическое использование шаблонов C++

Разработал: Ивин В.В. [vivin@dev.rtsoft.ru]

Версия 1. 03.07.2017

RTSoft

Цели

- Знакомство с приемами использования шаблонов C++
- Знакомство с улучшениями в синтаксисе C++1x

Рекомендуемая литература

- Alexandrescu A. *Modern C++ Design: Generic Programming and Design Patterns Applied* [AA_MCD]
- Meyers S.
  - *Effective Modern C++*
  - *Effective STL*
  - *Effective C++ 3rd Edition*
- Sutter H.
  - *Exceptional C++*
  - *More Exceptional C++* [HS_MEC]
  - *Exceptional C++ Style* [HS_ECS]
- Vandevoorde D., Josuttis N. *C++ Templates: The Complete Guide*

Рекомендуемые сайты

- wikipedia.org
- boost.org
- cppreference.com
- wikibooks.org
- herbsutter.com/gotw/

Содержание

- Введение
- Особенности синтаксиса C++1x
- Приемы использования шаблонов C++
- Заключение
- Задания

Введение

- Шаблонные функции
- Шаблонные классы

Введение

Шаблонные функции

Определение шаблонной функции

```
template<typename T>
void PrintSum(T left, T right) {
    std::cout << "Sum of " << left << " and " << right
        << " is " << left + right << std::endl;
}

PrintSum<int>(1, 2);
PrintSum(1, 2);  // template argument is deduced
Sum of 1 and 2 is 3

PrintSum<double>(1.1, 2.2);
PrintSum(1.1, 2.2); // template argument is deduced
Sum of 1.1 and 2.2 is 3.3

PrintSum<int>(1.1, 2.2);
warning: 'argument' : conversion from 'double' to 'int',
possible loss of data
Sum of 1 and 2 is 3
```

Специализация шаблонной функции

- Что если требуется другой алгоритм сложения?

```
// defined as before
template<typename T> void PrintSum(T left, T right);

template<>
void PrintSum<std::complex<double>>(
    std::complex<double> left,
    std::complex<double> right) {
    std::cout << "Norm sum of " << left << " and "
        << right << " is "
        << std::norm(left) + std::norm(right) << std::endl;
}

PrintSum(1, 2); // prints as before
PrintSum(1.1, 2.2); // prints as before

PrintSum(std::complex<double>(1.1, 0.5),
    std::complex<double>(2.2, 1.5));
Norm sum of (1.1,0.5) and (2.2,1.5) is 8.55
```

Перегрузка шаблонных функций

- Но ведь **`std::complex`** уже шаблон
  - Как можно определить другой алгоритм сложения для всех возможных подстановок **`std::complex`**?

```
// defined as before
template<typename T> void PrintSum(T left, T right);

template<typename T>
void PrintSum(std::complex<T> left, std::complex<T> right) {
    std::cout << "Norm sum of " << left << " and "
    << right << " is "
    << std::norm(left) + std::norm(right) << std::endl;
}

PrintSum(1, 2); // prints as before
PrintSum(1.1, 2.2); // prints as before

PrintSum(std::complex<double>(1.1, 0.5),
    std::complex<double>(2.2, 1.5));
Norm sum of (1.1,0.5) and (2.2,1.5) is 8.55
```

Перегрузка функций в общем случае

- Пример из [HS_MEC], Item 10

```
template<typename T> void g( T );          // 2
template<typename T> void g( T* );         // 4
template<> void g<int>( int );             // 8
void g( int );                             // 10
```

- См. также [HS_ECS], Item 7

Введение

Шаблонные классы

Определение шаблонного класса

- Статический массив фиксированного размера

```
// StaticArray.h

template<typename T, size_t N>
class StaticArray
{
public:
    StaticArray();
    StaticArray(const StaticArray<T, N>& other);

    T& operator[](size_t i);
    const T& operator[](size_t i) const;

private:
    T m_data[N];
};
```

Определение функций шаблонного класса

```
template<typename T, size_t N>
StaticArray<T, N>::StaticArray () {
    for (size_t i = 0; i < N; ++i) {
        m_data[i] = T(); // default initialization
    }
}
template<typename T, size_t N>
StaticArray<T, N>::StaticArray(const StaticArray<T, N>& other) {
    for (size_t i = 0; i < N; ++i){
        m_data[i] = other.m_data[i]; // 'other' is of same type!
    }
}

template<typename T, size_t N>
T& StaticArray<T, N>::operator[](size_t i) {
        return m_data[i];
}
template<typename T, size_t N>
const T& StaticArray<T, N>::operator[](size_t i) const {
        return m_data[i];
}
```

RISoft

Использование шаблонного класса

```
#include "StaticArray.h"
#include <iostream>

int main()
{
    StaticArray<int, 5> a; // default ctor
    for (size_t i = 0; i < 5; ++i) {
        a[i] = i * i;
    }

    StaticArray<int, 5> b(a); // copy ctor
    const StaticArray<int, 5> c(a); // copy ctor
    for (size_t i = 0; i < 5; ++i) {
        std::cout << "i = " << i
            << ", b[i] = " << b[i] // op[]
            << ", c[i] = " << c[i] // op[] const
            << std::endl;
    }
}
```

Специализации шаблонного класса: частичная

```
template<size_t N>
class StaticArray<char, N>
{
    // ...
    StaticArray(const char* in);

    const char* c_str() const {
        return &(m_data[0]);
    }

private:
    char m_data[N+1]; // for trailing '\0'
};
```

Специализации шаблонного класса: полная

```
template<>
class StaticArray<char, 255>
{
    // ...
    StaticArray(const char* in);

    const char* c_str() const;

private:
    char m_data[256]; // for trailing '\0'
};
```

Еще частичная специализация шаблонного класса

- Пример из [AA_MCD], §2.10.1 (адаптирован к C++11)

```
template <typename T>
class TypeTraits {
private:
    template <class U>
    struct PToMTraits {
        const static bool result = false;
    };
    template <class U, class V>
    struct PToMTraits<U V::*> {
        const static bool result = true;
    };

public:
    const static bool is_p2mem_func = PToMTraits<T>::result;
};
```

Частичная специализация шаблонного класса в работе

```
class X {};

std::cout << std::boolalpha << "is X a p2mf?"
    << TypeTraits<X>::is_p2mem_func << std::endl;
is X a p2mf? false

using XMemF = void (X::*)();
std::cout << std::boolalpha << "is X::M a p2mf?"
     << TypeTraits<XMemF>::is_p2mem_func << std::endl;
is X::M a p2mf? true
```

Типизация шаблонных подстановок

```
StaticArray<int, 5> a;

StaticArray<int, 6> b(a); // different type!
error: cannot convert argument 1 from 'StaticArray<int,5>' to
'const StaticArray<int,6> &'

const StaticArray<double, 5> c(a); // different type!
error: cannot convert argument 1 from 'StaticArray<int,5>' to
'const StaticArray<double,5> &'
```

- Что если именно такое поведение требуется воспроизвести?

Шаблонные функции шаблонного класса

```
template<typename T, size_t N>
class StaticArray
{
public:
    // ...

    template<typename U, size_t M>
    StaticArray(const StaticArray<U, M>& other);

    // ...
};

template<typename T, size_t N>
template<typename U, size_t M>
StaticArray<T, N>::StaticArray(const StaticArray<U, M>& other) {
    for (size_t i = 0; i < std::min(N, M); ++i){
        m_data[i] = other[i]; // 'other' is of different type!
    }
    for (size_t i = std::min(N, M); i < N; ++i){
        m_data[i] = T(); // default initialization if 'other' is shorter
    }
}
```

RISoft

Расширенный StaticArray

```
StaticArray<int, 5> a;

StaticArray<int, 6> b(a); // OK, template ctor

const StaticArray<double, 5> c(a); // OK, template ctor
```

- **`std::array`** – часть стандарта (C++11)

```
// array

namespace std {
    template<class T, std::size_t N>
    class array {
    public:
        T elems[N]; // fixed-size array of elements of type T
    ...
};
```

Нетривиальное использование шаблонного класса

```
#include <vector>

template<template <typename> class Cont>
struct UseContainer {
    Cont<int> key;
    Cont<float> value;
};

template<class T>
using MyVector = std::vector<T>;

UseContainer<MyVector> uc;
```

Особенности синтаксиса C++1x

- Определение/использование шаблонов
- Шаблоны с переменным числом аргументов
- Статические операторы контроля
- Характеристики типов

Особенности синтаксиса C++1x

Определение/использование шаблонов

Двойные завершающие угловые скобки

- До C++11

```
std::vector<std::vector<int>> m;
error: '>>' should be '> >' within a nested template argument
list
```

- Начиная с C++11

```
std::vector<std::vector<int>> m;     // OK
```

Псевдонимы шаблонных имен: до C++11

```
template<typename T>
typedef std::vector<T> AnotherVectorName;
error: template declaration of 'typedef'

AnotherVectorName<int> v;
error: 'AnotherVectorName' was not declared in this scope
```

- Обходной путь

```
template<typename T>
struct AnotherVectorName {
    typedef std::vector<T> type;
};

AnotherVectorName<int>::type v; // OK
```

Псевдонимы шаблонных имен: обходной путь в шаблонных классах

- Неправильно

```
template<typename T>
class SomeClass {
    AnotherVectorName<T>::type m_data;
};
error: need 'typename' before 'AnotherVectorName<T>::type'
because 'AnotherVectorName<T>' is a dependent scope
```

- Правильно

```
template<typename T>
class SomeClass {
    typename AnotherVectorName<int>::type m_data;
};
```

Псевдонимы шаблонных имен: начиная с C++11

```
template<typename T>
using AnotherVectorName = typedef std::vector<T>;

AnotherVectorName<int> v; // OK

template<typename T>
class SomeClass {
    AnotherVectorName<T> m_data; // OK
};
```

auto и decltype: до C++11

```
template < class T, class U >
void CalculateSumAndPrint(const T& t, const U& u) {
    ??? sum = t + u;   // T, U, or X?
    std::cout << "sum is " << sum << std::endl;
}
```

- Результат зависит от выбора

```
T sum = t + u;

CalculateSumAndPrint(3.14, 'p'); // ASCII code of 'p' is 112
sum is 115.14

CalculateSumAndPrint('p', 3.14);
warning: 'initializing' : conversion from 'const double' to
'char', possible loss of data
sum is s
```

auto и decltype: до C++11

- Что делать?

```
template <class R, class T, class U>
void CalculateSumAndPrint(const T& t, const U& u) {
    R sum = t + u;
    std::cout << "sum is " << sum << std::endl;
}
```

- И кто виноват?

```
CalculateSumAndPrint<double>(3.14, 'p');
sum is 115.14

CalculateSumAndPrint<double>('p', 3.14);
sum is 115.14
```

auto и decltype: начиная с C++11

```
// with 'auto'
template <class T, class U>
void CalculateSumAndPrint(const T& t, const U& u) {
    auto sum = t + u;
    std::cout << "sum is " << sum << std::endl;
}

// with 'decltype'
template <class T, class U>
void CalculateSumAndPrint(const T& t, const U& u) {
    decltype(t+u) sum = t + u;
    std::cout << "sum is " << sum << std::endl;
}


CalculateSumAndPrint('p', 3.14);
sum is 115.14

CalculateSumAndPrint(3.14, 'p');
sum is 115.14
```

Синтаксис объявления/определения функций: до C++11

- Требуется явный тип возвращаемого значения

```
template<class L, class R>
??? Add(const L &lhs, const R &rhs) { // L, R, or X?
    return lhs + rhs;
}
```

- ... который надо указывать явно при использовании

```
template<class Ret, class L, class R>
Ret Add(const L &lhs, const R &rhs) {
    return lhs + rhs;
}

double val = Add<double>(3.14, 'p');
```

Синтаксис объявления/определения функций: начиная с C++11

- Выведение типа возвращаемого значения

```
template<class L, class R>
auto Add(const L &lhs, const R &rhs)-> decltype(lhs+rhs) {
    return lhs + rhs;
}
```

- Внимание (неправильно)

```
template<class L, class R>
decltype(lhs+rhs) Add(const L &lhs, const R &rhs) {
    return lhs + rhs;
}
error: 'lhs' was not declared in this scope
error: 'rhs' was not declared in this scope
```

Синтаксис объявления/определения функций: начиная с C++14

- Автоматическое выведение типа возвращаемого значения

```
template<class L, class R>
auto Add(const L &lhs, const R &rhs) {
    return lhs + rhs;
}
```

Особенности синтаксиса C++1x

Шаблоны с переменным числом аргументов

Кортеж

- Коллекция фиксированного размера с гетерогенными значениями
- Обобщение std::pair

```
template<class T1, class T2, class T3>
struct MyTuple {
    T1 t1;
    T2 t2;
    T3 t3;
};

             // ID,     name,     salary
typedef MyTuple<int, std::string, double> Employee;

Employee GetEmployeeRecord() {
    // ...
};
```

Реализация кортежа до C++11

- Наивная попытка определить классы поддерживаемых кортежей

```
template<typename T1, typename T2>
class MyTuple {
    T1 t1;
    T2 t2;
};

template<typename T1, typename T2, typename T3>
class MyTuple {
    T1 t1;
    T2 t2;
    T3 t3;
};
error: 'MyTuple' : too many template arguments
```

Реализация кортежа до C++11: обходной путь

- [AA_MCD], §3.13.2
  - Использование списков типов

- boost::tuple implementation v1.46.1

```
template <class T0, class T1, class T2, class T3, class T4,
          class T5, class T6, class T7, class T8, class T9>
class tuple {
...
};
```

- Число возможных аргументов ограничено *по определению* (шаблонного класса)

Реализация кортежа начиная с C++11

```
// tuple

namespace std {
    template<class... Types>
    class tuple;
};
```

- Число возможных аргументов ограничено конкретной реализацией компилятора; стандарт рекомендует 1024 для
  - Template arguments in a template declaration.
  - Recursively nested template instantiations.

Особенности синтаксиса C++1x

Статические операторы контроля

Статические операторы контроля

```
template<class T>
int SerializeToInt(T t) {
    return *(reinterpret_cast<int*>(&t));
};

template<class T>
T DeserializeFromInt(int i) {
    return *(reinterpret_cast<T*>(&i));
};

float f = 4.567f;
int f_ser = SerializeToInt(f);
float f_deser = DeserializeFromInt<float>(f_ser);
```

- Что, если `sizeof(T) != sizeof(int)`?

Статические операторы контроля: до C++11

- `BOOST_STATIC_ASSERT(condition);`
- «Самописный» код, например как в [AA_MCD], §2.1

```
template<bool>
struct CompileTimeChecker {
    CompileTimeChecker(...);
};
template<>
struct CompileTimeChecker<false> {
};

#define STATIC_CHECK(expr, msg) {\
    class ERROR_##msg {} ERROR_##dummy; \
    (void)sizeof(CompileTimeChecker<(expr) != 0>( \
        ERROR_##dummy)); \
    }
```

«Самописный» код в действии

- Итоговое сообщение об ошибке зависит от компилятора

```
template<class T>
int SerializeToInt(T t) {
    STATIC_CHECK(sizeof(T) == sizeof(int),
        T_and_int_differ_in_size);
    return *(reinterpret_cast<int*>(&t));
};

SerializeToInt(1.0);

// MS VC 2013: error: '<function-style-cast>' : cannot convert
from 'SerializeToInt::ERROR_T_and_int_differ_in_size' to
'CompileTimeChecker<false>'

// GCC 4.9.2: error: no matching function for call to
'CompileTimeChecker<false>::CompileTimeChecker(SerializeToInt(T)
[with T = double]::ERROR_T_and_int_differ_in_size&)'
```

Статические операторы контроля: начиная с C++11

```
template<class T>
int SerializeToInt(T t) {
    static_assert(sizeof(int) == sizeof(T),
        "T shall be of equal size with int!");
    return *(reinterpret_cast<int*>(&t));
};

SerializeToInt(1.0);

// MS VC 2013: error: T shall be of equal size with int!

// GCC 4.9.2: error: static assertion failed: T shall be of
equal size with int!
```

Особенности синтаксиса C++1x

Характеристики типов

Характеристики типов

- C++98 содержит определение шаблонного класса `std::numeric_limits`
  - `is_signed, is_integer, has_infinity, ...`
  - `min, max, epsilon, ...`
- Хотелось бы иметь более общие характеристики типов
  - `is_integral, is_floating_point, is_enum, ...`
  - `is_fundamental, is_reference, is_object, ...`
  - `is_constructible, is_destructible, ...`
  - `...`

Характеристики типов: пример

```
template<class T, class S>
T Serialize(S source) {
    static_assert(sizeof(S) == sizeof(T),
        "Target and source types shall be of equal size!");
    return *(reinterpret_cast<T*>(&source));
};
template<class T, class S>
T Deserialize(S source) {
    static_assert(sizeof(S) == sizeof(T),
        "Target and source types shall be of equal size!");
    return *(reinterpret_cast<T*>(&source));
};

int i = 4;
float i_ser = Serialize<float>(i);
int i_deser = Deserialize<int>(i_ser);
```

- Можно ли ограничить генерацию подстановок только для целочисленных типов?

Характеристики типов: до C++11

- Библиотека Boost.TypeTraits
  - Явная специализация для каждого интегрального типа
- Самостоятельная реализация идей из [AA_MCD], §2.10 Type Traits
  - Списки типов для реализации `is_integral`

Характеристики типов: начиная с C++11

- `#include <type_traits>`

```
template<class T, class S>
T Serialize(S source) {
    static_assert(sizeof(S) == sizeof(T),
        "Target and source types shall be of equal size!");
    static_assert(std::is_integral<T>::value,
        "Target type shall be integral!");
    return *(reinterpret_cast<T*>(&source));
};

float f = 4.567f;
int f_ser = Serialize<int>(f); // OK

int i = 4;
float i_ser = Serialize<float>(i);
error: static assertion failed: Target type shall be integral!
```

Приемы использования шаблонов C++

- Производящие функции
- Классы политик
- CRTP
- SFINAE
- Шаблонное мета-программирование
- Стирание типов

RISoft

Приемы использования шаблонов C++

Производящие функции

Разные способы «связывания» шаблонов с их параметрами

- Шаблонный класс: требуется явное указание параметров при подстановке

```
std::vector<int> v;
```

- Шаблонная функция: компилятор может вывести параметры шаблона из аргументов, передаваемых при вызове функции

```
std::cout << 1 << 1.0 << std::complex<double>(1.0, -1.0);
```

Что, если объединить?

```
template<typename T>
std::vector<T> make_vector(
    typename std::vector<T>::size_type count,
    const T& value = T()) {
    return std::vector<T>(count, value);
}

auto v2 = make_vector(16, 1);
```

- Широко используемая идиома в стандартной библиотеке шаблонов C++
  - **std::make_pair**
  - **std::make_shared**
  - **std::make_unique**

Приемы использования шаблонов C++

Классы политик

Еще один взгляд на `StaticArray`

```
template<typename T, size_t N>
T& StaticArray<T, N>::operator[](size_t i) {
        return m_data[i];
}
```

- Как можно реализовать проверку выхода за границы массива?
- Как можно сделать такую проверку настраиваемой?

Классы политики BoundsCheck

```
struct ThrowExceptionBoundsChecker {
    static void Check(size_t i, size_t Length) {
        if (i >= Length) {
            throw "out of bounds";
        }
    }
};

struct DoNothingBoundsChecker {
    static void Check(size_t i, size_t Length) {
    }
};

template<typename T, size_t N,
    class BoundsCheckPolicy = DoNothingBoundsChecker>
class StaticArray
{
    // ...
};
```

Классы политики в действии

```
template<typename T, size_t N, class BoundsCheckPolicy>
T& StaticArray<T, N, BoundsCheckPolicy>::operator[](size_t i) {
    BoundsCheckPolicy::Check(i, N);
    return m_data[i];
}

StaticArray<int, 5> a1;
a1[10] = 3; // default policy: nothing done

StaticArray<int, 5, ThrowExceptionBoundsChecker> a2;
a2[10] = 3; // throws an exception at runtime
```

- Широко используются в стандартной библиотеке шаблонов C++

```
template<class T, class Allocator = std::allocator<T>>
class vector;
```

Приемы использования шаблонов C++

CRTP

CRTP

- Curiously Recurring Template Pattern

```
template<class T>
class Base {
    // ...
};

class Derived : public Base<Derived> {
    // ...
};
```

- Примеры использования (wikipedia.org)
  - Статический полиморфизм
  - Подсчет числа экземпляров класса
  - Полиморфное клонирование объекта

Non-Virtual Interface

- Также известен как паттерн проектирования «Шаблонный метод»

```
class Base {
public:
    void do_some_work() {
        // ...
        overridable_behavior();
        // ...
    }

private:
    virtual void overridable_behavior() = 0;
};

class Derived : public Base {
    void overridable_behavior() override {
        // ...
    }
};
```

Статический полиморфизм

```
template <class T>
class Base {
public:
    void do_some_work() {
        // ...
        overridable_behavior();
        // ...
    }

private:
    void overridable_behavior() {
        static_cast<T*>(this)->overridable_behavior();
    }
};

class Derived : public Base<Derived> {
    void overridable_behavior() {
        // ...
    }
};
```

Приемы использования шаблонов C++

SFINAE

SFINAE

- Substitution Failure Is Not An Error

```
struct Test {
    typedef int foo;
};

template <typename T>
void f(typename T::foo) {} // Definition #1

template <typename T>
void f(T) {}               // Definition #2

int main() {
    f<Test>(10); // Call #1.
    f<int>(10);  // Call #2 (even though there is no int::foo).
}
```

SFINAE: Почему это работает?

- Требование стандарта [temp.deduct]
  - Начиная с C++98

```
template <typename T>
void f(typename T::foo) {} // Definition #1

template <typename T>
void f(T) {}               // Definition #2

f<int>(10);

Definition #1: void f(int::foo);  // ill-formed, consider next
Definition #2: void f(int);       // OK
```

SFINAE: Где это НЕ работает?

- Если подходящая шаблонная подстановка уже определена

```
template <typename T>
void f(T) {
    typename T::foo dummy;
}

f<int>(10);

void f(int) {
    int::foo dummy; // not an SFINAE => compiler error
}
```

enable_if

- Часть стандарта C++11
- Возможность использования SFINAE без порождения дополнительных классов

```
template <bool, typename T = void>
struct enable_if {
};

template <typename T>
struct enable_if<true, T> {
    typedef T type;
};
```

enable_if: применение

- Как лучше передавать аргументы в функцию – по ссылке или по значению?

```
template<typename T>
void DoSmth(T t,
    typename std::enable_if<std::is_scalar<T>::value, T>::type*
    = nullptr) {
    std::cout << "pass by value" << std::endl;
}

template<typename T>
void DoSmth(T& t,
    typename std::enable_if<!std::is_scalar<T>::value, T>::type*
    = nullptr) {
    std::cout << "pass by reference" << std::endl;
}
```

enable_if в работе

```
DoSmth(1); // T => 'int'
struct std::enable_if<true, int> { typedef int type; };
struct std::enable_if<false, int> { };

std::is_scalar<int>::value == true
void DoSmth(int t, std::enable_if<true, int>::type* = nullptr);
void DoSmth(int& t, std::enable_if<false, int>::type* = nullptr);
pass by value

class X {};
X x;
DoSmth(x); // T => 'X'
struct std::enable_if<true, X> { typedef X type; };
struct std::enable_if<false, X> { };

std::is_scalar<X>::value == false
void DoSmth(int t, std::enable_if<false, int>::type* = nullptr);
void DoSmth(int& t, std::enable_if<true, int>::type* = nullptr);
pass by reference
```

Приемы использования шаблонов C++

Шаблонное метапрограммирование

https://en.wikipedia.org/wiki/Template_metaprogramming

- **Template metaprogramming (TMP)** is a metaprogramming technique in which templates are used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled. The output of these templates include compile-time constants, data structures, and complete functions. The use of templates can be thought of as compile-time execution.

TMP: compile-time constants

- https://en.wikipedia.org/wiki/Template_metaprogramming (адаптировано к C++11)

```
template <unsigned int n>
struct factorial {
    static const unsigned int value = n *
        factorial<n - 1>::value;
};

template <>
struct factorial<0> {
    static const unsigned int value = 1;
};

auto fact0 = factorial<0>::value; // yields 1
auto fact4 = factorial<4>::value; // yields 24
```

TMP: data structures

- Пример из [AA_MCD], §2.6

```
template <typename T>
class NiftyContainer {
    std::vector<T> m_data; // if T is not polymorphic
};
```

- Модификация для полиморфного класса

```
template <typename T>
class NiftyContainer {
    std::vector<T*> m_data; // T is polymorphic
};
```

Выбор класса реализации

```
template<bool flag, typename T, typename U>
struct Select {
    typedef T Result;
};

template<typename T, typename U>
struct Select<false, T, U> {
    typedef U Result;
};

template <typename T>
class NiftyContainer {
    typedef typename Select<std::is_polymorphic<T>::value,
        T*, T>::Result DataType;
    std::vector<DataType> m_data;
};
```

- Осуществляется во время компиляции

TMP: functions

```
#include "StaticArray.h"

template<typename T>
using Vector3 = StaticArray<T, 3>;

template<typename T>
T ScalarMult(const Vector3<T>& left, const Vector3<T>& right) {
    T result = 0;
    for (size_t i = 0; i < 3; ++i) {
        result += left[i] * right[i];
    }
    return result;
}
```

- После подстановки компилятор может развернуть внутренний цикл в

```
result = left[0] * right[0] + left[1] * right[1] +
    left[2] * right[2];
```

Приемы использования шаблонов C++

Стирание типов

Идиома Handle-Body с шаблонами

```
struct Handle {
    virtual ~Handle() {}
};

template<typename T>
struct Body : Handle {
    Body(const T& in) : value(in) {
    }

    T value;
};
```

- Что нам это дает?

```
Handle* pHandle = new Body<int>(2);

auto pBody1 = dynamic_cast<Body<int>*>(pHandle);    // valid ptr
auto pBody2 = dynamic_cast<Body<double>*>(pHandle); // nullptr
```

Упрятываем Handle-Body

```
class Holder
{
public:
    template<typename T> Holder(const T& in) {
        m_content = std::make_unique<Body<T>>(in);
    }

    template<typename T> T& Get() {
        Body<T>* pContent =
            dynamic_cast<Body<T>*>(m_content.get());
        if (pContent == nullptr) {
            throw std::exception("Invalid content!");
        }
        else {
            return pContent->value;
        }
    }

private:
    std::unique_ptr<Handle> m_content;
};
```

Проверяем Holder

```
Holder h(2);

try {
    auto i = h.Get<int>();    // OK
    auto d = h.Get<double>(); // exception
}
catch (const std::exception&) {
}
```

- Фактически, «самописная» реализация
  - **boost::any**
  - **std::any** (C++17)

Заключение

Шаблоны C++

- Мощный инструмент создание эффективного и компактного кода
- Изменения в стандартах C++1x сделали написание шаблонного кода удобнее и проще

Задания

Задача 1

- Уровень сложности – средний
- Определить шаблонный класс матриц NxN и реализовать для него функцию вычисления определителя матрицы

Задача 2

- Уровень сложности – высокий
- Реализовать класс обобщенного функтора, подобный описанному в [AA_MCD], гл. 5, с помощью Variadic Templates