
# Введение в метапрограммирование

ЛЕКЦИЯ №6

# Является ли тип указателем?

## Специализация шаблонов

```
template <class T> struct is_pointer{
enum {Value = false};
};

template <class T> struct is_pointer<T*>{
enum {Value = true};
};
```

```
std::cout << is_pointer<int*>::Value?"Pointer":"Not pointer";
```

# SFINAE

## Substitution Failure Is Not An Error

https://en.cppreference.com/w/cpp/language/sfinae

Техника при которой компилятор пытаясь вывести тип для параметра шаблона встречая ошибку в конкретной специализации, не выдает ошибку пользователю, а анализирует все возможные варианты.

# enable_if.cpp

```
template <bool condition,class T>
struct enable_if{
};

template <class T>
struct enable_if<true,T>{
    using value = T; // value only in specialization
};
```

# #include<type_traits>

http://www.cplusplus.com/reference/type_traits/

## Type traits

**Primary type categories**

| | |
|---|---|
| **is_array** | Is array (class template ) |
| **is_class** | Is non-union class (class template ) |
| **is_enum** | Is enum (class template ) |
| **is_floating_point** | Is floating point (class template ) |
| **is_function** | Is function (class template ) |
| **is_integral** | Is integral (class template ) |
| **is_lvalue_reference** | Is lvalue reference (class template ) |
| **is_member_function_pointer** | Is member function pointer (class template ) |
| **is_member_object_pointer** | Is member object pointer (class template ) |
| **is_pointer** | Is pointer (class template ) |
| **is_rvalue_reference** | Is rvalue reference (class template ) |
| **is_union** | Is union (class template ) |
| **is_void** | Is void (class template ) |

**Composite type categories**

| | |
|---|---|
| **is_arithmetic** | Is arithmetic type (class template ) |
| **is_compound** | Is compound type (class template ) |
| **is_fundamental** | Is fundamental type (class template ) |

Много категорий метафункций для построения собственных шаблонов.

# type_traits.cpp

```
emplate <class T>
typename std::enable_if<std::is_array<T>::value,void>::type print(T& va
lue){
    for(auto a: value)
        std::cout << a << " ";
    std::cout << std::endl;
}

template <class T>
typename std::enable_if<std::is_pointer<T>::value,void>::type print(T&
value){
    std::cout << "pointer:" << value << std::endl;
}
```

# Variadic template

## variadic_1.cpp

```
template <class T> void print(const T& t) {
std::cout << t << std::endl;
}

template <class First, class... Rest>
void print(const First& first, const Rest&... rest) {
std::cout << first << ", ";
print(rest...); // рекурсия на стадии компиляции!
}
```

# basic_tuple.cpp

```
template <class... Ts> class tuple {
};

template <class T, class... Ts>
class tuple<T, Ts...> : public tuple<Ts...> {
    public:
    tuple(T t, Ts... ts) : tuple<Ts...>(ts...), value(t) {
    }
    tuple<Ts...> &next = static_cast<tuple<Ts...>&>(*this);
    T value;
};
```

# Что внутри?

```
class tuple {
}

class tuple<const char*> : public tuple {
  const char* value;
}

class tuple<uint64_t, const char*> : public tuple<const char*>{
 uint64_t value;
}

class tuple<double, uint64_t, const char*> : public
tuple<uint64_t, const char*>{
 double value;
}
```

# Вспомогательный тип **elem_type_holder**

```
// специальная структура для определения типа конкретного элемента в
tuple
template <size_t, class> struct elem_type_holder;

// без параметра - это тип базового класса
template <class T, class... Ts> struct elem_type_holder<0, tuple<T, T
s...>> {
  typedef T type;
};

// это тип k-го класса в цепочке наследования
template <size_t k, class T, class... Ts>
  struct elem_type_holder<k, tuple<T, Ts...>> {
  typedef  typename elem_type_holder<k - 1, tuple<Ts...>>::type type;
};
```

# Для значения 2

```
struct elem_type_holder<2, tuple<T, Ts...>> {
typedef typename elem_type_holder<1, tuple<Ts...>>::type type;
}

struct elem_type_holder<1, tuple<T, Ts...>> {
typedef typename elem_type_holder<0, tuple<Ts...>>::type type;
}

struct elem_type_holder<0, tuple<T, Ts...>> {
typedef T type;
}
```

# Получение элемента
## tuple.cpp

```
template <size_t index,class ...Ts>
typename std::enable_if<index == 0,
        typename elem_type_holder<0, tuple<Ts...>>::type&>::type
      get(tuple<Ts...>& t){
    return t.value;
}

template <size_t index,class T,class ...Ts>
typename std::enable_if<index != 0,
      typename elem_type_holder<index, tuple<T,Ts...>>::type&>::type
get(tuple<T,Ts...>& t){
    tuple<Ts...> &base = t;
    return get<index-1>(base);
}
```

# if constexpr

```
// C++17
template<size_t index,class T,class ...Ts>

auto get_c(tuple<T,Ts...>& t){
    if constexpr (index ==0)
        return t.value;
    else {
        tuple<Ts...> &base = t;
        return get_c<index-1>(base);
    }
}
```

# constexpr_if.cpp //c++17

```cpp
template <class T>
std::string to_string(T x)
{
    if constexpr (std::is_same<T, std::string>::value)
    {
        return x;
        // ERROR, if no conversion to string
    }
    else if constexpr (std::is_integral<T>::value)
    {
        return std::to_string(x); // ERROR, if x is not numeric
    }
    else
    {
        return std::string(x);
        // ERROR, if no conversion to string
    }
}
```

# constexpr_if_tuple.cpp //c++17

```
template <class T,size_t index=0>
void print_tuple(T value){
    if constexpr(index < std::tuple_size<T>::value){
        std::cout << std::get<index>(value) << " ";
        print_tuple<T,index+1>(value);
    } else {
        std::cout << std::endl;
    }
}
```

# CRTP.cpp

```
template <class T> class base{
};

class derived : public
base<derived> {
};
```

Такая конструкция делает возможным обращение к производному классу из базового!

# Множественное наследование в шаблонах
variadic_2.cpp

```
template <typename... BaseClasses>
class Printer : public BaseClasses... {
public:
   Printer(BaseClasses&&... base_classes) :
          BaseClasses(base_classes)...{
          }
};
```

# Шаблоны в качестве параметров шаблонов

## template_parameter.cpp

Шаблон можно указать в качестве параметра шаблона!
Все типы, которые используются при конструировании нового типа с помощью шаблона – должны быть его параметрами.

```
template <class T> class Payload{
// …
};
template <template <class> class PL,
class T> class Printer{
// …
};

Printer<Payload, int> printer;
```

# alias.cpp

```cpp
template <class A, class B>
class Pair
{
public:
    A a;
    B b;
    Pair(A v1, B v2) : a(v1), b(v2)
    {
        std::cout << a << "," << b << std::endl;
    };
};

template <class A>
using SamePair = Pair<A, A>;
```

# Templates
# две модели

1. Наиболее популярный подход – модель включения(inclusion model), определения шаблонов полностью размещаются в заголовочном файле.
2. Модель явного инстанцирования (explicit instantiation model), как правило реализуется директивой явного инстанцирования (explicit instantiation directive).

# Inclusion model

И объявление и описание шаблона располагается в header файле (.h)
Фактически, при любом подключении к .cpp файлу – это будет новый шаблон для компилятора.
Минус такой модели в том, что трудно читать код (все перемешано).

# explicit instantiation model
## explicit.h explicit_main.cpp explicit.cpp

```
template <class T> class MyStack {
private:
    struct StackItem {
        StackItem *next;
        T          item;
        StackItem(T value) : item(value), next(nullptr)
{};
    };

    StackItem* current;

public:
    MyStack(void);
    void push(T item);
    void pop(void);
    T    top();
    bool empty();

    ~MyStack(void);
};
```

```
template class MyStack<int>;
template class MyStack<int*>;
```

# Разберем задание лабораторной работы laba_04_2020

Разработать шаблоны классов согласно варианту задания.
Параметром шаблона должен являться скалярный тип данных задающий тип данных для оси координат.
Классы должны иметь только публичные поля. В классах не должно быть методов, только поля.
Фигуры являются фигурами вращения (равнобедренными), за исключением трапеции и прямоугольника.
Для хранения координат фигур необходимо использовать шаблон std::pair.

Необходимо реализовать две шаблонных функции:

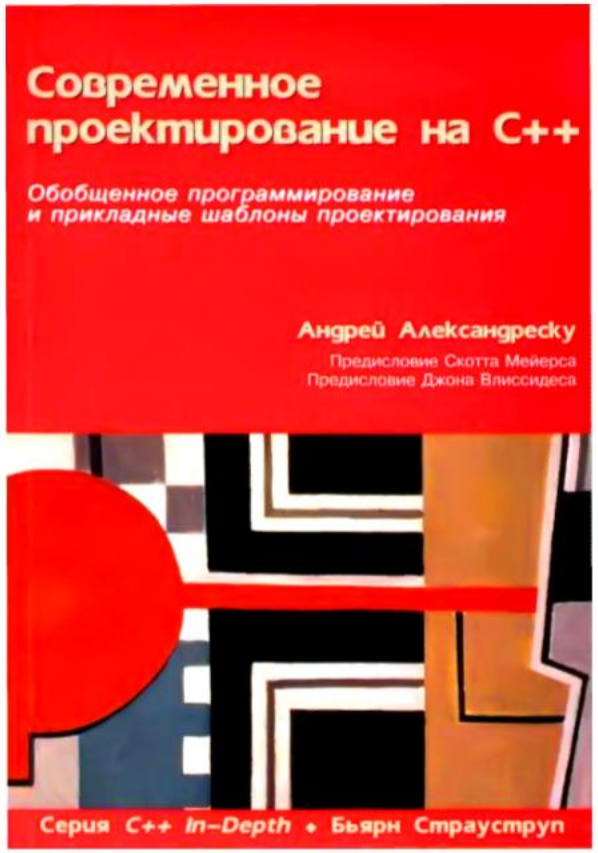
1. Функция print печати фигур на экран std::cout (печататься должны координаты вершин фигур).
2. Функция должна принимать на вход std::tuple с фигурами, согласно варианту задания (минимум по одной каждого класса).
3. Функция square вычисления суммарной площади фигур.
4. Функция должна принимать на вход std::tuple с фигурами, согласно варианту задания (минимум по одной каждого класса).

Создать программу, которая позволяет:

- Создает набор фигур согласно варианту задания (как минимум по одной фигуре каждого типа с координатами типа int и координатоми типа double).
- Сохраняет фигуры в std::tuple
- Печатает на экран содержимое std::tuple с помощью шаблонной функции print.
- Вычисляет суммарную площадь фигур в std::tuple и выводит значение на экран.

При реализации шаблонных функций допускается использование вспомогательных шаблонов std::enable_if, std::tuple_size, std::is_same.

# Книга про шаблоны




Год выпуска: 2002
Автор: Андрей Александреску
Жанр: Программирование [C++]
Издательство: Издательский дом "Вильямс" Москва - Санкт-Петербург - Киев
ISBN: 5-8459-0351-3(рус), 0-201-77581-6(англ.)
Количество страниц: 326
Описание: В книге изложена новая технология программирования, представляющая собой сплав обобщённого программирования шаблонов и объектно - ориентированного программирования на C++. Обобщённые компоненты, созданные автором, высоко подняли уровень абстракции, наделив язык C++ чертами языка спецификации проектирования, сохранив всю его мощь и выразительность
В книге изложены способы реализации основных шаблонов проектирования. Разные компоненты воплощены в библиотеке Loki, которую можно загрузить с Web-страницы автора. Книга предназначена для опытных программистов на C++


# Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ