



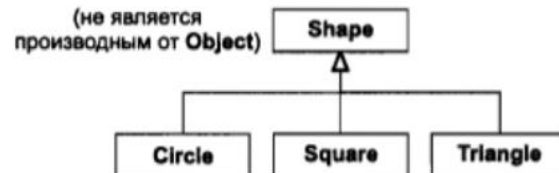
Введение в метапрограммирование

ЛЕКЦИЯ №5

Два вида многократного использования кода

Наследование

- Создаем структуру для работы с «базовым классом»
- Создаем классы-наследники на каждый случай.



Шаблоны

- Описываем «стратегию работы» с «неопределенным» классом.
- Компилятор в момент создание класса по шаблону, сам создает нужный «код» для конкретного класса.



override.cpp

```
void foo(){  
    std::cout << "foo()" << std::endl;  
}  
  
void foo(int a){  
    std::cout << "foo(int)" << std::endl;  
}  
  
void foo(double a){  
    std::cout << "foo(double)" << std::endl;  
}
```

Template это ...

- Шаблон – это параметрическая функция или класс.
- Параметром может являться как значение переменной (как в обычных функциях) так и тип данных.
- Параметры подставляются на этапе компиляции программы.
- Подставляя параметры в шаблон – мы конструируем новый тип данных (или функцию, если это шаблон функции)

C++ Core Guidelines

T.120: Use template metaprogramming only when you really need to

<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rt-metameta>

simple_template.cpp

```
template <class T>
struct Container{
    T payload;
    Container(const T& value) : payload(value){};
};

template <class T>
void print(T value){
    std::cout << "Value:" << value << std::endl;
}

template<int V>
struct foo{
    static const int value = V;
};
```

Template

Перед описанием класса ставим ключевое слово `template <class T>` или `template <typename T>`

T – используем вместо имени класса, который будет заменяться при создании конкретного экземпляра класса.

```
template <class T> struct print {};
```

```
// print– это шаблон
```

```
// print<int> - это класс, сконструированный по шаблону
```

Template vs Class

```
struct foo {  
    static const int value = 10;  
};  
  
template<int V>  
struct foo {  
    static const int value = V;  
};  
  
foo::value; // есть всегда  
foo<10>::value; // появляется при использовании
```


two_arguments.cpp

```
// два параметра, через запятую
template <class A, class B>
class Sum {
private:
    A a;
    B b;
public:
    Sum(A a_value, B b_value) : a(a_value), b(b_value) {
    }
    // параметры C и D (а не A,B) поскольку это ссылка на другой шаблон
    // описанный ниже
    template <class C, class D> friend
    std::ostream& operator<<(std::ostream & os, Sum<C,D> &sum);
};
```

Параметры – переменные

Example52_ComplexParameters

```
1.template <class TYPE, TYPE def_value, size_t SIZE = 10 > class Array {
2.protected:
3.    TYPE _array[SIZE];
4.public:
5.    Array() {
6.        for (int i = 0; i < SIZE; i++) {
7.            _array[i] = def_value;
8.        }
9.    }
10.    const size_t size() {
11.        return SIZE;
12.    }
13.    const TYPE operator[](size_t index) {
14.        if ((index >= 0) && (index < SIZE)) return _array[index];
15.        else throw BadIndexException(index, SIZE);
16.    }
17.};
```

array.cpp

```
template <class TYPE, TYPE def_value, size_t SIZE = 10 > class Array {
protected:
    TYPE _array[SIZE];
public:
    Array() {
        for (int i = 0; i < SIZE; i++) {
            _array[i] = def_value;
        }
    }
    const size_t size() {
        return SIZE;
    }
    TYPE* begin() {
        return &_array[0];
    }
    TYPE* end() {
        return &_array[SIZE]; // element beyond the array
    }
    TYPE& operator[](size_t index) {
        if ((index >= 0) && (index < SIZE)) return _array[index];
        else throw BadIndexException(index, SIZE);
    }
};
```

class_specialization.cpp

```
template <class T> class MyContainer {  
    T element;  
    // тело класса  
};  
  
// Специализация для типа char  
template <> class MyContainer <char> {  
    char element;  
    // тело класса  
}
```

partial_specialization.cpp

```
template <class A,class B,class C>
struct Foo{
    A add(B b, C c){
        return static_cast<A>(b+c);
    }
};

template <class A>
struct Foo<A,char,char>{
    A add(char b,char c){
        return static_cast<A>(b+c-'0'-'0');
    }
};
```

Частичная специализация для функций

не разрешена

abs.cpp

```
namespace example{  
    // метафункция  
    template <int V>  
    struct abs{  
        static const int value = V<0 ? -V : V;  
    };  
}
```

factorial.cpp

```
template<uint64_t n>
struct fact{
    static const uint64_t value = fact<n-1>::value * n;
};

template<>
struct fact<0>{
    static const uint64_t value = 1;
};
```


types.cpp

//тип как параметр

```
template <class T>
struct is_int{
    static const bool value = false;
};

template <>
struct is_int<int>{
    static const bool value = true;
};
```

types.cpp

//тип как результат

```
template<class T>
struct remove_const {
    using type = T;
};

template<class T>
struct remove_const<const T> {
    using type = T;
};
```

inheritance.cpp

```
template <typename T>
struct type_is{
    using type = T;
};

template <bool C, class T, class F>
struct conditional : type_is<T>{
};

template <class T, class F>
struct conditional<false, T, F> : type_is<F>{
};

template <class T, class F>
struct conditional<true, T, F> : type_is<T>{
};
```



Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ