

Шаблоны и функторы

Амиран Мстоян

AI System Engineer (C++) в Huawei
Moscow Research Center



Проверка связи





Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включён звук
- обновите страницу вебинара или закройте страницу и заново присоединитесь к вебинару
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти



Поставьте в чат:

-  если меня видно и слышно
-  если нет

Амиран Мстоян

О спикере:

AI System Engineer (C++) в Huawei Moscow Research Center

Математик-алгоритмист лаборатории научного центра при МФТИ



Вспоминаем прошрое занятие

Вопрос: что такое unit-тестирование?



Вспоминаем прошрое занятие

Вопрос: что такое unit-тестирование?

Ответ: тестирование мелких компонентов программы: функций, классов



Вспоминаем прошрое занятие

Вопрос: какие 2 макроса выполняют проверку истинности выражения в библиотеке Catch2?



Вспоминаем прошрое занятие

Вопрос: какие 2 макроса выполняют проверку истинности выражения в библиотеке Catch2?

Ответ: REQUIRE, CHECK



Вспоминаем прошное занятие

Вопрос: в чём отличие между REQUIRE и CHECK?



Вспоминаем прошрое занятие

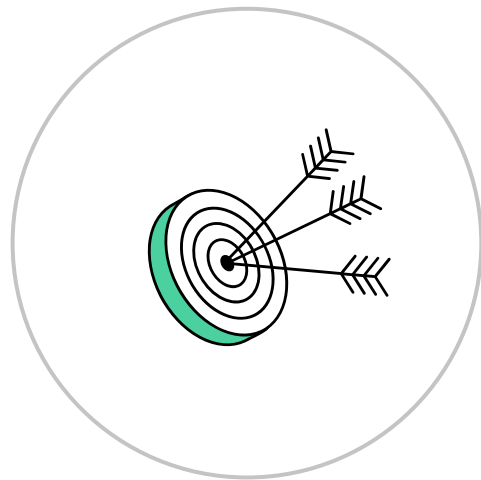
Вопрос: в чём отличие между REQUIRE и CHECK?

Ответ: после невыполнения условия в REQUIRE выполнение дальнейших тестов не происходит



Цели занятия

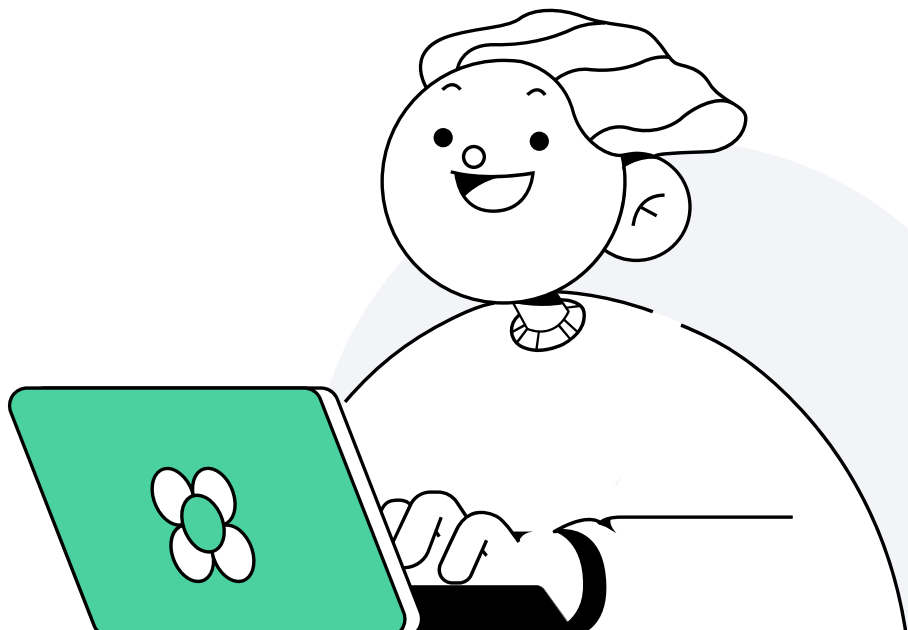
- Узнаем, зачем нужны шаблоны
- Изучим шаблоны функций
- Изучим шаблоны классов



План занятия

- 1 Шаблоны функций
- 2 Шаблоны классов
- 3 Функторы
- 4 Использование в STL
- 5 Домашнее задание

*Нажмите на нужный раздел для перехода



Зачем нужны шаблоны

Рассмотрим простую функцию:

```
int max(int a, int b)
{
    if (a > b)
        return a;
    else return b;
}
```

```
double max(double a, double b)
{
    if (a > b)
        return a;
    else return b;
}
```

```
char max(char a, char b)
{
    if (a > b)
        return a;
    else return b;
}
```

Что, если нам понадобится такая функция для других типов, например: `double`, `string`?

Зачем нужны шаблоны

Рассмотрим простую функцию:

```
int max(int a, int b)
{
    if (a > b)
        return a;
    else return b;
}
```

```
double max(double a, double b)
{
    if (a > b)
        return a;
    else return b;
}
```

```
char max(char a, char b)
{
    if (a > b)
        return a;
    else return b;
}
```

Что, если нам понадобится такая функция для других типов, например: double, string?

Придётся каждый раз определять функцию с одинаковым кодом, отличающуюся только типами возвращаемых и входных аргументов. Это плохо, так как получится дублирование кода

Зачем нужны шаблоны

Рассмотрим простую функцию:

```
int max(int a, int b)
{
    if (a > b)
        return a;
    else return b;
}
```

```
double max(double a, double b)
{
    if (a > b)
        return a;
    else return b;
}
```

```
char max(char a, char b)
{
    if (a > b)
        return a;
    else return b;
}
```

Если нам понадобится такая функция для других типов, например: double, string?

Придётся каждый раз определять функцию с одинаковым кодом, отличающуюся только типами возвращаемых и входных аргументов. Это плохо, так как получится дублирование кода.

Решение: использовать шаблоны

Шаблоны функций



1

Шаблоны функций

Для реализации шаблона необходимо:

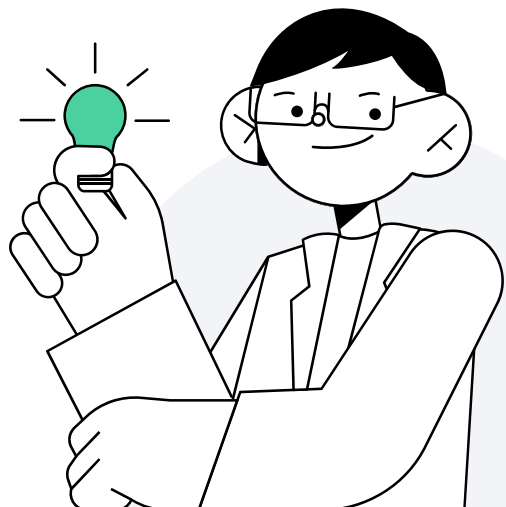
1. Написать перед объявлением функции `template <class T>` или `template<typename T>`
2. В реализации функции имя типа заменить на `T`

```
template <class T>
T my_max(T a, T b)
{
    if (a > b)
        return a;
    else return b;
}
```


Шаблоны функций

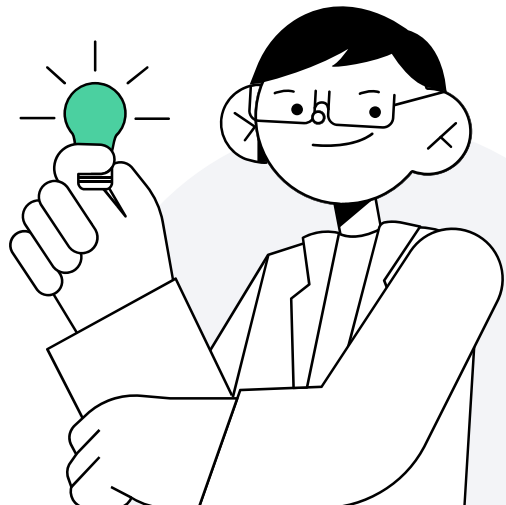
Пока нет вызова шаблонной функции, она не создаётся компилятором в бинарном файле.

При вызове функции компилятор автоматически пытается создать реализацию функции с нужными типами, а в остальном вызов шаблонной функции эквивалентен вызову обычной функции



Шаблоны функций

Шаблонную функцию необходимо определять в той единице трансляции, в которой она описана. Если вы реализуете функцию в другом файле, то получите ошибку компиляции



Вызов шаблонной функции

```
template <class T>
T my_max(T a, T b)
{
    if (a > b)
        return a;
    else return b;
}
```

Вызывать функцию можно как обычную: **my_max (0, 5)**

Однако если написать **my_max (0, 5.0)**, компилятор выдаст ошибку.

Вопрос: почему так происходит?



Вызов шаблонной функции

```
template <class T>
T my_max(T a, T b)
{
    if (a > b)
        return a;
    else return b;
}
```

Вызывать функцию можно как обычную: **my_max (0, 5)**

Однако если написать **my_max (0, 5.0)**, компилятор выдаст ошибку.

Вопрос: почему так происходит?

Ответ: компилятор сам пытается вывести тип, а на вход приходят аргументы разных типов (int, double). Можно явно писать так: **my_max<double>(0, 5.0)**



Вызов шаблонной функции

Вопрос: скомпилируется ли этот код?

```
template <class T>
T my_max(T a, T b)
{
    if (a > b)
        return a;
    else return b;
}
class dog {
    unsigned m_age;
    void bark() { std::cout << "bark!"; }
public:
    dog(unsigned age) : m_age{ age } {};
};
int main(){
    my_max(dog(1), dog(5));
    return 0;
}
```



Вызов шаблонной функции

Ответ: нет, так как у класса dog не определён оператор >

```
template <class T>
T my_max(T a, T b)
{
    if (a > b)
        return a;
    else return b;
}
class dog {
    unsigned m_age;
    void bark() { std::cout << "bark!"; }
public:
    dog(unsigned age) : m_age{ age } {};
};
int main(){
    my_max(dog(1), dog(5));
    return 0;
}
```



Специализация шаблонной функции

Бывают случаи, когда поведение, определённое шаблонной функцией для некоторых типов, работает не так, как нам хочется.

Например, строки в нашей реализации функции `my_max` будут сравниваться в лексикографическом порядке, а мы хотим их сравнивать по длине.

Вопрос: как быть в этом случае?

Специализация шаблонной функции

Ответ

Определим свою реализацию функции для типа `string`.

Для этого **после** шаблонной функции нужно определить реализацию функции для нужного типа. Это называется **специализацией шаблона**.

Для всех остальных типов компилятор будет использовать шаблонную функцию, а для типа `string` будет использовать свою отдельную реализацию

Полная специализация шаблона

```
template <class T>
T my_max(T a, T b)
{
    if (a > b)
        return a;
    else return b;
}
```

```
template <>
std::string my_max(std::string a, std::string
b) {
    if (a.length() > b.length())
        return a;
    else return b;
}
```

Полная специализация
шаблона для типа
std::string



Полной специализация называется потому, что мы заменили параметр `T` явным типом — в нашем случае `string`

Шаблоны классов



2

Шаблоны классов

Можно создавать шаблоны не только функций, но и классов*. Шаблоны классов так же, как и функции, позволяют писать наиболее обобщённый код.

При создании экземпляра шаблонного класса компилятор автоматически пытается создать реализацию класса с нужными типами.

Синтаксис объявления шаблонных классов схож с шаблонными функциями

*С ними вы наверняка сталкивались при использовании стандартной библиотеки

Шаблоны классов. Пример

Посмотрим на примере:

```
template<class T>
class simple_array
{
public:
    simple_array(int elements_count)
        : elements(new Type[elements_count]), num(elements_count){}
    T getElement(int inIndex) const
    {
        return elements[inIndex];
    }
    void setElement(int inIndex, Type inValue)
    {
        elements[inIndex] = inValue;
    }
    ~simple_array()
    {
        delete[] elements;
    }
private:
    T* elements = nullptr;
    int num = 0;
};
```

Шаблоны классов. Пример

Теперь в массиве могут храниться элементы любого типа, и мы можем проводить с ними нужные нам операции.

Создание объекта класса будет выглядеть следующим образом:

```
simple_array<int> arr(4);  
simple_array<char> arr(1);
```

Шаблоны классов. Больше аргументов

Аргументов может быть много

```
template <class T, class U> class test_class {  
    T x;  
    U y;  
  
public:  
    test_class () { cout << "Constructor Called" << endl; }  
};  
  
int main()  
{  
    test_class <char, std::string> a;  
    test_class <int, double> b;  
    return 0;  
}
```

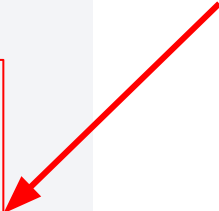
Шаблоны классов. Полная специализация

```
template <class T>
class Test
{
public:
    Test(){
        cout << "Class template \n";
    }
};

template <>
class Test <float>
{
public:
    Test() {
        cout << "Float template specialization\n";
    }
};

int main() {
    Test<int> a;
    Test<char> b;
    Test<float> c;
    return 0;
}
```

Полная специализация
шаблона для типа float



Вывод программы:
Class template
Class template
Float template
specialization

Шаблоны классов. Частичная специализация

Частичная специализация — обобщение полной специализации. Например, вы хотите специализировать только один из аргументов шаблона, а другие оставить шаблонными

```
template<class T1, class T2, int I>
class A {}; // шаблон

template<class T, int I>
class A<T, T*, I> {}; // #1: частичная специализация, где T2 — указатель на T1

template<class T, class T2, int I>
class A<T*, T2, I> {}; // #2: частичная специализация, где T1 — указатель

template<class T>
class A<int, T*, 5> {}; // #3: частичная специализация, где
// T1 — int, T2 — указатель, I — 5
```


Повторим пройденный материал

Вопрос: что нужно написать, чтобы объявить шаблон функции или класса?



Повторим пройденный материал

Вопрос: что нужно написать, чтобы объявить шаблон функции или класса?

Ответ: `template <class T>`



Повторим пройденный материал

Вопрос: как объявить полную специализацию шаблона?



Повторим пройденный материал

Вопрос: как объявить полную специализацию шаблона?

Ответ: `template<>`



Перерыв



Функторы

3





Функтор — функциональный объект. Это класс, в котором определён оператор ()

Зачем это нужно

Если мы определим оператор (), то получим объект, который действует, как функция, но может также хранить состояние. Например, можем хранить количество вызовов функтора

```
class simple_functor {
    std::string m_name;
    int m_counter; // количество вызовов функтора
public:
    simple_functor(const std::string& name) : m_name(name), m_counter{
0 } {}
    void operator()() { // определение оператора круглых скобок
        std::cout << "Hello, " << m_name << std::endl;
        m_counter++;
    }
};

int main() {
    simple_functor sf("Alex");
    sf();
    sf();
    return 0;
}
```


Зачем это нужно

Вопрос: чему будет равно количество вызовов после выполнения программы?

```
class simple_functor {
    std::string m_name;
    int m_counter; // количество вызовов функтора
public:
    simple_functor(const std::string& name) : m_name(name), m_counter{
0 } {}
    void operator()() { // определение оператора круглых скобок
        std::cout << "Hello, " << m_name << std::endl;
        m_counter++;
    }
};

int main() {
    simple_functor sf("Alex");
    sf();
    sf();
    return 0;
}
```



Зачем это нужно

Ответ: 2

```
class simple_functor {
    std::string m_name;
    int m_counter; // количество вызовов функтора
public:
    simple_functor(const std::string& name) : m_name(name), m_counter{
0 } {}
    void operator()() { // определение оператора круглых скобок
        std::cout << "Hello, " << m_name << std::endl;
        m_counter++;
    }
};

int main() {
    simple_functor sf("Alex");
    sf(); // 1
    sf(); // 2
    return 0;
}
```



Зачем это нужно

Мы можем вызывать sf(), так как мы определили оператор () для класса. Хранить количество переменных в классе удобнее, чем оперировать какими-то глобальными переменными

```
class simple_functor {
    std::string m_name;
    int m_counter; // количество вызовов функтора
public:
    simple_functor(const std::string& name) : m_name(name), m_counter{
0 } {}
    void operator()() { // определение оператора круглых скобок
        std::cout << "Hello, " << m_name << std::endl;
        m_counter++;
    }
};

int main() {
    simple_functor sf("Alex");
    sf(); // m_counter = 1
    sf(); // m_counter = 2
    return 0;
}
```

Функтор. Ещё пример

Также можно передавать аргументы в наш функтор

```
#include <iostream>
#include <string>

class simple_functor {
    int m_counter;
public:
    simple_functor() : m_counter{ 0 } {}
    int operator()(int a, int b) {
        m_counter++;
        return a + b;
    }
};

int main() {
    simple_functor sf;
    auto result1 = sf(1, 4);
    auto result2 = sf(2, 7);
    return 0;
}
```

Функтор и лямбда-функция

Лямбда-функции, с которыми вы сталкивались ранее, — это тоже функтор. Функтор, который не имеет названия. Некоторый «синтаксический сахар».

```
[](const exam_results& lhs, const exam_results& rhs){  
    return lhs.score > rhs.score;  
}  
);
```

Вы просто отдельно реализуете тело функции — реализацию оператора (). Это сделано, чтобы ваш код был чище

Функтор. Преимущества

1. Функтор можно параметризовать при создании объекта, используя конструктор
2. Создаётся временный объект на время вызова функции
3. Можно хранить множество дополнительной информации

Использование в STL



4

Использование в качестве предикатов

Предикат — это функция или функциональный объект, который обычно имеет булев тип, используется для настройки алгоритмов. Например, можно создать функтор, в котором определить нужный порядок сортировки и передать функтор в `std::sort`.

```
class simple_functor {
public:
    simple_functor() = default;
    int operator()(int a, int b) {
        return a > b;
    }
};

int main() {
    std::vector<int> vec{ 1, 10, 7, -9, 4 };
    std::sort(vec.begin(), vec.end(), simple_functor());
}
```

Вопрос: в каком порядке будут храниться элементы вектора?

Использование в качестве предикатов

Предикат — это функция или функциональный объект, который обычно имеет булев тип, используется для настройки алгоритмов. Например, можно создать функтор, в котором определить нужный порядок сортировки и передать функтор в `std::sort`.

```
class simple_functor {
public:
    simple_functor() = default;
    int operator()(int a, int b) {
        return a > b;
    }
};

int main() {
    std::vector<int> vec{ 1, 10, 7, -9, 4 };
    std::sort(vec.begin(), vec.end(), simple_functor());
}
```

Ответ: в порядке убывания

Использование для настройки алгоритмов

С помощью функторов можно определять то, что будет делать алгоритм.

Например, используя функцию transform, можно изменить разом все элементы массива

```
class simple_functor {
    int value_to_add; // что будем добавлять к элементам массива
public:
    simple_functor(int value) : value_to_add{ value } {};
    int operator()(int array_elem) {
        return array_elem + value_to_add;
    }
};

int main() {
    std::vector<int> vec{ 1, 10,7,-9,4 };
    std::transform(vec.begin(), vec.end(), vec.begin(), simple_functor(4));
    // содержимое вектора теперь: 5, 14, 11, -5, 9
    return 0;
}
```

Итоги



Итоги занятия

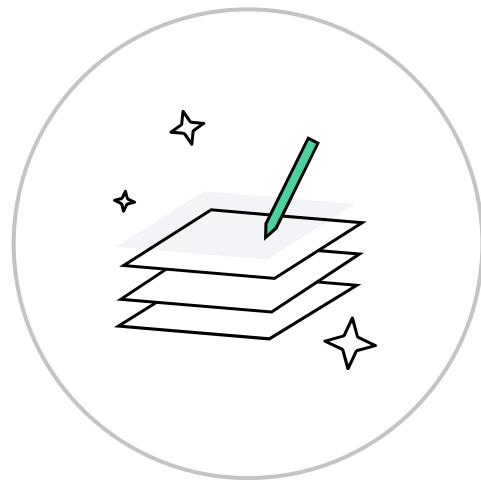
- 1 Познакомились с шаблонами функций и классов
- 2 Разобрались с понятиями полной и частичной специализации
- 3 Познакомились с функторами
- 4 Посмотрели основные сценарии использования



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- 1 Вопросы по домашней работе задавайте в чате группы
- 2 Задачи можно сдавать по частям
- 3 Зачёт по домашней работе ставят после того, как приняты все задачи



Дополнительные материалы

- [Статья](#) о шаблонах



Задавайте вопросы и пишите отзыв о лекции

Амиран Мстоян

AI System Engineer (C++) в Huawei
Moscow Research Center

