

C++

Лекция VIII

Templates

Зачем нужны шаблоны?

Мотивация

Функция `min`

```
long min(long a, long b) {  
    return (a < b) ? a : b;  
}  
double min(double a, double b) {  
    return (a < b) ? a : b;  
}  
// аналогично для других типов параметров
```

Проблема: чтобы функция поддерживала новый тип, может потребоваться новая перегрузка функции с аналогичной реализацией

Решение 1 — Макросы

Механизм из языка C.

Использование макроса решает указанную ранее проблему:

```
#define MY_MIN(a, b) (a < b) ? a : b
int main() {
    std::cout << MY_MIN(1, 2);
}
```

КАК ЭТО РАБОТАЕТ

Макрос выполняют подстановку текста во время препроцессинга.

```
clang++ -E main.cpp
```

Output:

```
int main() {
    std::cout << (1 < 2) ? 1 : 2;
}
```

Проблемы с решением 1

имена макросов не связаны с пространствами имен

```
namespace ns {  
#define MY_MIN(a, b) (a < b) ? a : b  
}  
int main() {  
    std::cout << MY_MIN(1, 2); // OK  
}
```

Проблемы с решением 1

написание макросов не так просто, как кажется на первый взгляд

```
#define MY_MIN(a, b) (a < b) ? a : b
int main() {
    int i = 1;
    std::cout << MY_MIN(3 | 1, ++i);
}
```

После препроцессинга:

```
int main() {
    int i = 1;
    std::cout << (3 | 1 < ++i) ? 3 | 1 : ++i;
}
```

Получили код, который явно не ожидали.

Исправить можно, но текст макроса значительно усложнится.

Проблемы с решением 1

```
#define MY_MIN(a, b) (a < b) ? a : b
int main() {
    int i = 1;
    double d = 0;
    std::cout << MY_MIN(i, d);
}
```

Макросы не работают с типами:

- сложно задать ограничения на типы ¹
- нельзя воспользоваться механизмом перегрузки ²

(1) например, чтобы типы выражений `a` и `b` были одинаковы в `MY_MIN`

(2) для некоторых типов выражений `a` и `b` может быть особая реализация функции

Выводы 1

- Макросы хоть и помогли решить исходную проблему, но добавили и новых проблем.
- Макросы иногда полезны ¹, но их использование может привести к ошибкам, которые сложно найти.
- *Используйте макросы только в самом крайнем случае.*

В этом семестре макросы подробно рассматриваться не будут.

Решение 2 — Templates

```
template <class T>
const T& my_min(const T& a, const T& b) {
    return (a < b) ? a : b;
}

int main() {
    std::cout << my_min(1, 2);
}
```

Решение 2 — Templates

- Избавляет от дублирования кода
- Позволяет добавить специализацию для конкретных типов
- Типобезопасность
- Компилятор проверяет, что код, не зависящий от шаблонных параметров, написан верно
- Отсутствует ряд проблем присущих макросам:
 - учитывает пространства имен
 - выражения в аргументах вычисляются однажды непосредственно до вызова
 - перегрузка

Решение 2 — Templates

Минусы

- Увеличивает время компиляции
- Трудночитаемые сообщения об ошибках
- Механизм шаблонов сложен: поэтому такой код сложно как писать, так и читать

Templates - I

Общие сведения

Общие сведения

Шаблон задает семейство сущностей¹.

Пример:

```
template <class T>
const T& my_min(const T& a, const T& b) {
    return (a < b) ? a : b;
}
```

Шаблон задает семейство функций, определяющих минимальный элемент.

Подставляя конкретный тип вместо шаблонного параметра T, получаем сущность из этого семейства.

(1) Сущность -- свободная функция, функция-член, класс или переменная

Общие сведения

Сам по себе шаблон не является какой-либо сущностью.

Его код используется для генерации кода сущностей.

Чтобы код был сгенерирован, шаблон нужно **инстанцировать**¹.

Генерация и компиляция сгенерированного кода может значительно увеличить общее **время компиляции**.

Пример с генерацией кода: [Click me](#)

(1) подробнее инстанцирование разберем позже, этот слайд про генерацию кода

Syntax

```
template < parameter_list > declaration
```

- `declaration` — объявление класса, функции, переменной или псевдонима типа
- `parameter_list` — непустой список шаблонных параметров. Параметр может иметь одну из следующих форм:
 - `type parameter`
 - `non-type parameter`
 - `template parameter`

Пример кода: [Click me](#)

non-type parameter

Ограничения

- ссылочный тип ¹
- интегральный тип или тип перечисления
- указатель

Пример кода: [Click me](#)

(1) lvalue reference type

Инстанцирование

Создание сущности¹(специализации) по имеющемуся шаблону.

Для того чтобы сделать это, компилятор подставляет известные на этапе компиляции аргументы вместо шаблонных параметров.

Инстанцирование может быть **явным** и **неявным**.

Пример кода с неявным инстанцированием: [Click me](#)

Пример кода с явным инстанцированием: [Click me](#)

(1) Сущность -- свободная функция, функция-член, класс или переменная

Explicit instantiation

Syntax

Функция:

```
template return-type name < argument-list > ( parameter-list ) ;  
template return-type name ( parameter-list ) ;
```

Класс:

```
template class-key template-name < argument-list > ;
```

Явная специализация

Позволяет задать особую реализацию сущности при определенных аргументах шаблона. Например:

```
template <class T>
void f(const T& t) {
    std::cout << "template implementation\n";
}

template <>
void f<double>(const double& d) {
    std::cout << "specific implementation\n";
}

int main()
{
    f(10.0); // specific
    f(1);
}
```

Полная специализация

Для всех параметров шаблона фиксируется значение

Syntax:

```
template <> declaration
```

Частичная специализация

Используется для фиксации значений или "уточнения" непустого подмножества параметров шаблона.

Применимо только к шаблонам классов и переменных¹.

Syntax:

```
template <param-list> class-key class-head-name <arg-list> declaration  
template <param-list> decl-specifier-seq declarator <arg-list> [initializer]
```

Пример кода: [Click me](#)

(1) Для функций не поддерживается. Считается, что перегрузки достаточно.

Значения по умолчанию

```
template <class T = void, int I = 32>
struct S {
    using type = T;
    static constexpr int value = I;
};

int main()
{
    // <> after S is necessary
    static_assert(std::is_same<S<>::type, void>::value)
    static_assert(S<>::value == 32);
}
```

Общие сведения

- Шаблоны обеспечивают статический полиморфизм
- Шаблон, используемый в нескольких единицах трансляции, следует поместить в заголовочном файле

Аргументы шаблонных функций

Вывод типов

при выводе типов обрасываются крайние ref и cv-qualifiers

```
template<class T>
T max(T a, T b) {
    // static_assert(is_same_v<T, SomeExpectedType>);
    return a < b ? b : a;
}

max(42, 10);           // max<int>
const double& x = 3.14;
max(x, x);             // max<double>
```

Перегрузка функций и шаблоны

```
template<class T> T norm(T);  
template<class T> T norm(point<T>);  
double norm(double);
```

```
norm(-2);           // ?  
norm(point<double>(3, 17)); // ?  
norm(3.14);        // ?
```

```
template<class T> T norm(T);  
template<class T> T norm(point<T>);  
double norm(double);  
  
norm(-2); // norm<int>  
norm(point<double>(3, 17)); // norm<point<T>>  
norm(3.14); // norm(double)
```

Шаблонные классы

```
template<class T>
struct vector {
    // don't need to write vector<T> inside
    vector& operator=(vector const& other);

    T const& operator[](size_t index) const;
    size_t size() const;
private:
    T* data_;
    size_t size_;
};
```

Шаблонные методы

```
template<class T>
struct vector {
    template<class U>
    void push_back(const U&);
};
```

- шаблонный метод не используется в коде \Rightarrow не инстанцируется, не компилируется
- виртуальный шаблонный метод — невозможен

Автогенерация

Для повторения:

Таблица

Автогенерация

Для повторения:

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

int main()
{
    C<int> a; // OK: Default-ctor
    C<int> b = a; // OK: Copy-ctor
    a = b; // OK: Copy-assignment
}
```

Автогенерация

Для повторения:

```
template <class T>
class C {
public:
    C(int) { }
};

int main()
{
    // C<int> c; // FAIL: Default-ctor
    C<int> a(1);
    C<int> b = a; // OK: Copy-ctor
    a = b; // OK: Copy-assignment
}
```

Автогенерация

```
template <class T>
class C {
public:
    template <class P>
    C(P) { }
};

int main()
{
    C<int> a; // FAILED
}
```

Автогенерация

```
template <class T>
class C {
public:
    template <class P>
    C(P) { }
};

int main()
{
    C<int> a(3.2); // OK
    C<int> b(a);   // OK (which one ctor selected?)
    a = b;        // OK
}
```

Автогенерация

```
#include <iostream>
#include <typeinfo>

template <class T>
class C {
public:
    template <class P>
    C(const P&) { std::cout << typeid(P).name() << std::endl; };
};

int main()
{
    C<int> a(1);           // OK
    C<int> b(a);           // OK (Autogenerated ctor)
    a = b;                // OK
}
```

Output:
i

Вывод типов для классов

- Не работает (until C++17)
- Но можно поддержать через функцию

```
template <class F, class S>
struct Pair {
    Pair(F const& first, S const& second);
};

template <class F, class S>
Pair<F, S> makePair(F const& f, S const& s) {
    return Pair<F, S>(f, s);
}

void foo(Pair<int, double> const& p);

int main() {
    foo(Pair<int, double>(3, 4.5));
    foo(makePair(3, 4.5));
}
```

Type Erasure

[CLICK ME](#)

