ШАБЛОНЫ ФУНКЦИЙ

Строительные блоки обобщения: шаблоны функций

К. Владимиров, Intel, 2019
mail-to: konstantin.vladimirov@gmail.com

- Шаблоны функций
- Вывод типов шаблонами
- Перегрузка функций и шаблонов
- Пространства имён

Возводим число в степень

- *"The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for"* – Alex Stepanov
- Начнём с первого

```
unsigned nth_power(unsigned x, unsigned n); // returns x^n
```

- Как написать тело этой функции?

Возводим число в степень

- *"The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for"* – Alex Stepanov

```
unsigned nth_power(unsigned x, unsigned n) {
  unsigned acc = 1;
  if ((x < 2) || (n == 1)) return x;
  while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { x *= x; n /= 2; }
  return acc;
}
```

- Разумеется вариант перемножить x ровно n раз в цикле не рассматривается
- Теперь настало время заняться вторым пунктом (см. след. слайд)

Возводим число в степень

- *"The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for"* – Alex Stepanov

```
unsigned nth_power(unsigned x, unsigned n) {
  unsigned acc = 1;
  if ((x < 2) || (n == 1)) return x;
  while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { x *= x; n /= 2; }
  return acc;
}
```

- Как обобщить этот алгоритм?

Возводим число в степень

- *"The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for"* – Alex Stepanov

```
template <typename T> T nth_power(T x, unsigned n) {
  T acc = 1;
  if ((x < 2) || (n == 1)) return x;
  while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { x *= x; n /= 2; }
  return acc;
}
```

- Тут всё хорошо?

Возводим число в степень

- *"The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for"* – Alex Stepanov

```
template <typename T> T nth_power(T x, unsigned n) {
  T acc = 1;
  if ((x < 2) || (n == 1)) return x;
  while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { x *= x; n /= 2; }
  return acc;
}
```

- Тут всё хорошо?
- Нет. Теперь присвоение единицы сомнительно (вдруг T это матрица?) а сравнение просто неверно (вдруг T отрицательное число?)

Возводим число в степень

- *"The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for"* – Alex Stepanov

```
template <typename T> T nth_power(T x, unsigned n) {
  T acc = id<T>();
  if ((x == acc) || (n == 1)) return x;
  while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { x *= x; n /= 2; }
  return acc;
}
```

- Это вариант предполагает, что где-то есть функция `id<T>` и она правильно работает
- Он возможен, но предъявляет многовато требований. Ещё варианты?

Возводим число в степень

- *"The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for"* – Alex Stepanov

```
template <typename T> T do_nth_power(T x, T acc, unsigned n) {
  while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { x *= x; n /= 2; }
  return acc;
}

unsigned nth_power(unsigned x, unsigned n) {
  if (x < 2u || n == 1u) return x;
  return do_nth_power<unsigned>(x, 1u, n);
}
```

- Всё ещё не совершенно, но приемлимо. Теперь надо посмотреть как это работает.

Порождение функций

```
template <typename T> T do_nth_power(T x, T acc, unsigned n) {
  while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { x *= x; n /= 2; }
  return acc;
}

unsigned do_nth_power<unsigned>(unsigned x, unsigned acc, unsigned n);

unsigned nth_power(unsigned x, unsigned n) {
  if (x < 2u || n == 1u) return x;
  return do_nth_power<unsigned>(x, 1u, n);
}

unsigned do_nth_power<unsigned>(unsigned x, unsigned acc, unsigned n) {
 while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { x *= x; n /= 2; }
  return acc;
}
```

Зависимые шаблоны

- Внутри шаблонной функции можно вызвать шаблонную функцию

```
template <typename U> void square(U &x) { x *= x; }

template <typename T> T do_nth_power(T x, T acc, unsigned n) {
  while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { square<T>(x); n /= 2; }
  return acc;
}

unsigned nth_power(unsigned x, unsigned n) {
  if (x < 2u || n == 1u) return x;
  return do_nth_power<unsigned>(x, 1u, n);
}
```

Порождение зависимых функций

```
unsigned do_nth_power<unsigned>(unsigned x, unsigned acc, unsigned n);

unsigned nth_power(unsigned x, unsigned n) {
  if (x < 2u || n == 1u) return x;
  return do_nth_power<unsigned>(x, 1u, n);
}



unsigned do_nth_power<unsigned>(unsigned x, unsigned acc, unsigned n) {
 while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { square<unsigned>(x); n /= 2; }
  return acc;
}
```

Порождение зависимых функций

```
unsigned do_nth_power<unsigned>(unsigned x, unsigned acc, unsigned n);

unsigned nth_power(unsigned x, unsigned n) {
  if (x < 2u || n == 1u) return x;
  return do_nth_power<unsigned>(x, 1u, n);
}

void square<unsigned>(unsigned &x);

unsigned do_nth_power<unsigned>(unsigned x, unsigned acc, unsigned n) {
 while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { square<unsigned>(x); n /= 2; }
  return acc;
}

void square<unsigned>(unsigned &x) { x *= x; }
```

Обсуждение

- Почему шаблоны функций нельзя добавить в язык C?

Манглирование

- Почему шаблоны функций нельзя добавить в язык C?
- Правильный ответ: шаблоны требуют манглированных имён

```
do_nth_power<unsigned>(unsigned x, unsigned acc, unsigned n)
```

- В ассемблере

```
.globl __Z12do_nth_powerIjET_S0_S0_j
```

- Язык C даёт гораздо более строгие гарантии по именам (для C++ они доступны с модификатором extern "C", но шаблоны с ним не доступны).

Предварительные объявления

- Поскольку экземпляр шаблонной функции это просто функция, она может быть предварительно объявлена

```
template <typename T> T do_nth_power(T x, T acc, unsigned n);

unsigned nth_power(unsigned x, unsigned n) {
  if (x < 2u || n == 1u) return x;
  return do_nth_power<unsigned>(x, 1u, n);
}

template <typename T> T do_nth_power(T x, T acc, unsigned n) {
  while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { x *= x; n /= 2; }
  return acc;
}
```

Многомодульные программы

- Увы, для многомодульных программ это работает со сложностями

header.h

```
template <typename T> T do_nth_power(T x, T acc, unsigned n);
```

module1.cc

```
unsigned nth_power(unsigned x, unsigned n) {
  if (x < 2u || n == 1u) return x;
  return do_nth_power<unsigned>(x, 1u, n); // требует видимого тела шаблона
}
```

module2.cc

```
int nth_power(int x, unsigned n) {
  if (x < 2 || n == 1u) return x;
  return do_nth_power<int>(x, 1, n); // требует видимого тела шаблона
}
```

Интермедия: One Definition Rule

- Что мы знаем про ODR?

Интермедия: One Definition Rule

- У каждой сущности в C++ есть declaration, когда определяется её тип и (опционально) definition, когда определяется её положение в памяти.

Declarations	Definitions
`class X;`	`class X { int foo(); }`
`int bar();`	`int bar() { return 42; }`
`extern int x;`	`int x = 20;`
`template <typename T>` `int buz(T x);`	`template <typename T>` `int buz(T x) { return x; }`

- ODR гласит: сколько угодно declarations, не более, чем один definition
- И вроде как шаблоны функций в хедере нарушат ODR?

Многомодульные программы

- Для шаблонов исключения из ODR гарантируют, что будет работать:

header.h

```
template <typename T> T do_nth_power(T x, T acc, unsigned n) {
  while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { x *= x; n /= 2; }
  return acc;
}
```

module1.cc

```
// использует do_nth_power<unsigned>(x, 1u, n);
```

module2.cc

```
// использует do_nth_power<int>(x, 1, n);}
```

Шаблонный минимум и максимум

- Классическим примером пары шаблонных функций являются

```
template <typename T> T max (T x, T y) { return x > y ? x : y; }

template <typename T> T min (T x, T y) { return x <= y ? x : y; }
```

- Например, тест некоторых их свойств это тоже шаблонная функция

```
template <typename T> bool test_minmax (const T &x, const T &y) {
  assert (x <= y);
  return (min<T> (x, y) == x) &&
         (max<T> (x, y) == y);
}
```

- Далее они будут часто использоваться как дрозофилы в мире обобщённости

Нужно ли всегда указывать типы?

- В рассмотренных выше примерах всегда явно указывались типы

```
unsigned nth_power(unsigned x, unsigned n) {
  if (x < 2u || n == 1u) return x;
  return do_nth_power<unsigned>(x, 1u, n);
}
```

- Или например

```
template <typename T> bool test_minmax (const T &x, const T &y) {
  return (min<T> (x, y) == x) &&
         (max<T> (x, y) == y);
}
```

- Но в общем эти типы ясны из контекста. Неужели компилятор не способен сам о них догадаться?

- Шаблоны функций
  - Вывод типов шаблонами
- Перегрузка функций и шаблонов
- Пространства имён

Вывод типов до подстановки

- Для параметров, являющихся типами, работает вывод типов

```
int x = max (1, 2); // → int max<int> (int, int);
```

Вывод типов до подстановки

- Для параметров, являющихся типами, работает вывод типов

```
int x = max (1, 2); // → int max<int> (int, int);
```

- При выводе режутся ссылки и внешние cv-квалификаторы

```
const int& a = 1;
const int& b = 2;
int x = max (a, b); // → int max<int> (int, int);
```

Вывод типов до подстановки

- Для параметров, являющихся типами, работает вывод типов

```
int x = max (1, 2); // → int max<int> (int, int);
```

- При выводе режутся ссылки и внешние cv-квалификаторы

```
const int& a = 1;
const int& b = 2;
int x = max (a, b); // → int max<int> (int, int);
```

- Вывод не работает, если он не однозначен

```
unsigned x = 5; do_nth_power(x, 2, n); // FAIL
int a = 1; float b = 1.0; max(a, b); // FAIL
```

Вывод типов после подстановки

- Вывод типов внутри шаблонной функции даёт точку вывода, где разрешить тип можно только после подстановки

```
template <typename T> T max (T x, T y) { return x > y ? x : y; }

template <typename T> T min (T x, T y) { return x <= y ? x : y; }

template <typename T> bool
test_minmax (const T &x, const T &y) {
  assert (x <= y);
  return (min (x, y) == x) &&
         (max (x, y) == y);
}
```

Вывод уточнённых типов

- Иногда шаблонный тип аргумента может быть уточнён ссылкой или указателем и cv-квалификатором

```
template <typename T> T max (const T& x, const T& y);
```

- В этом случае выведенный тип тоже будет уточнён

```
int a = max(1, 3); // → int max<int> (const int& x, const int& y);
```

- Уточнённый вывод иначе работает с типами: он сохраняет cv-квалификаторы

```
template <typename T> void foo (T& x);

const int &a = 3;

int b = foo(a); // → void foo<const int> (const int& x);
```

Вывод ещё более уточнённых типов

- Вывод типов работает шире, чем люди обычно думают

```
template<typename T> int foo(T(*p)(T));
int bar(int);
foo(bar); // → int foo<int>(int(*)(int));
```

- Могут быть выведены даже параметры, являющиеся константами

```
template<typename T, int N> void buz(T const(&)[N]);
buz({1, 2, 3}); // → void buz<int, 3>(int const(&)[3]);
```

Частичный вывод типов

- Возвращаемое значение из функции не создаёт контекст вывода

```
template <typename DstT, typename SrcT>
inline DstT implicit_cast (SrcT const& x) {
  return x;
}

double value = implicit_cast (-1); // fail!
```

- Но при этом возможен частичный вывод

```
double value = implicit_cast<double, int>(-1); // ok

double value = implicit_cast<double>(-1); // ok
```

Обсуждение

- Иногда возникает контекст, где при выводе хочется убрать параметр-другой

```
template <typename T> int foo (T x) {
  return 42;
}

int x = foo ();
```

- Представьте, что вы в комитете. Вы бы разрешили такое?

Tricky part: параметры по умолчанию

- Допустим у вас есть функция, берущая по умолчанию плавающее число

```
template <typename T> void foo(T x = 1.0);
```

- Увы, такой вывод работать не будет

```
foo(1); // ok, foo<int>(1);
foo(); // fail
```

- Тем не менее, ситуацию можно исправить. Трюк не так уж и сложен. Догадки?

Tricky part: параметры по умолчанию

- Допустим у вас есть функция, берущая по умолчанию плавающее число

```
template <typename T = double> void foo(T x = 1.0);
```

- Увы, такой вывод работать не будет

```
foo(1); // ok, foo<int>(1);
foo(); // ok
```

- Параметр по умолчанию шаблона в данном случае подсказывает компилятору что делать
- Главный урок тут такой: параметры по умолчанию функций не используются в выводе типов

Домашняя наработка

- Проблема гетерогенного максимума

```
template <typename T, typename U> ??? max(T t, U u) {
  return (t > u) ? t : u;
}
```

- Какой тип должна возвращать эта функция?
- Представьте что на дворе тёмный и страшный 2001-й. В списке литературы есть статья Андрея Александреску, который предлагает type traits для решения этой задачи.
- Ваши идеи?

- Шаблоны функций
- Вывод типов шаблонами
- Перегрузка функций и шаблонов
- Пространства имён

Функции могут быть перегружены

- Одно и то же имя может соответствовать многим сигнатурам.

```
float sqrt (float x);                 // 1
double sqrt (double x);               // 2
long double sqrt (long double x); // 3

sqrt(1.0);  // → 2
sqrt(1.0f); // → 1
```

- Перегрузка может создавать неоднозначности и требовать разрешения

```
sqrt(1); // → ???
```

Ограничения перегрузки

- Функция не может быть перегружена по cv-квалификаторам

```
void foo (int);
void foo (const int); // FAIL

void bar (char *);
void bar (char * const); // FAIL
```

- Однако это не относится к cv-квалификаторам внешнего типа

```
void foo (int&);
void foo (const int&); // OK

void foo (char *);
void foo (const char *); // OK
```

Разрешение перегрузки

- Для разрешения перегрузки есть набор простых мнемонических правил:
- Три правила для обычных функций
  - Идеальное совпадение выигрывает
  - Все стандартные преобразования равны
  - Троеточия проигрывают почти всему
- Три правила для шаблонов
  - Точно подходящая функция выигрывает у шаблона
  - Более специальный шаблон выигрывает у менее специального
  - Меньшее количество аргументов выигрывает против большего

Идеальное совпадение выигрывает

- Для любых функций если в множестве перегрузок, есть идеальное совпадение

```
void foo (int x);   // 1
void foo (short x); // 2

foo(1); // → 1
```

- При этом идеальное совпадение это ещё и ссылка правильного типа

```
void foo (const int& x); // 1
void foo (short x);      // 2

foo(1); // → 1
```

Идеальное совпадение выигрывает

- В том числе и обычная левая ссылка может быть точным совпадением

```
void foo (int& x);         // 1
void foo (const int& x); // 2

foo(1);          // → 2
int y; foo(y); // → 1
```

- Два идеальных совпадения это конфликт

```
void foo (const int& x); // 1
void foo (int x);          // 2

foo(1); // → FAIL
```

Все стандартные преобразования равны

- Все стандартные преобразования одноранговые

```
void foo (char x); // 1
void foo (long x); // 2

foo(1); // → FAIL
```

- Любые стандартные преобразования выигрывают у пользовательских

```
struct MyClass { MyClass(int x) {} };

void foo (char x);    // 1
void foo (MyClass x); // 2

foo(1); // → 1
```

Троеточия проигрывают почти всему

- Троеточия проигрывают и стандартным и пользовательским преобразованиям

```
void foo (int x);      // 1
void foo (long x);     // 2
void foo (MyClass x); // 3
void foo (...);        // 4

foo(1); // → 1, 2, 3 и только потом 4
```

- Они выигрывают только у неправильных ссылок

```
void foo (...);    // 1
void foo (int &x); // 2

foo(1); // → 1
```

Перегрузки можно запрещать

- Начиная с 2011 года можно явно запрещать (стирать?) перегрузки для конкретных аргументов

```
int foo (int x) { return x + 42; }

int foo (bool) = delete;
int foo (char) = delete;
```

Но они всё ещё участвуют в подстановке, так что

```
int t = foo (true); // ошибка, а не преобразование к int
```

Разрешение перегрузки

- Для разрешения перегрузки есть набор простых мнемонических правил:
- Три правила для обычных функций
  - Идеальное совпадение выигрывает
  - Все стандартные преобразования равны
  - Троеточия проигрывают почти всему
- Три правила для шаблонов
  - Точно подходящая функция выигрывает у шаблона
  - Более специальный шаблон выигрывает у менее специального
  - Меньшее количество аргументов выигрывает против большего

Точная функция выигрывает у шаблона

- Это так даже если шаблон подходит точно

```
int foo (int a);            // 1
<typename T> T foo (T a); // 2

foo(1); // → 1
```

- Но можно явно указать, что мы хотим шаблон

```
foo<>(1);     // → 2
foo<int>(1); // → 2
```

- При этом стандартные преобразования проигрывают шаблону

```
foo(1.0); // → 2
```

Tricky part: пиррова победа

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

double min(double a, double b) { return a < b ? a : b; }

template <typename T>
const T& min(const T& a, const T &b, const T &c) {
  return min(min(a,b), c);
}

double a = 1.0, b = 2.0, c = 3.0;

double m = min(a, b, c);
```

- Кто видит тут проблемы?

Тут нужно показать демо pirr.cc

Более специальный шаблон выигрывает у менее специального

```
template <typename T> void f(T);      // 1
template <typename T> void f(T*);     // 2
template <typename T> void f(T**);    // 3
template <typename T> void f(T***);   // 4
template <typename T> void f(T****); // 5

int ***a;
f(a);          // → 4
f<int**>(a); // → 2
```

Меньшее количество параметров выигрывает против большего

```
template <typename T1, typename T2> void f (T1, T2); // 1
template <typename T> void f (T, T*);                // 2

double t, s;
f (t, &s); // → 2
```

Но при конфликте с предыдущим правилом это не работает

```
template <typename T> void g (T, T);                    // 1
template <typename T1 typename T2> void g (T1, T2*);    // 2
template <typename T1 typename T2> void g (T1*, T2*); // 3

g (&t, &s); // → FAIL
```

Обсуждение

- Вы сидите в комитете и вам приносят пример:

```
class Bar {
  int foo (int);
public:
  int foo (char);
};

Bar b;

b.foo(1);
```

- Как по вашему: должна быть ошибка или вызов public функции?

Контроль доступа после перегрузки

- Вы сидите в комитете и вам приносят пример:

```
class Bar {
  int foo (int);
public:
  int foo (char);
};

Bar b;

b.foo(1);
```

- Как по вашему: должна быть ошибка или вызов public функции?
- Можно аргументировать оба решения. Сейчас по стандарту ошибка, так как контроль доступа идёт после разрешения перегрузки

Разбор примера с прошлой лекции

- В конце прошлой лекции была поставлена задача написать operator== для basic_string
- Один из простых вариантов решения

```
template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs,
                const basic_string<CharT, Traits, Alloc>& rhs) {
  return lhs.compare(rhs) == 0;
}
```

- Чем он плох?

Разбор примера с прошлой лекции

- В конце прошлой лекции была поставлена задача написать `operator==` для `basic_string`
- Один из простых вариантов решения

```
template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs,
                const basic_string<CharT, Traits, Alloc>& rhs) {
  return lhs.compare(rhs) == 0;
}
```

- Он неэффективен. Подумайте про (`"hello" == str`), тут явно создаётся лишняя копия. Мы бы хотели его перегрузить, как обычную функцию.

Лучший вариант сравнения

- Принятый (в т.ч. в libstdc++) вариант решения использует перегрузки

```
template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs,
                const basic_string<CharT, Traits, Alloc>& rhs) {
  return lhs.compare(rhs) == 0;
}

template<typename CharT, typename Traits, typename Alloc>
bool operator==(const CharT* lhs, const basic_string<CharT, Traits, Alloc>& rhs) {
  return rhs.compare(lhs) == 0;
}

template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs, const CharT* rhs) {
  return lhs.compare(rhs) == 0;
}
```

- Шаблоны функций
- Вывод типов шаблонами
- Перегрузка функций и шаблонов
- Пространства имён

Проблема конфликта имён

- Если (как это сделано в языке C) все имена принадлежат одному (в терминах C++ – глобальному) пространству имён, то неизбежны конфликты, решаемые кривым ручным манглированием

```
int zlib_open (const char *);
```

- Всем очень хорошо известно, что в C++ эти проблемы решаются пространствами имён:

```
namespace zlib {
  int open (const char *);
};
```

- Вся стандартная библиотека живёт в пространстве имён std

Правильный hello world

```
// вносим старую библиотеку обёрнутой в std
#include <cstdio>

namespace hellowapp {
  // не засоряем нашими именами
  // глобальное пространство имён
  const char * const helloworld = "Hello, world";
}

int main() {
  // явно квалифицируем функции
  std::printf("%s\n", hellowapp::helloworld);
  return 0;
}
```

Снова проблема: operator <<

- Функция `operator <<` может находиться в любом пространстве имён
- Допустим мы пишем:

```
std::cout << "Hello\n"!;
```

- Это вполне может быть эквивалентно следующему:

```
operator << (std::cout, "Hello\n"!);
```

- У нас, кажется, проблемы. Чтобы это работало, это должен быть оператор из пространства имён std

```
std::operator << (std::cout, "Hello\n"!);
```

- Но компилятор не может об этом догадаться из записи `std::a << b;`

Решение: поиск Кёнига

- Эндрю Кёниг предложил решение в начале 90-х

1. Компилятор ищет имя функции из текущего и всех охватывающих пространств имён
2. Если оно не найдено, компилятор ищет имя функции в пространствах имён её аргументов

```
namespace N { struct A; int f(A*); }
int g(N::A *a) { int i = f(a); return i; }
```

Решение: поиск Кёнига

- Эндрю Кёниг предложил решение в начале 90-х

1. Компилятор ищет имя функции из текущего и всех охватывающих пространств имён
2. Если оно не найдено, компилятор ищет имя функции в пространствах имён её аргументов

```
typedef int f;

namespace N { struct A; int f(A*); }

int g(N::A *a) { int i = f(a); return i; }
```

Поиск Кёнига и шаблоны

- Следующий пример не работает

```
namespace N {
  struct A;
  template <typename T> int f(A*);
}

int g(N::A *a){
  int i = f<int>(a); // FAIL
  return i;
}
```

- Кто-нибудь знает причину?

Поиск Кёнига и шаблоны

- Следующий пример не работает

```
namespace N {
  struct A;
  template <typename T> int f(A*);
}

int g(N::A *a){
  int i = f<int>(a);
  return i;
}
```

- Причина: странности синтаксического анализа C++. Имя f не введено как имя шаблонной функции, поэтому компилятор предполагает, что это переменная, а треугольная скобка – сравнение на меньше

Поиск Кёнига и шаблоны

- Можно заставить это работать, введя f как имя шаблонной функции

```
namespace N {
  struct A;
  template <typename T> int f(A*);
}

template <typename T> void f(int); // неважно какой параметр

int g(N::A *a){
  int i = f<int>(a); // теперь всё ok
  return i;
}
```

Литература

- ISO/IEC, Information technology – Programming languages – C++, 14882:2017
- Bjarne Stroustrup, The C++ Programming Language (4th Edition)
- Alexander A. Stepanov, Paul McJones – Elements of programming, Addison-Wesley, 2009
- Alexander A. Stepanov, Daniel E. Rose – From mathematics to generic programming, Addison-Wesley, 2014
- David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor – C++ Templates - The Complete Guide, 2nd Edition, Addison-Wesley, 2017
- Stephan T. Lavavej – Core C++, lectures 1, 2 and 3
- Andrei Alexandrescu, Generic: Min and Max Redivivus, Dr. Dobb's, 2001

Шаблоны 4x4

	Типы	**Целые числа**	**Указатели и ссылки**	**Шаблоны**
Функции				
Классы				
Синонимы типов				
Переменные				

Управление инстанцированием

- Инстанцирование может быть явно запрещено в этой единице трансялции

```
extern template int max <int> (int, int);
```

- Инстанцирование может быть явно вызвано в этой единице трансляции

```
template int max <int> (int, int);
```

- Эта техника может использоваться для уменьшения размера объектных файлов при инстанцировании тяжёлых функций

Тут нужно показать демо по управлению инстанцированием