# Шаблоны функций

# В начале была буква... (C-style функции)

```
int abs(int x) { return x > 0 ? x : -x; }

long labs(long x) { return x > 0 ? x : -x; }

long long llabs(long long x) { return x > 0 ? x : -x; }

float fabsf(float x) { return x > 0 ? x : -x; }

double fabs(double x) { return x > 0 ? x : -x; }

long double fabsl(long double x) { return x > 0 ? x : -x; }
```

# Перегрузка функций (C++)

```
int abs(int x) { return x > 0 ? x : -x; }

long abs(long x) { return x > 0 ? x : -x; }

long long abs(long long x) { return x > 0 ? x : -x; }

float abs(float x) { return x > 0 ? x : -x; }

double abs(double x) { return x > 0 ? x : -x; }

long double abs(long double x) { return x > 0 ? x : -x; }
```

# Шаблоны (C++)

```
template <class T>
T abs(T x) {
  return x > 0 ? x : -x;
}
```

# Синтаксис шаблонных функций

- В начале объявляется список шаблонных параметров:

```
template <class T>  или template <typename T>
```

- Далее следует определение (или объявление) шаблонной функции.

```
template <class T>
T Abs(T x) { return x > 0 ? x : -x; }

template <class T>
T Sum(T x, T y) { return x + y; }
```

- Допустимо использование нескольких шаблонных параметров.

```
template <class T, class U>
void Print(T x, U y) { std::cout << x << ' ' << y; }
```

# Использование шаблонов

После того как шаблон функции объявлен его можно использовать.

Нужный тип выводится автоматически по *переданным аргументам* (при наличии):

```
template <class T>
T Abs(T x) { return x > 0 ? x : -x; }

Abs(0.0);  // double Abs(double)
```

```
template <class T>
T Sum(T x, T y) { return x + y; }

Sum(1, 1);    // Ok [T == int]
Sum(1, 0.0);  // CE - конфликт ([T == int] или [T == double] ?)
```

```
template <class T>
T GetZero() { return 0; }

GetZero();  // CE - вывести тип невозможно
```

# Использование шаблонов

Если результат вывода не устраивает или вывод типа невозможен, можно заставить компилятор вызвать функцию с конкретным типом:

```
template <class T>
T Sum(T x, T y) { return x + y; }

Sum<long>(1, 1);       // Ok [T == long]
Sum<double>(1, 0.0);   // Ok [T == double]
```

```
template <class T>
T GetZero() { return 0; }

GetZero<float>(); // Ok [T == float]
```

# Вывод типа шаблона: передача по значению

# Вывод типа шаблона: передача по значению

```
template <class T>
void f(T x, T y);
```

При передаче аргумента по значению тип `T` выводится по следующим правилам:

1. CV-квалификаторы (`const`, `volatile`) игнорируются.
2. Ссылки отбрасываются.
3. Массивы низводятся до указателей.
4. Функции низводятся до указателей на функцию.
5. Типы соответствующие одному шаблонному типу `T` должны совпадать (после выполнения всех действий выше).

# Вывод типа шаблона: передача по значению

```
template <class T>
void f(T x, T y);
```

```
int x = 0;
const int cx = 1;

f(x, x);
f(x, cx);
f(cx, cx);

int& rx = x;
int arr[11];

f(cx, rx);
f(&rx, arr);
f(&cx, &x);
f(0, 0.0);

f<double>(0, 0.0);
```

# Вывод типа шаблона: передача по значению

```
template <class T>
void f(T x, T y);
```

```
int x = 0;

const int cx = 1;
f(x, x);    // Ok: [T=int]
f(x, cx);   // Ok: [T=int]
f(cx, cx);  // Ok: [T=int]

int& rx = x;
int arr[11];

f(cx, rx);   // Ok: [T=int]
f(&rx, arr); // Ok: [T=int*]
f(&cx, &x);  // CE: [T=const int*] или [T=int*]?
f(0, 0.0);   // CE: [T=int] или [T=double]?

f<double>(0, 0.0); // Ok: [T=double] (тип не выводится, а подставляется)
```

# Вывод типа шаблона: передача по значению

```
template <class T, class U>
void f(T x, U y);
```

```
// Можно вывести оба параметра (если они выводимы)
f(0, 0.0); // Ok: [T=int, U=double]

// Можно явно указать оба
f<double, double>(0, 0.0); // Ok: [T=double, U=double]

// Можно указать первый, второй выведется автоматически (если выводим)
f<float>(0, 0.0); // Ok: [T=float, U=double]
```

# Вывод типа шаблона: передача по ссылке

# Вывод типа шаблона: передача по ссылке

При передаче по ссылке или указателю низведений типов не происходит.

```
template <class T>
void f(T& x) { ... }
```

```
int x = 0; const int cx = 1; int& rx = x; int arr[10];

f(x);    // Ok: void f<int>(int& x);
f(cx);   // Ok: void f<const int>(const int& x);
f(rx);   // Ok: void f<int>(int& x); нет ссылок на ссылки
f(arr);  // Ok: void f<int[10]>(int (&x)[10])
f(0);    // CE: нельзя создавать ссылок на временные значения

f<const int>(0);  // Ok: void f<const int>(const int&)

// или

template <class T>
void f(const T& x) { ... }
```

# Параметры шаблона по умолчанию

# Параметры шаблона по умолчанию

```
template <class T>
T GetZero() { return 0; }

GetZero();           // CE: невозможно вывести тип T
GetZero<double>();   // Ok
```

Можно указать значение шаблонного параметра по умолчанию, тогда, в случае если тип невозможно вывести, будет использоваться значение по умолчанию

```
template <class T = int>
T GetZero() { return 0; }

GetZero();           // Ok: [T=int]
GetZero<double>();   // Ok
```

# Параметры шаблона по умолчанию

Можно ссылаться на предыдущие шаблонные параметры

```
template <class T, class U = T>
U f(T x) { ... }

f<int, double>(0);  // Ok: [T=int, U=double]
f<float>(0);        // Ok: [T=float, U=float]
f(0);               // Ok: [T=int, U=int]
```

# Параметры шаблона по умолчанию

**Важно!** Значения аргументов по умолчанию не могут использоваться для вывода шаблонного типа

```
template <class T>
void f(T x = 0) { ... }

f();  // CE: тип T не выведен!
```

Только так

```
template <class T = int>
void f(T x = 0) { ... }

f();  // Ok: [T=int]
```

# Инстанцирование шаблона

# Инстанцирование шаблона

Сколько экземпляров функции `f` создастся в этой программе?

```
template <class T>
void f(T x) { ... }

int main() {
  std::cout << "Hello, world!\n";
  return 0;
}
```

`f<int>`? `f<double>`? `f<const char*>`?..

# Инстанцирование шаблона

Сколько экземпляров функции `f` создастся в этой программе?

```
template <class T>
void f(T x) { ... }

int main() {
    std::cout << "Hello, world!\n";
    return 0;
}
```

Ответ: 0

# Инстанцирование шаблона

- Шаблон функции - это **не** функция!
- Генерации исполняемого кода не происходит, если шаблон ни разу не вызван.
- В случае вызова шаблонной функции создается лишь нужная версия (с вызываемыми параметрами).
- Процесс генерации кода из шаблона называется *инстанцированием* шаблона

```
template <class T>
void f(T x) { ... }

int main() {
    f(0);       // Инстанцируется f<int>
    f(0.0);     // Инстанцируется f<double>
    f(1);       // Используется f<int> (уже инстацирована)
    return 0;  // других версий f<T> не существует!
}
```

# Инстанцирование шаблона

Можно явно попросить инстанцировать шаблон (даже если он не используется).

```
template <class T>
void f(T x) { ... }

template void f(float); // Явно инстанцируем f<float>

int main() {
    f(0);       // Инстанцируется f<int>
    f(0.0);     // Инстанцируется f<double>
    f(1);       // Используется f<int> (уже инстацирована)
    return 0;  // Существуют f<int>, f<double>, f<float>
}
```

Это может быть полезно в случае многофайловых программ, когда вы не хотите инстанцировать одну и ту же функцию несколько раз в разных единицах трансляции.

# Инстанцирование шаблона

Компиляция шаблонов происходит в два этапа:

1. При определении проверяется лишь синтаксис языка и условия, которые не зависят от параметра шаблона.
2. Во время инстанцирования происходит полная проверка кода на корректность и генерация машинного кода.

```
template <class T>
void f(T x) {
    x = x + x;
    g();
    g(x);
    static_assert(sizeof(char) == 1);
    static_assert(sizeof(T) > sizeof(char));
}
```

# Инстанцирование шаблона

Компиляция шаблонов происходит в два этапа:

1. При объявлении проверяется лишь синтаксис языка и условия, которые не зависят от параметра шаблона
2. Во время инстанцирования происходит полная проверка кода на корректность и генерация машинного кода

```
template <class T>
void f(T x) {
    x = x + x;  // 2 этап (можно ли складывать и присваивать T?)
    g();        // 1 этап (не зависит от T)
    g(x);       // 2 этап (зависит от T)
    static_assert(sizeof(char) == 1);          // 1 этап (не зависит от T)
    static_assert(sizeof(T) > sizeof(char));  // 2 этап (зависит от T)
}
```

# Перегрузка шаблонов функций

# Перегрузка шаблонов функций

Как и обычные функции шаблоны можно перегружать

```
template <class T, class U>
int f(T x, U y) { return 1; }

template <class T>
int f(T x, T y) { return 2; }

int f(int x, int y) { return 3; }
```

Общие правила:

- Точные соответствия всегда побеждают остальные перегрузки.
- Если есть несколько точных соответствий, выиграет соответствие с меньшим числом подстановок и приведений типов.
- При прочих равных обычная функция предпочтительнее шаблона.

# Перегрузка шаблонов функций

```
template <class T, class U>
int f(T x, U y) { return 1; }

template <class T>
int f(T x, T y) { return 2; }

int f(int x, int y) { return 3; }
```

```
f(0, 0.0);    // ?
f(0.0, 0.0);  // ?
f(0, 0);      // ?

// А если очень хочется шаблон?
```

# Перегрузка шаблонов функций

```
template <class T, class U>
int f(T x, U y) { return 1; }

template <class T>
int f(T x, T y) { return 2; }

int f(int x, int y) { return 3; }
```

```
f(0, 0.0);    // 1 (точное соответствие)
f(0.0, 0.0);  // 2 (меньше подстановок)
f(0, 0);      // 3 (приоритет у нешаблонной функции)

// А если очень хочется шаблон?
f<>(0, 0);    // 2
```

# Специализация шаблона

# Специализация шаблона

Представим такую ситуацию:

```
template <class T>
T abs(T x) { return x > 0 ? x : -x; }

struct Complex {
    double re;
    double im;
};

Complex c{3, 4};  // 3 + 4i
abs(c);  // CE
```

```
test.cpp: In instantiation of 'T abs(T) [with T = Complex]':
test.cpp:15:10:   required from here
test.cpp:10:14: error: no match for 'operator>' (operand types are 'Complex' and 'int')
     return x > 0 ? x : -x;
            ~~^~~
test.cpp:10:24: error: no match for 'operator-' (operand type is 'Complex')
     return x > 0 ? x : -x;
```

# Специализация шаблона

Можно объявить обычную функцию для комплексных чисел, тогда по точному совпадению будет выбрана она:

```
Complex abs(Complex x) { return {sqrt(x.re * x.re + x.im * x.im), 0}; }

Complex c{3, 4};
abs(c);  // Ok: {5, 0}
```

Но остается возможность некорректного вызова шаблона:

```
Complex c{3, 4};

abs<Complex>(c);  // Все еще CE
abs<>(c);         // И здесь
```

# Специализация шаблона

Решение - специализация шаблона.

Специализация шаблона позволяет определить реализацию для конкретного набора параметров.

```
// общий шаблон
template <class T>
T abs(T x) { return x > 0 ? x : -x; }

// специализация
template <>
Complex abs(Complex x) { return {sqrt(x.re * x.re + x.im * x.im), 0}; }



Complex c{3, 4};
abs(c);  // Ok: [T=Complex] {5, 0}
```

```
template <class T> T GetZero() { return 0; }
template <> Complex GetZero() { return {0, 0}; }
```

# Non-type template parameters

# Параметры шаблона не являющиеся типами

# Non-type template parameters

В качестве шаблонных параметров помимо типов могут выступать еще и:

- Целые числа
- Указатели
- Ссылки
- `std::nullptr_t`
- Числа с плавающей точкой (C++20)
- Некоторые классы специального вида (C++20)

# Non-type template parameters

```
template <int N, int M>
int Sum() { return N + M; }

Sum<3, 8>();  // Ok: 11

int x = 1;
Sum<1, x>();  // CE (N and M must be compile-time constants!)
```

```
template <class T, size_t N>
size_t ArraySize(const T (&array)[N]) { return N; }

int arr[11];
ArraySize(arr);  // Ok: 11 (N is deduced from the type of arr)
```

```
template <void (*FPtr)(int)>
void Call(int x) { FPtr(x); }

Call<f>(10);
```

# Ключевое слово `auto`

# Ключевое слово `auto`

- До C++11 `auto` использовалось для обозначения переменной в автоматической области памяти (локальная переменная).
- Возможность оказалась настолько неактуальной, что в стандарте C++11 кардинально поменяли смысл слова `auto` (довольно исключительная ситуация).
- Прежний смысл отдали ключевому слову `register`.
- Однако в C++17 отказались и от него и теперь `register` считается устаревшим (слово зарезервировано для дальнейшего использования).

# Ключевое слово `auto` в C++11

В наше время `auto` используется для автоматического вывода типа переменной при инициализации

```
int main() {
    auto x = 0;    // int
    auto y = 0.0;  // double
    auto px = &x;  // int*

    auto z;  // CE: тип переменной определить невозможно
}
```

Очень удобно, когда речь идет о длинных именах типов

```
for (std::unordered_set<long long>::const_reverse_iterator it = s.crbegin();
     it != s.crend(); ++it)

// vs

for (auto it = s.crbegin(); it != s.crend(); ++it)
```

# Ключевое слово `auto` в C++11

Правила вывода типа `auto` совпадают с правилами вывода для шаблонных параметров (за некоторым исключением)

```
int x = 0; const int cx = 1; int& rx = x; int arr[10];

auto y = x;
auto cy = cx;
auto ry = rx;
auto arr_y = arr;

auto& z = x;
auto& cz = cx;
auto& rz = rx;
auto& arr_z = arr;
```

Исключение

```
auto x = {1, 2, 3};
```

# Ключевое слово `auto` в C++11

Правила вывода типа `auto` совпадают с правилами вывода для шаблонных параметров (за некоторым исключением)

```
int x = 0; const int cx = 1; int& rx = x; int arr[10];

auto y = x;         // int
auto cy = cx;       // int
auto ry = rx;       // int
auto arr_y = arr;   // int*

auto& z = x;         // int&
auto& cz = cx;       // const int&
auto& rz = rx;       // int&
auto& arr_z = arr;   // int(&)[10]
```

Исключение

```
auto x = {1, 2, 3};  // std::initializer_list<int>
```

# Резюме

- Шаблоны развивают идеи перегрузки, позволяя писать общую реализацию для множества типов.
- Шаблоны можно перегружать, а также специализировать для конкретных типов.
- Шаблоны инстанцируются по мере необходимости, по одному экземпляру на каждый набор типов.
- Использование объявлений `auto` позволяет упростить код, уберечь от ошибок и в некоторых случаях ускорить программы.