

Лекция 07 — Аргументы функции и перегрузка

Содержание

1. Аргументы функции
 1. Способы передачи аргументов
 2. Аргументы по умолчанию
 3. Неопределенное число аргументов
2. Перегрузка функций
 1. Общие представления
 2. Разрешение перегрузок
3. Перегрузка операторов
 1. Общие представления
 2. Особенности перегрузки некоторых операторов
 1. `operator ()` и функциональные классы
 2. `operator <<` и дружественные функции
 3. `operator ++` и постфиксный инкремент
4. Общие правила объявления перегрузок

”То поведение, которое вы задаёте перегрузкой операторов, должно базироваться на общем представлении человечества о прекрасном.”
— А. П. Хвастунов

Аргументы функции

Передача аргументов в функцию

Теперь, когда мы знакомы с концепцией ссылок, имеет смысл обобщить знания о способах передачи аргументов в функцию. Так, среди них выделяются следующие:

1. **Передача по значению** (pass by value)

```
void print(int x) {  
    std::cout << x << '\n';  
}
```

При передачи аргумента по значению функция получает его полную копию. Отсюда немедленно следует, что любое изменение такого аргумента в теле функции никак не затрагивает его прообраз.

Этот метод чаще всего используется при работе с небольшими или встроенными типами при условии, что нам не требуется изменять реальные значения переменных, или в случае, когда нам явно необходима копия объекта.

2. Передача по константной ссылке (pass by const reference)

```
void print(const int& x) {  
    std::cout << x << '\n';  
}
```

Практически, на этот метод можно смотреть как на "дешёвый" вариант передачи по значению. По определению константной ссылки, функция будет обращаться непосредственно к оригиналу передаваемого объекта, избегая копирования данных, однако он будет доступен лишь для чтения. Это также позволяет передавать в функцию временные значения, в частности, литералы:

```
print(5); // никакой ошибки!
```

На практике этот способ используется наиболее часто, особенно для сложных типов.

3. Передача по ссылке (pass by reference)

```
void print(int& x) {  
    std::cout << x << '\n';  
}
```

Передача по ссылке имеет место быть в ситуациях, когда нам требуется непосредственно изменять значение передаваемой переменной. Формально, мы можем и не совершать изменений с исходным объектом, однако из соображений безопасности и семантической ясности в таких ситуациях настоятельно рекомендуется использовать константные ссылки. Помимо этого, стоит помнить, что этот способ передачи аргумента не работает с литералами:

```
print(5); // ошибка компиляции!
```

Наконец, именно этот способ позволяет наиболее оптимально решить задачу, с которой началось обсуждение ссылок:

```
void swap(int& a, int& b) {  
    int c = a;  
    a = b;  
    b = c;  
}
```

4. Передача по указателю (pass by pointer)

```
void print(int* x) {  
    std::cout << *x << '\n';  
}
```

Во многом этот способ передачи, как и сами указатели, является архаизмом, унаследованным от языка С.

В ситуациях, когда наша задача ограничивается изменением исходного значения переменной, предпочтительнее пользоваться более удобным и современным методом передачи аргумента по ссылке. Передача же по указателю используется лишь в отдельных случаях, когда этого требует логика функции.

Important

Важно понимать, что способ передачи зависит от преследуемой цели и почти всегда влияет на семантику реализуемой функции.

Аргументы по умолчанию

В некоторых ситуациях может оказаться полезным иметь возможность задать стандартные значения, которые будут принимать аргументы функции, если явно не определить их иначе, или вовсе делать некоторые из аргументов необязательными. Это представляется возможным при использовании *аргументов по умолчанию*:

```
void print(char* str = "Default argument\n") {  
    std::cout << str;  
}  
  
int main() {  
    print("Hello, World!\n"); // Вывод: Hello, World!  
    print(); // Вывод: Default argument  
}
```

Все аргументы по умолчанию должны располагаться строго после аргументов, не имеющих значения по умолчанию:

```
void foo1(int a = 238, int b, int c); // CE

void foo2(int a, int b = 238, int c = 29); // OK!
```

При повторном объявлении функции список аргументов по умолчанию может дополняться. При этом важно, чтобы в результате не нарушилось условие, обозначенное выше, а также чтобы ни один из заданных ранее аргументов по умолчанию не был переопределён:

```
void foo(int a, int b, int c = 238);

void foo(int a, int b, int c); // OK!

void foo(int a, int b, int c = 238); // CE: переопределили аргумент

void foo(int a = 29, int b, int c); // CE: нарушили порядок

void foo(int a, int b = 29, int c); // OK!
```

Аргументами по умолчанию могут также выступать объявленные ранее объекты. При этом реальные значения вычисляются только в момент вызова функции (в том числе если они заданы константно), причём каждый раз:

```
int a = 237;

int inc(int x) {
    return x + 1;
}

void foo(int arg = inc(a)) {
    std::cout << arg << '\n';
}

int main() {
    foo(); // Вывод: 238
    a = 28;
    foo(); // Вывод: 29
}
```

Определение аргументов по умолчанию доступно только в списках параметров функций или лямбда-выражений. В частности, не представляется возможным определить аргументы по умолчанию при объявлении указателей или ссылок на функции.

Неопределенное число аргументов

Достаточно часто нам приходится сталкиваться с ситуациями, в которых удобно иметь функцию, способную принимать произвольное количество аргументов. Наиболее известным примером такой функции является `printf`:

```
printf("%d %d %d", 1, 2, 3); // Число аргументов — любое!
```

Для определения функции с произвольным числом аргументов или, иначе, вариативной функции необходимо указать в качестве последнего её параметра многоточие (или *эллипсис*): ^[1]

```
void CustomPrintf(int count, ...);
```

Последний именованный аргумент и все неименованные аргументы, переданные в вариативную функцию, претерпевают цепь преобразований, известную как *default argument promotions*. Так,

- `std::nullptr_t` преобразуется в `void*`;
- `float` преобразуется в `double`;
- `bool`, `char`, `short` и глобальные `enum` преобразуются в `int` или в один из старших целочисленных типов.

Последний именованный аргумент перед эллипсисом не может иметь ссылочный тип или тип, несовместимый^[2] с полученным в результате описанных выше конверсий. Несоблюдение этого принципа приводит к неопределённому поведению.

Механизм работы с неименованными аргументами реализуется макросами из библиотеки `<cstdarg>`, представленной ещё в языке C. В числе них:

- `va_list` — тип объекта, содержащего всю информацию, необходимую для оперирования неименованными аргументами.
- `va_start(va_list ap, parm_n)` — функция, предоставляющая доступ к неименованным аргументам.

Здесь `ap` — объект типа `va_list`, к которому они будут привязаны; `parm_n` — последний именованный параметр, после которого следует эллипсис.

- `va_end(va_list ap)` — функция, прерывающая доступ к неименованным аргументам. Здесь `ap` — объект типа `va_list`, к которому они привязаны.
- `va_copy(va_list dest, va_list src)` — функция, копирующая данные из `src` в `dest`.
- `va_arg(va_list ap, T)` — функция, возвращающая следующий неименованный аргумент. Здесь `ap` — объект типа `va_list`, к которому привязаны аргументы, `T` — тип следующего параметра.

Важно, что в ситуации, когда тип `T` повышаем, поведение программы не определено. Причиной этого являются описанные выше преобразования, применяемые к неименованным аргументам.

Примечательно, что внутреннее устройство семейства `va_*` никак не описывается Стандартом, поскольку написание переносимых (в смысле разных машинных архитектур) функций от переменного числа аргументов не представляется возможным вообще, и ответственность за решение этой проблемы целиком лежит на компиляторе.

Так, например, будет выглядеть реализация простейшей функции, реализующей вывод произвольного числа целочисленных переменных с использованием эллипсиса:

```
void CustomPrintf(int count, ...) {
    std::va_list args;
    va_start(args, count); // ВАЖНО! Второе значение — не количество
                           // неименованных аргументов, а последний
именованный                // аргумент
    for (int i = 0; i < count; ++i) {
        std::cout << va_arg(args, int) << ' ';
    }
    va_end(args);
}

// ...

CustomPrintf(4, 52, 238, 30, 11) // Output: 52 238 30 11
```

При этом вызов `va_end` критически важен, поскольку в `va_start` могут быть такие операции со стеком (в частности, манипуляции с регистрами), из-за которых корректный возврат из функции становится невозможным. `va_end` устраняет все нежелательные изменения.

До стандарта C23 указание именованного параметра перед эллипсисом было обязательным, однако позже это требование упразднили, и появилась возможность объявления функций без именованных параметров вообще:

```
void CustomPrintf(...);
```

Полноценного доступа к аргументам такой функции мы, конечно, получить не сможем, поскольку их число нам неизвестно. Впрочем, в некоторых специфических случаях их может быть полезно применять при работе с перегрузками (см. следующий модуль), поскольку они имеют последний приоритет при их разрешении, позволяя самостоятельно обрабатывать ситуации, при которых нужной перегрузки не нашлось. Это активно используется при разрешении шаблонов функций. [\[3\]](#)

Более того, в современном C++ это практически единственный случай, когда разумно использовать вариативные функции в стиле C. В остальных ситуациях рекомендуется отказаться от этой идеи в пользу вариативных шаблонов, которые позволяют создавать типобезопасные функции от произвольного числа переменных.

Перегрузка функций

Общие представления

Достаточно пытливому и внимательному читателю известно, что имена функций в C++ не обязаны быть уникальными. Дело в том, что сигнатура функции определяется не только её именем, но и списком её аргументов — это значит, что при определении одноимённых функций с различными наборами параметров они будут существовать независимо друг от друга. Такое поведение называется **перегрузкой функций**:

```
int max(const int& x, const int& y) {  
    return (x > y ? x : y);  
}  
  
float max(const float& x, const float& y) {  
    return (x > y ? x : y);  
}  
  
char* max(char* x, char* y) {  
    return (std::strcmp(x, y) > 0 ? x : y);  
}
```

Важно, что возвращаемое значение не является частью сигнатуры функции. Это означает, что изменение, затрагивающее только его, не провоцирует перегрузку — более того, оно вызывает ошибку компиляции:

```
int max(int a, int b) {
    return (a > b ? a : b);
}

float max(int a, int b) { // CE!
    return (a > b ? a : b);
}
```

Следует учитывать, что для параметров, передаваемых по значению, удаляется квалификатор, поэтому соответствующие функции не будут считаться перегруженными:

```
void Foo(int x) {
    std::cout << "Foo(int x)";
}

void Foo(const int x) {
    std::cout << "Foo(const int x)";
}

// error: redefinition of 'Foo'
```

Разрешение перегрузок

Несмотря на то, что на концептуальном уровне явление перегрузки не является чем-то сложным для осознания, существует бесчисленное множество тонкостей, с которыми мы вынуждены столкнуться при попытке их разрешить, особенно в случае, когда программист пренебрегает здравой логикой при их объявлении.

Когда компилятор встречает в коде имя функции, он начинает поиск соответствующей декларации и формирует множество функций-кандидатов. В первую очередь просматриваются все одноимённые объявления, находящиеся в текущем поле видимости. Если в результате нашлось хотя бы одно такое объявление, то поиск приостанавливается. Если поиск не увенчался успехом, мы переходим к объемлющему полю видимости и повторяем процедуру для него.

Процесс продолжается вплоть до момента достижения глобального поля видимости. При этом если он остановился, то, конечно, совершенно не гарантируется, что сигнатура хоть какой-то из найденных деклараций будет соответствовать той, которую мы ожидаем (или что найденная декларация вообще будет представлять собой функцию):

```
char* max(char* x, char* y) {
    return (std::strcmp(x, y) > 0 ? x : y);
}
```



```

}

int main() {
    int max(const int& x, const int& y);
    std::cout << max("abacaba", "helloworld"); // Ошибка компиляции!
}

// НО!

char* max(char* x, char* y) {
    return (std::strcmp(x, y) > 0 ? x : y);
}

int main() {
    // int max(const int& x, const int& y);
    std::cout << max("abacaba", "helloworld"); // ОК!
}

```

Warning

С практической точки зрения это означает, что мы **не можем** смешивать перегрузки из разных полей видимости!

Если в результате поиска не было найдено ни одной подходящей декларации, компилятор начинает просматривать пространства имён и классы, которые ассоциированы с аргументами вызова функции — этот процесс называется *argument-dependent lookup* (или *ADL*).

Следующим шагом среди всех функций-кандидатов выделяются подходящие (*viable functions*). Чтобы считаться таковой, функция должна удовлетворять следующим условиям:

1. Если число аргументов в выражении вызова равно N , то верно одно из следующего:
 - в функции-кандидате ровно N параметров;
 - в функции-кандидате больше, чем N параметров, причём все параметры, начиная с $N+1$, имеют значения по умолчанию;
 - в функции-кандидате не более, чем $N+1$ параметр, причём последним из них является эллипсис.
2. Для каждого аргумента существует хотя бы одна последовательность неявных преобразований, которая приводит его тип к типу соответствующего параметра.

При этом пользовательские касты допускаются, если при этом не возникает более одного способа выполнить конверсию.

Теперь, когда у компилятора есть множество подходящих функций, перед ним встаёт задача выбора лучшей из них.

Функция `F1` считается лучшей функцией, чем `F2`, если: ^[4]

1. каждое из неявных преобразований, необходимых для `F1`, не хуже, чем каждое из таких преобразований, необходимое для `F2`;
2. существует хотя бы один аргумент, неявное преобразование которого к соответствующему параметру `F1` лучше, чем его преобразование к соответствующему параметру `F2`.

Сравнение последовательностей преобразований происходит по следующему принципу: ^[5]

1. стандартная последовательность преобразований всегда лучше, чем пользовательская;
2. пользовательская последовательность преобразований всегда лучше, чем эллипсис-преобразование;
3. стандартная последовательность преобразований `S1` лучше, чем стандартная последовательность `S2`, если верно одно из следующего:
 1. `S1` — собственная подпоследовательность `S2`;
 2. `S1` не выполняет никаких преобразований, в то время как `S2` выполняет;
 3. `S1` приводит к `promotion`-преобразованию, в то время как `S2` приводит к `conversion`-преобразованию.
4. пользовательская последовательность конверсий `U1` лучше, чем пользовательская последовательность `U2`, если они вызывают один конструктор или инициализируют один класс, и при этом соответствующая стандартная цепочка преобразований в `U1` лучше стандартной цепочки в `U2`.

Если существует ровно одна функция, которая лучше всех остальных, процесс поиска завершается и компилятор обращается к ней. Если же такой функции не существует, мы сталкиваемся с соответствующей ошибкой компиляции:

- `no matching function for call to <имя функции>`

Эта ошибка возникает, если множество подходящих функций пусто:

```
void foo(int a, int b)
```

```
int main() {  
    foo(1);  
}
```

- call to <имя функции> is ambiguous

Эта ошибка возникает, когда множество подходящих функций непусто, но выбрать из них лучшую невозможно:

```
void foo(double x);  
  
void foo(float x);  
  
int main() {  
    foo(1);  
}
```

Перегрузка операторов

Общие представления

При работе с определенной категорией пользовательских классов возникает вполне естественное желание иметь возможность складывать, умножать, сравнивать или присваивать объекты этих классов так же, как мы можем делать это с объектами стандартных типов вроде `int` или `double`.

Хотя формально перечисленные операторы для стандартных типов являются первичными элементами синтаксиса, их можно условно воспринимать как функции, под вызов которых зарезервировано некоторое ключевое слово:

```
int x = a + b; // ==> int x = sum(a, b)
```

и, поскольку они, хотя не являясь функциями непосредственно, несут функциональную семантику, C++ предоставляет возможность перегружать их. Перегрузка поддерживается следующими операторами:

Арифметические операторы

- +
- -
- *

- /
- %

Битовые операторы

- ^
- &
- |
- ~
- <<
- >>

Логические операторы

- &&
- ||
- !

Операторы сравнения

- ==
- !=
- <
- >
- <=
- >=
- <=>

Операторы присваивания

- =
- +=
- -=
- *=
- /=
- &=
- ^=
- |=
- <<=
- >>=

Особые операторы

- `++`
- `--`
- `,`
- `->`
- `->*`
- `()`
- `[]`
- `new`
- `new[]`
- `delete`
- `delete[]`

Когда компилятор встречается имя оператора в выражении (или, иначе, *инфиксную форму* записи оператора), хотя бы один из операндов в котором представляет собой объект пользовательского типа (в т.ч. типа `enum`), он начинает процесс разрешения перегрузки с целью поиска функции с соответствующей сигнатурой:

Выражение	Оператор как член класса	Оператор как глобальная функция	Пример
<code>@a</code>	<code>a.operator@()</code>	<code>operator@(a)</code>	<code>!std::cin</code> ВЫЗЫВАЕТ <code>std::cin.operator!()</code>
<code>a@b</code>	<code>a.operator@(b)</code>	<code>operator@(a, b)</code>	<code>std::cout << 238</code> ВЫЗЫВАЕТ <code>std::cout.operator<<(238)</code>
<code>a=b</code>	<code>a.operator=(b)</code>	—	<code>std::string s = "M3104"</code> ВЫЗЫВАЕТ <code>s.operator=("M3104")</code>
<code>a(b...)</code>	<code>a.operator() (b...)</code>	—	<code>std::mt19937 rng()</code> ВЫЗЫВАЕТ <code>rng.operator()()</code>
<code>a[b...]</code>	<code>a.operator[] (b...)</code>	—	<code>std::vector arr[30]</code> ВЫЗЫВАЕТ <code>arr.operator[](30)</code>
<code>a-></code>	<code>a.operator->()</code>	—	<code>std::unique_ptr<S> pt->bar()</code> ВЫЗЫВАЕТ <code>pt.operator->()->bar()</code>

Выражение	Оператор как член класса	Оператор как глобальная функция	Пример
<code>a@</code>	<code>a.operator@(0)</code>	<code>operator@(a, 0)</code>	<code>std::iterator i++</code> вызывает <code>i.operator++(0)</code>

Следует заметить, что упомянутый ранее ADL наиболее часто используется именно при работе с перегруженными операторами.

Как видно из таблицы, в общем случае оператор можно перегрузить двумя способами: как функцию-член и как свободную функцию, однако стоит помнить, что операторы `=`, `()`, `[]` и `->` не допускают глобального определения.

Глобальная перегрузка операторов, если она возможна, более предпочтительна, поскольку предоставляет ряд возможностей, которых лишены операторы-члены класса. В частности, она позволяет менять порядок операндов, расширять интерфейс класса без объявления новых членов, а также автоматически перегружает соответствующий оператор для любого класса, допускающего неявное приведение к исходному.

Особенности перегрузки некоторых операторов

`operator ()` и функциональные классы

Оператор `()` можно определить только как член соответствующего класса. При этом класс становится **функциональным**, а его экземпляр будет называться *функциональным объектом* или *функтором*. Как ясно из названия, класс с перегруженным оператором `()` ожидаемо несёт функциональную семантику, а объект такого класса может использоваться как функция:

```
class Multiplier {
private:
    int times_ = 1;

public:
    Multiplier(int times)
        : times_(times)
    {}

    void operator()(int& x) const {

        x *= times_;

    }
}
```

```
};

int main() {

    Multiplier x2(2);
    int x = 5;
    std::cout << x << std::endl; // 5
    x2(x);
    std::cout << x << std::endl; // 10

}
```

Ключевое отличие функционального объекта от функции в том, что он, всё ещё будучи экземпляром некоторого класса, обладает *состоянием*, которое определяет его поведение. Семантически использование функциональных объектов вместо функций позволяет нам «упростить» бинарную операцию до унарной, что иногда бывает полезно.

Функциональные классы играют крайне важную роль в языке C++ — именно с помощью них реализуется парадигма функционального программирования.

Они активно используются в стандартной библиотеке: ими представлены, например, компараторы, предикаты или хэш-функции. Лямбда-выражения также являют собой не что иное, как анонимный функциональный класс.

operator << и дружественные функции

Крайне часто при работе с пользовательскими классами возникает необходимость выводить какие-то их данные в поток или считывать их из него — в частности, это может быть особенно важно при отладке.

Проблема, с которой мы немедленно сталкиваемся — инкапсуляция, которая не позволяет нам получить доступ к тем полям, к которым мы, собственно, сами же этот доступ и запретили. Это лишает нас возможности просмотреть полное состояние объекта и делает отладку невозможной.

Решений существует несколько. В их числе как очевидно плохие --- например, сделать все поля публичными, — так и кажущиеся сперва более разумными, вроде использования геттеров или определения специального метода `Print()`, однако и они неизбежно экстерминируют один из главных столпов объектно-ориентированного программирования. К сожалению читателя, способа решить поставленную задачу без разрушения принципов ООП в языке C++ просто невозможно, однако существует способ разрушить их чуть более элегантно.

Чтобы предоставить глобальной функции доступ к приватным полям класса, необходимо указать её как поле этого класса с использованием ключевого слова `friend`:

```

class University {
public:
    University(char* name)
        : name_(name)
    {}

    friend std::ostream& operator << (std::ostream& out, University
university);

private:
    char* name_;
};

std::ostream& operator << (std::ostream& out, University university) {
    std::cout << university.name_;
    return out;
}

int main() {
    University uni("ITMO");
    std::cout << uni;
}

```

Важно понимать, что `friend` — это, по сути, никакое не решение проблемы, а всего лишь красивый костыль, предоставляемый самим языком. В подавляющем большинстве случаев (а в идеале — всегда) `friend` используется только для реализации оператора вывода в поток. Здесь мы идём на послабления в принципиальности вокруг идеи инкапсуляции ради решения конкретной проблемы, однако в хорошо написанной программе, как водится, друзей быть не должно.

operator ++ и постфиксный инкремент

Внимательному читателю известно, что операторы префиксного и постфиксного инкремента (аналогично для декремента) обладают различной семантикой. В то время, как префиксный оператор возвращает непосредственно сам объект, к которому была применена операция, постфиксный возвращает копию его исходного состояния.

Тем не менее, операторы префиксного и постфиксного инкремента имеют одно название — `operator ++`. Для того, чтобы их различать, не было придумано ничего лучше, чем объявлять префиксный инкремент как `operator++()`, а постфиксный — как `operator++(int)`. При этом, конечно, в инфиксной форме постфиксный инкремент не принимает никакого аргумента, хотя туда действительно можно передать целое число, если записать его в функциональной форме.

Общие правила объявления перегрузок

Надеемся, что механизмы разрешения перегрузок, описанные в разделе (2), сумели в достаточной степени отбить у читателя желание сколь угодно усложнять работу с ними. С целью упростить себе жизнь следует ввести несколько фундаментальных принципов, которых следует придерживаться при работе с перегрузками:

Перегрузка не должна противоречить здравой логике

Многие из ситуаций, описанных ранее и кажущихся поначалу крайне проблемными, в естественной среде вообще не возникнут, если только не провоцировать их намеренно. Достаточно отдавать себе отчёт о том, что и зачем вообще происходит в коде, чтобы избежать доброй половины всего того, что могут вызвать необдуманные перегрузки.

Так, например, не стоит пытаться перегрузить оператор `,` или объявить собственный оператор. Второе, конечно, помешает сделать компилятор, а вот от перегрузки запятой спасёт лишь здравоеумие.

Не стоит использовать перегрузку только потому, что компилятор это позволяет

Помимо того, что злоупотребление механикой перегрузки сильно повышает вероятность ошибки, оно также банально снижает читаемость кода. Как сказано в Google Code Style, «используйте перегрузки только если читатель, посмотрев на вызов функции, может сформировать достаточно ясное понимание происходящего без необходимости узнавать, какая конкретно перегрузка была вызвана».

Перегрузки одной функции должны находиться в одном поле видимости

Этот принцип — прямое следствие из описанного ранее алгоритма просмотра полей видимости при разрешении перегрузок. Для неопытных программистов и особенно для сторонних читателей, недостаточно глубоко знакомых с языком, такое поведение компилятора будет контринтуитивным, поэтому рассчитывать на него нельзя.

Поведение перегруженного оператора должно согласоваться со стандартным

Чаще всего мы хотим, чтобы запись `a + b` обозначала сложение, а `a | b` — побитовое «или», чем бы `a` и `b` непосредственно не являлись. В рамках разумного допустимы отклонения от этого принципа — например, в классе стандартной библиотеки `std::filesystem::path` оператор `/` несёт семантику конкатенации, что кажется вполне естественным при работе с путями в файловой системе.

В целом достаточно понимать, что перегрузка — это инструмент в первую очередь упрощения и повышения читаемости кода, а (как ни странно) не наоборот. Если в какой-то момент код начинает страдать от их избытка, стоит серьёзно задуматься о том, чтобы что-то в своей жизни поменять.

Источники

1. А. П. Хвастунов — Лекции по основам программирования на C++, 1 семестр, 2025
 2. cppreference.com — Default arguments
(https://en.cppreference.com/w/cpp/language/default_arguments.html)
 3. cppreference.com — Variadic arguments
(https://en.cppreference.com/w/cpp/language/variadic_arguments.html)
 4. Fyret — «C++: сеанс спонтанной археологии и почему не стоит использовать вариативные функции в стиле C» (<https://habr.com/ru/articles/430064/>)
 5. cppreference.com — Overload resolution
(https://en.cppreference.com/w/cpp/language/overload_resolution.html)
 6. cppreference.com — Operator overloading
(<https://en.cppreference.com/w/cpp/language/operators.html>)
 7. dm_frox — «Перегрузка в C++. Часть II. Перегрузка операторов»
(<https://habr.com/ru/articles/489666/>)
-

1. До стандарта C++26 запятая перед многоточием может быть опущена. ↩
2. Практически это означает, что последний именованный аргумент не должен иметь повышаемый тип. Для более формального определения см. модуль Compatible types — https://en.cppreference.com/w/c/language/compatible_type.html#Compatible_types ↩
3. см. принцип SFINAE — <https://en.cppreference.com/w/cpp/language/sfinae.html> ↩
4. На самом деле шагов проверки, как можно догадаться, далеко не 2 (согласно cppreference.com, их 12), однако большинство из них опущено из соображений доступности и внятности изложения. ↩
5. Приведённый алгоритм по аналогичной причине также является упрощённым. ↩