

Ravesli [Ravesli](#)


- [Уроки по C++](#)
- [OpenGL](#)
- [SFML](#)
- [Qt5](#)
- [RegEx](#)
- [Ассемблер](#)
- [Купить .PDF](#)

Урок №102. Перегрузка функций

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 24 Ноя 2020 |

 48873

[↑](#)  7

На этом уроке мы рассмотрим перегрузку функций в языке C++, что это такое и как её эффективно использовать.

Оглавление:

1. [Перегрузка функций](#)
2. [Типы возврата в перегрузке функций](#)
3. [Псевдонимы типов в перегрузке функций](#)
4. [Вызовы функций](#)
5. [Несколько совпадений](#)
6. [Заключение](#)

Перегрузка функций

Перегрузка функций — это возможность определять несколько функций с одним и тем же именем, но с разными параметрами. Например:

```
1 int subtract(int a, int b)
2 {
3     return a - b;
4 }
```

Здесь мы выполняем операцию вычитания с целыми числами. Однако, что, если нам нужно использовать числа типа с плавающей запятой? Эта функция совсем не подходит, так как любые

параметры типа `double` будут конвертироваться в тип `int`, в результате чего будет теряться дробная часть значений.

Одним из способов решения этой проблемы является определение двух функций с разными именами и параметрами:

```
1 int subtractInteger(int a, int b)
2 {
3     return a - b;
4 }
5
6 double subtractDouble(double a, double b)
7 {
8     return a - b;
9 }
```

Но есть и лучшее решение — перегрузка функции. Мы можем просто объявить еще одну функцию `subtract()`, которая принимает параметры типа `double`:

```
1 double subtract(double a, double b)
2 {
3     return a - b;
4 }
```

Теперь у нас есть две версии функции `subtract()`:

```
1 int subtract(int a, int b); // целочисленная версия
2 double subtract(double a, double b); // версия типа с плавающей запятой
```

Может показаться, что произойдет конфликт имен, но это не так. Компилятор может определить сам, какую версию `subtract()` следует вызывать на основе аргументов, используемых в вызове функции. Если параметрами будут переменные типа `int`, то C++ понимает, что мы хотим вызвать `subtract(int, int)`. Если же мы предоставим два значения типа с плавающей запятой, то C++ поймет, что мы хотим вызвать `subtract(double, double)`. Фактически, мы можем определить столько перегруженных функций `subtract()`, сколько хотим, до тех пор, пока каждая из них будет иметь свои (уникальные) параметры.

Следовательно, можно определить функцию `subtract()` и с большим количеством параметров:

```
1 int subtract(int a, int b, int c)
2 {
3     return a - b - c;
4 }
```

Хотя здесь `subtract()` имеет 3 параметра вместо 2-х, это не является ошибкой, поскольку эти параметры отличаются от параметров других версий `subtract()`.

Типы возврата в перегрузке функций

Обратите внимание, **тип возврата функции** НЕ учитывается при перегрузке функции. Предположим, что вы хотите написать функцию, которая возвращает **рандомное число**, но вам нужна одна версия, которая возвращает значение типа `int`, и вторая — которая возвращает значение типа `double`. У вас может возникнуть соблазн сделать следующее:

```
1 int getRandomValue();
2 double getRandomValue();
```

Компилятор выдаст ошибку. Эти две функции имеют одинаковые параметры (точнее, они отсутствуют), и, следовательно, второй вызов функции `getRandomValue()` будет рассматриваться как ошибочное переопределение первого вызова. Имена функций нужно будет изменить.

Псевдонимы типов в перегрузке функций

Поскольку объявление **typedef** (псевдонима типа) не создает новый тип данных, то следующие два объявления функции `print()` считаются идентичными:

```
1 typedef char *string;
2 void print(string value);
3 void print(char *value);
```

Вызовы функций

Выполнение вызова перегруженной функции приводит к одному из 3-х возможных результатов:

- ➔ **Совпадение найдено.** Вызов разрешен для соответствующей перегруженной функции.
- ➔ **Совпадение не найдено.** Аргументы не соответствуют любой из перегруженных функций.
- ➔ **Найдены несколько совпадений.** Аргументы соответствуют более чем одной перегруженной функции.

При компиляции перегруженной функции, C++ выполняет следующие шаги для определения того, какую версию функции следует вызывать:

Шаг №1: C++ пытается найти точное совпадение. Это тот случай, когда фактический аргумент точно соответствует типу параметра одной из перегруженных функций. Например:

```
1 void print(char *value);
2 void print(int value);
3
4 print(0); // точное совпадение с print(int)
```

Хотя `0` может технически соответствовать и `print(char *)` (как **нулевой указатель**), но он точно соответствует `print(int)`. Таким образом, `print(int)` является лучшим (точным) совпадением.

Шаг №2: Если точного совпадения не найдено, то C++ пытается найти совпадение путем дальнейшего неявного преобразования типов. На **уроке №55** мы говорили о том, как определенные

типы данных могут автоматически конвертироваться в другие типы данных. Если вкратце, то:

- `char`, `unsigned char` и `short` конвертируются в `int`;
- `unsigned short` может конвертироваться в `int` или `unsigned int` (в зависимости от размера `int`);
- `float` конвертируется в `double`;
- `enum` конвертируется в `int`.

Например:

```
1 void print(char *value);  
2 void print(int value);  
3  
4 print('b'); // совпадение с print(int) после неявного преобразования
```

В этом случае, поскольку нет `print(char)`, символ `b` конвертируется в тип `int`, который затем уже соответствует `print(int)`.

Шаг №3: Если неявное преобразование невозможно, то C++ пытается найти соответствие посредством стандартного преобразования. В стандартном преобразовании:

- Любой числовой тип будет соответствовать любому другому числовому типу, включая `unsigned` (например, `int` равно `float`).
- `enum` соответствует формальному типу числового типа данных (например, `enum` равно `float`).
- Ноль соответствует типу указателя и числовому типу (например, `0` как `char *` или `0` как `float`).
- Указатель соответствует указателю типа `void`.

Например:

```
1 struct Employee; // определение упустим  
2 void print(float value);  
3 void print(Employee value);  
4  
5 print('b'); // 'b' конвертируется в соответствие версии print(float)
```

В этом случае, поскольку нет `print(char)` (точного совпадения) и нет `print(int)` (совпадения путем неявного преобразования), символ `b` конвертируется в тип `float` и сопоставляется с `print(float)`.

Обратите внимание, все стандартные преобразования считаются равными. Ни одно из них не считается выше остальных по приоритету.

Шаг №4: C++ пытается найти соответствие путем пользовательского преобразования. Хотя мы еще не рассматривали классы, но они могут определять преобразования в другие типы данных, которые могут быть неявно применены к объектам этих классов. Например, мы можем создать класс `W` и в нем определить пользовательское преобразование в тип `int`:

```
1 class W; // с пользовательским преобразованием в тип int
```

```
2 |
3 | void print(float value);
4 | void print(int value);
5 |
6 | W value; // объявляем переменную value типа класса W
7 | print(value); // value конвертируется в int и, следовательно, соответствует print(int)
```

Хотя `value` относится к типу класса `W`, но, поскольку тот имеет пользовательское преобразование в тип `int`, вызов `print(value)` соответствует версии `print(int)`.

То, как делать пользовательские преобразования в классах, мы рассмотрим на соответствующих уроках.

Несколько совпадений

Если каждая из перегруженных функций должна иметь уникальные параметры, то как могут быть возможны несколько совпадений? Поскольку все стандартные и пользовательские преобразования считаются равными, то, если вызов функции соответствует нескольким кандидатам посредством стандартного или пользовательского преобразования, результатом будет **неоднозначное совпадение** (т.е. несколько совпадений). Например:

```
1 | void print(unsigned int value);
2 | void print(float value);
3 |
4 | print('b');
5 | print(0);
6 | print(3.14159);
```

В случае с `print('b')` C++ не может найти точного совпадения. Он пытается преобразовать `b` в тип `int`, но версии `print(int)` тоже нет. Используя стандартное преобразование, C++ может преобразовать `b` как в `unsigned int`, так и во `float`. Поскольку все стандартные преобразования считаются равными, то получается два совпадения.

С `print(0)` всё аналогично. `0` — это `int`, а версии `print(int)` нет. Путем стандартного преобразования мы опять получаем два совпадения.

А вот с `print(3.14159)` всё несколько запутаннее: большинство программистов отнесут его однозначно к `print(float)`. Однако, помните, что по умолчанию все значения-литералы типа с плавающей запятой относятся к типу `double`, если у них нет окончания `f`. `3.14159` — это значение типа `double`, а версии `print(double)` нет. Следовательно, мы получаем ту же ситуацию, что и в предыдущих случаях — неоднозначное совпадение (два варианта).

Неоднозначное совпадение считается ошибкой типа `compile-time`. Следовательно, оно должно быть устранено до того, как ваша программа скомпилируется. Есть два решения этой проблемы:

Решение №1: Просто определить новую перегруженную функцию, которая принимает параметры именно того типа данных, который вы используете в вызове функции. Тогда C++ сможет найти точное совпадение.

Решение №2: Явно преобразовать с помощью [операторов явного преобразования](#) неоднозначный параметр(ы) в соответствии с типом функции, которую вы хотите вызвать. Например, чтобы вызов

`print(0)` соответствовал `print(unsigned int)`, вам нужно сделать следующее:

```
1 | print(static_cast<unsigned int>(0)); // произойдет вызов print(unsigned int)
```

Заключение

Перегрузка функций может значительно снизить сложность программы, в то же время создавая небольшой дополнительный риск. Хотя этот урок несколько долгий и может показаться сложным, но, на самом деле, перегрузка функций обычно работает прозрачно и без каких-либо проблем. Все неоднозначные случаи компилятор будет отмечать, и их можно будет легко исправить.

Правило: Используйте перегрузку функций для упрощения ваших программ.

Оценить статью:

★★★★★ (226 оценок, среднее: 4,90 из 5)



← [Урок №101. Встроенные функции](#)

[Урок №103. Параметры по умолчанию](#) →



Комментариев: 7



1. *tony:*

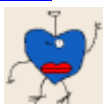
[17 февраля 2020 в 12:10](#)

По поводу примера:

```
1 | typedef char* string;  
2 | void print(string value);  
3 | void print(char* value);
```

А если подключена библиотека `<string>` разве это безопасно создавать псевдоним для уже зарезервированного типа данных?

[Ответить](#)



1. *Oleksii:*

[1 июня 2020 в 16:47](#)

Для вызова string из #include <string> нужно использовать пространство имен

[Ответить](#)



2. *Старый программист:*

[29 ноября 2019 в 15:33](#)

Цитата: При вызове перегруженной функции, C++ выполняет следующие шаги для определения того, какую версию функции следует вызывать:

НЕ при вызове, а при компиляции. Перегруженные функции имеют разные имена на объектном уровне — к имени добавляется суффиксы параметров

[Ответить](#)



1. *Юрий:*

[29 ноября 2019 в 19:07](#)

Спасибо, исправил)

[Ответить](#)



3. *Алексей:*

[27 августа 2019 в 16:50](#)

Неплохо, неплохо.

Правда на живом примере вопроса бы не возникло) Как обычно — внимательность.

Спасибо за курс)

[Ответить](#)



4. *Андрей:*

[23 июня 2018 в 14:31](#)

Спасибо большое. Очень полезный материал и доходчиво разложен. Автору большой респект и уважуха.

[Ответить](#)



1. *Юрий:*

[23 июня 2018 в 14:33](#)

Спасибо и Вам, что читаете 😊

[Ответить](#)

Добавить комментарий

Ваш E-mail не будет опубликован. Обязательные поля помечены *

Имя *

Email *

Комментарий






☐ Сохранить моё Имя и E-mail. Видеть комментарии, отправленные на модерацию

☐ Получать уведомления о новых комментариях по электронной почте. Вы можете [подписаться](#) без комментирования.

[TELEGRAM](#)  [КАНАЛ](#)

[ПАБЛИК](#) 

ТОП СТАТЬИ

-  [Словарь программиста. Сленг, который должен знать каждый кодер](#)
-  [Урок №1. Введение в программирование](#)
-  [70+ бесплатных ресурсов для изучения программирования](#)
-  [Урок №1: Введение в создание игры «SameGame» на C++/MFC](#)
-  [Урок №4. Установка IDE \(Интегрированной Среды Разработки\)](#)

- [Ravesli](#)
- - [О проекте/Контакты](#) -
- - [Пользовательское Соглашение](#) -
- - [Все статьи](#) -
- Copyright © 2015 - 2020