Тема 1.4 Функции. Заголовок и тело функции, классификация параметров. Вызов функций, особенности их использования

При разработке большой и сложной программы можно значительно облегчить свою работу и повысить наглядность создаваемого исходного кода путем разбиения программы на части, называемые функциями.

Пусть, например, требуется написать программу, работающую с текущими бухгалтерскими счетами. Можно предусмотреть функцию, которая выполняет операции главной книги, другую функцию для оформления счетов к оплате, третью – для оформления получаемых счетов, и четвертую – для выполнения балансового счета.

Если все операторы, реализующие данные действия, будут размещены внутри main(), то программа получиться очень длинной и трудной для понимания. При этом по мере возрастания размера и сложности программы увеличивается вероятность возникновения ошибки.

Функция по своей сути — это подпрограмма, которая может манипулировать данными и возвращать некоторое значение. Каждая программа C++ имеет по крайней мере одну функцию `main()`, которая при запуске программы вызывается автоматически. Функция `main()` может вызывать другие функции, те, в свою очередь, могут вызывать следующие и т.д.

Функцией называется именованный набор операторов, выполняющий определенную задачу.

Каждая функция обладает собственным именем, и, когда оно встречается в программе, управление переходит к телу данной функции. Этот процесс называется вызовом функции (или обращением к функции). По возвращении из функции выполнение программы возобновляется со строки, следующей после вызова функции. Такая схема выполнения программы показана на рис. 6.1.

Хорошо разработанные функции должны выполнять конкретную и вполне понятую задачу. Сложные задачи следует "разбивать" на несколько более простых, достаточно легко реализуемых с помощью отдельных функций, которые затем могут вызываться по очереди.

Различают два вида функций: определяемые пользователем (нестандартные) и встроенные. Встроенные функции являются составной частью пакета компилятора и предоставляются фирмой-изготовителем. Нестандартные функции создаются самим программистом.

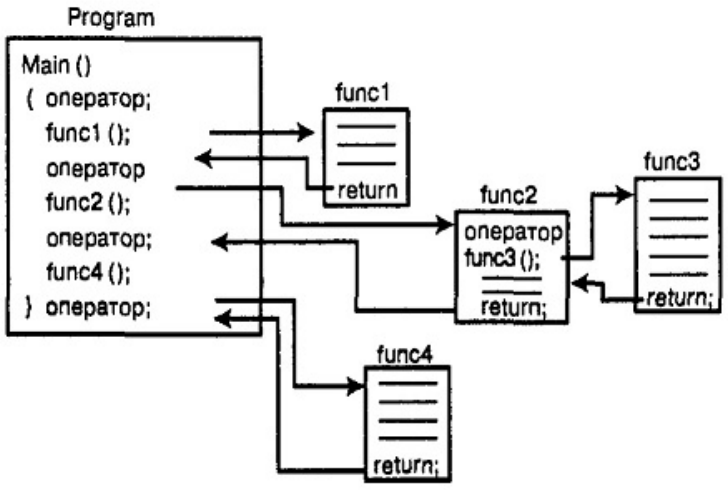

Рисунок 6.1. - Схема выполнения программы

Функции можно сгруппировать в две **категории**:

- функции, которые не имеют возвращаемых значений, называемые функциями типа void. Они имеют следующую общую структуру:

```
void имя_функции (список аргументов)
{
операторы;
```

return; //необязательная конструкция
}

- функции, которые имеют возвращаемое значения. Они имеют следующую общую форму:

имя_типа имя_функции (список аргументов)
{
операторы;
return значение;
}

Список-аргументов — это список разделенных запятыми объявлений тех аргументов, которые получает функция при ее вызове. Если функция не получает никаких значений, список-параметров задается как void. Тип должен быть указан явно для каждого параметра в списке параметров.

Функции с возвращаемыми значениями требуют использования оператора возврата, обеспечивающего возврат полученного значения вызывающей функции. Само значение может быть константой, переменной или выражением более общего типа. К выражению общего типа предъявляется единственное требование, чтобы это выражение приводило к значению, которое имеет тип «имя_типа» или могло быть преобразовано к этому типу.

Язык C++ накладывает ограничения на типы, которые могут быть использованы для возвращаемого значения: массив не может быть возвращаемым значением.

Объявление функций

Использование функций в программе требует, чтобы функция сначала была объявлена, а затем определена. Посредством объявления функции компилятору сообщается ее имя, тип возвращаемого значения и параметры. Благодаря определению функции компилятор узнает, как функция работает. Ни одну функцию нельзя вызвать в программе, если она не была предварительно объявлена. Объявление функции называется *прототипом*.

Существует три способа объявления функции:

1) Запишите прототип функции в файл, а затем используйте выражение с `"include"`, чтобы включить его в свою программу.

2) Запишите прототип функции в файл, в котором эта функция используется.

3) Определите функцию перед тем, как ее вызовет любая другая функция. В этом случае определение функции одновременно и объявляет ее.

Несмотря на то что функцию можно определить непосредственно перед использованием и таким образом избежать необходимости создания прототипа функции, такой стиль программирования не считается хорошим по трем причинам.

Во-первых, требование располагать функции в файле в определенном порядке затрудняет поддержку программы в процессе изменения условий ее использования.

Во-вторых, вполне возможна ситуация, когда функции `A()` необходимо вызвать функцию `B()`, но не исключено также, что при некоторых обстоятельствах и функции `B()` потребуется вызвать функцию `A()`. Однако невозможно определить функцию `A()` до определения функции `B()` и в то же время функцию `B()` до определения функции `A()`, т.е. по крайней мере одна из этих функций обязательно должна быть предварительно объявлена.

В-третьих, прототипы функций — это хорошее и сильное подспорье при отладке. Если согласно прототипу объявлено, что функция принимает определенный набор параметров или что она возвращает значение определенного типа, а затем в программе делается попытка использовать функцию, не соответствующую объявленному прототипу, то компилятор заметит эту ошибку еще на этапе компиляции программы, что позволит избежать неприятных сюрпризов в процессе ее выполнения.

Прототипы функции

Прототипы многих встроенных функций уже записаны в файлы заголовков, добавляемые в программу с помощью "include". Для функций, создаваемых пользователями, программист должен сам позаботиться о включении в программу соответствующих прототипов.

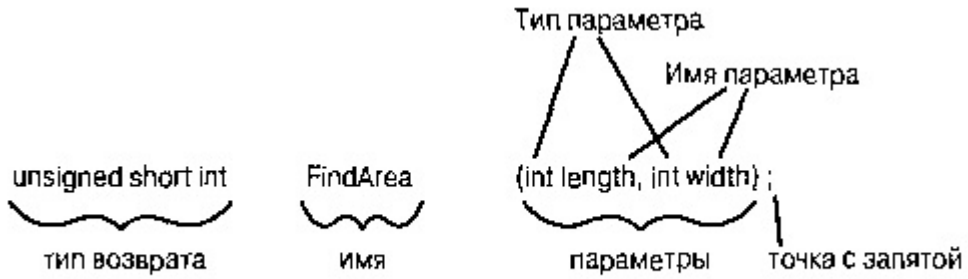

Рисунок 6.2 Составные части прототипа функции

Прототип функции представляет собой выражение, оканчивающееся точкой с запятой, и состоит из типа возвращаемого значения функции и сигнатуры. Под сигнатурой функции подразумевается ее имя и список формальных параметров.

[спецификатор-типа] имя-функции ([список-типов-аргументов]);

Список формальных параметров представляет собой список всех параметров и их типов, разделенных запятыми. Составные части прототипа функции показаны на рис. 6.2.

В прототипе и в определении функции тип возвращаемого значения и сигнатура должны соответствовать. Если такого соответствия нет, компилятор покажет сообщение об ошибке. Однако прототип функции не обязан содержать имена параметров, он может ограничиться только указанием их типов. Например, прототип, приведенный ниже, абсолютно правомочен:

```
long Area(int, int);
```

Этот прототип объявляет функцию с именем `Area()`, которая возвращает значение типа `long` и принимает два целочисленных параметра. И хотя такая запись прототипа вполне допустима, это не самый лучший вариант. Добавление имен параметров делает ваш прототип более ясным. Та же самая функция, но уже с именованными параметрами, выглядит гораздо понятнее:

```
long Area(int length, Int width);
```

Теперь сразу ясно, для чего предназначена функция и ее параметры. Обратите внимание на то, что для каждой функции всегда известен тип возвращаемого значения. Если он явно не объявлен, то по умолчанию принимается тип `int`. Однако ваши программы будут понятнее, если для каждой функции, включая `main()`, будет явно объявлен тип возвращаемого значения. В листинге 6.1 приводится программа, которая содержит прототип функции `Area()`.

Листинг 6.1. Объявление, определение и использование функции

```
    1: // Листинг 6.1. Использование прототипов функций
2: int
3: «include <iostream.h>
4: int Area(int length, int width); //прототип функции
5:
6: int main()
7: {
8: int lengthOfYard;
9: int widthOfYard;
10: int areaOfYard;
11:
12: cout << "\ nHow wide is your yard? ";
```

```
13: cin >> widthOfYard;
14: cout << "\ nHow long is your yard? ";
15: cin >> lengthOfYard;
16:
17: areaOfYard = Area(lengthOfYard,widthOfYard);
18:
19: cout << "\ nYour yard is ";
20: cout << areaOfYard;
21: cout << " square feet\ n\ n";
22: return 0;
23: }
24:
25: int Area(int yardLengtlr, int yardWidth)
26: {
27: return yardlength * yardWidth;
28: }
```

Результат:
How wide is your yard? 100
How long is your yard? 200
Your yard is 20000 square feet

Прототип функции `Area()` объявляется в строке 4. Сравните прототип с определением функции, представленным в строке 25. Обратите внимание, что их имена, типы возвращаемых значений и типы параметров полностью совпадают. Если бы они были различны, то компилятор показал бы сообщение об ошибке. Единственное обязательное различие между ними состоит в том, что прототип функции оканчивается точкой с запятой и не имеет тела.

Обратите также внимание на то, что имена параметров в прототипе — `length` и `width` — не совпадают с именами параметров в определении: `yardLength` и `yardWidth`. Как упоминалось выше, имена в прототипе не используются; они просто служат описательной информацией для программиста. Соответствие имен параметров прототипа именам параметров в определении функции считается хорошим стилем программирования; но это не обязательное требование.

Аргументы передаются в функцию в порядке объявления и определения параметров, но без учета какого бы то ни было совпадения имен. Если в функцию `Area()` первым передать аргумент `widthOfYard`, а за ним — аргумент `lengthOfYard`, то эта функция использует значение `widthOfYard` для параметра `yardLength`, а значение `lengthOfYard` — для параметра `yardWidth`. Тело функции всегда заключается в фигурные скобки, даже если оно состоит только из одной строки, как в нашем примере.

Если требуется вызвать функцию до ее определения в рассматриваемом файле, или определение функции находится в другом исходном файле, то вызов функции следует предварять объявлением этой функции. Объявление (прототип) функции имеет следующий формат:

Определение функции

Определение функции состоит из заголовка функции и ее тела. Заголовок подобен прототипу функции за исключением того, что параметры в данном случае именованные и в конце заголовка отсутствует точка с запятой.

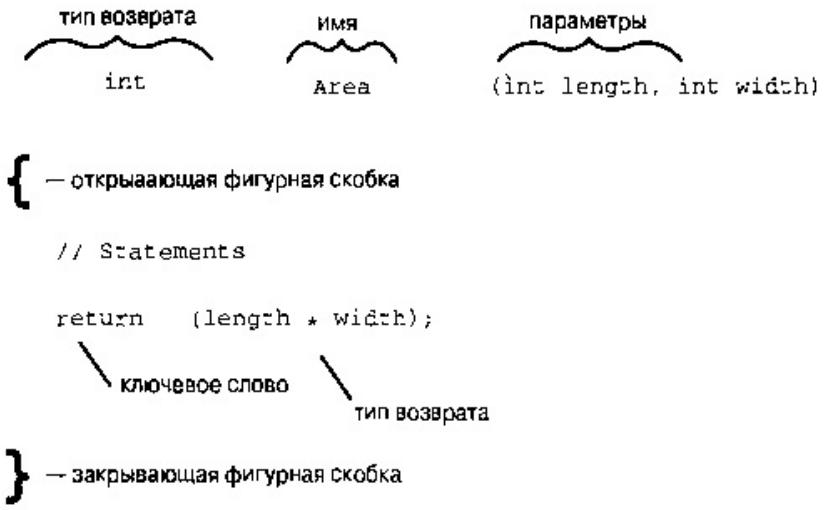

Рисунок 6.3. - Заголовок и тело функции

Тело функции представляет собой набор выражений, заключенных в фигурные скобки. Заголовок и тело функции показаны на рис. 6.3.

Локальные переменные

В функции можно не только передавать значения переменных, но и объявлять переменные внутри тела функции. Это реализуется с помощью локальных переменных, которые называются так потому, что существуют только внутри самой функции. Когда выполнение программы возвращается из функции к основному коду, локальные переменные удаляются из памяти.

О переменных, объявленных внутри функции, говорят, что они имеют *локальную область видимости*. Это означает, что они видимы и пригодны для использования только в пределах той функции, в которой определены. Фактически в C++ можно определять переменные в любом месте внутри функции, а не только в ее начале. Областью видимости переменной является блок, в котором она определена. Таким образом, если в теле функции будет блок, выделенный парой фигурных скобок, и в этом блоке объявляется переменная, то она будет доступна только в пределах блока, а не во всей функции.

Локальные переменные определяются подобно любым другим переменным. Параметры, переданные функции, тоже считаются локальными переменными, и их можно использовать как определенные внутри тела функции. В листинге 6.2 представлен пример использования параметров функции и переменных, локально определенных внутри функции.

Листинг 6.2.Использование локальных переменных и параметров функции

```
1: #include <iostream.h>
2:
3: float Convert(float);
4: int main()
5: {
6: float TempFer;
7: float TempCel;
8:
9: cout << "Please enter the temperature in Fahrenheit: ";
10: cin >> TempFer;
11: TempCel = Convert(TempFer);
12: cout << "\ nHere's the temperature in Celsius: ";
13: cout << TempCel << endl;
14: return 0;
15: }
16:
17: float Convert(float TempFer)
18: {
19: float TempCel;
```

```
20: TempCel - ((TempFer - 32) * 5) / 9;
21: return TempCel;
22: }
```

Результат:

```
Please enter the temperature ]n Fahrenheit: 212
Here's the temperature in Celsius: 100
Please enter the temperature in Fahrenheit: 32
Here's the temperature in Celsius: 0
Please enter the temperature in Fahrenheit: 85
Here's the temperature in Celsius: 29.4444
```

В строках 6 и 7 объявляются две переменные типа `float`: одна (`TempFer`) для хранения значения температуры в градусах по Фаренгейту, а другая (`TempCel`) — в градусах по Цельсию. В строке 9 пользователю предлагается ввести температуру по Фаренгейту, и это значение затем передается функции `Convert`().

После вызова функции `Convert()` программа продолжает выполнение с первого выражения в теле этой функции, представленного строкой 19, где объявляется локальная переменная, также названная `TempCel`. Обратите внимание, что эта локальная переменная — не та же самая переменная `TempCel`, которая объявлена в строке 7. Эта переменная существует только внутри функции `Convert()`. Значение, переданное в качестве параметра `TempFer`, также является лишь переданной из функции `main()` локальной копией одноименной переменной.

В функции `Convert()` можно было бы задать параметр FerTemp и локальную переменную `CelTemp`, что не повлияло бы на работу программы. Чтобы убедиться в этом, можете ввести новые имена и перекомпилировать программу.

Локальной переменной `TempCel` присваивается значение, которое получается в результате выполнения следующих действий: вычитания числа 32 из параметра `TempFer`, умножения этой разности на число 5 с последующим делением на число 9. Результат вычислений затем возвращается в качестве значения возврата функции, и в строке 11 оно присваивается переменной `TempCel` функции `main()`. В строке 13 это значение выводится на экран.

В нашем примере программа запускалась трижды. В первый раз вводится значение 212, чтобы убедиться в том, что точка кипения воды по Фаренгейту (212) сгенерирует правильный ответ в градусах Цельсия (100). При втором испытании вводится значение точки замерзания воды. В третий раз — случайное число, выбранное для получения дробного результата.

В качестве примера попробуйте запустить программу снова с другими именами переменных.

Должен получиться тот же результат.

Каждая переменная характеризуется своей областью видимости, определяющей время жизни и доступность переменной в программе. Переменные, объявленные внутри некоторого блока программы, имеют область видимости, ограниченную этим блоком. К ним можно получить доступ только в пределах этого блока, и после того, как выполнение программы выйдет за пределы, все его локальные переменные автоматически удаляются из памяти. Глобальные же переменные имеют глобальную область видимости и доступны из любой точки программы.

Обычно область видимости переменных очевидна по месту их объявления, но некоторые исключения все же существуют.

Аргументы, переданные функции, локальны по отношению к данной функции. Изменения, внесенные в аргументы во время выполнения функции, не влияют на переменные, значения которых передаются в функцию. Этот способ передачи параметров известен как передача *как значения*, т.е. локальная копия каждого аргумента создается в самой функции. Такие локальные копии внешних переменных обрабатываются так же, как и любые другие локальные переменные функции.

Глобальные переменные

Переменные, определенные вне тела какой-либо функции, имеют глобальную область видимости и доступны из любой функции в программе, включая `main()`.

Локальные переменные, имена которых совпадают с именами глобальных переменных, не изменяют значений последних. Однако такая локальная переменная скрывает глобальную переменную. Если в функции есть переменная с тем же именем, что и у глобальной, то при использовании внутри функции это имя относится к локальной переменной, а не к глобальной. Это проиллюстрировано в листинге 6.3.

Листинг 6.3. Демонстрация использования глобальных и локальных переменных

```
1: #include <iostream.h>
2: void myFunction(); // прототип
3:
4: int x = 5, y = 7; // глобальные переменные
5: int main()
6: {
7:
8: cout << "x from main: " << x << "\ n";
9: cout << "y from main; " << y << "\ n\ n";
10: myFunction();
11: cout << "Back from myFunction!\ n\ n";
12: cout << "x from main: " « x « n";
13: cout << "y from main: " << y << "\ n";
14: return 0;
15: }
16:
17: void myFunction()
18: {
19: int y = 10;
20:
21: cout << "x from myFunction: " << x << "\ n";
22: cout << "y from myFunction: " << y << "\ n\;
23: }
```

Результат:

```
x from main: 5
y from main: 7
x from myFunction: 5
y from myFunction: 10
Back from myFunction!
x from main: 5
y from main: 7
```

Эта простая программа иллюстрирует несколько ключевых моментов, связанных с использованием локальных и глобальных переменных, на которых часто спотыкаются начинающие программисты. В строке 4 объявляются две глобальные переменные - x и y, Глобальная переменная x инициализируется значением 5, глобальная переменная y — значением 7.

В строках 8 и 9 в функции `main()` эти значения выводятся на экран. Обратите внимание, что хотя эти переменные не объявляются в функции `main()`, они и так доступны, будучи глобальными.

При вызове в строке 10 функции `myFunction()` выполнение программы продолжается со строки 18, а в строке 19 объявляется локальная переменная y, которая инициализируется значением 10. В строке 21 функция `myFunction()` выводит значение переменной x. При этом используется глобальная переменная x, как и в функции `main()`. Однако в строке 22 при обращении к переменной y на экран выводится значение локальной переменной y, в то время как глобальная переменная с таким же именем оказывается скрытой.

После завершения выполнения тела функции управление программой возвращается функции `main()`, которая вновь выводит на экран значения тех же глобальных переменных.

Обратите внимание, что на глобальную переменную у совершенно не повлияло присвоение значения локальной переменной в функции `myFunction`().

Следует отметить, что в C++ глобальные переменные почти никогда не используются. Язык C++ вырос из C, где использование глобальных переменных всегда было чревато возникновением ошибок, хотя обойтись без их применения также было невозможно. Глобальные переменные необходимы в тех случаях, когда программисту нужно сделать данные доступными для многих функций, а передавать данные из функции в функцию как параметры проблематично.

Опасность использования глобальных переменных исходит из их общедоступности, в результате чего одна функция может изменить значение переменной незаметно для другой функции. В таких ситуациях возможно появление ошибок, которые очень трудно выявить.

Перегрузка функций

В языке C++ предусмотрена возможность создания нескольких функций с одинаковым именем. Это называется *перегрузкой функций*. Перегруженные функции должны отличаться друг от друга списками параметров: либо типом одного или нескольких параметров, либо различным количеством параметров, либо и тем и другим одновре-менно. Рассмотрим следующий пример:

```
int myFunction (int, int);
int myFunction (long, long);
int myFunction (long);
```

Функция `myFunction()` перегружена с тремя разными списками параметров. Первая и вторая версии отличаются типами параметров, а третья - их количеством.

Типы возвращаемых значений перегруженных функций могут быть одинаковыми или разными. Следует иметь в виду, что при создании двух функций с одинаковым именем и одинаковым списком параметров, но с различными типами возвращаемых значений, будет сгенерирована ошибка комплиляции.

Перегрузка функций также называется *полиморфизмом функций*.

Под полиморфизмом функции понимают существование в программе нескольких перегруженных версий функции, имеющих разные назначения. Изменяя количество или тип параметров, можно присвоить двум или нескольким функциям одно и то же имя. При этом никакой путаницы при вызове функции не будет, поскольку нужная функция определяется по совпадению используемых параметров. Это позволяет создать функцию, которая сможет, например, усреднять целочисленные значения, значения типа double или значения других типов без необходимости создавать отдельные имена для каждой функции - AverageInts(), AverageDoubles() и т.д.

Способы обращения к функциям: вызов по значению и вызов по ссылке.

При вызове функции её аргументы подставляются вместо формальных параметров. Есть несколько механизмов реализации такого процесса подстановки. Передача параметров по значению и передача параметров по ссылке. При передаче аргументов по значению передаётся копия аргумента (т.е. только её значение). Иногда требуется передать функции не значение переменной, а саму переменную. В этом случае надо использовать передачу параметров по ссылке. Для формального параметра, *передаваемого по ссылке,* соответствующий аргумент вызова функции должен быть переменной, и эта переменная подставляется в тело функции вместо формального параметра. После этого при выполнении тела функции значение переменной-аргумента может изменяться.

Чтобы компилятор мог отличить параметр, передаваемый по ссылке, от параметра, передаваемого по значению, необходимо явно указать способ передачи по ссылке. Для этого используется знак амперсанта (&), который располагается после имени типа формального параметра, передаваемого по ссылке, как в прототипе функции, так и в заголовке её определения.

```
#include <iostream.h>
void get_numbers(int& input1, int& input2);
void swap_values(int& variable1, int& variable2);
```

```
void show_results(int output1, int output2);
int main()
{
 int first_num, second_num;
 get_numbers(first_num, second_num);
 swap_values(first_num, second_num);
 show_results(first_num, second_num);
 return 0;
}
 void get_numbers(int& input1, int& input2)
{
   cout << " Введите два целых числа: ";
   cin >> input1 >> input2;
}
void swap_values(int& variable1, int& variable2)
{
  int temp = varieble1;
 varieble1= varieble2;
 varible2 = temp;
}
void show_results(int output1, output2)
{
  cout << " Числа в обратном порядке: "
        << output1 << " " << output2 << endl;
}
```

Вспомним, что переменные программы, не что иное, как ячейки памяти. Компилятор отводит каждой переменной определённое место в памяти. Например, при компиляции программы переменной first_num может быть выделена ячейка 1010, а переменной second_num – ячейка 1012. Эти ячейки используются при всех действиях, выполняемых над переменными. А теперь рассмотрим вызов функции в том же листинге:

Get_numbers(first_num, second_num);

При выполнении этого вызова функции передаются не имена аргументов first_num и second_num, а номера ячеек памяти, связанных с каждым из аргументов. В данном случае это ячейки 1010 и 1012, назначенные переменным input1 и input2, причем номера ячеек перечисляются в том же порядке, что и аргументы. Первая ячейка памяти связывается с первым формальным параметром, вторая -- со вторым и т.д.

first_num -> 1010 -> input1

second_num -> 1012 -> input2

Какие бы действия не осуществлялись над формальным параметром в теле функции, они будут выполняться над переменной в ячейке памяти, связанной с этим формальным параметром. В данном случае инструкции тела функции get_numbers сохраняют введенные пользователем значения в формальных параметрах input1 и input2, т.е. на самом деле – в ячейках памяти 1010 и 1012. Таким образом, какие бы инструкции, касающиеся формальных параметров input1 и input2, ни выполнял компьютер, на самом деле эти действия выполняются с переменными first_num и second_num.

Термины передача по ссылке и передача по значению обозначают механизм, использованный для подстановки. В случае передачи по значению используется только значение аргумента; формальный параметр является локальной переменной, которая инициализируется значением соответствующего аргумента. В случае передачи по ссылке аргумент представляет собой переменную, используемую как таковую. В этом случае происходит подстановка переменной, поэтому любые действия с формальным параметром фактически выполняются с переданной в качестве аргумента переменной.