

jenyay.net

Софт, исходники и фото

Поиск:

>>

[Домой](#) [Блог](#) [Контакты](#)[Печать](#) [Править](#)

Блог

Программки

OutWiker (rus)[Плагины](#)[Бета-версии](#)[Локализации](#)[Документация](#)[Предложения и](#)[баги](#)[Исходники](#)**OutWiker (en)**[Plug-ins](#)[Beta versions](#)[Translate](#)[Suggestions and](#)[bugs](#)[Source code](#)[Documentation](#)**Другие...**

Программирование

[Python](#)[Rust](#)[.NET/C#](#)[C++](#)[PHP](#)[Алгоритмы](#)[Инструменты](#)[Остальное](#)**Обзоры книг**

Программирование скриптов для Vim. Часть 6. Продвинутое использование функций

Предыдущие части

[Часть 1. Запуск скриптов](#)[Часть 2. Переменные](#)[Часть 3. Работа со списками](#)[Часть 4. Работа со строками](#)[Часть 5. Операции ветвления и функции](#)

Оглавление

- [Небольшие полезности](#)
- [Указатели на функции](#)
- [Вызов функции для диапазона строк](#)
- [Функции с переменным числом параметров](#)
- [Практика](#)
- [Комментарии](#)

Дополнение к предыдущей части

Перед тем как мы приступим к новому материалу хотелось бы сделать одно дополнение к [предыдущей части](#). Я не стал делать это дополнение там, где оно должно было бы быть, чтобы его прочитали те, кто ждет новые части статьи. Позже я перенесу этот кусок текста в предыдущую часть.

Как вы помните, аргументы функции записываются через запятую:

Студентам

Фото

Животные
Черно-белые
Пейзажи/Природа
Город
Закаты
Панорамы
Спорт
Репортаж
Разное

Контакты

```
function (param1, param2)
```

Так вот, важно, чтобы после имени аргумента функции и запятой не было пробелов, иначе интерпретатор выдаст ошибку. Следующая строка содержит ошибку:

```
function (param1 , param2)
```

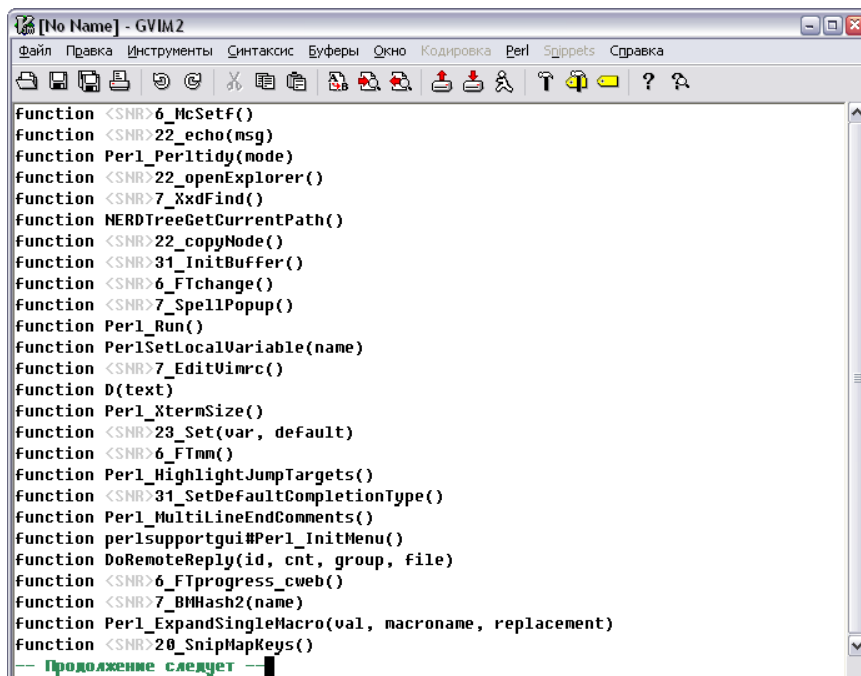
Собственно, это все, что хотелось бы дополнить, а теперь перейдем к новому материалу.

Небольшие полезности

Прежде чем начинать говорить про большие темы, касающиеся функций, хотелось бы рассказать про простые команды, которые могут быть полезны как при отладке функций, так и при работе со сторонними плагинами, например, чтобы убедиться, что они были удачно загружены.

В Vim есть полезная команда `:function` (или сокращенное название `:fu`), которая может работать в трех режимах в зависимости от указанных после команды параметров. Эти режимы лучше всего рассмотреть отдельно.

Первый из этих режимов используется, если после `:function` вообще не указаны параметры. В этом случае Vim просто выведет на экран список всех функций, которые в данный момент хранятся в памяти. Этот список очень большой и не помещается на один экран. Ниже вы можете увидеть результат работы этой команды на моем компьютере, у вас результат работы может быть другим в зависимости от установленных плагинов.



Если перед именем функции стоят символы "<SNR>", значит функция объявлена с областью видимости внутри скрипта, то есть с префиксом *s:*. Имя каждой функции после символов "<SNR>" начинается с цифр, после которых идет символ подчеркивания. Это связано с особенностями хранения функций, объявленных с префиксом *s:*. Число после символов "<SNR>" является своего рода идентификатором скрипта, где функция была объявлена. Но мы уже отвлеклись от основной темы, если хотите чуть подробнее узнать про <SNR>, то наберите в командной строке Vim команду `":help <SNR>"`, там написано чуть более подробно и рассказано, как это применить.

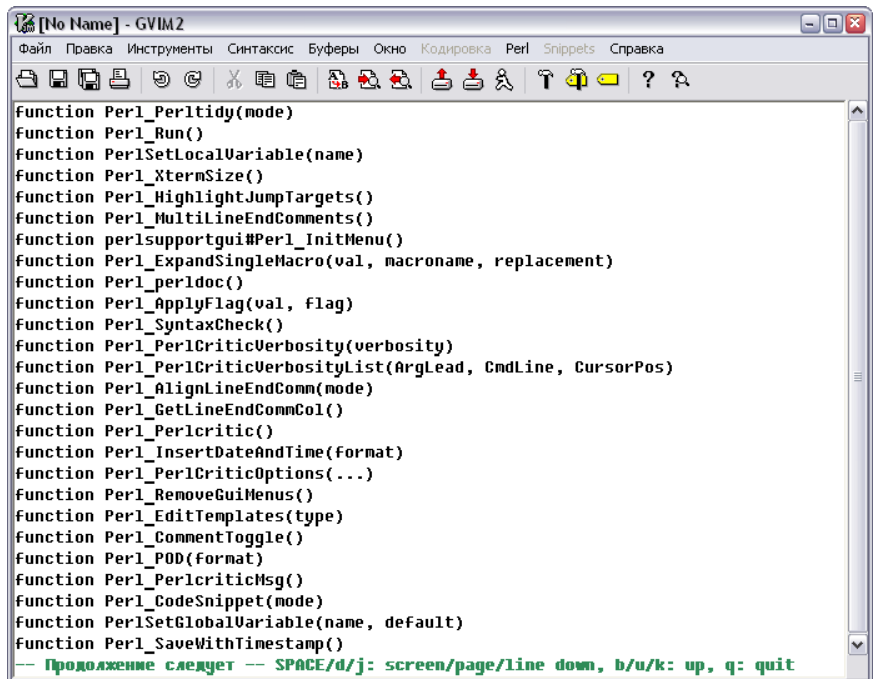
Разумеется, искать нужную функцию в этом списке дело неблагоприятное, поэтому разработчики предусмотрели второй режим работы команды `:function`, который позволяет вывести имена только тех функций, которые соответствуют регулярному выражению. Использование команды `function` в этом случае выглядит следующим образом:

```
:function /шаблон
```

Здесь *шаблон* - это регулярное выражение или, в простейшем случае, подстрока, которую должно содержать имя выводимой функции. Если, для примера, я выполняю команду

```
: function /Perl
```

, то в результате увижу имена следующих функций:



```

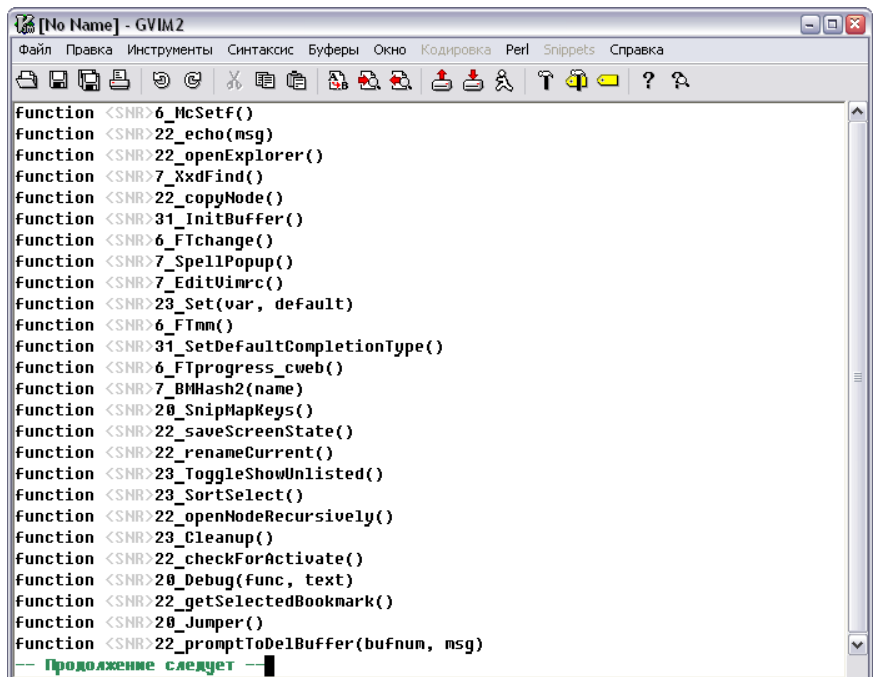
[No Name] - GVIM2
Файл  Правка  Инструменты  Синтаксис  Буферы  Окно  Кодировка  Perl  Snippets  Справка
function Perl_PerlTidy(mode)
function Perl_Run()
function PerlSetLocalVariable(name)
function Perl_XtermSize()
function Perl_HighlightJumpTargets()
function Perl_MultiLineEndComments()
function perlSupportGui#Perl_InitMenu()
function Perl_ExpandSingleMacro(val, macroname, replacement)
function Perl_perldoc()
function Perl_ApplyFlag(val, flag)
function Perl_SyntaxCheck()
function Perl_PerlCriticVerbosity(verbosity)
function Perl_PerlCriticVerbosityList(ArgLead, CmdLine, CursorPos)
function Perl_AlignLineEndComm(mode)
function Perl_GetLineEndCommCol()
function Perl_PerlCritic()
function Perl_InsertDateAndTime(format)
function Perl_PerlCriticOptions(...)
function Perl_RemoveGuiMenus()
function Perl_EditTemplates(type)
function Perl_CommentToggle()
function Perl_POD(format)
function Perl_PerlCriticMsg()
function Perl_CodeSnippet(mode)
function PerlSetGlobalVariable(name, default)
function Perl_SaveWithTimestamp()
-- Продолжение следует -- SPACE/d/j: screen/page/line down, b/u/k: up, q: quit

```

Можно воспользоваться регулярным выражением, чтобы увидеть функции, в именах которых содержатся цифры:

```
:function /\d
```

В результате получим:



```

[No Name] - GVIM2
Файл  Правка  Инструменты  Синтаксис  Буферы  Окно  Кодировка  Perl  Snippets  Справка
function <SNR>6_McSetf()
function <SNR>22_echo(msg)
function <SNR>22_openExplorer()
function <SNR>7_XxdFind()
function <SNR>22_copyNode()
function <SNR>31_InitBuffer()
function <SNR>6_FTchange()
function <SNR>7_SpellPopup()
function <SNR>7_EditVimrc()
function <SNR>23_Set(var, default)
function <SNR>6_FTmm()
function <SNR>31_SetDefaultCompletionType()
function <SNR>6_FTprogress_cweb()
function <SNR>7_BMHash2(name)
function <SNR>20_SnipMapKeys()
function <SNR>22_saveScreenState()
function <SNR>22_renameCurrent()
function <SNR>23_ToggleShowUnlisted()
function <SNR>23_SortSelect()
function <SNR>22_openNodeRecursively()
function <SNR>23_Cleanup()
function <SNR>22_checkForActivate()
function <SNR>20_Debug(func, text)
function <SNR>22_getSelectedBookmark()
function <SNR>20_Jumper()
function <SNR>22_promptToDelBuffer(bufnum, msg)
-- Продолжение следует --

```

Третий режим позволяет посмотреть исходный текст функции по ее имени. Для этого используется следующий синтаксис:

```
:function имя_функции
```

Например, пусть у нас есть скрипт со следующей функцией:

```
function! Hello()  
    echo "Hello, Vim"  
endfunction
```

[Исходник](#)

Выполним его с помощью команды `:source %`. Ничего заметного не произойдет, зато функция будет загружена в память. Вы можете убедиться в этом, выполнив команду `:function /Hello`. Но в данный момент нас больше интересует третий режим, поэтому выполним следующую команду:

```
:function Hello
```

В результате на экран будет выведен исходный текст нашей функции:

```
function Hello()  
1 |   echo "Hello, Vim"  
    endfunction  
Press ENTER or type command to continue
```

Указатели на функции

Среди встроенных типов объектов в Vim есть такой тип как указатель на функцию. Он может быть полезен, если мы захотим создать, например, список функций и вызывать их по индексу. Кроме того, используя указатель на функции вместе со словарями можно программировать в своеобразном объектно-ориентированном стиле, но об этом поговорим в будущих статьях.

Чтобы получить указатель на функцию, нужно вызвать функцию (уж извините за повторение) *function*, которая принимает строковый, параметр, содержащий имя функции, указатель на которую мы хотим получить. Важно, что имя переменной, которой присваивается указатель на функцию, должно начинаться с заглавной буквы, или у этой переменной должна быть ограничена область видимости с помощью одного префиксов "s:", "w:", "t:" или "b:", в противном случае интерпретатор выдаст ошибку. Рассмотрим пример.

```
function! Hello()  
    echo "Hello, Vim"  
endfunction  
  
let RefHello = function ("Hello")  
echo type (RefHello)
```

[Исходник](#)

В этом примере мы создаем указатель на функцию *RefHello*, а затем смотрим что вернет функция *type* для этой переменной. А вернет она значение 2, что и соответствует указателю на функцию. Тот же пример мы можем переписать с использованием области видимости, если мы не хотим создавать глобальную переменную:

```
function! Hello()  
    echo "Hello, Vim"  
endfunction  
  
let s:refHello = function ("Hello")  
echo type (s:refHello)  
  
unlet s:refHello
```

[Исходник](#)

Создавать указатели на функции мы теперь умеем, осталось понять что мы с ними можем делать. А можем мы с помощью них вызывать функцию, на которую они указывают. Для этого предусмотрено два способа. Первый, простой, состоит в том, что мы используем указатель на функцию как саму функцию:

```
function! Hello()  
    echo "Hello, Vim"  
endfunction  
  
let s:refHello = function ("Hello")  
call s:refHello()  
  
unlet s:refHello
```

[Исходник](#)

Здесь все происходит, как и при вызове самой функции.

Есть еще второй, более обходной путь, который состоит в вызове функции *call()* (не путайте с одноименной командой), которая принимает в качестве первого параметра указатель на функцию, а в качестве второго - список параметров (список в прямом смысле, т.е. объект типа *List*). Есть еще необязательный третий параметр, но он касается вызова функции из словаря в стиле ООП, поэтому пока не будем его использовать.

Предыдущий пример теперь можно переписать в следующем виде:

```
function! Hello()  
    echo "Helo, Vim"  
endfunction  
  
let s:refHello = function ("Hello")  
  
call call (s:refHello, [])  
unlet s:refHello
```

[Исходник](#)

Давайте перепишем последние два примера для функции, которая принимает параметры и возвращает значение. Здесь все просто, вызов функции в первом примере будет выглядеть как вызов обычной функции:

```
function! Summ(x, y)  
    return a:x + a:y  
endfunction  
  
let s:refSumm = function ("Summ")  
  
let s:result = s:refSumm (1, 2)  
  
echo s:result  
  
unlet s:refSumm s:result
```

[Исходник](#)

Во втором случае аргументы передаем в виде списка, а значение, возвращаемое функцией *call()* будет соответствовать тому, что возвращает наша функция, указатель на которую мы используем.

```
function! Summ(x, y)  
    return a:x + a:y  
endfunction  
  
let s:refSumm = function ("Summ")  
  
let s:result = call (s:refSumm, [1, 2])  
  
echo s:result  
  
unlet s:refSumm s:result
```

[Исходник](#)

Долго я пытался придумать ситуацию, когда нужно было бы использовать второй способ с использованием функции *call()*, придумалась только такая ситуация, когда у нас есть два списка: список указателей на функции и список, состоящий из списков параметров к этим функциям. То есть примерно следующее:

```
function! Summ(x, y)  
    return a:x + a:y
```

```

endfunction

function! Negative (x)
    return -a:x
endfunction

" Список указателей на функции
let s:refSumm = [function ("Summ"), function ("Negative")]

" Параметры для каждой функции
let s:params = [[1, 2], [50]]

for n in range (len (s:params))
    echo call (s:refSumm[n], s:params[n])
endfor

unlet s:refSumm

```

[Исходник](#)

Теперь проведем эксперимент, который покажет являются ли два указателя на одну и ту же функцию одним и тем же объектом или разными объектами.

```

function! Hello()
    echo "Hello!"
endfunction

let s:ref1 = function ("Hello")
let s:ref2 = function ("Hello")

echo s:ref1 is s:ref2

unlet s:ref1 s:ref2

```

[Исходник](#)

В результате будет выведено "1", т.е. *s:ref1* и *s:ref2* являются "указателем" на один и тот же участок в памяти. Для интерпретируемого языка, пожалуй, не совсем верно говорить про указатели, в том смысле как это понимается в C/C++, поэтому я это слово взял в кавычки. Однако, это поведение интерпретатора очень логично, учитывая, что нет способов (да и смысла) изменения функции по ее указателю, то и нет смысла каждый раз создавать новый объект указателя на одну и ту же функцию.

В завершении раздела рассмотрим пример, показывающий как Vim пытается преобразовывать указатели на функции в строки и обратно. В следующем примере мы получаем указатель на функцию *Hello()*, а затем передаем этот указатель в команду *echo*, после чего этот же указатель "скармливаем" функции *string()* и смотрим что из этого получится.

```

function! Hello()
    echo "Hello!"
endfunction

let s:ref = function ("Hello")

```



```
echo s:ref

let s:strfunc = string (s:ref)
echo s:strfunc

unlet s:strfunc s:ref
```

[Исходник](#)

В результате запуска этого скрипта мы увидим две разные строки:

```
Hello
function('Hello')
```

С первой строкой все понятно, это имя функции, но давайте теперь попробуем вторую строку преобразовать обратно в указатель на функцию.

```
function! Hello()
    echo "Hello!"
endfunction

let s:ref = function ("Hello")

let s:strfunc = string (s:ref)
let s:newref = function (s:strfunc)

echo s:newref
echo type (s:newref)

call s:newref()

unlet s:strfunc s:ref s:newref
```

[Исходник](#)

Этот скрипт вызывает ошибку "E117: неизвестная функция: function('Hello')", но, что самое интересное, само преобразование в указатель на функцию проходит удачно. Даже строки *echo s:newref* и *echo type (s:newref)* работают без ошибок, пока мы не попытаемся вызвать саму функцию. Уж не знаю что это, ошибка интерпретатора или так и было задумано, но вряд ли при реальном использовании такие преобразования в строку и обратно часто понадобятся.

Вызов функции для диапазона строк

Наверняка вы знаете, что многие команды можно вызывать таким образом, чтобы они влияли на определенный интервал строк. Чтобы пояснить это, пусть у нас есть файл с числами, в каждой строке есть цифра 1.

```
1324
```

```
3571
```

```
6125
```

```
1277
```

```
5841
```

```
6161
```

```
1111
```

```
3581
```

Теперь мы хотим заменить все единицы в строках с 3 по 5 на знак подчеркивания "_". Для этого мы выполним следующую команду:

```
:3,5s/1/_/g
```

Перед командой *s*, обозначающей, что мы хотим произвести замену, через запятую стоят два числа, первое обозначает первую строку, к которой нужно применить команду *s*, второе число - номер последней строки. Буква *g* в конце обозначает, что мы хотим все замены в каждой строке, иначе интерпретатор перейдет к следующей строке после первой найденной единицы. Все строки нумеруются, начиная с 1, таким образом, команда *s* будет применена к строкам с номерами 3, 4 и 5.

Было бы здорово, чтобы и наши функции тоже могли использовать такую запись. При таком вызове функции курсор перемещается последовательно по строкам, попадающим в диапазон, поэтому, текущую строку внутри функции можно получить с помощью встроенной функции *getline(".")*. Напомню, что параметр "." здесь означает, что нам нужна та строка, на которой стоит курсор.

Для реализации такой возможности есть два подхода. Первый подход состоит в том, чтобы возложить ответственность за получение очередной строки на вызывающую программу.

Давайте рассмотрим пример.

```
" 1324  
" 3571  
" 6125  
" 1277  
" 5841  
" 6161  
" 1111  
" 3581
```

```
function! Replace(str)
    echo substitute(a:str, "1", "_", "g")
endfunction

3,5call Replace(getline("."))
```

[Исходник](#)

Здесь первые закомментированные строки являются данными для скрипта, который будет обрабатывать сам себя. Это сделано для того, чтобы с одной стороны было видно обрабатываемые данные, а с другой сам скрипт, и чтобы не нужно было отделять скрипт от данных. Разумеется, в реальности так лучше не делать, но для демонстрации так удобнее.

Затем мы объявляем функцию, которая принимает строку, а выводит на экран ее же, но с заменой единиц на символ подчеркивания. Это обычная функция, которая ничем не отличается от предыдущих, она даже может не догадываться о том, как ее будут вызывать.

А вот затем происходит вызов этой функции для строк 3, 4 и 5 с помощью команды *3,5call* Каждый раз, когда будет вызываться функция для очередной строки, ей будет передаваться та строка, на которой стоит курсор.

В результате на экран будет выведено:

```
" 6_25
" _277
" 584_
```

Причем обратите внимание, что после этого курсор в буфере переместится на пятую строку, где бы он до этого ни стоял.

Тот же самый пример можно переписать таким образом, чтобы сама функция получала нужную ей строку.

```
" 1324
" 3571
" 6125
" 1277
" 5841
" 6161
" 1111
" 3581

function! Replace()
    let s:str = getline(".")
    echo substitute(s:str, "1", "_", "g")
```

```
unlet s:str
endfunction

3,5call Replace()
```

[Исходник](#)

Так вызывать ее уже удобнее. При этом курсор также переместится на пятую строку. При этом обратите внимание, что функция в обоих случаях вызывается несколько раз для каждой строки из указанного диапазона.

Другой подход заключается в том, чтобы указать функции диапазон строк, а затем она сама обрабатывала бы их. Для этого можно было бы добавить в функцию два параметра, но лучше воспользоваться возможностью, предоставляемой Vim'ом. Для этого после параметров функции нужно добавить ключевое слово *range*:

```
function FunctionName (param1, param2, ...) range
...
endfunction
```

[Исходник](#)

После такого объявления при вызове этой функции ей будут неявно переданы два параметра : *a:firstline* и *a:lastline*. Первый из них, как можно понять по имени, содержит номер первой строки диапазона, к которому должна быть применена функция, а второй - номер последней строки. В аргументах функции эти параметры объявлять не надо.

Изменим наш предыдущий пример так, чтобы он использовать функцию с ключевым словом *range*:

```
" 1324
" 3571
" 6125
" 1277
" 5841
" 6161
" 1111
" 3581

function! Replace() range
  for n in range(a:firstline, a:lastline)
    let s:str = getline(n)
    echo substitute(s:str, "1", "_", "g")
    unlet s:str
  endfor
endfunction

3,5call Replace()
```

[Исходник](#)

Здесь уже мы явно перебираем все строки в диапазоне, потому что функция будет вызвана только один раз. После окончания работы функции курсор будет перемещен на третью строку - начало диапазона. При необходимости перед завершением работы функция может переместить курсор на конец диапазона с помощью встроенной функции *cursor()*. В этом случае функция примет вид:

```
function! Replace() range
  for n in range(a:firstline, a:lastline)
    let s:str = getline(n)
    echo substitute(s:str, "1", "_", "g")
    unlet s:str
  endfor

  call cursor(a:lastline, 1)
endfunction
```

[Исходник](#)

В качестве первого параметра функции *cursor()* мы передаем номер последней строки в диапазоне, а в качестве второго параметра - номер столбца, единица означает, что курсор должен встать в начало строки.

Теперь заменим в предыдущем примере интервальный вызов функции обычным, без указания диапазона строк:

```
call Replace()
```

[Исходник](#)

В этом случае при запуске скрипта с помощью команды *:source %* функция *Replace()* отработает для одной единственной строки, той, где стоит курсор.

Давайте посмотрим чему в этом случае будут равны переменные *a:firstline* и *a:lastline*:

```
function! Test() range
  echo a:firstline ";" a:lastline
endfunction

call Test()
3,5call Test()
```

[Исходник](#)

Если теперь поставить курсор, например, на вторую строку, то в результате мы получим:

```
2 ; 2
3 ; 5
```

Т.е. если мы функцию с ключевым словом *range* запускаем без указания диапазона строк, то считается, что диапазон строк состоит из одной единственной строки, той, где стоит курсор.

Обратите внимание, что если в этом примере поменять местами последние строки с вызовами функции, то мы всегда будем получать результат

```
3 ; 5 3 ; 3
```

Это происходит из-за того, что после первого вызова функции курсор будет перемещен на 3-ю строку, где он и останется к моменту второго вызова функции.

Кроме ключевого слова *range* есть еще и другие подобные слова: *abort* и *dict*, но о них мы говорить не будем, по крайней мере пока.

Функции с переменным числом параметров

Как и во многих "взрослых" языках программирования, язык, встроенный в Vim позволяет писать функции, имеющие переменное число параметров. Типичным примером встроенной функции с этой возможностью является уже знакомая нам функция *printf()*, предназначенная для форматирования строк. Давайте тоже научимся делать подобные функции.

Для того, чтобы показать, что функция может принимать переменное число параметров, в ее объявлении после списка всех параметров нужно через запятую добавить три точки:

```
function FuncName (param1, param2, ...)  
    ...  
endfunction
```

Для того, чтобы получить доступ к параметрам, попадающими в это многоточие, в функцию неявно передаются следующие параметры:

- **a:000** - список всех переданных дополнительных параметров. Обратите внимание на то, что в этот список НЕ входят переменные, объявленные явно, то есть в приведенном выше примере *param1* и *param2*.
- **a:0** - количество переданных дополнительных параметров.
- **a:1**, **a:2** и т.д. до **a:20** - дополнительные параметры.

Таким образом, функция может принимать не больше 20 дополнительных параметров, при этом добраться до них можно несколькими способами:

- Использовать доступ по индексу в списке *a:000*. Например, *a:000[0] + a:000[2]*.
- Явно указывать переменную, которая нас интересует. Например, *a:1 + a:3*.
- Использовать запись вида *a:{n}*, где *n* может быть переменной. Если *n = 3*, то выражение *a:{n}* равносильно выражению *a:3*.

Наиболее универсальными являются первый и третий способ, т.к. второй требует явного указания номера переменной в виде константы, что не всегда возможно.

Рассмотрим пример:

```
function! s:Summ(x, y, ...)
    let summ = a:x + a:y

    for n in range (a:0)
        let summ += a:000[n]
    endfor

    return summ
endfunction

echo s:Summ(1, 2, 3, 4)

delfunction s:Summ
```

[Исходник](#)

В этом примере мы создаем функцию, суммирующую переданные числа. Первые два параметра являются обязательными (суммируются как минимум два числа), а затем можно добавить дополнительные параметры, если они необходимы.

В цикле происходит перебор всех дополнительных параметров по их индексу и суммирование с уже просуммированными обязательными параметрами *a:x* и *a:y*.

Здесь используется доступ по индексу, чтобы еще раз напомнить о параметре *a:0*, содержащем количество дополнительных параметров, а, в принципе, этот пример можно переписать следующим образом:

```
function! s:Summ(x, y, ...)
  let summ = a:x + a:y

  for s:element in a:000
    let summ += s:element
  endfor

  unlet! s:element
  return summ
endfunction

echo s:Summ(1, 2, 3, 4)

delfunction s:Summ
```

[Исходник](#)

Здесь мы перебираем уже сами элементы списка, а затем для чистоты удаляем переменную *s:element*, когда она нам больше не нужна. Обратите внимание, что здесь обязательно нужно использовать команду *unlet!* с восклицательным знаком. Ведь дополнительных параметров может и не быть, а тогда переменная *s:element* не будет создана и команда *unlet* вызовет ошибку.

Третий вариант того же примера может выглядеть следующим образом:

```
function! s:Summ(x, y, ...)
  let summ = a:x + a:y

  for n in range(1, a:0)
    let summ += a:{n}
  endfor

  return summ
endfunction

echo s:Summ(1, 2, 3, 4)

delfunction s:Summ
```

[Исходник](#)

Здесь мы используем конструкцию вида *a:{n}*, для чего список перебираемых индексов создается с помощью функции *range()*, которой передается в качестве первого параметра единица. Это происходит из-за того, что первый

дополнительный параметр имеет вид `a:{1}`, то есть индексация начинается с единицы, а не с нуля.

Практика

В завершение, как всегда, напишем что-нибудь полезное. К Vim'u есть очень удобный плагин [TODO List](#), который позволяет вести список дел в текстовом файле. По сути это плагин представляет собой просто подсветку слов TODO, DONE и INPROGRESS.

Этот плагин я использую при написании статей, чтобы ничего не забыть записать. Вот, для примера, как выглядит часть плана этой статьи в процессе ее написания:

```

19
20 ==> DONE Ключевое слово range
21 | | --> DONE [range]call
22 | | --> DONE Перемещение курсора
23 | | --> DONE a:firstline, a:lastline
24 | | | --> DONE Вызов вида 10,15call Number()
25 | | | --> DONE Курсор перемещается по этим позициям
26 | | | --> DONE line(".")
27 |
28
29 ==> INPROGRESS Переменное число параметров
30 | | --> DONE a:0 - количество параметров
31 | | --> TODO a:1... - параметры
32 | | --> TODO До 20 аргументов
33 | | --> TODO обращаться к ним можно в виде a:{number}
34 | | --> TODO a:000 - список параметров
35 | | | --> TODO a:000[0] == a:1
36 |
37 ==> DONE Ключевое слово abort
38
39 ==> TODO Примеры
40 | --> TODO Визуальный режим

```

Я человек ленивый, поэтому очень лень исправлять каждый раз статус TODO на DONE или INPROGRESS, особенно если нужно исправить сразу несколько строк. Поэтому для себя я сделал плагин, который переключает по очереди статус TODO сначала на INPROGRESS, затем на DONE, а затем опять на TODO. Исходник его очень простой:

```

function! s:NextStatus() range
  for n in range (a:firstline, a:lastline)
    let line = getline (n)

    if line =~ '\CTODO'
      let line = substitute (line, '\CTODO',
'INPROGRESS', "g")
    elseif line =~ '\CINPROGRESS'
      let line = substitute (line, '\CINPROGRESS',
'DONE', "g")
    elseif line =~ '\CDONE'
      let line = substitute (line, '\CDONE', 'TODO',
"g")
    endif
  endfor

```

```
        call setline (n, line)
    endfor
endfunction

command! -range NextStatus <line1>,<line2> call s:NextStatus()
```

[Исходник](#)

Плагин состоит всего из одной функции, работающей с диапазоном строк. Внутри нее нет ничего примечательного, все операторы, которые в ней используются были разобраны в [прошлой части статьи](#). Обратить внимание нужно на строку, которая создает пользовательскую команду:

```
command! -range NextStatus <line1>,<line2> call s:NextStatus()
```

[Исходник](#)

Она нам уже тоже встречалась в [прошлый раз](#), однако в более простой форме. Здесь мы используем дополнительный параметр *-range*, который обозначает, что к создаваемой пользовательской команде можно применять интервальный вызов. Если этот параметр не задать, то команда сможет обработать за один раз только одну строку (параметры *a:firstline* и *a:lastline* будут передаваться, но они будут равны между собой).

К параметру *-range* также можно задать дополнительные параметры. Сейчас мы их рассматривать не будем, скажу только, что сейчас, если не указать диапазон строк, то по умолчанию будет считаться, что команду нужно применить к одной строке. Если создать команду с использованием команды *-range=%*, то по умолчанию будет считаться, что ее нужно применить ко всему файлу.

Кроме параметра *-range* при создании команды можно задавать и другие параметры, но о них тоже пока говорить не будем, оставим это на будущее.

Затем в строке создания команды мы используем выражения *<line1>* и *<line2>*. Они представляют собой аналог параметров *a:firstline* и *a:lastline*, то есть *<line1>* хранит номер первой строки, к которой должна быть применена команда, а *<line2>* - номер последней строки. Таким образом, в команду передается диапазон строк.

Теперь если этот плагин сохранить в папку *plugins*, то можно использовать команду *:NextStatus*. Если она вызывается именно таким образом, то она будет применена

к текущей строке. Если ее вызвать с помощью выражения `:3,5NextStatus`, то она будет применена к диапазону строк. Больше того, строки можно выделить в визуальном режиме и для них выполнить команду, тогда не надо будет задумываться о номерах строк.

Для еще большего удобства выполнение этой команды можно "повесить" на горячую клавишу. Например, мне удобно выполнять ее с помощью комбинации `Ctrl-F12`, для чего в файле `_vimrc` (или `.vimrc` под `*nix`) нужно добавить строки:

```
nmap <C-F12> :NextStatus<CR>
vmap <C-F12> :NextStatus<CR>
```

[Исходник](#)

Теперь эта горячая клавиша работает и в нормальном, и в визуальном режиме, и не надо набирать довольно длинную команду.

На сегодня это пока все, продолжение следует. Следующая часть статьи будет посвящена словарям, чем мы и закончим рассмотрение основных типов в Vim и дальнейшие статьи можно будет посвятить уже непосредственно написанию различных типов плагинов.

[Часть 7. Словари и объектно-ориентированное программирование](#)

Вы можете подписаться на новости сайта через [RSS](#), [Группу Вконтакте](#) или [Канал в Telegram](#).



Рейтинг 5.0/5. Всего 20 голос(а, ов)

☐ Плохо ☐ Так себе ☐ Неплохо ☐ Хорошо ☐ Отлично

Голосовать

Andrey 05.04.2020 - 15:46

Vim

В закладки!!!

какой чудесный vim!!!



[Подписаться на комментарии](#)

Автор:

Тема:

Ваш комментарий

/

B

U

^

~

x²

x₂

h

≡

Ab

l@

T

T

T

☺

???

☺

😏

😈

😬

😇

😏

😈

😬

😇

Введите код

419

Послать

Human 26.02.2010 - 19:30

VIM

Спасибо за ваши уроки. Раньше VIM мне просто нравился, а теперь я его обожаю!