

# Введение в Perl

## Простейший сценарий

Все программы, написанные на языке Perl, должны начинаться с указания пути к интерпретатору Perl. Как правило, на сервере интерпретатор находится в директории /usr/bin/. Таким образом, первая строка сценария будет выглядеть как:

```
#!/usr/bin/perl
```

Далее, если программы выводит что-нибудь в браузер (как правило, это текстовая информация, но также можно выводить и изображения), то необходимо вначале вывести заголовок:

```
print "Content-type: text/html\n\n";
```

Функция print служит для вывода информации. Выведем с помощью этой функции приветствие:

```
print "Hello, World";
```

Таким образом, простейший сценарий, который выводит в браузер слова «Hello, World» будет выглядеть следующим образом:

```
#!/usr/bin/perl -w
print "Content-type: text/html\n\n";
print "Hello, World";
Hello, World
```

### Примечание

Курсивом в конце примера скрипта выделяется результат работы данного скрипта.

Далее необходимо разместить файл со сценарием на сервере в директории /cgi-bin/ и установить для данного файла права (CHMOD) на исполнение, такими правами могут быть следующие – 750, 755, 775, 777. Рекомендуется устанавливать права 750, однако это лучше всего уточнить у хостинг провайдера.

### Важно!

Все операторы в Perl должны заканчиваться точкой с запятой (;). В некоторых случаях: последний оператор в программе, подпрограмме или последний оператор в блоке, то точку с запятой в конце можно не ставить.

### Важно!

На сервере все скрипты, написанные на perl, должны находиться в директории /cgi-bin/, в других директориях эти скрипты выполняться не будут.

### Важно!

Файлы на сервер следует закачивать по FTP в режиме ASCII, иначе скрипты могут не работать. Это связано с тем, что в операционной системе Windows перевод строки выглядит как \r\n, а в операционных системах \*nix (под управлением, которых работают большинство серверов в интернете) перевод строки выглядит как \n. При закачивании в режиме ASCII происходит замена окончания строки на \n.

## Примечание

Программа CuteFTP последних версий (например, версии 4) сама переключает режимы загрузки файлов на сервер.

## Комментарии в Perl

Комментарии в Perl начинаются с символа #. Perl игнорирует весь текст, идущий от символа # до конца строки.

## Ключ -w и проверка синтаксиса

В предыдущем примере в первой строке при задании пути к перлу был указан ключ `-w`. Этот ключ выводит различные предупреждающие сообщения, среди них:

- имена переменных, упоминающихся только один раз,
- скалярные переменные (вроде простых переменных), которые используются до инициализации,
- переопределенные подпрограммы,
- ссылки на неопределенные дескрипторы файлов,
- дескрипторы файлов, открытых только для чтения, но для которых производится попытка записи,
- значения, используемые как числа, но выглядящие иначе, чем числа,
- использование массива как скалярной переменной,
- подпрограммы с глубиной рекурсии более 100.

## Скалярные переменные

Все скалярные переменные в Perl должны начинаться с символа \$, например:

```
$foo = 5;
```

В данном случае, переменной \$foo присвоили числовое значение 5. В отличие от других языков программирования, в Perl не нужно предварительно инициализировать.

### Важно

Имена скалярных переменных чувствительны к регистру: имя \$foo – это не то же самое, что имя \$Foo.

Символ \$, с которого начинается имя скалярной переменной, называется в Perl *разыменовывающим префиксом*. Вот другие разыменовывающие префиксы, которые используются в Perl:

- \$ – скалярные переменные,
- % – хэши (они же – ассоциативные массивы)
- @ – массивы
- & – подпрограммы

## Работа со строками

В скалярных переменных можно хранить как числа, так и строки:

```
$foo = "Hello";
```

Для объединения строк используется оператор конкатенации Perl, в роли которого используется точка (.):

```
$foo = "Hello ";  
$biz = "there\n";  
print $foo . $biz;
```

Строки могут задаваться с помощью одинарных или двойных кавычек:

```
$foo = "Hello";  
$biz = 'there';
```

Между этими способами есть различие. Если строка ограничена двойными кавычками, то Perl вычисляет переменные, встречающиеся в данной строке. Если строка ограничена одинарными кавычками, то Perl выводит данную строку как есть.

В строках, ограниченных двойными кавычками, можно использовать escape-последовательности, управляющие их форматированием и позволяющие задавать символы, которые иначе записать не удастся. Например, чтобы внести в текст двойную кавычку, можно использовать последовательность \"

```
print "I said \"Hello\".";
```

### Некоторые escape-последовательности

Символ	Значение
\'	Одинарная кавычка или апостроф (')
\"	Двойная кавычка (")
\\	Обратная косая черта (\)
\\$	Символ доллара (\$)
\@	Символ at-коммерческое (@)
\t	Символ табуляции (TAB)
\n	Символ новой строки (LF)
\e	Символ escape (ESC)
\u	Сделать следующую литеру заглавной
\l	Сделать следующую литеру строчной
\U	Сделать следующую группу литер (до команды \E) заглавными
\L	Сделать следующую группу литер (до команды \E) строчными
\Q	В следующей группе литер (до команды \E) считать, что ко всем небуквенным литерам добавлена обратная косая черта, – это заставляет Perl интерпретировать их как обычные символы
\E	Завершает команды \L, \U, \Q

### Подстановка переменных (интерполяция строк)

При использовании в строке, заключенной в двойные кавычки, имена переменных, Perl подставляет вместо них значения присвоенные переменным. Например, если есть переменная \$text, в которой храниться слово Hello:

```
$text = "Hello";
```

то можно использовать её имя в теле строки, и Perl подставит Hello вместо имени переменной:

```
$text = "Hello";  
print "Perl says: $text!\n";  
Perl says: Hello!
```

Однако если заключить тело строки в одинарные кавычки (апострофы), то Perl не будет выполнять интерполяцию:

```
$text = "Hello";  
print 'Perl says: $text!';  
Perl says: $text!\n
```

Если в строке идет переменная, а следом сразу следует слово, то для обозначения переменной её следует заключать в фигурные скобки:

```
$text = "un";  
print "Don't be ${text}happy.";  
Don't be unhappy.
```

## Массивы

Для создания массива необходимо присвоить переменной массиву в качестве значения список (в Perl такие переменные начинаются с префикса @):

```
@array = (1, 2, 3);
```

Чтобы сослаться на отдельные элементы массива, следует указать индекс элемента в квадратных скобках и заменить префикс @ на префикс \$ – это показывает, что работаем со скаляром. Следует обратить внимание, что индексы для массивов отсчитываются от нуля:

```
print $array[0];  
1
```

Для того чтобы вывести весь массив целиком, нужно записать:

```
print @array;  
123
```

Скалярная переменная \$#array содержит количество элементов в массиве:

```
print $#array;  
3
```

## Хэши

Хэш-таблицы (хэшированные таблицы, или просто хэши), называются также ассоциативными массивами для доступа к отдельным элементам данных, используют не индексы, а ключи. При использовании хэшей значения ассоциируются с текстовыми ключами, например:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
```

Теперь можно использовать эти данные, применив для доступа к ним ключ

```
print $hash{sandwich};
hamburger
```

Представление данных в виде хэшей, как правило, более интуитивно (в отличие от массивов), поскольку для извлечения данных используются ключи. Хэши являются идеальным средством для записи данных.

## Операторы циклов

Операторы циклов являются мощным инструментом программирования. Оператор цикла продолжает выполнять команды, входящие в его тело, пока не будет выполнено заданное условие.

### Оператор цикла *for*

Данный цикл задается следующим образом:

```
for ($i = 1; $i <= 6; $i++){
    print $i;
}
123456
```

### Оператор цикла *while*

Оператор цикла *while* задается следующим образом:

```
$i = 1;
while ($i <= 6){
    print $i;
    $i++;
}
123456
```

Тело цикла выполняется, пока выражение в заголовке цикла остается *истинным* (перед каждым выполнением тела цикла оно вычисляется повторно).

### Оператор цикла *until*

Оператор цикла *until* выполняет те же функции, что и *while*, за тем исключением, что для выполнения тела цикла требуется, чтобы проверяемое условие было *ложью*. Этот цикл записывается так:

```
$i = 1;
until ($i > 6){
    print $i;
    $i++;
}
123456
```

## Условный оператор if

Оператор if является базовым условным оператором в Perl. Он проверяет условие, заданное в круглых скобках, и если результат вычислений даст ненулевое значение, выполняется блок команд, ассоциированный с данной командой. Можно также задать блок команд, выполняемых в случае ложности проверяемого условия. Это делается с помощью блока else. Конструкция elsif (обратите внимание: не else if и не elseif) продолжает проверку дополнительных условий. Вот как записывается эта команда:

```
if (выражение) {блок}
if (выражение) {блок} else {блок}
if (выражение) {блок} elsif {блок} ... else {блок}
```

Рассмотрим пример. Оператор проверки на равенство оценивает, равно ли значение указанной переменной пяти, и если это так, сообщает о результате пользователю:

```
$variable = 5;
if ($variable == 5) {
    print "Yes, it's five.\n";
}
Yes, it's five.
```

Можно добавить дополнительный раздел else, который будет информировать пользователя о том, что проверка не прошла:

```
$variable = 6;
if ($variable == 5) {
    print "Yes, it's five.\n";
} else {
    print "No, it's not five.\n";
}
No, it's not five.
```

Наконец, для выполнения произвольного количества проверок можно добавить разделы elsif:

```
$variable = 2;
if ($variable == 1) {
    print "Yes, it's one.\n";
} elsif ($variable == 2) {
    print "Yes, it's two.\n";
} elsif ($variable == 3) {
    print "Yes, it's three.\n";
} else {
    print "Sorry, can't match it!\n";
}
Yes, it's two
```

### **Команда unless**

Команда unless является как бы изнанкой if: она работает так же, но ассоциированный с условием блок выполняется, если условие оказывается ложью. Вот как выглядит эта команда:

```
unless (выражение) {блок}
unless (выражение) {блок} else {блок}
unless (выражение) {блок} elsif {блок} . . . else {блок}
```

# Использование регулярных выражений

В Perl имеются три основных оператора, работающих со строками:

- `m/.../` – проверка совпадений (matching),
- `s/.../.../` – подстановка текста (substitution),
- `tr/.../.../` – замена текста (translation).

Оператор `m/.../` анализирует входной текст и ищет в нем подстроку, совпадающую с указанным шаблоном (он задан регулярным выражением). Оператор `s/.../.../` выполняет подстановку одних текстовых фрагментов вместо других, используя для этой цели регулярные выражения. Оператор `tr/.../.../` также изменяет входной текст, но при этом он не использует регулярные выражения, осуществляя замену посимвольно.

## Оператор проверки совпадений `m/.../`

Оператор `m/.../` пытается сопоставить шаблон, указанный в качестве аргумента, и заданный текст (по умолчанию текст берется из переменной Perl `$_`). В приведенном ниже примере мы ищем во вводимом пользователем тексте строку `exit` (модификатор `i` после второй наклонной черты делает проверку нечувствительной к регистру):

```
while (<>)
  { if (m/exit/i) {exit;} }
```

Вместо того чтобы использовать переменную `$_`, можно задать источник проверяемого текста с помощью оператора `=~`. В нашем примере для этой цели используется переменная `$line` (данный код не меняет содержимого переменной `$line`, хотя оператор `=~` и напоминает символ присвоения):

```
while ($line = <>)
  { if ($line =~ m/exit/i) {exit;} }
```

Смысл сравнения можно изменить на противоположный, если вместо оператора `=~` использовать оператор `!~`:

```
while ($line = <>)
  { if ($line !~ /exit/i) {} else {exit;} }
```

Поскольку в Perl оператор `m/.../` используется очень часто, можно использовать его сокращенную форму, опустив начальную букву `m`. Если же начальная буква `m` присутствует, то вместо символов наклонной черты (слэша) в качестве ограничителей можно использовать, за редким исключением, любой другой символ (см. далее описание оператора подстановки `s/.../.../`):

```
while ($line = <>) {
  if ($line =~ /exit/i) {exit;}
  if ($line =~ m|quit|i) {exit;}
  if ($line =~ m%stop%i) {exit;}
}
```

### Подсказка.

Если шаблон содержит символы косой черты, что зачастую встречается, например, при анализе имен файлов с указанием пути и/или меток HTML, то стоит отказаться от использования этих символов в качестве ограничителя. Это позволит не ставить обратную косую черту перед каждым символом косой черты внутри шаблона и сохранит его прозрачность.

## Оператор подстановки `s/.../.../`

Оператор `s/.../.../` выполняет замену одних фрагментов текста на другие. Например, в следующем случае мы заменяем подстроку **young** на подстроку **old**:

```
$text = "Pretty young.";
$text =~ s/young/old/;
print $text;
Pretty old.
```

По умолчанию оператор замены работает с переменной Perl `$_`. Как и в случае оператора `m/.../`, косую черту использовать не обязательно – годится любой символ, который не вступает в противоречие с заданным выражением. Например, вместо косой черты можно использовать:

```
$text = "Pretty young.";
$text =~ s\young\old\;
print $text;
Pretty old.
```

либо просто поставить в качестве ограничителя обычные скобки:

```
$text = "Pretty young.";
$text =~ s(young)(old);
print $text;
Pretty old.
```

Обратите внимание, что операторы `s/.../.../` и `m/.../` ведут поиск с первого символа текстовой строки до первого совпадения. Если оно найдено, без специального указания поиск не продолжается:

```
$text = "Pretty young, but not very young.";
$text =~ s/young/old/;
print $text;
Pretty old, but not very young.
```

## Оператор замены `tr/.../.../`

Кроме операторов `m/.../` и `s/.../.../` для работы со строками в Perl имеется оператор `tr/.../.../`. Он также выполняет замену одних фрагментов текста на другие, однако в отличие от `s/.../.../`, не пытается обрабатывать регулярные выражения, подставляя текст один к одному. В следующем примере мы заменяем с его помощью букву «o» на букву «i»:

```
$text = "His name is Tom.";
$text =~ tr/o/i/;
print $text;
His name is Tim.
```

# Подпрограммы

## Объявление подпрограмм

Объявление можно использовать, чтобы сообщить Perl о существовании подпрограммы и уточнить при этом типы передаваемых аргументов и возвращаемого значения. Объявление (declaration) подпрограммы отличается от определения (definition) - при определении задается код, составляющий тело подпрограммы.

В отличие от других языков программирования, в Perl перед использованием объявлять подпрограммы не надо. Исключением является вызов подпрограмм (функций) без круглых скобок, охватывающих список параметров (например, списочных операторов). В этом случае перед тем, как использовать подпрограмму, необходимо ее объявить.

Подпрограмма может быть объявлена одним из следующих способов:

```
sub имя;  
sub имя (список-аргументов);  
sub имя {блок};  
sub имя (список-аргументов) {блок};
```

Определение отличается от объявления тем, что в нем приводится код, составляющий тело подпрограммы. Чтобы определить подпрограмму, используется ключевое слово `sub`:

```
sub имя {блок};  
sub имя (список-аргументов) {блок};
```

Например, требуется задать подпрограмму **printhello**, просто выводящую сообщение «Hello!» (обратите внимание, что тело подпрограммы заключено в фигурные скобки `{ }`, хотя и состоит из одной команды):

```
sub printhello  
{  
  print "Hello!\n";  
}
```

Теперь можно вызвать эту подпрограмму:

```
printhello;  
Hello!
```

## Вызов подпрограмм

Если подпрограмма определена, ее можно вызвать с конкретными аргументами в качестве параметров:

```
&имя (список-аргументов);
```

В Perl выполнить вызов подпрограммы можно далеко не одним способом. Например, при использовании круглых скобок не обязательно ставить префикс `&`:

```
имя (список-аргументов);
```

При вызове подпрограммы передаваемые ей аргументы помещаются в специальный массив `@_`. Если подпрограмма вызывается с префиксом `&`, но без списка параметров, то в качестве последнего

ей передается текущее содержимое массива `@_`. Это полезно в том случае, когда одна подпрограмма вызывается из другой, причем ей требуется передать те же параметры, которые использовались при вызове первой подпрограммы.

## **Чтение аргументов, переданных подпрограмме**

Доступ к аргументам, переданным подпрограмме, осуществляется через специальный массив `@_`, в который заносятся эти аргументы. Например, если переданы два параметра, подпрограмма может обратиться к ним как `$_[0]` и `$_[1]`.

Предположим, что вы хотите сложить два числа и напечатать результат. Для этой цели создается процедура **addem**, которую можно вызвать, как **addem(2,2)**. Посмотрим, как **addem** получает значения через массив `@_`:

```
sub addem
{
    $value1 = $_[0];
    $value2 = $_[1];
    print "$value1 + $value2 = " . ($value1+$value2);
}
addem(2,2);
2 + 2 = 4
```

Чтобы извлечь параметры из массива `@_`, можно также использовать функцию **shift**:

```
sub addem
{
    $value1 = shift @_;
    $value2 = shift @_;
    print "$value1 + $value2 = " . ($value1+$value2);
}
addem(2,2);
2 + 2 = 4
```

Наконец, в качестве третьего метода, чтобы получить значение параметров за один раз, можно использовать присвоение списком:

```
sub addem
{ ($value1, $value2) = @_;
  print "$value1 + $value2 = " . ($value1+$value2); } addem(2,2); }
2 + 2 = 4
```

## **Значения, возвращаемые подпрограммами (функциями)**

Задать возвращаемое значение и выйти из подпрограммы можно также с помощью команды **return**. (В некоторых языках программирования функции возвращают значение, а подпрограммы этого делать не могут. Однако в Perl функция и подпрограмма - это одно и то же.)

Например, вот как передать подпрограмме два параметра и вернуть их сумму:

```
sub addem {
    ($value1, $value2) = @_;
    return $value1+$value2;
}
print "2 + 2 = " . addem(2, 2) . "\n";
2 + 2 = 4
```

# Работа с файлами

## ***open*** - открытие файла

Чтобы открыть файл, следует использовать функцию **open**:

open дескриптор, выражение  
open дескриптор

Эта функция открывает файл с заданным именем и создает указанный дескриптор файла. После ее вызова дескриптор может использоваться для ссылок на файл в самых разных операциях. Если имя не задано, Perl пытается открыть файл с именем, совпадающим с именем дескриптора.

Функция **open** возвращает ненулевое значение (соответствует условию *истина*), если файл успешно открыт, и неопределенное значение (соответствует условию *ложь*), если сделать этого не удалось.

Имя файла может содержать дополнительные символы, указывающие, как именно следует открыть его.

- Если имя имеет префикс < Или не имеет префикса, файл открывается для чтения.
- Если имя имеет префикс >, файл открывается для записи и полностью очищается (если он уже существует) или же создается новый файл.
- Если имя имеет префикс >>, файл открывается для записи, а данные дописываются в его конец. Если файл не существует, создается новый.
- Если имя имеет префикс +<, файл открывается для чтения и записи. Если файл существует, его содержимое сохраняется.
- Если имя имеет префикс +>, файл открывается и для чтения и записи, однако если он уже существует, то полностью очищается.
- Если в качестве имени файла задан дефис -, функция открывает стандартный поток ввода (обычно STDIN).
- Если в качестве имени файла заданы символы >-, функция открывает стандартный поток вывода (обычно STDOUT).

В следующем примере файл открывается для записи, и в него выводится некоторый текст:

```
open (FILEHANDLE, ">hello.txt") or die ("Cannot open file hello.txt");  
print FILEHANDLE, "Hello!";  
close (FILEHANDLE);  
Hello!
```

## ***close*** - закрытие файла

Функция **close** закрывает открытый файл или канал по окончании работы с ним. При этом в файл пересылаются все данные, еще находящиеся в буфере вывода, а дескриптор файла деинициализируется, так что дальнейшие операции с ним (кроме открытия нового файла) становятся невозможны:

close дескриптор

## ***flock – блокировка файлов***

Функция flock позволяет «заблокировать» файл, заданный дескриптором, для доступа со стороны других процессов:

```
flock(дескриптор, код-операции);
```

Здесь **код-операции** - это одна из следующих констант:

- \$LOCK\_SH = 1; – Shared-блокировка
- \$LOCK\_EX = 2; – Exclusive-блокировка
- \$LOCK\_NB = 4; – Неблокирующая блокировка
- \$LOCK\_UN = 8; – Разблокировать

В приведенном ниже примере программа добавляет запись в файл с использованием блокировки. Если другая копия процесса попытается обратиться к заблокированному файлу, она будет просто ожидать, пока файл не перейдет в разблокированное состояние. (Следует особо отметить, что во время ожидания процессорное время этому процессу операционной системой не выделяется.)

```
open(FILE,">>$mydatafne"); #откроем файл
flock(FILE,$LOCK_EX):      #заблокируем файл эксклюзивно
print FILE $data;          #запишем данные
flock(FILE,$LOCK_UN);      #снимем блокировку
close(FILE):               #закроем файл
```