УПРАВЛЕНИЕ ЦИКЛАМИ И ПЕРЕХОДЫ

— предложения для управления выполнением программы с помощью перехода в указанную точку программы.

Обычно это требуется при работе с циклами.

- Когда при работе цикла требуется досрочно закончить его выполнение при наступлении какого-то события, то для этого можно воспользоваться оператором ***last*** (аналог оператора ***break*** в языке C), который прерывает выполнение цикла и переходит к выполнению предложения, следующего за циклом.

Пример, напечатать 10 случайных чисел от 0 до 0,5 можно так:

```
$i = 0;
while (1) { # безусловно входим в цикл
    $random = rand; # получаем случайное число
    if ($i == 10) { # по достижении счетчика 10
        last; # ПРЕРЫВАЕМ выполнение цикла
    }
    if ($random < 0.5) { # числа меньше 0.5
        print "\n", $random; # выводим на консоль
        $i++; # и увеличиваем счетчик повторений
    }
}
# сюда произойдет переход по last
```

```
0.412671972823546
0.212211689669381
0.0776656116658963
0.374833484020652
0.455088786635422
0.342799212158759
0.406413982456701
0.17105096164282
0.168440052830476
```

- Оператор ***next*** (аналог оператора continue в языке C) применяется, когда требуется пропустить выполнение оставшихся предложений в теле цикла и перейти к следующей итерации, начав с проверки условия в заголовке цикла.

Пример. Изменить последний пример, применив next:

```
$i = 0;
while ($i < 10) { # пока счетчик не достигнет 10
    $random = rand; # получаем случайное число
    if ($random > 0.5) { # числа больше 0.5
        next; # ПРОПУСКАЕМ действия в теле цикла
    }
```

```
    print "\n", $random; # выводим число на консоль

    $i++; # и увеличиваем счетчик повторений

    # сюда произойдет переход по next

}
```

- Оператор ***redo*** используется, когда нужно повторить выполнение предложений в теле цикла без проверки условия в заголовке цикла.

Пример. Изменить последний пример, применив redo:

```
$i = 0;

while ($i < 10) { # пока счетчик не достигнет 10

    # сюда произойдет переход по redo

    $random = rand; # получаем случайное число

    if ($random > 0.5) { # числа больше 0.5

        redo; # СНОВА НАЧИНАЕМ действия в теле цикла

    }

    print "\n", $random; # выводим число на консоль

    $i++; # и увеличиваем счетчик повторений

}
```

- Во всех рассмотренных циклах может присутствовать необязательный блок continue.
- В нем располагаются действия, которые нужно выполнить в конце цикла, в том числе при переходе к новой итерации по next.
- Действия в блоке continue не выполняются при переходах по last и redo.
- голый блок рассматривается в Perl как цикл, выполняющийся один раз. В таком блоке может присутствовать фраза continue и использоваться переходы last, next и redo.

С учетом предложений управления циклом и блока continue ***циклическую структуру в общем виде можно изобразить так***:

```
# может быть заголовок цикла: while, until или for

{

    # сюда происходит переход по redo

    # действие 1, выполняемое в цикле

    if (условие выхода) {

        last; # выход из цикла

    }
```

```
    # действие 2, выполняемое в цикле
    if (условие продолжения) {
        next; # выполнение следующей итерации цикла
    }
    # действие 3, выполняемое в цикле
    if (условие возобновления) {
        redo; # выполнение тела цикла сначала
    }
    # действие N, выполняемое в цикле
    # сюда происходит переход по next
} continue {
    # действие, выполняемое перед новой итерацией цикла
}
# сюда происходит переход по last
```

Циклы могут быть вложены один в другой.

- Когда требуется прервать вложенный цикл, перед ним ставится метка.
- Метка — это идентификатор, состоящий из латинских букв, знаков подчеркивания и цифр и начинающийся с буквы, после которого стоит знак двоеточия.
- Соблюдая хороший стиль программирования, следует записывать метки заглавными буквами.
- В операторе управления циклом метка указывает, выполнение какого цикла нужно прервать:

```
CYCLE_1:
while (условие продолжения цикла 1) {
    CYCLE_2:
    while (условие продолжения цикла 2) {
        if (условие выхода из всех циклов) {
            last CYCLE_1;
        }
        CYCLE3:
        while (условие продолжения цикла 3) {
            if (условия прерывания 2-го цикла) {
                next CYCLE_2;
```

```
            }
        }
    # сюда произойдет переход по next CYCLE_2
    }
}
# сюда произойдет переход по last CYCLE_1
```

- Метка может ставиться перед любым предложением.
- При помощи блока и операторов управления циклом с меткой можно имитировать управляющую структуру, подобную оператору switch в языке C.

Пример. Так можно записать только одно присваивание переменной $say в зависимости от условия:

```
SWITCH: {
    unless (defined $t) { # если $t не определено
        $t = 25; redo SWITCH; # задать значение по умолчанию
    }
    if ($t < 10) { $say = 'холодно'; last SWITCH; }
    if ($t < 18) { $say = 'прохладно'; last SWITCH; }
    if ($t < 27) { $say = 'тепло'; last SWITCH; }
    $say = 'жарко';
}
print "Сегодня $say\n";
```

Функция defined

1 defined EXPR

2 defined

- Функция возвращает булево значение, и отвечает на вопрос: содержит EXPR определенное значение или нет.
- Функция вернет false, если переменная еще не была инициализирована или ей было присвоено значение undef. True — если переменная была инициализирована любым значением, в том числе ей может быть присвоено значение числового нуля, пустой строки или строки содержащей символ нуля.

Именно этим defined отличается от стандартной булевой проверки, которая не может отличить undef от "0".

Функция undef

undef EXPR

undef

- Функция undef всегда возвращает неопределенное значение. Если передать функции в качестве аргумента переменную — она присвоит ей неопределенное значение.

```
1 undef $variable;

2 # или так

3 $variable = undef;
```

- не стоит использовать эту функцию для сравнения с переменными. Сравнение заданной переменной будет выполняться с числовым нулем или пустой строкой, а не с неопределенным значением. Результатом могут стать парадоксальные ситуации и ошибки.

Пример:

```
1  my $variable = 0;

2

3  if ($variable == undef) {

4        print "var is undef";

5  } else {

6        print "var is defined";

7  }

8

9  if (defined $variable) {

10       print "var is defined";

11 } else {

12       print "var is undef";

13 }
```

```
РЕЗУЛЬТАТ

var is undef

var is defined
```

МОДИФИКАТОРЫ

- Порядок выполнения действий в простом предложении можно задавать с помощью модификаторов выражений. За любым выражением может стоять один из следующих модификаторов:

```
выражение if выражение_ модификатора

выражение unless выражение_ модификатора

выражение while выражение_ модификатора
```

```
выражение until выражение_ модификатора

выражение foreach список
```

- Модификатор задает условие выполнения (в случае if или unless) или повторения (в случае while, until или foreach) выражения.
- Выражение модификатора вычисляется в первую очередь, хотя и стоит в конце конструкции.
- Хотя модификаторы похожи на условные и циклические управляющие конструкции, но они формируют простые предложения и поэтому не могут быть вложенными.

Пример. Приведенную выше конструкцию выбора можно переписать с использованием условных модификаторов:

```
SWITCH: {

      $t = -36, redo SWITCH unless defined $t;

      $say = 'холодно', last SWITCH if $t < 10;

      $say = 'прохладно', last SWITCH if $t < 18;

      $say = 'тепло', last SWITCH if $t < 27;

      $say = 'жарко';

}
```

- повторение действия с помощью циклических модификаторов, например:

```
++$count, --$sum while (rand < 0.1);

$random = rand until $random > 0.7;
```

- Применение модификаторов делает программу легче для понимания, поскольку акцент переносится на основное действие, стоящее в начале предложения.
- К тому же запись упрощается, так как не используется блок, а условное выражение в модификаторе можно не заключать в круглые скобки.

ВЫРАЖЕНИЯ С DO И EVAL

В программах на Perl можно встретить ключевое слово do с последующим блоком, что похоже на управляющую структуру.

Но конструкция do выступает в качестве терма в выражении. Иначе говоря, do делает из блока выражение, значением которого будет значение последнего предложения в блоке.

Пример, в такой операции присваивания:

```
$result = do { $x=rand; $a=0; }; # в $result будет присвоен 0

print "$result";
```

Чтобы подобное выражение стало простым предложением, после него нужно поставить "точку с запятой".

Вот так можно записать третий вариант конструкции выбора, где выражение do будет операндом условной операции, управляющей вычислением результата:

```
SWITCH: {
      (defined $t) || do { $t = 15; redo SWITCH; };
      ($t < 10) && do { $say = 'холодно'; last SWITCH; };
      ($t < 18) && do { $say = 'прохладно'; last SWITCH; };
      ($t < 27) && do { $say = 'тепло'; last SWITCH; };
      $say = 'жарко';
}
```

- Выражение do, как и любое другое выражение, может использоваться с модификаторами.

Например, с его помощью можно организовать циклическое выполнение действий:

```
do { $sum += rand; } until ($sum > 25);
```

- Но поскольку эта конструкция — выражение, а не цикл, то операторы управления циклом в ней не работают.

eval

- Иногда требуется динамически вычислить значение строкового выражения или выполнить блок предложений, изолируя возможные ошибки выполнения. Для этого используется конструкция eval, которая применяется в одной из двух форм:

```
eval выражение # вычислить выражение как код на Perl
eval блок # выполнить блок, перехватывая возможные ошибки
```

- В любой форме eval возвращает значение последнего вычисленного выражения.
- В первой форме строковое выражение рассматривается eval как исходный код на Perl, который во время работы программы динамически компилируется и выполняется.
- Если при его компиляции или выполнении возникает ошибка, eval возвращает неопределенное значение, но программа не заканчивается аварийно, а сообщение об ошибке помещается в специальную переменную $@.

Например: `eval выражение`

```
$x = 0; $y = 5; # в выражении будет деление на 0
$expression = "$y/$x"; # строка, содержащая код для выполнения
$result = eval ($expression); # вычислить выражение
if ($@ eq '') { # проверить специальную переменную на ошибки
      print "Выражение вычислено: $result";
} else {
      print "Ошибка вычисления: $@";
```

```
}
```

Во второй форме `eval блок`

- блок предложений в конструкции eval, как и в конструкции do, становится выражением.
- Он компилируется обычным образом и выполняется во время работы программы, но возможные ошибки его выполнения также не приводят к аварийному завершению программы.
- Причину ошибки можно узнать из специальной переменной $@, а значением eval будет значение последнего предложения в блоке.

Пример обработки ошибок в выражении eval:

```
$result = eval { # выполнить блок

     $x = 0; $y = 5;

     $z = $y / $x; # здесь будет деление на 0

}; # завершаем предложение точкой с запятой

unless ($@) { # проверить специальную переменную на ошибки

     print "Выражение вычислено: $result";

} else {

     print "Ошибка вычисления: $@";

}
```

ПРАГМЫ

— указания компилятору и исполняющей системе выполнить какие-либо действия или начать работать в определенном режиме.

- позволяют управлять поведением программы при компиляции и выполнении, их довольно много, и полное их описание можно найти в документации.
- Очень рекомендуется в начало любой программы включать прагмы, отвечающие за более тщательную проверку правил и ограничений:

```
use strict; # ограничить применение небезопасных конструкций

use warnings; # выводить подробные предупреждения компилятора

use diagnostics; # выводить подробную диагностику ошибок
```

- Дополнительная диагностика компилятора поможет избежать многих ошибок при выполнении программы.
- Обычно прагмы могут включаться в любом месте программы с помощью ключевого слова use и выключаться при необходимости с помощью ключевого слова no

Пример:

```
use integer; # включить целочисленные вычисления

print 10 / 3; # результат: 3
```

```
no integer; # отключить целочисленные вычисления

print 10 / 3; # результат: 3.33333333333333
```

- С помощью прагмы use constant можно определять в программе именованные константы, которые по традиции записываются заглавными буквами.

Это делается таким образом:

```
use constant PI => 3.141592653; # число пи
```

- С помощью прагмы use locale в программе включается действие национальных системных установок для некоторых встроенных функций, например, при работе со строками на русском языке:

```
use locale;
```

По ходу изучения материала следующих лекций будут рассмотрены другие полезные прагмы, а в лекции 13 будет описано применение use для подключения внешних модулей.

СИНОНИМЫ И ИДЕОМЫ

В Perl существует несколько синонимичных конструкций, предоставляющих автору программы возможность наиболее точно выразить свой замысел в привычном для него стиле.

Perl — демократичный язык, и каждый пишет на нем так, как ему удобнее и привычнее. За многие годы использования Perl в нем сложились устойчивые выражения (idioms, идиомы), подобные пословицам и поговоркам в естественных языках. Для примера можно привести некоторые из них:

```
# 1. Выполнять бесконечный цикл

for (;;) # читается "forever" - "всегда"

{ } # тело бесконечного цикла

# 2. Открыть файл или аварийно завершить программу

open FILE or die; # "открой файл или умри!"

# 3. Читать строки из входного потока и печатать их,

# используя буферную переменную по умолчанию

while (<>) { print; }

# 4. Присвоить переменной значение по умолчанию

# только, если ее значение не определено

$variable ||= $default_value;
```

- Общепринятые рекомендации по стилю оформления программ изложены в разделе стандартной документации, который можно просмотреть с помощью команды:

perldoc perlstyle

ОФОРМЛЕНИЕ ПРОГРАММЫ

В соответствии с устоявшимися традициями, типичная программа на языке Perl скорее всего будет выглядеть примерно так:

```
# вводные комментарии к программе

use strict; # включение дополнительной...

use warnings; # ... диагностики

# use Env; # подключение нужных модулей

# package main; # объявление пакета

my $message = 'Привет!'; # объявление переменных и

print $message; # запись алгоритма программы

# описание форматов отчета

# описание подпрограмм

__END__ # необязательное обозначение конца программы
```

ТЕКСТ, СТРОКИ И СИМВОЛЫ

строковые литералы, заключаемые в апострофы и двойные кавычки, могут записываться в альтернативной форме:

'строка в апострофах' или q(строка в апострофах)

"строка в кавычках" или qq(строка в кавычках)

Преобразующие escape-последовательности

В Perl есть особые управляющие последовательности, предназначенные для преобразования символов в строковом литерале. Они приведены в таблице. С их помощью преобразуется либо один символ, следующий за escape-последовательностью, либо несколько символов до отменяющей последовательности.

Таблица ***Преобразующие escape-последовательности***

Управляющая последовательность		Преобразование
\u	Upper	следующий символ к верхнему регистру
\l	Lower	следующий символ к нижнему регистру
\U	Upper	символы до \E к верхнему регистру
\L	Lower	символы до \E к нижнему регистру
\Q	Quote	отменить специальное значение символов вплоть до \E
\E	End	завершить действие \U или \L или \Q

Пример

use locale; # для правильной обработки кириллицы

$name = 'мария'; # будем преобразовывать значение переменной

print "\u$name\n"; # будет выведено: Мария

print "\U$name\E\n"; # будет выведено: МАРИЯ

Аналогичного результата можно достигнуть при использовании некоторых строковых функций.

ФУНКЦИИ РАБОТЫ СО СТРОКАМИ

- ***chomp***(), удаляющая в конце строки символ-разделитель записей; (он фактически удаляет любой символ, соответствующий текущему значению $/ (разделитель входных записей), в $/ по умолчанию используется \n.)
- ***chop***(), отсекающая любой последний символ строки;
- ***join***(), объединяющая элементы массива в одну строку;
- ***split***(), разделяющая строку на список подстрок.

Пример

while (my $text = <STDIN>) {
 	
chomp($text);
 	
print "You entered '$text'\n";
 	
last if ($text eq '');
 	
}
 	
	asd
	
	You entered 'asd'
	
	zxc
	
	You entered 'zxc'
	
	You entered ''

Поиск подстроки

- ***index***() выполняет поиск подстроки в строке, начиная с определенного смещения, и возвращает номер позиции найденной подстроки.
- ***rindex***() ищет подстроку от конца строки и возвращает позицию последней подстроки в строке перед указанным смещением. Смещение можно не указывать, тогда поиск производится во всей строке. Номера позиций подстроки и смещения начинаются с нуля. Если подстрока не найдена, возвращается -

```
index("Perl is great", "g");     # Returns 8
index("Perl is great", "great"); # Also returns 8

index("Perl is great", "e", 5);  # Returns 10
```

Пример

```
$pos = index($string, $sub_string, $offset); # с начала
$last_pos = rindex($string, $sub_string, $offset); # с конца
print "есть правда!" if(index($life, 'правда') != -1);
```

Определение длины текста

- ***length***() возвращает длину в символах значения строки или выражения, возвращающего строку или преобразованного к строке

```
$string ="text\n";
$string_length = length($string); # строка в переменной
print $string_length,"\n";
$n=12;
print length($n),"\n";
$n = $n+ 3 until(length($n)>2); # длина числа
print $n,"\n";

$s1 ="aaa";
$s2 = "zz";
$limit = 3;
print 'Текст слишком длинный' if length($s1 . $s2) > $limit;
```

Выделение подстроки из строки

substr() всегда была очень популярной в большинстве языков (кроме Perl, в котором это действие чаще выполняется с помощью регулярных выражений).

- копирует из строки подстроку заданной длины, начиная с указанного смещения. Если смещение отрицательное, то оно отсчитывается от конца строки. Если длина подстроки не задана, то копируется строка после смещения до самого конца:

Необычность функции substr() в Perl состоит в том, что она может применяться для изменения строки, относясь к группе так называемых левосторонних функций, которые

- могут употребляться в левой части операции присваивания.
- В этом случае значение, стоящее в правой части присваивания, заменяет подстроку, которая извлекается из строки функцией substr(), стоящей слева от знака присваивания.

Пример.

- Можно подстроку длиной в два символа, начинающуюся с символа с индексом 5, заменить новой строкой:

```
 $string = 'Perl 5 нравится программистам.';

$new_string = '6 тоже по';
substr($string, 5, 2) = $new_string;
# в $string будет: 'Perl 6 тоже понравится программистам.'
```

- Подобным же образом можно ***удалить последние 5 символов строки, заменив их пустой строкой***:

```
 substr($string, -5) = ''; # удалить последние 5 символов
```

Сочетая уже известные функции, можно выполнять разные манипуляции с текстовой информацией. ***Пример.***

Чтобы переставить слова в строке, можно воспользоваться функциями split(), reverse() и join() в списочном контексте:

```
$text ="abc def";

print "$text\n";

$reverse_words = join(' ', reverse(split(' ', $text)));
```

```
print "$reverse_words"; # def abc
```

Набор функций для преобразования букв

— набор функций для преобразования букв из заглавных в строчные и наоборот.

Для правильного преобразования русских букв нужно включить поддержку национальных установок операционной системы с помощью прагмы use locale.

- Преобразовать текст к нижнему регистру (lower case) можно с помощью функции ***lc(),*** которая возвращает значение текстового выражения, преобразованное к строчным буквам:

```
use locale; # учитывать национальные установки, под Windows

$lower_case = lc($text); # преобразовать к маленьким буквам
```

- ***lcfirst***() возвращает значение строкового выражения, в котором только первый символ преобразован к нижнему регистру

```
$first_char_lower = lcfirst($text); # 'Perl' станет 'perl'
```

- ***uc***() — возвращает значение символьного выражения, преобразованное к заглавным буквам

```
$upper_case = uc($text); # преобразовать к большим буквам
```

- ***ucfirst***() возвращает значение строкового выражения, в котором только первый символ преобразован к верхнему регистру

```
$capitalized = ucfirst($name); # 'ларри' станет 'Ларри'
```

- ***quotemeta***() находит в символьном выражении метасимволы или escape-последовательности и возвращает строку, где у всех специальных символов отменено их особое значение: для этого перед каждым из них ставится символ обратной косой черты '\'.

```
$string_with_meta = '\n \032 \x00 text \t \v "';

$quoted = quotemeta($string_with_meta);

# в $quoted будет '\\n\ \\032\ \\x00\ text\ \\t\ \\v\ \"'
```

- ***sprintf***() возвращает строку, которая сформирована в соответствии с правилами форматирования, заимствованными из языка C: на основе формата преобразования, заданного первым аргументом, в результирующую строку подставляются отформатированные значения из списка остальных аргументов функции.

В общем виде вызов этой функции выглядит так: sprintf(ФОРМАТ, СПИСОК АРГУМЕНТОВ). В формате преобразования располагается любой текст, в котором могут присутствовать указания преобразования. Каждое указание начинается с символа процента (%) и заканчивается символом, определяющим преобразование.

```
$format = "'%12s' агента <%03d> = '%+-10.2f'";

@list = ('Температура', 7, 36.6);

$formatted_string = sprintf($format, @list);

print "$formatted_string"; #  'Температура' агента <007> = '+36.60    '
```

Преобразования в формате этого примера обозначают следующее:

%12s — преобразовать аргумент в строку (string) и поместить в поле шириной в 12 символов с выравниванием вправо (т. к. ширина поля положительная);

%03d — преобразовать аргумент в десятичное целое (decimal) и поместить в поле шириной в 3 цифры с ведущими нулями (т. к. ширина поля задана с ведущим нулем) и выравниванием вправо (поскольку ширина положительная);

%+-10.2f — преобразовать аргумент в дробное число (float) с явным знаком (т.к. указан +) и поместить в поле шириной в 10 цифр, из которых 2 отводятся на дробную часть, с выравниванием влево (поскольку ширина поля отрицательная).

- округления чисел — например, до трех знаков в дробной части:

```
$rounded = sprintf("%.3f", 7/3); # в $rounded будет 2.333
```

Функции работы с символами

- ***substr***($string, $index, 1) можно скопировать Один символ из строки

- функций ***ord()*** и ***chr()*** выполняются преобразования символа (а точнее односимвольной строки) в его ASCII-код и наоборот:

```
$code = ord($char); # ord('M') вернет число 77

$char = chr($code); # chr(77) вернет строку 'M'

# синоним: $char = sprintf("%c", $code);
```

- Разбить строку на отдельные символы и поместить их в массив можно с помощью уже знакомой функции ***split***() с пустой строкой в качестве разделителя:

```
@array_of_char = split('', $string);
```

Пример

С помощью списков и нескольких вызовов функции substr() можно поменять в строке местами символы с указанными индексами, например, 1 и 11:

```
$s = 'кОт видел кИта';

(substr($s, 1, 1), substr($s, 11, 1)) = (substr($s, 11, 1), substr($s, 1, 1));

# в $s будет 'кИт видел кОта'
```

- Известная по лекции о списках функция ***reverse() в скалярном контексте*** возвращает значение текстового выражения, в котором символы переставлены в обратном порядке, например:

```
$palindrom = 'А РОЗА УПАЛА НА ЛАПУ АЗОРА';

$backwards = reverse($palindrom);

# в $backwards будет 'АРОЗА УПАЛ АН АЛАПУ АЗОР А'
```