1 Ссылки

Ссылка это переменная скалярного типа, которая хранит адрес в памяти, где располагается данное. Ссылки используются для создания сложных структур данных (например, двумерных массивов), а также для передачи в функции параметров любых типов (не только скаляров).

Листинг 1: Пример объявления ссылки

```
my $scalar = 'Скаляр';          # объект ссылки

my $ref2scalar = \$scalar; # ссылка на скаляр
```

Ссылка всегда указывает на значение конкретного типа: скаляр, массив, хэш, подпрограмму. На какой именно тип объекта указывает ссылка, можно узнать с помощью функции ref(), которая возвращает строку, описывающую тип значения объекта ссылки. Например:

Листинг 2: Тип значения ссылки

```
print ref($ref2scalar); # выведет: 'SCALAR'
```

А что получится, если вывести значение самой ссылки? Ее значение будет преобразовано в строку, содержащую тип объекта ссылки и его адрес в виде шестнадцатиричного числа, например:

Листинг 3: Значения ссылки

```
print $ref2scalar; # выведет, например: 'SCALAR(0x335b04)'
```

Чтобы получить доступ к значению, на которое указывает ссылка, нужно выполнить разыменование ссылки

Листинг 4: Разименование ссылки

```
print "${$ref2scalar} ";        # или: $$ref2scalar
```

Значением ссылки может быть другая ссылка

Листинг 5: Разименование ссылки

```
$value = 'Полезное значение';
$ref1 = \$value; # ссылка на значение

$ref2 = \$ref1;   # ссылка на ссылку на значение

# или

$ref_chain = \\$value;
```

Чтобы получить доступ к значению

Листинг 6: Разименование ссылки

```
print $$$ref_chain;
```

Подобным же образом можно работать со ссылками на массивы.

Листинг 7: Ссылка на массив

```
my @array = (1, 2, 3);
my $ref2array = \@array; # ссылка на массив

#или сразу написать
my $ref2array =\[(1, 2, 3)];
```

Чтобы получить доступ к массиву, нужно поставить перед ссылкой знак @

Листинг 8: Доступ к массиву по ссылке

```
print "@$ref2array\n";
```

Для доступа к элементу массива используется оператор

Листинг 9: Доступ к элементу массива

```
# доступ по ссылке к значению элемента массива:

my $element_value = $ref2array->[0];

# изменение значения элемента массива:
$ref2array->[0] = 3;
```

Можно получить ссылку на элемент массива

Листинг 10: Ссылка на элемент массива

```
$ref2element = \$array[0];      # ссылка на элемент массива
${$ref2element} = 3; # изменение элемента массива
```

В элементах массива можно хранить ссылки на другие массивы: это позволяет создавать в Perl многомерные массивы или "массивы массивов". В этом случае доступ к элементам многомерного массива также обычно записывается с использованием операции "стрелка":

Листинг 11: Многомерный массив

```
@{$ref2NxM->[$n]} # вложенный массив
$ref2NxM->[$n][$m] # скалярный элемент двумерного массива
$ref2NxMxP->[$n][$m][$p] # элемент 3-мерного массива
```

Листинг 12: Пример: Двумерный массив с таблицей умножения

```
my $table = []; # ссылка на анонимный массив массивов

for (my $row = 0; $row < 9; $row++) { # цикл по строкам

    $table->[$row] = []; # строка: вложенный массив
```

```
    for (my $col = 0; $col < 9; $col++) { # по колонкам

        $table->[$row][$col] = ($row+1)*($col+1);

    }
}

#Теперь напечатаем

# цикл по строкам (элементам массива верхнего уровня)
for (my $row = 0; $row < @\{$table\}; $row++) {
# цикл по столбцам (элементам вложенных массивов)
    for (my $col = 0; $col < @{$table->[$row]}; $col++) {

        print "$table->[$row][$col]\t";

    }

    print "\n";
}
```

Аналогично оформляются ссылки на хэши:

Листинг 13: Ссылки на хэши

```
my %hash = ('Хэш' => 'ассоциативный массив');
my $ref2hash = \%hash; # ссылка на весь хэш


#Или сразу написать:
my $ref2hash = \{('Хэш' => 'ассоциативный массив')};

#Разыменование ссылки:

my %hash2=%$ref2hash;
$ref2hash->{'ключ'} = 'значение'; # изменение значения
print $ref2hash->{'ключ'}; # доступ к значению элемента хэша
```

2 Подпрограммы

Предположим, нам нужно смоделировать искусственную последовательность и сайтами для тестирования. Внутри последовательности расположено несколько сайтов:

Листинг 14: Пример без подпрограмм

```
my @alphabet=('a','t','g','c');

my $pattern="atgcgcgatg";
```

```
my $seq="";
my $sslen= rand 200;
for (my $i=0; $i<$sslen; $i++){

my $ind=rand 4;

$seq.=$alphabet[$ind];

}

my $nsites= rand 10;
for (my $j=0; $j<$nsites; $j++){
        $seq.=uc $pattern;

        $sslen=rand200;
        for (my $i=0; $i<$sslen; $i++){

                my $ind=rand 4;
                $seq.=$alphabet[$ind];
    }

}
print $seq."\n";
```

У нас есть в программе два одинаковых блока, которые генерируют фрагмент случайной последовательности. Чтобы не дублировать код, а также чтобы логически выделить этот блок, можно написать подпрограмму, которая делает случайную последовательность. Тогда наша программа будет выглядеть так:

Листинг 15: Пример с подпрограммой

```
my $pattern="atgcgcgatg";

my $seq="";
$seq.=randSeq();
$nsites=rand 10;
for (my $j=0; $j<$nsites; $j++){

    $seq.=(uc $pattern).randSeq();

}
print $seq."\n";

sub randSeq{

   my @alphabet=('a','t','g','c');
   my $seq="";
   my $sslen=rand 200;
   for (my $i=0; $i<$sslen; $i++){
```

```
        my $ind=rand 4;
        $seq.=$alphabet[$ind];

    }

    return $seq;

    }
```

Читать такую программу гораздо проще. Итак, если в нашей программе есть некая последовательность действий, которая осуществляется несколько раз, ее можно выделить в отдельный блок и вызывать этот блок по имени в нужных нам местах программы. Такой блок называется подпрограммой.

Из соображений практического удобства, определения подпрограмм часто располагают в исходном тексте после основной программы, чтобы вначале ознакомиться с общей логикой программы. Возврат значений Как правило, подпрограмма возвращает какое-то значение. В предыдущем примере подпрограмма возвращала строку — случайную последовательность, которую мы добавляли к нашей последовательности. Встроенные функции, например rand, sin, log, также являются подпрограммами и возвращают числовое значение.

Листинг 16: Пример вызова стандартной функции

```
my $y=sin($x);
```

Также подпрограмма может возвращать и массив. Для того, чтобы закончить выполнение подпрограммы и вернуть значение используется ключевое слово return. Так делается в пользовательской функции приветствия:

Листинг 17: Возврат значения

```
sub greeting { # приветствие в зависимости от времени суток
     my $hours = (localtime)[2]; # текущие часы
     if ($hours >= 4 and $hours < 12) {
        return 'Доброе утро';
     } elsif ($hours >= 12 and $hours < 18) {
        return 'Добрый день';
     } elsif ($hours >= 18 and $hours < 22) {
        return 'Добрый вечер';
     } else {
        return 'Доброй ночи';
          }
}
print greeting(), '!';
```

3 Параметры подпрограммы

Часто бывает нужно передать подпрограмме какие-то параметры. Например,

Листинг 18: Передача параметров подпрограмме

```
my $y=cube($x);
```

Когда подпрограмма вызывается с набором аргументов, ей передается специальный массив с предопределенным именем `@_`, содержащий список переданных аргументов. В теле подпрограммы переданные значения доступны в виде элементов массива `@_`, что видно из следующего примера:

Листинг 19: Параметры в подпрограмме

```
sub cube { # вычислить куб числа
    return $_[0] * $_[0] * $_[0]; # умножить аргумент
}
print cube(2); # будет напечатано 8
```

Количество переданных аргументов можно выяснить, запросив число элементов массива `@_`. Для обработки списка аргументов переменной длины часто используется встроенная функция shift(), которая извлекает из массива параметров очередное значение, переданное подпрограмме:

Листинг 20: Использование функции shift для извлечения параметров

```
                    print2files($message, $file1, $file2, $file3);
sub print2files { # вывести текст в несколько файлов
    my $text = shift; # 1-й параметр - текст
    while (@_) {
        my $file = shift; # очередное имя файла

                open FILE, ">>$file" or die;
                print FILE, $text;
                close FILE or die;
    }
}
```

Если переданные аргументы заданы переменными, то изменение элементов массива приведет к изменению значений соответствующих переменных в вызывающей программе. Это можно проиллюстрировать следующим (несколько искусственным) примером:

Листинг 21: Изменение значения параметра в подпрограмме

```
sub sum2 { # вычислить сумму 2-х чисел
    $_[0] = $_[1] + $_[2]; # поместить сумму в 1-й аргумент
    return;
}

my $a = 1, $b = 2, $sum = 0;

sum2($sum, $a, $b);

print "$a+$b=$sum"; # будет напечатано: 1+2=3
```

Опыт показывает, что изменение значения аргументов ведет к трудно обнаруживаемым ошибкам и осложняет сопровождение программы, поэтому должно использоваться

в исключительных случаях и всегда оговариваться в комментариях. Общепринятым способом работы с параметрами подпрограммы является присваивание значения аргументов списку переменных: это предохраняет аргументы от изменения и позволяет работать не с элементами массива, а с удобно названными переменными. Это видно из следующего примера:

Листинг 22: Копирование значений параметров в переменные

```
sub get_file { # считать данные из файла

    my ($path, $file) = @_; # присвоить аргументы в переменные

        open FILE, '<', "$path/$file" or die("Error: no file $path/$file");

    my @lines = <FILE>; # прочитать все строки файла в массив

    close FILE or die("Error: can't close file $path/$file");

    return @lines; # вернуть массив строк файла

}

my @data = get_file('/tmp', 'log.txt');
```

4 Вызов подпрограмм

Обращение к подпрограмме обозначается использованием круглых скобок после имени подпрограммы, даже если она вызывается без параметров. Как в этих примерах:

Листинг 23: Пример вызова функции

```
format_c(); # вызов подпрограммы без параметров
format_text($text, $font, $size); # и с параметрами
```

Бывает, функция вызывается внутри себя самой. Такая функция называется рекусивной. Например:

Листинг 24: Пример рекурсивной функции

```
sub factorial ($) { # вычислить N!
    my $n = shift;
    return ($n <= 1) ? 1 : $n * factorial($n-1);
}
```

5 Подпрограммы и ссылки

Массив аргументов всего один. Если, например, в функцию передать два массива, то они объединятся в один массив. Это неудобно (непонятно, где кончается первый массив и начинается второй). Если передать функции

хэш, то он преобразуется в массив, у которого четные элементы — ключи, а нечетные — значения. Это тоже неудобно. Но, поскольку ссылка является скаляром, в любой массив можно поместить список ссылок на другие программные объекты, например, таким образом:

Листинг 25: Массив из ссылок

```
@reference_list = (\$scalar, \@array, \%hash);

#или

@reference_list = \($scalar, @array, %hash);
```

И вот теперь-то мы сможем передавать в функцию все что угодно!!!! Например, мы можем передавать в функцию массив и хэш. Для этого нужно создать ссылки на них и передать их функции. Ссылки являются скалярами и помещаются в массив `@_`, откуда их можно потом извлечь и разыменовать, получив таким образом доступ к значениям. Напишем функцию, которая удваивает значения в массиве и в хэше

Листинг 26: Пример использования ссылок для передачи значений в подпрограмму

```
sub doublparms {

    $listref =$_[0];
    $hashref = $_[1];

    foreach $item (@$listref) {

          $item *= 2;
    }

    foreach $key (keys %$hashref) {

                $$hashref{$key} *= 2;

    }

}

@somelist=(1,2,3) ;

%somehash=("one"=>5, "two"=>15, "three"=>20);

doublparms(\@somelist,\%somehash);

print "итоговые значения:\n\@somelist=@somelist\n";

foreach $key (keys %somehash) {
```

```
    print "\$somehash{". $key ."}=". $somehash{$key}." ";

}

print "\n";
```

6 Области видимости переменных

По умолчанию переменная видна во всей программе и можно в любом месте программы изменить ее значение. Это не очень хорошо и может привести к трудно отслеживаемым ошибкам. Чтобы ограничить видимость переменных рамками блока или подпрограммы, нужно объявить для них лексическую область видимости с помощью функции my(), как это уже делалось в приводимых ранее примерах.

Чтобы проследить, как изменяются значения переменных, объявленных в главной программе и подпрограммах, внимательно прочитайте следующий пример (скучный, но полезный для понимания):

Листинг 27: Область видимости переменных

```
use strict;
my $var = 'm';                  # лексическая var в main
print "1(main)='$var'\n"; # выведет: 1(main)='m'
sub1();
print "7(main):'$var'\n"; # выведет: 7(main):'z'

sub sub1 {
    print "2(sub1)='$var'";
    $var = 's';                 # изменяется var из main!
        print "-->'$var'\n";        # выведет: 2(sub1)='m'->'s'
    my $var = '1';              # изменена var1 из sub1
    print "3(sub1)='$var'\n"; # выведет: 3(sub1)='1'
    sub2();
                                # снова видима var1 из sub1
        print "6(sub1):'$var'\n"; # выведет: 6(sub1):'1'
}
sub sub2 { # снова видима var из main
    print "4(sub2):'$var'";
    $var = 'z';                 # изменяется var из main!!
    print "-->'$var'\n";        # выведет: 4(sub2):'s'->'z'
        my $var = '2';              # изменена var1 из sub2
        print "5(sub2)='$var'\n"; # выведет: 5(sub2)='2'
}
```

Обратите внимание, что лексическая переменная *var*, , *sub*1*sub*2, .*sub*2var, объявленная в процедуре sub1. Из приведенного примера видно, что после объявления в подпрограмме лексических переменных с помощью my(), изменения этих переменных не затрагивают других переменных с теми же именами. Поэтому, чтобы избежать нежелательного изменения значений переменных в других частях программы, рекомендуется всегда объявлять для переменных

лексическую область видимости. Проконтролировать наличие объявлений для переменных в программе поможет прагма use strict.

7 Домашнее задание

- (1 балл) Написать подпрограмму, которая возвращает исходный массив с добавленной в конец суммой двух последних членов массива. Вызвать подпрограмму 10 раз и вывести массив на экран. Получится массив первых 12-ти чисел Фибоначи 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144.
- (2 балл) Пользователь вводит положительное число не больше 100. Написать программу которая раскладывает заданное число на простые множители. Программа содержит подпрограмму, которая на вход получает число и массив, в который дописывает наименьший множитель, на который делится данное число. Подпрограмма вызывает себя рекурсивно до тех пор пока переданное ей число не будет простым.