Using @ARGV

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script24.pl
# do the equivalent of a shell's echo:
use strict;
my $a;
while($a = shift @ARGV)
{
  print "$a ";
}
print "\n";
```

Using @ARGV

```
#!/usr/bin/perl -w
# 2005 09 25 - rdl Script25.pl
# count the number of arguments
use strict;
my $count = 0;
map { $count++ } @ARGV;
print "$count\n";
```

Loop control operators

Perl has three interesting operators to affect looping: `next`, `last`, and `redo`.

- `next` → start the next iteration of a loop immediately
- `last` → terminate the loop immediately
- `redo` → restart this iteration (very rare in practice)

The `next` operator

The `next` operator starts the next iteration of a loop immediately, much as `continue` does in C.

The next operator

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script26.pl
# sum the positive elements of an array to demonstrate next
use strict;
my $sum = 0;
my @arr1 = -10..10;
foreach(@arr1)
{
    if($_ < 0)
    {
        next;
    }
    $sum += $_;
}
print $sum;
```

The `last` operator

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script27.pl
# read up to 100 items, print their sum
use strict;
my $sum = 0;
my $count = 0;
while(<STDIN>)
{
    $sum += $_;
    $count++;
    if($count == 100)
    {
        last;
    }
}
print "\$count == $count, \$sum == $sum \n";
```

The `redo` operator

The rarely used `redo` operator goes back to the beginning a loop block, but it does not do any retest of boolean conditions, it does not execute any increment-type code, and it does not change any positions within arrays or lists.

The redo operator

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script29.pl
# demonstrate the redo operator
use strict;
my @strings = qw/ apple plum pear peach strawberry /;
my $answer;
foreach(@strings)
{
    print "Do you wish to print '$_'? ";
    chomp($answer = uc(<>));
    if($answer eq "YES")
    {
        print "PRINTING $_ ...\n";
        next;
    }
    if($answer ne "NO")
    {
        print "I don't understand your answer '$answer'! Please use ei
        redo;
    }
}
```

The `reverse` function

If used to return a list, then it reverses the input list.
If used to return a scalar, then it first concatenates the elements of the input list and then reverses all of the characters in that string.
Also, you can `reverse` a hash, by which the returned hash has the keys and values swapped from the original hash. (Duplicate `value` → `key` in the original hash are chosen randomly for the new `key` → `value`.)

Examples of `reverse`

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script30.pl
# demonstrate the reverse function
use strict;
my @strings = qw/ apple plum pear peach strawberry /;
print "\@strings = @strings\n";
my @reverse_list = reverse(@strings);
my $reverse_string = reverse(@strings);
print "\@reverse_list = @reverse_list\n";
print "\$reverse_string = $reverse_string\n";
```

Example of `reverse` for hash

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script31.pl
# demonstrate the reverse operator
use strict;
my %strings = ( 'a-key' , 'a-value', 'b-key', 'b-value', 'c-key', 'c-v
print "\%strings = ";
map {print " ( \$key = $_ , \$value = $strings{$_} ) "} (sort keys %st
print " \n";
my %reverse_hash = reverse(%strings);
print "\%reverse_hash = ";
map {print " ( \$key = $_ , \$value = $reverse_hash{$_} ) "} (sort key
print " \n ";
```

Example of `reverse` for hash with duplicate

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script33.pl
# demonstrate the reverse operator for hash with duplicate values
use strict;
my %strings = ( 'a-key' , 'x-value', 'b-key', 'x-value', 'c-key', 'x-v
print "\%strings = ";
map {print " ( \$key = $_ , \$value = $strings{$_} ) "} (sort keys %st
print " \n";
my %reverse_hash = reverse(%strings);
print "\%reverse_hash = ";
map {print " ( \$key = $_ , \$value = $reverse_hash{$_} ) "} (sort key
print " \n ";
```

Examples of `reverse`

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script32.pl
# demonstrate the reverse operator
use strict;
my $test = reverse(qw/ 10 11 12 /);
print "\$test = $test\n";
```

The `sort` function

The `sort` function is only defined to work on lists, and will only return sensible items in a list context. **By default,** `sort` **sorts lexically.**

The sort function

```
# Example of lexical sorting
@list = 1..100;
@list = sort @list;
print "@list ";
1 10 100 11 12 13 14 15 16 17 18 19 2 20 21 22
23 24 25 26 27 28 29 3 30 31 32 33 34 35 36 37
38 39 4 40 41 42 43 44 45 46 47 48 49 5 50 51
52 53 54 55 56 57 58 59 6 60 61 62 63 64 65 66
67 68 69 7 70 71 72 73 74 75 76 77 78 79 8 80
81 82 83 84 85 86 87 88 89 9 90 91 92 93 94 95
96 97 98 99
```

The sort function

You can define an arbitrary sort function. Our earlier mention of the <=> operator comes in handy now:

```
# Example of numerical sorting
@list = 1..100;
@list = sort { $a <=> $b } @list;
print "@list ";
@list = 1..100;
@list = sort { $a <=> $b } @list;
print "@list";
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
64 65 66 67 68 69 70 71 72 73 74 75 76 77 78
79 80 81 82 83 84 85 86 87 88 89 90 91 92 93
94 95 96 97 98 99 100
```

The `sort` function

The `$a` and `$b` in the function block are actually package global variables, and should not be declared by you as `my` variables.

The sort function

You can also use the cmp operator quite effectively in these type of anonymous sort functions:

```
@words = qw/ apples Pears bananas Strawberries cantaloupe grapes Blueb
@words_alpha = sort @words;
@words_noncase = sort { uc($a) cmp uc($b) } @words;
print "\@words_alpha = @words_alpha\n";
print "\@words_noncase = @words_noncase\n";
# yields:
@words_alpha = Blueberries Pears Strawberries apples bananas cantaloup
@words_noncase = apples bananas Blueberries cantaloupe grapes Pears St
```

Hashes

We have already used a few examples of hashes. Let’s go over exactly what is happening with them:

- A hash is similar to an array in that it has an index and in that it may take an arbitrary number of elements.
- An index for a hash is a string, not a number as in an array.
- Hashes are also known as “associative arrays.”
- The elements of a hash have no particular order.
- A hash contains key-value pairs; the keys will be unique, and the values are not necessarily so.

Hash declarations

- Hashes are identified by the % character.
- The name space for hashes is separate from that of scalar variables and arrays.

Hash element access

- One uses the syntax `$hash{$key}` to access the value associated with key `$key` in hash `%hash`.
- Perl expects to see a string as the key, and will silently convert scalars to a string, and will convert arrays silently.

Examples

```
$names[12101] = 'James';
$names[12101] = 'Bob';      # overwrites value 'James'
$name = $names[12101];      # retrieve value 'Bob';
$name = $names[11111];      # undefined value returns undef

%hash = ('1', '1-value', 'a', 'a-value', 'b', 'b-value');
@array = ('a');
print $hash{@array};
# yields
1-value
```

Examples

```
%names = (1, 'Bob', 2, 'James');
foreach(sort(keys(%names)))
{
  print "$_ --> $names{$_}\n";
}
# yields
1 --> Bob
2 --> James

map { print "$_ --> $names{$_}\n"; } sort(keys(%names));
# yields
1 --> Bob
2 --> James
```

Referring to a hash as a whole

As might have been gleaned from before, you can use the % character to refer a hash as a whole.

```
%new_hash = %old_hash;
%fruit_colors = ( 'apple' , 'red' , 'banana' , 'yellow' );
%fruit_colors = ( 'apple' => 'red' , 'banana' => 'yellow' );

print "%fruit_colors\n";       # only prints '%fruit_colors', not keys
@fruit_colors = %fruit_colors;
print "@fruit_colors\n";       # now you get output...
# yields
banana yellow apple red
```

The `keys` and `values` functions

You can extract just the hash keys into an array with the `keys` function.

You can extract just the hash values into an array with the `values` function.

```
%fruit_colors = ( 'apple' => 'red' , 'banana' => 'yellow' );
@keys = keys(%fruit_colors);
@values = values(%fruit_colors);
print "\@keys = '@keys' , \@values = '@values'\n";
# yields
@keys = 'banana apple' , @values = 'yellow red'
```

The `each` function

Perl has a “stateful” function `each` that allows you to iterate through the keys or the key-value pairs of a hash.

```
%fruit_colors = ( 'apple' => 'red' , 'banana' => 'yellow' );
while( ($key, $value) = each(%fruit_colors) )
{
  print "$key --> $value\n";
}
```

The `each` function

Note: if you need to reset the iterator referred to by `each`, you can just make a call to either `keys(%fruit_colors)` or `values(%fruit_colors)` – so don't do that accidentally!

```
%fruit_colors = ( 'apple' => 'red' , 'banana' => 'yellow' );
while( ($key, $value) = each(%fruit_colors) )
{
  print "$key --> $value\n";
  # ...
  @k = keys(%fruit_colors);   # resets iterator!!!
}
# yields loop!
banana --> yellow
banana --> yellow
banana --> yellow
banana --> yellow
banana --> yellow
 ....
```

The exists function

You can check if a key exists in hash with the `exists` function:

```
if(exists($hash{'SOMEVALUE'})
{
}
```

The delete function

You can remove a key-value pair from a hash with `delete`:

```
delete($hash{'SOMEVALUE'});
```