Scalar values “typecast” to boolean values

Many of Perl’s control structures look for a boolean value. Perl doesn’t have an explicit “boolean” type, so instead we use the following “typecasting” rules for scalar values:

☞ If a scalar is a number, then 0 is treated as false, and any other value is treated as true.

☞ If a scalar is a string, then “0” and the empty string are treated as false, and any other non-empty string is treated as true.
are treated as false, and any other value as true.

☞ If a scalar is not defined, it is treated as false.
If elsif else

Note that both elsif and else are optional, but curly brackets are never optional, even if the block contains more than one statement.

if(COND)
  {
  }
[elsif
  {
  }]*
[else
  {
  }]
if-elsif-else examples

if example:

if($answer == 12)
{
    print "Right -- one year has twelve months!\n";
}
if-elsif-else examples

if/else example:

if($answer == 12)
{
    print "Right -- one year has twelve months!\n";
}
else
{
    print "No, one year has twelve months!\n";
}
if-elsif-else examples

if-elsif-else example:

if($answer < 12)
{
    print "Need more months!\n";
}
elsif($answer > 12)
{
    print "Too many months!\n";
}
else
{
    print "Right -- one year has twelve months!\n";
}
if-elsif-else examples

if-elsif-elsif example:

if($a eq "struct")
{
}
elsif($a eq "const")
{
}
elsif($a ne "virtual")
{
}
You can test to see if a variable has a defined value with `defined()`:

```perl
if(!defined($a))
{
    print "Use of undefined value is not wise!";
}
```
The **while** construction

```perl
while(<boolean>)
{
    <statement list>
}
```

As with **if-elsif-else**, the curly brackets are not optional.
while examples

```perl
while(<STDIN>)
{
    print;
}

[You might note that we are using the implicit variable 
$_ in this code fragment.]
```
until control structure

```perl
until(<boolean>)
{
    <statement list>
}
```

The `until` construction is the opposite of the `while` construction since it executes the `<statement list>` until the `<boolean>` test becomes true.
until example

#!/usr/bin/perl -w
# 2006 09 20 -- rdl script22.pl
use strict;
my $line;
until(! ($line=<STDIN>))
{
    print $line;
}
for control structure

for(<init>; <boolean test>; <increment>)
{
   <statement list>
}

Very similar to the C construction. The curly brackets again are not optional.
for example

for($i = 0; $i<10; $i++)
{
    print "\$i * \$i = " . $i*$i . "\n";
}
Lists and Arrays

☞ A list in Perl is an ordered collection of scalars.

☞ An array in Perl is a variable that contains an ordered collection of scalars.
List literals

☞ Can represent a list of scalar values

☞ General form:

( <scalar1>, <scalar2>, ... )
List literals

Examples:

(0, 1, 5)  # a list of three scalars that are numbers
('abc', 'def')  # a list of two scalars that are strings
(1, 'abc', 3)  # can mix values
($a, $b)  # can have values determined at runtime
()  # empty list
Using `qw` syntax

You can also use the “quoted words” syntax to specify list literals:

('apples', 'oranges', 'bananas')
quw/ apples oranges bananas /
quw! apples oranges bananas !
quw( apples oranges bananas )
quw< apples oranges bananas >
List literals, cont’d

☞ You can use the range operator “..” to create list elements.

☞ Examples:

(0..5)    #
(0.1 .. 5.1) # same since truncated (not \tt floor()!)
(5..0)    # evals to empty list
(1,0..5,’x’ x 10) # can use with other types...
($m..$n)  # can use runtime limits
Arrays are declared with the “@” character.

my @a;
my @a = ('a', 'b', 'c');

Notice that you don’t have to declare an array’s size.
Arrays and scalars

Arrays and scalars are in separate name spaces, so you can have two different variables $a$ and @a.

Mnemonically, “$” does look like “S”, and “a” does resemble “@”.