

## 18. Perl: Control Statements

### Control Expressions

A control expression must evaluate to a value that can be interpreted as true or false.

How a control statement behaves depends on the value of its control expression.

Perl has no Boolean type.

A simple control expression is either an arithmetic expression or a string expression.

A string value counts as true unless it's the empty string (“”) or the zero string (“0”).

But “0.0” counts as true.

A numeric value counts as true unless it's 0.

An undefined value counts as false (since, as a number, it counts as 0 and, as a string, it counts as “”).

## Relational Expressions

A relational operator (such as `<`) produces 1 for true and the empty string for false.

There are two sets of relational operators, one for numeric operands, one for string operands.

These come in corresponding pairs.

The order relation on strings is lexicographical order over the collating sequence,

i.e., alphabetical order if we extend the “alphabet” to include all ASCII characters, ordered by the numerical order of their codes.

For example, `gt` is the string relational operator corresponding to `>`.

`"dog" gt "cat"` is true since “dog” follows “cat” in alphabetical order.

`"Dog" gt "cat"` is false since all ASCII codes for uppercase letters are less than any ASCII codes for lowercase letters.

The following table lists all the relational operators.

<b>Numeric</b>	<b>String</b>
<code>==</code>	<code>eq</code>
<code>!=</code>	<code>ne</code>
<code>&lt;</code>	<code>lt</code>
<code>&gt;</code>	<code>gt</code>
<code>&lt;=</code>	<code>le</code>
<code>&gt;=</code>	<code>ge</code>
<code>&lt;=&gt;</code>	<code>cmp</code>

All the numeric relational operators except `<=>` are like those in C with the same name.

```

$i <=> $j returns
    -1    if $i < $j
     0    if $i equals $j
    +1    if $i > $j

```

So `cmp` is similar to `strcmp()` in the C/C++ `string.h` library.

A relational operator coerces its operands to the proper type.

For example,

```

"cat" gt 7

```

counts as true.

Since `gt` is a string operator, `7` is coerced to `"7"`.

And `'7'` is greater than `'c'` in the ASCII sequence.

The following is false since `'2'` comes before `'3'`:

```

"23" gt "3"

```

But the following is true since `>` coerces its operands to numbers:

```

"23" > "3"

```

Strings known to be numbers can always be compared with the numeric operators to get the expected results.

But don't use numeric operators between strings that may not be numbers.

For example, the following is true since `"cat"` and `"dog"` are both converted to `0`.

```

"cat" == "dog"

```

The precedence of the relational operators is lower than that of the arithmetic operators.

So the following is OK:

```

$x + $y < $u - $v

```

Since the value of a relational expression is a scalar, it can be stored in a scalar variable – e.g.,

```

$flag = $count > $limit;

```

## Compound Expressions

Perl has two sets of Boolean operators, with the same meaning but different precedence levels.

One set is borrowed from C/C++: `&&`, `||`, `!`.

The other set is `and`, `or`, `not`.

These have lower precedence than any other Perl operators – they’re always evaluated last.

The value of a Boolean expression formed with `and`, `or`, `&&`, or `||` is the last value evaluated (as in C/C++) rather than 1 or “”.

The following table shows the precedence and associativity of all operators discussed so far.

The operators are listed from highest precedence (top) to lowest.

Operator	Associativity
<code>++</code> <code>--</code>	nonassociative
<code>**</code>	right
<code>!</code> unary <code>+</code> <code>and</code> <code>-</code>	right
<code>*</code> <code>/</code> <code>%</code> <code>.</code>	left
<code>&gt;</code> <code>&lt;</code> <code>&gt;=</code> <code>&lt;=</code> <code>lt</code> <code>gt</code> <code>le</code> <code>ge</code>	nonassociative
<code>==</code> <code>!=</code> <code>eq</code> <code>ne</code>	nonassociative
<code>&amp;&amp;</code>	left
<code>  </code>	left
<code>=</code> <code>+=</code> etc.	right
<code>not</code>	right
<code>and</code>	left
<code>or</code>	left

## Selection Statements: `if` and `unless`

Perl's `if` is like the `if` of C/C++.

But the *then* and *else* clauses are always blocks, enclosed in `{...}`, even when there is only one statement.

The last statement in a block need not end with a semicolon.

But it's good style to end every statement with a semicolon.

When logic might lead us to consider a conditional with only an *else* clause, we can use Perl's `unless` statement.

Given

```
unless ( control_expression )  
    block
```

the statements in *block* are executed when *control\_expression* is false.

*Example*

```
unless ( $count <= $limit ) {  
    print "Limit exceeded\n";  
}
```

In C/C++, we typically nest `if ... else` statements to implement a multi-branch conditional.

Perl has the special word `elsif` (not `else if` or even `elseif`) for this.

*Example*

```
if ( $age < 18 ) {
    print "A minor\n";
}
elsif ( $age < 35 ) {
    print "A young adult";
}
elsif ( $age < 65 ) {
    print "Middle aged";
}
else {
    print "Old";
}
```

Perl has no `switch` statement, but we'll show how to build a similar construct.

Perl, like C/C++, has conditional expressions.

*Example*

```
$x = ( $y < $z ) ? $y : $z;
```

sets `$x` to the minimum of `$y` and `$z`.

## Iterative Statements: **while**, **until**, and **for**

Perl's `while` statement is like that of C/C++.

Perl also has an `until` statement, which repeats *until* the control expression becomes true.

### *Example*

```
$sum = 0;
until ( $sum > 1000 ) {
    $sum += <STDIN>;
}
```

inputs numbers from the keyboard until their sum exceeds 1000.

The loop is equivalent to

```
$sum = 0;
while ( $sum <= 1000 ) {
    $sum += <STDIN>;
}
```

Perl's `for` statement is like that of C/C++;

Perl's `foreach` statement is like JavaScript's `for/in` statement.

We'll discuss it when we come to arrays.

## Exiting a Block: `next`, `last`, and `redo`

The `next` operator is like the `continue` of C/C++.

When `next` is executed in a block, control is transferred to the end of the block.

In a loop, this causes control to go back to the control expression.

The `last` operator is like the `break` of C/C++.

When `last` is executed in a loop body, the loop immediately terminates.

When the `redo` operator is executed in a loop body, control returns to the top of the loop body – not to the control statement.

So we redo the same iteration of the loop.

### *Example*

The following sums the even numbers in a sequence of numbers input from the keyboard until a negative number is entered.

```
$sum = 0;

while ( $num = <STDIN> ) {
    if ( $num < 0 ) {
        last;
    }

    if ( $num % 2 == 1 ) {
        next;
    }

    $sum += $num;
}
```

If a loop is nested within another loop, `next`, `last`, and `redo` as described so far affect only the loop in which they appear, not any enclosing loop.

A *label* of a statement is an *identifier*, which is followed by a colon, as in

```
LOOP:
    while ( ... ) {
        ...
    }
```

If

- a loop has a label and
- that label is the operand of a `next`, `last`, or `redo` operator appearing in a nested loop,

then the operator affects the labeled loop, not the smallest loop in which the operator appears.

### *Example*

The following finds a sum of the form  $\sum_{i=1}^n i \sum_{j=1}^m j$  but terminates if the partial sum exceeds some limit.

```
$sum = 0;

OUTER:
    for ( $i = 1; $i <= $n; $i++ ) {
        for ( $j = 1; $j <= $m; $j++ ) {
            $sum += $i * $j;

            if ( $sum > $limit ) {
                last OUTER;
            }
        }
    }
}
```

A *bare block* is a block that is not part of a loop construct.

Bare blocks can be labeled and can include `last`, `next`, and `redo` operators.

The following is an example of how to simulate a `switch` statement.

```
SWITCH: {
  if ( $num == 0 ) {
    $zeros++;
    last SWITCH;
  }
  if ( $num == 1 ) {
    $ones++;
    last SWITCH;
  }
  if ( $num == 2 ) {
    $twos++;
    last SWITCH;
  }
  $others++;
}
```

## Statement Modifiers

We can select or repeat the execution of a single statement by appending a modifier to it.

There is a modifier for each control-statement keyword except `for`.

The form is

*statement keyword control\_expression*  
where *control\_expression* is not enclosed in (...).

For example,

```
$ones++ if $num == 1;
```

The control expression is evaluated first even though it comes last.

Another example:

```
$sum += $i while $sum < 10;
```

A *do block* is a bare block preceded by `do` – e.g.,

```
do {
    $sum += $i++;
    print "\$i is: $i\n";
}
```

In this form, `do` does nothing.

But a `do` block can be followed by a modifier – e.g.,

```
$sum = $i = 0;

do {
    $sum += $i++;
    print "\$i is: $i\n";
} while $i < 10;
```

When a `while` or `until` is the modifier of a `do` block, the construct is a posttest loop.

The control expression is evaluated after the block is executed.

## Terminating Execution

The `die` function lets us terminate program execution after printing a message.

`die` takes a list of parameters (typically one).

When called, it

- concatenates its parameters together,
- prints the result on `STDERR`, and
- terminates program execution.

The Perl system usually adds information to what is printed:

- the program name,
- the line number of `die`, and
- a newline.

If a `\n` is included in the parameters, this information is not provided.

`die` is often called when a system function fails.

The implicit variable `$!` contains the number of the system error.

So `$!` is often included and `\n` left out of the string parameter to `die` – e.g.,

```
die "Input/Output error $!";
```

The `exit` function is like `die` but takes a numeric parameter, the error number.

0 means normal termination:

```
exit 0;
```

## More on Input

### Line Input Operators in Control Expressions

When the line input operator is the whole control expression of

- a `while` statement,
- a `while` statement modifier, or
- a `for` statement,

the value input is assigned to the implicit variable `$_`.

`$_` is the default parameter of several operators and functions, such as `print` and `chomp`.

This use of `$_` is a convenience.

`$_` should be used only as a temporary variable, for only a few lines of code.

#### *Example*

The following counts the number of times “Bob” occurs in the input.

We don’t have to initialize `$count` since `undef` counts as 0.

When the end of file is reached, `<STDIN>` returns `undef`, which also counts as false.

```
while ( <STDIN> ) {
    print;
    chomp;
    if ( $_ eq "Bob" )
        $count++;
}
```

## Command-Line Arguments

A command-line argument is a filename that appears after the rest of the command that executes a program – e.g.,

```
C:\someDirectory> perl prog.pl values.dat
```

A command-line argument can specify a file for either input or output.

In Perl, a command-line argument is implicitly redirected to a special filehandle that's the default for the input operator.

So `<>` reads lines from the file specified on the command line.

### *Example*

The following reads and prints lines from the file specified on the command line.

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

### *Example (from Sebesta, A Little Book on Perl, pp. 49-50)*

The following program reads a list of numbers from the file specified on the command line.

It finds and prints the second smallest number along with its position in the list.

```
# Get the first two numbers and initialize the
# smallest, its position, the second smallest,
# and its position.
# If there are fewer than two numbers, print a
# message and die.

if ( $first = <> ) {
  if ( $second = <> ) {
    if ( $first < $second ) {
      $min = $first;
      $pos_min = 1;
      $second_min = $second;
      $pos_2nd_min = 2;
    }
    else {
      $min = $second;
      $pos_min = 2;
      $second_min = $first;
      $pos_2nd_min = 1;
    }
  }
  else {
    die "The file has only one line!";
  }
}
else {
  die "The file is empty!";
}

$position = 2;
```

Continued next page

## Continued from previous page

```
# Loop to read and process the rest of the file

while ( $next = <> ) {
    $position++;

    # If the new number is the new smallest, replace
    # both the smallest and the second smallest

    if ( $next < $min ) {
        $second_min = $min;
        $min = $next;
        $pos_2nd_min = $pos_min;
        $pos_min = $position;
    }

    # If the new number is between the smallest and
    # the second smallest, replace the second
    # smallest with the new number

    elsif ( $next > $min and $next < $second_min ) {
        $second_min = $next;
        $pos_2nd_min = $position;
    }
}

print "The second smallest number is: ",
      "$second_min\n";

print "The position of the second smallest ",
      "number is: $pos_2nd_min\n";
```

## The Debugger

When the `-d` switch is used with the `perl` command, the program runs under the control of the debugger.

For example, if the name of the last example is `ch2.pl`, the data file is `ch2.dat`, and we use the `-d` switch, we get

```
C:\someFolder> perl -d ch2.pl ch2.dat
Default die handler restored.
```

```
Loading DB routines from perl15db.pl version 1.07
Editor support available.
```

```
Enter h or 'h h' for help, or 'perldoc perldebug' for more help.
```

```
main::(ch2.pl:8):          if ( $first = <> ) {
DB<1>
```

The cursor is sitting at the end of the last line, showing the debugger prompt.

The number in `<...>` increases by one after each debugger command.

The command `h` displays the help file.

Typing `h` followed by the name of a command gives information on that command.

The `s` command executes the current statement:

```
DB<1> s
```

For our example, this produces

```
main::(ch2.pl:9):          if ( $second = <> ) {
```

The `r` command executes to the end of the file.

A breakpoint is a place in a program where execution stops.

It lets you examine variable values and perhaps change some of them.

For example

```
DB<2> b 36
```

places a break point at line 36 (the top of the loop).

The `c` command executes the program up to the next breakpoint or, if there is none, to the end.

If a line number is included in a `c` command, execution goes to that line.

The `L` command lists all breakpoints.

The breakpoint on a given line can be deleted with the `d` command.

For example,

```
DB<4> d 36
```

eliminates the breakpoint at line 36.

The `D` command deletes all breakpoints.

The `w` command takes a line number parameter and displays a few lines of code around this line (including the line itself).

The command

```
p expression
```

evaluates *expression* and displays its value.

It's usually used to display the current values of program and implicit variables.

The `x` command takes an expression as a parameter and evaluates it.

It lets you change the values of variables (including implicit variables).

*Example*

```
DB<1> c 36
main::(ch2.pl:36):      while ( $next = <> ) {
DB<2> p $first
4
DB<3> x $first = 2
0 2
DB<4> p $first
2
```

The `t` command is a toggle for tracing.

If tracing is off, `t` turns it on.

If tracing is on, `t` turns it off.

When the program is traced, each line is listed (along with its line number) when it is executed.

To exit the debugger, use the `q` command.

To start the debugger over again, use the `R` command.

This preserves the breakpoints.

If the debugger starts not recognizing commands, exit from the DOS command window and start again.