Perl Programming Notes 2011

Perl's Building Blocks: Numbers and Strings

Numeric literals are numbers, and Perl accepts several different ways of writing numbers. All the examples shown in Table 2.1 are valid numeric literals in Perl.

Numeric Literals

Number	Type of Literal
6	An integer
12.5	A floating point number
15	Another floating point number
0.73	Yet another floating-point number
10000000000	Scientific notation
0	Scientific notation (e or E is acceptable)
4_294_296	A large number with underscores instead of commas

Escaping Literals

```
# Escaping quotes
print "Then I said to him, \"Go ahead, make my day.\"\n";
print "Then I said to him, \'Go ahead, make my day.\'\n";

print "\"Test Newline\"\n";
print "\"Test Carriage return\"\t \" after tab \"";
print "more text\n";
```

Sequence	Represents
0	Newline
0	Carriage return
0	Tab
0	Backspace
0	Change next character to uppercase
0	Change next character to lowercase
0	A literal backslash character
0	A literal ' inside of a string surrounded by single-quotation marks ('').

\"	A literal " inside of a string surrounded by quotation marks.

Scalar Variables

The dollar sign—called a type identifier—indicates to Perl that the variable contains scalar data. Other variable types (hashes and arrays) use a different identifier or no identifier at all (filehandles). Variable names in Perl—whether for hashes, arrays, filehandles, or scalars—must conform to the following rules:

Variable names can contain alphabetic (a to z, A to Z) characters, numbers, or an underscore character (_) after the type identifier. The first character of a variable name can't be a number, though.

Variable names are case sensitive. This means that upper- and lowercase are significant in variable names. Thus, each of the following represents a different scalar variable:

```
$value
$VALUE
$Value
$valuE
```

Perl reserves to itself single-character variable names that do not start with an alphabetic character or underscore. Variables such as $_, $", $/, $2, and $$ are special variables and should not be used as normal variables in your Perl programs. The purpose of these special variables will be covered later.
Scalar variables in Perl—in contrast to some other languages—do not have to be declared or initialized in any way before you can use them. To create a scalar variable, just use it. Perl uses a default value when an uninitialized variable is used. If it's used as a number (such as in a math operation), Perl will use the value 0 (zero); if it's used like a string (almost everywhere else), Perl will use the value "", the empty string.

The Special Variable $_

Perl has a special variable, $_, whose value is used as a "default" by many operators and functions. For example, if you simply state print by itself—without specifying a scalar variable or string literal to print—Perl will print the current value of $_:

```
$_="Dark Side of the Moon";
print;    # Prints the value of $_, "Dark side..."
```

Using $_ like this can understandably cause some confusion. It's not really apparent what print is actually printing, especially if the assignment to $_ occurs higher up in the program.
Some operators and functions are actually easier to use with the $_ variable, notably the pattern-matching operators.

Basic Operators

As you saw in Listing 2.1, to assign scalar data to a scalar variable, you use the assignment operator, =. The assignment operator takes the value on the right side and puts it in the variable on the left:

```
$title="Gone With the Wind";
```

```
$pi=3.14159;
```

The operand on the left side of the assignment operator must be something that a value can be assigned to—namely, a variable. The operand on the right side can be any kind of expression. The entire assignment itself is an expression; its value is that of the right-hand expression. This means that, in the following snippet, $a, $b, and $c are all set to 42:

```
$a=$b=$c=42;
```

Here, $c is first set to 42. $b is set to the value of the expression $c=42 (which is 42). $a is then set to the value of the expression $b=42.
The variable being assigned to can even appear on the right side of the assignment operator, as shown here:

```
$a=89*$a;
$count=$count+1;
```

The right side of the assignment operator is evaluated using the old value of $a or $count, and then the result is assigned to the left side as the new value. The second example has a special name in Perl; it's called an increment. You'll read more about incrementing values later.

Operator	Sample Usage	Result
int	int(5.6234)	Returns the integer portion of its argument (5).
length	length("nose")	Returns the length of its string argument (4).
lc	lc("ME TOO")	Returns its argument shifted to lowercase letters ("me too").
Uc	uc("hal 9000")	Returns its argument shifted to uppercase letters ("HAL 9000").
cos	cos(50)	Returns the cosine of 50 in radians (.964966).
rand	rand(5)	Returns a random number from 0 to less than its argument. If the argument is omitted, a number between 0 and 1 is returned.

Increment and Decrement

In the "Numeric Operators" section, you read about a special type of assignment called an increment, which looks like the following:

```
$counter=$counter+1;
```

An increment is typically used to count things, such as the number of records read, or to generate sequence numbers, such as numbering the items in a list. It's such a common idiom in Perl that you can use a special operator called an autoincrement operator (++). The autoincrement operator adds 1 to its operand:

```
$counter++;
```

When this code is executed, $counter is increased by 1.
Perl also offers a shortcut for decreasing a variable's value called, not surprisingly, the autodecrement operator (--). You use the autodecrement exactly the same way you use the autoincrement:
$countdown=10;

```
$countdown--;       # decrease to 9
```

Let me add a final note on the autoincrement operator: When the operator is applied to a text string, and the text string starts with an alphabetic character and is followed by alphabetic characters or numbers, this operator becomes magical. The last (rightmost) character of the string is increased. If it's an alphabetic character, it becomes the next letter in sequence; if it's numeric, the number increases by 1. You can carry across alphabetic and numeric columns as follows:

```
$a="999";
$a++;
print $a;          # prints 1000, as you'd expect
$a="c9";
$a++;
print $a;          # prints d0.  9+1=10, carry 1 to the c.
$a="zzz";
$a++;
print $a;         # prints "aaaa".
```

The autodecrement operator does not decrement strings like this.

Angle Operator (<>)

The angle operator (<>), sometimes called a diamond operator, is primarily used for reading and writing files; it will be covered fully in Hour 5, "Working with Files." However, a brief introduction now will make the exercises more interesting, and when you begin Hour 5, the operator will look somewhat familiar to you.
Until then, you can use the angle operator in its simplest form: <STDIN>. This form indicates to Perl that a line of input should be read from the standard input device—usually the keyboard. The <STDIN> expression returns the line read from the keyboard:

```
print "What size is your shoe? ";
$size=<STDIN>;
print "Your shoe size is $size. Thank you!";
```

The preceding code, when executed (assuming you type 9.5 as your shoe size), would print this on the screen:

```
What is size is your shoe?  9.5
Your shoe size is 9.5
. Thank you!
```

The <STDIN> expression reads from the keyboard until the user presses the Enter key. The entire line of input is returned and is placed in $size. The line of text returned by <STDIN> also includes the newline character that the user typed by pressing Enter. That's why the period and "Thank you!" appear on a new line in the preceding display—the newline was part of the value of $size. Often, as here, you don't want the newline character at the end of the string—just the text. To remove it, you can use the chomp function as follows:

```
print "What size is your shoe?";
$size=<STDIN>;
chomp $size;
print "Your shoe size is $size. Thank you!\n";
```

chomp removes any trailing newline character at the end of its argument

More Assignment Operators

Earlier, you learned that to assign a value to a scalar variable you use the assignment operator (=). Perl actually has an entire set of operators for doing assignments. Every Perl arithmetic operator and quite a few others can be combined to do an assignment and

an operation all at the same time. The general rule for building an assignment with an operator is as follows:
variable operator=expression

This form of the assignment produces the same result as the following:
variable=variable operator expression

Using combined assignments generally doesn't make your programs any more readable but can make them more concise. Following this rule, the statement

```
$a=$a+3;
```

can be reduced to

```
$a+=3;
```

The following are some more examples of assignments:

```
$line.=", at the end";          # ", at the end" is
appended to $line
$y*=$x                          # same as $y=$y*$x
$r%=67;                         # Divide by 67, put
remainder in $r
```

if, elsif, else

```
$r=10;
if ($r==10) {
    print '$r is 10!';
} elsif ($r == 20) {
    print '$r is 20!';
} else {
    print '$r is neither 10 nor 20';
}
```

The Other Relational Operators

So far, you've been comparing numeric quantities in your if statements with the equality operator, ==. Perl actually has quite a few operators for comparing numeric values, most of which are listed in Table

Numeric Operators

Operator	Example	Explanation
==	$x == $y	True if $x equals $y
>	$x > $y	True if $x is greater than $y
<	$x < $y	True if $x is less than $y
>=	$x >= $y	True if $x is greater than or equal to $y
<=	$x <= $y	True if $x is less than or equal to $y
!=	$x != $y	True if $x is not equal to $y

Alphanumeric Relational Operators

Operator	Example	Explanation
Eq	$s eq $t	True if $s is equal to $t
Gt	$s gt $t	True if $s is greater than $t
Lt	$s lt $t	True if $s is less than $t
Ge	$s ge $t	True if $s is greater than or equal to $t
Le	$s le $t	True if $s is less than or equal to $t
Ne	$s ne $t	True if $s is not equal to $t

A Small Number Guessing Game

```
#!/usr/bin/perl -w

$im_thinking_of=int(rand 10);
print "Pick a number:";
$guess=<STDIN>;
chomp $guess;   # Don't forget to remove the newline!

if ($guess>$im_thinking_of) {
    print "You guessed too high!\n";
} elsif ($guess < $im_thinking_of) {
    print "You guessed too low!\n";
} else {
    print "You got it right!\n";
}
```

What Truth Means to Perl

Up to this point, you've been reading about "if this expression is true…" or "…evaluates to true…," but you haven't seen any formal definition of what Perl thinks "true" is. Perl has a few short rules about what is true and what is not true, and the rules actually make sense when you think about them for a bit. The rules are as follows:

The number 0 is false.

The empty string ("") and the string "0" are false.

The undefined value undef is false.

Everything else is true.
Make sense? The only other point to remember is that when you're testing an expression to see whether it's true or false, the expression is simplified—functions are called, operators are applied, math expressions are reduced, and so on—and then converted to a scalar value for evaluation to determine whether it is true or false.
Think about these rules, and then take a look at Table 3.3. Try to guess whether the expression is true or false before you look at the answer.

Expression	True or False?
0	False. The number 0 is false.
10	True. It is a nonzero number and therefore true.

9>8	True. Relational operators return true or false, as you would expect.
-5+5	False. This expression is evaluated and reduced to 0, and 0 is false.
0	False. This number is another representation of 0, as are 0x0, 00, 0b0, and 0e00.
""	False. This expression is explicitly mentioned in the rules as false.
" "	True. There's a space between the quotes, which means they're not entirely empty.
"0.00"	True. Surprise! It's already a string, but not "0" or "". Therefore, it is true.
"00"	True also, for the same reason as "0.00"
"0.00" + 0	False. In this expression, 0.00+0 is evaluated, the result is 0, and that is false.

Careful with == and =

```
$a = 1;
$b = 2;
print qq(The statement "$a = $b" is );
if ($a = $b) { # value is 2, therefore true
    print "true";
} else {
    print "false";
}
```

Operator	Alternative Name	Example Analysis	
&&	and	$s && $t	True only if $s and $t are true
		$q and $p	True only if $q and $p are true
||	or	$a || $b	True if $a is true or $b is true
		$c or $d	True if $c is true or $d is true
!	not		

Looping

As you read in the Introduction, sometimes just making decisions and running code conditionally are not enough. Often you need to run pieces of code over and over again. The exercise presented in Listing 3.1 wasn't much fun because you could take only one guess (well, that and because it's a pointless game). If you want to be able to take multiple guesses, you need to be able to repeat sections of code conditionally, and that's what looping is all about.

Looping with while

The simplest kind of loop is a while loop. A while loop repeats a block of code as long as an expression is true. The syntax for a while loop looks like this:

```
while (expression) block
```

When Perl encounters the while statement, it evaluates the expression. If the expression is true, the block of code is run. Then, when the end of the block is reached, the expression is re-evaluated. If it's still true, the block is repeated, as in the following snippet:
Listing 3.2. Sample while Loop

```
$counter=0;
while ($counter < 10 ) {
    print "Still counting...$counter\n";
    $counter++;
}
```

Looping with for

The for statement is the most complicated and versatile of Perl's looping constructs. The syntax looks like this:

```
for ( initialization; test; increment ) block
```

The three sections of the for statement—initialization, test, and increment—are separated by semicolons. When Perl encounters a for loop, the following sequence takes place:

1. The initialization expression is evaluated.
2. The test expression is evaluated; if it's true, the block of code is run.
3. After the block is executed, the increment is performed, and the test is evaluated again. If the test is still true, the block is run again. This process continues until the test expression is false.

The following is an example of a for loop:

```
for( $a=0; $a<10; $a=$a+2 ) {
    print "a is now $a\n";
}
```

`Or initialize outside the for loop`

```
$i=10;           # initialization
for( ; $i>-1; ) {
    print "$i..";
    $i--;          # actually, a decrement.
}
print "Blast off!\n";
```

Odd Arrangements
The if statements have one more possible syntax. If you have only one expression inside the if block, the expression can actually precede the if statements. So, instead of writing

```
if (test_expression ) {
    expression ;
}
```

you can write

```
expression if (test_expression );
```

The following are a couple of examples of this variation of the syntax:

```
$correct=1 if ($guess == $question);
print "No pi for you!"  if ( $ratio != 3.14159);
```

Fine-Grained Control

In addition to blocks, for, while, if, and other flow-control statements that control blocks of code, you can use Perl statements to control the flow within the blocks.
The simplest statement that gives you this control is last. The last statement causes the innermost currently running loop block to be exited. Consider this example:

```
while($i<15) {
    last if ($i==5);
    $i++;
}
```

The last statement causes the while loop to exit when the value of $i is 5, instead of normally when the while test is false. When you have multiple nested loop statements, last exits the loop currently running.
Listing 3.3. Example of the last Statement

```
for($i=0; $i<100; $i++) {
    for($j=0; $j<100; $j++) {
        if ($i * $j == 140) {
            print "The product of $i and $j is 140\n";
            last;
        }
    }
}
```

Next If

The set of nested loops in Listing 3.3 finds all the whole numbers less than 100 whose products are 140—2 and 70, 4 and 35, and so on—rather inefficiently. The point to note here is the last statement. When a product is found, the result is printed, and the inner loop (the loop iterating over $j) is exited. The outer loop continues executing (by incrementing $i) and reruns the inner loop.
The next statement causes control to be passed back to the top of the loop and the next iteration of the loop to begin, if the loop isn't finished:

```
for($i=0; $i<100; $i++) {
    next if (not $i % 2);
    print "An odd number=$i\n";
}
```

This loop prints all the even numbers from 0 to 98. The next statement causes the loop to go through its next iteration if $i is not even; the $i % 2 expression is the remainder of $i divided by 2. In this case, the print statement is skipped. (A much more efficient way to write this loop would be simply to increase $i by 2, but that wouldn't demonstrate next, would it)?

The redo statement is similar to next, except that the condition isn't re-evaluated. Perl resumes execution back at the beginning of the block and doesn't check whether the termination condition has been met yet.

Labels
Perl allows blocks and some loop statements (for, while) to be labeled. That is, you can place an identifier in front of the block or statement:
MYBLOCK: {
}

The preceding block is labeled as MYBLOCK. Label names follow the same conventions as variable names, with one small exception: Label names do not have an identifying character—%, $, @—as variables do. It's important to make sure that label names do not clash with Perl's built-in keywords. As a matter of style, it's best if label names are all uppercase. You should not have any conflicts with any current or future Perl keywords that way. The for and while statements can all have labels as well.

```
OUTER: while($expr ) {
    INNER: while($expr) {
        statement;
    }
}
```

The last, redo, and next statements can each take a label as an argument. You therefore can exit a specific block. The code in Listing 3.2 found two factors of 140 by using a nested pair of for loops. Suppose you wanted to exit the loops as soon as a factor is found. Without labels, you would need a complex arrangement of flag variables (variables whose only purpose is to store a true or false value for program flow control) and if statements between the two loops because you cannot exit the outer loop from within the inner loop. Labels solve this problem:

```
OUTER: for($i=0; $i<100; $i++) {
    for($j=0; $j<100; $j++) {
        if ($i * $j == 140) {
            print "The product of $i and $j is 140\n";
            last OUTER;
        }
    }
}
```

Now the last statement can specify which loop it wants to exit—in this case, the OUTER loop. This snippet prints only the first pair of factors of 140 that it finds.

Leaving Perl

The exit statement is the ultimate flow-control tool. When Perl encounters an exit statement, the program stops executing, and an exit status is returned by Perl to the operating system. This exit status is usually used to indicate successful completion of the program. You'll learn more about exit statuses in Hour 11, "System Interaction." For now, an exit status of zero means everything went okay. The following is an example of exit:

```
if ($user_response eq 'quit') {
    print "Good Bye!\n";
    exit 0;         # Exit with a status of 0.
}
```

Putting Things into Lists and Arrays

Putting things into a literal list is easy. As you just saw, the syntax for a literal list is a set of parentheses enclosing scalar values. The following is an example:

```
(5, 'apple', $x, 3.14159)
```

This example creates a four-element list containing the numbers 5, the string 'apple', whatever happens to be in the scalar variable $x, and pi.

If the list contains only simple strings, and putting single quotation marks around each string gets to be too much for you, Perl provides a shortcut—the qw operator. An example of qw follows:

```
qw( apples oranges 45.6 $x )
```

This example creates a four-element list. Each element of the list is separated from the others by whitespace (spaces, tabs, or newlines). If you have list elements that have embedded whitespace, you cannot use the qw operator. This code works just as though you had written the following:

```
('apples', 'oranges', '45.6', '$x')
```

Notice that the $x is encased in single quotation marks. The qw operator does not do variable interpolation on elements that look like variables; they are treated as though you wanted them that way literally. So '$x' is not converted to whatever the value of the scalar variable $x is; it's left alone as a string containing a dollar sign and the letter x.
Perl also has a useful operator that works in literal lists; it's called the range operator. The range operator is designated by a pair of periods (..). The following is an example of this operator:

```
(1..10)
```

The range operator takes the left operand (the 1) and the right operand (the 10) and constructs a list consisting of all the numbers between 1 and 10, inclusive. If you need several ranges in a list, you can simply use multiple operators:

```
(1..10, 20..30);
```

The preceding example creates a list of 21 elements: 1 through 10 and 20 through 30.
Giving the range operator a right operand less than the left, such as (10..1), produces an empty list.
The range operator works on strings as well as numbers. The range (a..z) generates a list of all 26 lowercase letters. The range (aa..zz) generates a much larger list of 676 letter pairs starting with aa, ab, ac, ad and ending with zx, zy, zz.

Arrays

Literal lists are usually used to initialize some other structure: an array or a hash. To create an array in Perl, you can simply put something into it. With Perl, unlike other languages, you don't have to tell it ahead of time that you're creating an array or how large the array is going to be. To create a new array and populate it with a list of items, you could do the following:

```
@boys=qw( Greg Peter Bobby );
```

This example, called an array assignment, uses the array assignment operator—the equals sign, just as in a scalar assignment. After that code runs, the array @boys contains three elements: Greg, Peter, and Bobby. Notice also that the code uses the qw operator; using this operator saves you from having to type six quotation marks and two commas.
Array assignments can also involve other arrays or even empty lists, as shown in the following examples:

```
@copy=@original;
@clean=();
```

Here, all the elements of @original are copied into a new array called @copy. If @copy already had elements before the assignment, they are now lost. After the second statement is executed, @clean is empty. Assigning an empty list (or an empty array) to an array variable removes all the elements from the array.

If a literal list contains other lists, arrays, or hashes, these lists are all flattened into one large list. Observe this snippet of code:

```
@boys=qw( Greg Peter Bobby );
@girls=qw( Marcia Jan Cindy );
@kids=(@girls, @boys);
@family=(@kids, ('Mike', 'Carol'), 'Alice');
```

Getting Elements Out of an Array

So far in this hour, you've been slinging around whole arrays and lists and putting information into arrays. How can you get that information back out?
One way to get the contents of the entire array is to put the array variable in double quotation marks:

```
print "@array";
```

```
$array[index]
```

where array is the array name and index is the index of the element you want (also called a subscript). The array doesn't have to exist before you refer to individual elements; if it does not already exist, it just automagically springs into existence. Some examples of accessing array elements follow:

```
@trees=qw(oak cedar maple apple);
print $trees[0];        # Prints "oak"
print $trees[3];        # Prints "apple".
$trees[4]='pine';

@trees=qw(oak cedar maple apple cherry pine peach fir);
@trees[3,4,6];          # Just the fruit trees
@conifers=@trees[5,7];  # Just the conifers
```

Finding the End of an Array

Sometimes you need to find the end of the array—for example, to see how many trees are in the @trees array or to cut some trees out of the @trees array. Perl provides a couple of mechanisms for finding the end. The first is a special variable in the form $#arrayname. It returns the number of the last valid index of the array. Check out this example:

```
@trees=qw(oak cedar maple apple cherry pine peach fir);
print $#trees;
```

Working backwards in the array with $array[-1]
You can also specify negative indexes for arrays. Negative index numbers start counting from the end of the array and work backward. For example, $array[-1] is the last element of @array, $array[-2] is the next to the last element, and so on.

Learning More about Context

Perl is also sensitive to context. Functions and operators in Perl can behave differently depending on what context they're used in. The two most important contexts in Perl are list context and scalar context.

As you've seen, you can use one operator—the equals sign—to perform assignment with both arrays and scalars. The type of expression (list or scalar) on the left side of the assignment operator determines what context the things on the right side are evaluated in, as shown in the following lines of code:

```
$a=$b;        # Scalar on the left: this is scalar context.
@foo=@bar;    # Array on the left: this is list context
($a)=@foo;    # List on the left: this is also list context.
$b=@bar;      # Scalar on the left: this is scalar context.
```

The last line is interesting, because it puts an array into scalar context. As was stated in the previous section, evaluating an array in a scalar context returns the number of elements in the array.

More about the Size and End of an Array

Observe $a and $b in the following few lines of code; they do almost the same thing:

```
@foo=qw( water cola juice lemonade );
$a=@foo;
$b=$#foo;
print "$a\n";
print "$b\n";
```

At the end of this code, $a contains the number 4, and $b contains the number 3. Why the difference? $a is @foo evaluated in a scalar context, and it contains the number of elements. $b, on the other hand, is set to the index of the last element, and indexes start counting at 0.

Because arrays in a scalar context return the number of elements in the array, testing whether an array contains elements becomes this simple:

```
@mydata=qw( oats peas beans barley );
if (@mydata) {
    print "The array has elements!\n";
}
```

Here, the array @mydata is evaluated as a scalar, and it returns the number of elements—in this case, 4. The number 4 evaluates to true in an if statement, and the body of the if block is run.

By the Way

Actually, @mydata here is used in a special kind of scalar context called a Boolean context, but it behaves the same way. Boolean context occurs when Perl expects a true or false value, such as in an if statement's test expression. One other context, called void context, will be explained in Hour 9, "More Functions and Operators."

Context with Operators and Functions

Many of Perl's operators and functions force their arguments to be either scalar context or list context. Sometimes the operators or functions behave differently depending on what context they're in. Some functions you've already encountered have these properties; however, this fact hasn't been important until now, because they have only had scalars to work on.
The print function expects a list as an argument. It doesn't particularly matter what context the list is evaluated in, though. So printing an array with print like this causes the array to be evaluated in a list context, yielding the elements of @foo:

```
print @foo;
```

You can use a special pseudofunction called scalar to force something into a scalar context:

```
print scalar(@foo);
```

This example prints the number of elements in @foo. The scalar function forces @foo to be evaluated in a scalar context, so @foo returns the number of elements in @foo. Then the print function simply prints the number returned.
The chomp function you learned about in Hour 2, "Perl's Building Blocks: Numbers and Strings," takes either an array or a scalar as an argument. If chomp is presented with a scalar, it removes the record separator from the end of the scalar. If it is presented with an array, it removes the record separator from the end of each scalar in the array.

The chomp function you learned about in Hour 2, "Perl's Building Blocks: Numbers and Strings," takes either an array or a scalar as an argument. If chomp is presented with a scalar, it removes the record separator from the end of the scalar. If it is presented with an array, it removes the record separator from the end of each scalar in the array.

Manipulating Arrays

Now that you've learned the basic rules for building arrays, it's time to learn some tools to help you manipulate those arrays to perform useful tasks.
Stepping Through an Array

In Hour 3, "Controlling the Program's Flow," you learned about making loops with while, for, and other constructs. Many tasks you'll want to perform involve examining each element of an array. This process is called iterating over the array. One way you could do so is to use a for loop, as follows:

```
@flavors=qw( chocolate vanilla strawberry mint sherbet );
for($index=0; $index<@flavors; $index++) {
    print "My favorite flavor is $flavors[$index] and..."
}
print "many others.\n";


foreach $cone (@flavors) {
    print "I'd like a cone of $cone\n";
}
```

Converting Between Arrays and Scalars

Perl doesn't have one general rule about converting between scalars and arrays. Rather, Perl provides many functions and operators for converting between the two types.
One method to convert a scalar into an array is the split function. The split function takes a pattern and a scalar, uses the pattern to split the scalar apart, and returns a list of the pieces. The first argument is the pattern (here surrounded by slashes), and the second argument is the scalar to split apart:

```
@words=split(/ /, "The quick brown fox");
```

After you run this code, @words contains each of the words The, quick, brown, and fox —without the spaces. If you don't specify a second argument, the variable $_ issplit. If you don't specify a pattern or a string, whitespace is used to split apart the variable $_.
One special pattern, // (the null pattern), splits apart the scalar into individual characters, as shown here:

```
while(<STDIN>) {
   ($firstchar)=split(//, $_);
   print "The first character was $firstchar\n";
}
```

The first line reads from the terminal one line at a time, setting $_ equal to that line. The second line splits $_ apart using the null pattern. The split function returns a list of each character from the line in $_. That list is assigned to the list on the left side, and the first element of the list is assigned to $firstchar; the rest are discarded.

By the Way

The patterns used by split are actually regular expressions. Regular expressions are a complex pattern-matching language introduced in Hour 6, "Pattern Matching." For now, the examples will use simple patterns such as spaces, colons, commas, and such. After you've learned about regular expressions, I will give examples that use more complex patterns to pull apart scalars with split.
This method of splitting a scalar into a list of scalar variables is common in Perl. When you're splitting apart a scalar in which each piece is a distinct element—such as fields in a record—it's easier to figure out which piece is what when you name each piece as it's split. Observe the following:

```
@Music=('White Album,Beatles',
       'Graceland,Paul Simon',
       'A Boy Named Sue,Goo Goo Dolls');
foreach $record (@Music) {
    ($record_name, $artist)=split(',', $record);
}
```

When you split directly into a list with named scalars, you can clearly see which fields represent what. The first field is a record name, and the second field is the artist. Had the code split into an array, the distinction between fields might not have been as clear.
To create scalars out of arrays—the reverse of split—you can use the Perl join function. join takes a string and a list, joins the elements of the list together using the string as a separator, and then returns the resulting string. Consider this example:

```
$numbers=join(',', (1..10));
```

This example assigns the string 1,2,3,4,5,6,7,8,9,10 to $numbers.
In Perl the output (return value) of one function can be used as an input value (argument) in another function. You can use split and join to pull a string apart and put it back together all at the same time, as seen here:

```
$message="Elvis was here";
print "The string \"$message\" consists of:",
         join('-', split(//, $message));
```

In this example, the $message is split into a list by split. That list is used by the join function and put back together with dashes. The result is the following message:
The string "Elvis was here" consists of: E-l-v-i-s- -w-a-s- -h-e-r-e

Reordering Your Array

When you're building arrays, often you might want them to come out in a different order than you built them. For example, if your Perl program reads a list of customers in from a file, printing that customer list in alphabetical order would be reasonable. For sorting data, Perl provides the sort function. The sort function takes as its argument a list and sorts it in (roughly speaking) alphabetical order; the function then returns a new list in sorted order. The original array remains untouched, as you can see in this example:

```
@Chiefs=qw(Clinton Bush Reagan Carter Ford Nixon);
print join(' ', sort @Chiefs), "\n";
print join(' ', @Chiefs), "\n";
```

This example prints the sorted list of presidents (Bush Carter Clinton Ford Nixon Reagan) and then prints the list again in its original order.
Be forewarned that the default sort order is ASCII order. This means that all words that start with uppercase characters sort before words that begin in lowercase letters.
Numbers do not sort in ASCII order the way you would expect. They don't sort by value. For example, 11 sorts higher than 100. In cases like this, you need to sort by something other than the default order.
The sort function allows you to sort in whatever order you want by using a block of code (or a subroutine name, discussed in Hour 8, "Functions") as the second argument. Inside the block (or subroutine), two variables, $a and $b, are set to two elements of the list. The block's task is to return –1, 0, or 1 depending on whether the $a is less than $b, equal to $b, or greater than $b, respectively. The following is an example of the hard way to do a numeric sort, assuming that @numbers is full of numeric values:

```
@sorted=sort { return(1) if ($a>$b);
            return(0) if ($a==$b);
            return(-1) if ($a<$b); } @numbers;
```

(spaceship)> (spaceship operator)>)>The preceding example certainly sorts @numbers numerically. But the code looks far too complicated for such a common task. As you might suspect for anything this cumbersome, Perl has a shortcut: the "spaceship" operator, <=>. The spaceship operator gets its name because it somewhat resembles a flying saucer, seen from the side. It returns –1 if its left operand is less than the right, 0 if the two operands are equal, and 1 if the left operand is greater than the right:

```
@sorted=sort { $a<=>$b; } @numbers;
```

This code is much cleaner, easier to look at, and more straightforward. You should use the spaceship operator only to compare numeric values.
To compare alphabetic strings, use the cmp operator, which works exactly the same way.
You can put together more complex sorting arrangements by simply making a more sophisticated sort routine. Section 4 of the Perl Frequently Asked Questions (FAQ) has some more sophisticated examples of this if you need them.
The final function for this hour is an easy function, reverse. The reverse function, when given a scalar value in a scalar context, reverses the string's characters and returns the reversed string. The call reverse("Perl") in a scalar context, for example, returns lreP.

When given a list in a list context, reverse returns the elements of the list in reverse order, as in this example:

```
@lines=qw(I do not like gree
```