Perl Data Types and Variables

Data, variables, expressions, and much more

Data types in Perl

- Perl is unlike most high-level languages in that it does not make a formal distinction between numeric data and character data, nor between whole numbers and numbers with fractional parts.
- Most modern languages invented before Perl ask you to declare in advance whether a variable will store a character, a whole number, a floating point number, or something else. ***Not Perl***.

Typelessness in Perl

- To illustrate, if we declare a variable named **`$anything`** using the statement

```
my $anything;
```

then all of the following assignment statements are valid:

```
$anything = "Now I am a string";
$anything = 10;
$anything = 3.141592;
```

- In short, in Perl, variables are ***untyped***.

Three data classes

- However, Perl does distinguish the ***class*** of data, i.e., whether it is primitive or structured. ***Scalars*** are primitive and ***lists*** and ***hashes*** are structured:

scalar data	a single data item
list data	a sequence or ordered list of scalars
hash data	an unordered collection of (key,value) pairs

- Scalars may be numbers such as **`12`** or **`44.3`** or strings like **`"the swarthy toads did gyre and gimble in the wabe"`**. There are other kinds of scalars as well, as you will now see.

Literals

- A ***literal*** is a value explicitly represented in a program. For example, in

```
print 56;
```

  the numeral **`56`** is a ***numeric literal***, and in

```
print "This is a string literal.";
```

  the string **`"This is a string literal."`** is a ***string literal***.

- Some people call literals ***constants***. It is more accurate to call them literals.

Numeric literals in Perl

Integers:

```
-52  6994   6_994
 #Note that _ can be used in place of ','
```

Fixed decimal:

```
      3.1415   -2.635  1.00
```

Floating point (scientific notation):

```
      7.25e45       # 7.25  x 10^45
      -12e-48       # -12.0 x 10^(-48)
      1.000E-5      # 1.000 x 10^(-5)
```

Octal: `0377 # octal, starts with 0`

Hexadecimal: `0x34Fb # hexadecimal, case insensitive`

Internal representation of numbers

- Internally, all numbers are stored and computed on as ***floating point values***. Floating-point representation is like scientific notation; a number is stored as a ***mantissa*** and an ***exponent***, either of which may be positive or negative or zero.
- The advantage of floating-point is that it can represent fractions, extremely large magnitudes such as the number of stars in the universe, and extremely small magnitudes such as the distance from a nucleus to an electron in meters.
- The disadvantage is that not all numbers can be represented accurately. This is not only because of scientific notation, but because of the nature of digital computers.

Inaccuracy of numbers

- Not all numbers can be represented exactly in a digital computer, because computers can only represent numbers as sums of powers of 2. Try writing 1/3 as a sum of powers of 2 and you will see it cannot be done. How about 1/5th? Not that either. Not 1/10th either. In fact, most numbers are represented with some small inaccuracy.

Gaps in numeric representation

- Imagine that the black rectangles below represent numbers that can be represented in floating point and blue are those that cannot. The rectangle below is like a piece of the number line showing that there are big gaps between representable numbers.

- The inaccuracies and gaps can lead to large errors in calculations. Numerical Methods is a branch of computer science concerned with methods of computation that do not produce large errors.

String literals: Double-quoted strings

- Double-quoted strings can contain special characters, such as:

  `\n, \t, \r, \177` (octal 177)

  You have already seen that `\n` is a newline character. `\t` is the tab. You can see a more complete list of these special characters in the textbook.
- `\c` is the prefix for inserting control characters into strings, as in:

  `\cC  \cD` ***control-C and control-D***

String literals: Case conversion

- Other useful special characters are **\L**, **\U**, and **\E**. The pair
- **\L***text***\E** converts text to lowercase

  \U*text***\E** converts text to UPPERCASE:

```
"\LABc\E"   # is abc
"\UaBc\E"    # is ABC
"\L123\E"   # is 123
```

Backslash escape character

- If you want to insert a character that has special meaning, you use the backslash to "escape" the special meaning. Since `"` is a special character, to insert a `"` into a string, write `\"`
- To insert a backslash, use `\\`. Example:

```
print "Use \\ before \" to put a \" in a string.";
```

- This will print:

```
Use \ before " to put a " in a string.
```

Variables in double-quoted strings

- You can put a variable in a double-quoted string. When the string is evaluated, the variable's value will be substituted into the string in the given position.

```
$greeting = "Welcome to my planet";
$count = 40_000_000_000;
print "$greeting.\nThere are $count of us here.\n";
```

will print

```
Welcome to my planet.
There are 40000000000 of us here.
```


String literals: Single-quoted strings

- Perl also has single-quoted strings, ***and they behave differently*** than double-quoted strings.
- Within a single quoted string, variables are not substituted and backslashed special characters have no meaning:

```
$number = 100;
print '$number';# prints $number
print '
';
                    # prints a blank line
```

String literals: single-quoted strings

- To put `'` or `\` into a single-quoted string, precede with a `\` as in

```
'don\'t do it!'   # don't do it!
'\\/\\/'          # \/\/
```

Scalar variables

- Scalar variables can store numeric data, strings, and references. We will get to references later.
- Although Perl does not require you to declare variables before using them, in this class, you will be required to use the `strict` pragma, which forces you to declare them first (or use another alternative.)
- The `my` declaration declares variables. To declare more than one, put them in parentheses, otherwise only the first will be declared properly:

```
my ($count, $pattern);
```

The scoop on `my` declarations

- When you use `my` to declare a variable, as in

  ```
  my $var;
  ```

  you are telling Perl two things:

  1. That `$var` is a name that can be used from that point forward in the innermost enclosing block (pair of curly braces), and
  2. that there is storage set aside for `$var`

  In this case you are also telling Perl that it is a ***scalar variable.***

Variables, names, and storage

- You should picture the relationship between the variable, its name, and its storage like this:

The declaration creates the empty storage container and the name, and ***binds*** (attaches) the name **`$var`** to the container. The blue dot represents the connection between the name and the storage in the schematic.

The assignment operator

- The assignment operator, **=**, is used to store a value into a variable, as in:

```
$count = 0;
$pi = 3.141592;
$title = "A First Perl Program";
```

- The assignment operator is also used for ***list assignments***, as in:

```
($firstname, $initial, $lastname) =
                    ('Alfred', 'E', 'Newman');
```

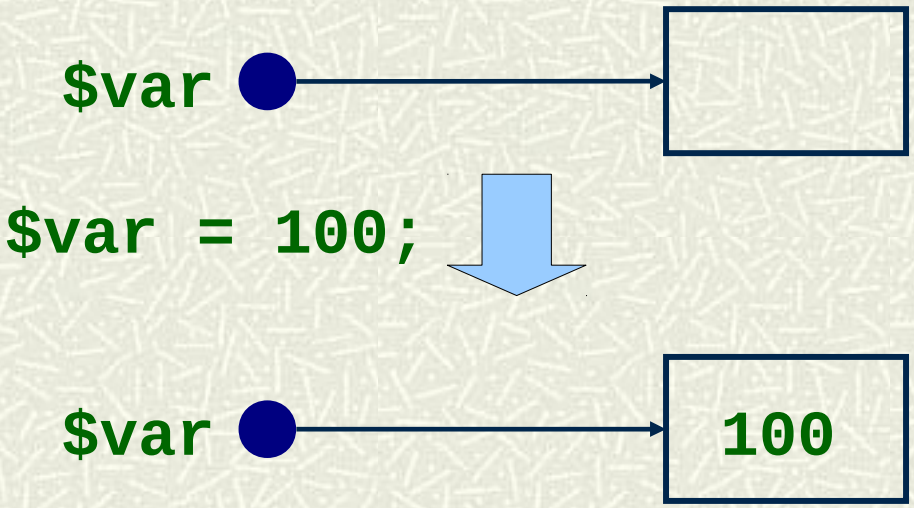
- This assigns `'Alfred'` to `$firstname`, `'E'` to `$initial`, and `'Newman'` to `$lastname`.

Assignment statement semantics

- The effect of the assignment statement,

```
$var = 100;
```

on the schematic diagram of the name-variable relationship is as follows:



Rules to remember

- Reading a value from a variable does not remove the value from the variable.
- Storing a value in a variable replaces any value that was there before.
- If you have not assigned a value to a variable, it will have the value `0` or `undef` or `""`, depending on the context in which it is used. `undef` is a special value in Perl that represents the idea of nothingness.

Input

- You saw in the first Perl program an example of input:

```
my $response = <STDIN>;
```

  declares the variable **`$response`** and assigns to it whatever the user enters on the keyboard, up to and including the first newline character.
- The **`<STDIN>`** symbol represents an input operation in Perl.
- For now, this is the only means you have of obtaining user input in your program. Later you will learn about other options.

Reading Multiple Values

- To read multiple values, one way is to write repeated instructions like this:

```
my $x1 = <STDIN>;
my $x2 = <STDIN>;
my $x3 = <STDIN>;
```

and then enter the values on separate lines (using the **ENTER** key to separate them). Later you will see how to read a list of values from **STDIN** using a single instruction.

Expressions

- An ***expression***, informally, is either a simple variable or literal or function that returns a value, or a composition of these things built up by applying various kinds of operations such as arithmetic operations and other operations as well.
- These are examples of legal Perl expressions:

```
12 + 9          # 21 :  12 + 9 == 21
6 * 7           # 42 : '*' is 'times', 6*7 == 42
$sum/$count     # division of $sum by $count
3 ** 2          # 9  : 3 raised to 2nd power
$count - 1      # $count minus 1
16 % 10         # 6  : 16 % 10 is remainder of 16 / 10
```

More examples

- Assume that `$x == 5`, `$y == 11`, and `$z == 3`.

```
$x ** $z     #  5**3 == 125
$y % $z      #  11 % 3 == 2 because 11/3 = 3 rem 2
2 ** $x      #  2**5 == 32
```

- As in math, operators have precedence. For example, `3+5*4` is really `3+(5*4)` because `*` has higher precedence than `+`.

```
5+4 * 3        #  5 + 12 == 17
72 /6/3        #  (72/6)/3 == 12/3 == 4
2 ** 3 ** 2    #  2 ** 9 == 512
```

- I.e., all operators are left-associative except `**`, which is right associative.

Parentheses

- You can use parentheses to alter the precedence of operators.

```
(5 + 4) * 3    # 9*3 == 27
72 / (6 / 3)   # 72/2 == 36
(2 ** 3) ** 2 # 8**2 == 64
```

- Remember that it does not matter how much white space you put in an expression -- it will not alter the precedence!

String expressions

- While the arithmetic operators are familiar enough, string operators may be something new for you. Perl provides two that you should know.
- The `.` is a ***concatenation operator***:

```
print 'Humpty' . 'Dumpty.';
```

prints **HumptyDumpty.**

```
$x = "1234567890";
print $x . $x . $x;
```

prints **123456789012345678901234567890**

Repetition

The **x** is a ***repetition operator***. It replicates the preceding string. Thus,

```
print 'walla' x 2  . 'bingbang';
```

prints **wallawallabingbang**

- Repetition can be convenient. Consider these two statements:

```
$dots = '. . ' x 40;
print $dots x 24;
```

which fills the screen with alternating dots very succinctly.

Mixed types

- Since a variable might contain a string or a number, how does Perl execute an instruction like

```
$z = $x + $y;
```

- The answer is simple: the operator is the key. Perl uses the operator to decide.
- `+` is an addition operator. It only acts on numeric values. So Perl tries to interpret whatever is in `$x` and `$y` as numbers.

Mixed types

- Similarly, in `$first . $last`, the `.` is a string operator, so Perl tries to convert whatever is in `$first` and `$last` into strings and then concatenate them.
- In short, Perl's solution to mixed types is to try to convert operands to the type that matches the operator when the program reaches that point.

Type conversion rules

- A string will be converted to a number as follows:
- Any leading white space is stripped.
- If the first character looks like the start of a number (a `+` or `-`, or a digit), all characters that can be part of the number are taken to be part of the number. As soon as a non-number character is found, the rest of the string is discarded.
- If there is no number found by this rule, it has the value `0`.

`"   +32.7"` becomes `32.7`

`"hello"` becomes `0`

`"  12.3xy"` becomes `12.3` because the `xy` is dropped

Type conversion

- Numbers are converted to strings in the obvious way: Perl just converts the number to its decimal numeral. Thus

  `32.5` becomes the string `"32.5"`.

  `3.141592` becomes the string `"3.141592"`.

- With the `-w` switch turned on, you will see warnings when Perl does conversions from strings that do not look like numbers to numbers.

Shorthand assignments

- Perl, like some other languages, provides shorthand assignment operators. These combine a binary operator such as `+`, `*`, `-`, `%`, `/` or string operators with the assignment operator.

`$count += 1;`	is shorthand for	`$count = $count + 1;`
`$prod *= 2;`	is shorthand for	`$prod = $prod * 2;`
`$str .= "/";`	is shorthand for	`$str = $str . "/";`

- These are, quite literally, just shorthand notations to reduce the amount of typing you have to do. They are useful, but it is not essential that you know them.

List data

- In Perl, a ***list*** is an ***ordered collection of scalar values***. A list is written as a comma-separated list of scalar expressions enclosed in parentheses:

```
( 11, 7, 5, 3, 2.0 )
("Foollum", 'N.E.', "Howe", 'U', "Khan")
( $base, $height, $base * $height / 2 )
( $position, $salary * 7 , 3.141592 )
```

- Notice that the elements do not have to be the same "type". In Perl there is no concept of type. You can mix expressions containing any numbers, strings, and variables.

Lists inside lists

- You can even ***interpolate*** one list into another:

  ```
  ( (1, 2, 3), (4, 5), 6) )
  ```

  is the same list as

  ```
  (1, 2, 3, 4, 5, 6 )
  ```

- If `$a == 4` and `$b == 3`, then

  ```
  (($a/2,$b, $a ),($a + $b),(),$a + $b )
  ```

  is the list

  ```
  ( 2, 3, 4, 7, 7 )
  ```

- Notice the empty list `()` in this example. Lists can be empty. Interpolating an empty list into a list has no effect on it.

Lists formed from ranges

- You can also create a list using the ***ellipsis*** to create a range. For example:

```
(1..5)          means (1,2,3,4,5)
(1..5,6,8)      means (1,2,3,4,5,6,8)
('a' .. 'd')    means ('a', 'b', 'c', 'd')
('aa'.. 'cc')   means ('aa','ab',...'az',
                      'ba','bb',...'bz',
                      'ca', 'cb', 'cc')
```

- This last one shows that Perl enumerates the strings as if they were numbers, treating `'a'` like a 1 and `'c'` like a 2.

Printing lists

- The **`print`** function can also take a list as its argument, instead of a quoted string. In

```
print ( 'See', 'Spot', 'run.', "\n");
```

  the print function is given a list with four elements. But it prints "**`SeeSpotrun`**." which is not what you really want.

- If the **`print`** function is given a list, it prints the elements literally, without padding them with white space. You need to supply the white space yourself:

```
print ( 'See ', 'Spot ', 'run.', "\n");
```

  which prints **`See Spot run.`**

Quoteword **qw()**

- The **quoteword** function, **qw()**, takes a sequence of whitespace-separated words and returns a list of quoted strings. This is a convenient function because it reduces the chance of mistyping a quote.

```
qw( See Spot run. )
```

is the same as

```
( 'See', 'Spot', 'run.' )
```

- You can use a different pair of delimiters. In fact any non-alphanumeric character can be used:

```
qw/See Spot run/
```
or
```
qw#See Spot run#
```

Array (list) variables

- An array or list variable is a variable that can store list data. Array variables start with the @ symbol, not the **$**. (Think of the @ as the letter 'a' for array.)

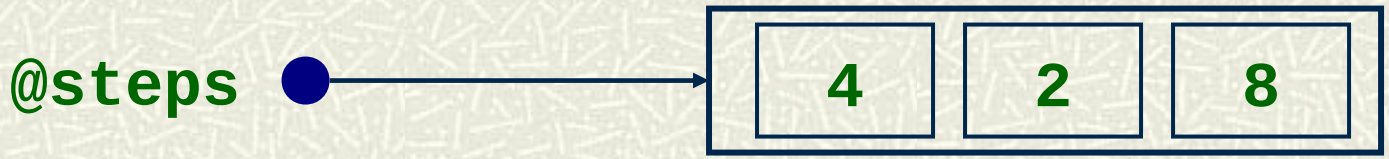
```
@steps = (1, 2, 3);
@shortcut = (@steps,6,7,8);
# @steps is interpolated into shortcut, so
# @shortcut is (1,2,3,6,7,8)
@steps = (@steps,@steps);
# @steps is now (1,2,3,1,2,3)
@newsteps = @steps;
# @newsteps is (1,2,3,1,2,3)
```

Accessing the Elements

- The assignment

```
@steps = (4, 2, 8);
```

causes the array named **@steps** to be filled as shown below.



- To refer to an individual element of the array, you need to use the ***subscript operator***, **[]** and change the **@** to a **$**

```
print $steps[0];
```

prints **4**, the first value. You must use **$** because a single array element is a scalar, not a list. The **$** is always the first symbol in a scalar.

Subscript operator

- Notice two things:
  - ***you have to use* `$` *instead of* `@` *when subscripting the array*** to access a single element because a single element is a scalar value, and `$` indicates scalars, and
  - ***the first subscript is 0, not 1.***
- For example

```
print $steps[1];
```

prints the second value, `2`, and

```
$steps[0] = $steps[1];
```

copies the second value into the first.

Array assignment

- The statement,

```
@steps = (4, 2, 8);
```

  is an ***array assignment***, also called a ***list assignment***. Although the assignment operator '**=**' looks the same in the following two statements:

```
@steps = (4, 2, 8);  # list assignment
$size = 12;          # scalar assignment
```

  it is not the same operation. In the first, a list of things is copied into a list on the left. In the second, a single value is copied into the variable on the left.

The Importance of arrays

- Array variables are important because most of the time, programs must process lists of values. Arrays give you a convenient way to access the elements of a list.
- Soon you will see how arrays and loops go hand in hand, and how very small Perl programs can be used to apply the same set of operations to data sets of large size, precisely because of loops and arrays.
- It is important to master this concept now.

Array slices

- An ***array slice*** is a sublist of an array. Suppose that we have an array named **@scores**. Suppose

  @scores == (80,85,90,70,75,100,95)

  Then if we assign a slice to **@sample_scores** as follows:

  my @sample_scores = @scores[0, 2, 5];

  @sample_scores will be **(80,90,100)**

- This is a very powerful tool. In general, it is of the form

  @array_name[list of index values]

- The list of values can even be a range or another list.

Examples of Array Slices

```
# Fill @numbers with 0,1,2,3,4,…, 40
my @numbers = (0..40);
# declare @some_numbers to have 10 0's
my @some_numbers = (0,0,0,0,0,0,0,0,0,0);

# use slice to copy 2,3,6 into 1,2,3
@some_numbers[1,2,3] = @numbers[0,3,6];
# @some_numbers is now (0,2,3,6,0,0,0,0,0,0)

# declare an array containing the numbers (1,4,7)
my @indices = (1,4,7);

# print $numbers[1], $numbers[4], $numbers[7]:
print @numbers[@indices], "\n";
```


Array functions

- Perl has a number of functions that act on arrays. For now, we will look at these: **`unshift()`**, **`shift()`**, **`push()`**, and **`pop()`**:
  - **`shift()`** removes the first element from an array;
  - **`unshift()`** inserts a new element before the first element in the array;
  - **`push()`** adds a new element after the last element in an array;
  - **`pop()`** removes the last element from the array.
- The following slides demonstrate.

`shift()` and `unshift()` Functions

- ***Shifting*** an array means sliding all of its elements to the left. Imagine you are in a theatre in a full row, and the usher comes along and asks everyone in the row to move over one seat to the left. The person in the leftmost seat of the row has to leave. That is a ***shift***.
- ***Unshifting*** is just the opposite of shifting. It is an unintuitive name for the operation perhaps. Unshifting is what would happen if the usher arrived with a new patron who wants to sit in the leftmost seat. The usher tells everyone to move one seat to the right, and seats the new person in the vacant leftmost seat.
- The next slides illustrate.

Shifting

- The effect of executing `$x = shift(@list)`

before:


```
$x = shift(@list);
```

after:

Unshifting

- The effect of executing **`unshift(@list,$x);`**

before:


```
unshift(@list, $x);
```

after:

Examples of **shift()** and **unshift()**

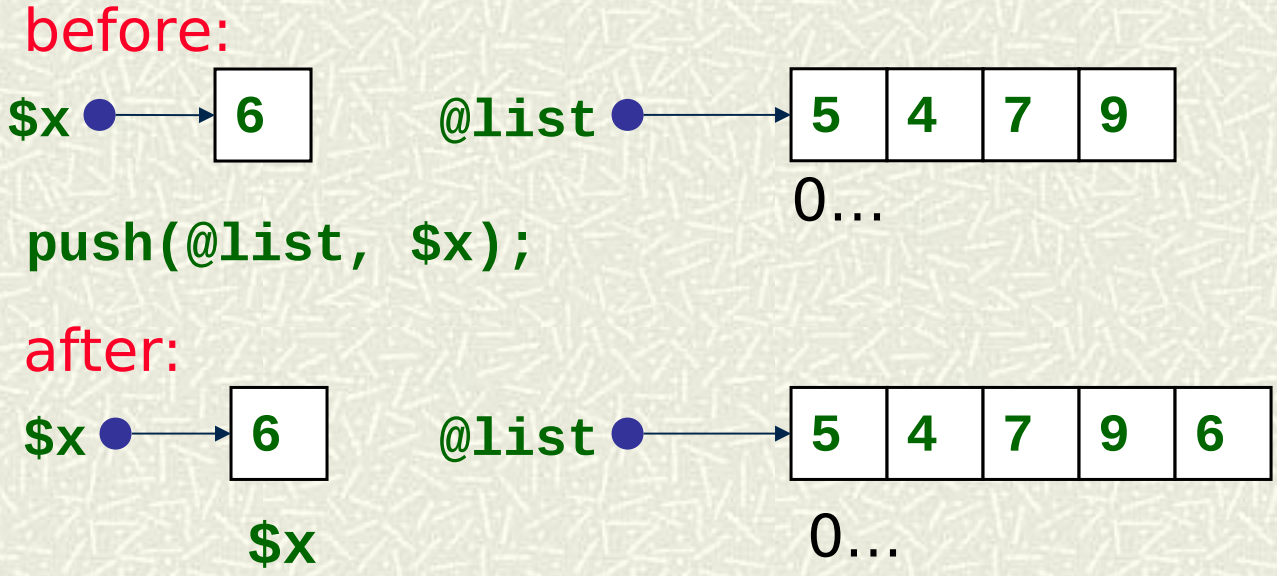
```
my $x = 25;
my @mylist = (5,10,15,20);

unshift(@mylist,$x);
# @mylist is now (25, 5, 10, 15, 20)
unshift(@mylist,4,6);
# @mylist is now (4, 6, 25, 5, 10, 15, 20)
# NOT (6, 4, 25, 5, 10, 15, 20) !!!!
$x = shift(@list);
# $x is 4 and @mylist is (6, 25, 5, 10, 15, 20)
```

The `push()` function

- If `@list` is an array, then

  `push(@list, $x);`

  puts the value in `$x` at the end of the `@list` array, as shown



More examples

```
@a = (1, 2, 3);    # @a is (1,2,3)
push(@a, 5,6,7);   # @a is now (1,2,3,5,6,7)
push(@a,(10,12)); # @a is now (1,2,3,5,6,7,10,12)
@b = (1 ,2, 3);    # @b is (1,2,3)
push(@b, @b);      # @b is now (1,2,3,1,2,3)
```

The `pop()` function

- If `@list` is an array, then

  ```
  pop(@list);
  ```

  removes the last element from `@list`. It returns its value so that it can be accessed:

  ```
  print pop(@list);
  ```

  prints the value that was removed.

The `pop()` function

- The effect of executing `$x = pop(@list)`

before:


```
$x = pop(@list);
```

after:

The concept of a hash

- A ***hash*** is an unordered collection of ***key-value pairs***. A key-value pair consists of a key and a value associated uniquely with that key. Keys must always be strings, but values can be any scalars.
- Examples of key-value pairs:

KEY	VALUE
last name	***phone number***
ss#	***salary***
country	***capital city***

- In general, a key is a look-up term, and the value associated with the key is the thing you want to find.

A hash is a finite function

- A hash is really just a finite function. A finite function is a function on a finite set that assigns to each member of the set a unique value.
- For example, let S be the set of usernames in our class. For each username, there is a unique password assigned to that username on the server. Let ***passwd(x)*** be the function that returns the password for username x. Then, ***passwd( )*** is essentially a hash: it is a collection of pairs of the form

  `(username, password)`

Defining a hash

- In Perl, I can create a hash variable named **%passwd** that can contain the collection of pairs. I can initialize it as follows:

```
%passwd =  (
    'lsnicket' => 'kjashdu',
    'ijones'   => 'yn89234j',
    'jcricket' => '63ndcj8fs',
    'aeinstein'=> 'emc2jk'
     # Notice the commas, not semicolons!
);
```

- The notation *key* **=>** *value* creates a pair; this hash has four pairs, such as key **'ijones'** with value **'yn89234j'**.

Hash literals

- The form of a hash literal looks like a list literal -- it is enclosed in parentheses and consists of a comma-separated list of pairs. The first member of each pair does not have to have quotes. This is a hash literal:

```
(
    lsnicket  => 'kjashdu',
    ijones    => 'yn89234j',
    jcricket  => '63ndcj8fs',
    aeinstein => 'emc2jk'
)
```

- There are other forms, but for now, we will stick to this one.

Hash variables

- Hash variables always begin with a percent sign **%**.
- The value associated with a single key is a scalar, and to access it, you use the syntax:

  `$hashname{key}`
- Notice that the **$** is used instead of **%** and that curly braces **{}** are used around the key. ***This is important; the only difference in syntax between array element access and hash element access is the difference between*** **[]** ***and*** **{}**.

Examples

```
my %phones = ( zach => "123-4567",
               anna => "335-7877" );
# add a new pair dynamically:
$phones{taylor} = "634-5789";

# change zach's phone number:
$phones{zach} = "123-9999";

# remove anna and her phone from the hash using the
# delete function:
delete $phones{anna};
```

Example Continued

```
# here's how to look-up a (key,value) in a hash
print "Enter name to look up:";
my $name = <STDIN>; # read user's input into $name
chomp($name); # remove newline character from $name

# This will be explained shortly:
if $phones{$name} { # if exists a key-value pair
    print "$name : $phones{$name}\n";
} else {
    print "$name not found.\n";
}
```

Example Continued Further

```
# The keys() function returns a list of the keys
# in its hash argument:
my @names = keys(%phones);
print "@names\n";

# The values() function returns a list of the values
# in the hash:
my @numbers = values(%phones);
print "@numbers\n";

# Run the hashes_01.pl demo in chapter04 to see how
# this works.
```


Some reminders about hashes

- Hashes do not have any ordering. You cannot assume that the pairs are in the same order in which you first wrote them.
- If you print out the elements of a hash as if they were a list, the order will not be predictable.
- Use **`$hashname{key}`** to access a single value.
- If **`key`** does not exist in the hash, **`$hashname{key}`** is undefined and the same as logical false.
- Assigning to **`$hashname{key}`** replaces the value that was there before or creates a pair if it did not exist.

Context

- In Perl, every operation or evaluation takes place in a specific ***context***. Roughly, there are two contexts, ***list*** and ***scalar***.
- For example, consider the two assignment statements:

```
@scores = (18, 20, 24);
$x  = 10;
```

- The first is list assignment because the variable on the left of the **=** is an array, which forces the expression on the right to be evaluated as a list. The second is scalar assignment, because the variable on the left is a scalar, so the expression on the right is evaluated in scalar context.

Scalar Context

- But consider this statement:

```
$z = @scores;
```

- There is a scalar on the left side and an array on the right. What will Perl do?
- The rule remains the same: The scalar **`$z`** on the left forces the assignment to be scalar assignment, which forces **`@scores`** to be evaluated in scalar context. The scalar value of an array variable is the number of elements in the array, which is **`3`**, so **`$z`** is assigned the number **`3`**.

List Context

- Now consider this statement:

```
@scores = 5;
```

- There is an array on the left side and a scalar on the right.
- The rule remains the same: The array `@scores` on the left forces the assignment to be list assignment, which forces `5` to be evaluated in list context. The list value of a scalar is the list consisting of that scalar alone, `(5)`, so `@scores` gets `(5)`.

Determining context

- There are different types of scalar context. For example, sometimes an expression is a string expression and sometimes it is numeric. How can you determine how Perl will evaluate expressions?
- Unfortunately, there is no simple rule. But usually it is intuitive.
- One rule that will help in most cases is that the operators and functions themselves are specific to contexts.

Determining context

- For example, the string repetition operator **`x`** expects a string on the left and a number on the right, so it tries to evaluate its left as a string and its right as a number:

```
print 32 x "4a";
```

- will print **`32323232`** because Perl will convert **`32`** to **`"32"`** and use its rules for converting strings to numbers, which converts **`"4a"`** to **`4`**.
- In the expression **`("smith" == $name)`**, the **`==`** operator is numeric comparison, so Perl will convert **`"smith"`** to a number, which will be **`0`** and compare it to whatever value is in **`$name`** as a number.

Scalar value of a list literal

- Perl throws us for a loop here. *Although the scalar value of an array variable is the number of elements in the array, the scalar value of a list literal is the value of its last element.* In

```
$foo = (10, 20, 30, 40, 50);
```

$foo will have the value **50**, not **5**. But in

```
@bar = (10, 20, 30, 40, 50);
$huh = @bar;
```

$huh will have the value **5**.

References

- When I introduced variables, I showed you this schematic diagram:

I wrote in that slide that the blue dot represents the connection between the name and storage in the schematic. It is time to say this more accurately.

Variables, names, and addresses

- The storage container for a variable exists somewhere in memory when the program is running. It has a ***memory address***.
- ***When you create a variable in Perl, associated to the variable's name is the address of the storage container.***
- Now that you have learned about hashes, think of it this way. The set of variable names and the addresses of their storage containers is essentially a hash that Perl uses when your program is running! The pair is essentially

  (variable name, address of storage)

Variables exposed

- The blue dot in the schematic

is the ***address*** of the storage container. When you write **`$var`** in your program, Perl looks up the name **`var`** in its look-up table and finds the address of **`$var`**. It can then access the storage container.

Backslash operator

- The address of a variable's storage container can be assigned to a different variable. The backslash operator '\' applied to a scalar variable, returns the ***address*** of its storage container:

```
$var = 21;
```

Reference variables store addresses

- The variable **`$ref`** created in the preceding slide contains a reference to **`$var`**. It is a full-fledged variable, just like **`$var`**, with its own name and storage container, but inside its storage container is an address of another variable. (In some languages, it would be called a ***pointer variable***.)
- You can print the value stored in **`$ref`** using an ordinary print statement, and you will see that it is an address, in hexadecimal in the form

```
SCALAR(0x9025324)
```

Dereferencing references

- To access the storage container referenced by the reference variable, you need to ***dereference*** the variable.
- As I said before, when Perl sees a variable like **`$var`**, it looks up **`var`** in its look-up table to get the address of its storage container. The '**`$`**' tells Perl to look for either a variable name or something that evaluates to an address.
- If **`$ref`** contains an address, it follows that **`$$ref`** is the value in the variable it refers to. This takes some getting used to!
- In short, to dereference a reference, put the **`$`** before it.

Examples

```
$var = 21;          # $var has 21
$ref = \$var;       # $ref has address of $var
$newref = $ref;     # $newref also has address of $var
$x   = $$ref + 1;   # $x = 22
$y   = $$newref;    # $y = 21
$ref = \$y;         # $ref has address of $y
print $$ref -1;     # prints 20 because $$ref == 21
```

References to arrays

- A variable can store a reference to any kind of object that has an address. Later we will see examples of this. Now you are ready to see how references to arrays can be used.

```
@fibs = (1, 1, 2, 3, 5);
$rfibs = \@fibs;  # $rfibs has address of @fibs
```

- In this example, **`$rfibs`** is a reference variable that stores the address of an array. To dereference it and get the entire array, use

```
@$rfibs
```

- To access a single array element, use

```
$$rfibs[1]
```

Example

```
my @fibs = (1, 1, 2, 3, 5);
my $rfibs = \@fibs;
print "@$rfibs\n";       # prints 1 1 2 3 5
print $$rfibs[2], "\n"; # prints 2

# this pushes 8 (=3+5) onto end of array:
push(@$rfibs, $$rfibs[3] + $$rfibs[4]);

print "@$rfibs\n"; # prints 1 1 2 3 5 8
```

Summary

- **Literals are constants in your program.**
- **Scalars can be numeric, string, or references.**
- **Perl has 3 data classes: scalars, lists, and hashes.**
- **Variables are declared with my, and use $ for scalar, @ for arrays, and % for hashes.**
- **Assignment statements copy values into variables.**
- **Expressions are built from operands and operators.**
- **Arrays are list variables.**
- **Hashes are like look-up tables and are unordered.**
- **Perl uses context to determine how to evaluate variables and expressions at run-time.**
- **References are addresses.**