For Examples

```
# print 100..1 on separate lines
for ($i=0; $i < 100; $i++) {
    print 100-$i, “\n”;
}

# read n and print summation of 1..n
chomp($n = <STDIN>);
$sum = 0;
for ($i=1; $i <= $n; $i++) {
    $sum += $i;
}
print “summation of 1..$n is $sum.\n”;
```

For Examples (cont.)

```
# infinite loop (no condition means the
# condition defaults to be true each time)
for ( ; ; ) {
    ...
}
```

Lists and Arrays

- A list in Perl is an ordered collection of scalar data.
- An array in Perl is a variable that contains a list.
- Each element of a list can contain an independent scalar value, which can be a number or a string.

List Literals

- Can represent a list of values in Perl.
- General form.

  (<scalar_value>, <scalar_value>, ..., <scalar_value>)
- Examples:

```
(1, 3, 5)          # three numbers
(“cat”, “dog”)     # two strings
(1, “cat”, 0.5)    # can mix numbers and
                   # strings
(0, $a, $a+$b, 0)  # some values can be
                   # determined at run-time
( )                # can have an empty list
```

The *qw* Shortcut

- Can use the *qw* shortcut to create a list literal of *quoted words*.

```
# list literal below contains strings
# representing fruit
( “orange”, “apple”, “pear”, “lemon”, “grape”)

# below is a similar assignment, but requires
# fewer chars
qw/ orange apple pear lemon grape /

# can use other delimiters besides '/'
qw! orange apple pear lemon grape !

# can use delimiters with “left” and “right”
# characters
qw( orange apple pear lemon grape )
qw< orange apple pear lemon grape >
```

List Literals (cont.)

- Can use the range (..) operator to create list values by counting from the left scalar to the right scalar by ones.
- Examples:

```
(1..4)      # same as (1, 2, 3, 4)
(1.1..4.4)  # same as (1..4) since range
            # values have to be integers
(4..1)      # empty list since left value must
            # be less than the right value
(1,4..6,9)  # can be used along with explicit
            # list values
($m..$n)    # range values can be determined
            # at run time
```

Array Variables

- Arrays are declared using the '@' character.
- General form. Note that the size of the array is not specified.

```
my @arrayname;
```

- Examples:

```
my @a;          # array a
my @nums;       # array of numbers
my @strings;    # array of strings
```

Array vs. Scalar Names

- Easy way to remember names:

  $ looks like an S: $calar

  @ looks like an a: array

- Scalar and array names are in different name spaces. Could reuse the same names, but it is not recommended.

```
$b = $b[0];  # Assigns array element
             # $b[0] to scalar $b
             # The above code is confusing!
```

Accessing Array Elements

- Accessing array elements in Perl has similar syntax to accessing array elements in C.
- General form. The '$' is used since you are referring to a specific scalar value within the array. The expression is evaluted as an integer value. The first index of every array is zero.

  $arrayname[<expression>]

Examples of Accessing Array Elements

```
$a[0] = 1;          # can assign numeric
                    # constants
$s[1] = “Report”;   # can assign string literals
print $m[$i];       # can use a scalar variable
                    # as an index
$a[$i] = $b[$i];    # can copy one element to
                    # another
$a[$i] = $a[$j];    # another example
$a[$i+$j] = 0;      # can index by an expression
$a[$i]++;           # incrementing $a[$i] by one
```

Assigning List Literals

- Can assign list literals to arrays or scalars.

```
($a, $b, $c) = (1, 2, 3);    # $a=1; $b=2; $c=3;
($m, $n) = ($n, $m);         # can perform swaps
@nums = (1..10);             # can update entire arrays
                             # $nums[0]=1; $nums[1]=2; ...
($x, $y, $z) = (0, 1);       # $x=1; $y=2; $z=undef;
@t = ();                     # array with no elements
($a[0], $a[1]) = ($a[1], $a[0]);
                             # another swap
@fruit = (“pear”, “apple”); # fruit has two elements
@fruit = qw/ pear apple /;   # similar assignment
```

Accessing Entire Arrays

- Entire arrays can sometimes be accessed. Use @arrayname instead of $arrayname[...].

```
@x = @y;             # copy array y to array x
@y = 1..1000;        # range oper does not have
                     # to be inside parentheses
@lines = <STDIN>;    # read all lines of input
                     # $lines[0]=<STDIN>;
                     # $lines[1]=<STDIN>;
                     # ...
print @lines;        # print all array elements
```

Printing Entire Arrays

- Can print an entire array at once.

```
@fruit = ( “apple”, “orange”, “pear” );
print @fruit, “\n”; # prints “appleorangepear”
```

- Can print all array elements separated by spaces.

```
print “@fruit\n”;   # prints “apple orange pear”
```

Using the Array Name in a Scalar Context

- Using the array name when assigning it to a scalar or with a scalar operator results in the number of values being returned. It will not give a warning.

```
@array1 = (“cat”, 2, “dog”, 1, “hamster”, 3);
@array2 = @array1;  # copies array1 to array2
$m = @array2;       # $m = 6;
$n = $m + @array2;  # $n = 12;
```

Using a Scalar in a List Context

- Assigning a scalar to an array will result in the array containing a one element list.'

```
$m = 1;
@array = $m;       # @array = ( 1 );
@fruit = “apple”;  # @fruit = ( “apple” );
@array = undef;    # @array = ( undef );
@array = ( );      # Empties the array.
```

Size of Arrays

- Perl arrays can be of arbitrary size, provided there is enough memory to hold it. The number of elements can vary during run-time.

```
my @fruit;          # at this point @fruit has no
                    # elements
...
$fruit[0]=”apple”;  # now @fruit has one element
$fruit[1]=”orange”; # now @fruit has two elements
$fruit[99]=”mango”; # now @fruit has 100 elements
                    # $fruit[2]..$fruit[98] have
                    # undef values
```

The Last Element Index

- $#*arrayname* contains the current last element index, which is one less than the number of elements.

```
# Can be used to iterate through the array
# elements.
for ($i=0; $i <= $#fruit; $i++) {
    print $fruit[$i], "\n";
}

# Can be used to resize an array.
$a[99] = $i;  # assigns value to 100th
              # element of @a
...
$#a = 9;      # now @a has only 10 elements
```

Using Negative Array Indices

- Can use negative array indices to access elements from the end of the array.

```
print $a[-1]; # print the last element of @a
              # similar to using $a[$#a]
print $a[-2]; # print the 2nd to last
              # element of @a
```

Push and Pop Operators

- Arrays are often used like a stack, so there is support for push and pop operations.
- The *push* operator takes two arguments:
  - an array
  - value to be pushed, which can be a list value
- The *pop* operators takes one argument:
  - an array

Push and Pop Examples

```
push @nums, $i;     # same as "$nums[++$#nums] = $i;"
push @a, "end"      # adds "end" as a new element
push(@a, 1..5)      # assigns 1..5 as 5 new elements
push(@a,@b)         # adds @b elements at the end of @a
push @a, (1, 2, 3) # adds 1..3 as new elements to the
                    # end of @a
print pop @a;       # same as "print $a[$#a]; $#a -= 1;"
pop @a;             # same as "$#a -= 1;"
push @b, pop @a;    # pops $a[$#a] and pushes it onto @b
@a = ( );           # makes @a become empty
$b = pop @a;        # $b now contains undef
```

Shift and Unshift Operators

- The *shift* and *unshift* operators are analogous to the *pop* and *push* operators, except they work on the first instead of the last element.
- *Shift*, like in the shell for the command line arguments, returns the first element of an array and shifts the other elements over to the *left*.
- *Unshift* adds a value to an array by shifting the current elements to the *right* and assigning the new value to the first element.

Shift and Unshift Examples

```
@a = (“cat”, 4, “dog”); # @a now has 3 elements
$b = shift @a;          # $b == “cat” &&
                        # @a == (4, “dog”)
$c = shift @a;          # $c == 4 && @a == (“dog”)
$d = shift @a;          # $d == “dog” && @a == ( )
$e = shift @a;          # $e == undef && @a == ( )
unshift @a, 1;          # @a == (1)
unshift @a, (“cat”, “dog”);
                        # @a == (“cat”, “dog”, 1)
```

Foreach Control Structure

- The *foreach* control structure is used to process an entire array or list.
- General form. The $scalar gets assigned one value of the list or array for each iteration.

```
foreach $scalar (<list_or_array>) {
        <one_or_more_statements>
}
```

Foreach Examples

```
# prints each element of the array nums,
# one per line
foreach $num (@nums) {
    print $num, “\n”;
}

# pushes items in the list onto the fruit array
foreach $item (qw/ apple orange pear grape /) {
    push @fruit, $item;
}
```

Perl's Default Variable

- $_ is Perl's default variable and is used as a shortcut to reduce the number of characters typed. It is used as a default when reading input, writing output, and as a default for the foreach control structure.

```
while (<stdin>) {  # Reads into $_ by default.
    print;         # Prints from $_ by default.
}

$sum = 0;
foreach (@nums) {  # Assigns to $_ by default.
    $sum += $_;
}
```

Input from the Diamond Operator

- Reading input from the <> operator causes programs to read from standard input when there are no command line arguments or from files specified on the command line.
- Allows Perl programs to mimic the behavior of Unix utilities. One difference is that the list of files specified on the command line are treated as a single file that is concatenated together.

Example of Input from <>

```
# mimics the cat Unix utility
while ($line=<>) {
   print $line;
}

# can invoke by redirecting from standard input
cat.pl < input.txt

# can invoke by passing arguments on the
# command line
cat.pl input1.txt input2.txt > output.txt
```

The @ARGV Array

- The @ARGV array contains the strings representing the command line arguments at the start of the execution.
- Can process other command line options by shifting them from the @ARGV array before the first <> operation is performed.
- Note that $ARGV[0] contains the first command line argument, not the name of the Perl file being invoked.