MC4081

INTRO TO PERL PROGRAMMING

Software Support & Training

Information Systems

MC4081 INTRO TO PERL PROGRAMMING

Target Audience

Micron Team Members who use or will be using Perl

Course Goal

After completing this course, team members will be able to recognize and apply the basic elements of Perl programming.

Prerequisite

MC4024 Intro to UNIX Basics
MC4061 Programming Fundementals
MC4040 Exceed Basics (highly recommended)

Course Hours

Six hours

Total Sessions

Two (a homework assignment will be given out during the first session)

Test-Out Available

None at this time

Objectives

Other Information

PERL HISTORY

Early History

1987: Perl 1.000 is unleashed, written by Larry Wall, Perl's creator and chief architect, implementor, and maintainer. Perl is short for "Practical Extraction and Report Language" or "Pathologically Eclectic Rubbish Lister."

1988: Perl 2.000 is released with some enhancements from Perl 1.

1989: Perl 3.000 is released and distributed by Larry Wall for the first time under the GNU General Public License. The goal of the General Public License is "to make sure that you have the freedom to distribute copies of free software, that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know that you can do these things."

1991: Perl 4.000 is released.

Perl 5

1994: The much anticipated Perl 5.000 is unveiled, which is a complete rewrite of Perl.

1995: Rasmus Lerdorf created a Perl CGI script which inserted a tag into the HTML code of his page, and collected the information on the visitors to his website.

Today

From Tom Christiansen's and Nathan Torkington's excellent Perl FAQ:

> Perl is a high-level programming language with an eclectic heritage written by Larry Wall and a cast of thousands. It derives from the ubiquitous C programming language and to a lesser extent from sed, awk, the Unix shell, and at least a dozen other tools and languages. Perl's process, file, and text manipulation facilities make it particularly well-suited for tasks involving quick prototyping, system utilities, software tools, system management tasks, database access, graphical programming, networking, and world wide web programming. These strengths make it especially popular with system administrators and CGI script authors, but mathematicians, geneticists, journalists, and even managers also use Perl.

USING EXCEED

The **Exceed X** server transforms your computer into a fully functional X Window terminal, allowing you to access UNIX-based applications (X clients) from within the familiar Microsoft Windows environment.

Installing Exceed

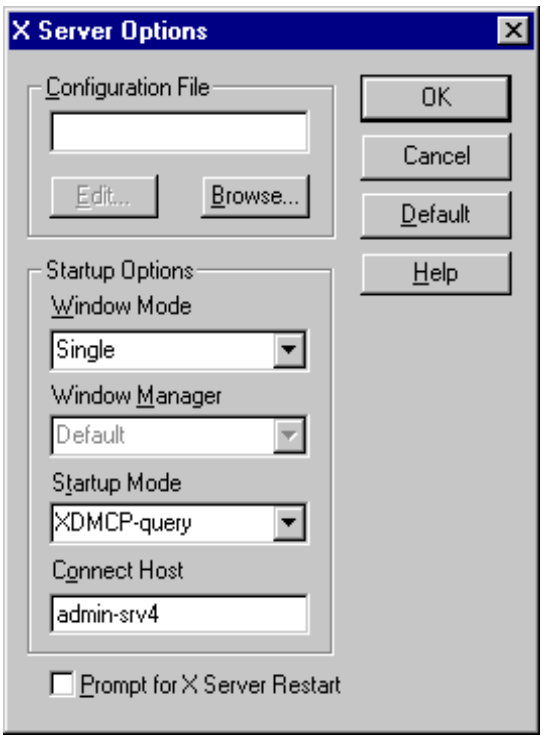
To install Exceed:

1. Go to **START** > **SERVER APPS** > **SERVER APPS SETUP.**
2. Click **OK.**
3. Click **SPECIAL USE.**
4. Click **EXCEED 6.2 (UNIX).**
5. Click **OK.** - The PC will have to be rebooted.

How to Start a Xsession

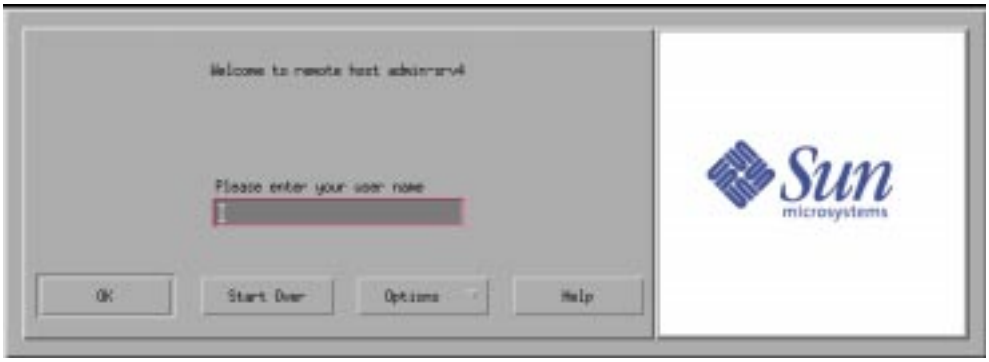
Launch Xsession from Exceed program group

1. Go to **START** > **PROGRAMS** > **EXCEED** > **XSESSION.**
2. Click on the **OPTIONS** button in the X Session Starter window.
3. Select **SINGLE** for the **WINDOW MODE.**
4. Select **XDMCP-QUERY** for the **STARTUP MODE.**
5. Enter the host name in the **CONNECT HOST** text box (e.g. hobbes, admin-srv94).



6. Click **OK.**
7. Click the **RUN** button in the X Session Starter window. Exceed launches the CDE (Solaris) login window.

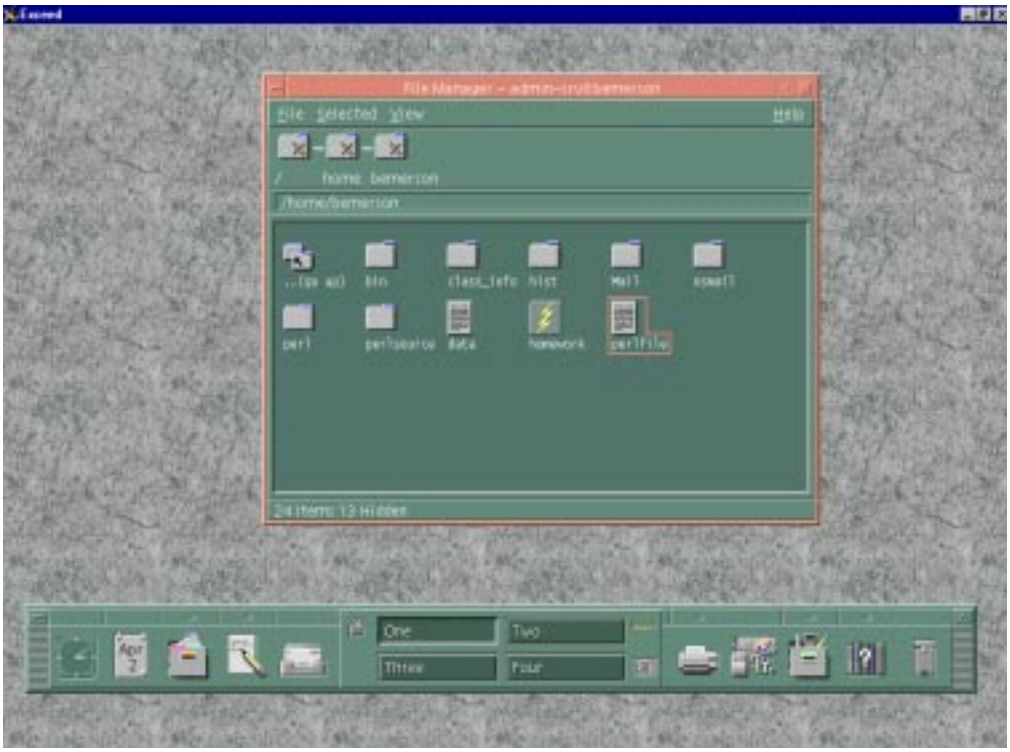
8. Enter your **UNIX LOGIN NAME** and press **ENTER.**



9. Enter your **UNIX PASSWORD** and press **ENTER**. Wait until the CDE (Common Desktop Environment) loads.

See “Appendix 3 - Requesting a UNIX Account” on page 48 for information on obtaining a UNIX account.

Exceed Main Screen



Opening Terminal

1. Select **APPLICATION MANAGER** from the **MAIN PANEL CONTROL**.

2. Select **DESKTOP_APPS.**
3. Double-click on **TERMINAL.**

DISCUSS AND IDENTIFY BASIC PERL CONCEPTS

Perl Programs

- A shell script is a sequence of shell commands in a text file. The text file is made executable by using the *chmod* command and then the name of the file is typed at the command prompt. In the same way, a Perl program is a group of Perl statements and definitions inserted into a text file. The file is made executable and the file name is typed in at the command prompt. The main difference is that the Perl file has to indicate that it is a Perl program. To designate the file as a Perl program, place the following code at the first line of every new text file:

  ```
  #!/usr/local/bin/perl
  ```

  It is important that this line is the first line of every Perl program written.

- Perl is mainly a free format language where whitespace (spaces, tabs, newlines, returns) is optional.

- All statements in Perl must end with a semi-colon (;). You can think of the semi-colon as a punctuation mark, like a period in English.

- Although virtually any Perl program can be written entirely on one line, usually a Perl program is indented, with nested parts of statements indented more than the surrounding parts. There will be examples of typical indentation style covered in this class.

- All comments in Perl are preceded by a pound sign (#) until the end of the line.

CREATE A PROGRAM USING SCALAR DATA

What is Scalar Data?

A scalar is the simplest kind of data that Perl manipulates. A scalar is either a number (like 8 or 3.25e20) or a string of characters (like *hello* or this manual). Numbers and strings may appear to be very different things, but Perl uses them almost interchangeably. A scalar value can be acted upon with operators (like minus and greater than), usually resulting in a scalar result. A scalar value can be stored into a scalar variable.

Numbers

Although a scalar can be either a number or a string, it is useful to look at them separately for now. In Perl, you can specify both integers (8 or 365) and floating-point numbers (4.145 or 2.35 times 5^2).

A *literal* is the way a value is represented in the text of a Perl program. Literals are the way data is represented in the source code of your program as input to the Perl compiler. For example: 8.5 (8 and about a half), 4.25e24 (4.25 times 10 to the 24th power).

On the contrary, integer literals are simple, such as: 8, 42, -4. You should not start your number with a 0, because Perl supports octal and hexadecimal (hex) literals. Octal numbers start with a leading 0, and hex numbers start with a leading 0x.

Strings

Strings are simply sequences of characters (like *hello*). Each character is an 8-bit value from the entire 256 character set. The shortest possible string has no character; the longest string could fill all of your available memory. Typical strings are printable sequences of letters and digits and punctuation in the ASCII 32 to ASCII 126 range.

Like numbers, strings have a literal representation. There are two types of literal strings: *single-quoted strings* and *double-quoted strings.*

Single-Quoted Strings

A *single-quoted string* is a sequence of characters enclosed in single quotes. The single quotes are not part of the string itself but are there to let Perl identify the beginning and the ending of the string. Any character between the single quote marks is permitted inside a string. The two exceptions: to place a single quote inside of a single-quoted string, precede it by a backslash. To place a backslash into a single-quoted string, precede the backslash by a backslash. Some examples:

```
'hello'          # five characters: h, e, l, l, o
'Larry\'s'       # six characters: L, a, r, r, y, single-quote, s
'here\\there'    # here\there
```

Double-Quoted Strings

A *double-quoted string* is a sequence of characters, like a single-quoted string, but enclosed in double quotes. Now the backslash character has full power to specify certain control characters or any character at all through octal and hex representations. Some examples:

```
"hello world\n"   # hello world, and a newline character
"hey \177"        # hey, space, and the delete character (octal 177)
"time\tspace"     # time, a tab, and space
```

The backslash can precede many different characters to mean different things (usually called a *backslash escape*).

Table 1 provides a complete list of double-quoted string escapes:

Table 1: Double-Quoted String Representations

Construct	**Meaning**
\n	Newline
\r	Return
\t	Tab
\f	Formfeed
\b	Backspace
\a	Bell
\e	Escape
\007	Any octal ASCII value (here, 007 = bell)
\x7f	Any hex ASCII value (here, 7f = delete)
\cC	Any "control" character (here, CNTL-C)
\\	Backslash
\"	Double quote
\l	Lowercase next letter
\L	Lowercase all following letters until \E
\u	Uppercase next letter
\U	Uppercase all following letters until \E
\Q	Backslash-quote all nonalphanumerics until \E
\E	Terminate \L, \U, or \Q

Another feature of double-quoted strings is that they are *variable interpolated*, meaning that scalar and array variables within strings are replaced with their current values when the strings are used. See "Interpolation of Scalars into Strings" on page 15 for more detail.

Scalar Operators

An operator produces a new value (the result) from one or more other values (the operands). For example, - is an operator because is takes two numbers (the operands, like 8 and 3), and produces a new value (5, the result). An operator expects either numeric or string operands, or possibly a combination of both.

Operators for Numbers

Perl provides the classic addition, subtraction, multiplication, and division operators. For example:

```
8 + 3            # 8 plus 3, or 11
4 - 1            # 4 minus 1, or 3
10 * 2.5         # 10 times 2.5, or 25
33 / 11          # 33 divided by 11, or 3
```

Perl also provides the *exponentiation* operator which is represented by the double asterisk, like 2**4, which is two to the fourth power, or 16. In addition, Perl supports a *modulus* operator. The value of the expression 10 % 3 is the remainder when 10 is divided by 3, which is 1.

The logical comparison operators are < <= == >= > !=. These operators compare two values numerically, returning a value of either *true* or *false*. For example, 4 > 3 returns true because four is greater than three, while 8 != 8 returns false because it is not true that 8 is not equal to 8. Think of the return value for true as a non-zero number and the return value for false as zero.

Operators for Strings

String values can be concatenated with the "." operator. Concatenation is to arrange strings of characters into a chained list. The resulting longer string is then available for further computation or to be stored into a variable. For example:

```
"Mic" . "ron"        # same as "Micron"
"Hey" . " " . "you"  # same as "Hey you"
```

Another set of operators for strings are the string comparison operators. The operators compare ASCII values of the characters of the strings as usual, see "Appendix 2 - ASCII Character Set" on page 47. The complete set of comparison operators (for both numbers and strings) is outlined in Table 2:

Table 2: Numeric and String Comparison Operators

Comparison	**Numeric**	**String**
Equal	= =	eq
Not equal	!=	ne
Less than	<	lt
Greater than	>	gt
Less than or equal to	<=	le
Greater than or equal to	>=	ge

In addition, string operator is the string repetition operator, consisting of the single lowercase letter x. This operator takes its left operand (a string), and makes as many concatenated copies of that string as indicated by its right operand (a number). For example:

```
"dog" x 4        # is "dogdogdogdog"
(4+3) x 5        # is "7" x 5, which is "77777"
```

Operator Precedence and Associativity

Operator precedence defines how to resolve the ambiguous case where two operators are trying to operate on three operands. For example, in the expression 4+2*5, do we do the addition first or the multiplication first? Luckily, Perl uses common mathematical definition, performing the multiplication first in this example. Based upon this, we say multiplication has a higher precedence than addition.

The operator precedence can be overridden by using parentheses. Anything in parentheses is completely computed before the operator outside of the parentheses is applied.

While precedence is intuitive for addition and multiplication, problems arise when faced with an expression like string concatenation compared with exponentiation. The best way to resolve this is to consult the chart shown in Table 3:

Table 3: Associativity and Precedence of Operators: Highest to Lowest

Associativity	**Operator**
Left	The "list" operators (leftward)
Left	-> (method call, dereference)
Nonassociative	++ -- (autoincrement, autodecrement)
Right	** (exponentiation)
Right	! ~ \ + - (logical not, bit-not, references operator, unary plus, unary minus)
Left	=~ !~ (matches, doesn't match)
Left	* / % x (multiply, divide, modulus, string replicate)
Left	+ - . (add, subtract, string concatenate)
Left	<< >>
Nonassociative	Named unary operators (like chomp)
Nonassociative	< > <= >= lt gt le ge
Nonassociative	== != <=> eq ne cmp
Left	& (bit-and)
Left	| ^ (bit-or, bit-xor)
Left	&& (logical and)
Left	|| (logical or)
Nonassociative	.. ... (noninclusive and inclusive range)
Right	?: (if-then-else)
Right	= += -= *=, etc. (assignment and binary-assignment)
Left	, => (comma and comma-arrow)
Nonassociative	List operators (rightward)
Right	not (logical not)
Left	and (logical and)
Left	or xor (logical or, logical xor)

In the chart above, any given operator has higher precedence than those listed below it, and lower precedence than all of the operators listed above it.

Operators at the same precedence level resolve according to rules of associativity instead. Just like precedence, associativity resolves the order of operations when two operators of the same precedence compete for three operands. For example:

```
4 ** 2 ** 3       # 4 ** (2 ** 3), or 4 ** 8, or 65,536
28 / 4 * 3        # (28/4)*3, or 21
```

In the first case, the ** operator has right associativity, so the parentheses are implied on the right. The * and / operators have left associativity, putting the implied parentheses on the left.

Conversion Between Numbers and Strings

If a string value is used as an operand for a numeric operator (like +), Perl automatically converts the string to its equivalent numeric value. Trailing nonnumerics and leading whitespace are ignored, so " 128.4opps" converts to 128.4. Something that is not a number at all, like "opps", converts to zero with a warning.

Similarly, if a numeric value is given when a string value is needed, the number value is expanded into whatever string would have been printed for that number. For example:

```
"S" . (2 * 300)  # same as "S" . 600, or "S600"
```

In other words, you don't need to worry about whether you have a number or a string (in most cases). Perl will perform all of the conversions for you.

Scalar Variables

A variable is the name for a container that stores one or more values. The name of the variable is constant throughout the program, but the value(s) contained in the variable usually change throughout the execution of the program.

A scalar variable holds a single value (a number, a string, or a reference). Scalar variable names begin with a dollar sign followed by a letter, and then possibly more letters, numbers, or underscores. Upper- and lowercase letters are distinct: the variable `$Z` is a different variable from `$z`. You should try to select variable names that mean something regarding the value of the variable. For example, `$gr81` is not as descriptive as `$secret_word`.

Scalar Operators and Functions

The most common operation on a scalar variable is assignment, which is the way to assign a value to a variable. The Perl assignment operator is the equal sign, which takes a variable name on the left side and gives it the value of the expression on the right side. For example:

```
$a = 8;           # assign $a the value 8
$b = $a + 4;      # assign $b the current value of $a plus 4 (12)
$b = $b / 2;      # assign $b the value of $b divided my 2 (6)
```

The last line uses the `$b` variable twice: once to get its value (on the right side of the =), and once to define where to put the computed expression (on the left side of the =). This is legal and common in Perl.

Binary Assignment Operators

Expressions like `$z = $z + 8` (where the same variable appears on both sides of an assignment) occur regularly in Perl, and there is a shorthand for the operation of altering a variable: the *binary assignment operator*. Almost all binary operators that compute a value have a corresponding binary assignment form with an appended equal sign. For example:

```
$a = $a + 4;      # without the binary assignment operator
$a += 4;          # with the binary assignment operator
$z = $z * 8;      # without
$z *= 8;          # with
```

Another common assignment operator is the string concatenate operator:

```
$word = $word . " ";  # append a space to $word
$word .= " ";         # same thing but with the assignment operator
```

Autoincrement and Autodecrement

Perl also has a way to shorten `$a += 1` even further. The ++ operator (known as the *autoincrement* operator) adds one to its operand and returns the incremented value. For example:

```
$a += 1;          # with assignment operator
++$a;             # with prefix autoincrement
$z = 8;
$x = ++$z;        # $z and $x are both 9 now
```

As shown above, the ++ operator is being used as a *prefix* operator (it appears to the left of its operand). The autoincrement can also be used in a suffix form (to the right of its operand). In this instance, the result of the expression is the old value of the variable before the variable is incremented. For example:

```
$y = 12;
$w = $y++;        # $w is 12, but $y is now 13
```

The autodecrement operator (--) is similar to the autoincrement operator, but it subtracts one instead of adding one. The autodecrement operator also has both the prefix and suffix forms. For example:

```
$a = 8;
--$a;             # $a is now 7
$b = $a--;        # $b is 7, and $a is now 6
```

The chop and chomp Functions

A useful built-in function in Perl is chop. This function takes a single argument within its parentheses and removes the last character from the string value of that variable. For example:

```
$a = "micronx";
chop ($a);       # $a is now "micron"
```

Notice that the value of the argument is altered here, showing the requirement for a scalar variable, rather than simply a scalar value.

When you chop a string that has already been chopped, another character disappears. For example:

```
$b = "Micron\n";
chop ($b);       # $b is now "Micron"
chop $b;         # $b is now "Micro"
```

If you are not sure whether the variable has a newline on the end, you can use the chomp operator, which removes only a newline character. For example:

```
$b = "Micron\n";
chomp ($b);      # $b is now "Micron"
chomp $b;        # $b is still "Micron", no change
```

Interpolation of Scalars into Strings

When a string literal is double-quoted, it is subject to variable interpolation. This means that the string is scanned for possible scalar variable names (a dollar sign followed by letters, digits, and underscores). When a variable reference is found, it is replaced with its current value. For example:

```
$a = "micron";
$b = "We all work for $a tech"; # $b is now "We all work for
                                # micron tech"
$c = "This is $wrong";          # $c is now "This is "
```

To prevent the substitution of a variable with its value, you must either change that part of the string so that is appears in single quotes, or precede the dollar sign with a backslash, which turns off the dollar sign's significance. For example:

```
$dram = 'wow';
$sram = "testing of " . '$dram';   # literally: 'testing of $dram'
$sram = "testing of \$dram";       # same as above
```

<STDIN> as a Scalar Value

Now we will consider how to get a value into a Perl program. Every time you use <STDIN> in a place where a scalar value is expected, Perl reads the next complete text line from *standard input*, and uses that string as the value of <STDIN>.

The string value of <STDIN> typically has a newline character on the end of it. Usually, you will want to discard the newline immediately. The chomp function is the best way to accomplish this. For example:

```
$b = <STDIN>;     # get the text from the user
chomp $b;         # discard the newline (\n)
```

A common way these two lines are combined is:

```
chomp($b = <STDIN>);
```

The assignment inside the parentheses continues to refer to $b, even after it has been given a value with <STDIN>. Thus, the chomp function is working on $b.

Output with print

To get input we use <STDIN> and to get output we use the print function. This function takes the values within its parentheses and puts them out onto standard output. For example:

```
print “hello world\n”;     # hello world is printed followed by a
                           # newline
```

The Undefined Value

If you use a scalar variable before you assign it a value you will get the undef (undefined) value. This value looks like a zero when used as a number, or the zero-length empty string when used as a string. Many operators (like <STDIN>) return undef when the arguments are out of range or don’t make sense.

Exercise 1: welcome

1. Open a Terminal window in Exceed
2. At the command line, type: **mkdir perl**
3. Press **ENTER**
4. Open Text Editor in Exceed
5. Type this code into Text Editor:

```
#!/usr/local/bin/perl

print "What is your name: ";
$name = <STDIN>;
chomp ($name);
print "Welcome to Perl, $name!\n";
```

6. Click **FILE/SAVE**
7. Select the **perl** Folder
8. Name the file: **welcome**
9. Click **OK**
10. In the Terminal window at the command line type: **cd perl**
11. Press **ENTER**
12. At the command line type: **chmod a+x welcome**
13. Press **ENTER**
14. At the command line type: **welcome**
15. Press **ENTER** to execute the welcome program

Exercise 2: circ

1. Write a program to prompt for and accept a radius from the user so you can compute the circumference of a circle. The circumference is 2Π multiplied by the radius, or 2 multiplied by 3.141592654 multiplied by the radius.
2. Follow the steps in Exercise 1 and name this program **circ.**

Exercise 3: x2

1. Write a program that prompts for and reads two numbers, and prints out the result of the two numbers multiplied together.
2. Follow the steps in Exercise 1 and name this program **x2**.

CREATE A PROGRAM USING ARRAYS AND LIST DATA

A List or Array

A *list* is ordered scalar data. An array is a variable that holds a list. Each element of the array is a separate scalar variable with an independent scalar value. These values are ordered, which means they have a particular sequence from the lowest to the highest element. Arrays can have any number of elements. The smallest array has no elements, while the largest array can fill all available memory.

Literal Representation

A list literal is the way you represent the value of a list within your program. A list literal consists of comma-separated values enclosed in parentheses. These values form the elements of the list. For example:

```
(7,8,9)          # array of three values 7, 8, and 9
(“micron”,.15)   # array of two values, “micron” and .15
```

The elements of a list do not need to be constants; they can be expressions that will be re-evaluated each time the literal is used. For example:

```
($cool,”whip”)  # two values: the current value of $cool and “whip”
($z+$x,$a+$b)   # two values
```

The empty list (array with no elements) is represented by an empty pair of parentheses. For example:

```
()               # the empty list (no elements)
```

An item of the list literal can include the *list constructor operator*, indicated by two scalar values separated by two consecutive periods. This operator creates a list of values starting at the left scalar value up through the right scalar value, incrementing by one each time. For example:

```
(5 .. 9)         # same as (5, 6, 7, 8, 9)
(12 .. 15,18,19) # same as (12, 13, 14, 15, 18, 19)
($a .. $z)       # range determined by current values of $a and $z
```

If the right scalar value is less than the left scalar value the result will be an empty list. Perl doesn’t allow you to count down by switching the order of values.

List literals with large amounts of short text strings will start to look messy with all of the quotes and commas. Perl provides a shortcut known as the “quote word” function, which creates a list from the non-whitespace parts between the parentheses. For example:

```
@u2 = (“bono”,“edge”,“larry”,“adam”,“boy”,”war”);  # messy
@u2 = qw(bono edge larry adam boy war);            # much better
```

Array Variables

An array variable holds a single list value (zero or more scalar values). Array variable names are similar to scalar variable names, differing only in the initial character, which is an "at sign" (@) as opposed to a "dollar sign" ($). For example:

```
@micron              # the array variable @micron
@sst_training_group  # a longer array variable
```

The array variable `@micron` is completely unrelated to the scalar variable `$micron`.

The value of an array variable that has not yet been assigned is (), an empty list.

Array Operators and Functions

Array functions and operators act on entire arrays. Some will return a list, which can then either be used as a value for another array function, or assigned into an array variable.

Assignment

Possibly the most important array operator is the *array assignment operator*, which assigns an array variable a value. The array assignment operator is an equal sign, exactly like the scalar assignment operator. Perl will determine whether the assignment is a scalar assignment or an array assignment by discerning whether the assignment is to a scalar or an array variable. For example:

```
@micron = (.18,.15,.13);   # the @micron array gets a three-element
                           # literal
@test = @micron;       # the three-elements are now copied to @test
```

If a scalar value gets assigned to an array variable, the scalar variable becomes the single element of an array. For example:

```
@zoo = 1;         # 1 is promoted to the list (1) automatically
```

Array variable names can appear in a literal list. When the value of the list is computed, Perl will replace the names with the current values of the array. For example:

```
@micron = qw(efficient innovative);
@team = (1,2,@micron,3,4);   # @team is now (1,2,"efficient",
                             # "innovative",3,4)
@team = (7, @team);          # puts a 7 in front of @team
@team = (@team,"agile");     # adds "agile" to the end of the array
```

If a list literal contains only variable references, the list literal can also be treated as a variable. This kind of list literal can be used on the left side of an assignment. Each scalar variable in the list literal takes on the corresponding value from the list on the right side of the assignment. For example:

```
($a,$b,$c) = (1,2,3);    # this gives 1 to $a, 2 to $b, 3 to $c
($a,$b) = ($b,$a);       # this swaps $a and $b
($z,@oh) = ($a,$b,$c);   # this gives $a to $z and ($b,$c) to @oh
($x,@oh) = @oh;  # this removes the first element of @oh to $x and
                 # makes @oh = ($c) and $x = $b
```

When the number of elements assigned do not match the number of variables to hold the values, the excess values (right side of =) are discarded, and the excess variables (left side of =) are given the undef value.

An array variable appearing in the array literal must be last, because the array variable is "greedy" and will consume all remaining values.

If an array variable is assigned to a scalar variable, the number assigned is the length of the array. For example:

```
@micron = (1,2,3);  # initialize @micron
$z = @micron;       # $z gets 3, the current length of @micron
```

Array Element Access

Until now, we have been treating the array as a whole. Perl also provides a traditional subscripting function to access an array element by numeric index. For the subscripting function, array elements are numbered using sequential integers, starting at zero and increasing by one for each array element. The first element of the `@micron` array is accessed as `$micron[0]`. Please note that the `@` on the array name becomes a `$` on the element reference. This is because accessing an element of the array identifies a scalar variable (part of the array), which can either be assigned to or have its current value used in an expression. For example:

```
@micron = (3,4,5);
$fab = $micron[0];  # this gives 3 to $fab (the first element of
                    # @micron)
$micron[0] = 1;     # now @micron = (1,4,5)
```

Here are some more examples of accessing elements of an array:

```
$probe = $micron[1]; # this gives 4 to $probe
$micron[2]++;        # this increments the third element of @micron
$micron[1] -= 2;     # this subtracts 2 from the second element
```

Accessing a list of elements from the same array is known as a *slice*, and is used often enough that there is a special representation for it. For example:

```
@micron[0,1];     # this is the same as ($micron[0], $micron[1])
@micron[0,1] = @micron[1,0];   # swaps the first two elements
@micron [0,1,2] = @micron [2,2,2];   # makes all 3 elements like
                                     # the 3rd
@micron[0,2] = (7,8);     # this changes the first and third value
                          # to 7 and 8
```

Please notice that slices use an @ prefix instead of a $. This is because you are creating an array variable by selecting part of the array rather than a scalar variable accessing just one element.

The index values in the above examples are literal integers, but the index can also be any expression that returns a number, which is then used to select the appropriate elements. For example:

```
@micron = (4,5,6);
$fab = 2;
$test = $micron[$fab];  # same as $micron[2], or the value 6
$is = $micron[$fab-1];  # $is gets $micron[1], or 5
```

If you access an array element beyond the end of the array, the undef value is returned. For example:

```
@micron = (4,5,6);
$test = $micron[8];     # $test is now undef
```

Assigning a value beyond the end of the current array automatically extends the array and gives undef to all intermediate values. For example:

```
@micron = (4,5,6);
$micron[3] = "tech";  # @micron is now (4,5,6,"tech")
$micron[6] = "hi"; # @micron is now (4,5,6,"tech",undef,undef,"hi")
```

The push and pop Functions

One common use of an array is as a stack of information, where new values are added to and removed from the right-hand side of the list. These operations occur often enough to have their own special functions know as the push and pop functions. For example:

```
push(@arr,$new);       # like @arr = (@arr,$new)
$oldie = pop(@arr);    # removes the last element of @arr
```

The pop function will return undef if it is given an empty list.

The push function also accepts a list of values to be pushed. The values are pushed together onto the end of the list. For example:

```
@arr = (1,2,3);
push(@arr,4,5,6);     # @arr = (1,2,3,4,5,6)
```

The shift and unshift Functions

The push and pop functions affect the right side of a list. Likewise, the unshift and shift functions perform the corresponding actions on the left side of a list. For example:

```
unshift(@arr,$z,$y); # like @arr = ($z,$y,@arr);
$a = shift @arr);    # like ($a,@arr) = @arr;
@arr = (4,5,6);
unshift(@arr,1,2,3); # @arr is now (1,2,3,4,5,6)
$a = shift(@arr);    # $a is now 1, @arr is now (2,3,4,5,6)
```

As with pop, shift returns undef if given an empty array variable.

The reverse Function

The `reverse` function reverses the order of the elements of its arguments, returning the resulting list. For example:

```
@arr = (3,4,5);
@rev = reverse(@arr);   # @rev gets (5,4,3)
@rev = reverse(3,4,5);  # same result as example above
```

The argument list remains unaltered because the `reverse` function works on a copy. If you want to reverse an array already in place, you need to assign it back into the same variable. For example:

```
@rev = reverse(@rev);   # this gives @rev the reverse of itself
```

The sort Function

The `sort` function sorts its arguments as if they were single strings in ascending ASCII order. It returns the sorted list without altering the original list. For example:

```
@s = sort(qw(small medium large));  # @s gets “large”,“medium”,
                                    #“small”
@b = (1,2,4,8,16,32,64);
@b = sort(@b);           # @b gets 1, 16, 2, 32, 4, 64, 8
```

Variable Interpolation of Arrays

As with scalars, array values may be interpolated into a double-quoted string. A single element of an array will be replaced by its value. For example:

```
@micron = (“test”,”probe”,”is”);
$a = 2;
$b = “Shaun works in $micron[1]”;   # “Shaun works in probe”
$b = “Laura works in $micron[$a]”;  # “Laura works in is”
```

A list of values from an array variable can also be interpolated. The simplest interpolation is an entire array, specified by giving the array name. The array elements are interpolated in sequence with a space character between them. For example:

```
print( “Some departments at Micron are @micron\n” );
    # Some departments at Micron are test probe is
```

It is also possible to select a portion of an array with a slice. For example:

```
print ( “Two large departments are @micron[0,1]\n” );
    # Two large departments are test probe
```

<STDIN> as an Array

As described previously, `<STDIN>` returns the next line of input in a scalar context. However, in a list context, it returns all remaining lines up to the end of a file. Each line is returned as a separate element of the list. For example:

```
@input = <STDIN>;   # read standard input in a list context
```

If a user types three lines, then presses CNTL-D (to indicate “end of file”), the array now contains three elements. Each element will be a string that ends in a newline, corresponding to the three newline-terminated lines entered.

Exercise 4: input

1. Type this code into Text Editor:

   ```
   #!/usr/local/bin/perl

   print ("Enter a list of strings (CNTL-D to end):\n");
   @array = <STDIN>;
   print (@array);
   ```

2. Click **FILE/SAVE**
3. Select the **perl** Folder
4. Name the file: **input**
5. Click **OK**
6. At the command line in the Terminal window type: **chmod a+x input**
7. Press **ENTER**
8. At the command line type: **input**
9. Press **ENTER** to execute the input program

Exercise 5: tupni

1. Write a program that reads a list of strings on separate lines and prints out the list in reverse order.
2. Follow the steps in Exercise 4 and name this program **tupni.**

CREATE A PROGRAM USING CONTROL STRUCTURES

All the programs we have looked at thus far are unconditional statements, which means they are executed sequentially, regardless of what is happening in the program. In some situations, however, you might want to have statements that are executed only when certain conditions are true. This is where Control Structures come in. Perl supports a variety of conditional statements including `if`, `if-else`, `if-elsif-else`, `while-until`, `for` and `foreach`, which will be covered in the following section.

Statement Blocks

A statement block is a sequence of statements that are enclosed in matching curly braces. For example:

```
{
     first_statement;
     second_statement;
     third_statement;
     ...
     final_statement;
}
```

Perl will execute each statement in sequence, from the first to the final.

The if/unless Statement

The `if` statement construct takes a control expression and a block. It can optionally have an `else` followed by a block as well. For example:

```
if (an_expression)  {
     true_statement1;
     true_statement2;
     true_statement3;
} else {
     false_statement1;
     false_statement2;
     false_statement3;
}
```

During execution, Perl evaluates the control expression. If the expression is true, the first block (`true_statements`) is executed. If the expression is false, the second block (`false_statements`) is executed instead.

How do we determine what is true and what is false? In Perl, the control expression is evaluated for a string value in scalar context. If this string is either the empty string or a string consisting of the single character “0” (zero), then the value is false. Everything else is true. For example:

```
0               # converts to “0”, false
1-1             # converts to 0, then to “0”, false
1               # converts to “1”, true
“ “             # empty string, false
“00”            # not “ “ or “0”, true (be careful)
“0.0000”        # not “ “ or “0”, true
undef           # equal to “ “, false
```

Here is an example of a complete `if` statement:

```
print “What is your age? “;
$age = <STDIN>;
chomp($age);
if ($age < 21) {
     print “Well, no wine for you!\n”;
} else {
     print “Old enough to enjoy a nice glass of wine!\n”;
}
```

It is possible to omit the `else` block, leaving just a “`then`” part. For example:

```
print “What is your age? “;
$age = <STDIN>;
chomp($age);
if ($age < 21) {
     print “Well, no wine for you!\n”;
}
```

You can leave off the “`then`” part and have just an `else` part, which is more natural than “do that if this is not true.” Perl does this with the `unless` variation. For example:

```
print “What is your age? “;
$age = <STDIN>;
chomp($age);
unless ($age < 21) {
     print “Old enough to enjoy a nice glass of wine!\n”;
}
```

Replacing `if` with `unless` is the same as saying “If the control expression is false, do....”

If you have more than two possible choices, add an `elsif` branch to the `if` statement. For example:

```
if (an_expression_one)  {
     one_true_statement1;
     one_true_statement2;
     one_true_statement3;
} elsif (an_expression_two)  {
     two_true_statement1;
     two_true_statement2;
     two_true_statement3;
} elsif (an_expression_three)  {
     three_true_statement1;
     three_true_statement2;
     three_true_statement3;
} else {
     false_statement1;
     false_statement2;
     false_statement3;
}
```

Each expression (an_expression one, two, and three) is computed in turn. If an expression is true, the corresponding branch is executed, and all remaining control expressions and corresponding statement blocks are skipped. If all expressions are false, the `else` branch is executed. You don't need to have an `else` block, but it is always a good idea. You can have as many `elsif` branches as you need.

The while/until Statement

Iteration is the repeated execution of a block of statements. In Perl, you can iterate using the `while` statement. For example:

```
while (an_expression) {
     statement_one;
     statement_two;
     statement_three;
}
```

To execute this `while` statement, Perl evaluates the control expression. If its value is true, the body of the `while` statement is evaluated once. This is repeated until the control expression becomes false, at which point Perl moves on to the next statement after the `while` loop. For example:

```
print "What is your age? ";
$age = <STDIN>;
chomp($age);
while ($age > 0) {
     print "Not so long ago, you were $age.\n";
     $age--;
}
```

Sometimes it is easier to say "until something is true." Replacing the `while` with `until` yields the desired effect. For example:

```
until (an_expression) {
     statement_one;
     statement_two;
     statement_three;
}
```

In both the `while` and the `until` form, the body statements are skipped entirely if the control expression is the starting termination value.

It is possible that the control expression never lets the loop exit. This is legal, and sometimes desired, and is not considered an error. For example, you may want a loop to repeat as long as you don't have an error, and then have some error-handling code following the loop.

The do {} while/until Statement

The `while/until` statement tests its condition at the top of every loop and before the loop is entered. If the condition was already false, the loop won't be executed at all.

There are times when you may want to test the condition at the bottom of the loop instead of the top. The `do {} while` statement doesn't test the expression until after executing the loop once. For example:

```
do {
     statement_one;
     statement_two;
     statement_three;
} while an_expression;
```

Perl executes the statements in the `do` block. When it reaches the end, it evaluates the expression for truth. If the expression is false, the loop is done. If it is true, then the whole block is executed one more time before the expression is checked again.

As with a normal while loop, you can invert the sense of the test by changing `do {} while` to `do {} until`. The expression is still tested at the bottom but its sense is reversed.

The for Statement

Another Perl iteration construct is the `for` statement. For example:

```
for ( initial_exp; test_exp; re-init_exp ) {
     statement_one;
     statement_two;
     statement_three;
}
```

To put it into terms that have already been covered:

```
initial_exp;
while (test_exp); {
     statement_one;
     statement_two;
     statement_three;
     re-init_exp;
}
```

In both cases, the `initial_exp` expression is evaluated first. This expression usually assigns an initial value to an iterator variable, but there are no restrictions on what it can contain. Then the `test_exp` expression is evaluated for truth or falsehood. If the value is true, the body is executed, followed by the `re-init_exp`. Perl will then re-evaluate the `test_exp`, repeating as necessary.

This example prints the numbers 1 through 8, each number followed by a space:

```
for ($a = 1; $a <= 8; $a++) {
    print "$a ";
}
```

Initially, the variable $a is set to 1. Then this variable is compared with 8, which it is less than or equal to. The body of the loop (print statement) is executed, and then the re-init expression ($a++) is executed, changing the value in $a to 2. Because the value is still less than or equal to 8, the process is repeated until the last iteration where the value of 8 in $a is changed to 9. The value is then no longer less than or equal to 8, so the loop exits.

The foreach Statement

Another iteration construct is the foreach statement. This statement takes a list of values and assigns them one at a time to a scalar variable, executing a block of code with each successive assignment. For example:

```
foreach $a (@a_list) {
     statement_one;
     statement_two;
     statement_three;
}
```

When this loop exits, the original value of the scalar variable is automatically restored. The scalar variable is local to the loop. Here is an example of a foreach loop:

```
@a = (1,2,3,4,5);
foreach $b (reverse @a) {
     print “$b\n”;
}
```

It is possible to omit the name of the scalar variable, in which case Perl presumes that the $_ variable name has been specified. The $_ variable is used as a default for many of Perl’s operations, so it can be thought of as a scratch area. For example, the print function prints the value of $_ if no other value is specified. Here is an example that works like the previous one:

```
@a = (1,2,3,4,5);
foreach (reverse @a) {
     print;
}
```

The implied $_ variable can make things easier. Once you learn more functions and operators that default to $_, this construct will become even more useful.

Exercise 6: temp

1. Type this code into Text Editor:

```
#!/usr/local/bin/perl

print ("What temperature is it today? ");
chomp($temp = <STDIN>);
if ($temp > 74) {
    print “Too hot!\n”;
} else {
    print “Too cold!\n”;
}
```

2. Click **FILE/SAVE**
3. Select the **perl** Folder
4. Name the file: **temp**
5. Click **OK**
6. At the command line in the Terminal window type: **chmod a+x temp**
7. Press **ENTER**
8. At the command line type: **temp**
9. Press **ENTER** to execute the temp program

Exercise 7: thetemp

1. Modify the above program so that it prints “too hot” if the temperature is above 75, “too cold” if the temperature is below 67, and “perfect!” if it is between 67 and 75.
2. Follow the steps in Exercise 7 and name this program **thetemp.**

CREATE A PROGRAM USING HASHES

What Is a Hash?

A hash is similiar to an array because it is a collection of scalar data, with individual elements selected by some index value. Unlike a list array, the index values of a hash are not small non-negative integers, but instead are arbitrary scalars. These scalars (known as *keys*) are used later to retrieve the values from the array.

The elements of a hash have no particular order. Perl puts them in the order that will optimize search time. Consider the elements of a hash like a deck of filing cards. The top half of each card is the key, and the bottom half is the value. Each time you put a value into the hash, a new card is created. Later, when you want to modify the value, you give the key and Perl finds the right card. So the order of the cards is not important.

Hash Variables

A hash variable name is a percent sign (%) followed by a letter, followed by zero or more letters, numbers, and underscores. The part after the percent sign is the same as what was discussed earlier for scalar and array variable names.

Rather than referencing the entire hash, usually the hash is created and accessed by referring to its elements. Each element of the hash is a separate scalar variable, accessed by a string index, called the key. Elements of the hash `%micron` are thus referenced with `$micron{$key}` where `$key` is any scalar expression.

As with arrays, you can create new elements merely by assigning to a hash element. For example:

```
$micron{“aaa”} = “bbb”; # creates key “aaa”, value “bbb”
$micron{128.5} = .15;   # creates key “128.5”, value .15
```

These two statements create two elements in the hash. Subsequent accesses to the same element (using the same key) return the previously stored value:

```
print $micron{“aaa”};   # prints “bbb”
$micron{128.5} += 10;   # makes the value 10.15
```

Referencing an element that does not exist returns the `undef` value, just as with a missing array element or an undefined scalar variable.

Literal Representation of a Hash

At times you may wish to access the hash as a whole, either to initialize the hash or to copy the hash to another hash. Perl does not have a literal representation for a hash; instead, it unwinds the hash as a list. Each pair of elements in the list defines a key and its corresponding value. This unwound representation can be assigned into another hash, which recreates the same hash. For example:

```
@micron_list = %micron; # @micron_list gets
                        #(“aaa”,”bbb”,”128.5”,.15)
%test = @micron_list;   # makes %test like %micron
%test = %micron;        # faster way to do the previous example
%probe = (“aaa”,”bbb”,”128.5”,.15);
       # creates %probe like %micron, from literal values
```

Hash Functions

The keys Function

The `keys(%hashname)` function produces a list of all the current keys in the hash `%hashname`. It is like the odd-numbered (first, third, fifth, etc.) elements of the list returned by unwinding `%hashname` in an array context. If there are no elements to the hash, the `keys` returns an empty list. For example, using the hash from the previous example:

```
$micron{“aaa”} = “bbb”;
$micron{128.5} = .15;
@eng = keys(%micron);      # @eng gets (“aaa”,128.5)
```

As with all other built-in functions, the parentheses are optional.

```
foreach $key (keys (%micron)) {     # once for each key of %micron
     print “At $key we have $micron{$key}\n”;  # show key and value
}
```

This example shows that individual hash elements can be interpolated into double-quoted strings. An entire hash cannot be interpolated.

The values Function

The `values(%hashname)` function returns a list of all the current values of the `%hashname` in the same order as the keys returned by the `keys(%hashname)` function. For example:

```
%lastname = ();    # force %lastname empty
$lastname{“miles”} = “davis”;
$lastname{“stan”} = “getz”;
@lastname = values(%lastname);   # get the values
```

Now `@lastname` contains either (`“davis”`,`“getz”`) or (`“getz”`,`“davis”`).

The each Function

Use `keys` to iterate over an entire hash, looking up each returned key to get the corresponding value. Although this method is frequently used, a more efficient way is to use `each(%hashname)`, which returns a key-value pair as a two-element list. On each evaluation of this function for the same hash, the next succesive key-value pair is returned until all the elements have been accessed. When there are no more pairs, `each` returns an empty list.

So, for example, to step through the `%lastname` hash from the previous example, you would do something like this:

```
while (($first,$last) = each(%lastname)) {
     print "The last name of $first is $last\n";
}
```

If you assign a new value to the entire hash it will reset the `each` function to the beginning. Adding or deleting elements of the hash will most likely confuse `each`.

The delete Function

Up to this point, you have learned how to add elements to a hash, but not how to remove them. Perl provides the `delete` function to remove hash elements. The operand of delete is a hash reference, just as if you were merely looking at a particular value. Perl removes the key-value pair from the hash. For example:

```
%micron = ("aaa","bbb","128.5",.15);  # give %micron two elements
delete $micron{"aaa"};   # %micron is now only a one key-value pair
```

Hash Slices

Like an array variable, a hash can be sliced to access a collection of elements instead of just one element at a time. For example:

```
$month{"febuary"} = 2;
$month{"july"} = 7;
$month{"october"} = 10;
```

This example seems redundant and can be shortened to:

```
($month{"febuary"},$month{"july"},$month{"october"}) = (2,7,10);
```

Even this example seems redundant. You can use a hash slice:

```
@month{"febuary","july","october"} = (2,7,10);
```

Hash slices can be used with variable interpolation also. For example:

```
@num_month = qw(febuary july october);
print "Three of the 12 months are @month{@num_month}\n";
```

Exercise 8: fruit

1. Type this code into Text Editor:

   ```
   #!/usr/local/bin/perl

   %fruit = qw(red apple green lime yellow banana);
   print "Please type a string: ";
   chomp($string = <STDIN>);
   print "The value from $string is $fruit{$string}\n";
   ```

2. Click **FILE/SAVE**
3. Select the **perl** Folder
4. Name the file: **fruit**
5. Click **OK**
6. At the command line in the Terminal window type: **chmod a+x fruit**
7. Press **ENTER**
8. At the command line type: **fruit**
9. Press **ENTER** to execute the fruit program

Exercise 9: wordy

1. Write a program that reads a series of words with one word per line until end-of-file, then prints a summary of how many times each word was seen.
2. Follow the steps in Exercise 9 and name this program **wordy.**

Input from STDIN

We have been reading from standard input already with the <STDIN> operation. Evaluating this in a scalar context gives the next line of input, or undef if there are no more lines. For example:

```
$val = <STDIN>;  # read the next line
```

Evaluating in a list context produces all remaining lines as a list where each element is one line, including its terminating newline. As a reminder, the code looks like this:

```
@val = <STDIN>;
```

Usually you will want to read all lines individually and process each line. One common way to do this is:

```
while (defined($input = <STDIN>)) {
     # process $input here
}
```

As long as a line has been read in, <STDIN> evaluates to a defined value, so the loop continues to execute. When <STDIN> has no more lines to read, it returns undef, which terminates the loop.

Reading a scalar value from <STDIN> into the $_ variable and using that value as the controlling expression of a loop happen frequently, and Perl has an abbreviation for it. Whenever a loop test consists solely of the input operator (like <...>), Perl automatically copies the line that is read into the $_ variable. For example:

```
while (<STDIN>) {   # like “while(defined($_ = <STDIN>)) {“
     chomp;   # like “chomp($_)”
     # the other operations with $_ go here
```

Input from the Diamond Operator

You can also read input with the diamond operator: <>. This works like `<STDIN>` in that it returns a single line in a scalar context or all remaining lines if used in a list context. On the other hand, unlike `<STDIN>`, the diamond operator gets its data from the file or files specified on the command line that invoked the Perl program. For example, you have a program named *reader*, consisting of:

```
#!/usr/local/bin/perl

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

You invoke *reader* with:

```
reader file1 file2 file3
```

Then, the diamond operator reads each line of file1 followed by each line of file2 and file3 in turn, returning `undef` only when all the lines have been read. If you don't specify any file names on the command line, the diamond operator reads from standard input automatically.

Output to STDOUT

Perl uses the `print` function to write to standard output.

Using print for Normal Output

The `print` function takes a list of strings and sends each string to standard output in turn, without any additional intervening or trailing characters. So far we have used `print` to display text on standard output. What might not be obvious is that `print` is really just a function that takes a list of arguments and returns a value like any other function. For example:

```
$m = print("micron ", "is great", "\n");
```

This is another way to say "micron is great." The return value of `print` is a true or false value, indicating the success of the print. It almost always succeeds, unless you get some I/O error, so `$m` here would usually be 1.

Sometimes it is necessary to add parentheses to `print`, especially when the first thing you want to print starts with a left parenthesis. For example:

```
print (5+3), "dinner";  # incorrect, this prints 8 but ignores
                        # "dinner"
print ((5+3, "dinner"); # correct, this prints 8dinner
print 5+3, "dinner";    # this also prints 8dinner
```

Exercise 10: inputline

1. Type this code into Text Editor:

```
#!/usr/local/bin/perl

while($inputline = <>) {
    print($inputline);
}
```

2. Click **FILE/SAVE**
3. Select the **perl** Folder
4. Name the file: **inputline**
5. Click **OK**
6. At the command line in the Terminal window type: **chmod a+x inputline**
7. Press **ENTER**
8. At the command line type: **inputline copy data**
9. Press **ENTER** to execute the inputline program

CREATE A PROGRAM USING FUNCTIONS

So far we have seen and used built-in functions, such as `chomp` and `print`. Now, let's look at functions that you can define for yourself.

Defining a User Function

A user function, usually called a *subroutine* or even just a *sub*, is defined in your Perl program using a construct like this:

```
sub subname {
     statement_one;
     statement_two;
     statement_three;
}
```

The subname is the name of the subroutine, which is any name similiar to the names we have had for scalar variables, arrays, and hashes. Once again, these come from a different namespace, which means you can have a scalar `$micron`, an array `@micron`, a hash `%micron`, and now a subroutine fred.

The block of statements following the subroutine name becomes the definition of the subroutine. When the subroutine is invoked, the block of statements that makes up the subroutine is executed, and any return value is returned to the caller. For example:

```
sub welcome {
     print "Welcome to Perl!\n";
}
```

Subroutine definitions can be located anywhere in your program text, but most Perl programmers put them at the end of the file so the main part of the program is at the beginning of the file.

Subroutine definitions are global and there are no local subroutines. If there are two subroutine definitions with the same name, the latter will overwrite the earlier. Within the subroutine body, you can access or give values to variables that are shared with the rest of the program. Any variable reference within a subroutine body refers to a global variable. For example:

```
sub welcome_who {
     print "Welcome, $name\n";
```

Here, `$name` refers to the global `$name` that is shared with the rest of the program.

Invoking a User Function

You can invoke a subroutine from within any expression by following the subroutine name with parentheses. For example:

```
welcome();       # a simple expression
$test = 8 - welcome();   # part of a larger expression
for ($a = begin_value(); $a < end_value(); $a += increment()) {
     ...
}                # invoke three subroutines to define values
```

A subroutine can invoke a second subroutine, that second subroutine can in turn invoke another subroutine, and so on.

Return Values

A subroutine is always part of some expression. The value of the subroutine invocation is known as the *return value*. The return value of a subroutine is the value of the return statement or of the last expression that is evaluated in the subroutine. For example:

```
sub sum_of_a_and_b {
      return $a + $b;
}
```

The last expression evaluated in the body of this subroutine is the sum of `$a` and `$b`, therefore `$a` and `$b` will be the return values. Here is how it works:

```
$a = 8;  $b = 4;
$c = sum_of_a_and_b();      # $c is 12
$d = 2 * sum_of_a_and_b();  # $d is 24
```

A subroutine is also able to return a list of values when evaluated in a list context. Consider this subroutine and invocation:

```
sub list_of_a_and_b {
     return($a,$b);
}
$a = 3;  $b = 5;
@z = list_of_a_and_b();    # @z gets (3,5)
```

The last expression evaluated really means the last expression evaluated, rather than the last expression defined in the body of the subroutine. For example, this subroutine returns `$a > 0`; otherwise it returns `$b`:

```
sub return_a_or_b {
     if ($a > 0) {
          print “choosing a ($a)\n”;
          return $a;
     } else {
          print “choosing b ($b)\n”;
          return $b;
     }
}
```

All of these examples are somewhat simple. Now we will look at passing values that are different for each invocation into a subroutine instead of relying on global variables.

Arguments

While subroutines that have one specific action are useful, an entire new level of usefulness becomes available when you pass arguments to a subroutine. In Perl, the subroutine invocation is followed by a list within parentheses, causing the list to be automatically assigned to a special variable name @_ for the duration of the subroutine. The subroutine can access this variable to determine the number of arguments and the value of those arguments. For example:

```
sub welcome_to_perl {
      print “Welcome to Perl $_[0].\n”;
```

Here, we have a reference to `$_[0]`, which is the first element of the @_ array. Please note that though they look similiar, the `$_[0]` value is not at all related to the `$_` variable. From the example, it appears to say Welcome to whomever we pass as the first parameter. This means we can invoke like this:

```
welcome_to_perl(“everyone”);    # gives Welcome to Perl everyone.
$b = “folks”;
welcome_to_perl($b);            # gives Welcome to Perl folks.
welcome_to_perl(“me”)+welcome_to_perl(“you”);   # and me and you
```

Note that in the last line, the return values were not actually used. But in evaluating the sum, Perl has to evaluate all of its parts, so the subroutine was invoked twice.

Here is an example using more than one parameter:

```
sub hey {
     print “$_[0], $_[1]!\n”;
}

hey(“hey”,”you”);      # hey you!
hey(“see”,”ya”);       # see ya!
```

The @_ variable is private to the subroutine. Therefore, if there is a global @_, it is saved before the subroutine is invoked and restored to its previous value upon return from the subroutine. This also means that a subroutine can pass arguments to another subroutine without losing its own @_ variable; the nested subroutine invocation gets it own @_ in the same way.

Let’s take look at the “add a and b” routine from earlier. Here is a subroutine that adds two values; specifically, the two values passed to the subroutine parameters. For example:

```
sub add_two {
      return $_[0] + $_[1];
}
print add_two(5,3);# prints 8
$x = add_two(8,4);# $x gets 12
```

Now let's generalize this subroutine. What if we had 5, 3, or 100 values to add together? It can be done with a loop. For example:

```
sub add {
     $sum = 0;             #intialize the sum
     foreach $_ (@_) {
          $sum += $_;      # add each element
     }
return $sum;    # last expression evaluated: sum of all elements
}
$z = add(7,8,9);           # adds 7+8+9 = 24, and assigns to $z
print add(1,2,3,4,5);      # prints 15
print add(1..5);           # also prints 15
```

Private Variables in Functions

We have looked at the `@_` variable and how a local copy gets created for each subroutine invoked with parameters. You can create your own scalar, array, and hash variables that work the same way. Do this with the `my` operator, which takes a list of variable names and creates local versions of them. Here's another look at the add function, this time using `my`:

```
sub add {
     my ($sum);         # make $sum a local variable
     $sum = 0;          # intialize the sum
     foreach $_ (@_) {
          $sum += $_;   # add each element
     }
     return $sum;  # last expression evaluated: sum of all elements
}
```

When the first body statement is executed, any current value of the global variable `$sum` is saved away, and a new variable named `$sum` is created. When the subroutine exits, Perl dumps the local variable and restores the previous (global) value. This works even if the `$sum` variable is currently a local variable from another subroutine. Variables can have multiple nested local versions, even though you can access only one at a time.

Here is an example of creating a list of all elements of an array greater than 100:

```
sub more_than_100 {
     my (@outcome);       # temporary for holding the return values
     foreach $_ (@_) {    # step through the arg list
          if ($_ > 100) { # is the number eligible
               push(@outcome,$_);   # add it
          }
     }
    return @outcome;      # return the final list
}
```

What if we wanted all elements greater than 50 instead of greater than 100? We would have to edit the program, changing the 100's to 50's. But what if we wanted both? We can replace the 50 and 100 with a variable reference instead. The program now will look like this:

```
sub more_than {
     my ($b,@values);     # create a local variable
     ($b,@values) = @_;   #split args into limit and values
     my(@outcome);        # temporary for holding the return values
     foreach $_ (@values) {   # step through the arg list
          if ($_ > $b) {      # is the number eligible
               push(@outcome,$_);   # add it
          }
     }
    return @outcome;     # return the final list
}

@new = more_than(100,@list);    # @new gets all @list > 100
@for = more_than(5,1,5,15,30);  # @for gets (15,30)
```

Notice that we used two additional local variables to give names to arguments. This is fairly common in Perl programming and it is easier to talk about `$b` and `@values` than to talk about `$_[0]` and `@_[1..$#]`.

The result of `my` is an assignable list, meaning that it can be used on the left side of an array assignment operator. This list can be given initial values for each of the newly created variables. This means that we can combine the first two statements of this subroutine, replacing:

```
my($b,@values);
($b,@values) = @_;    # split args into limit and values
```

With:

```
my($b,@values) = @_;
```

This a very common way of programming among Perl programmers. Local nonargument variables can be given literal values in the same way. For example:

```
my($sum) = 0;     # initialize local variable
```

Note: Despite its appearances as a declaration, `my` is really an executable operator.

Semiprivate Variables: the local Function

Perl provides another way to create private variables, using the `local` function. It is important to understand the difference between `my` and `local`. For example:

```
$value = “primary”;

tellme();
spoof();
tellme();

sub spoof {
     local ($value) = “temp”
     tellme();
}

sub tellme {
     print “Current value is $value\n”;
}
```

This will print out:

```
Current value is primary
Current value is temp
Current value is primary
```

If `my` had been used instead of `local`, the private reading of `$value` would be available only with the `spoof()` subroutine. But with `local`, the private value is not quite so private; it is also available within any subroutines called from `spoof()`. As a general rule, local variables are visible to functions called from within the block in which those variables are declared.

While `my` can be used only to declare simple scalar, array, or hash variables with alphanumeric names, `local` has no such restrictions. Because `$_` is used so often in Perl programs, it is a good idea to place a `local $_;` at the top of any function that uses `$_` for its own purposes. This assures that the previous value will be preserved and automatically restored when the function exits.

In general, you should probably use `my` over `local` because it is faster and safer.

File-Level my() Variables

The `my()` operator can also be used at the outermost level of your program, outside of any subroutines or blocks. Although this is not really a "local" variable, it can be rather useful. For example, when you place `use strict;` at the beginning of your file, you will not be able to use variables (scalars, arrays, and hashes) until you have first "declared" them. And you declare them with `my()`, like this example:

```
use strict;
my $a;                    # starts as undef
my @b = qw(bono edge larry);   # give intial value
...
push @b, qw(adam);        # can't forget adam
@c = sort @b;             # Will Not Compile
```

At compile time, the last statement will be flagged as an error because it referred to a variable (`@c`) that had not previously been declared with `my`. Your program will not even start running unless every single variable being used has been declared.

There are two main advantages of forcing variable declarations:

1. Your programs will run slightly faster because variables created with `my` are accessed slightly faster than ordinary variables.
2. You will catch mistakes in typing much faster, because you will no longer be able to accidentally reference a nonexisting variable named `$mircon` when you wanted `$micron`.

Because of these advantages, many Perl programmers automatically begin every new Perl program with `use strict`.

Exercise 11: one23

1. Type this code into Text Editor:

```
#!/usr/local/bin/perl

sub card {
    my %card_map;
    @card_map{1..9} = qw(
        one two three four five six seven eight nine);

    my($num) = @_;
    if ($card_map{$num}) {
        return $card_map{$num};
    } else {
        return $num;
    }
}
# driver routine
while (<>) {
    chomp;
    print "card of $_ is ",  &card($_), "\n";
}
```

2. Click **FILE/SAVE**
3. Select the **perl** Folder
4. Name the file: **one23**
5. Click **OK**
6. At the command line in the Terminal window type: **chmod a+x one23**
7. Press **ENTER**
8. At the command line type: **one23**
9. Press **ENTER** to execute the one23 program

APPENDIX 1 - REFERENCES

Schwartz, Randal L. Christiansen, Tom.
(1997) 2nd Edition Learning Perl
O'Reilly & Associates, Inc. California

Till, David.
(1996) 2nd Edition Teach Yourself Perl 5 in 21 Days
Sams Publishing Indiana

McMillan, Michael.
(1998) Perl From the Ground Up
Osborne/McGraw-Hill California

Corperate Products & Tools - Systems Programming
(2001) Micron's Perl Website
http://perl.micron.com

APPENDIX 2 - ASCII CHARACTER SET

ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@' or an action of some sort. ASCII was established to achieve compatibility between various types of data processing equipment.

ASCII, pronounced "ask-key", is the common code for microcomputer equipment. The standard ASCII character set consists of 128 decimal numbers ranging from zero through 127 assigned to letters, numbers, punctuation marks, and the most common special characters. The Extended ASCII Character Set also consists of 128 decimal numbers and ranges from 128 through 255 representing additional special, mathematical, graphic, and foreign characters. Here is an ASCII chart with the decimal number, the hexidecimal number, and the ASCII representation:

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

APPENDIX 3 - REQUESTING A UNIX ACCOUNT

Go to the following site:

http://unix.micron.com/general/genservices_index.html

and select Request Account. You can also reach this webpage by typing UNIX into the address bar and clicking the General Services tab.

or

For immediate assistance, please contact the Support Center at (208) 368-3000.

WHAT'S NEXT

Alternative Training

To obtain more information about alternative training or to check courseware out, visit:

http://hercules.micron.com/is/support/trnres/J3/j3table.htm

SST

For more information about SST, please visit our homepage at:

http://hercules.micron.com/is/depts/cs/sst/default.htm

UNIX Help

To obtain help from the Micron Perl webpages, including using Perl on different operating systems and Perl references, visit:

http://perl.micron.com/

Support Center

For immediate assistance with any computer related issue, please contact the Support Center at (208) 368-3000.