

NETWORK EQUIPMENT TECHNOLOGIES
PANAVUE MANAGEMENT PLATFORM
INTRODUCTION TO PERL SCRIPTING
RELEASE 2.0



.....

Issued **September 1998**

NETWORK EQUIPMENT TECHNOLOGIES, INC., (N.E.T.) PROVIDES THIS DOCUMENT AS IS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

This document constitutes the sole Specifications referred to in N.E.T.'s Product Warranty for the products or services described herein. N.E.T.'s Product Warranty is subject to all the conditions, restrictions, and limitations contained herein and in the applicable contract. N.E.T. has made reasonable efforts to verify that the information in this document is accurate, but N.E.T. reserves the right to correct typographical errors or technical inaccuracies. N.E.T. assumes no responsibility for any use of the use of the information contained in this document or for any infringement of patents or other rights of third parties that may result. Networking products cannot be tested in all possible uses, configurations or implementations, and interoperability with other products cannot be guaranteed. The customer is solely responsible for verifying the suitability of N.E.T.'s products for use in its network. Local market variations may apply. This document is subject to change by N.E.T. without notice as additional information is incorporated by N.E.T. or as changes are made by N.E.T. to hardware or software.

Copyright © 1998 Network Equipment Technologies, Inc.
All rights reserved.

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but not limited to, photocopy, photograph, magnetic, or other record, without the prior written permission of N.E.T.

**Restricted
Rights U.S.
Government
Rights**

The software accompanying this documentation is furnished under a license and may only be used in accordance with the terms of such license. This documentation is “commercial computer software documentation” as that term is used in 48 CFR 12.212. Unless otherwise agreed, use, duplication, or disclosure of this documentation and any related software by U.S. Government civilian agencies is subject to restrictions as set forth in 48 CFR 52.227-14 (ALT III) and 48 CFR 52.227-19, and use, duplication, or disclosure by DoD is subject to restrictions as set forth in 48 CFR 227.7202-1(a) and 48 CFR 227.7202-3(a) or, if applicable, 48 CFR 252.227-7013(c)(1)(ii) (OCT 1988).

Unpublished-rights reserved under the copyright laws of the United States.

Network Equipment Technologies, Inc./N.E.T. Federal, Inc.
6500 Paseo Padre Parkway
Fremont, CA 94555

.....

Trademarks IDNX, ADNX, and the N.E.T. logo are registered trademarks, and CellXpress, FrameXpress, Frame Relay Exchange, ISDNX, LAN/WAN Exchange, Network Equipment Technologies, NetOpen, N.E.T., PanaVue, PortExtender, PrimeSwitch, PrimeVideo, PrimeVoice, Promina, SONET Transmission Manager, STM, and SPX are trademarks of Network Equipment Technologies, Inc. All other trademarks are the sole property of their respective companies.

Apache Server source, binaries, and documentation copyright © 1995,1996, 1997, 1998 The Apache Group. All rights reserved. This product includes software developed by the Apache Group for use in the Apache HTTP server project (<http://www.apache.org/>).

SunOS and Solaris software copyright held by Sun Microsystems, Inc. Sun Microsystems is a registered trademark and Sun, SunOS, OpenWindows, Solaris, and Ultra are trademarks of Sun Microsystems, Inc.

SPARC is a registered trademark of SPARC Internal, Inc. SPARCstation is a registered trademark of SPARC International, Inc. licensed exclusively to Sun Microsystems, Inc.

ORACLE and SQL*Plus are trademarks of Oracle Corporation.

X-Window System software copyright held by Massachusetts Institute of Technology.

Openview®, HP, and the HP logo are trademarks of Hewlett-Packard Company.

OpenSoftware Foundation, OSF, the OSF logo, OSF/MOTIF, and MOTIF are trademarks of the Open Software Foundation.

All other trademarks are the sole property of their respective companies.

Note: In this manual, any reference to *PanaVue* refers to the PanaVue Management Platform product line, unless specified differently.

.....



Preface

About This Document

This document provides an introduction and short tutorial to the Perl scripting language, as it is used on the PanaVue workstation. It assumes that the user has a working knowledge of how scripting languages have been implemented on the PanaVue workstation; see the *PanaVue Scripting Guide* for more information on this topic.

Document Organization

The document contains the following sections:

Title	Description
<i>Introduction to Programming in Perl</i>	Provides an overview to the Perl language, what this guide covers, and other resources.
<i>Basics of the Perl Programming Language</i>	Provides an introduction to the syntax of Perl scripts and advice for programmers who are new to Perl.
<i>Using Variables in Perl</i>	Describes how Perl implements scalar variables, arrays, and associative arrays.
<i>Perl Output Commands</i>	Describes the commands used to output information to STDOUT, STDERR, and files.

Title	Description
<i>Operators in the Perl Language</i>	Describes Perl's built-in arithmetic, logical, and relational operators.
<i>Control Structures and Loops</i>	Describes the Perl control structures: <i>if</i> , <i>unless</i> , <i>while</i> , <i>for</i> , <i>foreach</i> .
<i>Perl's Built-In Functions</i>	Describes Perl's built-in arithmetic, timekeeping, and string functions.
<i>File Access in Perl</i>	Describes Perl's commands for accessing files and directories.
<i>Using Regular Expressions in Perl</i>	Describes how to perform matchings and substitutions on strings using Perl's regular expressions.
<i>Using Subroutines in Perl</i>	Describes how to write and use procedures (subroutines) in Perl.


Document Conventions



The following conventions are used in this document:

Convention	Example	Description
Promina	Configure the Promina SNMP agent.	Refers either to a Promina 800 Series node or to a Promina 4000 ATM switch, depending on the context.
Promina 800 Series	Install the Promina 800 Series node.	Refers generally to the Promina 100, 200, 400, and 800 Multiservice Access Platform.
Promina 4000	Configure the Promina 4000 switch.	Refers generally to the Promina 4000 ATM switch.
node	The node can be queried using SNMP GET requests.	Refers generally to any SNMP-aware network device. Depending on the context, this term can also refer specifically to Promina 800 Series nodes or Promina 4000 switches.
switch	The switch generates SNMPv2 traps.	Refers to the Promina 4000 ATM switch, unless specified differently.
Key name	Press the Delete key.	Refers to non-printing keys on the keyboard that you press.
Simultaneous Key	Press Shift+F1.	Refers to non-printing keys on the keyboard that need to be pressed simultaneously.

Convention	Example	Description
bold	Install Card.	Indicates a command to be typed. Also used for emphasis.
Enter	Enter Install Card.	Indicates that after typing the information, press the Return or Enter key.
<i>italic</i>	The <i>Alarms Pending</i> message displays on the screen.	Refers to parameter options and other information displayed by the software.
	A <i>gateway node</i> is any node that connects to a domain.	Refers to a new term that is defined.
	For more information, see the <i>Hardware Description</i> manual.	Refers to a document or book title.

The following icons are used in this document to provide important information:

Icon	Description	Definition
	Warning	Provides information on how to avoid a potentially hazardous situation that, if not avoided, could result in death or serious injury.

Icon	Description	Definition
	Caution	Provides information on how to avoid possible disruption of traffic or damage to files or equipment.
	Note	Provides information that helps the user and should be read before proceeding.

Associated Documents

The following documents provide additional information about Perl and the other scripting languages on the PanaVue workstation:

Document	Description
<i>PanaVue Scripting Guide</i>	Describes how to use Expect, Perl, Scenarios, Scotty, and Perl in the PanaVue environment.
<i>Perl Man Pages</i>	Online documentation for Perl (online the PanaVue workstation at http://idDocs/scripts/Perl).
<i>Expect Tutorial and Introduction</i>	An introduction to the Expect Scripting language (online the PanaVue workstation at http://idDocs/scripts/expect).
<i>Scotty Man Pages</i>	Online documentation for the Scotty Scripting language, an extension to TCL providing SNMP support (online the PanaVue workstation at http://idDocs/scripts/scotty).
<i>TCL Tutorial and Introduction</i>	An introduction and tutorial to the TCL Scripting language (online the PanaVue workstation at http://idDocs/scripts/tcl).

Reader's Response

We encourage comments on the content of this document. Please address any comments to:

Manager, Information Development
Network Equipment Technologies, Inc.
6500 Paseo Padre Parkway
Fremont, California 94555

infodev@net.com

N.E.T. may use or distribute, without incurring any obligation, and in any way it believes appropriate, any information supplied.

.....

Contents

.....

Introduction to Programming in Perl

Topics Not Covered in This Manual	3
What is Perl?	4
Basic Features of Perl	5
For More Information	7

Basics of the Perl Programming Language

Using the Perl Interpreter	9
Specifying a #! Line	10
Command Line Options	10
-c (check)	11
-d (debugger)	11
-D (Debugger)	11
-e (execute)	12
-n (next/loop)	13
-p (print and loop)	13
-s (switches)	13
-S (Search)	14
-T (Taint checking)	14

.....

-v (version)	15
-w (warnings)	15
-x (extract)	16
Perl Basics	17
Perl Syntax	17
Naming Conventions	20

Using Variables in Perl

Scalar Variables	25
Arrays	31
Associative Arrays	38
Avoiding Confusion with Perl's Variables	46
Special and Predefined Variables	48

Perl Output Commands

Using the print Command	52
Using the printf Command	55
Using the write Command	59
Writing to Multiple Lines	63
Outputting Variables Containing Multiple Lines	66
Adding Page Headers to a Report	67

.....

Problems with Buffering of Output	69
---	----

Operators in the Perl Language

Assignment	73
Common Shortcuts	75
Autoincrement and Autodecrement	76
Relational	79
Logical	82

Control Structures and Loops

if/else Command	84
unless/else Command	86
while Command	88
for Command	90
foreach Command	93

Perl's Built-In Functions

Arithmetic Functions	97
abs	97
atan	97
cos	97

.....

exp	97
hex	97
int	98
log	98
oct	98
rand	99
sin	99
sqrt	99
srand	99
Timekeeping Functions	100
time	100
localtime	100
gmtime	100
String Functions	101
chomp	101
chop	101
chr	101
index	102
length	102
lc	102

.....

lcfirst	102
ord	102
rindex	103
substr	103
uc	103
ucfirst	103
undef	104

File Access in Perl

Using Filehandles	106
Using STDIN, STDOUT, and STDERR	107
Using STDIN to Read One Line	108
Using STDIN to Read An Entire File	110
Using STDOUT and STDERR	110
Reading Input From the Diamond Operator (<>)	112
Changing the File List	112
Finding the Current Filename	114
Accessing Files on the Command Line	117
Using Filehandles	119
Opening a File for Read-Only Access	119

.....

Opening a File for Write Access	120
Closing a Filehandle	122
Directory Operations	124
Changing the Working Directory	124
Listing the Files in a Directory	125
Reading a Directory Entry Directly	126
Deleting Files	128
Renaming Files	129
Creating and Removing Directories	130
File Test Operators	132

Using Regular Expressions in Perl

Defining Regular Expressions	138
Rules in Using Regular Expressions	140
Simple Matching and Replacing	144
Using Wildcards in Regular Expressions	155
Wildcards are “Greedy”	157
Using WildCards for Matching	159
Using Wildcards for Substitutions	161
Using Subroutines in Perl	

.....

Defining Subroutines	167
Using a Subroutine's Return Value	169
Defining Local Variables	174
Passing Arguments	178

Introduction to Programming in Perl

This document provides an introduction to the Perl language and should be sufficient to allow anyone with any previous programming experience (shell scripts, Basic, C) to begin using and modifying the template scripts that are shipped with the PanaVue Network Management System.

This section can also be useful if you do not have any programming experience, but you are strongly advised to also read some of the other Perl resources listed in *For More Information* on page 7. In particular, the book “*Learning Perl*” by Randal L. Schwartz, O’Reilly and Associates is recommended; this book is useful to all newcomers to the Perl language but especially to beginning programmers.

This introduction to Perl is divided into the following sections, each of which describes a major feature of the Perl language:

- *Basics of the Perl Programming Language* – provides an overview to the structure and common conventions of Perl programs.
- *Using Variables in Perl* – describes the types of variables and data structures most commonly used in Perl programs.
- *Perl Output Commands* – describes Perl’s *print*, *printf*, and *write* commands.
- *Operators in the Perl Language* – describes the basic arithmetic and string operators that are built-in to Perl.

- *Control Structures and Loops* – describes the control structures and loops most commonly used in Perl programs.
- *Perl's Built-In Functions* – briefly lists the basic built-in functions of the Perl language.
- *File Access in Perl* – describes the use of filenames and filehandles in reading, writing, creating, and deleting files.
- *Using Regular Expressions in Perl* – describes the use of regular expressions, one of the major features of the Perl language.
- *Using Subroutines in Perl* – describes how to declare and use subroutines.

See the following sections for a list of topics that are not covered in this document, as well as a description of Perl's basic features and a list of other references about Perl.



Note: This chapter provides only a basic introduction to the Perl language and how it can be used on the PanaVue workstation. It assumes you have some working knowledge of basic programming concepts such as variables, control loops, and subroutines. For a complete reference to these topics and to the Perl language, see the books and online references listed in *For More Information* on page 7.

Topics Not Covered in This Manual

This manual does not cover the following features of Perl version 5:

- Accessing the Unix-style socket interface for network communications
- *Do/while* and *do/until* control structures
- Managing and accessing database (DBM) files
- Object-oriented programming and variables
- Operators that change the operation of loops and control structures (*continue*, *goto*, *last*, *next*, *redo*, *return*)
- The Perl built-in debugger
- Reading files in binary mode (*binmode*)
- Running external programs using the *system* and *exec* functions, and system-level functions such as *fork*, *getppid*, and *syscall*
- Simplifying control structures by using *&&*, *||*, and *?* in a more C-like syntax.
- The more arcane and sophisticated aspects of regular expressions
- Using and creating Perl modules (*packages*)
- Using *pack* to store more than one value in each element of an associative array

For more information about these topics, see the online Perl documentation or any of the Perl reference works listed in *For More Information* on page 7.

What is Perl?

Perl is the “*Practical Extension and Report Language*,” a scripting language written by Larry Wall to produce reports for an early Unix system. Since its first incarnation as version 1.0, Perl has become increasingly popular, particularly because it is well suited for writing CGI scripts to be used on the World Wide Web (WWW). Perl’s most recent release, version 5, has also encouraged its use by being accessible to both beginning and experienced programmers.

Beginning programmers usually find Perl is relatively easy to learn and that they can begin writing useful scripts with only a basic knowledge of the language. They can then gradually expand their knowledge of Perl by learning bits and pieces as needed.

One of Perl’s most attractive features is that it usually offers more than one way to do any particular task. Beginning programmers can therefore start off using the simpler approaches that are possible, and then as their experience grows, they can use the more sophisticated techniques.

Perl is also attractive to experienced programmers because it is derived from existing utilities and languages such as the Unix shell and the C programming language. Thus, many of the concepts and techniques used in Perl are very familiar to experienced programmers.

Also, because Perl scripts are interpreted and does not have to be compiled into a binary program before being run, programmers do not have to endure lengthy compile and link cycles before running their programs. However, the Perl interpreter does do a quick internal compile of each program before it is run, so that any syntax errors can be found and corrected beforehand.

Basic Features of Perl

Larry Wall has called Perl “a shell for C programmers,” and he intended it to be a free-form language that programmers could use to quickly write programs that would be too simple or tedious for a fully structured language such as C. He kept many of the best features of C, while adding in the strengths of other tools, such as *sed* and *awk*, that are used in Unix systems to manipulate text files and databases.

Perl was an “overnight success” with experienced programmers, but Larry Wall has continued to refine the language over the years, broadening its appeal, so that version 5 now offers the following features and capabilities:

- Perl combines the best features of interpreted and compiled languages. Perl programs are interpreted, so you do not need to perform a manual compile and link before running your programs. However, the Perl interpreter does do a quick internal compile of each script that is run, which not only finds any syntax errors in your program but also increases the speed of your program’s execution.
- Perl is not as structured as the C language, but contains the flow control and array structures that are used most often in C programming.
- Perl has no built-in limitations. The only limitations as to data size and file size are those of the host system.
- Perl uses sophisticated regular expressions that simplify text manipulation, pattern matching, and text substitution. Perl often can do in a few lines what it would take a complicated C language program to do.
- Perl’s variables can accept both text and binary data without the programmer having to declare the data type in advance.

- In addition to scalar data types and standard arrays, Perl offers an associative array that can be used for quick database access and management.
- Since Perl was originally written to generate reports, it contains many features to help in creating and formatting reports.
- Perl is easily extensible, so options such as SNMP communication can be incorporated into your programs as if they were part of the Perl language.

For More Information

In addition to this document and the *PanaVue Scripting Manual*, you can find more information on Perl from the following online sources:

- The online man pages, which are accessible through your PanaVue workstation and web browser at <http://idDocs/scripts/perl/htmldocs>. You can also access the man pages at the Solaris command line by entering the command **man perl**.
- If you have internet access, you can visit the Perl home page at <http://www.perl.com/perl>, which has links to many other Perl sites, including the Comprehensive Perl Archive Network (CPAN), whose home base is in Finland but which has mirrored archives throughout the world.
- *news:comp.lang.perl.announce* – a Usenet newsgroup that contains announcements of new releases of Perl and Perl extensions.
- *news:comp.lang.perl.misc* – a Usenet newsgroup for general discussions about Perl. It is a good place to ask questions about specific programming problems when you cannot find an answer on your own.
- *news:comp.lang.perl.modules* – a Usenet newsgroup devoted to discussion of Perl extensions, libraries, and other modules that are not part of the standard Perl package.

The following books are recommended as reference guides and tutorials for the Perl language:

-
- *Learning Perl*, second edition, by Randal L. Schwartz and Tom Christiansen, O'Reilly & Associates, Inc. Sebastopol, CA. July, 1997 (<http://www.oreilly.com>) – a short introduction to Perl that introduces the major concepts and commands.
 - *Perl 5 Desktop Reference*, by Johan Vromans, O'Reilly and Associates, Inc. 1996 – a pocket-sized quick reference to the Perl programming language.
 - *Programming Perl*, second edition, by Larry Wall and Randal Schwartz, O'Reilly and Associates, Inc. 1996 – the definitive reference to Perl version 5 by its author. It includes many complete programs that can either be used as is or as templates for your own programs.

The following books describe Perl's use for writing web-based CGI scripts:

- *CGI Programming on the World Wide Web*, by Shishir Gundavaram, O'Reilly & Associates, Inc. Sebastopol, CA. 1996. ISBN: 1-56592-168-2.
- *Web Client Programming with Perl*, by Clinton Wong, O'Reilly & Associates, Inc. Sebastopol, CA. 1996. ISBN: 1-56592-214-X.

Basics of the Perl Programming Language

This section describes how to use the Perl interpreter and what command line options can be used with it. It also describes the Perl programming language's basic syntax rules and naming conventions.

Using the Perl Interpreter

On the PanaVue workstation, the Perl interpreter (*/usr/thirdParty/perl/bin/perl*) is run using a command line similar to the following:

```
/usr/thirdParty/perl/bin/perl options script.pl command-line-args
```

where *options* are the options for the Perl interpreter, *script.pl* is the Perl script being run, and *command-line-args* are the arguments that should be passed to the script. As a general habit, you should also include the “-w” option for all your scripts since that it warns you of many potential problems, such as misspelling a variable name:

```
/usr/thirdParty/perl/bin/perl -w script.pl command-line-args
```

See *Command Line Options* on page 10 for more information about this and other options.

Specifying a #! Line

If the first line of a script begins with the “#!” characters, the Perl interpreter first verifies that it should be running the script by looking at this line. If the interpreter does not find the word “*perl*” anywhere on this line, it does not execute the script; instead, it calls whatever program is specified and passes the script to it.

For example, if “*script.sh*” is a shell script that begins with the line “*#!/bin/sh*”, and you give the command “*perl script.sh*”, the Perl interpreter executes the command “*/bin/sh script.sh*” so that the shell program runs the script itself.

This is a quick way of running other scripts if you are not sure which shell or program they belong to, but it is not recommended since it takes several seconds for the Perl interpreter to load, to examine the script file, and to execute the proper command.

Command Line Options

Perl’s command line options can be specified either on the command line or as part of the first line of a standalone script. For example, the **-w** option instructs the interpreter to print a warning about possible typographical errors and anything else it considers to be a bad programming practice.

If you invoke the Perl interpreter directly, specify this option on the command line as follows:

```
/usr/thirdParty/perl/bin/perl -w script.pl command-line-args
```

If, on the other hand, your scripts are run as standalone commands, add the **-w** option at the end of the first line of each script:

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



Note: It is strongly recommended you use the **-w** option for all your scripts because it is a way of highlighting potential problems and typographical errors.

The following are the most useful command line options for the Perl interpreter. See the online documentation for a complete listing:

-c (check)

The **-c** option checks the syntax of a Perl script without actually executing it. This option is especially useful in verifying that the script's opening and closing braces match, and that all of the library files referenced by **"use"** statements actually exist. For example:

```
perl -c myscript.pl
```

-d (debugger)

The **-d** option turns on the interactive Perl debugger and it should be used only when running scripts at the command-line. See the online documentation for the Perl debugger for information on using it. For example:

```
perl -d myscript.pl
```

-D (Debugger)

The **-D** option customizes the debugger so as to focus on specific operations within your scripts. This option is not enabled in the Perl interpreter that is shipped with PanaVue because including it can significantly slow down regular operations. To use

.....

this feature you must recompile the Perl source code with the “-DDEBUGGING” flag turned on.

If you recompile Perl, be certain that you do not overwrite the Perl interpreter that is shipped with the PanaVue system. Instead, put the debugger version of Perl in a separate directory.

-e (execute)

The **-e** option executes Perl statements that are given on the command line instead of executing a Perl script. This option is most commonly used with the **-n** and **-p** options (see below) to create shell aliases.

For example, to create a command named **print-old** that lists the filenames of all files in the */opt/Panavue/reports* directory that are more than 14 days old, define the following alias in your *.cshrc* file:

```
alias print-old "find /opt/Panavue/reports -mtime +14 -print | perl -ne 'print;' | more"
```

This is a trivial example since the *find* command can also print out the filenames, but Perl could be used to delete the old files by modifying the alias as follows:

```
alias remove-old "find /opt/Panavue/reports -mtime +14 -print | perl -ne 'chop; unlink;'"
```

Be careful when using Perl to delete or modify files in this manner. Before using a script that can modify or delete files, you should first test your Perl script by having it print out the filenames as shown in the first example. When you are satisfied that only the proper files are being specified, then modify the alias so that it actually deletes or changes the files.

-n (next/loop)

The **-n** option instructs the Perl interpreter to continuously loop your script until the end of input (from STDIN or a specified file). This option is commonly used with the **-e** option (see above).

-p (print and loop)

The **-p** option, like the **-n** option (see above), instructs the Perl interpreter to continuously loop your script until the end of input (from STDIN or a specified file). However, unlike **-n**, this option prints each line of input after it has been processed.

If this option is used with the **print-old** alias shown above, you do not need Perl's *print* statement:

```
alias print-old "find /opt/Panavue/reports -mtime +14 -print | perl -pe | more"
```

-s (switches)

The **-s** option instructs the Perl interpreter to convert into variables any switches that appear on the command line after the script's filename. Switches must start with a hyphen (-) and contain only letters, numbers, or underscores (such as "**-switch**" or "**-var2**"). An optional value can be appended using an equal sign (such as "**-switch=value**"); if no value is given, a value of **1** is automatically assigned.

For example, the following command line has three switches, two which are assigned specific values ("friday" and "alarms") and one which is given the value of 1:

```
perl -s myscript.pl -day=friday -flag -report_type=alarms
```

When this command is executed, the three switches are converted into variables that the script *myscript.pl* can access. For example, you could print out these variables using the following lines:

```
print "The day variable is $day\n"; # prints out "friday"
print "The flag variable is $flag\n"; #prints out "1"
print "The report_type variable is $report_type\n"; # prints out "alarms"
```

-S (Search)

The **-S** option instructs the Perl interpreter to search for the specified script using the `PATH` environment variable. This is useful only when the `PATH` variable exists and when the desired script is in one of the `PATH` directories.

For example, if *myscript.pl* is in your search `PATH`, you could execute it from any directory by giving the following command:

```
perl -S myscript.pl
```

-T (Taint checking)

The **-T** option turns on taint checking, which prevents any user input (command line arguments, environment variables, or input from `STDIN`) from being used in a command that when run by the root user could damage the system or evade the system's built-in security features. You should not use this feature unless you are a system administrator who understands the *setuid* and *setgid* functions of the Solaris operating system.

-v (version)

The **-v** option displays the version and patchlevel of the Perl interpreter and then exits. To get the version of Perl from within a script, use the **\$]** special variable:

```
print "The Perl version number is $]\n";
```

-w (warnings)

The **-w** option prints a warning when the following situations occur in your script:

- A variable or other identifier is used only once, which could indicate a typographical error
- A variable is used before a value has been assigned to it
- A subroutine is defined more than once
- A filehandle is used before being defined
- The script attempts to write to a filehandle that was opened read-only
- A subroutine recursively calls itself until it is nested 100 or more levels deep
- The numeric equality (**==**) or numeric inequality (**!=**) operator is used with variables that appear to contain strings (which require the string operators *eq* and *ne*)

These warnings do not halt the execution of the script, but they do indicate it is likely the script is not operating as originally intended.



Note: It is strongly recommended that you use the **-w** command line option for all your scripts until you have tested them thoroughly and are confident they perform exactly as intended.

-x (extract)

The **-x** option instructs the Perl interpreter to extract the Perl script from the specified input file. The interpreter reads the input file and discards all lines until it finds a line that starts with “**#!**” and that contains the word “*perl*”. The interpreter then treats all of the following lines as a Perl script until it finds one of the following:

- the End of File (EOF)
- a CTRL-D (ASCII 4)
- a CTRL-Z (ASCII 26)
- a line with the “*_END_*” keyword

This option must be specified on the command line (and not as part of the **#!** line) so it is especially useful if you want to include uncommented explanatory documentation at either the beginning or end of your scripts. If so, you can run the scripts without Perl complaining about the non-commented text by giving the following command:

```
perl -x myscript.pl
```

Perl Basics

Perl, in comparison with other interpreted languages, is fairly unstructured. A Perl script is executed in a linear fashion, from the beginning to the end, with each line being executed sequentially (except for comments and subroutine definitions). The following sections describe the common features and requirements of typical Perl scripts.

Perl Syntax

A Perl program needs to follow only a few simple rules:

- For scripts that are used on the PanaVue workstation, the first line should always be the following:

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

This is not required when you run your scripts by directly calling the Perl interpreter but it is still recommended.

- All Perl statements, except the opening and closing brackets of a control structure or loop, must end with a semicolon (;). For example:

```
if ($node_num > 250) {  
    print ("Illegal node number.\n");  
    print ("Resetting node number to 0.\n");  
    $node_num = 0; }
```

- For the most part the Perl interpreter ignores whitespace (spaces, tabs, newlines, carriage returns, and formfeeds), so you can format your Perl scripts however is most convenient. For example, all of the following versions of code are identical as far as the Perl interpreter is concerned:

```
if ($node_num > 250) {print ("Illegal node number.\n");}
or
if ($node_num > 250) {
print ("Illegal node number.\n"); }
or
if ($node_num > 250)
{
print ("Illegal node number.\n");
}
```

However, although Perl ignores whitespace and formatting, these things make your programs more readable to humans, so using them is recommended. Choose a style of indentation and formatting that you find convenient and enhances readability.

- Comments are indicated by the number sign (#). The Perl interpreter ignores anything on a line that follows the comment sign, so you can put comments on their own lines or on the same line as Perl code:

```
# Check for a legal node number
if ($node_num > 250)
{
print ("Illegal node number.\n");
}
or
if ($node_num > 250) # check for a legal node number
{
print ("Illegal node number.\n"); # show error message
}
```

To spread a comment across multiple lines, use a comment character for each line:

```
# Check to see if the node number is greater than the
# maximum allowable number (250) and if it is, print an
# error message informing the user
if ($node_num > 250) {
    print ("Illegal node number.\n");
}
```

It is strongly recommended you comment your scripts thoroughly, explaining what the script is attempting to do, the logic it is using, and how it is implemented. Doing so makes it easier for others to understand your scripts and helps you when you want to update a script later on.

- Perl defines “true” and “false” slightly differently than other programming languages such as C. When control structures such as “*if*” and “*while*” and logical operators such as “&&” and “//” evaluate an expression, “true” is any nonzero or non-null value. “False” is the “undefined” value, which in Perl is a null string (“”) when used in a string context or the number zero (“0”) when used in a numerical context.

In practice, the dual definition of the undefined value is very convenient, but it can cause some problems in isolated cases when you are converting programs originally written in other languages to Perl. In these cases you should evaluate all test expressions to ensure that they interpret the undefined value properly.

- If Perl has any single distinguishing characteristic, it is that you can accomplish the same task more than one way. For example, there are three obvious ways to read a list of files specified on the command line and many more not so obvious methods. If you look at the various Perl scripts available on the internet, you will see that different programmers routinely use different techniques to accomplish

the same tasks, and for the most part which one you choose is a matter of personal preference.

However, sometimes these different methods have slightly differently requirements and side-effects, so if you find a method of doing something that works, be cautious about changing it until you have thoroughly tested the alternatives.

- If Perl has any secondary distinguishing characteristic, it is that a default exists for most operations. Using these defaults where applicable can simplify your scripts but also make them more difficult for others to read. This, though, is also a matter of personal preference.

Naming Conventions

Perl uses the same naming conventions for variables, subroutines, and filehandles:

- A name can use only “word” characters, which are defined in Perl as being letters (both uppercase and lowercase), numbers (0 through 9), and the underscore (`_`) character.
- If the name does not start with a letter, it can be only one character long. Typically this is not significant because most variables of this type have predefined meanings (see *Special and Predefined Variables*).
- Names are case-sensitive, so “*variable*” refers to a different object than “*Variable*” or “*VARIABLE*”. As a general rule, lowercase names refer to variables and subroutines, while uppercase names refer to filehandles, but this is only a matter of custom and convention, not a requirement of Perl.

- With the exception of filehandles, an object's name is preceded by a single character that defines what it refers to. See Table 1:

Table 1 Naming Conventions

Symbol	Identifies	Examples
\$	scalar variable or array element	\$string, \$input_line, \$number, \$answer \$array[1], \$namelist[22], \$months[3] \$cards{"prc"}, \$days_in_month{"oct"}
@	array	@array, @namelist, @months
%	associative array	%cards, %phone_numbers, %days_in_month
&	subroutine	&toupper, &get_input, &output_line

Programmers familiar with other languages often get confused by Perl's use of symbols to differentiate between different variable types, especially when accessing elements of an array. This confusion is increased by the fact that Perl allows you to use the same name for scalar variables, array variables, and subroutines.

For example, *\$name*, *@name*, and *%name* all refer to different variables with different values and structures. Furthermore, *\$name[0]* (a single element of the *@name* array) and *\$name{0}* (a single element of the *%name* array) are different from each other and from *\$name*. Furthermore, *&name* refers to a subroutine, not a variable of any type.

.....

Until you become comfortable with Perl's naming conventions and use of variables, it is highly recommended that you use unique names for each variable and subroutine in your scripts.

See *Special and Predefined Variables* on page 48 for more information on the different types of variables and how to use them. See *Using Subroutines in Perl* on page 166 for information on defining and using subroutines.

Using Variables in Perl

Unlike other languages that use a dozen or more different data types for variables, Perl has only three basic variable types:

- *Scalar Variables* – scalar variables have a dollar sign (\$) prefix and contain either a number or a string. Perl uses the context of a scalar variable’s use to determine whether it contains numeric or string data. Examples of scalar variables are *\$string*, *\$number*, *\$day_of_week*. See *Scalar Variables* on page 25 for more information.
- *Arrays* – an array is an ordered list of one or more scalar variables, each of which could contain either numeric or string data. Arrays have an at-sign (@) prefix and their elements are accessed by an index number that starts with 0 and ends with the last number of the array. Examples of arrays are *@array*, *@lines*, and *@last_names*. See *Arrays* on page 31 for more information.
- *Associative Arrays* – an associative array is a special type of array, one that contains only pairs of scalar variables. The first variable of each pair is the “key” and the keys are used instead of numerical indexes to access the array’s data. Associative arrays have a percent-sign (%) prefix. Examples of arrays are *%passwords*, *%days_of_the_week*, and *%daily_schedule*. See *Associative Arrays* on page 38 for more information.

Unlike other programming languages, Perl does not require you to declare your variables in advance. Instead, the Perl interpreter scans your program to determine

what variables are used so it can allocate and deallocate variable space as needed. Because of this approach, Perl has no way of knowing when you have mistyped a variable's name; if you mistype “*\$datf*” instead of “*\$date*”, Perl does not complain but instead assumes you want to use a new variable. (You can catch many of these errors, though, by using the “-w” command line option; see *Command Line Options* on page 10 for more information.)

Perl also features a number of predefined and special variables that your programs can access to get information such as the script's name, the version of Perl that is being run, and so forth. See *Special and Predefined Variables* on page 48.



Note: Filehandles are a specialized data type; see *File Access in Perl* for their use. Perl 5 also supports object-oriented programming and data types, but their use is beyond the scope of this manual. See the Perl reference manual (online the PanaVue workstation at <http://idDocs/scripts/perl>) for more information.

Scalar Variables

Scalar variables are the basic data type in Perl and have the following characteristics:

- A scalar variable starts with a dollar sign (\$) followed by a letter but other than that, it can use any combination of “*word*” characters (letters, numbers, and the underscore) as part of its name.
- Variable names are case-sensitive, so *\$var*, *\$VAR*, and *\$Var* are different scalar variables.
- A scalar variable can contain either numeric data or string data. Perl uses the context of a variable’s use to determine the type of data it contains. For example, the following line sets the variable *\$var* equal to “12”:

```
$var = "12"; # unknown whether a number or string
```

This assignment is not enough to tell Perl whether you intended to assign the number twelve or a two-character string to *\$var*. This becomes clear only when *\$var* is used in an expression:

```
$var = $var + 1; # $var is being used as a number
$var = $var . "1"; # $var is being used as a string
```

In the first example above, the number one is added to *\$var*, so Perl interprets it as numeric. In the second, the string concatenation operator (.) is used to append the character “1” to *\$var*, so Perl treats its data as a string.

In fact, both of these statements can be used in the same program. You can switch between treating a variable as a string and as a number whenever needed, and Perl interprets the variable’s data accordingly.

- Perl supports the use of both string and numeric constants (called “literals”) when using scalar variables. Numeric literals do not need to be quoted when assigned to a variable (but the quotes can be used in most cases):

```
$number1 = 10; # $number1 contains numeric value of 10
$number2 = 235; # $number2 contains numeric value of 235
$number3 = -20; # $number 3 contains numeric value of -20
$number3 = "-20";# $number 3 still contains numeric -20
```

String literals do not need to be quoted as long as they do not contain any whitespace characters and as long as they do not conflict with any previously defined variable or keyword. Because such conflicts can easily occur, it is recommended you always quote string literals:

```
$string1 = "John"; # $string1 contains the name "John"
$string2 = "Bob"; # $string2 contains the name "Bob"
$string3 = "Mary and I" # $string3 contains the text "Mary and I" (including spaces)
```

- String literals can be quoted either by single quotes (') or double quotes ("). The only difference between the two types of quotes is how special characters are interpreted.

When a string literal is enclosed within single quotes ('), its characters are interpreted exactly as they appear, with only two exceptions:

- the combination of a backslash/single quote (\')
- the combination of two backslashes (\\) is translated to one backslash

See Example 1:

Example 1 Using Literals Within Single Quotes

```
$var = 'hello';     # $var contains 5 characters
$var = 'hello\'';  # $var contains 6 characters, including
                  #     one single quote at end
$var = 'hello\n';  # $var contains 7 characters including
                  #     one backslash and one 'n'
$var = 'hello\\n';# $var contains 7 characters
                  #     (same as above because '\\\' = '\')
```

Double quotes are used whenever you want to specify special characters such as the newline character ("`\n`"). See Example 2:

Example 2 Using Literals Within Single Quotes

```
$var = "hello\n";  # $var contains 6 characters, including
                  #     a final newline character
$var = "hello\t\n";# $var contains 7 characters,
                  #     including a final tab and newline
$var = "hello, \"Jo\"";# $var contains 12 characters,
                  #     including the name Jo in double quotes
```

Table 2 lists the most common special characters that can be used within double quotes. As a general rule, use double quotes for all string literals unless you do not need any of these special characters.

Table 2 Special Characters in Perl (must be double-quoted)

Character	Description	Character	Description
\a	Bell (ASCII 7)	\t	Tab (ASCII 9)
\b	Backspace (ASCII 8)	\0xx	Any octal value between \000 and \0377
\cX	Control Character (where X is any letter from A-Z)	\xff	Any hexadecimal value between 0x00 and 0xff
\f	Formfeed (ASCII 12)	\\	Backslash
\n	Newline (ASCII 10)	\"	Double Quote
\r	Carriage Return (ASCII 13)		

- Perl stores all numeric data in a double-precision floating-point format, but you can use whatever numeric format is most convenient when assigning numbers to scalar variables. See Table 3:

Table 3 Allowable Numeric Data Formats

Format	Description	Examples of Use
x -x	Integer notation	<code>\$word_count = "2";</code> <code>\$days_of_year = 365;</code>
x.xxx ¹ -x.xxx	Decimal notation	<code>\$price = 1.25;</code> <code>\$ratio = "0.114";</code> <code>\$overdraft = "-1.92"</code>
xExx ² xE-xx -xExx -xE-xx	Exponential notation (base 10)	<code>\$byte = 2E8; # 2x10**8</code> <code>\$avogardo="6.023E23"; #Avogardo's number</code> <code>\$num = "-3.1E-23"; # -3.1x10**-23</code> <code>\$num = "-31E-24"; # same number as above</code>
0xxx -0xxx	Octal notation	<code>\$EOL = 0015; # decimal value is 13 (cannot use quotes when specifying octal)</code> <code>\$byte = 0377; # decimal value is 255</code> <code>\$byte = -0377; # decimal value is -255</code>
0xFF -0xFF	Hexadecimal notation	<code>\$byte = 0xFF; # decimal value is 255 (cannot use quotes when specifying hexadecimal)</code> <code>\$word = 0xFFFF; # decimal value is 65535</code> <code>\$word = -0xC000; # decimal value is -49152</code>

1. Since Perl stores all numeric values in the same double-precision floating-point format, assigning a value of "1.00" does not provide a greater degree of precision than assigning a value of "1".
2. The use of quotes is optional when specifying numbers in exponential notation, but quotes cannot be used when specifying numbers in hexadecimal or octal notation.

- Perl itself does not have any limitations as to number size or string size, except whatever limitations are imposed by the computer hardware and operating system. For all practical uses on the PanaVue workstation, strings and arrays have no limitations, but numbers are limited to a maximum of 14 significant digits to the right of the decimal sign in exponential notation.

For example, you might try to set the value of pi to 30 significant digits using the following script, but Perl still prints out 14 digits to the right of the decimal point:

```
$pi = 3.1415926535897932384626433832795;  
print "The value of pi is: $pi\n";  
# prints out "The value of pi is: 3.14159265358979"
```

Arrays

Perl supports arrays in much the same manner as other programming languages, except that you do not have to declare the array and its size in advance. Perl automatically grows and shrinks the array as elements are added and taken from it.

Perl's arrays have the following additional characteristics:

- An array variable starts with an at-sign (@) but other than that, it can use any combination of “*word*” characters (letters, numbers, and the underscore) as part of its name.
- Variable names are case-sensitive, so `@var`, `@VAR`, and `@Var` are all different arrays.
- Elements of the array are referenced by putting a dollar sign (\$) to the front and a subscript within square brackets ([]) to the back of the variable name, creating a new form of scalar variable. The first element of any array is always numbered 0 (zero), so the first elements of the above arrays are `$var[0]`, `$VAR[0]`, and `$Var[0]`.



Note: Do not confuse the scalar variables used to access arrays with other scalar variables that have the same name. The scalar `$var` is totally independent of `$var[0]` or `$var[1]`, which are used to access elements in the `@var` array. To avoid such confusion, it is recommended that you use different names for scalar variables and arrays.

- Like scalar variables, elements of an array can contain either numeric or string data. The elements in an array do not have to have the same type of data; some elements can contain numeric data, other elements can contain strings.

- The easiest way to assign values to an array is with a list, which can be either another array, a set of scalars (literals or variables) within parentheses, or a function or command that returns a list of values. See Example 3 for examples of each method:

Example 3 Using a List to Add Elements to an Array

```
# Adding elements to an array using another array
@array1 = @array2; # @array1 becomes an exact copy of @array2

# Adding elements to an array using a list of scalar values (the scalars can
# be either literals or variables)
@array1 = (1,2,3,4);           # array1 contains four numbers
@array2 = (1,"two",3,"four"); # array2 contains numbers and strings
@array3 = ($a, $b, $c, $d);   # array3 contains the values contained
                              #   in the four scalar variables

# Adding elements to the front or end of an existing array by including the
# array within the list
@array1 = (0,@array1,99);     # 0 is added to beginning, 99 to end

# Adding elements to an array using the output of a function (in this case,
# the split command, which takes a line of input and breaks it into individual
# words that are returned in a list)
$input = "this is a line of input"; # typical input line
@array1 = split($line); # @array1 now contains six elements (words)
# the above two lines are equivalent to doing the following:
@array1 = ("this","is","a","line","of","input");
```

- You can also assign scalar values to the individual elements of an array. See Example 4, where the first four elements of the *@array* are assigned strings:

Example 4 Adding Individual Elements to an Array

```
$array[0] = "first element";  
$array[1] = "second element";  
$array[3] = "third element";  
$array[4] = "fourth element";
```

- The end of the array is indicated by the first element containing the undefined value (a null string or the number 0). This makes it easy to use loops to access all elements in an array, by testing each element until an undefined value is found.

Example 5 shows one way all of the elements in an array could be printed, using a *while* loop that stops only when it reaches an array element that does not have any data in it:

Example 5 Printing the Contents of an Array

```
$index = 0; # set index for first element  
while ($array[$index]) { # as long as array has data  
    print "$array[$index]\n"; # print the array element  
    $index = $index + 1; # point to next element  
} # end while
```



Note: See *Control Structures and Loops* for an explanation of loops such as the *while* loop shown above.

- Another way to find the number of elements in an array is to assign the array to a scalar variable, which then contains the length of the array:

```
$length = @array; # put # of array elements in $length
```

Since the first element of an array is indexed by zero, you must subtract 1 from the length to get the index of the last element of the array. For example, the code in Example 6 assigns the contents of the last element of the *@array* into the *\$last_element* scalar variable:

Example 6 Getting the Last Element of an Array

```
$length = @array;           # $length = number of elements
$last_element = $array[$length - 1]; # get last element
```

- The most efficient way of accessing the last element of an array is by using the special variable *\$#array*, which is the index number of the last element in *@array*. Perl automatically changes *\$#array* whenever the array size changes, so it can always be used to access the last element of the array:

Example 7 Getting the Last Element of an Array with *\$#array*

```
$last_element = $array[$#array];
```

- To add additional elements, simply assign a value to the element at the end of the array; Perl grows the array automatically. As shown in Example 8, the length of an array makes a convenient subscript for adding on new elements to the array:

Example 8 Adding a New Element to An Array

```
$length = @array;           # $length = number of elements
$array[$length] = "new data"; # add a new element
$array[$length+1] = "more data"; # add another new element
```

When new elements are added to an array, the length of the array automatically increases, so this technique can be used repeatedly in loops. See Example 9:

Example 9 Adding Multiple New Elements to An Array

```
while ($input = <STDIN>) { # get new line from STDIN
    $length = @array;      # get current length of array
    $array[$length] = $line; # put input at end of array
} # do this until input ends
```



Note: See *Control Structures and Loops* for an explanation of loops such as the *while* loop shown above. See *File Access in Perl* for an explanation about using STDIN.

The routine shown in Example 9 reads a line of input from the standard input device (STDIN, usually the user's keyboard) and then adds the line to the end of an array. Each time an element is added to the array, its length increases, so each time the while loop executes, the value of the *\$length* variable increases by one.



Note: Elements can also be added to (or removed from) an array using the array operators listed in Table 4, below.

- A number of operators can be used on arrays. The most common ones are shown in Table 4:

Table 4 Array Operators (1 of 2)

Operator	Description
chop	Removes the last character from each element in the array: <pre>@array = ("one", "two", "three"); chop(@array); # @array now = ("on", "tw", "thre")</pre>
push	Adds one or more new entries to the end of an array: <pre>@array = (1,2,3); push(@array,4,5,6); # @array now = (1,2,3,4,5,6)</pre>
pop	Removes and returns the last entry at the end of an array: <pre>@array = (1,2,3,4,5,6); \$last_element = pop(@array); # \$last_element = 6 # @array = (1,2,3,4,5)</pre>
reverse	Returns an array in reverse order, leaving the original array unchanged: <pre>@array1 = (1,2,3,4,5,6); @array2 = reverse(@array1); # @array1 is unchanged # @array2 = (6,5,4,3,2,1)</pre>

Table 4 Array Operators (2 of 2)

Operator	Description
sort	<p>Returns an array sorted in ascending ASCII order, leaving the original array unchanged:</p> <pre>@array1 = (1,"one",2,3,"four",10,20); @array2 = sort(@array1); # @array1 is unchanged # @array2 = (1,10,2,20,3,"four","one")</pre> <p>Note: You can change the sort order by specifying your own sort routine to be used with the <i>sort</i> operator. See the Perl documentation (http://idDocs/scripts/perl) for details.</p>
shift	<p>Removes and returns the first element of an array:</p> <pre>@array = (1,2,3,4,5,6); \$first_element = shift(@array);# \$first_element = 1 # @array = (2,3,4,5,6)</pre>
unshift	<p>Adds one or more elements to the beginning of an array:</p> <pre>@array = (1,2,3,4,5,6); unshift(@array,"a","b","c"); # @array = ("a","b","c",1,2,3,4,5,6)</pre>



Note: The *push* and *pop* operators add and remove elements from the end of an array. The *unshift* and *shift* operators add and remove elements from the beginning of an array.

Associative Arrays

Associative arrays (called “hash tables” or “hash lists” in other programming languages) are a particular type of array designed for database-type operations. Instead of using numerical indexes to access elements of the array, associative arrays use *keys*, which can have any scalar value (numbers or strings).

For example, the first element of a normal array is always indexed by the number 0 (such as `$array[0]`). An element of an associative array, though, can have any value: `$array{"John"}`, `$array{"Jill"}`, `$array{$name}`, `$array{16235}`, or even `$array{0}`.



Note: It is easy to confuse elements of normal arrays with those of associative arrays, but as shown above, elements of associative arrays use curly brackets (`{ }`), while those of regular arrays use square brackets (`[]`).

Associative arrays are useful whenever you want to associate two arbitrary types of data with one another, such as a phone number with a person’s name or a node number with its physical location. Perl optimizes its storage and handling of associative arrays, so they are the fastest way possible in Perl to store and retrieve data like this.

Associative arrays have the following characteristics:

- Associative arrays are prefixed by a percent sign (`%`) instead of the at-sign (`@`) used by regular arrays. Other than that, an associative array can use any combination of “*word*” characters (letters, numbers, and the underscore) as part of its name.

- Variable names are case-sensitive, so `%var`, `%VAR`, and `%Var` are all different associative arrays.
- Elements of an associative array are referenced by putting a dollar sign (\$) to the front and a key word within curly brackets ({ }) to the back of the variable name, creating a new form of scalar variable. Typical elements of the above arrays could be `$var{"robert"}`, `$VAR{"1-212-555-1212"}`, or `$Var{23}`.
- Like other scalar variables, the elements of an associative array can contain any scalar value, numeric or string. The elements in an associative array do not have to have the same type of data; some elements can contain numeric data, other elements can contain strings.
- The keys used to define elements of an array can have any scalar value, numeric or string. The keys are case-sensitive, so `$var{"key"}` refers to a different element than `$var{"KEY"}`.
- The easiest way to assign values to an associative array is with a list, which can be either another array, a set of scalars (literals or variables) within parentheses, or the a function or command that returns a list of values. Unlike regular arrays, though, the input list for an associative array must be properly ordered into pairs, where the first scalar is the key that is used to access the second scalar.

Example 10 gives examples of each way that elements can be added to an associative array:

Example 10 Using a List to Add Elements to an Associative Array

```

# Adding elements to an associative array using another associative array
%array1 = %array2; # %array1 becomes an exact copy of %array2

# Adding elements to an associative array using a list of ordered scalar values
# (the list must be composed of key/value pairs)
%array1 = (1,2,3,4);           # %array1 contains 2 elements
                                # $array{1} = 2, $array{3} = 4

%array2 = (1,"two",3,"four"); # %array2 contains 2 elements
                                # $array{1} = "two", $array{3} = "four"

%array3 = ($a, $b, $c, $d); # %array3 contains 2 elements
                                # $array{$a} = $b, $array{$c} = $d

# Adding elements to an array using the output of a function (in this case,
# the split command, which takes a line of input and breaks it into individual
# words that are returned in a list)
$input = "john x7990 jill x5917 joan x6134"; # typical input line
%extens = split($line); # %extens now contains three elements:
                        # $extens{"john"} = "x7990"
                        # $extens{"jill"} = "x5917"
                        # $extens{"joan"} = "x6134"

```

- To assign scalar values to an individual elements of an associative array, use the appropriate key. If you specify a new key, a new element is added to the array; if you specify a previously used key, that element's previous value is replaced. See Example 11:

Example 11 Adding and Modifying Elements of an Associative Array

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

# Initialize array with two elements
%computers = ("robert", "pc", "judy", "mac");

# Add new elements and modify existing ones
$computers{"jerry"} = "sparc5"; # add new element
$computers{"linda"} = "sparc20"; # add new element
$computers{"robert"} = "sparc20"; # modify element
$computers{"judy"} = "powerbook"; # modify element

print %computers; # print out all keys and element values
```



Note: Perl orders associative arrays in the most efficient internal format for the given keys and computer system. If you print out an associative array as shown in Example 11, you cannot easily predict which elements will be printed first.

- To delete an element from an associative array, use the **delete** operator on the element to be deleted:

```
delete $computers{"robert"}; # this entry no longer exists
```
- Using a previously unknown key with an associative array returns the undefined value. If you were to take the array defined in Example 11 and access `$computers{"william"}`, you would get an undefined value. No indication is given that you have used a previously unknown key, which is inconvenient if you want to access only current elements of the array.

Perl offers a way around this with its **keys** operator, which returns all of the keys used for a particular associative array, which you can then use to access all of the elements in that associative array. The **keys** operator returns the keys in the same order that they are used in the associative array at that time, so there is no guarantee as to how the keys are ordered.

However, since the **keys** operator returns the keys as a regular array, you can access all of the elements in the associative array by manipulating the regular array created to hold all of the keys. See Example 12 for one way to do this:

Example 12 Printing the Contents of an Associative Array

```
%cars = ("alan","ford","jill","toyota","jack","chrysler");

@keys = keys(%cars); # @keys now = ("alan","jill","jack")
foreach $key (@keys) {
    print "$key drives a $cars{$key}\n";
} # end foreach
```



Note: See *Control Structures and Loops* for an explanation of loops such as the **foreach** loop shown above.

The code shown in Example 12 can be shortened by eliminating the use of the *@keys* array, as shown in Example 13:

Example 13 Printing the Contents of an Associative Array (modified)

```
%cars = ("alan","ford","jill","toyota","jack","chrysler");

foreach $key ( keys(%cars) ) {
    print "$key drives a $cars{$key}\n";
} # end foreach
```

- When the **keys** operator is used in a scalar context, it returns the number of keys found, which gives you the number of elements in an associative array:

Example 14 Finding the Number of Elements in an Associative Array

```
%cars = ("alan","ford","jill","toyota","jack","chrysler");

print "The number of cars is ",$length=keys(%cars)," \n";
```

- The counterpart to the **keys** operator is the **values** operator, which returns a list of element values found in an associative array. The list provided by **values** operator is in the same order as the one provided by the **keys** operator, so you could use both to print the associative array. See Example 15:

Example 15 Printing the Values of an Associative Array

```
%cars = ("alan","ford","jill","toyota","jack","chrysler");
@car_type = values(%cars); # create array of values
@driver = keys(%cars); # create array of keys
$index = 0;
```

```
while ($car_type[$index]) {  
    print "$driver[$index] drives a $car_type[$index++]\n";  
} # end while
```

- The most efficient method to loop through an associative array is to use the **each** operator, which returns key/value pair each time it is used. When **each** reaches the end of the associative array, it returns the undefined value, which makes it a perfect match for use with the **while** command.

Example 16 Printing the Values of an Associative Array Using each

```
%cars = ("alan","ford","jill","toyota","jack","chrysler");  
while ( ($driver, $car_type) = each(%cars) ) {  
    print "$driver drives a $car_type\n";  
} # end while
```

- You can convert between an associative array and a regular array by assigning the one to another.

Example 17 Converting Between Associative and Regular Arrays

```
# Converting Associative Array to a Regular Array  
%cars = ("alan","ford","jill","toyota","jack","chrysler");  
  
@cars = %cars; # @cars has six separate elements  
  
# Converting Regular Array to an Associative Array  
@books = ("camel","Programming Perl",  
          "llama","Learning Perl",
```

```
        "rhino", "Javascript",  
        "koala", "HTML, the Definitive Guide");  
%books = @books; # %books now has four elements, indexed  
                # by the type of animal on the cover
```



Note: When converting a regular array to an associative array, the associative array interprets the even elements of the array (those with index numbers 0, 2, 4, and so forth) as the keys and the odd elements (those with indexes 1, 3, 5, and such) as the element values. Be certain this is what you want before using this technique.

Avoiding Confusion with Perl's Variables

Scalar variables, arrays, and associative arrays can have the same names, so do not confuse normal scalar variables with those used to access regular arrays and associative arrays. To avoid confusion, do not give associative arrays, regular arrays, and scalar variables the same names.

Other than that, use the following rules to keep each type of variable straight:

- Scalar variables begin with a dollar sign (\$) and do not have any brackets.
- Arrays begin with the at-sign (@) and their elements have names that include both a dollar sign (\$) and square brackets ([]).
- Associative arrays begin with a percent sign (%) and their elements have names that include both a dollar sign (\$) and curly brackets ({ }).

Table 5 summarizes the major characteristics of Perl's major variable classes (as used in this table, “*string*” can be any combination of *word* characters (letters, numbers, and the underscore), and “*number*” must contain only the digits 0 through 9):

Table 5 Comparing Perl's Variable Types (1 of 2)

Variable Template	Description	Examples
<code>\$string</code>	Scalar variable	<code>\$var</code> , <code>\$VAR</code> , <code>\$Var</code>
<code>@string</code>	Regular array	<code>@array</code> , <code>@ARRAY</code> , <code>@Array</code>
<code>\$string[number]</code>	Regular array element	<code>\$array[0]</code> , <code>\$ARRAY[0]</code> , <code>\$Array[0]</code>

Table 5 Comparing Perl's Variable Types (2 of 2)

Variable Template	Description	Examples
<code>%string</code>	Associative array	<code>%array</code> , <code>%ARRAY</code> , <code>%Array</code>
<code>%string{string}</code>	Associative array element	<code>\$array{2.54}</code> , <code>\$ARRAY{"Bill"}</code> , <code>\$Array{"/opt/Panavue/reports"}</code>

Special and Predefined Variables

Perl offers a wide number of special variables and predefined variables that contain system information of one type or another. You probably will not use more than a handful of these special variables in your own scripts, but when you need some sort of system-level information, you can usually find a special variable that contains it.

Table 6 shows the most commonly used special variables; see the online Perl documentation (<http://idDocs/scripts/perl>) for a complete list.

Table 6 Commonly Used Special and Predefined Variables (1 of 3)

Variable	Description
\$_	The default operator used for many commands, especially input and output commands.
\$/	The input record separator (" <i>ln</i> " by default). When reading input files as text files, Perl returns one line of input at a time, where a line is defined as any string that ends with <i>\$/</i> .
\$]	The Perl major and minor version numbers (for example, "5.0004").
\$!	When error conditions occur, this contains the error number if used in a number context or the error message if used in a string context.
\$	When set to any nonzero value, forces Perl to always use line-buffering for print , printf , and write commands. When set to zero (default), this output is block buffered whenever the script is run as part of a piped process (as is the case with CGI scripts). As a general rule, this variable should be set to 1 in all CGI scripts to avoid block buffering.

Table 6 Commonly Used Special and Predefined Variables (2 of 3)

Variable	Description
Variables used for operating system access	
\$0	The name of the currently executing Perl script.
\$\$	The Unix process ID of the currently executing Perl program.
\$<	The real user ID (uid) of the currently executing process.
\$>	The effective user ID of the currently executing process.
\$(The real group ID (gid) of the currently executing process.
\$)	The effective group ID of the currently executing process.
Variables used for file access (see <i>File Access in Perl</i>)	
\$ARGV	The name of the current file when using the diamond operator (<>) for input.
@ARGV	An array containing the script's command line arguments.
Variables used for regular expressions (see <i>Using Regular Expressions in Perl</i>)	
\$&	The string matched by the last successful match.
\$`	The string preceding the string that was last matched.

Table 6 Commonly Used Special and Predefined Variables (3 of 3)

Variable	Description
\$'	The string following the string that was last matched.
\$1 ... \$9	Represents the appropriate subpatterns when matching and substituting regular expressions.



Note: Table 6 shows the shortcut form of these special variables. Most of these variables also have an “English name” that is more descriptive. For example, “\$ ” can also be referred to as “\$ARG”. See the Perl documentation (<http://idDocs/scripts/perl>) for these alternative names.

Perl Output Commands

Perl uses three main commands for output of text-based information:

- **print** – outputs a list of variables, strings, and numbers with no modification or formatting. By default the output goes to STDOUT, but you can also print to any file you have previously opened for writing.
- **printf** – outputs a formatted list of variables, strings, and numbers to either STDOUT or an output file. This command is almost identical to the *printf* command that is used in the C programming language.
- **write** – outputs one or more lines of variables, strings, and numbers to either STDOUT or an output file, using a previously defined report format. The *write* command can output the output data over multiple lines, depending on the defined format, and keeps track of how many lines have been printed so that the appropriate page numbers can be printed.



Note: Perl offers a number of ways to read and write binary information to files. See the descriptions of the **read**, **seek**, **sysread**, and **syswrite** commands in the online Perl documentation (<http://idDocs/scripts/perl>).

If you are writing CGI scripts, also see *Problems with Buffering of Output* on page 69 for information on a potential problem with how Perl buffers its output for such scripts.

Using the print Command

The **print** command outputs a list to the specified filehandle, and its format is the following:

```
print FILEHANDLE list
```

The *FILEHANDLE* must have been previously opened for writing; if not, **print** returns "0" to indicate it failed to output the *list*. If *FILEHANDLE* is not given, **print** uses the default output device (which is STDOUT, unless changed by the **select** command).



Note: See *File Access in Perl* for a discussion of both filehandles and STDOUT.

The *list* can contain any or all of the following: literals, scalar variables, arrays, associative arrays, and the output from functions, commands, and expressions. Strictly speaking, all items in the list should be separated by Perl's *list operators* (commas), as shown in Example 18:

Example 18 Using the print Statement (strict style)

```
$name = "Roger";  
$car = "Ford";  
$color = "blue";  
print "\n", $name, " owns a ", $color, " ", $car, "\n";  
# prints "Roger owns a blue Ford"
```

.....

This example prints a new line and then the text “Roger owns a blue Ford”, followed by another new line. Fortunately, Perl allows variables to be placed inside the quotes, so this **print** statement could be more easily written as follows:

Example 19 Using the print Statement

```
$name = "Roger";
$car = "Ford";
$color = "blue";
print "\n$name owns a $color $car\n";
# prints "Roger owns a blue Ford"
```

The only time you must use commas is if you put a function or expression in the **print** statement that must be evaluated before being printed. For example, if you wanted to print the value of π from within a Perl program, you might think you can use the statement shown in Example 20:

Example 20 Printing the Value of π (incorrect)

```
print "The value of pi is atan2(1,1)*4\n";
```

However, this statement just prints “The value of pi is atan2(1,1)*4”, without evaluating the expression but just treating it as a string of characters. To force Perl to evaluate the expression, move it out of the double quotes and delimit it with commas. See Example 21:

Example 21 Printing the Value of π (correct)

```
print "The value of pi is ",atan2(1,1)*4,"\n";
```

This line correctly prints “The value of pi is 3.14159265358979.” With few exceptions, if you want the **print** statement to display the output of a command or expression, you must move the command or expression outside double quotes and separate it from the rest of the list using commas.

Since arrays are just another type of list, you can specify them in the **print** command. Each element of the array is then printed in sequence, without any separation. See Example 22:

Example 22 Printing an Array

```
@names = ("john","jill","bob","bruce","jean");  
print "@names\n";
```

The code in Example 22 prints out the following line:

```
johnjillbobbrucejean
```

Using the printf Command

Perl's **printf** command is almost identical to that used in the C programming language; it prints out a list of data that is formatted according to a set of format specifications. The **printf** command has the following format:

```
printf FILEHANDLE formats, list
```

The *FILEHANDLE* must have been previously opened for writing; if not, **printf** returns "0" to indicate it failed to output the *list*. If *FILEHANDLE* is not given, **printf** uses the default output device (which is STDOUT, unless changed by the **select** command).



Note: See *File Access in Perl* for an explanation of filehandles and the STDOUT device.

The *list* is the same as that used in the **print** command (see *Using the print Command* on page 52), but it is formatted according to the format specification, which contains one or more of the format types shown in Table 7:

Table 7 printf Format Types

Format	Description
%c	print the list item as a single character
%d	print the list item as a decimal character
%e	print the list item as an exponential floating-point decimal number
%f	print the list item as a fixed point floating-point decimal number

Table 7 printf Format Types

Format	Description
%g	print the list item as a compact floating-point decimal number
%ld	print the list item as a long decimal number
%lo	print the list item as a long octal number
%lu	print the list item as a long unsigned decimal number
%lx	print the list item as a long hexadecimal number
%o	print the list item as an octal number
%s	print the list item as a string
%u	print the list item as an unsigned decimal number
%x	print the list item as a hexadecimal number

As shown, most of the format types are for numeric data, but if the output list also contains string or character data, you must use a corresponding number of `%s` and `%c` format specifiers.

You can also specify the minimum and maximum sizes of each item with a format like “`%m.nx`”, where *m* specifies the minimum length of the field and *n* specifies the maximum length (except with exponential formats, which use *n* to specify precision).

If the output is too long for the given format, it is truncated (or rounded if numeric). If the output is too short to fill the maximum length, it is right justified, using either

spaces or zeroes, as appropriate. Example 23 shows how a number of different formats would print the value of π :

Example 23 Examples of printf Formatting

```
printf "%s%d%s", "The value of pi is ", atan2(1,1)*4, "\n";
# prints "The value of pi is 3"

printf "%s%3.5d%s", "The value of pi is ", atan2(1,1)*4, "\n";
# prints "The value of pi is 00003"

printf "%s%e%s", "The value of pi is ", atan2(1,1)*4, "\n";
# prints "The value of pi is 3.141593e+00"

printf "%s%6.10e%s", "The value of pi is ", atan2(1,1)*4, "\n";
# prints "The value of pi is 3.1415926536e+00"

printf "%s%f%s", "The value of pi is ", atan2(1,1)*4, "\n";
# prints "The value of pi is 3.141593"

printf "%s%8.9f%s", "The value of pi is ", atan2(1,1)*4, "\n";
# prints "The value of pi is 3.141592654"

printf "%s%g%s", "The value of pi is ", atan2(1,1)*4, "\n";
# prints "The value of pi is 3.14159"
```



Note: When printing dollar amounts, use a format of "%3.2f".

When using **printf**, you must supply the proper number of formatting specifiers because Perl ignores any list items that do not have a corresponding format. If an



incorrect format is given (such as an exponential format for string data), Perl tries its best to interpret the list item in that format, with unpredictable results.



Using the write Command

Perl is the “Practical Extension and Report Language,” so it is not surprising that it includes an easy way to generate formatted reports. Perl uses the **write** command for this purpose, which has the following format:

```
write FILEHANDLE;
```

where *FILEHANDLE* is a filehandle for a file that has been previously opened for writing; if no *FILEHANDLE* is listed, **write** uses the default output device (which is STDOUT, unless changed by the **select** command).



Note: See *File Access in Perl* for an explanation of filehandles and the STDOUT device.

The **write** command does not take a list of items to be printed, as the **print** and **printf** commands do. Instead, the **write** command outputs one or more lines in a predefined format. This format must have the same name as the *FILEHANDLE* being used and includes the list of variables and expressions to be printed.

A format is defined with the **format** command, as shown in Example 24:

Example 24 Defining a Format for the write Command

```
format formatname =      # formatname must = filehandle name
fieldline                # contains formatting information
variable-list          # contains list to use in above line
fieldline                # contains formatting information
variable-list          # contains list to use in above line
.
                        # last line for the format must
```

```
# begin with a period and have  
# nothing else on the line
```

Each *fieldline* describes one line of output, and there can be as many fieldlines as desired. A fieldline can contain literals, variables, or both. A *variable-list* must follow each fieldline that prints variables. The last line of the format must contain only a single period as its first and only character.

Each *fieldline* can contain *placeholders*, *fieldholders*, or both. A *placeholder* is text that is reproduced exactly as it appears and can include anything except the special characters used in *fieldholders*.

A *fieldholder* represents the space taken up by a variable and determines the output format of that variable. The number of fieldholders in a fieldline must match the number of variables in a *variable-list* on the next line. The fieldholders can be surrounded by any number of placeholders, but there must be a one-to-one correspondence between the fieldholders on one line and the variables on the next.

The fieldholder determines the output format of its corresponding variable, using the basic set of format types given in Table 8:

Table 8 Fieldholder Formats

Format	Description
@<<<<<	Left-justified fixed-length field. The length of output is determined by the number of less-than signs (<) plus 1 for that at-sign (@). If the variable requires less space, the output is padded on the right with space; if the variable requires more space, its output is truncated.
@>>>>>	Right-justified fixed-length field. The length of output is determined by the number of greater-than signs (>) plus 1 for that at-sign (@). If the variable requires less space, the output is padded on the left with spaces; if the variable requires more space, its output is truncated.
@	Center-justified fixed-length field. The length of output is determined by the number of pipe signs () plus 1 for that at-sign (@). If the variable requires less space, the output is padded on both sides to fill out the field and keep it center; if the variable requires more space, its output is truncated.
@#####.##	<p>Fixed-precision numeric field. The minimum number of digits before the decimal point is determined by the at-sign (@) and the number of number signs (#) before the decimal point. If the number has more digits, they are still printed; if the number has fewer digits, the number is padded on the left with spaces.</p> <p>The number signs (#) after the decimal point determines the precision of output. If the number has more digits than this after the decimal point, the number is rounded up or down. If the number has fewer digits, it is padded with zeroes on the right.</p>

For example, Example 25 shows a script that writes a very simple report to a file named “*phone.lst*” (see *File Access in Perl* for information about opening and closing files).

A format named “PHONES” is defined for this report, and it specifies that each line in the report contains a name and the corresponding phone number; the name of each person is allocated 11 spaces and the phone number is allocated eight spaces. Spaceholders include the text “*Name:* ” and “*Extension:* ”, which precede the two fieldholders.

Example 25 Sample Report

```
%phones = ("Alan","5990","Betty","4301",
           "Bob","6100","Cathy","7132",
           "Doris","2395","Edith","6175",
           "Frank","5986","Geena","5810",
           "Harry","6323","Jane","7788");

open(PHONES,">phone.lst") ||
    die("Could not open 'phone.lst' for writing.\n");

foreach $name (sort(keys %phones)) {
    write PHONES;
}

close(PHONES);

format PHONES =
Name: @<<<<<<<<<<<<      Extension: @<<<<<<<<
      $name,                  $phones{$name}
.
```



Note: As shown in Example 25, the last line of a format definition must contain only a single period. No other characters can appear on this line, including comments; otherwise, Perl generates an error message.

When the program in Example 25 is run, it generates the following output:

Example 26 Sample Report Output

```
Name: Alan           Extension: 5990
Name: Betty          Extension: 4301
Name: Bob             Extension: 6100
Name: Cathy           Extension: 7132
Name: Doris           Extension: 2395
Name: Edith           Extension: 6175
Name: Frank           Extension: 5986
Name: Geena           Extension: 5810
Name: Harry           Extension: 6323
Name: Jane            Extension: 7788
```

Writing to Multiple Lines

The simple script shown in Example 25 used an array defined within the script as its data source. Typically, though, reports involve reading a data file of some type and extracting the desired information from it, putting that information into the proper variables, and using the *write* command to generate a formatted report.

Although sometimes the data from a file can fit on just one line, it often must be split across multiple lines. You can accommodate this data by using a caret (^) instead of an at-sign (@) when specifying your formats; repeat the format specification on as many lines as needed.



Note: The backticks (‘) are used in Example 32 to get the output from the Solaris **date** command. See the online Perl reference guide (<http://idDocs/scripts/perl>) for more information about executing system commands.

This produces a page header similar to the following for each page of the report:

Example 33 Typical Page Header Output

```
Phone Number List for Thu Oct 17 15:37:31 PDT Page 1
```

As shown in Example 32, you can print the page number by using the special page numbering variable (`$%`). Perl automatically increments this variable each time a new page is printed.

Problems with Buffering of Output

The **print**, **printf**, and **write** commands work slightly differently, depending on how the Perl script is run:

- When the script is run at the command line, the output of these commands is *line buffered* – characters are sent to the console or output file whenever a newline character (“\n”) is encountered.
- When the script is run as part of a pipe process (such as being run as a CGI script), the output of these commands is *block buffered* – characters are sent to the output device or file only when a complete block of characters is available. (The definition of a block depends on whether the output device is a file, output device, another process, etc.)

This difference usually does not matter except when CGI scripts use the **system** command in addition to the **print**, **printf**, and **write** commands. A common mistake is for a script to use the **print** command to generate the required HTML headers and then use **system** to run another command that displays its own output. Because the **print** command’s output is buffered, the **system** command will end up sending its output out first, before the HTML headers, resulting in an error message from the web server.

For example, Example 34 shows a script that attempts to use the Solaris **date** command to display the current date and time. However, although this script works at the command line, it fails as a CGI script because the **print** output is buffered and the HTML headers are sent out after the output from the **date** command.

Example 34 Script with Buffering Problems

```
#!/usr/bin/perl

print "Content-type: text/html\n\n";
print "<HTML><HEAD><TITLE>Current Date and Time</TITLE></HEAD>\n";
print "<BODY>\n";
print "<PRE>\n";

system("/usr/bin/date");

print "</PRE>\n";
print "</BODY></HTML>\n";
```

You can simulate what happens with this CGI script by running it as part of a pipe from the command line. For example, if this script is named *show_date.pl*, give the following command at the command line:

```
show_date.pl | cat
```

This produces output similar to that shown in Example 35 (note that the output from the **date** command is shown before that of the **print** commands, even though the **print** commands are run first in the script).

Example 35 Simulating a CGI Script's Pipe Behavior

```
% ./test.flush.pl | cat
Mon Jun 15 10:43:13 PDT 1998
Content-type: text/html

<HTML><HEAD><TITLE>The Current Date and Time</TITLE></HEAD>
```

```
<BODY>
<PRE>
</PRE>
</BODY></HTML>
```

This problem can be solved by setting a special variable (`$|` or `$OUTPUT_AUTOFLUSH`) to any non-zero value to force line-buffering of the `print`, `printf`, and `write` commands. Example 36 shows the same script, but now because `$|` is set to a nonzero value it will work as a CGI script.

Example 36 Script without Buffering Problems

```
#!/usr/bin/perl

$| = 1; # force line-buffering of print commands
# You could also use "$OUTPUT_AUTOFLUSH = 1"

print "Content-type: text/html\n\n";
print "<HTML><HEAD><TITLE>Current Date and Time</TITLE></HEAD>\n";
print "<BODY>\n";
print "<PRE>\n";

system("/usr/bin/date");

print "</PRE>\n";
print "</BODY></HTML>\n";
```

Operators in the Perl Language

Perl has borrowed operators from every major programming language, so they should be familiar to anyone with previous programming experience. Unless otherwise noted, these operators work on all variable types and data types, but there are different sets of operators that should be used only on numeric data and on string data.



Note: This section can give only a brief description of these operators. For complete information see the online Perl documentation (<http://idDocs/scripts/perl>).

Assignment

The basic assignment operator is the equal sign (=). In a simple assignment, the variable that is on the left side is assigned the value of whatever is on the right (which can be a numerical literal, a string literal, a set of values, another variable, or the output of a subroutine):

Example 37 Simple Assignments

```
$number = 4;
$string = "Hello, world";
@array = (1,2,3,4);
$var1 = $var2;
$answer = &get_answer;
```

Typically, the equal sign is combined with one of the operators shown in Table 9. The examples given in the table use numeric and string literals (such as *1* and *"dog"*), but variables can be used as well.

Table 9 Assignment Operators (1 of 3)

Operator	Description	Examples
Basic Arithmetic Operators		
+	addition	\$a = 2 + 3; # \$a becomes 5
-	subtraction	\$a = 6 - 2; # \$a is now 4
*	multiplication	\$a = 3 * 4; # \$a is now 12

Table 9 Assignment Operators (2 of 3)

Operator	Description	Examples
/	division	<code>\$a = 12/4; # \$a is now 3</code>
**	exponentiation	<code>\$a = 3**4; # \$a is 81 (3 to the fourth power)</code>
%	modulo division	<code>\$a = 10%3; # \$a is 1 (the remainder of 10/3)</code>
Binary (Bitwise) Operators (you should be familiar with binary numbers and binary logic to use these operators)		
&	bitwise AND (binary AND)	<code>\$a = 8 & 2; # \$a is 0</code> <code>\$a = 20 & 4; # \$a is 4</code>
	bitwise OR (binary OR)	<code>\$a = 8 2; # \$a is 10</code> <code>\$a = 20 4; # \$a is 20</code>
^	bitwise XOR (binary XOR)	<code>\$a = 8 ^ 2; # \$a is 10</code> <code>\$a = 20 ^ 4; # \$a is 16</code>
~	bitwise NOT (binary negation)	<code>\$a = ~8; # \$a is 4294967287 (the 32-bit one's complement of 8)</code>
>>	bitwise shift right (binary shift right)	<code>\$a = 15>>2; # \$a is 3</code> <code>\$a = 4>>3; # \$a is 0</code>
<<	bitwise shift left (binary shift left)	<code>\$a = 15<<2; # \$a is 60</code> <code>\$a = 4<<3; # \$a is 32</code>

Table 9 Assignment Operators (3 of 3)

Operator	Description	Examples
String Operators		
.	string concatenation	<code>\$a = "dog" . "cow"; # \$a is now "dogcow"</code> <code>\$a = "a" . "b" . "c"; # \$a is now "abc"</code>
x	string multiplication	<code>\$a = "abc" x 4; # \$a is now "abcabcabcabc"</code> <code>\$a = "moof" x 2; # \$a is now "moofmoof"</code>



Note: A common mistake in Perl programming is to use arithmetic operators on string data. Although this is allowed, it usually produces unpredictable results; at best, the string is treated as the number 0 (zero), but this is not guaranteed in all cases.

Common Shortcuts

The operators shown in Table 9 are most commonly used to modify the existing value of a variable, as is shown in Example 38:

Example 38 Standard Assignment Notation

```
$number = $number + 4;
$string = $string . "Hello, world";
$var1 = $var1/$var2;
$bits = $bits << 2;
```

Because this is such a common use of these operators, Perl supports a shortcut notation for it, by appending the equal sign (=) to the operator and eliminating the second instance of the variable being changed. See Example 39:

Example 39 Shortcut Assignment Notation

```
$number += 4;  
$string .= "Hello, world";  
$var1 /= $var2;  
$bits <<= 2;
```

**Autoincrement
and
Autodecrement**

One further shortcut is with the autoincrement and autodecrement operators (“++” and “--”), which are shorthand for the following common operations:

```
$a = $a + 1;  
$a = $a - 1;
```

Using the autoincrement and autodecrement operators, these commands become:

```
++$a;  
--$a;
```

The autoincrement and autodecrement operators can be appended either before or after the variable to which they refer. Although this is not important in the examples shown above, it becomes an important consideration in more complicated expressions because the placement determines when the increment or decrement happens.

Example 40 shows four examples of using the autoincrement and autodecrement operators. In the first two cases, the *\$a* variable is modified before its value is assigned to *\$b*, but the reverse happens in the final two cases:

Example 40 Using the Autoincrement and AutoDecrement Operators

```
$a = 10;
$b = ++$a;
# $a=11, $b=11 because increment happens before assignment

$a = 10;
$b = --$a;
# $a=9, $b=9 because decrement happens before assignment

$a = 10;
$b = $a++;
# $a=11, $b=10 because increment happens after assignment

$a = 10;
$b = $a--;
# $a=9, $b=10 because decrement happens before assignment
```

Unlike the other arithmetic operators, the autoincrement operator also works with string variables in the following circumstances:

- The variable has been used only in string contexts (it has not had any arithmetic operations performed on it nor has it been assigned numeric data).
- The string is alphanumeric only, containing only letters and digits, but not spaces, punctuation, or any special characters.

Under these circumstances, strings are autoincremented by advancing the rightmost character by one letter or digit. When a number is incremented by 9, it rolls over back to 0; when a letter is incremented past “z,” it wraps around to “a.” In both cases, the rollover increments the character in the next column to the left. If there is no next column, a new one is created.

The script in Example 41 demonstrates this process:

Example 41 Autoincrementing a String Variable

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

$string = "Z89";

for ($i = 1; $i <= 100; $i++) {
    print $string++, "\n";
}
```

When this script it runs, its first dozen lines are the following:

Example 42 Output from Autoincrementing a String

```
Z89
Z90
Z91
Z92
Z93
Z94
Z95
Z96
Z97
Z98
Z99
AA00
```

Unfortunately, the autodecrement operator cannot be used on strings in this manner.

Relational

As with most languages, Perl offers a wide array of relational operators that are most commonly used as part of tests for control structures such as “*if*” and “*while*” statements (see *Control Structures and Loops*). Perl is relatively unusual, though, in that it uses different operators for numeric and string comparisons; this is necessary because variables are not predefined in advance and can be used for either data type at will.

Table 10 lists the common Perl relational operators, and unless otherwise noted the operators return **1** if the expression is true and the undefined value (interpreted as a zero in numerical calculations and the null string (“”)) in string operations) if not.



Note: A large number of operators are available for testing for the presence and type of files on the PanaVue workstation. See *File Access in Perl* for information on those operators.

Table 10 Relational Operators (1 of 3)

Operator	Description	Examples
Numerical Relational Operators		
==	numeric equality	if (\$a==\$b) { print "\$a is equal to \$b"; }
!=	numeric inequality	if (\$a!=\$b) { print "\$a is not equal to \$b"; }
>	numeric greater than	if (\$a>\$b) { print "\$a is greater than \$b"; }

Table 10 Relational Operators (2 of 3)

Operator	Description	Examples
<	numeric lesser than	if (\$a<\$b) { print "\$a is less than \$b"; }
<=	numeric lesser than or equal	if (\$a<=\$b) { print "\$a is less than or equal to \$b"; }
>=	numeric greater than or equal	if (\$a>=\$b) { print "\$a is greater than or equal to \$b"; }
<=>	numeric compare	\$a <=> \$b returns 0 if \$a == \$b 1 if \$a > \$b -1 if \$a < \$b
String Relational Operators		
eq	string equality	if (\$name1 eq \$name2) { print "\$name1 is equal to \$name2"; }
ne	string inequality	if (\$name1 ne \$name2) { print "\$name1 is not equal to \$name2"; }
gt	string greater than	if (\$name1 gt \$name2) { print "\$name1 is greater than \$name2"; }
lt	string lesser than	if (\$name1 lt \$name2) { print "\$name1 is less than \$name2"; }
ge	string lesser than or equal	if (\$name1 ge \$name2) { print "\$name1 is greater than or equal to \$name2"; }

Table 10 Relational Operators (3 of 3)

Operator	Description	Examples
le	string greater than or equal	if (\$name1 le \$name2) { print "\$name1 is less than or equal to \$name2"; }
cmp	string compare	\$name1 cmp \$name2 returns 0 if \$name1 eq \$name2 1 if \$name1 gt \$name2, -1 if \$name1 lt \$name2
String Transformation Operators¹		
=~	match or substitution found	if (\$name1 =~ /"John"/) { print "\$name1 matches the name 'John'.\n"; } if (\$name1 =~ s/"John"/"Bob"/) { print "changed 'Bob' for 'John'\n"; }
!~	match or substitution not found	if (\$name1 !~ /"John"/) { print "\$name1 does not match the name 'John'.\n"; } if (\$name1 !~ s/"John"/"Bob"/) { print "Did not change 'Bob' for 'John'\n"; }

1. See *Using Regular Expressions in Perl* for more information on these operators.

Logical

Like the relational operators, Perl's logical operators are used primarily as part of the tests used in control structures such as “*if*” and “*while*.” These operators are commonly used when attempting to match a line of input, as shown in Table 11, but they can be used wherever an expression evaluates to either FALSE (the number 0 (zero), the string "0" (zero), or a null string ("")) or to TRUE (anything else).

Table 11 Logical Operators

Operator	Description	Examples
<code> </code>	logical OR	<pre>if (/Bob/ /John/) { print "Found either Bob or John.\n"; }</pre>
<code>&&</code>	logical AND	<pre>if (/Bob/ && /John/) { print "Found both Bob and John.\n"; }</pre>
<code>!</code>	logical NOT	<pre>if !(/Bob/ /John/) { print "Did not find either Bob or John.\n"; }</pre>

Control Structures and Loops

Perl includes the major control structures that are used in other programming languages. Each control structure performs the same basic function: it executes a block of statements if a particular condition exists or until or unless a particular condition changes. The program then continues on with an alternative control block, if it exists, or otherwise continues on with the rest of the program.

- The “*if*” loop performs a block of statements if its test evaluates to true; otherwise an “*else*” block of statements are performed, if they exist.
- The “*unless*” loop performs one or more statements if its test does not evaluate to true; otherwise an “*else*” block of statements are performed, if they exist.
- The “*while*” loop repeats its block of statements as long as its test remains true. If the test ever fails, the *while* loop exits.
- The “*for*” loop performs a block of statements until a predefined expression is true (usually a control variable is incremented past a preset value).
- The “*foreach*” loop performs a block of statements for each member of a given list or array.

Each type of structure uses curly brackets ({}) to delimit its blocks of statements. Any type of statement is allowed within these blocks, including other control structures.

if/else Command

The **if/else** statement has the following form:

Example 43 The if/else Statement

```
if (some-expression) {  
    one-or-more-statements;  
} # end of if  
else {  
    one-or-more-statements;  
} # end of else
```

The **if** statement evaluates the expression within the parentheses and if it evaluates to a true value (non-zero and non-null), the first block of statements is executed. If the expression evaluates to the undefined value (either "" or "0"), the second block of statements (the *else* statement) is executed.

For example, the **if/else** clause in Example 44 determines which line to print on the basis of whether two variables are equal or not:

Example 44 Sample if/else Statement

```
if ($a == $b) {  
    print "The two variables are equal.\n";  
} else {  
    print "The two variables are NOT equal.\n";  
}
```

The **else** statement is optional; if it does not exist, the **if** statement simply falls through to the rest of the script if the expression inside its parentheses is not true.

Because you sometimes have to test for more than one possible value, the *else* statement can also have the form “**elsif**”, which is shorthand for “*else if*”. Example 45 shows an **if** statement that has two **elsif** clauses and one **else** clause:

Example 45 Sample if/else Statement

```
if ($a > $b) {
    print "$a is greater than $b.\n";
} # end if
elsif ($a < $b) {
    print "$a is less than $b.\n";
} # end elsif
elsif ($a == $b) {
    print "The two variables are equal.\n";
} # end elsif
else {
    print "This line should never be printed because all ";
    print "possibilities have already been accounted for.\n";
} # end else
```



Note: Example 45 gives an example of testing for a possibility that should never occur, in this case *\$a* not being equal to, less than, or greater than *\$b*. It is recommended you get into the habit of testing for all situations unless you are absolutely certain of your program’s logic.

unless/else Command

The **unless/else** statement is the logical opposite of the **if/else** statement: whereas **if/else** executes the first block of statements only when the expression is true, **unless/else** executes the first block only when the expression is not true. For example, Example 46 is the same as Example 44 but with the logic reversed so it can use an **unless/else**:

Example 46 Sample unless/else Statement

```
unless ($a == $b) {  
    print "The two variables are NOT equal.\n";  
}  
else {  
    print "The two variables are equal.\n";  
}
```

The choice of which type of control structure to use is a matter of personal preference, although **unless/else** is typically used when you are looking for a special case and want to exclude the vast majority of alternative possibilities.

For example, if you wanted to compare two arrays to see if they are identical, you could use **unless/else** to first exclude all arrays that do not have the same length. This avoids having to do an element-by-element comparison of what could be very lengthy arrays unless there is a chance that the two arrays would match.

Example 47 gives one possible way that this could be done using **unless/else** statements:

Example 47 Comparing Two Arrays

```
unless (@array1 == @array2) {
    print "The two arrays are not the same length.\n";
} # end unless
else {
    $array1 = join(" ",@array1); #convert arrays to strings
    $array2 = join(" ",@array2);
    unless ($array1 cmp $array2) {
        print "The two arrays are equal.\n";
    } else {
        print "The two arrays are not equal.\n";
    } # end else
} # end else
```

while Command

The **while** command specifies that a block of statements should be continuously repeated as long as a specified condition exists. The **while** loop ends only when that condition is no longer true.

The most common use of the **while** statement is in reading input from a user or from a file. As long as input exists, the **while** loop remains in force, allowing that input to be processed, but as soon as the input ends, the **while** loop exits, allowing the rest of the program to continue.

For example, Example 48 shows a program that simply echoes all input from STDIN (typically the user's keyboard) back to the user. The program ends only when the user types a CTRL-D, which is the EOF (end of file) character.

Example 48 Typical while Statement

```
#!/usr/thirdParty/perl/bin/perl -w
while ($_ = <STDIN>) { # get input from the user
    print $_;         # print it back to the user
} # end while
```



Note: See *File Access in Perl* for information about using STDIN to get input from the user.

Be cautious in devising the expression that controls the **while** loop – if the expression immediately evaluates to false, the **while** loop is never executed; the program just ignores the statements in the **while** loop and continues on with the rest of the

program. Conversely, if the expression never evaluates to false, the **while** loop becomes an infinite loop.

Example 49 shows an example of each type of loop:

Example 49 Poorly Written while Loops

```
# Example of an while loop that is never executed
$a = 0;          # set $a to zero
while ($a > 0) {
    print "$a "; # these lines are never executed
    $a = $a + 1;
}

# Example of an endless while loop
$a = 1;          # set $a to 1
while ($a > 0) { # this loop continues forever
    print "$a ";
    $a = $a + 1;
}
```

The first **while** statement executes its block of statements only when a is greater than zero. Since a is initialized to zero, this test fails and the **while** loop's statements are never run.

Similarly, the second **while** statement shows a loop that never ends because the statement being executed within the block makes sure that a is always greater than zero. When devising your own **while** loops, be sure that they do not get caught in endless loops such as this.

for Command

Perl's **for** statement is written almost identically as in the C programming language:

Example 50 Syntax of the for Statement

```
for ( initial-expression; test-expression; increment ) {  
    block-of-statements;  
} # end for
```

The *initial-expression* can be either any valid Perl expression but typically it is an expression that sets a variable of some type to a known value. The *test-expression* must evaluate to true (i.e. non-zero and non-null) before the block of statements can be executed; typically it involves some form of the *initial-expression*. The *increment* is an expression that is executed each time through the **for** loop, and typically it involves changing the *initial-expression*.

For example, the script shown in Example 51 reads any number of lines from STDIN and puts each line into an array. A **for** loop is then used to print out each line.

Example 51 Example Use of the for Statement

```
#!/usr/thirdParty/perl/bin/perl -w  
@array = <STDIN>; # read all lines at once  
for ( $i = 0; $array[$i]; $i++ ) {  
    print "Line $i is: $array[$i]\n";  
} # end for
```



Note: See *File Access in Perl* for information about using STDIN to get input.

In Example 51 the *test-expression* was simply “`$array[$i]`”, which evaluates to true when that array element contains any data. However, when the end of the array is reached, the array elements contain the null string (“”), so this expression evaluates to false and the **for** loop ends.

Like the **while** loop, the *test-expression* in the *for* statement is evaluated at the beginning of the loop; if the *test-expression* evaluates to false immediately, none of the lines in the loop are ever executed. Similarly, if the *test-expression* never evaluates to false, the **for** loop continues forever.

Example 52 shows an example of both types of **for** loops; these examples are the same shown in Example 49 (page 89), except that the **while** loops in the previous example have been rewritten as **for** loops.

Example 52 Poorly Written for Loops

```
# Example of a for loop that is never executed
for ($a = 0; $a > 0; $a+1) {
    print "$a "; # this statement is never executed
}

# Example of a for loop that never ends
for ($a = 1; $a > 0; $a+1) {
    print "$a "; # this loop never ends
}
```



Note: As shown in Example 52, a **while** loop can be usually be easily rewritten as a **for** loop and vice versa. Which type of loop you use is usually a matter of personal preference and programming style. As a general rule, though, if the loop's exit depends on an incremental type of expression (such as "a++"), use **for**; otherwise, use **while**.

foreach Command

Perl's **foreach** statement is written almost identically as in the C language:

Example 53 Syntax of the foreach Statement

```
foreach $element (@list) {  
    block-of-statements-to-do-for-each-element;  
} # end foreach
```

The *\$element* variable is any scalar variable, and it becomes a placeholder for each element in the given *@list*, which can be an array, a literal list, or anything that evaluates to a list. The **foreach** statement executes the block of statements within the loop for each element of the list, substituting the actual element of the array for the *\$element* variable.

For example, Example 54 shows a script that reads in a series of lines from STDIN and uses two **foreach** loops; the first **foreach** loop uses the **tr** (translate) function to convert each line to uppercase, and the second loop prints out the entire array.

Example 54 Typical foreach Loop

```
#!/usr/thirdParty/perl/bin/perl -w  
@array = <STDIN>; # read all of the lines at once  
  
foreach $string (@array) { # converts array to uppercase  
    $string =~ tr/a-z/A-Z/; # one line at a time  
} # end foreach  
foreach $string (@array) { # array is now in uppercase
```

```
    print "$string";  
} # end foreach
```

The code in Example 54 is equivalent to the following two **for** loops:

Example 55 Using for Instead of foreach

```
#!/usr/thirdParty/perl/bin/perl -w  
@array = <STDIN>;  
for ($i=0; $array[$i]; $i++) {  
    $array[$i] =~ tr/a-z/A-Z/; # convert to uppercase  
} # end for  
for ($i=0; $array[$i]; $i++) {  
    print "$array[$i]";  
} # end for
```

It might appear that the **foreach** loop is redundant and unnecessary, but it offers two major advantages over **for**:

- The **foreach** loop can access any list, including lists that are not in predefined arrays. Example 56 shows **foreach** being used to convert a line to uppercase on a word-by-word basis, without having to first put the words in an array:

Example 56 Using foreach on a Non-Array List

```
#!/usr/thirdParty/perl/bin/perl -w  
while (<>) {  
    foreach $word (split()) { # split line into words  
        $word =~ tr/a-z/A-z/; # convert to uppercase  
        print "$word\n";  
    }  
}
```

```
} # end foreach  
} # end while
```

The technique shown in Example 56 is often used when creating *filters*, which are programs that accept lines of input, usually from STDIN, process each line in some way, and output the lines, usually to STDOUT.

- The second advantage of **foreach** loops is that the *@list* array in a **foreach** loop can be processed by an array function before being used. The most common functions used in this way are the **reverse** and **sort** commands.

For example, the script in Example 56 could be easily modified so that the words in each line are sorted before they are converted to uppercase and printed. See Example 57:

Example 57 Using foreach on a Sorted Array

```
#!/usr/thirdParty/perl/bin/perl -w  
while (<>) {  
    foreach $word (sort split()) { # split and sort words  
        $word =~ tr/a-z/A-z/; # convert to uppercase  
        print "$word\n";  
    } # end foreach  
} # end while
```

To reverse the order of the words, use the **reverse** command instead of **sort** in the third line of the above script.

Perl's Built-In Functions

Perl contains a great many built-in functions for the convenience of the programmer. The most commonly used functions are shown in this chapter, divided up into the following categories:

- arithmetic functions – perform the standard mathematical and trigonometric functions
- timekeeping functions – return and process the time and date values
- string functions – process strings in some way

See the Perl reference guide that is online the PanaVue workstation at <http://idDocs/scripts/perl/htmldocs> for more information on using these functions and for a list of the other, less commonly used functions.

Arithmetic Functions

The following arithmetic expressions should be used only on numeric data, as using them on string values can produce unpredictable results. Unless otherwise specified, *expression* must be a scalar variable or expression that evaluates to a numeric value.

abs `abs(expression)` – Returns the absolute value of *expression*:

```
$a = 1;  
$a = abs($a); # $a = 1  
$a = -1;  
$a = abs($a); # $a = 1
```

atan `atan2(y,x)` – Returns the arctangent of *X/Y* in the range of $-\pi$ to π :

```
$pi = atan2(1,1)*4; # get the value of pi
```

cos `cos(expression)` – Returns the cosine, in radians, of *expression*:

```
$a = cos(3.14159); # $a is (approximately) -1
```

exp `exp(expression)` – Returns *e* to the power of *expression*.

```
$a = exp(0); # $a = 1  
$a = exp(1); # $a is (approximately) 2.7182818
```

hex `hex(expression)` – Returns the decimal value of *expression*, as interpreted as hexadecimal; *expression* should be a string that contains only digits and the letters

“A” through “F”. If the string starts with “0x”, use the *oct* function (see below), to convert it to hexadecimal.

```
$a = hex(FF30); # $a is 65328 decimal
```

int *int(expression)* – Returns the integer portion of *expression* (rounded down to the nearest integer):

```
$a = int(exp(1)); # $a is 2
```

If you want to round a number up or down to the nearest integer, add “0.5” to the *expression*:

```
$a = int( exp(1) + 0.5 ); # $a is 3
```

log *log(expression)* – Returns the natural logarithm (base **e**) of *expression*.

```
$a = log(1); # $a is 0
```

oct *oct(expression)* – Returns the decimal value of *expression*, as interpreted as either an octal or hexadecimal string. To be interpreted in octal, *expression* must be a string that starts with “0” and contain only the digits 0 through 7:

```
$a = oct(100); # $a = 64 decimal  
$a = oct(377); # $a = 255 decimal
```

To be interpreted in hexadecimal, *expression* must be a string that starts with “0x” and contain only the digits 0 through 9 and letters “A” through “F”:

```
$a = oct("0x100"); # $a = 256 decimal  
$a = oct("0xF77"); # $a = 2959 decimal
```

rand `rand(expression)` – Returns a random number (with up to 14 significant digits) between 0 and *expression*. If *expression* is not given, it is assumed to be 1:

```
$a = rand; # $a can be anything between 0 and 1  
$a = rand(100); # $a can be anything between 0 and 100
```

Use the *srand* function (below) to increase the randomness of this function.

sin `sin(expression)` – Returns the sine, in radians, of *expression*.

```
$a = sin(3.14159); # $a is (approximately) zero
```

sqrt `sqrt(expression)` – Returns the square root of *expression*.

```
$a = sqrt(4); $a = 2
```

srand `srand(expression)` – Seeds the random number generator and should be used before using *rand* (see above). If no *expression* is given, *srand* uses whatever is currently returned by the *time* function (see below)

A good *expression* to use is `srand(time | $$)`, which uses the result of a bitwise OR between the current time and the script's unique process ID (pid):

```
srand( time | $$ ); # randomize  
$a = rand(1000); # $a can be anything between 0 and 1000
```

Timekeeping Functions

time `time` – Returns the number of seconds since January 1, 1970 Greenwich Mean Time (GMT), not counting the dozen or so *leap seconds* that have been added since then.

```
$a = time; # $a contains # of seconds from 1/1/70
```

localtime `localtime(expression)` – Converts the *expression* (number of seconds since January 1, 1970) into a 9-element array showing the equivalent local time (as determined by the workstation's system clock). This function is typically used with the output of the *time* function:

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime(time);  
print "Today's date is $mon/$mday/$year\n";  
# prints a line like "Today's date is 11/15/96"  
print "The time is $hour:$min:$sec\n";  
# prints a line like "The time is 12:13:43"
```

gmtime `gmtime(expression)` – Converts the *expression* (number of seconds since January 1, 1970) into a 9-element array showing the equivalent Greenwich Mean Time (GMT). The typical use is with the output of the *time* function:

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = gmtime(time);  
print "Today's date is $mon/$mday/$year\n GMT";  
# prints a line like "Today's date is 11/15/96 GMT"  
print "The time is $hour:$min:$sec GMT\n";  
# prints a line like "The time is 12:13:43 GMT"
```

String Functions

Unless otherwise specified, all arguments to the string functions are scalar variables containing non-null strings.

chomp *chomp(expression)* – Removes the line ending (typically the “*n*” characters) from the specified string, or if an array is specified, from all strings in the array. This function modifies the string or array given and returns the number of characters that were chomped.

```
$string = "this is a line\n";  
$a = chomp($string); # $a = 1 and $string has no newlines
```

chop *chop(expression)* – Removes the last character of the specified string, or if an array is specified, removes the last character of all elements in the array. Like *chomp*, this function modifies the string or array given, but unlike *chomp*, it removes any character that is at the end of the string; it also returns the character that was chopped.

```
$string = "this is a line\n";  
$a = chop($string); # $a = "\n" and $string = "this is a line"  
$a = chop($string); # $a = "e" and $string = "this is a lin"
```

chr *chr(expression)* – Returns the ASCII character (as a one-character string) specified by *expression* (which should be between 0 and 255 for most uses).

```
$a = chr(65); # $a = "A"
```

index `index($str,$substr,off)` – Returns the index position (zero-based) of the first occurrence of the `$substr` string in the `$str` string at or after the given offset (`off`, which should be numeric). If `off` is omitted, it defaults to 0, which means the string is searched from its first character. Returns -1 if the substring is not found.

```
$string = "This is a line of data\n";  
$a = index($string,"data",0); # $a = 18  
$a = index($string,"not here"); # $a = -1
```

length `length($string)` – Returns the length, in characters, of the given string. This count includes any special characters such as newlines and carriage returns:

```
$string = "This is a line of data\n";  
$a = length($string); # $a = 23 (the newline is included)
```

lc `lc($string)` – Returns the specified string in lowercase (does not change the original string).

```
$string = "THIS IS A LINE OF DATA\n";  
$a = lc($string); # $a = "this is a line of data\n"
```

lcfirst `lcfirst($string)` – Returns the specified string with its first character converted to lowercase (does not change the original string).

```
$string = "THIS IS A LINE OF DATA\n";  
$a = lcfirst($string); # $a = "tHIS IS A LINE OF DATA\n"
```

ord `ord($string)` – Returns the ASCII numeric value of the first character in the specified string.

```
$string = "THIS IS A LINE OF DATA\n";  
$a = ord($string); # $a = 84 (the ASCII value of "T")
```

rindex `rindex($str,$substr,off)` – This is “reverse index” (see *index* above) and returns the index position (zero-based) of the last occurrence of the *\$substr* string in the *\$str* string from the given offset (*off*, which should be numeric). If *off* is omitted, it defaults to the length of the string, which means the string is searched backwards from its last character. Returns -1 if the substring is not found.

```
$string = "This is a line of data\n";  
$a = rindex($string,"is"); # $a = 5  
$a = rindex($string,"not here"); # $a = -1
```

substr `substr($str,off,length)` – Returns a string that is a substring of the specified *\$str* string at the given offset (*off*, which should be numeric) and of the given *length* (which should be numeric). If *length* is omitted, the substring is from the given offset to the end of the *\$str* string. The original *\$str* string is left unchanged.

```
$string = "This is a line of data\n";  
$a = substr($string,5,4); # $a = "is a"
```

uc `uc($string)` – Returns the specified string in uppercase (does not change the original string).

```
$string = "This Is A Line Of Data\n";  
$a = uc($string); # $a = "THIS IS A LINE OF DATA\n"
```

ucfirst `ucfirst($string)` – Returns the specified string with its first character converted to uppercase (does not change the original string).

```
$string = "this is a line of data\n";  
$a = ucfirst($string); # $a = "This is a line of data\n"
```

undef `undef(expression)` – Returns the undefined value (either a null string ("") or a zero ("0")), and initializes the specified *expression* to the undefined value. The expression can be either a scalar variable (either numeric or a string) or an array (in which case all of its elements are set to the undefined value). If *expression* is omitted, returns the undefined value, which can be useful for certain tests.

```
$a = undef(b); # both $a and $b = ""
```

File Access in Perl

One of Perl's strongest points is its ease in reading and writing files, particularly the files that are specified as arguments on the command line. As with everything else, Perl offers many ways to accomplish this task:

- Reading and writing the standard filehandles of STDIN (standard input), STDOUT (standard output), and STDERR (standard error).
- Using the diamond operator (<>), which automatically reads input from the set of files specified on the command line, or if no files were specified, from STDIN.
- Accessing the files specified on the command line using the predefined *@ARGV* array.
- Opening files and using the resulting filehandles to read from and write to those files.

In addition to reading and writing files, Perl's directory operators allow you to read directories to find the filenames in them. Perl's file test operators also allow you to test files for various characteristics.

Using Filehandles

As is typical with Unix-style programs, Perl scripts access files using *filehandles*, which are a special type of variable used for file access. Unlike other variables, filehandles do not have any special prefix such as “\$” or “@”, but otherwise they follow the same rules: they can be composed of any “*word*” characters (letters, numbers, or the underscore), they are case-sensitive, and they can have the same name as another variable of another type.

Thus *file*, *FILE*, *File*, and *FIL_E* are all valid filehandles, even if other variables have these same names. Perl treats all of these filehandles as separate objects and does not confuse them with other variables of different types.

However, it is highly recommended that you use the standard Perl convention for filehandles, which is to use only capital letters. This makes it immediately obvious which variables are filehandles and helps avoid confusion when others read your programs.



Note: In addition to the above file techniques, Perl includes functions to open, read, and write database (DBM) files. See the online Perl reference guide (<http://idDocs/scripts/perl>) for details.

Using STDIN, STDOUT, and STDERR

The Solaris operating system automatically assigns three standard input/output devices to every program, including Perl scripts:

- **STDIN** (standard input) – used for input from the user. The default assignment is to the user’s terminal (keyboard).
- **STDOUT** (standard output) – used for output to the user. The default assignment is to the user’s console (screen).
- **STDERR** (standard error) – used to output error messages to the user. The default assignment is to the to the user’s console (screen).



Note: Your Perl scripts can change the assignments of the standard filehandles by closing them and reopening them so they refer to files or other devices.

Normally when **STDIN** is used, it reads a line of input from the user’s keyboard. Similarly, when **STDOUT** and **STDERR** are used, they output to the user’s screen. These default assignments, though, can be changed on the Solaris command line, using the standard I/O redirection operators (< and >).

For example, to set **STDIN** to be a file, enter a command similar to the following:

```
/usr/thirdParty/perl/bin/perl myscript.pl < input-file
```

If *myscript.pl* is run like this, whenever it reads a line from **STDIN**, it actually reads a line from *input-file*. Similarly, to redirect **STDOUT** to a file, enter a command similar to the following:

```
/usr/thirdParty/perl/bin/perl myscript.pl > output-file
```

Both redirection operators can be combined on the same command line:

```
/usr/thirdParty/perl/bin/perl myscript.pl < input-file > output-file
```

When this command is run, input from STDIN comes from *input-file* and output sent to STDOUT is saved in *output-file*.



Note: STDERR can be redirected as well, but the exact syntax depends on the type of shell you are using. This normally is not done, though, since messages to STDERR are usually intended to inform the user of an immediate problem.

Using STDIN to Read One Line

Perl scripts can get the next line of input from the STDIN device by using the STDIN filehandle. For example, the following command reads one line from STDIN, including the newline (*\n*) character that ends the line:

Example 58 Reading One Line from STDIN

```
$line = <STDIN>;
```

After all input has been read from STDIN, any further attempts to read from it return the undefined value, which can be defined either as the null string ("") or as a zero ("0"). For this reason, STDIN is usually read as shown in Example 59:

Example 59 Reading All Input from STDIN

```
while ($line = <STDIN>) {  
    process-line-here;  
} # end while
```

The *while* loop continues as long as input is available (if STDIN is reading from the console, this is when the user enters a CTRL-D character; if STDIN is redirected to a file, this is when all lines have been read). After the last line has been read from STDIN, the *while* loop exits and the rest of the program is executed.

Because the method shown in Example 59 is so commonly used in Perl, it can be abbreviated to use the default operator ($\$_$), as shown in Example 60:

Example 60 Using the Default Operator with STDIN

```
while (<STDIN>) {  
    process $_ here;  
} # end while
```

The default operator ($\$_$) is automatically used when you use a *while* loop to read from STDIN (or any filehandle). However, when accessing STDIN outside of such loop tests, you cannot use this shortcut but must explicitly name the variable to receive in the input, as shown in Example 58.

The code in Example 60 can be shortened even further by using the diamond operator, which in some circumstances is assigned to STDIN. See *Reading Input From the Diamond Operator (<>)* on page 112 for details.

Using STDIN to Read An Entire File

In the previous section a scalar variable (either *\$line* or *\$_*) was used to read from STDIN one line at a time. If, however, you want to read all of the available lines at once, you can use an array variable, as shown in Example 61:

Example 61 Reading All of STDIN at Once

```
@file = <STDIN>;
```

This one line reads all of the lines available from STDIN and puts each line in one element of the *@file* array. Thus, *\$file[0]* contains the first line of input, *\$file[1]* contains the second line, and so on.

As with any array, you can use the special variable *\$#file* to get the index number of the last element of the array to determine the number of lines read. You could also loop through the array until you reach an element that is set to the *undef* value (the null string "" or a single zero "0").

Using STDOUT and STDERR

The **print** command can be used to send output to the STDOUT and STDERR devices as shown in Example 62:

Example 62 Printing to STDOUT and STDERR

```
print STDOUT "This line goes to STDOUT.\n";  
print STDERR "This line goes to STDERR.\n";
```



Note: STDOUT and STDERR can also be used with the **printf** and **write** commands.

.....

Since `STDOUT` is the default filehandle for the **print** command, you can omit it, as shown in Example 63:

Example 63 Printing to `STDOUT`

```
print "This line goes to STDOUT by default.\n";
```



Note: You can change the default filehandle for the **print**, **printf**, and **write** commands using the **select** command. See the online Perl reference guide (<http://idDocs/scripts/perl>) for details.

Reading Input From the Diamond Operator (<>)

One of Perl's most convenient features is the diamond operator (<>), which returns the next line of input from the files that were specified on the command line when the script was executed. If no files were specified on the command line, the diamond operator reads directly from the STDIN device and becomes the equivalent of <STDIN>.

The diamond operator can be invoked exactly like <STDIN>, with one line of input being returned until all files have been read, at which point the diamond operator returns the *undef* value (the null string "" or a single zero "0"). See Example 64:

Example 64 Using the Diamond Operator

```
$line = <>; # read one line of input from files or STDIN
           # "" is returned if no more input
```

or

```
while (<>) { # while input is available from files/STDIN
    process $_ here ;
}
```

Changing the File List

The diamond operator processes the command line arguments found in the @ARGV array (see *Accessing Files on the Command Line* on page 117), selecting the arguments that appear to be filenames. Using the diamond operator, therefore, is a convenient way to avoid having to parse the command line yourself.

However, there are times you might want to change the list of files that will be read by the diamond operator. You can do this by changing the contents of the `@ARGV` array before the diamond operator is first invoked.

For example, you might have a number of scheduled scripts that periodically gather data from nodes on the network, and each night you want to generate a report on that day's findings. If your scheduled scripts store their data in the `/opt/Panavue/logs` directory, you could specify this directory on the command line when running the reporting script:

```
/opt/Panavue/scripts/perl/doreports.pl /opt/Panavue/logs/*
```

The `doreports.pl` script could then use the diamond operator to read each file in this directory and process it. If, though, you wanted only today's reports to be processed, you could modify the `@ARGV` array so that it includes only files that have been modified in the past 24 hours.

Example 65 shows some sample code that does this:

Example 65 Modifying the Files Accessed by the Diamond Operator

```
$i = $j = 0;           # set indexes to 0
while ($ARGV[$i]) {   # while array has valid data
    if (-M $ARGV[$i] <= 1) {      # has file been modified in
                                    # the past 24 hours?
        $ARGV[$j++] = $ARGV[$i++]; # yes, save it
    }
    else {                  # no, skip it
        $ARGV[$i++] = "";
    }
    $ARGV[$j] = "";        # ensure have final null in array
}
```

```
} # end while  
# @ARGV now contains only files modified within the past 24 hours
```



Note: See *File Access in Perl* on page 105 for information about the **-M** operator and other file test operators.

The code shown in Example 65 could easily be inserted into a subroutine (see *Using Subroutines in Perl*) that is called before the first use of the diamond operator. It assumes, however, that no command line switches were specified (such as `-x` or `-A`); if your scripts also accept switches on the command line, you either have to write code to strip them from the `@ARGV` array or invoke your scripts using the `-s` Perl command line option (see *Command Line Options* on page 10).

If you want to change the `@ARGV` array, do it before the first use of the diamond operator; otherwise, unpredictable behavior can result since the diamond operator has its own internal variables it uses to keep track of which element of the array it is accessing. Depending on the program's current state, the diamond operator might not use your changes.

Finding the Current Filename

Perl keeps the name of the file currently being read in the special variable `$ARGV`. When one file is closed and the next opened and a line is read, Perl automatically updates `$ARGV` with the new filename, including any path information that was specified on the command line. If no files were specified on the command line and the diamond operator is set to `STDIN`, `$ARGV` is set to a single hyphen ("-") character.



Note: Do not confuse \$ARGV with the @ARGV array. Also, \$ARGV is an uninitialized string that matches the null string ("") until the first time the diamond operator is actually invoked and a line of input is read. \$ARGV then remains at its current setting until the diamond operator opens up a new file and reads one line from it. Thus, \$ARGV cannot tell you the name of the next file that is about to be opened, only the name of the file from which the last line of input was read. Use the @ARGV array to find the files that remain to be read.

If you do not care which files are being read and want to treat them as just one giant source of input, it is very convenient to have the diamond operator automatically and transparently move from one file to another. However, if you do want to know what files are being read (typically so you can modify them), you must test for a new filename every time you read a line of input.

Example 66 shows one possible way this can be done:

Example 66 Monitoring \$ARGV for a New Filename

```
$lastargv = $ARGV; # both variables start off as ""
while (<>) {      # a line of input is read here
                 # so must immediately test for new file
    if ($ARGV ne $lastargv)
    {
        $lastargv = $ARGV; # reset with new filename
        do-other-new-file-processing-here
    }
    continue-with-rest-of-while-loop
} # end of while
```

.....

Like the `@ARGV` array, the `$ARGV` variable can be modified by your program to have a new value. Doing so, however, does not change the behavior of the diamond operator, which continues reading the files listed in the `@ARGV` array.

The diamond operator also continues to update the global version of `$ARGV` whenever it opens and reads from a new file, thereby overwriting any change you have made to it. It is recommended, therefore, you modify `$ARGV` only when it is a local variable within a subroutine (see *Defining Local Variables* on page 174).

Accessing Files on the Command Line

Although the diamond operator can be used to automatically read the files on the command line (see *Reading Input From the Diamond Operator* (<>) on page 112), you might want to access these files independently of the order they were specified on the command line. You might also want to test each file first before accessing it, to see that it has the proper date stamp, to verify that it is in the proper directory, to check its size, and so forth.

As mentioned earlier, Perl automatically sets up the @ARGV array with all of the arguments specified on the command line after the script's name. \$ARGV[0] is the first argument, \$ARGV[1] is the second argument, and so forth.



Note: Unlike the similar array used in C programs, the @ARGV array does not contain the program's name; use the special variable \$0 to get the script's name.

The script shown in Example 67 displays the script's name and all of the script's command line arguments:

Example 67 Displaying all Command Line Arguments

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

$i = 0;
print "This script is named $0.\n\n";
while ($ARGV[$i]) {
    print "This is command line arg # $i: $ARGV[$i++].\n";
}
```

.....

The arguments on the command line can be anything that the user typed, including switches (such as *-x* and *-y*), directory names, commands, and filenames. Your script should therefore parse this information and verify it before use. See *File Test Operators* on page 132 for ways you can test a string to see if it is a valid file.

Using Filehandles

As explained at the start of this section, filehandles are a special type of variable that is used for file access. The `STDIN`, `STDOUT`, and `STDERR` variables are really nothing more than filehandles that are automatically opened by the Perl interpreter when it runs your script; if you want to use any other filehandles, your script must specifically create them.

Opening a File for Read-Only Access

Use the **open** command to create a filehandle for a file you want to read:

```
open(FILEHANDLE,"filename");
```

where *FILEHANDLE* is the name of your filehandle and *filename* is the name of the file you want to open (you can either use a literal such as *"myfile.txt"* or a scalar variable such as *\$filename*). The *open* command returns *true* if it was successful in opening the file and the undefined value if not.

Perl allows you to read undefined filehandles, so you should test the results of the **open** command before attempting to use a filehandle:

Example 68 Testing for a Valid Open Operation

```
if (open(FILEHANDLE,"filename")) {
    print "The open successfully created the filehandle.\n";
} else {
    print "Could not open the file.\n";
}
or
unless (open(FILEHANDLE,$filename)) {
```

```
    print "Could not open $filename.\n";
} else {
    print "The open successfully created the filehandle.\n";
}
```

Being unable to open a file usually means your script has encountered a fatal error and should exit. Perl offers a quick way of doing this by using the **die** command, which prints a user-defined message to STDERR and then exits. See Example 69:

Example 69 Using the die Operator

```
open(FILEHANDLE,$filename) ||
    die("$0:Could not open $filename - exiting.\n");
```

If being unable to open a file is not a fatal error but is something the user should be notified about, you can use the **warn** operator. See Example 70:

Example 70 Using the warn Operator

```
open(FILEHANDLE,$filename) ||
    warn("$0:Could not open $filename - continuing...\n");
program continues here;
```

Opening a File for Write Access

To open a new file for writing, add a greater-than sign (>) to the front of the file's name:

Example 71 Opening a New File for Writing

```
open(FILEHANDLE, ">filename");  
or  
open(FILEHANDLE, ">${filename}");
```



Note: You must use double quotation marks when opening a file for writing, both when specifying a literal for the filename (such as ">*filename*") and when you specify a scalar variable (such as ">\${*filename*"}). You can also use the **die** or **warn** operator, as shown in Example 69, when opening files for writing.

The commands shown in Example 71 automatically delete, without any warning, any existing file that has the same name as the file you are attempting to open. If this is not what you want, you should open the file using the append operator (>>). This specifies that the file should be opened for writing, but that all output should be appended to the end of the currently existing file. See Example 72:

Example 72 Opening a New File for Appending

```
open(FILEHANDLE, ">>filename");  
or  
open(FILEHANDLE, ">>${filename}");
```



Note: You can also test for a file's presence before opening it. See *File Test Operators* on page 132.

To write to an opened filehandle, specify the filehandle when using the **print** command:

Example 73 Writing to a Filehandle

```
$logfile = "/opt/Panavue/logs/perl.log";
open(LOGFILE,">>$logfile") ||
    die("$0:Could not open $logfile.\n\n");
print LOGFILE "This line is being written to $logfile\n";
```

Since writing to a file after it has been opened can fail for a number of reasons, such as running out of disk space, consider testing the result of any **print** commands that write to files. For example, Example 74 shows the OR operator (||) being used with the **print** command; if the **print** to the LOGFILE filehandle is unsuccessful, the second **print** statement sends an error message to STDERR.

Example 74 Writing to a Filehandle

```
$logfile = "/opt/Panavue/logs/perl.log";
open(LOGFILE,">>$logfile") ||
    die("$0:Could not open $logfile.\n\n");
print LOGFILE "This line is being written to $logfile.\n"
    || print STDERR "$0:Could not write to $logfile\n";
```

Closing a Filehandle

Use the **close** command to close a filehandle:

```
close(FILEHANDLE);
```

Perl automatically closes any open files when your script terminates, but closing files when they are no longer needed is still a good habit to get into since it ensures that any data that was cached by the operating system is written to the file, which might not happen if a user abnormally terminates the program with a CTRL-C. Closing the file also frees up any memory structures that were used for the filehandle and its buffers.

Directory Operations

Perl includes a set of directory operators to allow you to perform such functions as listing the files that are in a given directory, moving between directories, and removing a file from a directory.

Changing the Working Directory

The **chdir** command changes the working directory to whatever string is specified. The working directory is the default directory for reading and writing files whose filenames do not include a full path, so changing the working directory is advised when you plan to read or write a lot of files in a certain directory. See Example 75:

Example 75 Changing the Working Directory

```
chdir("dirname");  
chdir($dirname);
```

If you are getting the directory name from a user, you must remove the final newline character from the input; otherwise, the **chdir** fails. See Example 76:

Example 76 Changing the Working Directory

```
print "Enter the working directory?\n";  
$dirname = <STDIN>; # get the directory from the user  
chop($dirname); # eliminate the last char (newline)  
chdir($dirname) ||  
    print STDERR "$0:Could not chdir to $dirname\n\n";
```

Listing the Files in a Directory

Any easy way to list the files in the current directory is to use shell-type wildcards within angle brackets (<>). For example, to find all of the normal files (files that do not start with a period) in the current directory and put their filenames into the `@files` array, give the following command:

```
@files = <*>;
```

This command finds all files that match the wildcard pattern and puts each matching filename into one element of the specified array. The files are listed in the same order they are stored in the directory, so `$file[0]` is the first file listed, `$file[1]` is the second file listed, and so forth.

Perl actually spawns an instance of the C-Shell (`/bin/csh`) to expand the list of filenames, so any wildcards that work at the `csh` command line (known as *glob-style* wildcards) also work here. For example, to list all files in the `/etc` directory, use the same wildcard that you would use with the `ls` command (`/etc/*`):

Example 77 Listing All Files in the /etc Directory

```
@etcfiles = </etc/*>;
```



Note: The wildcards used in “globbing” are similar to but not the same to the wildcards used in the Bourne, C, and Korn shells. See the `sh`, `csh`, and `ksh` manpages for details.

You can specify any wildcard pattern as long as it can not be interpreted as a variable or filehandle. For example, the following matches would fail because Perl interprets the string inside the brackets as referring to a Perl variable of one type or another:

Example 78 Invalid Globbing Patterns for Perl

```
@a = <$files>;# Perl interprets $files as scalar variable
@a = <@files>;# Perl interprets @files as array
@a = <files>; # Perl interprets files as filehandle
@a = <FILES>; # Perl interprets FILES as filehandle
```

Because this approach requires spawning another process (the */bin/csh* shell), it is not the most efficient approach, especially if you expect to deal with a large number of files. It also returns directory names in addition to regular files, so you also have to test each name before using it to see whether it refers to a file or a directory entry.

Because of these limitations, reading the directory directly, as described in the next section, is usually the preferred method of getting a file list from a directory.

Reading a Directory Entry Directly

To read a directory entry, use the **opendir** command, which works like the **open** command but on directories:

Example 79 Opening a Directory Entry

```
$etcdir = "/etc/";
opendir(ETCDIR,$etcdir) ||
    die("$0:Could not open the $etcdir directory.\n");
```

As shown in Example 79, the **opendir** command uses a filehandle, just as the **open** command does, but **opendir** creates a directory filehandle, which can be meaningfully read only by the **readdir** command, which has the following format:

Example 80 Reading a Filename Using `readdir`

```
$filename = readdir(DIRHANDLE);
```

Only the actual filename is returned by **`readdir`**; the path information is not included as part of the string (for example, `$filename` is set to `"passwd"` and not `"/etc/passwd"` when parsing the `/etc/` directory).

As with other forms of input, **`readdir`** returns the undefined value (`""` or `0`) when it has returned all of the names in the files in the directory. Typically, therefore, a *while* loop is used to read the directory:

Example 81 Reading All Files in a Directory

```
#!/usr/thirdParty/perl/bin/perl -w
$etcdire = "/etc/";
opendir(ETCDIR,$etcdire) ||
    die("$0:Could not open the $etcdire directory.\n");
while($filename = readdir(ETCDIR)) {
    print "The $etcdire directory contains: $filename\n";
} # end while
```

The **`readdir`** command returns all of the filenames in a directory, including those starting with a period. The filenames are returned in the same order that they exist in the directory entry, which is not necessarily a sorted list. To produce a sorted list, use the **`sort`** command while reading the list of filenames into an array:

Example 82 Reading and Sorting All Files in a Directory

```
#!/usr/thirdParty/perl/bin/perl -w
$etcdire = "/etc/";
opendir(ETCDIR,$etcdire) ||
    die("$0:Could not open the $etcdire directory.\n");
@files = sort(readdir(ETCDIR));
foreach $filename (@files) {
    print "The $etcdire directory contains: $filename\n";
}
```



Note: Use the **closedir** command to close a directory filehandle after use.

Deleting Files

The **unlink** command deletes a file and it operates on filenames, not filehandles, so you do not need to open a file before using this function. The **unlink** command can delete either a single file or a list of files, and it has the following formats:

Example 83 unlink Command Formats

```
unlink("filename"); # file is specified as a literal
or
unlink($filename); # file is specified as a scalar var
or
unlink(@files);     # list of files specified as an array
or
unlink(<*>);        # list of files specified as glob
```

The **unlink** command attempts to delete all of the specified files and then returns the number of files that have been successfully deleted. If you want to verify that all files

.....

have been deleted, you must either count the number of files in advance or test for the supposedly deleted files' presence (see *File Test Operators* on page 132).



Note: The **unlink** command deletes files without warning, so use it very carefully, particularly when specifying a list of files or when using wildcards.

Renaming Files

The **rename** command renames a file and it operates on filenames, not filehandles, so you do not need to open a file before using this function. Example 84 shows the format of the **rename** command:

Example 84 Renaming a File

```
rename("oldfile","newfile"); # using literals
rename($oldfile,$newfile);   # using scalar variables
```

If the specified filenames do not have complete pathnames, they are assumed to be in the current working directory. If **rename** does successfully rename the given file, it returns true; otherwise, it returns the undefined value. See Example 85:

Example 85 Example of Renaming a File

```
if (rename($oldfile,$newfile)) {
    print "Successfully renamed $oldfile to $newfile\n";
} else {
    print "Could not rename $oldfile to $newfile\n";
} # end if
```



Note: Perl scripts can rename or delete files only if the scripts are run by a user who has the permissions needed to change those files at the command line.

Creating and Removing Directories

Perl has its own versions of the Unix **mkdir** and **rmdir** commands to create and remove directories. The **mkdir** function takes two arguments, the name of the directory and its set of permissions:

```
mkdir("dirname",chmod-value);  
or  
mkdir($dirname,$chmod-value);
```

The directory name can be either a literal string or a scalar variable that specifies either a full or partial pathname. The permissions must be in the octal format used by the Unix **chmod** program (see the **chmod** manpage for details).

The **mkdir** function returns true if successful and the undefined value if not. If the directory could not be created, the special variable `!` is also set with the appropriate error code.

Example 86 attempts to create a directory named “*tmp*” in the current working directory, giving all users complete read, write, and execute permissions to the directory:

Example 86 Creating a Directory

```
if ( mkdir("tmp",0777) ) {  
    print "Successfully created the tmp directory\n";  
} else {  
    print "$0:Unable to create directory because of:\n";  
}
```

```
    print "\t$!, error number ", $!+0, "\n";  
}
```

The **rmdir** command removes a current directory, assuming the directory has no files in it and the script's user ID has the proper permissions. The **rmdir** command removes one directory at a time, and like **mkdir**, it sets the `!` variable if it fails. See Example 87:

Example 87 Removing a Directory

```
if ( rmdir("tmp") ) {  
    print "Successfully removed the tmp directory\n";  
} else {  
    print "$0:Unable to remove directory because of:\n";  
    print "\t$!, error number ", $!+0, "\n";  
}
```

File Test Operators

Because you often need to know something about a file before using it, Perl offers a number of file test operators that can tell you almost anything you need to know about a file. Table 12 lists the available operators according to category; unless otherwise specified, each operator returns true (non-zero) if the test succeeds. Where appropriate, the file tests also work with directories as well as files.

Table 12 File Test Operators

Operator	Description
Information about the File and Its Contents	
-B	File is a binary file (opposite of -T)
-b	File is a block special file
-c	File is a character special file
-d	File is a directory, so it must be opened with the <i>opendir</i> function and read using the <i>readdir</i> function.
-e	File exists – it could be of any size, though, including zero bytes, so use either -s or -z, below, to test it further.
-f	File is a plain file
-l	File is a symbolic link
-p	File is a named pipe (FIFO)

Table 12 File Test Operators

Operator	Description
-S	File is a socket
-s	Returns the size of the file (a non-zero size evaluates to true, a zero size evaluates to false). Also see -z below.
-T	File is a text file
-t	Filehandle is opened to a tty
-z	File has zero size
Information about the File or Directory's Timestamp	
-A	Returns the time that the file or directory was last accessed in days (including fraction for the part that is less than 24 hours).
-C	Returns the time that the file or directory was created in days (including fraction for the part that is less than 24 hours).
-M	Returns the time since the file or directory was last modified in days (including fraction for the part that is less than 24 hours).
Information about the File or Directory's Permissions	
-O	File is owned by real user ID
-R	File is readable by real user ID/group ID
-W	File is writable by real user ID/group ID

Table 12 File Test Operators

Operator	Description
-X	File is executable by real user ID/group ID
-g	File has setgroup ID bit set
-k	File has sticky bit set
-o	File is owned by effective user ID
-r	File is readable by effective user ID/group ID
-u	File has setuser ID bit set
-w	File is writable by effective user ID/group ID
-x	File is executable by effective user ID/group ID

The most common tests that are done are to see if a file or directory exists (*-e*), if it is readable or writable by the script (*-r* and *-w*), its size (*-s* and *-z*), and its modification age (*-M*). The tests can be used within any control structure or loop, but typically they are used within an “*if*” clause; the major exception to this are the tests that return specific information, such as the size of the file or its age, which can be used anywhere any other expression can be used.

Example 88 demonstrates the use of these file tests by getting a list of files and then testing each one in turn. Note, however, that the script’s first test is to see if the file exists, which is recommended because some user or process could always delete a file between the time it is first found and the time it is used.

Example 88 Testing Files

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

@files = <*>; # get list of files in current working dir
foreach $filename (@files) {
    if (-e $filename) {
        print "$filename does exist\n";
        if (-z $filename) {
            print "\tIt is zero bytes in size\n";
        } else {
            print "\tIts size is ",(-s $filename), "bytes\n";
        } # end else
        if (-r $filename) {
            print "\tIt is readable by this script\n";
        } # end if
        if (-w $filename) {
            print "\tIt is writable by this script\n";
        } # end if
        print "\tIt was modified ",(-M $filename), "days ago\n";
    } # end if

    # should fall through here only if someone deleted the
    # file between the time the script did the glob command and
    # the time it tested for the file's presence
    else {
        print "The file $filename no longer exists in the ";
        print "current working directory.\n";
    } # end else
} # end foreach
```

Using Regular Expressions in Perl

One of the reasons Perl has become so popular is the power and flexibility of its regular expressions (commonly referred to as *regex*). Programmers familiar with the type of regular expressions used in Unix commands such as *awk*, *grep*, *sed*, and *vi* find Perl's regular expressions to be similar, but Perl includes a number of features that are not available in these other programs.

At their simplest, regular expressions embody the same principle as the “Find and Replace” features of the typical word processor. The user of a word processor enters one or more words to find, and the word processor searches the file to find every place this word appears. When the desired word is found, the word processor can either replace it with another word of the user's choosing or continue searching for the next occurrence.

Regular expressions in Perl are used in much the same manner to do the following:

- To find a particular pattern of characters
- To delete a particular pattern of characters
- To replace one pattern of characters with another

Unlike most word processors, though, regular expressions can use very complex patterns that check for many different words, phrases, or sets of characters at the

same time. For example, if you have a report that lists all of the alarms in a Promina 800 Series network, you might want to print out only the trunk and link alarms.

This would be difficult to do with a typical word processor because you would have to first search for “link alarms” and then do another search for “trunk alarms.” Then you would have to separate those lines out and print them.

In Perl, though, you can use one statement to search for both patterns at once. You could also narrow the search to a specific set of nodes and trunk cards, or even do a search that excludes all trunk and link alarms, printing out the rest. Many other options exist when using Perl’s regular expressions.

Defining Regular Expressions

In Perl, a regular expression is defined in the same way it is done with the *sed* or *vi* utilities: by enclosing it within two forward slashes, as shown in Example 89:

Example 89 Typical Regular Expressions

```
/trunk alarms/  
/Node 152/  
/Robert/  
/Monday/
```

Similarly, a substitution pattern is defined as it is in *sed* or *vi* – the letter “s” is followed by three forward slashes that separate the pattern to be matched from the pattern that is to replace it. See Example 90:

Example 90 Typical Substitution Patterns

```
s/trunk alarms/link alarms/  
s/Robert/Bob/  
s/Monday/Tuesday/  
s/not needed//
```

In the examples given in Example 90, the text “trunk alarms” is replaced by “link alarms,” the name “Robert” is replaced by “Bob,” and “Monday” is replaced by “Tuesday.” The final example deletes the two words “not needed”.



Note: The forward slashes are only the default delimiters for regular expressions and substitution patterns. You can specify other nonalphanumeric character for this purpose by preceding the first delimiter with “**m**”, as in “**m#Monday#Tuesday#**”. See the online Perl reference guide (<http://idDocs/scripts/perl>) for further details.

Rules in Using Regular Expressions

When using regular expressions, a few basic rules apply in all cases:

- When matching a regular expression, the match is done against the default variable for standard input (`$_`) unless another variable is specified by using the `regexp` operator (`=~`). This greatly simplifies using regular expressions when the input is either STDIN or the diamond operator, which can use the default variable.

For example, a Perl script that reads a set of files and looks for a particular pattern could have a basic structure no more complex than that shown in Example 91:

Example 91 Template for Using Regular Expressions for Matching

```
#!/usr/thirdParty/perl/bin/perl -w
while (<>)                   # read each line from STDIN/files
  if (/pattern-to-match/) {
    stuff-to-do-when-match-is found
  }                         # end if
  else {
    do-this-other-stuff
  }                         # end else
}                            # end while
```

- To match against a specific variable, use the following syntax:
`$variable =~ /pattern-to-match/;`

To rewrite Example 91 so that the match is made against the variable *\$var*, rewrite the “*if*” clause as follows:

Example 92 Performing a Match Against a Variable

```
if ($var =~ /pattern-to-match/) {  
    stuff-to-do-when-match-is-found  
}  
# end if  
else {  
    do-this-other-stuff  
}  
# end else
```

Alternatively, you could use the negation regexp operator (*!~*) to reverse the logic of the “*if*” clause:

Example 93 Performing a Negative Match Against a Variable

```
if ($var !~ /pattern-to-match/) {  
    stuff-to-do-when-match-is-NOT-found  
}  
# end if  
else {  
    do-this-other-stuff  
}  
# end else
```

- Substitutions are handled the same way as simple pattern matching: the substitution is done against the default variable for standard input (*\$_*) unless another variable is specified by using the *=~* operator.

Example 94 Performing Substitutions

```
$_ = "This line is from the default variable";  
$var = "This line is from the variable";  
s/ is / was /; # $_ = "This line was from the default variable"  
$var =~ s/
```

- You can match against more than one pattern by using the OR operator (`|`) to separate the different patterns. For example, the following regular expression matches either “up” or “down”:

```
/up|down/
```

- Whenever a match is successfully done (but not a substitution), Perl automatically fills in the following read-only variables:

`$&` – contains the text that was first matched

`$'` – contains the text that appeared on the line before the first match

`$'` – contains the text that appeared on the line after the first match

These variables are not set until a successful match is made, but when set they remain so until the next successful match is done. These variables are also read-only, so your program cannot change their values. Therefore if you want to save the information in these variables, you should copy their values into your own variables.

- By default, a match or substitution is case-sensitive; `/abc/` matches only the lowercase letters “abc” not “ABC”. To change this append the ignore case operator (“`i`”) at the end of the regular expression: `/abc/i` or `s/abc/def/i`.

- Similarly, a match or substitution is done only on the first successful match on a line. To have a match or substitution occur everywhere possible on a line, append the global operator (“g”) to the regular expression: **/abc/g** or **/abc/def/g**.

Simple Matching and Replacing

The simplest possible match is one where the text being searched matches the desired pattern exactly. Such searches can be done with the equality operators for strings (*eq* and *ne*), as shown in Example 95. This script reads lines from a file specified on the command line (or from STDIN) and prints out all lines that are not “*Robert*”:

Example 95 A Simple Match

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

while (<>) {          # read from file/STDIN
    chop;            # chop off the newline char
    $name = $_;      # assign the line of input to $name
    if ($name ne "Robert") { # is the name Robert?
        print "$name\n";    # if not, then print it
    } # end of if
} # end of while
```

Text can also be replaced in this manner, as shown in the script in Example 96, which prints all lines unchanged except for “*Robert*,” which is printed as “*Bob*”:

Example 96 A Simple Substitution

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

while (<>) {          # read from file/STDIN
    chop;            # chop off the newline char
    $name = $_;      # assign the line of input to $name
    if ($name ne "Robert") { # is the name Robert?
```

```

    print "$name\n";           # if not, then print it
  } # end if
else {
    $name = "Bob";           # otherwise, if the name is "Robert"
    print "$name\n";         # replace "Robert" with "Bob"
                              # and print the new name
  } # end of if/else
} # end of while

```

The matching and substitution in these examples is extremely simple. The only time a match can be made is when the input line contains nothing but the name “Robert”; lines containing text such as “robert” or “Robert Johnson” are not matched.

Also, this sort of matching requires an “*if*” or “*else*” (or “*elsif*”) construction for each possible match. This is not too much of an inconvenience in Example 95 and Example 96 since they have only two possible conditions: either the text matches “Robert” or it does not. However, this approach rapidly becomes unwieldy if you want to match more than one pattern.

For these reasons, regular expressions are typically used as shown in Example 97:

Example 97 A Simple Use of Regular Expressions

```

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

while (<>) {
    chop;                # chop off the newline char
    $name = $_;          # assign the line of input to $name
    unless ($name =~ /Robert/) { # is the name Robert?
        print "$name\n";    # if not, then print it
    } # end of unless
} # end of while

```

The program shown in Example 97 is almost identical to that shown in Example 95, except that the “*if*” clause has been replaced by an “*unless*” clause, and its test has become a regular expression that matches “Robert” wherever it appears on the input line. For example, this program would not print any of the following lines since they all contain the text “Robert” at some point:

```
Robert
Roberta
Robert Johnson
My name is Robert
abcdefgRoberthijklmnopqrstuvwxyz
```

This program can be enhanced by the addition of the letter “*i*” after the regular expression, which instructs Perl to ignore the case of the letters. Because of this, the program in Example 98 ignores all lines containing “Robert”, “robert”, or any other permutation of upper and lowercase letters in that name. Example 98 also has been simplified to eliminate the redundant use of the *\$name* variable by using the default operator of *\$_*.

Example 98 Specifying Case-Insensitive Regular Expressions

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

while (<>) {
    unless (/Robert/i) {
        print;
    } # end of unless
} # end of while

# read from file/STDIN
# does the line contain Robert,
# robert, or any variation?
# if not, then print it
```

Regular expressions also greatly simplify substitutions. For example, Example 99 is a rewrite of the script shown in Example 96 (page 144). However, whereas the previous example needed “*if*” and “*else*” clauses to determine whether to change the line before printing, the rewritten script does not because the substitution process is automatic – a substitution is made only if “Robert” (or a variation such as “robert”) appears on the line:

Example 99 A Simple Substitution Using Regular Expressions

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

while (<>) {           # read from file/STDIN
    s/Robert/Bob/i;   # if Robert exists in either upper
                    # or lowercase letters, change to Bob
    print;           # print the line
}                   # end of while
```

The script in Example 99 prints out all lines from the input file (or STDIN), substituting “Bob” for “Robert” whenever it appears. However, this substitution is pretty simplistic and has two potential problems.

- The substitution is done for any form of the name “Robert.” Thus, names such as “Roberta,” “Roberto,” and “Robertson” become “Boba,” “Bobo,” and “Bobson” respectively. This is probably not what is intended.
- The substitution is done for only the first occurrence of “Robert” on the input line. A line reading “Robert, my name is Robert” becomes “Bob, my name is

Robert.” The second appearance of “Robert” is ignored by the substitution expression because unless otherwise specified, regular expressions find only the first match on a line.

The first problem can be solved by limiting a match only to “Robert” when it appears as a complete word, not when it is part of a larger word. This is done by bracketing the text with the word boundary marker “\b”.

The regular expression “\bRobert\b” specifies that a match can be made only when the text “Robert” is both preceded and followed by whitespace or other word terminators such as punctuation. As a result, words such as “Roberta” and “Robertson” are not matched.

The second problem can be solved by adding the global operator (“g”) at the end of the regular expression. This indicates that the substitution should be done at all places on the line where a match is made. Example 100 shows the modified script:

Example 100 Enhancing a Simple Substitution

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

while (<>) {
    s/\bRobert\b/Bob/ig; # read from file/STDIN
                        # if Robert exists in either upper
                        # or lowercase letters, but only as
                        # a complete word, change to Bob
                        # everywhere on the line
    print;              # print the line
} # end of while
```

The script shown in Example 100 thus can make more intelligent substitutions, such as the following:

Example 101 Typical Substitutions

My name is Robert.	becomes	My name is Bob.
robert robertson	becomes	Bob robertson
Robert, Robert	becomes	Bob, Bob
My name is Roberta.	remains	My name is Roberta.

The word boundary marker (“\b”) is one of four anchoring patterns that can be used to limit a match to a specific situation. Table 13 lists these patterns and illustrates their use:

Table 13 Anchoring Patterns for Regular Expressions

Anchor Pattern	Description	Example
\b	matches a word boundary	/\bmail\b/ matches “mail” but not “email” nor “mailman”
\B	matches anything but a word boundary	/\Bmail/ matches “emailer” and “re-mail” but not “mail” nor “mailman” /mail\B/ matches “mailman” and “emailer” but not “mail” nor “email”
^	matches the beginning of a line	/^mail/ matches “mail” when it appears in the line “mail is easy to use” but not when it appears in the line “I find mail easy to use.”
\$	matches the end of a line	/mail\$/ matches “mail” when it appears in the line “I like mail” but not in the lines “I like mail.” (because the line ends with a period) nor the line “I like mail sometimes.”

The anchoring patterns shown in Table 13 are a subset of a large number of special characters that can be used to limit a match using regular expressions. Most of these special characters are used to match nonprintable characters (such as the newline character), but some of them refer to a group of characters (such as whitespace characters or numeric digits). See Table 14:

Table 14 Special Characters for Regular Expressions (1 of 2)

Anchor Pattern	Description	Example
<code>\d</code>	matches any digit (0-9)	<code>/N\d\d/</code> matches “N20” or “N91” but not “NOPr”
<code>\D</code>	matches any non-digit (any character that is not 0-9)	<code>/N\d\d/</code> matches “Nor” or “Not” but not “N00” through “N99”
<code>\f</code>	matches a formfeed character (ASCII 12 or CTRL-L)	<code>/end of page\f/</code> matches the words “end of page” only if they are immediately followed by a formfeed character
<code>\n</code>	matches a newline character (ASCII 10 or CTRL-J)	<code>/end of line\n/</code> matches the words “end of line” only if they are immediately followed by a linefeed character
<code>\r</code>	matches a carriage return character (ASCII 13 or CTRL-M)	<code>/end of line\r/</code> matches the words “end of line” only if they are immediately followed by a carriage return character (this match is not usually used in Unix systems, where only the newline character is defined as only linefeed, but it could be useful for files from DOS or Macintosh computers that use the carriage return)

Table 14 Special Characters for Regular Expressions (2 of 2)

Anchor Pattern	Description	Example
<code>\s</code>	matches any whitespace character (space, tab, newline, carriage return, or formfeed)	<code>/Node\sCard/</code> matches the words “Node” and “Card” only if they are separated by one space, tab, newline, carriage return, or formfeed character
<code>\S</code>	matches any non-whitespace character	<code>/NodeSCard/</code> matches the words “Node” and “Card” only if they are not separated by a whitespace character (for example: “Node-Card” or “NodesCard”)
<code>\t</code>	matches a tab character (ASCII 9 or CTRL-I)	<code>/Node\tCard/</code> matches the words “Node” and “Card” only if they are separated by one tab character.
<code>\w</code>	matches any word character (0-9, a-z, A-Z, or the underscore)	<code>/N\w\w\w/</code> matches “N120”, “N_21”, “Node”, and “Nick”, but not “N.12” or “N1-2” (since the period and hyphen characters are not “word” characters).
<code>\W</code>	matches any non-word character	<code>/N\W/</code> matches “N.”, “N-”, or the letter “N” followed by whitespace, but not “No” or “N1”

Finally, to match a character with a specific ASCII value, specify that value in one of the three ways shown in Table 15:

Table 15 Specifying an ASCII Value

Anchor Pattern	Description	Example
<code>\OOO</code>	<i>OOO</i> specifies an octal value.	<code>\115/</code> matches ASCII 13 (CTRL-M) <code>\141/</code> matches ASCII 97 (the letter "a")
<code>\xHH</code>	<i>HH</i> specifies a hexadecimal value	<code>\x0D/</code> matches ASCII 13 (CTRL-M) <code>\x61/</code> matches ASCII 97 (the letter "a")
<code>\cX</code>	<i>X</i> specifies a control character (A-Z)	<code>\cM/</code> matches ASCII 13 (CTRL-M) <code>\cZ/</code> matches ASCII 26 (CTRL-Z)

The special characters given in Table 13, Table 14, Table 15 can be combined as desired to find very specific patterns. For example, if you log in to a node using the Operator Interface and query the event log, the events are displayed in the following format:

Example 102 Typical Event Log

```

*** Event Record from Event Log on Node 20(NODE20) ***
Event Type = TRUNK (2), Subtype = 5
Orig Node = 20(NODE20), Orig TaskId = TRUNK (16.6)
Occurred at 16:55:52 TODAY, Sequence Nbr = 0
Alarm Level = MINOR, Network Event Log = NO
>>> Card N20C6 experienced a SUPER FRAME LOSS. TxStatus was 10, RxStatus
was 70.

```

To print out only the second line that describes the “Event Type” and the sixth line that contains a description of the event, you could use the following script:

Example 103 Searching for Multiple Matches

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

while (<>) {
if (    /^\\s\\s\\s\\sEvent Type/i # read from file/STDIN
      ||    /^>>>\\s\\w/          # if found "Event Type"
      ||    /\\.$/              # or if found ">>> "
      )    # or line ending with a period
{ print; } # print the line
}         # end of while
```

The script in Example 103 uses the logical OR operator (||) to specify three regular expressions to be matched; the input line is printed if at least one match is made.

- The first regular expression (`/^\\s\\s\\s\\sEvent Type/i`) searches for a line that begins with four whitespace characters followed by the text “Event Type” in either upper or lowercase.
- The second regular expression (`/^>>>\\s\\w/`) searches for a line that begins with three right angle brackets followed by a whitespace character and a word character (either a digit or a letter). The ignore operator (“i”) is not needed in this case because the word special character (`\\w`) matches both upper and lowercase letters automatically.
- The third regular expression (`/\\.$/`) searches for a line that ends with a period, assuming that such lines are a continuation of the description that began in the

.....

sixth line, as shown in Example 102. This, though, might not be a safe assumption to make, and you might end up printing unwanted lines.

In fact, although the script in Example 103 does perform as intended in the great majority of cases, it makes a number of assumptions (such as that there will always be four whitespace characters before an “Event Type” line) that might not hold true in all cases. This script could be significantly improved by using wildcard specifiers, which are described in the next section.

Using Wildcards in Regular Expressions

Much of the power and flexibility of regular expressions comes from the use of wildcards, which allow a single character to represent an almost infinite combination of other characters. The wildcards used in Perl are almost identical to those used in other Unix programs such as *sed* and *vi*; see Table 16:

Table 16 Perl Wildcards (1 of 2)

Wildcard Character ¹	Description
.	The period matches any single character except for the newline (<code>\n</code>) character. For example: <code>/./</code> matches the first character on a line; <code>/.g</code> matches all characters on a line except for the newline character.
* ²	The asterisk matches zero or more consecutive instances of the immediately previous character (including special characters). For example: <code>/1*/</code> matches zero or more consecutive instances of the number “1” such as “”, “1”, or “11”; <code>/s*/</code> matches zero or more whitespace characters.
+	The plus sign matches one or more consecutive instances of the immediately previous character (including special characters). For example: <code>/1+/</code> matches one or more consecutive instances of the number 1; <code>/w+/</code> matches one or more “word” characters, which typically means it matches one word at a time.
?	The question mark matches zero or one instances of the immediately previous character (including special characters). For example: <code>/1?/</code> matches zero or one instances of the number 1; <code>/s?/</code> matches zero or one instances of whitespace characters.

Table 16 Perl Wildcards (2 of 2)

Wildcard Character ¹	Description
{x,y}	<p>The square brackets match a specific number of consecutive instances of the immediately previous character, where <i>x</i> and <i>y</i> define the range of allowable matches. For example: <code>/1{5,10}/</code> matches from five through ten consecutive instances of the number 1.</p> <p>Both the comma and second number of the range are optional. If the comma is present but not the second number, the second number is assumed to be infinity. Thus, <code>\s{5,}/</code> matches five or more whitespace characters.</p> <p>If both the command and second number are missing, the first number specifies the exact number of characters that must be found. Thus, <code>\s{5}</code> matches exactly five whitespace characters. If six whitespace characters are present on a line, only the first five are matched.</p>
[char list]	<p>The square brackets specify a match with any of the enclosed characters. The characters can be listed singly (<code>[abcdefg]</code>) or as a range (<code>[a-g]</code>). For example: <code>[0-9]</code> is equivalent to the <code>\d</code> special character and matches any single digit; <code>[a-zA-z0-9_]</code> is equivalent to the <code>\w</code> special character and matches any “word” character.</p>
[^char list]	<p>When followed by a carat (^), the square brackets specify a match with anything except the enclosed characters. The characters can be listed singly (<code>[^abcdefg]</code>) or as a range (<code>[^a-g]</code>). For example: <code>[^0-9]</code> is equivalent to the <code>\D</code> special character and matches any single character <i>except</i> a digit; <code>[^a-zA-z0-9_]</code> is equivalent to the <code>\W</code> special character and matches any single character <i>except</i> a “word” character.</p>

1. To match any of the wildcard characters themselves, put a backslash before the character. For example: `\./` matches the period (.) character, `*` matches one asterisk character, `\?` matches a single question mark, and so forth.
2. Be careful when using the asterisk (*) and question mark (?) since they are “optional” matches (they can match zero instances of a character or pattern). You might think the expression `\d*\.\?d*` matches any set of decimal numbers (such as “1.0” or “0.899”) but in reality it always matches any string because all of the search patterns are optional. When using * and ? in regular expressions, be sure you include at least one non-optional pattern to ensure a valid match is made.



Note: Do not confuse Perl's wildcards with those used on the Unix command line. When used on the command line, the asterisk (*) refers to zero or more of *any* characters; also, the question mark (?), not the period (.), is used on the command line to represent any single character.

Wildcards are “Greedy”

Perl's wildcards are governed by two overriding rules:

1. Given a choice, earlier matches take precedence over matches that start later in a line.
2. If more than one match is possible with a given starting point, Perl returns the longest possible match.

This behavior is called “greedy” and is why `./` matches everything on a line except the newline character. This “greedy” behavior of wildcards, though, can be both useful and troublesome.

If, for example, you wanted to find the last word in a sentence, you might consider using `/\b.*\./` because you assume it finds only the last word boundary before a period. In reality, this regular expression returns everything between the first whitespace on the line and the last period.

Example 104 shows a Perl program that demonstrates this aspect of wildcards:

Example 104 Demonstrating a “Greedy” Match

```
#!/usr/thirdParty/perl/bin/perl -w
while (<>) {
    /\b.*\./;
    # read from file/STDIN
    # match last word in sentence
```

```
print "$&\n";           # print what was matched (using
                        # the read-only variable Perl
                        # defines for this purpose)
}                       # end while
```

If you were to give the following line to this script:

Perl's wildcards exhibit greedy behavior.

the script simply reprints the entire sentence:

Perl's wildcards exhibit greedy behavior.

To find only the last word in this sentence (the word “behavior”), you must modify the script so it uses a more specific regular expression, in this case the “\w+” expression, which matches only “word” characters, not spaces nor punctuation. See Example 105:

Example 105 Demonstrating a “Less Greedy” Match

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

while (<>) {           # read from file/STDIN
  /\b\w+\./;          # match last word in sentence
  print "$&\n";       # print what was matched (using
                      # the read-only variable Perl
                      # defines for this purpose)
}                     # end while
```

When using wildcards, therefore, choose the expression that is as specific as possible (such as using “\w+” instead of “.*”). This makes it less likely that the greedy behavior of wildcards will end up matching more than you intended.



Note: The question mark (?), asterisk (*), and plus sign (+) operators can be made “ungreedy” by appending another question mark after them (for example: ??, *?, +?). This greatly changes their behavior and should be used only by Perl experts.

Using WildCards for Matching

The variety of wildcards that is present in Perl allows you to match a wide variety of very specific patterns that would be very difficult to match otherwise. For example, to search an event log to find all lines that contain a node or card number, without using wildcards, you would have to use all of the following regular expressions:

Example 106 Matching Node and Card Numbers Without Wildcards

```

/N\d/           # to find N0 through N9
/N\d\d/        # to find N10 through N99
/N\d\d\d/      # to find N100 through N250
/N\dC\d/       # to find N0C0 through N9C9
/N\d\dC\d/     # to find N10C0 through N99C9
/N\d\d\dC\d/   # to find N100C0 through N250C9
/N\dC\d\d/     # to find N0C10 through N9C99
/N\d\dC\d\d/   # to find N10C10 through N99C99
/N\d\d\dC\d\d/ # to find N100C10 through N250C99
/N\dC\d\d\d/   # to find N0C100 through N9C127
/N\d\dC\d\d\d/ # to find N10C100 through N99C127
/N\d\d\dC\d\d\d/ # to find N100C100 through N250C127

```

.....

If you also wanted to find port and bundle numbers, you would have to add another 48 possible patterns to ensure you found all possibilities. Wildcards, however, simplify all of this to one single regular expression, as shown in Example 107:

Example 107 Matching Node and Card Numbers With Wildcards

```
#!/usr/thirdParty/perl/bin/perl -w
while (<>) { # read from file/STDIN
    if (/N\d+C?\d*/) # if node/card #
    { print; } # print the line
} # end while
```

The single line shown in Example 107 matches anything that starts with one “N” and is followed by one or more digits (such as “N10” or “N204”). It also matches an “N” followed by one or more digits and a “C” followed by zero or more digits (such as “N10C23” or “N204C1”).

Modifying this script to also search for port or bundle numbers is trivial, as shown in Example 108:

Example 108 Matching Node, Card, Port, and Bundle Numbers With Wildcards

```
#!/usr/thirdParty/perl/bin/perl -w
while (<>) { # read from file/STDIN
    if (/N\d+C?\d*[BP]?\d*/) # if node/card/port/bundle #
    { print; } # print the line
} # end while
```

Actually, though, the regular expression in Example 108 is a bit redundant since testing for the card, port, and bundle numbers is unnecessary – testing for the node number also finds any card, port, and bundle numbers since they always include a node number as well.

However, this script can be easily modified to print only those lines that contain a port number or a bundle number. Example 109 shows the same script, except that now a line must include a card number and either a port or bundle number to be printed:

Example 109 Matching Port and Bundle Numbers With Wildcards

```
#!/usr/thirdParty/perl/bin/perl
while (<>) {
    if (/N\d+C\d+[BP]\d+/) # read from file/STDIN
        { print; } # if port or bundle #
    } # print the line
    } # end while
```



Note: When devising complex regular expressions, it is recommended you use the approach shown in Example 107 through Example 109: start off by writing and testing a regular expression that matches the common features of a desired pattern (in this case, the node and card numbers). Then make the regular expression more specific in a step-by-step manner, testing each step to make sure the proper patterns are being matched.

Using Wildcards for Substitutions

Wildcards can also be used for substitutions, as shown in Example 110, which substitutes the words “PORT/BUNDLE” for every port or bundle number found on a line:

Example 110 Substituting Port and Bundle Numbers

```
#!/usr/thirdParty/perl/bin/perl
while (<>) { # read from file/STDIN
    s/N\d+C\d+[BP]\d+/PORT\BUNDLE/g; # do substitution on
                                     # all port/bundle #s
    print; # print the line
} # end while
```

This script prints lines as shown in Example 111:

Example 111 Examples of Substitutions for Port and Bundle Numbers

Node N75	remains	Node N75
Card N10C12	remains	Card N10C12
Port N10C12P13	becomes	Port PORT/BUNDLE
Bundle N23C7B1	becomes	Bundle PORT/BUNDLE

Although this sort of blanket pattern substitution might be useful on occasion, wildcard substitutions are more often done using Perl's ability to separate a regular expression into subpatterns using parentheses. When a set of parentheses surrounds a portion of a regular expression, that part of the expression can be referenced later by using a special set of read-only substitution variables: \$1 refers to the first parenthetical expression (as read from left to right), \$2 refers to the second such expression, and so forth.



Note: Perl also allows you to refer to the read-only substitution variables as \1 through \9, but this usage is discouraged since it does not work in all situations.

Example 112 shows the port/bundle substitution script rewritten to include parenthetical expressions so that more intelligent substitutions can be done:

Example 112 Substitution Using Read-Only Variables

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

%pbwords = ("B", "Bundle", "P", "Port",
            "b", "Bundle", "p", "Port");
            # define an associate array to translate
            # "P" and "B" to the corresponding words
while (<>) {
    # read from file/STDIN
    s/N(\d+)C(\d+)([BP])(\d+)/Node $1, Card $2, $pbwords{$3} $4/ig;
    # do substitution using
    # memorized values
    print;
    # print the line
} # end while
```

The script in Example 112 still searches for port and bundle numbers, but now when one is found, it is converted from the NCP or NCB format into one that spells out the words “node,” “card,” “port,” and “bundle.” To achieve this, the new script has undergone three major changes from the previous one:

- The script now starts out by defining an associative array (*%pbwords*) which matches the letters “b” and “B” to the word “Bundle” and the letters “p” and “P” to the word “Port”. Since the regular expression matches either port or bundle numbers, this array uses the third letter (“B” or “P”) as the key to determine which word (“Bundle” or “Port”) should be printed. Since the regular expression is case-insensitive, the array needs to include both upper and lowercase letters.

- The matching portion of the regular expression has been rewritten to include four sets of parentheses:

```
N(ld+)C(ld+)([BP])(ld+)
```

If a match is made, the first four read-only substitution variables (\$1, \$2, \$3, and \$4) are filled in with whatever was matched. Specifically, the \$1, \$2, and \$4 variables are set to the actual node, card, and port or bundle numbers that were matched. The \$3 variable is filled with either a “B”, “b”, “P”, or “p”, depending on what was found in the input line.



Note: The read-only substitution variables are set to new values only when a regular expression contains the appropriate number of parenthetical expressions AND a match is made. If no matches are ever made, these variables remain empty, and once a match is made, these variables retain their values until another match is made. As a general rule, therefore, assume these variables do not contain valid data unless the last regular expression had a successful match.

- The substitution portion of the regular expression has been rewritten so that it can substitute the words “Node”, “Card”, “Port”, and “Bundle” in front of the appropriate node, card, port, and bundle numbers:

```
Node $1, Card $2, $pbwords{$3} $4
```

Because the third variable (\$3) could contain either a “B” or “P” (in either upper or lowercase), the substitution uses the *%pbwords* associative array to determine what word should be used. The word “Bundle” is substituted for either a “B” or “b” and the word “Port” is substituted for either a “P” or “p” in the original line.

.....

These changes to Example 112 allow more intelligent substitutions to be performed, where the original text determines the type of substitution that should be done:

Example 113 Examples of Substitutions Using Read-Only Variables

N75	remains	Node N75
N10C12	remains	Card N10C12
N10C12P13	becomes	Node 10, Card 12, Port 13
N23C7B1	becomes	Node 23, Card 7, Bundle 1

Using Subroutines in Perl

Like most languages, Perl supports the use of subroutines, which are useful for organizing your scripts into discrete parts, each part performing a specific function. Organizing your scripts into subroutines can make it easier to test and debug them, without having to worry about whether one part of your script is having unintended side-effects on another section.

Perl allows you to define subroutines anywhere in your program, unlike other languages that require them to be defined before they are first used. One common convention among Perl programmers is to put subroutines at the end of the program, where they are easily locatable without getting in the way of the main part of the program. However, this choice is an arbitrary one and Perl allows you to follow your personal preference in the placing of subroutines.

Defining Subroutines

A subroutine is defined using the *sub* keyword and following format:

Example 114 Defining a Subroutine

```
sub subroutine-name {  
    insert one or more Perl statements here  
} # end of subroutine definition
```

The rest of your Perl script then accesses the subroutine by attaching an ampersand (&) to the front of its name:

```
&subroutine-name;      # call this subroutine
```

By default a subroutine can access any variables in your program; the only variables it cannot access are those defined as “local” within other subroutines (see *Defining Local Variables* on page 174). Many subroutines can therefore perform their needed functions without any arguments being passed to or from them.

For example, the script in Example 115 uses a subroutine named “*convert_to_upper*” to translate whatever is in the default operator (*\$_*) to uppercase. The script reads one line, calls the subroutine to convert it, and then prints the converted line.

Example 115 Using a Typical Subroutine

```
#!/usr/thirdParty/perl/bin/perl -w  
  
while (<>) {                # while reading files/STDIN
```

```
        &convert_to_upper;          # convert line to uppercase
    print;                          # print the uppercase line
}                                    # end while

#####
### SUBROUTINE DEFINITIONS
#####

sub convert_to_upper {
    tr/a-z/A-Z/;                    # translate whatever is in $_
                                    # to uppercase
} # end of subroutine definition
```

While it is often convenient to have subroutines directly affect the data in the main program, this can lead to subroutines having unexpected side-effects on the rest of the program. To avoid this, you can instead pass data to and from subroutines, as described in the following sections.

Using a Subroutine's Return Value

All subroutines pass back a *return value* to the main program or subroutine that called it. The return value is whatever is returned by the last function or expression executed by the subroutine. This value can be accessed by assigning it to a variable when the subroutine is called:

```
$value = &subroutine-name; # put return value into $value
```

For example, Example 116 is the same script shown in Example 115 except that now the variable *\$return* is assigned the return value of the *convert_to_upper* subroutine. Since the last function or expression evaluated in this subroutine is the *translate (tr)* function, the return value is whatever *tr* returns (the number of characters translated by the function):

Example 116 Using a Typical Subroutine

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

while (<>) {
    $return = &convert_to_upper; # while reading files/STDIN
    print; # convert line to uppercase
    print; # print the uppercase line
    print "$return characters were converted.\n\n"; # show # of chars translated
} # end while

#####
### SUBROUTINE DEFINITIONS
#####

sub convert_to_upper {
```

```

        tr/a-z/A-Z/;                # translate whatever is in $_
                                   # to uppercase
    } # end of subroutine definition

```

The return value can be anything, depending on whether the last function or expression evaluated was a number, a string, a simple array, or an associative array. This final expression can be as simple as a variable assignment or as complex as any of Perl's functions.

For example, the script in Example 117 reads a line of input, calls a subroutine to divide the line into individual words, and returns those words in an array. This array is then sorted and printed as a list.

Example 117 A Subroutine that Returns an Array as a Value

```

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

while (<>) {                        # while reading files/STDIN
    @return = &tokenize;           # put line in tokens (words)
    # Combine the tokens in @return into a sorted comma-separated
    # string and print them
    print("The words found in the line are:\n");
    $token_list = join(', ', sort(@return));
    print("\t$token_list\n\n");
}                                    # end while

#####
### SUBROUTINE DEFINITIONS
#####

sub tokenize {
    s/[^\w\s]//g;                  # eliminate all non-word, non

```

```

        split(/\s+/);           # whitespace characters
                                # split line into words and
                                # return as an array
    } # end of subroutine definition

```

Example 117 works because the *split* function automatically returns its tokens in a simple array, and since the *split* function was the last expression evaluated in the subroutine, what it returns becomes the subroutine's return value as well.

This script could be simplified, though, by removing the references to the *\$return* variable, and having the *join* function operate directly on the output of the *tokenize* subroutine. See Example 118:

Example 118 Modified Subroutine that Returns an Array as a Value

```

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

while (<>) {                    # while reading files/STDIN
# Tokenize the line and combine the returned tokens into a
# comma-separated string and print them
    print("The words found in the line are:\n");
    $token_list = join(', ', sort(&tokenize));
    print("\t$token_list\n\n");
}                                # end while

#####
### SUBROUTINE DEFINITIONS
#####

sub tokenize {
    s/[^\w\s]//g;                # eliminate all non-word, non
                                # whitespace characters
}

```

```

    split(/\s+/);          # split line into words and
                          # return as an array
} # end of subroutine definition

```

You might also think that you could simplify the *tokenize* subroutine into a single line, as shown in Example 119:

Example 119 Poorly Modified Subroutine that Returns an Array as a Value

```

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

while (<>) {                # while reading files/STDIN
# Tokenize the line and combine the returned tokens into a
# comma-separated string and print them
    print("The words found in the line are:\n");
    $token_list = join(' ', sort(&tokenize));
    print("\t$token_list\n\n");
} # end while

#####
### SUBROUTINE DEFINITIONS
#####

sub tokenize {
    split(/\s+/, ($_ =~ s/[^\w\s]//g));
                                # split line into words and
                                # return as an array
} # end of subroutine definition

```

However, if you make this modification the script no longer prints out a list of words; at most it prints out only one number. This is because the substitution function, like

the *tr* function, returns how many characters were changed. The *split* function ends up operating on this number, doing operations such as “*split(/s+/, '1')*” instead of “*split(/s+/, 'This is an input line')*”.

This example points out the need of making sure you know what values are being returned by both functions and subroutines. For this purpose, it is highly recommended that you write your scripts as simply and straight-forwardly as possible, particularly near the end of subroutines, so that there is no doubt about what value is being returned.

Defining Local Variables

By default, subroutines access “*global*” variables that are available to the main part of the program and any other subroutines. While this might be what you want in certain cases (such as in Example 118 (page 171), where the *tokenize* subroutine accesses the global `$_` variable), it can also lead to situations where a subroutine inadvertently changes the value of a variable used elsewhere in a program.

You can avoid such side-effects by using *local* variables within your subroutines. Local variables are declared using the *local* operator from within a subroutine’s declaration. See Example 120:

Example 120 Defining Local Variables Within a Subroutine

```
sub subroutine-name {  
    local($local-var1, $local-var2, @local-array);  
    insert one or more Perl statements here  
} # end of subroutine definition
```

Example 120 shows three local variables being created: two scalar variables and one array. There is no limit to the type of local variables you can create or to the number of *local* statements you have within a subroutine.



Note: The *local* operator can be used anywhere within a subroutine, but it is good programming practice to put all such statements at the beginning of a subroutine’s declaration, as shown in Example 120, so it is immediately apparent what variables are local.

Local variables are created when a subroutine is called and exist only as long as the subroutine runs. Changing them does not affect any other variables in your script, even if those other variables have the same name as your local variables.

For example, if a local variable has the same name as a global variable, the global variable's value is saved before the subroutine is called. While the subroutine runs, only the local variable is used, and when the subroutine ends, the local variable disappears and the global variable is restored with its previous value.

If a subroutine calls other subroutines, all of the variables in the top-level subroutine become “global” to the lower-level subroutines. Those lower-level subroutines must declare their variables local as well to prevent changing the top-level subroutine's variables.

Example 121 is a simple demonstration of this: the main part of the script calls the subroutine *sub1*, which in turn calls the subroutines *sub2* and *sub3*. Because *sub2* has declared a local version of the default operator (*\$_*), its *print* statement outputs the same line over and over again, instead of the input line that is output by the *print* statements in the main routine and the *sub1* and *sub3* subroutines.

Example 121 Using Local Variables in More than One Subroutine

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

while (<>) {                                # while reading files/STDIN
    print "The main routine thinks the input line is:\n";
    print "\t$_\n";                          # original input line
    &sub1;
}                                             # end while
```

```
#####
### SUBROUTINE DEFINITIONS
#####

sub sub1 {
    print "The sub1 subroutine thinks the input line is:\n";
    print "\t$_\n";           # original input line
    &sub2;
    &sub3;
} # end of subroutine definition

sub sub2 {                    # sub2 defines a local $_
    local($_) = "sub2's input string\n";
    print "The sub2 subroutine thinks the input line is:\n";
    print "\t$_\n";           # always the sub2 version
} # end of subroutine definition

sub sub3 {
    print "The sub3 subroutine thinks the input line is:\n";
    print "\t$_\n";           # original input line
} # end of subroutine definition
```

Two things of note should be noted about the script in Example 121. First, as shown in the *sub2* declaration, you can assign a value to a local variable when you declare it, using Perl's standard assignment rules concerning scalar variables and arrays.

For example, all of the following are valid ways of assigning values to local variables:

Example 122 Assigning Values to Local Variables

```
local($str1,$str2);          # both $str1 and $str2 are
                             # undefined (equal to null)
local($str1,$str2)=$_,$_;   # puts copy of default operator
                             # into both $str1 and $str2
local($str1,$str2)=@array;  # puts first two elements of
                             # array into $str1 and $str2
local(@array)=@array;       # copies global @array into
                             # local version
local(@array)=split();      # tokenize global $_ and puts
                             # the words into local array
```

Local variables can be given the same names as global variables without Perl getting confused, but extensive use of this feature can make debugging and maintaining your scripts a difficult matter. It is not immediately apparent, for example, when looking at the script in Example 121, when `$_` refers to a global variable and when it refers to a local variable.

Consider, therefore, adopting a naming convention for local variables that makes it easy to tell when a subroutine is using a local variable. For example, you could give all of your local variables a prefix such as “`sub_`” (`$sub_name`, `$sub_length`) or a suffix such as “`_loc`” (`$name_loc`, `$length_loc`).

Using a consistent naming convention for local variables not only simplifies the process of debugging your scripts but can also aid you when you revise a script later on.

Passing Arguments

Unlike other languages, Perl does not require you to define a subroutine's arguments when the subroutine is defined. You can specify any number of arguments when calling a subroutine by putting them within parentheses after the subroutine's name:

```
&subroutine($arg1, $arg2, . . . $argn);  
or  
&subroutine(@array1, @array2, . . . $arrayn);
```

You can pass any type of variable to a subroutine: scalar, array, associative array, or a mixture of these. Neither the number of variables nor their types need to be declared in advance because Perl automatically puts a subroutine's arguments into a local array named “@_”.

When creating this array, Perl inserts the passed arguments into the array in the order they were passed to the subroutine. Scalar variables are assigned one array slot each, while arrays are copied in element by element. A subroutine can access the @_ array as with any other array; it can act upon the array as a whole with operators such as *split*, or it can act upon individual elements of the array (*\$_[0]*, *\$_[1]*, and so forth).

Thus, if you pass a subroutine two scalar variables and an array with 12 elements, that subroutine's @_ array contains 14 separate elements. It is up to the subroutine to determine which elements came from which source and which ones it wants to use.

You can determine the number of elements either by setting a scalar variable equal to the array (such as “*\$length=@_;*”) or by looping through the elements of the array until you reach the first one that is undefined. Example 123 demonstrates both methods:

Example 123 Subroutine Using an Array as Passed Argument

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

while (<>) {
    &to_upper(split());
}

sub to_upper {
    local($i,$length) = (0,0); # initialize local variables
    $length = @_; # get number of array elements
    print("$length words were passed.\n\n");
    while ($_[ $i]) { # as long as element is defined
        $_[ $i] =~ tr/a-z/A-z/; # convert to uppercase
        print "Word # $i is: $_[ $i]\n"; # and print it
        $i = $i + 1 ; # increment $i
    } # end while
} # end to-upper
```

The main routine in Example 123 simply reads a line from the input files (or STDIN), uses the *split* function to convert it into an array of words, and passes that array to the *to_upper* subroutine. The subroutine gets the length of the array and prints it, and then loops through the array, converting each individual word to uppercase and printing it.



Note: Do not confuse the `@_` array and its elements (`$_[0]`, `$_[1]`, and so forth) with the default operator `$_`. In particular, `$_[0]` is not the same as `$_`.

.....

Associative arrays can be passed as arguments as well, but be aware that because Perl automatically optimizes its storage of associative arrays, the order in which they are passed to a subroutine might not be the order in which they were created. For example, Example 124 shows a script that creates one regular array and one associative array with the same elements; both arrays are then passed to a subroutine that prints out all of their elements:

Example 124 Passing an Associative Array to a Subroutine

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

%array = ("john","boy","jill","girl");
@array = ("john","boy","jill","girl");
&sub1(%array,@array); # pass both arrays to sub1

sub sub1 {
    $i = 0;
    while ($_[ $i ]) {
        print "Element $i is: $_[ $i ]\n";
        $i = $i + 1;
    } # end while
} # end subroutine definition
```

The two arrays shown in Example 124 are created in the exact same order, but the script's output (see Example 125) shows that the associative array's first element (identified as element 0 of the `@_` array) has been changed to "jill" while the regular array's first element (identified as element 4 of the `@_` array) is still "john":

Example 125 Output Comparing an Array to an Associative Array

```
Element 0 is: jill
Element 1 is: girl
Element 2 is: john
Element 3 is: boy
Element 4 is: john
Element 5 is: boy
Element 6 is: jill
Element 7 is: girl
```

Since Perl does not guarantee any sort of ordering of its associative arrays (except that keys are always paired with their associated values), your subroutines should not count on any particular ordering of the arrays' elements.

Another point that should be made is when associative arrays are passed to a subroutine, they lose their associativity and become regular arrays. If your subroutines need to access an associative array through its key values, the array should be accessed as a global variable, not as arguments passed to the subroutine.

It is possible, though, to recreate an associative array within a subroutine by loading a local associative array with the elements in the `@_` array, using a technique similar to that shown in Example 126. However, this technique has limited usefulness and is not recommended for large associative arrays.

Example 126 Recreating an Associative Array Within a Subroutine

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

%array = ("john","boy","jill","girl");
```

```
&recreate_array(%array);

sub recreate_array {
    local($i) = 0;
    local(%_);          # local associative array

    while ($_[ $i ]) {
        $_{$_[ $i ]} = $_[ $i + 1 ]; # recreate associative array
        $i = $i + 2;                # point to next key value
    } # end while
    print "John is a ", $_{"john"};
    print " and Jill is a ", $_{"jill"}, ".\n";
} # end subroutine definition
```

Index

Symbols

! operator 82
!= operator 79
!~ operator 81, 141
#! variable 48, 130, 131
 \$# variable 34, 110
 \$\$ variable 49, 99
 \$% variable 68
 \$& variable 49, 142
 \$(variable 49
 \$) variable 49
 \$/ variable 48
 \$< variable 49
 \$= variable 67
 \$> variable 49
 \$] variable 48
 \$_ variable 48
 using while reading files 109
 \$| variable 48, 71
 \$' variable 49, 142
 \$'' variable 50, 142
 \$0 variable 49, 117
 \$1 variable 50
 \$ARGV variable 49, 114
 \$OUTPUT_AUTOFLUSH variable 71
 % operator 74
 && operator 82
 ****** operator 74
 < operator 80

<= operator 80
 <=> operator 80
 <> operator 112 to 116
 getting a list of files 125
 getting the current file name 114
 == operator 79
 =~ operator 81, 140
 > operator 79
 >= operator 80
 @* operator 66
 @_ variable 178
 @ARGV array 49, 112, 117 to 118
 || operator 82

A

abs command 97
 anchoring patterns for regular expressions 149
 arithmetic functions 97
 arithmetic operators 73
 arrays 31 to 37
 \$# variable 34
 @ARGV 49, 112, 117 to 118
 adding elements using a list 32
 avoiding confusion 46
 converting to an associative array 44
 operators 36
 reading an entire file 110
 rules 31

- using \$# to get the last element number 110
- using with subroutines 178

assignment operators 73

associative arrays 38 to 45

- adding or modifying elements 39
- avoiding confusion 46
- converting to a normal array 44
- deleting an element 41
- extracting the keys 42
- extracting the values 43
- using with subroutines 178

atan command 97

B

binary bitwise operators 74

boolean operators 82

buffering of output commands 69

C

chdir command 124

chomp command 101

chop command 36, 101

chr command 101

close command 122

closedir command 128

cmp operator 81

command line arguments 112

command line options 10 to 16

commands

- abs 97
- atan 97
- chdir 124
- chomp 101
- chop 36, 101
- chr 101
- close 122
- closedir 128
- comments (#) 18
- cos 97
- die 120
- each 44
- exp 97
- for 90 to 92
- foreach 93 to 95
- format 59
- gmtime 100
- hex 97
- if 84 to 85
- index 102
- int 98
- lc 102
- lcfirst 102
- length 102
- localtime 100
- log 98
- mkdir 130
- oct 98
- open 119 to 122
- opendir 126

ord 102
pop 36
print 52 to 54
printf 55 to 58
push 36
rand 99
readdir 126
renaming 129
reverse 36, 95
rindex 103
rmdir 130
select 55, 59, 111
shift 37
sin 99
sort 37, 95, 127
sqrt 99
srand 99
sub 166 to 182
substr 103
time 100
tr 93
uc 103
ucfirst 103
undef 104
unless 86 to 87
unlink 128
unshift 37
warn 120
while 88 to 89
write 59 to 68

comments 18
cos command 97

D

delete operator 41
diamond (<=>) operator 112 to 116
die command 120

E

each operator 44
eq operator 80
escaped characters 28
exp command 97
exponentiation operator 74

F

filehandles 106, 119 to 123
files 105 to 135
 accessing the command line 117 to 118
 accessing the command line arguments 112
 changing the current working directory 124
 deleting a file 128
 directory operations 124 to 131
 file test operators 132 to 135
 finding the current file name when using the <>
 operator 114
 listing the files in a directory 125
 making a new directory 130

opening a file for appending 121
opening a file for writing 120
reading a directory 126
reading STDIN one line at a time 108
reading STDIN using the diamond (<=>) operator 112 to 116
removing a directory 130
renaming 129
sorting a file list 127
using filehandles 106, 119 to 123
using STDIN to read an entire file 110
using STDOUT and STDERR 110
using STDOUT, STDIN, and STDERR 107 to 111
for command 90 to 92
foreach command 93 to 95
format command 59
functions 96 to 104
 arithmetic 97
 string 101
 timekeeping 100

G

ge operator 80
gmtime command 100
gt operator 80

H

hex command 97

I

if command 84 to 85
index command 102
int command 98

K

keys operator 42

L

lc command 102
lcfirst command 102
le operator 81
length command 102
localtime command 100
log command 98
logical operators 82
lt operator 80

M

mkdir command 130
modulo operator 74

N

ne operator 80

O**oct command** 98**open command** 119 to 122

opening a file for appending 121

opening a file for writing 120

using the die command 120

opendir command 126**operators**

! 82

!= 79

!~ 81, 141

% 74

&& 82

** 74

< 80

<=> 80

<> 112 to 116

== 79

=~ 81, 140

> 79

>= 80

@* 66

|| 82

arithmetic 73

assignment 73

binary bitwise 74

cmp 81

delete 41

each 44

eq 80

file test 132 to 135

ge 80

gt 80

keys 42

le 81

logical 82

lt 80

ne 80

relational 79

values 43

ord command 102**output commands** 51 to 68**P****Perl**

accessing command line arguments 112

accessing the command line 117 to 118

additional documentation 7

anchoring patterns 149

arithmetic functions 97

array operators 36

arrays 31 to 37

assignment operators 73

associative arrays 38 to 45

basic features 5

basic rules 17

buffering of print commands 69

built-in functions 96 to 104

changing the current working directory 124

command line options 10 to 16

- comments 18
- control structures 83 to 95
- debugger 11
- definitions of true and false 19
- deleting a file 128
- directory operations 124 to 131
- displaying the version number 15
- file access 105 to 135
- file test operators 132 to 135
- floating-point format of numeric data 28
- generating reports 59 to 68
- getting the program or script's name 117
- introduction 4
- listing the files in a directory 125
- making a new directory 130
- naming conventions 20
- output commands 51 to 68
- producing warnings 15
- reading a directory 126
- reading an entire file using STDIN 110
- regular expressions 136 to 165
- removing a directory 130
- renaming files 129
- report formats 61
- scalar variables 25 to 30
- simple matching and replacing 144 to 154
- special characters 28
- string functions 101
- subroutines 166 to 182
- timekeeping functions 100

- topics not covered 3
- using filehandles 106, 119 to 123
- using STDOUT, STDIN, and STDERR 107 to 111
- using subpatterns 162
- using the interpreter 9
- using wildcards 155 to 165
- variables 23 to 50

- pop command** 36
- print command** 52 to 54
 - changing the buffering behavior 69
 - defaults to STDOUT 111
 - sending to STDOUT and STDERR 110
- printf command** 55 to 58
 - changing the buffering behavior 69
 - format types 55
- procedures** 166 to 182
- Process ID**
 - getting from the \$\$ variable 99
- push command** 36

R

- rand command** 99
- readdir command** 126
- regular expressions** 136 to 165
 - !~ operator 141
 - \$& variable 142
 - \$' variable 142
 - \$' 142
 - ==~ operator 140
 - anchoring patterns 149

.....

- defining 138 to 139
- replacing subpatterns 162
- rules 140 to 143
- simple matching and replacing 144 to 154
- special characters 150
- special variables 142
- specifying ASCII values 152
- using wildcards 155 to 165

relational operators 79

rename command 129

reports 59 to 68

- \$% variable 68
- \$= variable 67
- @* operator 66

- adding page headers 67
- formats 61
- outputting multiple lines 66
- writing multiple lines 63

return values, subroutines 169 to 173

reverse command 36, 95

rindex command 103

rmdir command 130

S

- scalar variables** 25 to 30
 - rules 25
 - strings and numeric data 26
 - using double and single quotes 26
- select command** 55, 59, 111
- shift command** 37

- sin command** 99
- sort command** 37, 95, 127
- special characters** 28
- sqrt command** 99
- srand command** 99
- STDERR** 107 to 111
 - sending a fatal error using the die command 120
 - sending a warning message using the warn command 120
- STDIN** 107 to 111
 - reading an entire file 110
 - reading using the diamond (<=>) operator 112 to 116
 - using to read one line 108
- STDOUT** 107 to 111
 - default for the print command 111
- string functions** 101
- sub command** 166 to 182
 - defining 167 to 168
 - passing arguments 178 to 182
 - using local variables 174 to 177
 - using the return value 169 to 173
- subroutines** 166 to 182
 - defining 167 to 168
 - passing arguments 178 to 182
 - using local variables 174 to 177
 - using the return value 169 to 173
- substr command** 103

T

time command 100
timekeeping functions 100
tr command 93

U

uc command 103
ucfirst command 103
undef command 104
unless command 86 to 87
unlink command 128
unshift command 37

V

values operator 43
variables 23 to 50
 \$! 48, 130, 131
 \$# 34, 110
 \$\$ 49, 99
 \$% 68
 \$& 49, 142
 \$(49
 \$) 49
 \$/ 48
 \$< 49
 \$= 67
 \$> 49
 \$] 48

 \$_ 48, 109
 \$| 71
 \$' 49, 142
 \$' 50, 142
 \$0 49, 117
 \$1 48, 50
 \$ARGV 49, 114
 @_ 178
 @ARGV 49, 112, 117 to 118
 avoiding confusion 46
 defining local variables 174 to 177
 filehandles 106
 OUTPUT_AUTOFLUSH 71
 scalars 25 to 30
 special characters 28
 string and numeric data 26
 use in subroutines 174 to 177

W

warn command 120
while command 88 to 89
wildcards 155 to 165
 replacing subpatterns 162
 rules 157
 using for matching 159
 using for substitutions 161
write command 59 to 68
 changing the buffering behavior 69